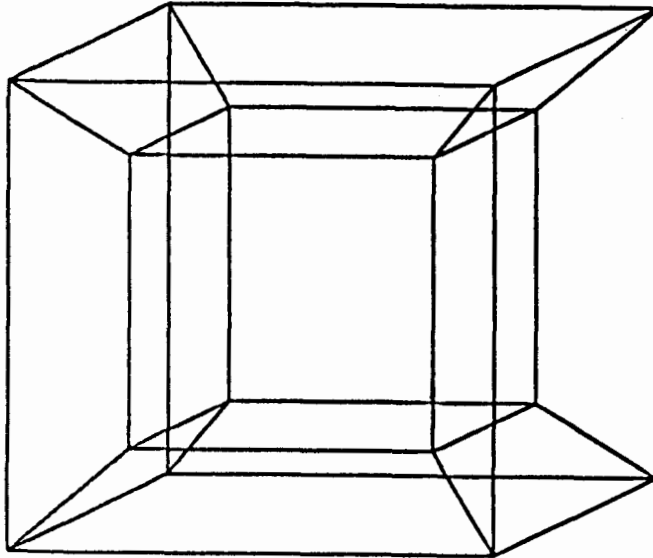


The Design of a Parallel Knowledge-based Optical Character-recognition System

J. Henseler/J.C. Scholtes/C.R.J. Verhoest



**THE DESIGN OF A
PARALLEL KNOWLEDGE-BASED
OPTICAL CHARACTER-RECOGNITION SYSTEM**

MASTER OF SCIENCE THESES

DEPARTMENT OF MATHEMATICS AND INFORMATICS

DELFT UNIVERSITY OF TECHNOLOGY

(12 November 1987)

J. HENSELER
J.C. SCHOLTES
C.R.J. VERHOEST

CONTENTS

PREFACE

OUTLINE

INTRODUCTION

PART ONE: A LITERATURE STUDY

1	<i>THE NCUBE AND OTHER PARALLEL COMPUTERS</i>	1
1.1	Background	1
1.1.1	The Hypercube Architecture	1
1.1.2	Philosophy behind the NCube	3
1.2	A further investigation of the NCube 4/+	3
1.2.1	Hardware	3
1.2.2	Software	5
1.2.3	Performance	6
1.3	Differences with Other Parallel Computers	7
1.3.1	The Connection Machine	7
1.3.2	Transputers	9
1.3.3	The DADO Production Machine	12
1.3.4	Array and Pipelined Machines	13
1.4	Conclusions on the NCube 4/+	15
1.5	NCube Bugs	15
1.6	Miscellaneous on the NCube	16
1.6.1	The NCube Usergroup	16
1.6.2	Vision Algorithms on the Hypercube	16
1.6.3	Intelligent- Machine Vision	17
1.6.4	Hypercube Algorithms for Robot Dynamics	17
1.6.5	Embedding Trees in a Hypercube	17
2	<i>PARALLEL PROCESSING</i>	19
2.1	Background	19
2.1.1	Historical Background	19
2.1.2	The Main Methods	19

2.2 Pipeline Processors	20
2.3 Array Processors	22
2.4 Multiple Processor Systems	23
2.5 Data-Flow Machines	24
2.6 Parallelising a Problem	24
2.6.1 Vector Processing	24
2.6.2 Array Processing	25
2.6.3 Multiple Instruction Multiple Data Processing	27
2.6.4 Data Flow Processing	27
2.7 Conclusions	28
3 PATTERN RECOGNITION	29
3.1 Background	29
3.2 The Structure of a Pattern Recognition System	30
3.3 Different Approaches in Pattern Recognition	31
3.3.1 Statistical Pattern Recognition	31
3.3.2 Syntactical Pattern Recognition	37
3.3.3 Hierarchical Pattern Recognition	37
3.3.4 Knowledge-Based Pattern recognition	37
3.4 Conclusions	37
4 OPTICAL CHARACTER RECOGNITION	39
4.1 Background	39
4.1.1 Historical Background	39
4.1.2 What is Character Recognition?	39
4.2 Different Methodologies	40
4.3 The Optical Character Recognition System	41
4.3.1 Preprocessing	42
4.3.2 Classification	52
4.3.3 Postprocessing	58
4.4 Major Research Projects and the State of the Art of O.C.R.	59
4.4.1 Bell Laboratory's Multifont, Multisize O.C.R.	59
4.4.2 Recognition of Isolated Handwritten Numerals	61
4.4.3 Benchmarks	63
4.5 Conclusions	64
4.6 Research Outline	

PART TWO: DESIGN, REALIZATION AND IMPLEMENTATION

5	THE PROGRAMMING ENVIRONMENT	67
5.1	Background	67
5.2	Scheme	67
5.2.1	The Philosophy and History of Scheme	68
5.2.2	The Scheme Language	68
5.2.3	Flisp is a Scheme Implementation	72
5.3	Distributed Scheme	73
5.3.1	Communication	73
5.3.2	Interaction	77
5.4	Examples of Parallel Programs	79
6	PREPROCESSING	83
6.1	Background	83
6.2	Split: A Decomposition Algorithm	84
6.2.1	Distributed Processing of the Image	85
6.2.2	The Split Algorithm	87
6.2.3	The Split-Function Description	89
6.3	Hough: A Line-SegmentDetection Algorithm	89
6.3.1	Aspects and consequences of the Implementation	89
6.3.2	The Implementation of the Hough Transform	94
6.3.3	The Hough-FunctionInterface Description	98
6.4	Graph : Connectivity-based algorithms	99
6.4.1	Implementation of a Graph Environment	99
6.4.2	Graph Environment Interface Description	104
7	THE CLASSIFIER	107
7.1	Background	107
7.2	A Classifier	107
7.2.1	A Theoretical Approach to a Classifier	107
7.2.2	Data Needed for a Classifier	110
7.2.3	The Conceptual Model of a Classifier for our OCR System	114
7.3	Topological Knowledge	115
7.3.1	Introduction	115
7.3.2	Functional Specifications	115
7.3.3	Using Frames for Constructing a Decision Graph	116
7.3.4	Implementation of the Funtions QUERY and TRACE	121

7.4 The Contextual Knowledge	122
7.4.1 Background	122
7.4.2 The Advantages of Trigrams	122
7.4.3 Analysis of Trigram Frequencies	123
7.4.4 Search Optimization Obtained by Trigram Analysis	129
7.4.5 Sort and Permute Trigrams	130
7.4.6 Conclusions	133
7.5 The Statistical Knowledge	133
7.6 The Controller	138
7.6.1 Functional description	138
7.6.2 The Implementation	140
 8 POSTPROCESSING	 143
8.1 Background	143
8.2 Advantages of Spelling Correction	143
8.3 The Method used for Spelling Correction	143
8.3.1 Building the Hash- Table	143
8.3.2 Controll ingthe NCube Channels	145
8.3.3 The Confusion Groups	146
8.3.4 Building the ASCII- File.....	148
8.4 Implementation of a Spelling Correcting on the Host	148
8.5 Conclusions	149
 PART THREE: RESULTS, CONCLUSIONS AND RECOMMENDATIONS	
 9 RESULTS	 151
9.1 Background	151
9.2 The Preprocessor Results	151
9.2.1 Results of the Distribute and the Split Algorithm	152
9.2.2 Results of the Hough Transform	154
9.3 Results on Classification	157
9.3.1 Results on Trigram Analysis	157
9.3.2 Recognition Performance	159
9.4 Results on Postprocessing	163
9.4.1 Testing the Postprocessor	163
9.4.2 Time Performance of the Postprocessing	167
9.4.3 Observations on Postprocessing	167

<i>10 CONCLUSIONS AND RECOMMENDATIONS</i>	169
10.1 Prolegomena	169
10.2 Conclusions on Preprocessing	169
10.2.1 The Distribute Algorithm	169
10.2.2 The Split Algorithm	170
10.2.3 The Hough Transform	170
10.3 Conclusions Classification	170
10.3.1 Trigram Analysis	170
10.3.2 Statistical Knowledge	171
10.3.3 Skeleton Representation	171
10.3.4 Frames	172
10.3.5 The Classifier	172
10.4 Conclusions on Postprocessing	173
10.5 Overall Conclusions	173
10.5.1 Parallel Processing	173
10.5.2 Software Engineering	174
10.5.3 Pattern Recognition	174
10.5.4 The Ncube	174

PART FOUR: REFERENCES AND APPENDICES

REFERENCES

- A STATISTICAL KNOWLEDGE FOR TWO FONTS*
- B TRIGRAM TABLES*
- C PREFIXES AND SUFFIXES*
- D FRAMES*
- E SOURCE CODES*
- F FLISP DOCUMENTATION*

PREFACE

This treatise on Knowledge-Based Optical Character Recognition contains three Master of Science Theses. The theses project is carried out at the Department of Mathematics and Informatics at Delft University of Technology in the Netherlands. The work reflects our factual knowledge, abilities and ideas in a broad field of Computer Science, best circumscribed as interdisciplinary work in the fields of Artificial Intelligence, Software Engineering, Parallel Computing and Pattern Recognition. The research is performed in close operation as is the writing of this treatise. This implies that it is impossible to distinguish the exact contribution of each author. Obviously, Part 1 is not an MSc Thesis in itself, but is essential for understanding the other three Parts. Moreover, it leads the readers to the frontiers of current research in the highly interesting fields described above.

We emphasize that Artificial Intelligence may be regarded as a new phase in Software Engineering. Clearly, it encompasses many programming techniques and methods being a welcome addition to traditional Software-Engineering techniques. Some examples of these methods are Knowledge Engineering, the separation between Logic and Control as proposed by Kowalski [1979]; the knowledge representation in Horn clauses [Kowalski, 1979], together resulting into the realization of an important Artificial-Intelligence language called Prolog, i.e., Programming in Logic, which originated as a theorem prover written in Fortran by Colmerauer and his group [1972] in Marseille. Artificial Intelligence has also justified the existence of the list processing language called Lisp [Winston and Horn, 1984]. We could supply many more examples that show the relevance of Artificial Intelligence in computer science today. As it seems, the role of Artificial Intelligence in the near future is rather important especially with a view on the Japanese Fifth-Generation Project and the American Strategic Defense Initiative (SDI).

As a working definition we pose: Artificial-intelligence programs try to solve unstructured problems, whereas up to a few years ago computers were only used to resolve clearly-structured problems. Straight-forward computation of unstructured problems is almost never possible and whenever it is, it will probably require an unacceptable amount of CPU time. Dealing with unstructured problems in an artificially-intelligent manner, AI programs may reduce computation times to acceptable limits. The idea is to employ foreknowledge about a problem when executing the computation required in such a way that no difficult conversion of the available knowledge occurs; thus ensuring that all available knowledge can be used in principle.

The true aim of our three combined theses is to underline the relevance of Artificial Intelligence as a collection of Software Engineering techniques in Pattern Recognition and Parallel Computing.

Hence we have designed an Optical Character Recogniser combining statistical methods available from traditional Pattern Recognition with Artificial-Intelligence techniques that interpret topological knowledge about character patterns. To direct the reasoning process also contextual knowledge in the form of trigram frequencies is employed. Such a recognition system is only relevant if it recognises the characters within acceptable time limits. According to our opinion a big leap forwards is made when such a system can be adequately parallelised: in our research we have concentrated the attention to the realization on a hypercube structured parallel computer. This parallelisation enables the program to run on as much as 1024 processors making the Recogniser potentially capable to outperform all existing Optical Character Recognisers.

The research could not have been realized without the *donation* of a NCube parallel computer by Shell Nederland B.V. which we gratefully acknowledge. We would like to thank all our collaborators on the HYDRA project being our first acquaintance with Parallel Knowledge-Based Systems. The combined theses appear in this form due to the valuable suggestions of the examination committee consisting of Prof. Dr. H.G. Sol, Prof. Dr. Ir. E. Backer, Prof. Dr. H.J. van den Herik, Dr. Ir. E.J.H. Kerckhoffs and Dr. H. Koppelaar. Moreover, we would like to thank Westmount Technology B.V. for the use of their laser writer and other computer facilities which enabled us to print the theses. Finally, we are greatly indebted to Prof. Dr. H.J. van den Herik who has accompanied us in our research work for the last three years and whom we always could rely on.

In the end we acknowledge and thank all family, friends and teachers that have taken part in our education in Delft.

OUTLINE

Three theses are combined into a treatise consisting of four parts: 1. A Literature Study, 2. The Design and Implementation, 3. Results, Conclusions and Recommendations, and 4. References and Appendices A through F.

The first Part is a self-contained literature study on Parallel Processing and Pattern Recognition. It consists of four Chapters. The first Chapter introduces the NCube four/+ parallel computer on which the knowledge-based optical character-recognition system has been implemented; it also discusses computers based on other structures than a hypercube structure. The second Chapter is a survey on Parallel Processing. In this section the most common methods to parallelise a problem on existing architectures are described. The third Chapter provides some background to Pattern Recognition. An important issue will be the difference between statistical and syntactical Pattern Recognition. In Chapter four the pattern-recognition process is examined more closely and is partitioned in preprocessing, classification and post-processing. Some currently available Optical Character Recognition systems are discussed and as an extension of our literature research a model of the recogniser is presented.

The second Part specifies in extenso the design, realization and implementation of the model presented in Chapter four of Part 1. Part 2 consists of four Chapters being the Chapter 5 through 8. Chapter five describes the programming environment consisting of a Scheme interpreter and a concurrent user interface that has been implemented. In the Chapters 6, 7 and 8 the design and implementations of the Preprocessor, Classifier and Postprocessor are respectively thoroughly investigated.

In the third Part the results of the implemented system, some (tentative) conclusions and recommendations are presented. Part 3 contains two Chapters; 9 and 10. In Chapter nine the Results are presented, and Chapter ten provides the reader with the conclusions. Here, as a conclusion we want explicitly remark that it has turned out to be possible to combine Artificial Intelligence with traditional Pattern Recognition and that the resulting system can be speeded up by parallelisation.

The fourth Part holds the appendices. Appendix A contains statistical knowledge about the fonts used in test results. In appendix B the trigram frequency tables of the ICCA Journal, the UNIX manual and this treatise can be found. Appendix C describes pre- and suffixes used in the spelling corrector. Appendix D describes the Frames used. Appendix E contains all C sources of implemented programs. Appendix F comprises the documentation concerning the Flisp interpreter.

INTRODUCTION

In 1984, the HYDRA has been launched when three students, A. Groen, A.G. Hofland and J.C. Stoop, who previously had designed the Delfi I expert system shell, together with Prof. Dr. H.J. van den Herik, Dr. Ir. E.J.H. Kerckhoffs and Dr. N. Kasabov decided to focus on a *parallel* rule-based system. After his return from the McGill University (Montreal, Canada), Prof. Dr. H.J. van den Herik became a member of the Information Systems Group headed by Prof. Dr. H.G. Sol; Dr. Ir. E.J.H. Kerckhoffs was and still is a member of the Applied Physics Group headed by Prof. Dr. Ir. L. Dekker; finally, Dr. N. Kasabov was a temporarily research fellow from the University of Sophia, Department of Electrical Engineering (Sophia, Bulgaria).

As a first result, a parallel rule-based system called HYDRA II [A. Groen et al., 1985] was realized by the three students. The implementation of such a parallel system was actually simulated on the Parallel Computer Simulator, programmed by P. Varkevisser [1986]. The concept for the HYDRA I system has been realized at a later time [Cohen and Houck, 1986], for it was expected that its performance would suffer due to communication bottlenecks. After the first year the HYDRA project as a whole turned out to contain fruitful testing grounds for research that could well be performed by students; their work have resulted in many internal and external papers. Recently, the HYDRA II system has been realized on the Delft Parallel Processor, a multi-processor system that has been developed in the Applied Physics Group [Dekker, 1984].

The HYDRA I concept was rejected because of the centralized control approach that was expected to cause a communication bottleneck. Therefore, a decentralized control approach was designed to ensure that such a bottleneck would not exist. This approach led to the implementation of HYDRA II, based on a shared memory computer architecture. A decentralized control was realized by parallelising the inference engine which resulted in the construction of so called rule processors. A rule processor can process one rule at a time, whereas an inference engine can perform a complete inference that may involve the processing of several rules. The rule processors are coordinated by means of an agenda mechanism that employs a priority function to assign rules to free rule processors. Test results have shown that for a specific rule base there exists an optimum number of rule processors [Hofland, 1987], if this optimum is exceeded the performance decreases because of communication overhead. It was further shown that this optimum can be increased, implying a speed up, if the number of rules was increased and also if the effort needed to process a single rule was extra burdened. Moreover, the performance will increase if, for instance predicate logic would be used instead of binary logic.

The HYDRA I and II system have taught us a difference between parallelising a centralized and a decentralized controlled system. But another difference emerged, clearly indicating the existence of an optimum number of rule processors: the amount of knowledge needed for the reasoning process may vary, because all available processors try to find some work when solving a problem. This difference is strengthened if we have a parallel computer with local memory implying that the knowledge and input data should be distributed over the local memories. We have examined this phenomenon and as a result we may put forward that there is a direct relation between the amount of knowledge and the amount of input data when parallelising a knowledge-based system on a local memory parallel computer architecture. On the one hand if the amount of input data surpasses the amount of reasoning knowledge, the knowledge-based system is best parallelised by distributing the input data over several knowledge-based systems that all have the same knowledge. On the other hand whenever the amount of knowledge surpasses the input data it is better to distribute the knowledge.

An Optical Character Recognition System deals with a large amount of input data. According to the statement above we think it is best to recognize the characters by distributing parts of the input image over the processors in the parallel computer. Our parallel computer, the NCube four/+, has 16 processors in a so

called hypercube configuration. Each processor has its own local memory that can not be accessed directly by another processor. Distribution of a text image is easy because a text consists by nature of lines, words and letters representing perfect area descriptions that can be used for splitting up the image.

The Optical Character Recogniser itself preprocesses the image data so that line segments are detected. These line segments are matched against topological knowledge of characters by the classifier. The matching can be performed in an inference process that may employ either forward or backward reasoning [Rich, 1983], applying the topological knowledge to recognise patterns in the line segments. Readers familiar with backward reasoning know that one or more hypotheses are needed. The classifier obtains these hypotheses by means of a table containing frequency counts of tri-letter sequences in english texts. Such tri-letter sequences are called trigrams and their frequencies are a relative measure of the certainty that a tri-letter sequence occurs in a text. A hypothesis for the next character to follow a two-letter sequence is the character that results in the trigram that has the highest frequency, if it is added to the two-letter sequence. If a hypothesis is rejected, the character that results in the trigram with the second highest frequency is taken as a hypothesis. In the end, some patterns may still be unresolved. These patterns are unknown characters and will be filled in by the Postprocessor that uses a dictionary for spelling correction.

The goal to be aimed at is to underline the relevance of Artificial Intelligence as a collection of Software Engineering techniques in Pattern Recognition and Parallel Computing. This goal is reached through the design of a parallel knowledge-based optical character-recognition system that combines Artificial Intelligence, Software Engineering, Pattern Recognition and Parallel Computing.

The problem at issue is to develop a methodology for the design of Parallel Systems in relation to knowledge manipulation. Subsequently, the design, realisation and implementation of the system is a problem that can be defined in three parts. Firstly, we have to combine Artificial Intelligence with traditional Pattern Recognition in order to classify an amount of input data into different classes and to provide a description of the objects. Secondly, to increase its performance, the system should be adequately parallelised. Thirdly, we have to show that the knowledge that is employed by the classifier is designed and implemented in the same manner as traditional software. Finally, we argue that this knowledge is a valuable addition to the programs which are the fundamentals of the system, thus proving the relevance of Artificial Intelligence in Software Engineering.

This concludes our introduction to the design of a Parallel Knowledge-Based Optical Character Recognition System.

PART 1

A LITERATURE STUDY

- **THE NCUBE and OTHER PARALLEL COMPUTERS**
- **PARALLEL PROCESSING**
- **PATTERN RECOGNITION**
- **OPTICAL CHARACTER RECOGNITION**

CHAPTER 1

THE NCUBE AND OTHER PARALLEL COMPUTERS

Machine Descriptions and Architectural Characteristics

ABSTRACT

In this chapter, we describe the NCube family's architecture and some system features. The NCube/4+ architecture is emphasized; all our research programs are running on such a machine. Moreover, the advantages of the hypercube architecture above other parallel computers are discussed.

1.1. BACKGROUND

The NCube/4+ system is a supercomputer with the size of a microcomputer. It uses up to 16 identical high-speed processors connected in a hypercube network to deal with a single problem. An n -dimensional hypercube computer is characterised by the $N = 2^n$ processors. The NCube/4+ (i.e., $n = 4$ and thus $N = 16$, the '+' is denoting an extended amount of memory, namely 512 kByte instead of 128 kByte in the smaller model) is the smallest system of the NCube family; other members are the NCube/7 and the NCube/10 systems consisting of 128 and 1024 processors respectively.

1.1.1. The Hypercube Architecture

An n -dimensional hypercube computer, also known as a binary n -cube computer, is a multiprocessor characterised by $N = 2^n$ processors interconnected as an n -dimensional binary cube. Each processor P forms a node, or vertex, of the cube and has its own CPU and local memory. P has direct, hardware implemented communication paths to n other processors (its neighbours); these paths correspond to the edges of the cube. It is possible to assign 2^n distinct n -bit addresses to the various processors. The conventional method to address the nodes in a hypercube structure is to use the so-called Hamming Codes; whenever two nodes are neighbours, their addresses differ in one bit only.

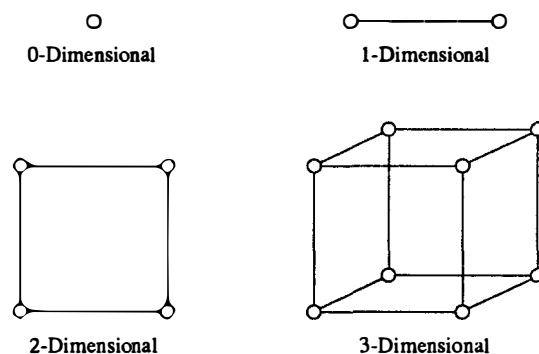


Figure 1.1.1: Hypercubes for $n = 0, 1, 2$, and 3 .

Figure 1.1.1 illustrates the topology for $n \leq 3$; a zero-dimensional hypercube is a conventional processor. An one-dimensional cube consists of two nodes which are labeled by a 0 and 1. The general procedure of constructing a hypercube and assigning binary addresses to its nodes employs the following recursive procedure: A n -dimensional cube is constructed from two $(n-1)$ -dimensional cubes. The labels in one of both latter cubes are prefixed with a 0 (the zero- $(n-1)$ cube) and those in the other cube with a 1 (the one- $(n-1)$ cube); then each node in the zero-cube is connected to its counterpart in the one-cube, i.e., to the node that has the identical address except for the prefix. Thus node P_{0x} is connected to P_{1x} [Hayes, 1986]. All hypercubes of higher dimensions can be generated by this procedure (see the 4-dimensional hypercube of figure 1.1.2).

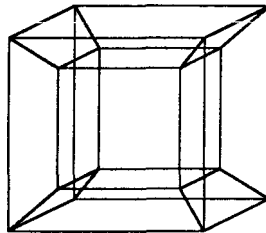


Figure 1.1.2: A 4-dimensional Hypercube.

The advantage of a hypercube structure above for example an array structure, is that the maximum distance between two processors, grows linear with respect to the dimension of the hypercube. Even in a 10-dimensional hypercube (with $2^{10} = 1024$ processors), the maximum distance between two processors is 10. If we compare this with a array structure (with $32 \times 32 = 1024$ processors), then the maximum distance between 2 processors will be 64. So, the underlying structure of hypercube machines provides us with a slowly increasing (logarithmically) maximum distance relative to the increasing number of processors. History

Proposals to build large hypercube computers have been made for more than 20 years. In 1962, Squire and Palace at the University of Michigan carried out a detailed paper design of a hypercube computer. They estimated that a 4096-node (12-dimensional) version of their machine would require about 20 times more components as the IBM Stretch, one of the largest and most complex computers in those days. Around 1975, IMS Associates announced the development of a 256 nodes (8-dimensional) hypercube machine, based on the 8080 microprocessor. Unfortunately, the machine has never been realised. In 1977, another proposal, by the University of Colombia was made for the development of a hypercube machine with more than 1 million processors. Obviously, the early hypercube designs were impractical because of the large number of combinatorial logic units and processors required.

The situation changed rapidly after the introduction of the VLSI technology in the early 1980s. Then it turned out to be possible to implement powerful 16/32-bit microprocessors on a single chip, moreover the RAM densities moved up to 1 Mbit. The first-working hypercube was demonstrated at Caltech in 1983. Each node consisted of a single-board microcomputer containing a 8086 microprocessor and a 8087 floating-point processor. In 1985, INTEL, influenced by Caltech's work, announced the development of hypercube machines with the 80286/287 CPU as a node processor. In this design, the maximum number of processors amounted to 128 nodes. The performance of the INTEL machine (12 MFLOPS) remained far behind the CRAY-1 (160 MFLOPS). The first commercially successful Hypercube was introduced by NCube: the NCube/10 system with a performance of 500 MFLOPS. These benchmarks are theoretical maximum performances of the various computers. Other recently developed hypercube-style machines with supercomputing potential include among others the Connection Machine, the Transputer-based hypercubes and Caltech's Mark III [Hayes, 1986].

1.1.2. Philosophy behind the NCube

One of the main problems when building large supercomputers has always been the system's reliability. In general, one would like to assign too many IC packages per processor and to use too many cables connecting boards together, resulting in a system that turned out to be too large to be reliable. One solution was found in custom silicon; it was possible to build a processor or a processing node extremely compact. With such a compact processor node and a low package count, a parallel supercomputer of manageable size could be built; and it was expected to achieve high reliability. However, without an established application software base, such a new computer system must provide substantial financial benefit to offset the software development costs. Therefore, the primary goal for NCube's hypercube implementation was set as a price/performance ratio improvement of one (or two) orders of magnitude over the existing supercomputers. But even after having reached this goal, the system would be of little value unless it should be not too difficult to program the machine while being assured of its reliability. The first goal to achieve was the programmability, which was best supported by a standard-user environment. This led to the development of AXIS, an UNIX-like operating system and a familiar tool box (in corporation with Shell Oil Company). Since a large number of companies have an extensive base of FORTRAN and C programs, NCube planned to provide the appropriate compilers. The second goal, improvement of the reliability, was achieved by reducing the package count and the number of connections. Finally, the reliability was even more increased by building a memory interface with error checking and correction (ECC), and designating communication channels with error detection and retry [Mokhoff, 1986].

1.2. A FURTHER INVESTIGATION OF THE NCUBE/4+

In this section we shall describe the NCube computers with respect to hardware as well as software. Investigating the hardware architecture, we shall usually refer to the NCube/4+ computer available at the Faculty of Mathematics and Informatics of Delft University of Technology.

1.2.1. Hardware

1.2.1.1. The NCube processor

The NCube Node Processor is a 32-bit processor with 512-kByte local memory. It is a single VLSI chip, including a full floating-point instruction set and the combinatorial logic needed for memory management and interprocessor communication. In 1983, the NCube company started the design of a node processor. Providing an idea about the size of the chip we mention that a single chip contains 160,000 transistors and is housed in a pin-grid-array package having 68 pins. An entire NCube node contains only 7 chips. This unusual compactness has prompted the introduction of a four-node ($n = 2$) IBM AT board. Four of such boards can be combined to construct a 16-node ($n = 4$) hypercube. Each processor has 11 full-duplex communication channels. Thus, the maximum configuration is a 10-dimensional hypercube (10 channels in use for the direct communication to neighbour nodes and 1 for I/O, e.g., the HOST, terminals, graphic displays, etc.). A node processor can communicate with other nodes by means of asynchronous DMA (Direct Memory Access) operations. Once a processor has initiated a *send* or a *receive* operation, it can continue with other operations while the DMA channel completes the internode-communication operation. Interrupts can be used to signal whether a channel is ready for a new operation. Each I/O full-duplex channel can transport data at 90 MByte/s. The host can receive up to 22 MByte/s [Mokhoff, 1986].

1.2.1.2. The Overall-System Architecture

The NCube/4+ is built up by 2 main components: an IBM AT (80286) functioning as host and a 16-processor (4 dimensional) hypercube. The hypercube is connected to the host by 4 I/O channels (as indicated in figure 1.2.1 by the numbers 1, 2, 3, and 4), implying that 4 processors are directly connected with the host, 8 processors are at a distance of 1, and 4 processors are at a distance of 2 (see figure 1.2.1).

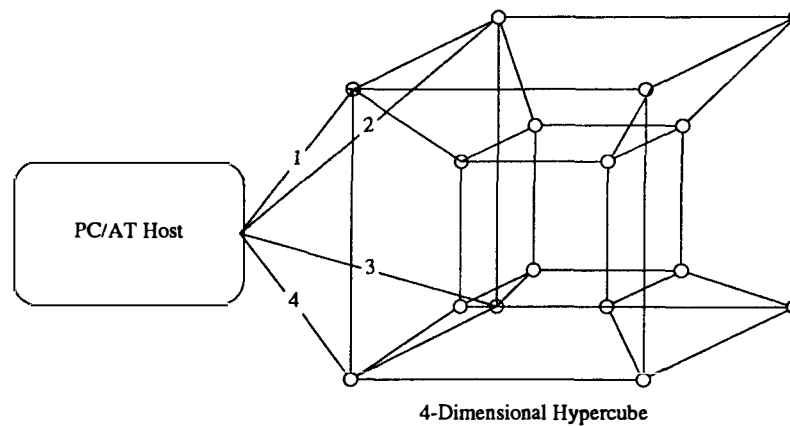


Figure 1.2.1: The Interconnection between the Hypercube and the Host.

The IBM System has a 72 MByte hard disk, from which 65 MByte is available for AXIS, the UNIX-like operating system of NCube. We have connected also two extra CIT-101 terminals to the NCube, as well as a Canon inkjet printer, a Modem and the DUNE entrance (Delft University of Technology neTwork system). We remark that it is only possible to have direct connection to the system with either the DUNE, or a Modem, or a terminal because only two serial ports are present. The overall system is shown in figure 1.2.2.

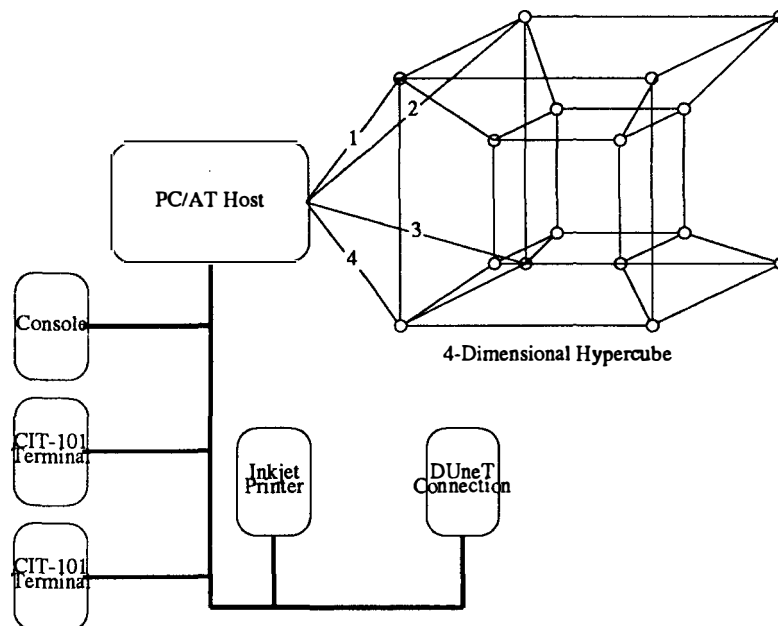


Figure 1.2.2: The overall-system architecture.

1.2.2. Software

The emergence of several commercial available hypercube computers has shown the feasibility of constructing a low-cost massively parallel computer. Much more difficult is the task of designing a sophisticated operating system (including several tools) and an adequate parallel programming language. The operating system is a major design issue, since memory management and interprocessor communication are critical to the functioning of the programming language. From these, three main issues are to be considered:

- The operating system as to be in use for the developing application programs on the hypercube.
- The operating system as to be in use for providing run-time support for application programs running on the various hypercube nodes.
- The set of application languages to be used.

On the NCube/4+ the operating system running on the IBM AT (80286) Host is called AXIS, the nucleus (a small operating system that provide communication between nodes by an efficient routing algorithm) running on the nodes is called VERTEX, the languages provided are FORTRAN and C [NCube Company, 1986].

1.2.2.1. AXIS

A well-known operating system for application program development providing the sort of environment associated with a *programmer's workbench* is UNIX. Unfortunately, there are two main versions of UNIX, System V and Bsd 4.3, as well as many less known variants. The system designer is left with a dilemma; he can work in a proven, widely-known development environment, but he cannot exploit the results of standardization. Because of the power provided by the UNIX-like operating systems, NCube has chosen to develop a new variant which fit on the NCube machines. It was not possible to use a standard UNIX version, because of the different structure of a parallel computer. AXIS, the NCube's variant on the UNIX family, embodies the features, common to the UNIX main dialect. Changes or additions can be readily made to AXIS when a true UNIX standard is agreed upon. Two features of AXIS greatly facilitate program development on a large hypercube:

- The ability to share files.
- How AXIS manages the main cube and subcubes.

On the NCube/10; AXIS runs on the 80286 host processor, this processor acts as a CPU for each I/O board. The AXIS environment supplies the user with editors, compilers, cross compilers, a symbolic debugger, text-formatting tools, etc. As in UNIX, all devices are treated as files. Massive parallel computers require a high I/O bandwidth if they must be useful for applications with a high communication grade (please read: not simply computation intensive). To avoid this problem, the NCube/10 has up to 8 I/O boards, meaning that there can be 8 separate file systems. However, AXIS provides capability to organise the eight file systems as one distributed file system.

AXIS controls a hypercube of nodes simply as a device, which is a file. A device can be opened, closed, written to, and read from as if it were a normal file. AXIS permits users to allocate subcubes that have the appropriate size for their applications. So one or more users treating large problems or several users (up to 3 on the NCube/4+ and 10 on the NCube/10+) with smaller problems can use the hypercube simultaneously. This flexibility greatly increases the system's efficiency and gives a hypercube supercomputer a significant advantage over conventional supercomputers. Partitioning the main hypercube into subcubes is simplified by the fact that each subcube is protected from access by another one.

1.2.2.2. VERTEX

Vertex is a small nucleus (less than 4k bytes) resident in each of the nodes; it serves as an operating system. Its primary function is to provide communication between the nodes in order not to bother the user about routing. This is achieved with *send* and *receive* system calls which transfer messages between any two nodes in the hypercube by activating the DMA interfaces. There are four basic functions:

- **Whoami**, that allows a program to determine the logical address of the node on which it is executing and the I/O processor (In the 4+ the IBM AT 80286) to which it is interconnected. It is implemented as a subroutine.
- **Nwrite**, the internode *send* function, implemented as subroutine call.
- **Nread**, the internode *receive* function, implemented as subroutine call.
- **Ntest**, a function that tests whether a message has arrived.

The maximum size of a message is 32k bytes. It is possible to assign a type to a message, distinguishing different messages and admitting a selective reception at a destination node or I/O board.

1.2.2.3. Parallel C and FORTRAN

The current NCube application languages (apart from the host and node assembly languages) are Fortran 77 and C. They were chosen because the computer is targeted to users interested in technological and scientific problems only; this particular group of users has traditionally programmed in Fortran and has started, recently, to program in C. Compilers for other languages, including OCCAM, are presently being developed. In the programming model chosen for the first set of languages (Fortran and C), a node is treated as a separate processor; no symbols are shared between nodes— the scope of a name is contained within a node; values are shared only by means of calls to Vertex subroutines *nwrite* and *nread*.

1.2.3. Performance

In table 1.2.1, the maximum performances of the different NCube configurations are shown. We note that the NCube performances are maximum performances achieved at maximum parallelisation, in fact, they are theoretical performances. In practice, it is a great result to achieve around 70% of this maximum performance. Each NCube processor has approximately a performance of 1.2 times the performance of a VAX 11/780. Below two different benchmarks for the NCube processor will be compared to benchmarks running on a VAX 11/780 (see table 1.2.2).

Type	#proc.	MIPS	MFLOPS
NCube 4	16	30	8
NCube 7	128	250	60
NCube 10	1024	2000	500

Table 1.2.1: Maximum Performance of the Different NCube Machines.

Processor	Fortran Drystones/s	Fortran Whetstones instr./s*
NCube node processor at 8 MHz	999	381,000
NCube node processor at 10 MHz (est.)	1249	476,000
Intel 80286 (NCube host) at 8 MHz with 80287 floating point co-processor	510	101,000
DEC VAX-11/780 with floating point accelerator	741	426,000

* Double precision.

Table 1.2.2: NCube-Processor Performance for two Benchmarks.

The benchmarks in table 1.2.2 are based on a general scientific calculation, the Drystone benchmark does

not use floating-point operations, the Whetstone benchmark does use a large number of floating-point operations [Weisbin, 1987].

1.3. DIFFERENCES WITH OTHER PARALLEL COMPUTERS

In this section, we shall describe some other parallel tree-oriented structures, and compare them with the NCube Hypercube Machines. The machines discussed are:

- The Connection Machine
- Transputers-Based Machines
- The DADO Production Machine
- Array- and Pipeline-Oriented Machines

1.3.1. The Connection Machine

The Connection Machine is a fine-grained parallel machine with general-purpose communication (this is, the different types of data structures and programs can be send over the same communication channels). Current Connection-Machine systems have either 16,384 or 65,536 processors, each with 4,096 bits of memory. There is nothing special about these particular numbers, other than being powers of two, but they are indicative of the machine's size. (The models sensibility increases with the number of processors.) The system has two parts: a front-end computer of the usual von Neumann style, and an array of Connection-Machine processors. Each processor in the array has a small amount of local memory, and to the front-end computer, the processor looks like a Memory. A typical front-end processor is a VAX or a Symbolics 3600 (see figure 1.3.1). The processor array is connected to the memory bus of the front end such that the local processor memories can be randomly accessed by the front-end; one word at the time, as if it were any other memory. This part of the memory of the front end is smart in the sense that the front-end can issue special commands that cause many parts of the memory to be simultaneously operated upon, or cause data to move around within the memory. The processor array therefore effectively extends the instruction set of the front-end processor to include instructions that operate on large amounts of data simultaneously. The processor array executes commands in SIMD form: there is a Single Instruction stream coming from the front end; the instructions act on Multiple Data items, on the order of one (or a few) per processor.

Two additional characteristics of the Connection-Machine Programming Model distinguish it from other systems:

- general, pointer-based communication.
- virtual processors.

- **The Communication**

Communication is provided by three independent communication networks:

- a one-bit wide global-OR network,
- a two-dimensional grid,
- a 12-dimensional hypercube with 16 processors at each vertex.

A global-OR network allows operations such as global-minimum, global-maximum, and global-OR to be performed in a time span proportional to the length of the operands. The grid allows instantaneous communication between nearest neighbours. The hypercube network supports full packet-switched communication between arbitrary processors. Communication is implemented in a SEND † instruction. The communication is very much like a postal system, where you can send a message to any one directly, provided you know the address, and where many letters can be routed at the same

† This instruction should not be confused with the *send* system call of the NCube system.

time. The SEND instruction can also take an additional operand that specifies what happens if two or more operands are sent to the same destination. The operands are to deliver to the destination the sum, maximum, etc. of the message. From a global point of view, the SEND instruction performs something like an arbitrary permutation on an array of items, although it is rather more general than a permutation because more than one item may be sent to the same destination. The pattern is not encoded in hardware, but is encoded in an array of pointers, and is completely under software control. A pattern, once constructed, can be used time and again to transfer data repeatedly in the same pattern.

- **Virtual Processors**

The Connection-Machine programming model is abstracted from hardware supporting details, and in particular, the number and size of its hardware processors. Programs are described in terms of virtual processors. In actual implementations, hardware processors are multiplexed as necessary to support this abstraction.

The Connection Machine's unique contribution to parallel processing is that it uses the hypercube architecture for artificial-intelligence applications. The 64,000 processors are said to operate at a performance level that exceeds 1000 times the logical-inference capabilities of current Lisp work stations.

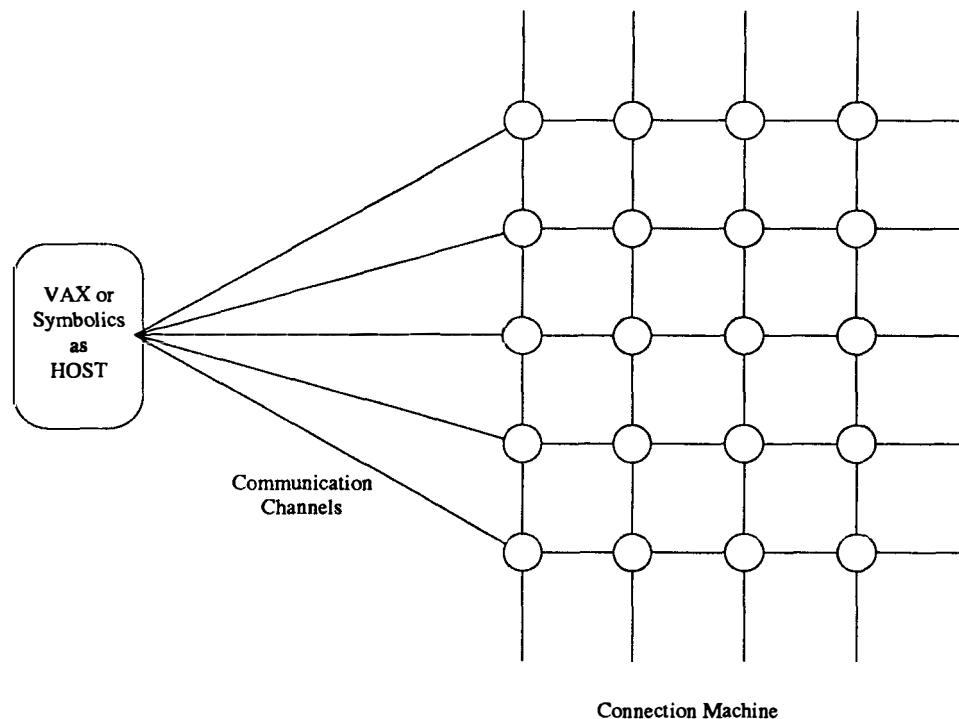


Figure 1.3.1: The Connection Machine.

The processors in the Connection Machine are not as powerful as the NCube processors, but there is an enormous amount of little processors. It may be regarded as a great-intelligent memory. Therefore, the Connection Machine is expected to fit especially applications such as VLSI design, real image analysis and free text data-base search [Stanfill and Waltz, 1986] [Stanfill and Kahle, 1986] [Hills et. al., 1986].

1.3.2. Transputers

1.3.2.1. The Transputer Concept

The name *transputer* reflects its ability to be used as a system building block. The word is derived from transistor and computer, since a transputer is both a computer on a chip and a silicon component like a transistor [May, 1986][Pountain, 1986].

1.3.2.2. The Programmable Component and Architecture

A transputer is programmable in OCCAM [May, 1986], a language especially designed for parallel computing. With OCCAM it is possible to exploit the potentialities of VLSI, providing performance and convenience through simplicity. The transputer provides a direct implementation of the process model of computing. A process is an independent computation, with its own program and data (see figure 1.3.2); they can communicate with other processes which are executing at the same time. Communication is by message passing using explicitly defined channels. The transputer is designed so that it can implement a set of concurrent processes. Special instructions share the processor time between the concurrent processes and perform interprocess communication (see figure 1.3.3). In addition, the transputer is designed so that its external behaviour corresponds to the formal model of a process. As a consequence, it is possible to program systems containing multiple interconnected transputers in which each processor implements a set of processes. Since a program is defined as a set of processes, it can be mapped onto such a system in a variety of ways, for example to minimise cost, or to optimise throughput, or to maximise the responsiveness to specific events (see figure 1.3.4). Whenever a transputer is programmed in OCCAM, the special properties are best exploited. Compiled OCCAM is as efficient as hand coding, thus the need for an assembly language is eliminated. OCCAM may be used on its own, or as a harness linking modules written in other high-level languages, thereby enabling them to be used concurrently. In these characteristics of a transputer we may see that it was originally designed as a Reduced Instruction Set Computer (RISC architecture).

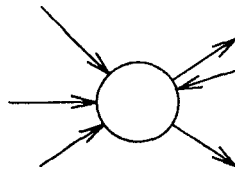


Figure 1.3.2: A Process is a Black Box.

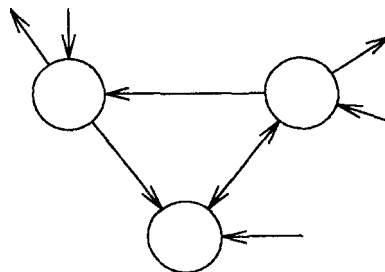


Figure 1.3.3: Processes can be Connected Together by Channels.

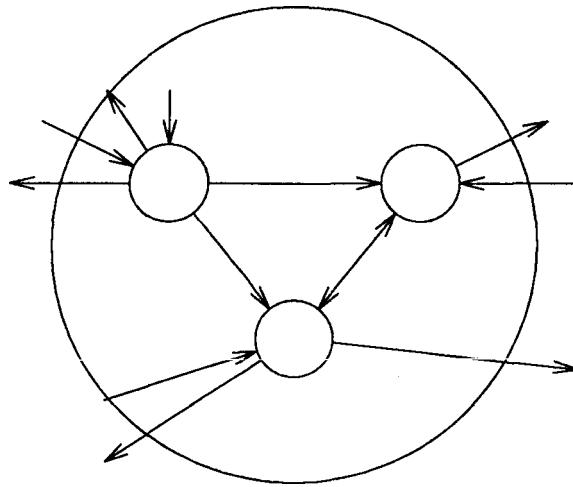


Figure 1.3.4: A Collection of Processes is itself a Process.

1.3.2.3. Applications

Pipelines and arrays of transputers can be used to provide greatly increased performance by exploiting the concurrency inherent in many applications. Signal processing, such as the fast-fourier-transform (FFT) algorithm, maps easily onto a pipeline.

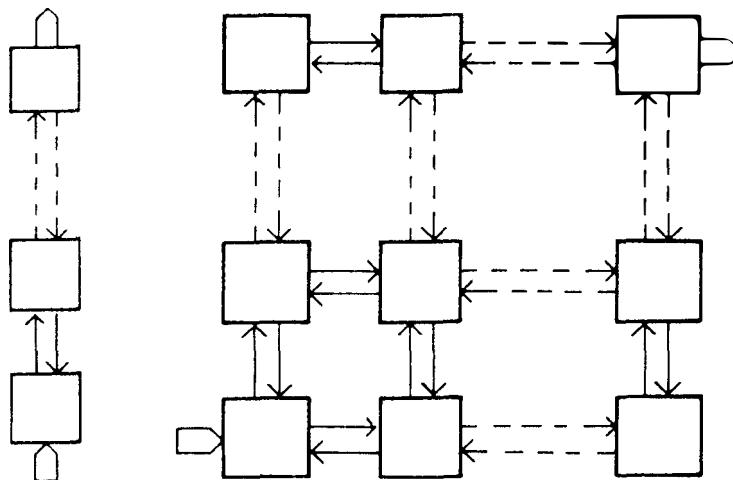


Figure 1.3.5: Different Transputer Configurations.

Transputers make it viable to add such arrays to CAD engineering workstations, enabling cost-effective and immediate-design analysis and hence improving designers productivity. Other applications, such as

image processing, finite element analysis (as used in weather forecasting), matrix manipulation, telephone switching systems, etc, can be easily implemented on arrays or networks of transputers (see figure 1.3.5). At this very moment, transputers are quite popular in the Computer Graphic Applications.

1.3.2.4. The T414 Transputer

The T414 is a 32-bit transputer providing up to 10 MIPS processing power with a memory and communication capability, all on a single CMOS chip in a 84 pin package. They are totally compatible at program and interface level with all transputer products. The processor has a reduced instruction set for compact programs, efficient high level language implementation and direct support for the OCCAM model of concurrency. The static memory (2k bytes) is giving a maximum internal data transfer rate of 80 Mbyte/sec. There is a memory interface providing a direct address space of up to 4 Gbyte with a maximum data transfer of 25 Mbyte/sec. The separation of the communication system from the memory interface enables these two interfaces to be optimised for their individual functions. In particular, the T414 is readily interfaced to other systems by means of a Link Adaptor, without compromising the flexibility or efficiency of its external memory interface. This typical configuration can be the basis for a performance accelerator for the IBM Personal Computer and ATARI ST computers. A single add-in board holds two modules: first, a T414 with 2 Mbyte of memory and four buffered links; second, an interfacing board, connecting a single buffered link to the PC (see figure 1.3.6).

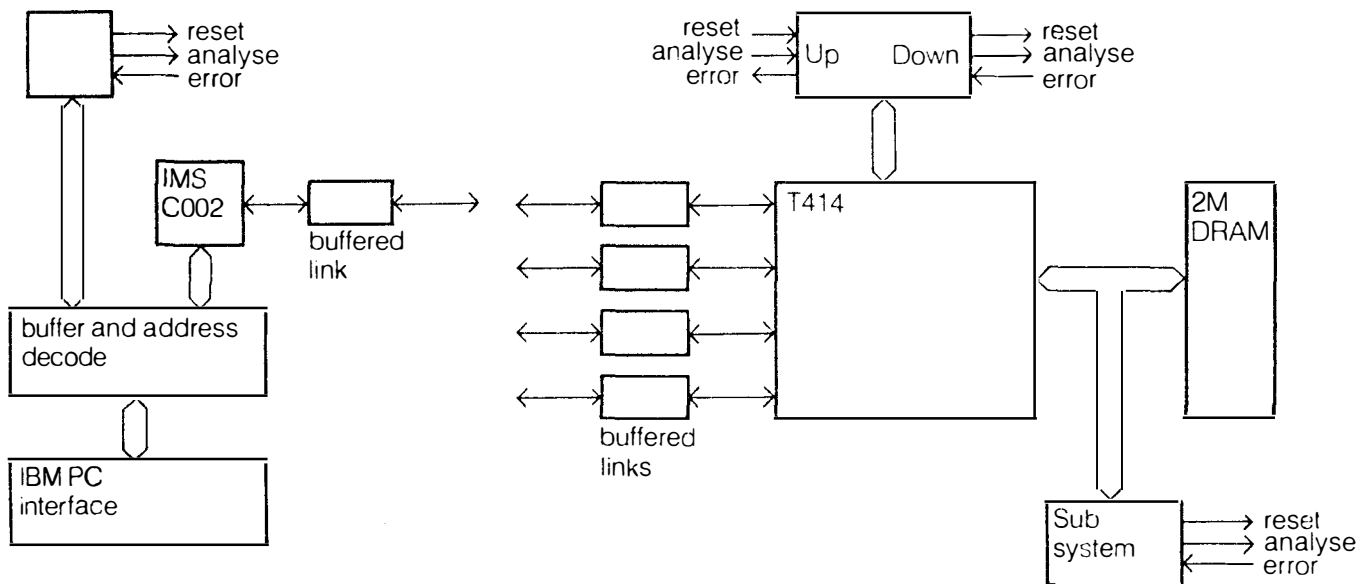


Figure 1.3.6: A Transputer on a IBM PC add-in board

The buffered links enable the board to be connected to other transputer boards, in networks of arbitrary size. A small amount of logic around the system services enables the board to act as a master or as a daisy-chained slave in the network.

1.3.2.5. Conclusions on Transputers

The Inmos Transputers are much faster than, for example, the NCube processors. OCCAM seems to be a better language for solving parallel problems, and the transputers also support High Level Languages as Fortran and C; all are advantages of the transputer over the NCube processors. However, a problem is that the only compiler yet available is the OCCAM II compiler, which is still bug sensitive.

Moreover, transputers have also some disadvantages. First, the communication routing is not arranged for the engineer, this can cause great problems, especially with the debugging of complex parallel programs. Second, the communication between transputers is not so fast as the communication between NCube processors. Third, all the available standard-library software for numerical programs written in Fortran or C cannot yet be used.

One of the companies that combined the Inmos transputers and the hypercube structure is Floating Point Systems (FPS). Using the OCCAM software language they have come up with an extremely modular computer that can accommodate from 8 to 16,384 nodes. While NCube's machines will run applications written in most of the common languages, such as Fortran and C, the programs for FPS systems are written in OCCAM. According to some authors OCCAM is a better solution because: "The idea of using parallel processing to solve the problems that cannot be solved with current architectures is not to try to fit a program that has been written for von Neumann-type machines into a hypercube, but to break the problem into chunks that can then be written into a parallel-processing language such as OCCAM." [Pountain, 1986]. It is agreed that one of the disadvantages of this approach is that most people still think in terms of UNIX environments instead of OCCAM environment.

Generally, we can say that the transputers are a promising new development, but especially the software support leaves us with some unsatisfied questions.

1.3.3. The DADO Production System

A production system is a pair $H = (V, P)$, where V is an alphabet and P is a finite subset of $V^* \times V^*$. The elements $(\alpha, \beta) \in P$ are called production rules. With the use of a production system, words can be transformed to other words by applying a production rule on the old word.

The DADO production machine is a medium-grain, parallel machine in which processing and memory are extensively merged. A full-scale production version of the DADO machine would comprise a large set of *processing elements* (PEs) (in the order of thousand), each containing its own processor, a small amount (20 kByte) of local RAM, and a specialised I/O switch. The PEs are interconnected to form a *complete binary tree*. Within the DADO Machine, each PE is capable of executing in either of two modes under the control of run-time software. The modes are:

- **SIMD mode** (Single Instruction stream / Multiple Data stream), the PE executes instructions broadcasted by some ancestor PE within the tree.
- **MIMD mode** (Multiple Instruction stream / Multiple Data stream), each PE executes instructions stored in its own local RAM. Because of the simplicity of the processor definitions, many simple processors can be implemented on a single chip. The reason why the binary-tree architecture was chosen are:
 - Binary trees are efficiently implemented in VLSI technology (see figure 1.3.7)
 - Broadcasting data to a large number of recipients is handled efficiently by tree structures.
 - Most important, the binary-tree topology is a natural fit for production-system programs.

There is a problem, for some applications (permutation of data within the tree, for example) the effective bandwidth of communication is restricted by the top of the tree. This *binary -tree bottleneck* does not arise in the execution of production systems. Because of the particular interest of the developers for production systems, and because Prolog may be considered as a special case of a production system, the DADO machine might support quite naturally performance improvements over conventional implementations. The lack of sufficient memory and the one-way communication make the machine suited for a restricted set of applications. In figure 1.3.8 a prototype Processing Element is shown. At this moment there is research on Speech Recognition, Expert Systems, Lisp and Prolog implementations [Salvatore, 1986].

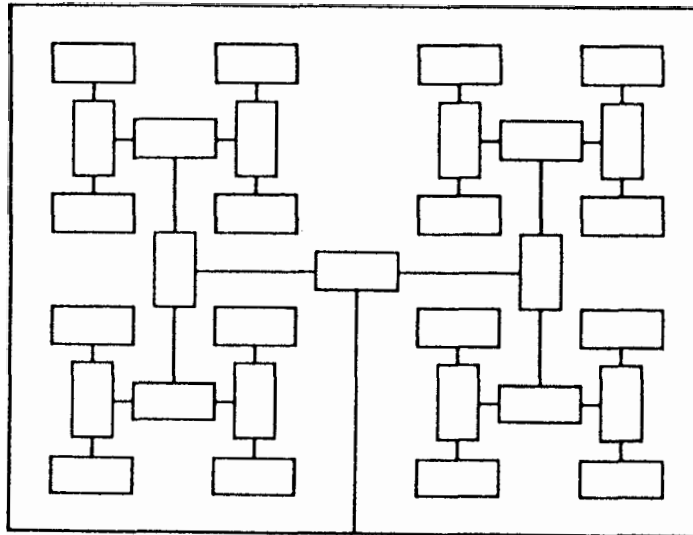


Figure 1.3.7: Implementation of a Binary Tree in the DADO Machine.

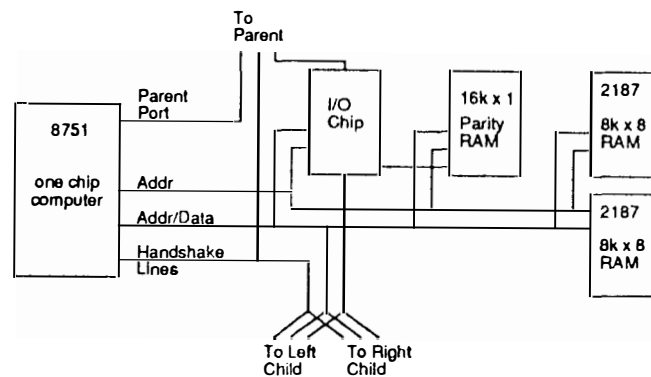


Figure 1.3.8: DADO2 Prototype Processing Element.

1.3.4. Array and Pipelined Machines

The earlier parallel machines were the array (ICL-DAP) and pipelined (CRAY) parallel computers. (In the chapter "Parallel Processing", these machines will be discussed further). These machines mostly work on SIMD or MISD (Multiple Instruction stream / Single Data stream). The ICL-DAP (SIMD) machine is a matrix of 64 processors (8x8), it is very well suited for parallel computation in the numerical field. The Cray's vector-oriented pipeline machines, are able to perform very fast calculations, but they are also very expensive. In figure 1.3.9 and 1.3.10 a typical array processor and a typical pipeline computer structure are shown.

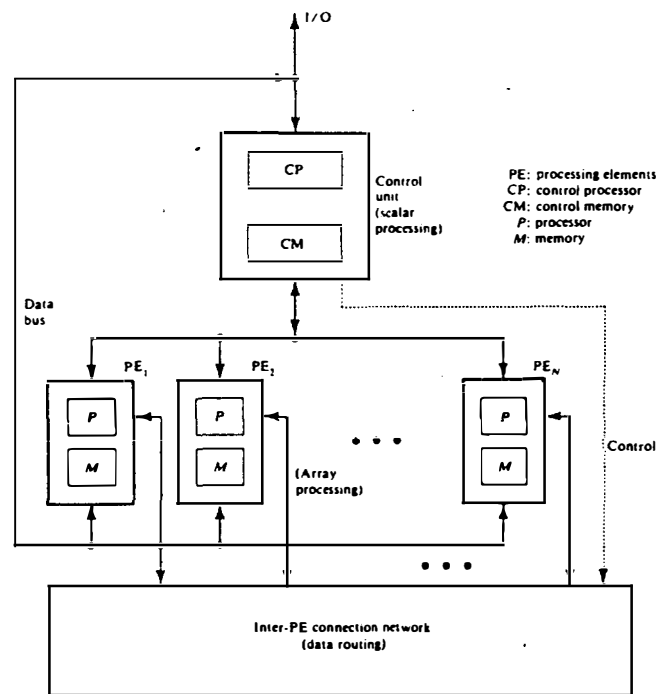


Figure 1.3.9: A Typical Array Processor

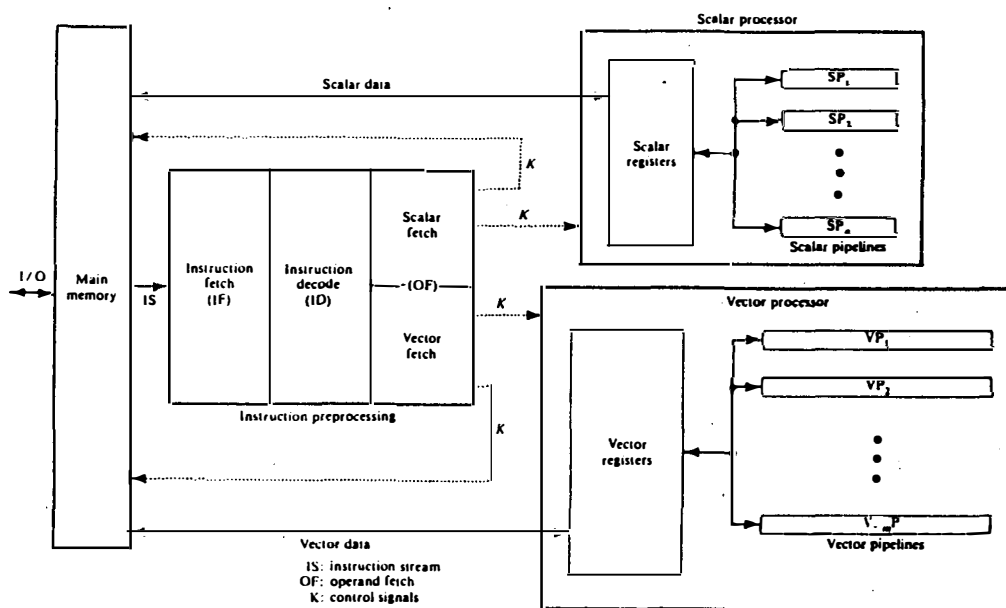


Figure 1.3.10: A Typical Pipeline-Computer Structure.

The main disadvantage of these classical parallel machines is that they are not very flexible and that they

are best suited for numerical problems. As soon as a symbolic problem has been attempted to be implemented on the machines, the performance decreases dramatically because of communication overhead [Japan, 1985]. We shall discuss this topic more carefully in chapter 2: "Parallel Processing".

1.4. CONCLUSIONS ON THE NCUBE/4+

In this section we shall summarize the advantages and the disadvantages of the NCube/4+ computer. We have dealt with them all in the previous part of this chapter. Here, we epitomise them item by item in order to provide the reader with a clear view at the various aspects of the NCube/4+.

- A significant difference between the hypercube and most other parallel processors is that the Multiple Instruction stream / Multiple Data stream (MIMD) machine uses message passing instead of shared variables for communication between processes. The dissimilarity of a message-passing machine, such as a hypercube, and a shared storage multiprocessor is that the former employs no switching network between processors and storage. The advantage of *this* architecture is that one can distinguish engineering concerns into processor-storage communication and interprocess communication. As a result, the so-called von Neumann bottleneck — the critical path in communications between an instruction processor and its Random Storage — can be engineered to exhibit a much smaller latency than when the processor and storage are physically localised.
- Since a hypercube can be defined recursively, all hypercubes are logically equivalent. According to NCube's Topman, John Palmer [1985], this implies two advantages:
 - 1 the program can be designed so that the dimension of the hypercube within which it will operate is a run-time parameter;
 - 2 the full hypercube can be treated as a resource that can be allocated in subcubes. Therefore, programs can be designed so that only their performance changes when they are running on different hypercubes.
- The subcube allocation assures that no computing resources are wasted. The user can request an appropriate subcube.
- The hypercube structure ascertain the smallest "maximum" distance between two nodes with as few as possible direct connections between the nodes.
- Another advantage of the NCube Hypercube over, for example, the transputers is that the routing of the interprocess communication is already taken care of.
- It may be said that the DADO machine is best suited for implementing binary trees, better than a hypercube (considering the costs of implementation). But the hypercube is doing average in implementing different kind of graphs. Although much of the software is written in C and Fortran, OCCAM seems to be a easier language for programming.
- In the beginning, the bad documentation provided by the NCube company is quite a problem, this holds especially for the hardware configuration.

1.5. NCUBE BUGS

Every computer, and especially a parallel computer, has a bug sensitive part. There are two main reasons why it is difficult to program an NCube Hypercube. First the NCube processors in the hypercube are 32-bit processors and the INTEL 80286 processor is a 16-bit processor, so that the communication between the nodes and the host is bug sensitive. For example, a integer instantiated on a node is a long integer on the host. Second, a software reason, AXIS is UNIX-like, and that is all, so to speak. An internal difference between AXIS and UNIX is: UNIX is not using shared memory for system and program variables. AXIS does not distinguish these two variables. Many of the operating problems are caused just by this reason. Especially when the system crashes unexpectedly, some parts of the file system sometimes cannot be restored: they are damaged. Another bug, is the absence of a large memory model on the NCube C compiler for the host. The INTEL 80286 processors can address a maximum of 64 kByte of memory for one segment. Whenever an user wants to exploit more memory for code or data, the compiler provides a overlay structure for the various segments. Unfortunately, the C compiler only knows the small memory model for 64 kByte code and 64 kByte data, and a medium memory model for more than 64 kByte code and 64

kByte data. The Large memory model is not implemented. This means that it is absolutely impossible to use more than 64 kByte data memory on the host.

1.6. MISCELLANEOUS ON THE NCUBE

In this section, some interesting non-technical information on the NCube will be given, i.e., the NCube User Group, and other NCube-research fields.

1.6.1. The NCube Usergroup

The NCube User Group is a gathering of NCube hypercube machine users. It is founded to provide a quick distribution of NCube software and to increase the machine's popularity by extending the software library. The User Group supplies editors, chess games, window systems, etc. The User Group's address is:

Dr. R. Peskin
RUTGERS State University
C.A.I.P.
Hill Center
Bush Campus
New Brunswick
New Jersey, USA.
Phone: 08903-201/932-4208

1.6.2. Vision Algorithms on the Hypercube

Mudge et al. [1987] have illustrated how a number of typical Computer-Vision algorithms can be executed on a hypercube. The steps in the automatic inspection of Thick-Film (TF) circuits are described. These circuits are a network of conductors and dielectrics printed onto a ceramic substrate. The circuits are populated with electronic components, but, prior to this, and as a quality control step, each TF circuit is inspected to see if it satisfies a set of geometric specifications. A TF circuit is imaged as a composite of about 40 frames each composed of 512x512 1-byte pixels to obtain the resolution desired. This has resulted in about 10 Mbyte of data to be processed per substate. The following steps are performed during the inspection:

- **Tonal mapping**
This implies the adjusting of unevenness or imperfections in the image device.
- **Allignment**
The translation and rotation of the image to a reference position and absolute orientation is performed. It is very calculation intensive.
- **Edge Detection**
In this phase, a plain edge operator is applied on the data, such as the Sobel operator.
- **Reference Check**
Here, the segmented image output from the previous stage is compared with prestored templates to determine if there are any geometric violations.
- **Error Reporting**
The final step interprets the results of the reference checking.

A model for a hypercube machine has been developed and its performance was estimated for this characteristic task. Based on simple operation counts, a lower bound of 0.7 seconds was obtained for the inspection problem. These results indicated that hypercube machines have great potential for low- to intermediate-level computer-vision algorithms. The fact that hypercubes are MIMD machines makes them also good candidates for the high-level function processing [Mudge, 1986].

1.6.3. Intelligent-Machine Vision

Over the last decade, advances in machine perception (vision, tactile sensing, etc.) have been closely associated with progress in computer architectures. In the past, emphasis has been on SIMD architectures for simple preprocessing tasks as convolution and mask evaluation; pipelined architectures have been used for executing sequences, e.g., blurring, differentiation, zero-crossing. To develop a realistic vision system one need to go beyond these retina-level tasks and include some of the higher level components of human perception. For example, humans easily perceive characteristics such as collinearity, direction, periodicity, coarseness and continuity. This suggests that a machine-vision system needs efficient mathematical transformations to handle these effects. For such complex processing options, where the type of processing is influenced by the context of the image, MIMD concurrent computers are required. Despite the fact that neurons process data more slowly than silicon, human perception is extremely fast, implying that human perception achieves its speed through massive parallelism, involving perhaps billions of processing elements. The same will almost certainly be true of successful machine-vision systems. To achieve this goal, the CESAR research program on human analogue vision is investigating algorithms that can be executed on concurrent computer architectures such as the NCube/10 [Weisbin, 1987].

1.6.4. Hypercube Algorithms for Robot Dynamics

For many of the uses of mechanical manipulators today, real-time performance can be achieved in a well-structured environment, using quasi standard, bus-based multi-microprocessor architectures. Such manipulators generally consists of a base-anchored, open-link articulated chain, composed of a few rigid links connected by rotational or prismatic joints. The dynamic behaviour of a manipulator is modeled by a set of coupled, highly non-linear equations of motion. These equations must be solved for the design of algorithms or at the machine-intelligence level of the autonomous robot. Future robot designs will emphasise structural flexibility and joint compliance, developments that will significantly increase the complexity of the equations of motion and associated control algorithms. Furthermore, since these calculations need to be carried out in concert with other activities as vision, sensor fusion, navigation, etc., robots will need on-board concurrent computers. The NCube/10 is a promising new development in this research area [Barhen, 1986].

1.6.5. Embedding Trees in a Hypercube

In the study of parallel computing, networks of processors are often organised with the help of various configurations such as trees, pyramids and mesh-arrays. Often, these configurations are represented as graphs. If the properties and the structure of the underlying graph are used accurately, the computation and communication speed can often be improved. The dimension of the cube grows very slowly (logarithmically) with respect to the number of nodes in the hypercube. The hypercube and hypercube-like structures are very well suited for sorting and computing the fast Fourier transform. The hypercube can transmit a large number of data quickly. Wu [1985] has proved that a hypercube is a good host graph for the embedding of processor networks because of its low degree and low diameter (read longest distance) with respect to the number of nodes in a cube (e.g., in a cube with 16 nodes, the dimension and the diameter are both 4). Graphs such as trees and arrays can be embedded into a hypercube with small costs. The embedding of (full) binary (k -ary) trees and graphs in hypercubes are described in Wu [1985]. It seems that k -ary (with $k \geq 3$) trees can be embedded best in hypercubes. (Binary trees have a very high dilation cost (a large number of processors are not used), and graphs have a very high expansion cost (there is often a need for more directly connected processors, i.e., the graph splits up in 12 directions).)

CHAPTER 2

PARALLEL PROCESSING

A Brief Survey

ABSTRACT

In this chapter we describe some of the well known methods to parallelise a process; pipelining, array processors, multiprocessors, and data-flow systems are treated. The various architectures will be compared in considering some simple applications. Finally, we present some methods taken from the literature to parallelise a special problem on a specific architecture.

2.1. BACKGROUND

Parallel processing (e.g., computations on parallel computers) has always been a difficult issue in computer technology. Up to this moment no general methodologies exist in order to parallelise any problem without deep study of its characteristics. Only specific problems have been successfully parallelised after an thorough examination of its intricacies. A well-known method to investigate the communication of parallel algorithms on deadlocks and other problems, is using Petri Networks [Houck, 1987]. In the previous chapter we have briefly discussed some parallel architectures and their functional properties briefly. In this chapter we focus on the main principles of parallel processing. Various topics will be introduced, examples will be given, and, finally, we shall compare the different architectures towards some applications. The reason why we discuss these subjects is to provide a basis for the recommendation of a methodology to parallelise a problem. We discuss this in Part 3: "Results and Conclusions".

2.1.1. Historical Background

In the past, there was always a need for faster computers. The speed-up required was achieved by faster components, super conduction and also by better architectures and superior processing techniques. The main changes in architectures have been Direct Memory Access (DMA), I/O channels, time sharing, multi programming, multiplexing, etc. The biggest change in architectures has been, however, the development of parallel architectures. In the last decade this development has shown a tremendous increase because of the exponential grow of hardware quality, VLSI, etc. In the previous chapter a summary of the different possibilities is given. Below, we concentrate on the main concepts for parallel processing.

2.1.2. The Main Methods

The main concepts used in parallel processing are:

- Pipeline Computers
- Array Processors
- Multiprocessor Systems
- Data Flow Machines

A pipeline computer performs overlapped computations to exploit temporal parallelism. An array processor uses multiple synchronised arithmetic logic units to achieve spatial parallelism. A multiprocessor system achieves asynchronous parallelism through a set of interactive processors frequently with shared resources

(memories, databases, etc.) [Japan, 1985]. The above given classification of parallel mini supercomputers is not the only possible classification. Besides this, it is a classification with fuzzy borders; most modern supercomputers use pipelines and multiprocessor concepts at the same time. The main difference between an array processor and a multiprocessor concept is, that in the array processor, the processors work *synchronously*; multiprocessor systems, on the other hand, may work *asynchronously* [Japan, 1985]. The data-flow machines are a new concept [Stanfill et al, 1986]. They demands extensive hardware to achieve massive parallelism. The rapid progress in VLSI has made this possible.

2.2. PIPELINE PROCESSORS

A particular vector instruction operates on a vector (i.e. a sequence of operands). Most existing pipeline computers emphasise vector processing. Due to the close relation, pipeline and vector computers are often used synonymously in discussion on parallel and supercomputers.

Pipelining can be understood by reference to an assembly line. Consider a car manufacturing plant. It may be segmented into m stages, where each stage carries out a particular operation on the vehicle. If each stage requires no more than t minutes to complete, then the assembly line can move at a rate of one stage per t minutes. Consequently, it requires tm minutes at most for one car to be produced. Now consider a car that has just finished stage 1 and is moving to stage 2; as soon as stage 1 is passed a new car can be moved into stage 1, and so on. After tm minutes, the first car has passed through the entire assembly line. Since there is a car following, the second car can be ready after another t minutes. In fact, when the assembly line is filled, each t minutes a car will be produced.

The same principle may be applied to computers. For example, a pipelined floating-point multiplication is shown in figure 2.2.1 [Japan, 1985].

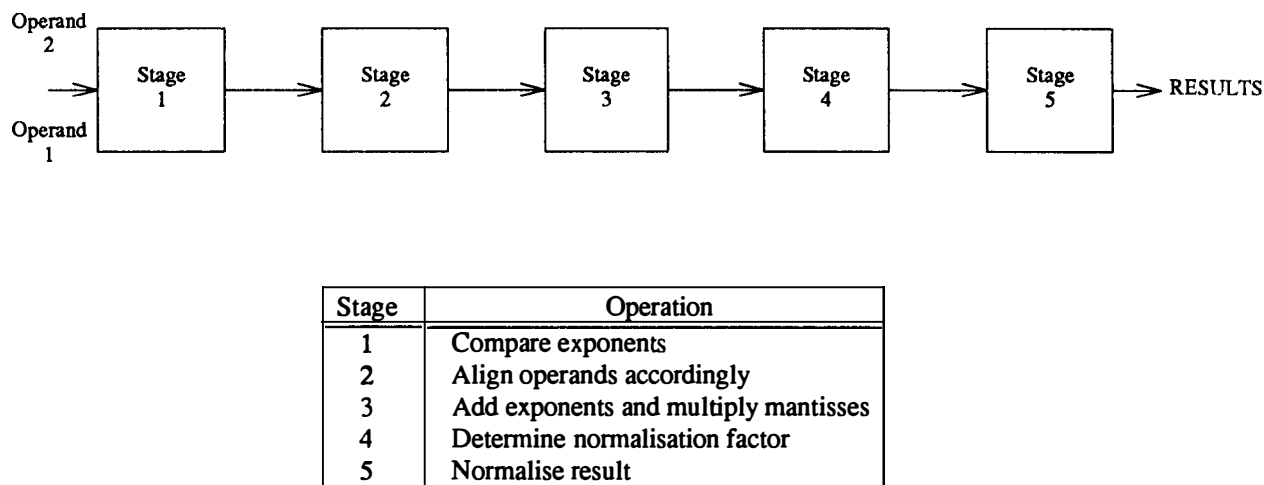


Figure 2.2.1: A Pipelined Floating-Point Multiplication.

Suppose, that each stage requires at most t clock cycles to complete. Then we can say that:

- The start-up time to fill the pipe is $5t$.
- The total data movement to keep the pipe full is three items (2 operands and 1 result) per t clock periods.

Three points are important:

- The shorter t is the faster the result can be produced ($t = \frac{1}{\text{result rate}}$). If a floating point multiplication requires at best k clock cycles, then there must be at least k segments in the pipeline for $t = 1$.

- The start-up time of the pipe is given by kt . Therefore, for a given t , k should be kept as minimal as possible. But on the other hand, the speed-up is k , so k should be as great as possible. However, a physical limitation on k exists, so this does not work.
- The shorter t is, the greater the bandwidth for the data movement outside the pipe. For $t = 1$ this is three items per clock cycle per pipe.

The high bandwidth is the reason why vector processing became a more and more popular technique. This technique is designed to exploit the bandwidth by keeping the pipes full for a substantial period of time.

The vector-processing technique uses instructions that can work on vectors. The instructions can send a number of vector elements to a number of pipes via multi-paths buses in the first clock cycle, the next elements of the vectors are sent to the same pipelines after t clock cycles. This is continued until the pipe is filled, then the first results will appear at the output end of the pipe. After each clock cycle, some result will follow. The total processing time for a vector is:

$$\text{total processing time} = \text{start-up time} + t \times (\text{vector length} - 1).$$

Since the start-up time is static for a particular pipeline and independent of the length of the vector to be processed, the relative speed of the vector instruction (that is, the number of operations per unit time achieved) is increased by as well maximising the start-up time by maximising the length of the vector to be processed. [Japan, 1985]. in figure 2.2.2 a functional structure of a modern pipeline computer can be seen.

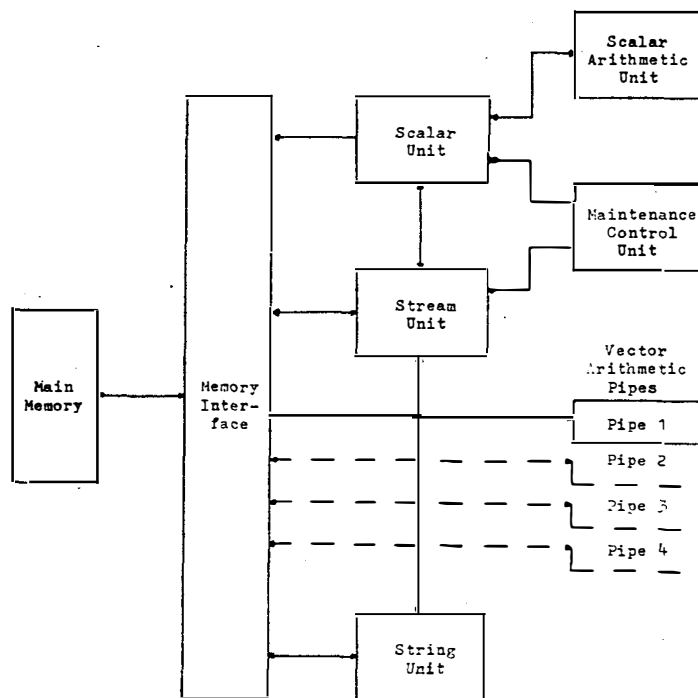


Figure 2.2.2: Functional Structure of a Modern Pipeline Computer.

According to the level of processing, the following classification scheme for pipeline processors can be given:

- **Arithmetic Pipelining.** In a pipeline computer, the different arithmetic units can be segmented into various data formats for pipeline operations. Wellknown arithmetic pipeline operations are the four stage pipes used in a Star 100, the eight-stage pipes used in the TI ASC, the up to 14 pipeline stages used in the Cray 1 and the up to 26 pipeline stages used in the Cyber 205.
- **Instruction Pipelining.** The execution of a stream of instructions can be pipelined by overlapping the current instruction with fetch, decode and operand fetch of subsequent instructions. This technique is also known as the instruction look-ahead. Almost all high performance computers are equipped with instruction-execution pipelines.
- **Processor Pipelining.** The pipeline processes the same data stream by a cascade of processors, each of which processes a specific task. The data stream passes the first processor and the result is stored in a memory block, which is also accessible by the second processor. The second processor then passes the refined results on to the third, etc.

Other classification schemes are unfunction versus multifunction pipelines, static versus dynamic pipelines and scalar versus vector pipelines [Japan, 1985].

2.3. ARRAY PROCESSORS

An array processor is a single instruction stream multiple data stream (SIMD) computer. They are best suited to perform vector computations on matrices or arrays of data.

An array processor is a synchronous parallel computer with multiple arithmetic logic units (ALUs), called processing elements (PEs). The PEs operate under the supervision of a single control unit (CU). By replication of the ALUs one can achieve spatial parallelism. Each PE consists of an ALU with registers and a local memory. In order to provide the system with an efficient communication method, a routing algorithm must be present among the PEs. For each application a special interconnection pattern may be under control of the central CPU.

Processors of this kind use a single instruction to act on many sets of operands or data simultaneously. This requires the availability of many arithmetic units (PEs). In figure 2.3.1, the functional structure of a SIMD processor is shown.

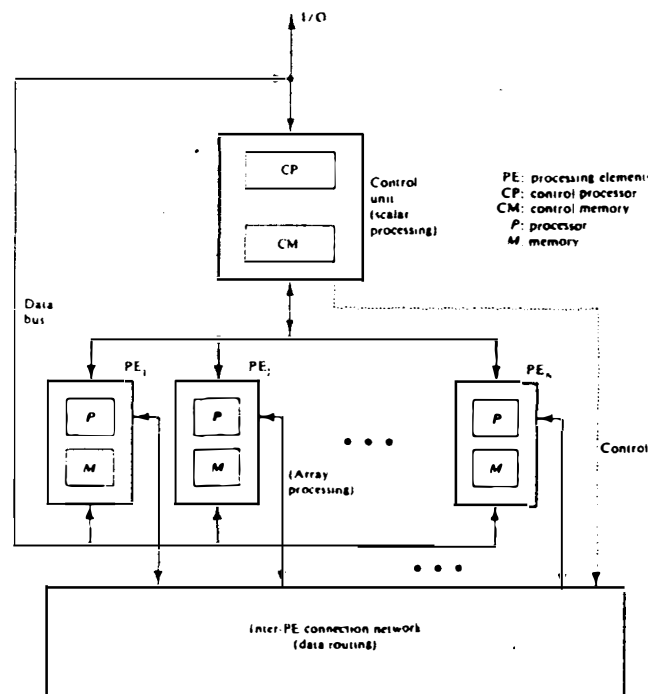


Figure 2.3.1: Functional Structure of an SIMD Processor.

The system can be seen as a host (a computer), a central controller and an array (a number of PEs). The host has its local memory, arithmetic unit, registers, and control unit (see figure 2.3.1). The difference between

the control processor and the PEs is that the PEs cannot interpret conditional branch instructions and, in effect, they operate as a slave to the control processor. Sometimes, in array processors the local memories may be supplemented or even replaced by a global and common memory with a suitable organisation and controller.

The computer works as follows: the control unit fetches an instruction and determines whether or not it is a matrix operation; if it is not a matrix operation, then the control unit executes the operation. If it is a matrix operation, the control unit passes the instruction to the PEs, each PE holding the matrix part being operated on inside its local memory. Each PE holds some element of the matrix operands; however, if the size of the matrices exceeds the number of PEs in the array, the PEs may hold a row column, or submatrix of the matrices. All PEs receive instruction from the control processor simultaneously, therefore the entire matrix operation proceeds in parallel. Each PE may be active or disabled during an instruction cycle. A masking vector is used to control the status of all PEs. The same reason why there is a need for a high bandwidth in vector processing, can be resumed here: the PEs should maintain a high bandwidth link to the memory and a high bandwidth link to the other PEs and the control processor. Therefore the instruction stream from the host could best be buffered. Instruction fetching can be done in parallel with operations that do not require the use of a common memory, and throughput can be enhanced substantially by the use of techniques such as pipelining and multiple instruction issuing for many instruction pairs. The latter of these two options allows scalar operations in the control processor that are being performed in parallel with matrix operations, that are being performed within the PEs

Examples of array processors are the Illiac-IV system (which does no longer exists), ICL-DAP and the Burroughs Parallel Element Processor Ensemble (PEPE).

2.4. MULTIPLE-PROCESSOR SYSTEMS

A typical multi-processor organisation is conceptually depicted in figure 2.4.1.

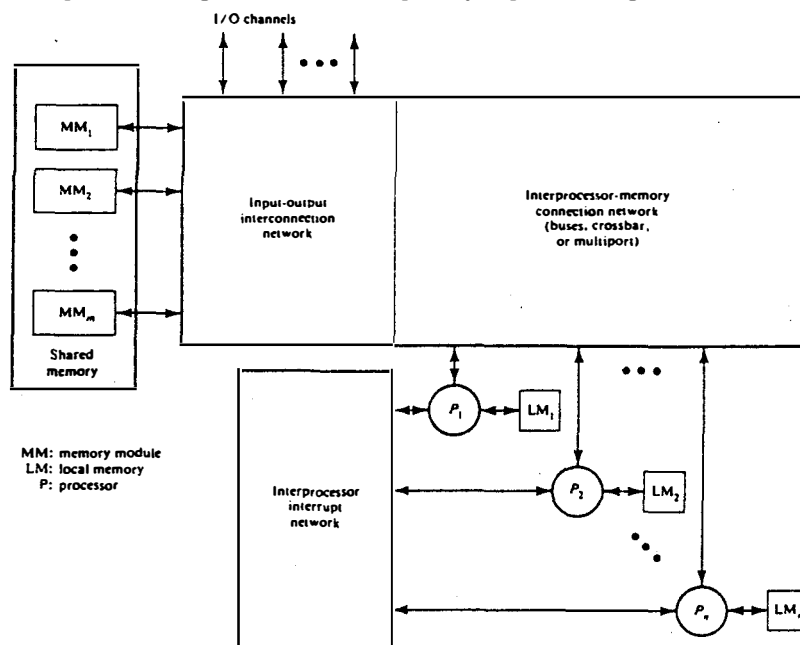


Figure 2.4.1: A Typical Multi-processor Concept.

All processors are connected to common sets of memory modules, I/O channels, and peripheral devices. The entire system is controlled by a single integrated operating system, which provides interactions between processors and the programs at various levels. Besides the shared memories and the I/O devices, each processor has its own local memory and private devices. Interprocessor communication can be done through shared memories or through an interrupt network.

The interconnection between the memories and the processors is determined primarily by the hardware-

system organisation. In more recent systems, software-based connection schemes, are used next to the hardware schemes. Three different types of interconnections have been practised in the past: A time-shared single or multiple paths bus, a crossbar-switch network and multiport memories.

The NCube/4+ computer (with MIMD structure and separated control units for each PE) is a good example of interrupt-based communication and multiple processors, each having their own local memory and no shared memory.

2.5. DATA-FLOW MACHINES

Recently, the data-flow machines have obtained much more interest. The Connection Machine and the Neural Networks are relevant instances of these machines. It seems that that SIMD and the MIMD still conform to the von Neuman model. In the latter two computers there is but an original sequence of instructions determining which operations will be executed in parallel, which data is shared, and which memory addresses will be used in parallel. These machines rely, in principle, too much on the centralised concept. As long as this concept is maintained the new computer architectures will suffer from more complex forms of the von Neuman bottleneck. Of course, it is possible to use algorithm's that does not suffer so much from inter-process communication, but recent research in, e.g., parallel knowledge-based systems has shown that the bottleneck can be suppressed temporarily. However, when the number of processors exceeds the number required for massive parallelism, the bottleneck as indicated will occur. [Hofland, 1987] [Herik, 1987] [Henseler, 1986]. Therefore, hundreds of thousands processors, working together on a single problem seems to be impossible. A new direction in computer science propagates the executing of instruction streams, whose sequence has not been determined in advance. Data-flow machines solve a problem by having separate copies of data carried by each instruction that need them, rather than referring to memory address references. Instead of having separate instruction counters to tell the processors what to do next, the processors can work independently. These programs can be designed in terms of data streams with simple instructions, instead of terms of operations working on data in memory locations. For these data flow machines, massive parallelism is needed with many processors, each able to contain a little memory (some kByte) and able to execute some basic instructions, but this fits with the concept of giving priority to speed and programming ease at the expense of hardware.

2.6. PARALLELISING A PROBLEM

One of the biggest puzzles in parallelising a problem is to find independent parts of the application that can be processed concurrent. This holds true the most for pipelining and array processing. Whenever one has a MIMD or a Data-Flow machine, an application can be split into some independent and some dependent parts, but the programmer must be aware, that the more dependent the processes on the different processors are, the higher the need for communication becomes, and as told above, increasing communication always ends up in a bottleneck. In this part we describe some applications for pipelining, array processing, MIMD processing and Data Flow Machines. This should provide us with a basis for further research on a recommendant concerning a methodology to parallelise a problem (see Part 3: "Results and Conclusions").

2.6.1. Vector Processing

The notions pipeline computers and vector processing are often used to indicate the same subject: a vectorised pipelined computer architecture. The main speed-up in pipeline processing is obtained by a constantly filled pipe. Hence, there must be a constant flow of data, without too big gaps. The speed-up in vector processing is obtained by executing a vector instead of a single operand. The problem must be parallelised, in order to have optimal profit of the parallelism. In vectorised pipeline-oriented machines, both aspects are important. The vector aspect requires a problem that can be vectorised, the pipeline aspect yields for a broad bandwidth of the data.

An example of a vectorised pipeline machine is the CONVEX C 1, a mini-super computer installed at the Computer Center of the Delft University of Technology. A compiler optimises the program code by vectorising it. Whenever a program is written on this machine, one should design the program in such a way that the inner-loop is greater than the outer-loop, because the compiler vectorises only the inner-loop. Next to the vectorising compiler, the user is also able to write vectorised program code himself. Besides these vector aspects, the user should provide the system with a constant stream of data, hence the pipes will be saturated constantly. More details can be found in [Kerckhoffs, 1987]. These computers are specially suited

to matrix manipulations and signal calculations. Differential equation solutions are more difficult, because these systems cannot be vectorised easily.

2.6.2. Array Processing

In this section, we discuss the possibility to parallelise a matrix multiplication on an array processor. Resume, that an array processor is a synchronous parallel computer, with a central control, providing the system with a instruction stream. The instruction stream is identical for all the processors, the data may vary. The instructions on the array processors cannot execute conditional branches. The data consist mostly of matrix elements, submatrices, matrix-columns or rows. The advantages of these architectures can be explained best by an example.

In normal serial executing, a matrix multiplication can be seen as the following problem:

$$c_{ij} = \sum_{k=1}^m a_{ik} b_{kj} \text{ for all } i,j$$

A Fortran program to execute this multiplication has the following lay-out [Hossfeld, 1983]:

```

DO 1 I = 1,N
  DO 2 J = 1,L
    T = 0
    DO 3 K = 1,M
      T = T + A(I,K)*B(K,J)
    3 CONTINUE
    C(I,J) = T
  2 CONTINUE
1 CONTINUE

```

Now, we want to execute some instructions simultaneously with the following instruction:

```
DO SIM { (Operation)/(Index Set) }
```

what means that the operation is executed (for a group of indices in parallel or by multiple processors). Now, it is possible to parallelise the problem with a orthogonal parallel method [Hossfeld, 1983]. The program looks as follows:

```

DO SIM { C(I,J) = 0 / (I = 1,N; J = 1,L) }

DO 1 K = 1, M

DO SIM { C(I,J) = C(I,J)+A(I,K)*B(K,J) / (I=1,N; J=1,L) }

1 CONTINUE.

```

If we use $P = N*L$ parallel processors, we can calculate a new index for the matrix multiplication really fast. Suppose we want to realise this algorithm on an array-processor architecture. The single processors need 4 registers, and to multiply a $N \times N$ matrix, we assume $N \times N$ processors ($PE(1,1) \dots PE(N,N)$ available).

The registers are called the AREG, BREG, CREG, TREG. The processors are connected with their 4 neighbours as shown in figure 2.6.1.

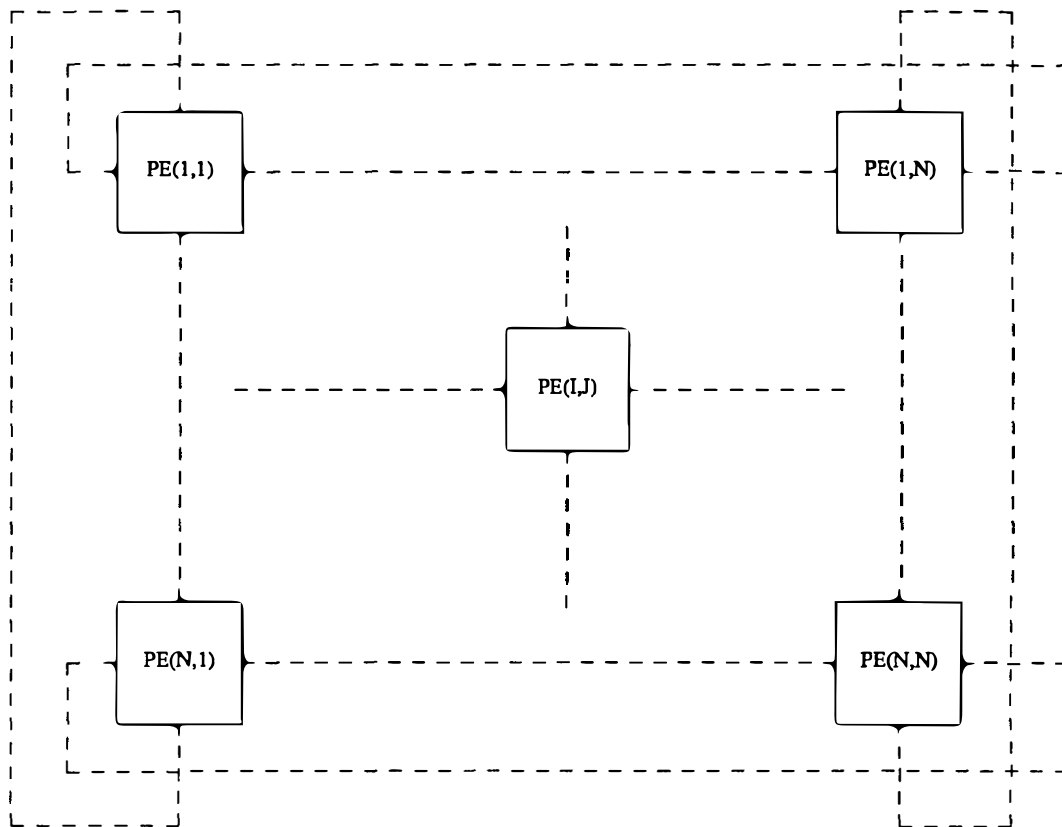


Figure 2.6.1: An (NxN)-Parallel-Processor Array.

Note, that the side-processors are connected cyclicly with the outer processors on the other side of the array-processor. The array is so constructed is done in order easily to shift results from one to another processor. Each processor has 4 neighbours. The algorithm can be split up in six parts [Hossfeld, 1983]:

- Initialisation:

$\text{CREG}(I,J) = 0$
 $\text{BREG}(I,J) = B(I,J)$
 $\text{AREG}(I,J) = A(I,J)$

for all (I,J) synchronously in the PE(I,J)s.

- Shifting of A,B:

The I^{th} part of A is shifted to the left for all $I \leq N$.
 The J^{th} part of B is shifted upwards for all $J \leq N$.

- Multiplication:

$\text{TREG} = \text{AREG} * \text{BREG}$

- Addition:

$$\text{CREG} = \text{CREG} + \text{TREG}$$

- Shifting:

- Shift all the AREGs to the neighbour on the right side, parallel for all PEs:

$$\text{AREG}(I, J+1) \text{ becomes } \text{AREG}(I, J)$$

the outer processors are shifted cyclicly.

- Shift all the BREG to the underlaying PEs parallel for all PEs, so:

$$\text{BREG}(I+1, J) \text{ becomes } \text{BREG}(I, J)$$

the outer PEs are shifted cyclicly.

- The Jump:

Until the N^{th} step in the parallel process, jump to the label multiplication. The final results are stored in $\text{CREG}(I, J)$.

This algorithm was made by Cannon in 1969. The array-processor systems are specially usable in scene analysis and signal calculations. There is a need for a broad bandwidth, in order to keep the system filled.

2.6.3. Multiple-Instruction Multiple Data Processing

MIMD architectures can operate independently of the central-host system. The host loads the programs in the parallel processors, provides them with data, and finally, get the result back. The computers in the parallel array can work asynchronously, and can communicate to each other, without intervention of the host. Whenever the local memory of these processors is about half a MByte, large systems can be built. One of the examples of these computer structures is a hypercube machine. This topological structure is fit specially for sorting and Fast Fourier Transform. (because there is a need for communication between various processors, laying all over the system; the hypercube structure provides the user with a high number of processors and a relatively low maximum distance between the processors). Examples of specific hypercube-oriented algorithms can be found in the NCube users manual version 1.1. [NCube, 1986]. An algorithm that uses the structure of a hypercube in order to split a A4 page in 16 different parts with good load balance, is described in the chapter Preprocessing of part 2 of this work.

2.6.4. Data-Flow Processing

Whenever the number of processors in MIMD exceeds the number used in massive parallelism (e.g., 2^{19} processors), and the instructions are simple basic operations, working on different data, we can call this MIMD method: Data Flow. The instruction sequence is determined by means of the data entering the processor elements. The PE fires depending on the data at the input and on the programmed basic instruction. A good example of the advantages of data-flow machines is the free-text-search application. Suppose we have around 2^{19} small scientific articles in a data base, and we want to search the text for the keywords "Artificial Intelligence". In a sequential computer, we have to parse all the texts for these keywords, an endless search. Now suppose we have a Connection Machine (discussed in the chapter 1 of part 1) with 2^{19} processors, then we can distribute all the different articles over the Connection Machine PEs and we let each processor search for the keywords "Artificial Intelligence". Supposing that the articles all fit in the PEs local memory, we have a linear speed up of 2^{19} . These machines are specially fit for dynamic analysis and free-text search. An even more futuristic approach are the neural networks, built of millions of very small combinatorial hardware elements, that fires new data depending on the input.

2.7. CONCLUSIONS

After having seen the various possibilities in parallel-processing architectures and a few examples of algorithms for these systems, we put forward the question: "Is there a generic method to parallelise a specific problem?". In other words, which architectures fit for which problems, and which algorithms are then available? After our literature study on this question we tentatively conclude, that there is almost no literature on this subject. Sometimes a little hint for a methodology is given by making a few careful remarks. The little experience in symbolic parallel computing is probably the reason for this lack of publication on methodology in this area.

If we compare the structures and the methods used in parallel processing in this chapter, we see that one of the biggest problem is to keep the data flowing, and to reduce communication to a minimum, in order to avoid bottlenecks in the system. Next to this, a good load-balancing is needed. Recently, the parallel computer architectures have changed rapidly, also due to the decreasing costs of expensive hardware. Yet, as we have seen in the software crisis in the late sixties, a good computer is nothing without good algorithms, and it is quite hard to build robust, well-working parallel algorithms without a good methodology.

The first hint we want to provide the reader is the following: whenever there are large amounts of data and less knowledge involved in the process, methods as pipelining, vectorising (if possible), and array processing (even on a MIMD array structure) can be used properly. More difficult becomes the problem if there are much knowledge and a few data, then a MIMD or a data-flow approach is preferable. A good knowledge load-balancing should avoid the occurrence of bottlenecks. In Part 3: "Results and Conclusions", we shall discuss this subject more thoroughly.

CHAPTER 3

PATTERN RECOGNITION

A Brief Introduction

ABSTRACT

In this chapter we give a brief description of the most interesting areas of pattern recognition. The main issues will be the difference between statistical and syntactical pattern recognition. We do not discuss the field of scene analysis and image processing, only the most basic principles of pattern recognition will be treated.

3.1. BACKGROUND

Pattern Recognition is a multi-disciplinary attempt to automate human capabilities such as vision and speech and understanding. [Devijfer et al., 1982]. Researchers of all kind of different areas are working on as well as the theoretical as the practical development of pattern-recognition systems. The automatic pattern recognition was made possible by the increasing computer power. Pattern recognition can be seen as a combination of the following research areas:

- Pure and Applied Mathematics
- Technical Science
- Linguistic Science
- Computer Science
- Artificial Intelligence

There are many influences of the biology, medical and psychological sciences. In psychology, the central problem within the study of pattern recognition is the biological and mental process by which external signals stimulating our sense organs are converted into meaningful perceptual experiences, or how we are able to assign a name of some kind to a complex stimulus. Before recognition, a pattern must first be perceived by our sense organs. If such a pattern should give a meaningful perceptual experience, it should have been recognised before, in other words, it should be recognised as a pattern of the same class.

Pattern recognition is a very broad field of activities with very fuzzy borders. Nevertheless, it can be said, that a pattern-recognition problem can be reformulated as a pattern-classification problem, or it can be divided in a number of subproblems, which consists each of a number (one or more) pattern classification problems. Optical character recognition is a problem which belongs to the first type of problems. There is a class for each of the 26 characters. Circuit-diagram analysis is a problem of the second category, there are different subsystems, e.g., symbols for electronic components, connecting lines, and terminals. While we consider a large amount of pattern-recognition problems as pattern-classification problems, the pattern classification can be seen as a partial solution for the pattern recognition [Duda et al., 1973].

The pattern recognition research started its history with a certain flourish of romanticism, they were called automatic learning machines, mechanisation of the thought processes, etc. These expectations were, however, deflated sooner or later. However, at this moment machines exist that read typed or hand-printed text, sort the mail, count blood cells, analyse electro-cardiograms, perform visual inspection, recognise spoken words, identify speakers, etc.

3.2. THE STRUCTURE OF A PATTERN-RECOGNITION SYSTEM

It can be said, that a pattern-recognition system is a information processing system able to classify an amount of input data into different classes, provide a description of the objects, provide the objects with a label, or perform an action. As input signals are considered different measured signals, descriptions of objects, or conditions retrieved from one or more sensors [Backer, 1981].

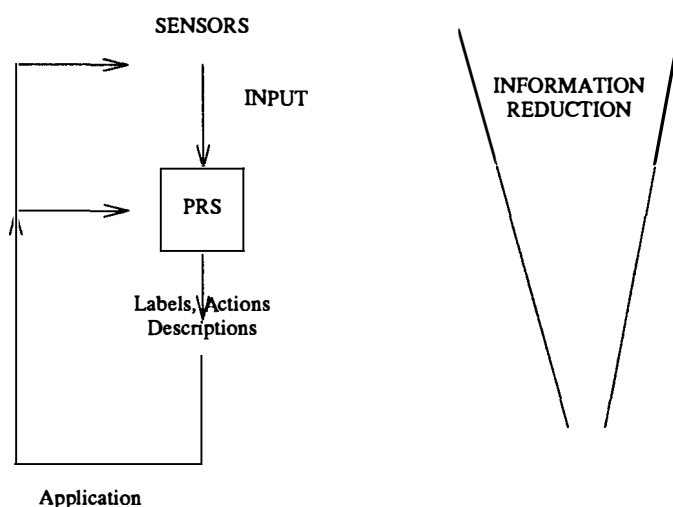


Figure 3.2.1: Information-Reduction by a Pattern-Recognition System.

In figure 3.2.1, a model of a pattern-recognition system is shown. It has two main aspects:

- There is an enormous information reduction (data reduction) from the sensor to e.g. an action or label. The system is only interested in the relevant information, which is, in practice, always much less than the provided information.
- Depending on the application, the sensors are chosen and the system's architecture is developed.

Generally, the organisation of a pattern-recognition system consists of the following two subsystems, namely: (i) from sensor to data representation and (ii) from data representation to classification (see figure 3.2.2).

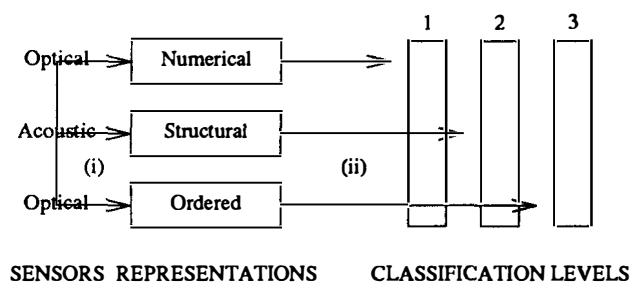


Figure 3.2.2: The Structure of a Pattern-Recognition System

The different requirements and constraints for the two subsystems are often controversial. There also seems to be a very strong dependency between (i) and (ii). Even with one sensor, one representation method and

one classification level we have a very complex optimisation problem which has to be solved by the Pattern Recognition System.

3.3. DIFFERENT APPROACHES IN PATTERN RECOGNITION

In pattern recognition, there are as many different tasks to be automated, as there are different approaches to do this. The first diversity can be found in the way a pattern is represented, this can be an array of numbers, obtained by, e.g., sampling a waveform, scanning an image at selected grid points, or performing a sequence of binary (logical) tests. Next to this difference in approaches, a pattern class (the group a pattern belongs to in order to be recognised) can also be represented by one or several early discovered patterns. In the pattern recognition, patterns are represented in a pattern-representation space, consisting of as many dimensions as the number of measurements represented by the corresponding array. In such a representation a pattern can be seen as a point in this space. In order to recognise a pattern, there is a need for a pattern variability for different patterns, and a pattern similarity for similar patterns. We can recognise patterns by assuming that the points in the pattern space for similar patterns should somehow cluster together in a region, and, dissimilar patterns should be represented by points laying in different regions in the pattern space. This "Gestalt" interpretation for patterns and pattern classes is the basis for a large amount of early research in pattern recognition [Devijver et al., 1982]. In the beginning, it was realised that statistics and probability theory had much to offer to pattern classification. Firstly, in some instances, classical, multivariate statistical distributions, defined over the pattern space, seemed to provide an adequate model for the variability of the pattern representations. Secondly, the question whether a given pattern "belongs" in some pattern class may naturally be treated as a test of a hypothesis, or as a special case of a statistical decision-theory problem.

However, as time past by, it was discovered that the statistical pattern recognition showed some built-in limitations. Therefore, some other approaches have been proposed. For instance, when a decision is made based upon statistical hypotheses which are mutually exclusive, one becomes aware quite fast that a large amount of real-world problems cannot be formulated in this terminology. The fuzzy set theory may provide some solution in this case. The membership of a pattern to a class can then be represented as a fuzzy membership in the underlying decision model.

There is, however, a more important weakness in statistical pattern recognition. This is the bad handling of contextual or structural information. Sometimes, contextual information can be found in statistical dependence of successive patterns (as in trigram analysis, see the chapter "Optical Character Recognition") as in "c.nt.xt". However there are also cases where it is impossible to fit a framework of structural knowledge in a statistical model. Earlier work in Artificial Intelligence provided the researchers with a model of formal languages, rule-based approaches and other knowledge-representing techniques. These techniques seemed to be especially good-working in the classification stage. This approach was called the structural (or linguistic, or syntactical) pattern recognition.

None of the two above methods is best; hence, the best method should be combination of the two methods. One of these combinations can be found in the hierarchical pattern recognition.

The next step was to use more artificial-intelligence techniques to control the pattern recognition process. AI techniques may help only in the first phase (sensor to representation) but also in the second phase (representation to classification). After the classification, AI techniques even may be used to improve the results by checking the contextual information.

In the next sections, the approaches mentioned above will be discussed in more detail.

3.3.1. Statistical Pattern Recognition

3.3.1.1. Pattern Similarity

As told above, in the beginning, the statistical approach was the most natural to classify different patterns. We can see the input of a pattern recognition system as d measurements, then it is logical to represent the input as a d -dimensional vector, a point in the pattern space (d -space). The components of this vector are the so-called features. The features are measured by a sensor, and then converted in a for the machine suitable representation. These sensor systems are the systems' acquisition mechanism. Now, the pattern recognition system must assign the input to one of c known classes. Two patterns should be assigned to the same class when they are similar, and assigned to different classes when they are dissimilar.

The acquisition of the data should be noiseless, bi-unique (two different patterns should provide the system with two different representations, and two of the same patterns should give a similar representation) and continuous. In other words, a small distance in the pattern space between two patterns, must correspond with a high degree of similarity (not the way around!).

In pattern recognition the conditions must be specified in a one-to-one relation between distance and similarity; this can be done by using a statistical approach.

3.3.1.2. Pattern Variability

We have assumed that pattern representations were obtained through an ideal, noiseless acquisition system. Unfortunately, those systems do not exist. There is always a lot of distortion and noise introduced by the acquisition system which cause the unwanted variability of the pattern representation, e.g., the drift in TV cameras, noise in communication channels, etc. Very frequently, patterns in the same class differ rather much. In many cases, the scope of the problem has been reduced in order to get a manageable number of measurements, because of too complex representations. A good example can be found in the history of Optical Character-Recognition Systems, first developed to read a single-font text, later multi-font machine's were introduced. Eventually, constrained handwritten text was to be read by machines (unconstrained handwritten recognition remains quite a challenge). Even with the first machine (single-font text), the intrinsic variability of the patterns is due to such diverse causes as grain, quality, aging, colour, etc. It goes without saying that the variability of multi-font and hand-printed characters increases by order of magnitude. The growing complexity can be seen as an intrinsic variability, the noisy input as a extraneous variability. The cumulative effect of those two variabilities provides the researcher with a large number of problems. What must be done is to reduce the inner-class variability and to increase the outer-class variability. This is called the feature-extraction problem. The second problem is to assemble the patterns in well-separated groups or clusters. Then, the pattern-classification problem becomes one of partitioning the feature space into regions, one region for each class.

3.3.1.3. Pattern Classification

3.3.1.3.1. The Bayes Decision Theory

The Bayes Decision Theory is a statistical approach to the pattern-recognition problem. As described above, the pattern-recognition problem can be regarded as a classification problem, where the decision of a pattern belonging to a class is a decision to be formulated in statistical terms, assuming that all the relevant probability values are known. This can best be illustrated with an example [Duda et al. 1973]. We have to build a classifier to separate two kinds of lumber: ash and birch. Suppose that an observer watching lumber emerge from the mill finds it so hard to predict what type will emerge next that the sequence of types of lumber appears to be random. We can say that as a piece of lumber emerges, nature is in one or the other of the two possible states: either the lumber is ash or the lumber is birch. We let ω denote the state of the nature, with $\omega = \omega_1$ for ash and $\omega = \omega_2$ for birch. We can consider ω to be a random variable. We assume that there is some *a priori* probability $P(\omega_1)$ that the next piece is ash, and some *a priori* probability $P(\omega_2)$ that the next piece is birch. Suppose that we are forced to make a decision whether a type of number will appear next without being allowed to see it, and the only information allowed to use is the value of the *a priori* probabilities, then we can use the following decision rule: Decide ω_1 if $P(\omega_1) > P(\omega_2)$; otherwise decide ω_2 . But, in most circumstances one is not asked to make a decision with so little information, e.g., we can use the brightness measurement x as evidence. Different samples of lumber will yield different brightness readings, and it is very natural to express this variability in terms of probabilities; we consider x to be a continuous random variable whose distribution depends on the state of nature. Let $P(x | \omega_j)$ be the state-conditional probability density function for x , then we can calculate $P(\omega_j | x)$ with the Bayes Rule:

$$P(\omega_j | x) = \frac{p(x | \omega_j)P(\omega_j)}{p(x)},$$

where

$$p(x) = \sum_{j=1}^2 p(x | \omega_j)P(\omega_j).$$

Part One

Our decision rule is then:

$$\omega_1 \text{ if } P(\omega_1 | x) > P(\omega_2 | x); \text{ otherwise } \omega_2.$$

The structure of the Bayes classifier is determined primarily by the conditional densities $p(x | \omega_i)$. The most popular of the various density functions that have been investigated, is the multi-variate normal density. To give a small indication of the complexity of the statistical pattern recognition we provide the reader with the formula for this distribution [Duda et al., 1973],

$$F(\underline{x}) = |G|^{-1/2} (2\pi)^{-d/2} e^{-1/2(\underline{x}-\underline{\mu})' G^{-1}(\underline{x}-\underline{\mu})} = N(\underline{\mu}, G),$$

with:

d = Dimension Parameter Space

G = Covariance Matrix

$\underline{\mu}$ = Mean

$\underline{x} = (x_1, x_2, \dots, x_k)'$

In figure 3.3.1, a representation is given of a bivariate normal distribution.

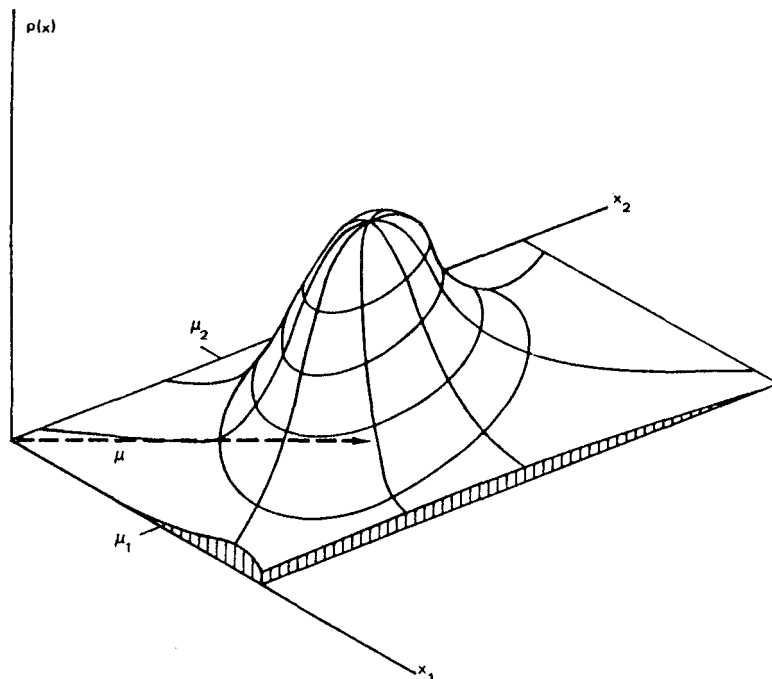


Figure 3.3.1: Bivariate Normal Distribution

There is a problem with this theory: for most pattern-classification applications, the main problem in applying these results is that the conditional densities $p(x | \omega_j)$ are not known. In some cases we may know the form of these densities, but may not know the characterising parameter values. The classical case occurs when the densities are known to be, or can be assumed to be, multi variate normal, but the values of the mean vectors and the covariance matrix are unknown. We can then try to estimate the parameter values.

This estimation can be done with parametric methods (when we know the kind of distribution but not the parameter values) and with non-parametric methods (when we don't even know the kind of distribution). These two techniques will be discussed in the next two sections.

3.3.1.3.2. Parametric methods

In typical pattern classification problems, the estimation of the *a priori* probabilities presents no serious problems, the estimation of the class-conditional densities is quite another matter. The number of available samples always seems to be too small, and serious problems arise when the dimensionality of the feature vector $\mathbf{x}$ is large. If we then for example suppose that $p(\mathbf{x} | \omega_j)$ is a normal density with mean μ and covariance matrix σ_j , we can reduce the problem significantly. Then, we only have to estimate the mean and covariance to find the underlying distribution. A way to estimate these parameters is the Maximum Likelihood Estimation. Suppose that we have X , containing n samples: $X = \{x_1, x_2, \dots, x_n\}$. Then, since the samples were independently we have [Duda et al., 1973],

$$p(X | \theta) = \prod_{k=1}^n p(x_k | \theta)$$

Viewed as a function of θ , $p(X | \theta)$ is called the likelihood of θ with respect to the set of samples. The maximum likelihood estimate of θ is, by definition, the value $\hat{\theta}$ that maximises $p(X | \theta)$ (see figure 3.3.2).

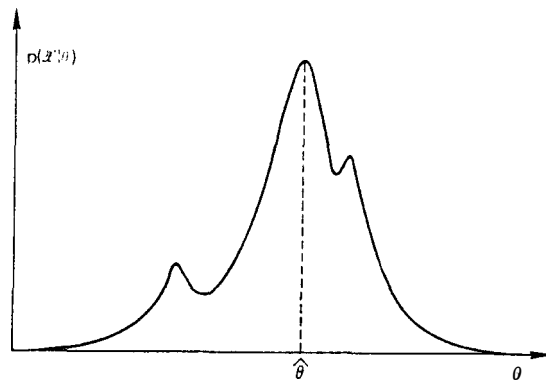


Figure 3.3.2: The Maximum Likelihood Estimate for a Parameter θ .

Suppose we have a multi-variate normal distribution with unknown mean:

$$P(x_k | \mu) = |G|^{-1/2} (2\pi)^{-d/2} e^{-1/2(x_k - \mu)' G^{-1} (x_k - \mu)}$$

then we can estimate the MLE for the mean by:

$$\hat{\mu} = \frac{1}{n} \sum_{k=1}^n x_k$$

If, however neither the mean or the covariance matrix is known, then we can estimate MLE parameters by :

$$\hat{\mu} = \frac{1}{n} \sum_{k=1}^n x_k$$

and

$$\hat{G} = \frac{1}{n} \sum_{k=1}^n (x_k - \hat{\mu})(x_k - \hat{\mu})^t.$$

But, it is known that the Maximum Likelihood Estimation for a covariance matrix is biased; that is, the expected value of $\hat{G}$ is not equal to G . The unbiased estimate is found by:

$$\hat{G} = \frac{1}{n-1} \sum_{k=1}^n (x_k - \hat{\mu})(x_k - \hat{\mu})^t$$

We can then calculate the *a priori* probabilities with a corresponding Bayes Rule. All the other formulas are then derived in a familiar way [Duda et al., 1973].

3.3.1.3.3. Non-Parametric Methods

In the past section we have assumed that the form of the underlying density functions are known; in most pattern recognition applications this assumption is suspect. The common parametric forms rarely fit in the densities actually encountered in practice. There are some non-parametric procedures that can be used without assuming that the forms of the underlying densities are known.

A well-known estimator for unknown distribution is the Parzen estimator [Devijver et al., 1982]. Let us consider the information about $P(\underline{x} | \omega_i)$ conveyed by each individual observation in the learning set. Given $\underline{x}_j \in \omega_i$, we can assert that $P(\underline{x} | \omega_i)$ takes a non-zero value at the point $\underline{x}_j$. Moreover, under the assumption that the distribution $P(\underline{x} | \omega_i)$ is continuous, we can infer that $P(\underline{x} | \omega_i)$ will assume non-zero values in the immediate vicinity of point $\underline{x}_j$.

This information about the distribution, gained by observing $\underline{x}_j \in \omega_i$ can be represented by a kernel function: $K(\underline{x}, \underline{x}_j)$. A one dimensional example of it is shown in figure 3.3.3.

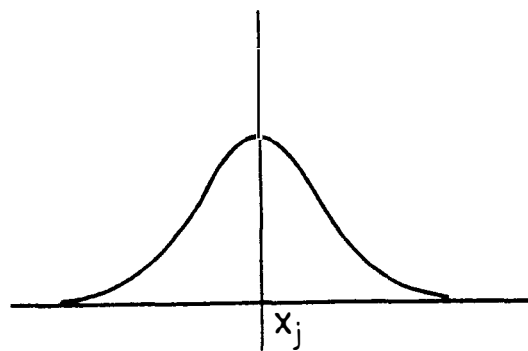


Figure 3.3.3: Example of a Kernel.

Adding the contributions of all these points and dividing their number, we get an estimate of $P(\underline{x} | \omega_i)$ (see figure 3.3.4). Then, the distribution is:

$$\hat{P}(\underline{x} | \omega_i) = \frac{1}{n_i} \sum_{j=1}^{n_i} K(\underline{x}, x_j)$$

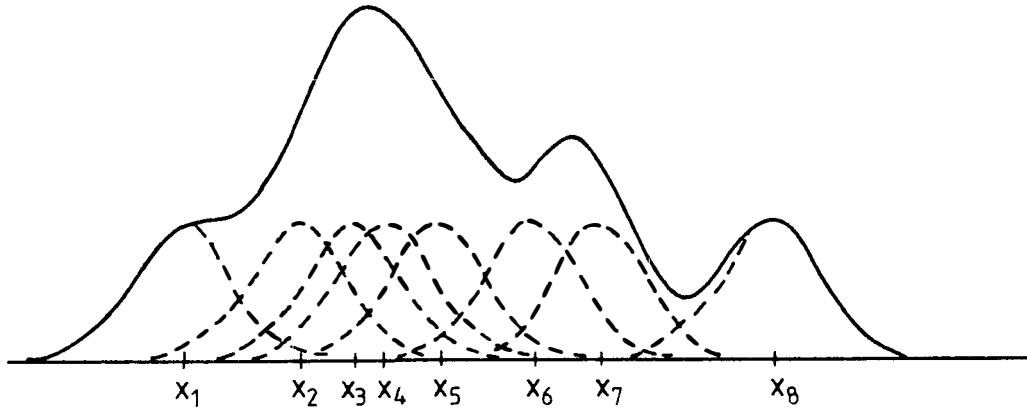


Figure 3.3.4: Summing the Distributions of the Kernels to Obtain a Parzen Probability Density Function Estimate.

3.3.1.4. Feature Extraction

As we have seen, patterns can be represented by d -dimensional measurements vectors obtained by some data-acquisition mechanism. The selection of measurements is a very crucial step in the pattern recognition system design.

Good features enhance within-class pattern similarity and between-class pattern dissimilarity. Sometimes, good cases are evident, e.g., to distinguish an O from a Q it should be sufficient to look for the tail, but these examples are exceptional. It is impossible to give the best method for determining what the best measurements are in all cases. The more complex the pattern becomes, the more complex this problem is going to be. A very common way is to select all those measurements that could provide valuable information. However, an increasing number of measurements brings about an increasingly complex classifier structure. This is what we call the feature extraction problem. There are two approaches to feature extraction: the first is to select a best set of d measurements out of D , and maximise the figure of merit for this subset of d elements. This involves a combinatorial explosion, and once a feature set has been selected, this approach calls for a redesigning of the data-acquisition system. The second method, feature extraction in the true sense of the word, is to select a number of d features, each of which is a combination of all of the initial D measurements, $d < D$. This method can be seen as a projection of the original D -dimensional feature space to a d -dimensional subspace. Both techniques reduce the complexity of designing the classifier. Two common used methods to search for the best combination of features given a certain criterion function [Devijfer et al., 1982] are:

- Branch and Bound
- Max-Min

3.3.1.5. Conclusions on Statistical Pattern Recognition

The complexity of the statistical pattern recognition increases not too rapidly as the complexity of the problem increases. It is possible to keep the number of statistical calculations under control, up to a certain limitation. It is quite impossible to build in fuzzy information. But, one of the biggest advantages is: once a system has learned to recognise some kind of patterns, it is fast in doing so all time and again. The statistical pattern-recognition system is fast, but not flexible. It has to be trained time and again, once it has to recognise a new problem, which has a certain relation with an already known problem.

3.3.2. Syntactical Pattern Recognition

The syntactical pattern recognition has been the answer to the missing possibilities of statistical pattern recognition. It is also called linguistic or structural pattern recognition. All the objects and classes of objects are described formally. The relation between properties and objects is of great importance. A pattern is not seen as an underlying statistical distribution, as it was in statistical pattern recognition, but it is, e.g., seen as a language of primitive objects. The recognition of a pattern is now called parsing.

3.3.3. Hierarchical Pattern Recognition

Up to now, we have seen that the classification of an object belonging to a certain kind of class is done, after the determination of an optimal number of features, by a classification rule (statistical or syntactical). In hierarchical pattern recognition, a decision-tree is built, with in each node a classifier with its own classification rule. In each node, a decision is made whether a object belongs to a certain kind of class. This has a number of advantages:

- Features are used more efficiently
- Feature extraction is made more simply
- Decision rules are less complex per node
- It is possible to use the underlying feature extraction in a better way.

There is, however, a big disadvantage: the optimisation and error-rate determination are quite complex.

3.3.4. Knowledge-Based Pattern Recognition

It can be said that the classification of an object in classes can be done by a knowledge-based system. There are always two levels of recognition: the segmentation and the interpretation. Almost everybody agrees that artificial-intelligence techniques are very well usable in the interpretation phase, especially when it is domain-dependent knowledge. However, it seems that artificial-intelligence techniques also can be used in the segmentation phase. Especially when a more flexible system is preferred above a fast, non-flexible system. [Nazif et al., 1984].

3.4. CONCLUSIONS

Above, we have discussed four kinds of pattern recognition approaches. Each one having its advantages and its disadvantages. For instance, the statistical methods are fast, but not flexible. The syntactical methods are flexible, but their classifiers are slower than the statistical and they become very complex as the problem's complexity increases a little. The hierarchical methods are fast, using all the information, but the classifiers and the error determination are difficult. The knowledge-based approach is just as the syntactical method flexible but slow.

A possible solution to these controversial properties of all the methods can be found in a knowledge-based approach. By combining statistical and syntactical methods, controlled by a knowledge-based system, one can try to use only the advantages of all the methods. For example in optical character recognition, the preprocessing such as filtering, line-thinning and transformations can be done with statistical methods (high speed and few flexibility is required), the classification can be done by a syntactical method (speed is nice, but quality and flexibility is of more importance), with artificial-intelligence techniques (i.e., Spelling Correction) the result can be improved.

In our opinion, it is worth the trouble to investigate the role of artificial intelligence techniques and knowledge based systems in this research area. The disadvantage of a slow classifier can be solved by parallelising the classifier.

CHAPTER 4

OPTICAL CHARACTER RECOGNITION

The Main Issue of this Work

ABSTRACT

In this chapter an overview of known techniques for Optical Character Recognition systems and benchmarks of some commercially available systems will be given. Finally, the philosophy and the structure of our optical character recognition system will be discussed.

4.1. BACKGROUND

Problems related to the replication of human functions are often attractive to researchers. In this case, the recognition of characters (hand- or machine-printed), has been the subject of intensive research for the past two decades. Presently, the state of the art in character recognition has advanced from the use of primitive schemes for the recognition of machine-printed numerals to the application of sophisticated techniques for the recognition of a wide variety of hand-printed characters and symbols. The problem of hand-written character recognition is of great interest for the researchers because even human beings, who have well-trained optical readers, their eyes, make about 4 percent mistakes when reading in the absence of context. Other problems concerning the recognition of hand-written characters are: (i) the techniques must be writer independent, (ii) the algorithms must be robust and yield high accuracies (about 99.9%), (iii) the algorithm must successfully segment connected characters into their individual units and recognise them.

We restrict ourselves to the discussion of machine-printed characters. The special problems which arise in hand-written character recognition, however, are only touched upon.

4.1.1. Historical Background

The first roots of optical character recognition (OCR) can be found in 1870 when Carey invented the retina scanner (this system could transmit a image through a photocell mosaic) and the efforts of Nipkow, who invented in 1890 the sequential scanner, which was a breakthrough for television and reading machines. There were some further attempts in 1900, when a Russian scientist, Tyurin, tried to develop a machine which could help visually handicapped persons. But the first serious OCR systems appeared after the invention of the digital computer, in 1940. The OCR problem was then solved by a typical data-flow approach. David Shepard, the founder of Intelligent Machines Research can be seen as one of the pioneers of the development and building of commercial OCR equipment [Mantas, 1986].

4.1.2. What is Character Recognition?

In practice, character recognition is better known under the term *optical character recognition*, since it deals with optically processed characters rather than with magnetically. In figure 4.1.1, the differences are outlined [Mantas 1986].

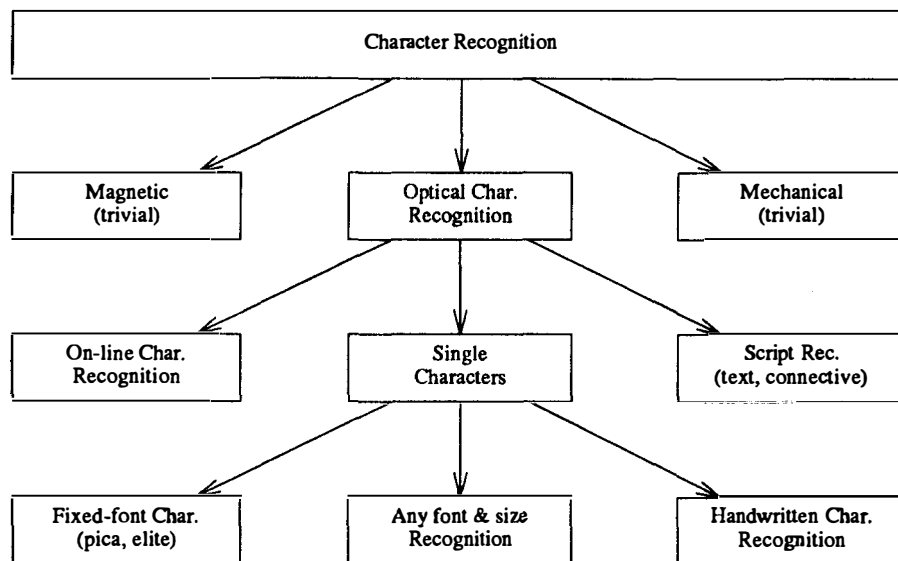


Figure 4.1.1: The Different Areas Covered under the Term "Character Recognition".

The definitions of the major schemes in OCR are as follows:

- a **Fixed-font Character Recognition** is the recognition of specific fonts type-written characters. Once the system is confronted with a different font or font-size, a new library has to be loaded. These systems work already on IBM-AT systems and compatibles. They are quite fast, but very inflexible.
- b **Recognition of any Font and Size** is the recognition of machine-written characters, independent of the font or size of the characters. All the characters are separated by blank space, but italic, roman, bold, etc are all used in the text.
- c **On-line Character Recognition** is the recognition of single hand-drawn characters where not only the character image is provided but also the timing information of each stroke.
- d **Handwritten Character Recognition** is the recognition of single hand-drawn characters of an alphabet which are unconnected and not written in calligraphy.
- e **Script Recognition** is the recognition of unrestricted hand-written characters which may be connected and cursive.

We give an order of difficulty to the different character-recognition problems (starting with the most difficult one): script recognition, hand-written character recognition, on-line character recognition that provide timing information as well as the character image, recognition of any font and size, fixed font character recognition and the trivial tasks of magnetic or mechanical recognition.

4.2. DIFFERENT METHODOLOGIES

The methodologies that are applied in optical character recognition do not differ much from those used in general image pattern recognition. However, because of the particularity of the character-recognition problem, the methodologies could be summarised according to [Mantas, 1986] as follows:

- a **Point by Point Global Comparison.** This process involves a point to point comparison of all the pixels.
- b **Global Transformations.** For example the Karhunen-Loeve, Fourier, Hough, moment calculations and finally rotations according to the principal axis of inertia.
- c **Extraction of the Local Properties.** Such as endpoints, pronounced angles, T-junctions and crossing of strokes. This extraction can be done with various methods, it is often better to thin the character and then apply the above methods for extraction.

- d **Search for the Extraction with Certain Lines.** The lines can either form rectangles or simply be vertical or horizontal.
- e **Analysis by Means of Curvatures.** This involves curve following, detection of concavities and geometrical analysis.
- f **Structural Methods.** These methods decompose a character in constituent elements, they provide topological description and a reduction of the character into a graph.

It must be clear that the different methodologies have some properties in common. In generally, it can be said that a methodology is either statistical or syntactical based. The analysis based on endpoint extraction, transformations, etc. can be seen as a statistical method. The methods based on curvi-linear analysis and, e.g., block-languages can be seen as a syntactical method. In the next section we shall discuss both the syntactical and statistical methods in the three main phases of a optical character recognition system.

4.3. THE OPTICAL CHARACTER RECOGNITION SYSTEM

In order to make clear what an optical reader is in practice, we show figure 4.3.1 [Suen et al., 1980], where a typical optical reader is outlined. Input characters are read and digitized by an optical scanner (see paragraph 4.3.1). Each character is located and segmented from the data sheet or OCR form under software control of the computer.

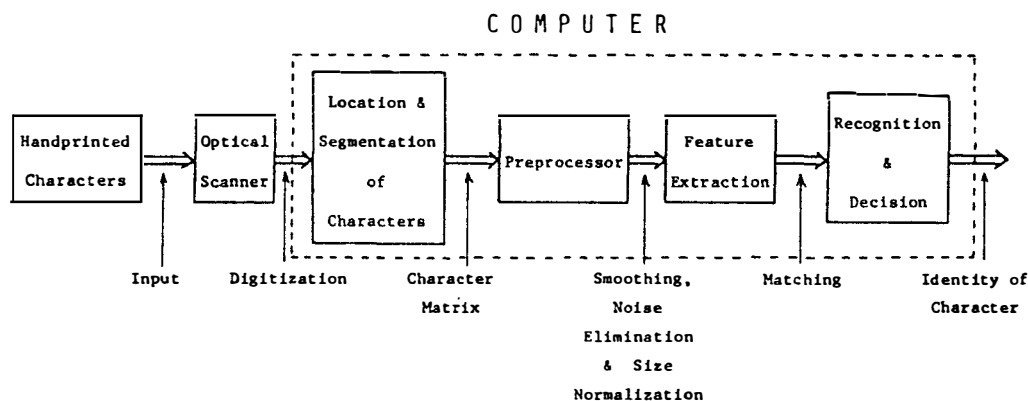


Figure 4.3.1: Block Diagram of a Typical Optical Reader

The resulting matrix is fed into a preprocessor for smoothing (e.g., filling of holes and breaks in line segments), elimination of noise, and size normalisation. Distinctive features are extracted from the preprocessed character for classification. The classification can be done in a statistical or in a syntactical way. The results of this classifier can be optimised by postprocessing techniques (e.g., spelling correction). This can be generalised to the representation shown in figure 4.3.2.

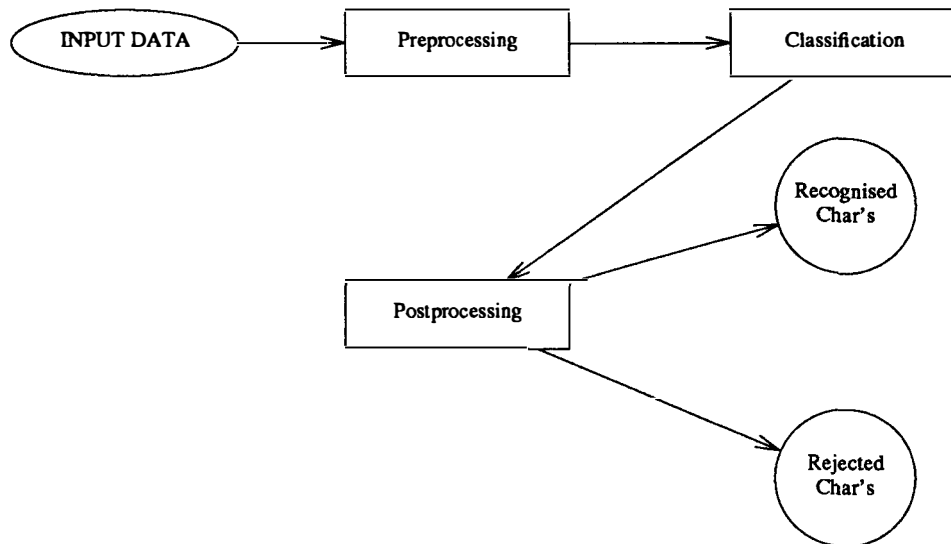


Figure 4.3.2: Generalised Optical Character-Recognition System

There exist several different methods for each phase of the above picture. In the next part of this chapter, a brief introduction to the most significant methods will be given.

4.3.1. Preprocessing

In Optical Character Recognition, preprocessing is used to reduce the mostly enormous amount of input data into relevant data and to transform the input data into a representation more understandable for a computer. The first phase consists mostly of the following subphases:

- **Scanning of the Paper.** This is the digitizing part. The characters on a piece of paper are converted to a large matrix with pixels, which are either 0 (white pixels) or 1 (black pixels). There are more sophisticated systems which convert a point on the paper into a vector with values for the intensity, contrast, etc.
- **Pixel-Based Line Thinning.** In most cases, a character contains a large amount of black pixels. The largest part of those pixels are not relevant, so they can be deleted. The biggest problem in this sub-phase is to delete the right pixels to reduce the amount of active pixels, without destroying any essential information for the classification of the character. The line-thinning methods are either pixel- or graph-based.
- **Identification of Line or Curve Segments.** In this phase an extra level of abstraction can be brought into the classification process. In the case of a structural classification method, it is better to work for example with lines and circles instead of a random-like number of active pixels. The best-known method for curve and line detection is the so called Hough transform [Hough, 1962].
- **Decomposition Techniques.** Decomposition techniques are used to detect contours in images. In Low Level Image Segmentation a normally statistical based method is used because of the large amount of data to be processed. However, it is also possible to employ AI-based techniques in this phase. For instance, an expert system, which has general knowledge about segmentation on a low level [Nazif et al, 1984].

- **Combination of Line Thinning, Identification and Decomposition Techniques.** These methods are based on the production of lines and curves by vectorization of text pages. The generated lines and curves should meet the following two objectives: (a) they are close to the original lines and curves, (b) they are useful as features for the recognition of characters. A possible algorithm may then provide information about strokes, holes, concave, arcs, crossings, and endpoints. In [Pavlidis, 1986] a graph-based algorithm is described : Line Adjacency Graph (LAG). This method is too complex and too specialistic to present in this context.

In the following part the most important techniques will be discussed. This involves different types of scanners, Pixel-Based Line Thinning , Rule-Based Decomposition and the Hough transform.

4.3.1.1. Scanners

The first stage of any Optical Character Recognition equipment involves the scanning or digitization of the printed characters into a form suitable for subsequent electronic processing with minimum further degradation. The most common scanning methods [Mantas, 1986] are:

- Divided-Slit Scan.** In this method, a vertical column of photo-electric devices can be used as a divided-slit scan to obtain information about the detailed vertical as well as horizontal information of the characters scanned.
- Mechanical Scanner.** This is also known as the Nipkow disc: a rotating disc with apertures is used in order to pass through the reflected light from the scanned character onto a measuring photocell.
- Laser-Beam Scanner.** This is one of the most current models, which uses-laser beam properties which are coherently monochromatic for extremely high definition scans.
- Flying-Spot Scanner.** This kind of scanner makes use of a cathode ray tube (CRT) and can readily be used to explore the format of a document to locate separate lines of print without any document movement.
- Vidicon Scanner.** The vidicon tube has been commercially developed for television cameras, hence it is available at a reasonable cost for the recognition systems. They are used to scan a single vertical line.
- Photocell-Matrix Scanner.** This scanner retains the two-dimensional image of the whole character by means of a two-dimensional matrix of photocells.

The first five methods provide us with a vertical scan of the image, while the horizontal scanning is provided by a horizontal movement of the paper. The divided-slit scan, the laser-beam scanner and the photocell-matrix scanner are the fastest. The speed can be around 3000 characters a second. In research environments, the scanners mostly provide the system with one character at the time in order to do better tests on large databases. Those systems are connected with a on-line graphics tablet. Recently the prizes of commercially available scanners are falling fast. These days, one can buy a good all-round scanner for about 1000.- US Dollar.

The result of the scanning process is for example an array containing 1024 by 512 elements, each element representing a pixel on the paper. The elements can be either 0/1 or a vector of measurements.

4.3.1.2. Pixel-Based Line Thinning

The information concerning this subject is based upon [Harris et al., 1982]. In this part, a system is discussed which has been set up to process the binary pixel data derived from a raster scan of a line draw map and to generate the vector representation of each polygon boundary. The most important part of the system is the raster to vector conversion process (where the vector is a list of pixels). The aim is, as described above, to convert a binary pixel representation of a network into a connected set of segments and nodes. (A segment is an ordered string of coordinate pairs joining two nodes, but not including the node coordinate points; a node is either a point at which three or more segments meet or a point at which a segment terminates). The segments and nodes are connected in the sense that each segment knows the nodes at its two ends, and each node knows the segments that are attached to it and their order around it.

The requirements for a raster to vector algorithm are that (i) resolution of the scan must be high enough, with the result that thinning must be performed. Other requirements are: (ii) it should represent the nodes well; (iii) it should generate a accurate centre line; (iv) it should preserve the connectivity of the original

network; and (v) it should not significantly shorten freely terminating lines.

The requirements for the preservation of the connectivity of the original networks and for the faithful representation of the centers of the lines suggests the use of topological skeletonising. This is also called connected line thinning, as its objective is to derive a centre line of the pixel data and not a true skeleton.

In the method discussed here, a skeleton is generated which is used for the later derivation of a final list of connected nodes and segments. The skeleton has a temporary existence and the net result is to read the image and output the nodes and segments in one serial pass through the image.

The Algorithm

The algorithm uses a set of three by three operators which, rather than just resetting the black pixels to white or leaving them as black pixels, generate a marked skeleton. In this scheme each pixel is left with one of the four possible labels:

Type	Meaning
0	deleted
1	segment end point
2	segment internal point
3	node point

Initially all the black points are labelled as type 3 and the white pixels are labelled as type 0. Then during the thinning process every type 3 pixel encountered is relabelled according to the following rules.

Type	Rule for Relabelling
0	if the pixel has two or more neighbours and erasing does not alter the connectivity.
0	if the pixel has one neighbour of type 3
1	if the pixel has one neighbour of type 2
1	if the pixel has no neighbours
2	if the pixel has exactly two black neighbours which are not neighbours of each other.
3	if the pixel has more than two black neighbours and erasing it would alter the connectivity.

When the application of the operators has iterated to completion, the network is left as unit-thickness eight-connected strings of type 2 pixels with free ends labelled as type 1. Nodes in the network are represented either as single type 3 pixels or, more frequently, as local groups of type 3 pixels.

The next phase is to delete all the type 3 pixels from the data set using node and segment extraction and to output into the node list a single average point for each local group of pixels. Adjacent type 2 pixels are reset to type 1, and their positions are recorded with the node in a temporary node list. By the time the segment extraction takes place only type 1 and type 2 pixels are left. By construction, these will be connected runs of type 2 pixels terminating at each end with type 1, so it is possible to follow each segment from raster line to raster line. The fact that the segment end point has been recorded in the temporary node list allows the final node and segment lists which are written out to be properly cross-referenced. This process can be seen in the figures 4.3.3a to 4.3.3d.

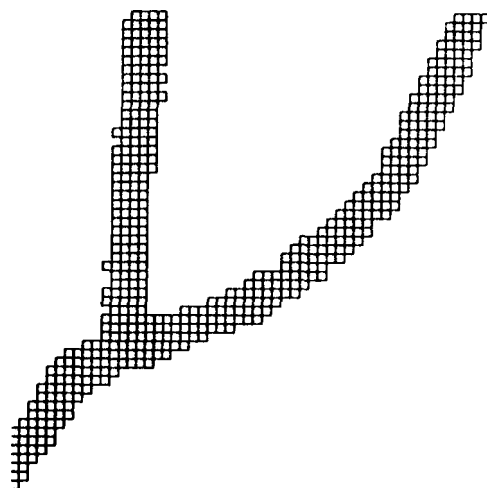


Figure 4.3.3a: Raster Input

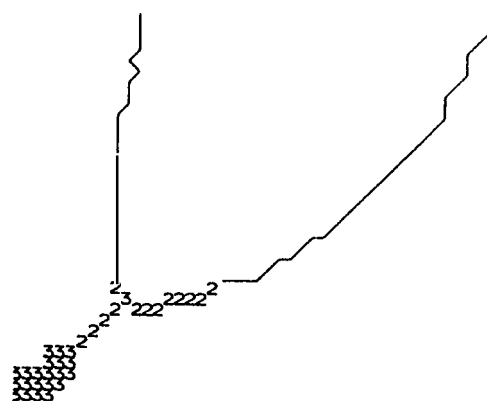


Figure 4.3.3b: The Data at an Intermediate Phase

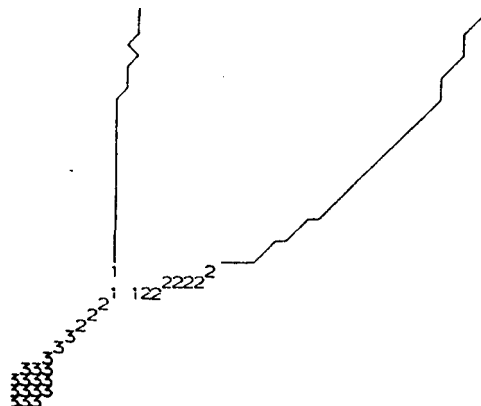


Figure 4.3.3c: The Data after the Nodes have been Extracted

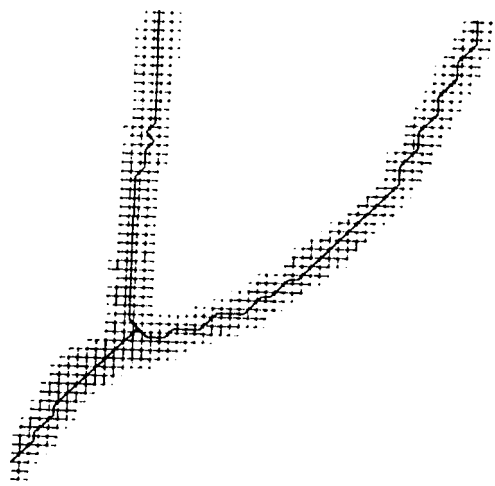


Figure 4.3.3d: The Vector Output Superimposed on the Raster Input

There are, however, some problems with the algorithm; it is namely quite noise sensitive. One way to solve this problem is to preprocess the data pixels. In figure 4.3.4a and 4.3.4b a noisy input raster and its corresponding noisy vector output can be seen. The output quality increases when the data raster is preprocessed with the following two techniques. The output is shown in figure 4.3.5a and 4.3.5b:

- First, all the so-called spike pixels are set to white, where a spike pixel is a black pixel with three and only three black neighbours which lie in the same vertical or horizontal line. The spike pixels are the reason of the so-called hairs shown in figure 4.3.5b. It is known *a priori* that there are no hairs in the input data for the present object.

- The second operation is to set white all the black pixels which have four or more eight-connected neighbours. This operation differs from the first one in that it is divergent. that successive applications, in general, continue indefinitely to find pixels to change.

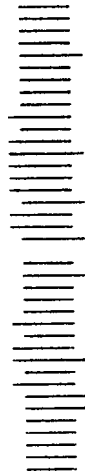


Figure 4.3.4a: Noisy Raster Data



Figure 4.3.4b: Resulting Noisy Vector Output



Figure 4.3.5a: Preprocessed Raster Data



Figure 4.3.5b: Resulting Smooth Vector Output

4.3.1.2.1. Identification of Line and Curve Segments: the Hough Transform

The best known method to detect lines and curves in a pixel based data space is the Hough Transform. The classical Hough technique is applicable if little is known about the location of a boundary, but its shape can be described as a line, circle or another parametric curve. Its main advantage is that it is relatively unaffected by gaps in curves and by noise. The method has been developed in 1962 by P.V.C. Hough, who patented it in the same year. [Hough, 1962] [Ballard et al., 1982] [Duda et al., 1972] [Ballard, 1981] [Kimme et al., 1975] [Kushnir et al., 1985].

Consider the problem of detecting straight lines in images, consisting of a raster of input pixels. Assume that a number of image points have been selected and have a likelihood of being boundaries. (In the problem concerning optical-character recognition, all the input pixels can be boundaries). The Hough transform then organises these points into straight lines, basically by considering all possible straight lines at once and rating each on how well it explains the data. Consider the point (x', y') in figure 4.3.6a and the equation $y = mx + c$. What are the lines that could pass through (x', y') ? The answer is all the lines with m and c satisfying $y' = mx' + c$. Regarding (x', y') as fixed, the last equation is that of a line in $m \times c$ space, or parameter space. Repeating this, a second point (x'', y'') will also have a associated line in the parameter space, and furthermore these lines will intersect at the point (m', c') which corresponds to the line AB connecting (x', y') and (x'', y'') . In fact, all points on the line AB will yield lines in parameter space which intersect at the point (m', c') , as shown in figure 4.3.6b.

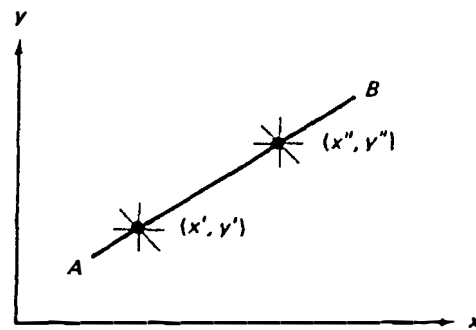


Figure 4.3.6a: A Straight Line in Image Space

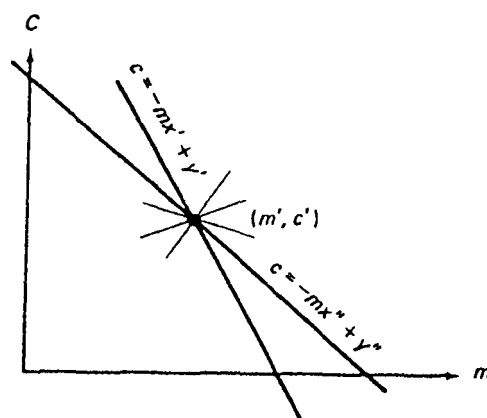


Figure 4.3.6b: A Straight Line in Parameter Space

This relation between image space and parameter space suggest the following algorithm for detecting lines:

Line Detection with the Hough transform	
Step	Description
a	Quantize parameter space between appropriate maximum and minimum values for c and m .
b	Form an accumulator array $A(c, m)$ whose elements are initially zero.
c	For each point (x, y) in a gradient image such that the strength of the gradient exceeds some threshold all points in the accumulator along the appropriate line, i.e., $A(c, m) := A(c, m) + 1$ for m and c satisfying $c = -mx + y$ within the limits of digitisation.
d	Local maxima in the accumulator array now corresponds to collinear points in the image array. The values of the accumulator array provide an exact measure of the number of points on the line.

This technique is generally known as the Hough Transform. Since m may be infinite in the slope-intercept equation, a better parametrization of the line is $x \cos \theta + y \sin \theta = r$. This produces a sinusoidal curve in (r, θ) space for a fixed x, y , but the rest of the procedure remains unchanged.

The generalisation of this technique to other curves is straightforward and this method works for any curve $f(x, a) = 0$ where a is a parameter vector. For example, we can parametrize with a circle:

$$(x - a)^2 + (y - b)^2 = r^2$$

Unfortunately, the computation and the size of the accumulator increases exponentially with the numbers of parameters, making this technique only practical for curves with a small number of parameters.

The Hough Transform is an efficient implementation of a generalised matched filtering strategy.

4.3.1.3. Decomposition Techniques

In this section, we describe the problem of low level image segmentation based upon [Nazif et al., 1984]. Normally one starts thinking of the use of expert systems in high level segment interpretation, not in low level segmentation, because of the great amount of data that should be processed. There are, however, a number of reasons, why also low level image segmentation could be done with the use of expert systems:

- (a) There are a large number of heuristics for segmentation.
- (b) There is no explicit scheme for the representation of this knowledge, an expert system could be of help.
- (c) There are no quantitative evaluations of a segmentation.
- (d) It could be said that also in pattern recognition there is an increasing need for better understanding of the methods by which the segmentation heuristics should be applied to an image.

These four reasons made Nazif and Levine decide to build an expert system for low level segmentation. The expert system is rule based. The system is represented in figure 4.3.7.

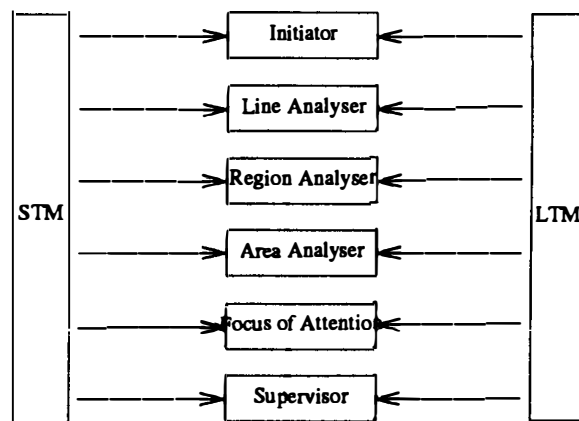


Figure 4.3.7: The System Block Diagram

The image is stored in the Short Term Memory (STM); the model representing the system knowledge about low level segmentation is stored in the Long Term memory (LTM). The data is modified and transferred between the two memories by the processes which are indicated in between. The Initiator generates an initial region and line maps, and uses them to produce focus of attention areas. The Region, Line and Area Analysers are the three main processing modules of the system. They are responsible for the rule matching and the data modification. The Focus-of-Attention module matches its own rules to execute a defined path strategy that brings about various data items to the attention of the system, in a particular order. The current line, region or area will be replaced by a new line, region or area, whose identity depends on the action specified by that rule, when the Focus of Attention action is executed. Finally the Supervisor acts as a monitor for system-control purposes. It matches the strategy rules to select the appropriate control strategy. Consequently, a set of meta rules is used to determine which set of knowledge rules, if any, is to be tested next. The hierarchical structure between the rules is shown in figure 4.3.8.

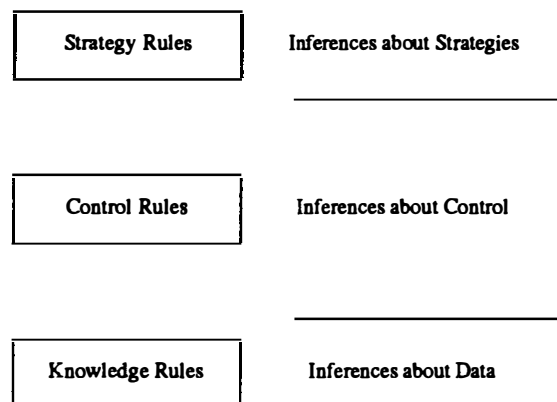


Figure 4.3.8: Three Levels of Rules.

The global structure of the Region Analysis Rules is for example [Nazif et al., 1984]:

RULE (803):

- IF: (1) The REGION SIZE is LOW
 (2) The REGION AVERAGE GRADIENT is HIGH
 (3) The DIFFERENCE in REGION FEATURE 1 is NOT HIGH
 (4) The DIFFERENCE in REGION FEATURE 2 is NOT HIGH
 (5) The DIFFERENCE in REGION FEATURE 3 is NOT HIGH

THEN: (1) MERGE the two REGIONS

RULE (804):

- IF: (1) The REGION SIZE is LOW
 (2) The REGION ASPECT RATIO is VERY HIGH
 (3) The DIFFERENCE in REGION FEATURE 1 is NOT HIGH
 (4) The DIFFERENCE in REGION FEATURE 2 is NOT HIGH
 (5) The DIFFERENCE in REGION FEATURE 3 is NOT HIGH

THEN: (1) MERGE the two REGIONS

A control rules reads as follows: RULE (201):

- IF: (1) The LINE GRADIENT is HIGH
 (2) The LINE LENGTH is HIGH
 (3) SAME REGION LEFT and RIGHT of the LINE

THEN: (1) GET the REGION to the LEFT of the LINE

A Metarule is for instance:

METARULE (6):

- IF: (1) Previous PROCESS was FOCUS
 (2) Previous PROCESS was NOT ACTIVE
 (3) Next previous PROCESS was FOCUS
 (4) Next previous PROCESS was NOT ACTIVE

THEN: (1) STOP

The biggest advantage made with this method was the possibility of domain-independent segmentation. Moreover, a better insight in the possibilities of heuristics and control strategies was found. Unfortunately the system processes 12 hours on a simple picture.

4.3.2. Classification

As described in the previous chapter, Pattern Recognition can be regarded as Pattern Classification. The most important problems are solved in the classification phase. Historically, Pattern Classifiers, used to work with Statistical Methods, because of their speed. Lateron, researchers became aware that syntactical Pattern Classification could provide us with a more flexible system. Modern technology made it possible to overcome the low-speed of the syntactical systems. The most important properties of a statistical Pattern Classifier can be read in the chapter: "Pattern Recognition, A Brief Introduction". Considering a syntactical classifier, we can say that the most important problems are:

- **How to Represent the Knowledge.** A data structure should be developed to represent the knowledge of the characters, words, sentences, etc. In [Minsky, 1975] a data structure is presented called **Frames**.
- **How to Describe Characters.** The Characters should be represented in a way that could be understood by the computer as well as by the system developer. The most important properties of an adequate representation should be (i) the system developer could bring abstractions into the representation and (ii) the description of a character is independent of the font, size or type (e.g., Bold and Curative). A possible method is to describe characters with **Skeletons**. [Cox, et al, 1982].
- **How to Train the System.** Once we have a data structure and a way to describe a character, we should develop a method to train the system. This training can best be done by examples concerning the application where it is used for. We can, for instance, make frequency tables for trigram (combinations of three letters appearing in words) distributions for different types of text (UNIX Manual and ICCA Journals). Depending on the type of input text, one can use a type of frequency tables for character-search optimisation . Spelling Correction is another possibility. In [Winston, 1975] a number of variations on the method: "Learning from Examples" is presented.

In the next sections we describe frames, skeletons and trigram frequency tables for search optimisation.

4.3.2.1. Frames

Frames [Minsky, 1975] [Winston, 1984] [Abelson et al., 1985] [Schneider, 1987] are structures used to represent general classes of objects. In fact they can be used as templates to structure information. Frames have so-called slots that describe and structure the properties associated with the class of objects represented by the frame (see figure 4.3.9).

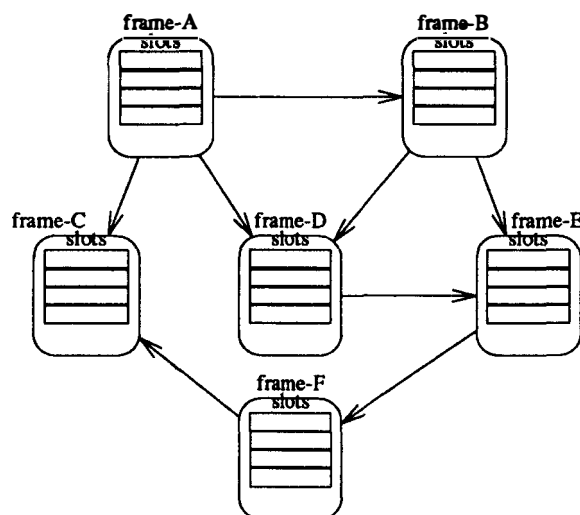


Figure 4.3.9: Example of a Frame Structure

The frame concept is based on the semantic network theory [Winston, 1984] and as such it has inherited many mechanisms that concern these networks. In fact, frames are *part* of the semantic network theory. Three main goals of frames are: (i) guide the description transformation from class description to individual description, (ii) determining descriptions in the absence of specific knowledge, and (iii) specifying procedures that enable descriptions to be domain dependent.

A summary of the basic mechanisms is given below :

- **Inheritance**, enabling description transformation from classes to instances. Every frame has one or more relational properties that can be typed as A-KIND-OF relations connecting frames with their superframes. For example, a particular wedge is triangular because all wedges are triangular.
- **If-Modified demons**, enabling the storage of functional or procedural information describing a procedure or computation that is activated when the property of an instance is modified.
- **Default values**, that are used to store *a priori* knowledge in a frame description so that it may be inherited by other instances or subframes of that frame.

Inheritance is common to all frames and may be conditioned to maintain context dependencies. However, demons and default values are optional and are applied to specific class properties. In frame terminology class properties are called *slots* that have a certain *value*. Demons and defaults are properties of a slot and are called *facets*. Based on these basic facets a list of other useful facets is given below :

- Constraints
- Multi-valued
- Documentation
- Certainties
- Value

As a whole, frames can be viewed as a structure consisting of name, slots, facets and values as depicted in figure 4.3.10.

```

(<frame-name>      (<slot1>   (<facet1>  <value1>)
                                     <value2>
                                     ...etc)
                                     (<facet2>  <value1>)))
                                (<slot2>   (<facet2>  <.....>)))
  
```

Figure 4.3.10 : Frame structure

Winston and Horn [1984], have suggested a Lisp model for implementing frames in their book [Lisp, 1984]. Assuming the structure for frames given in figure 4.3.9, they have implemented the following procedures :

- **FGET** fetches information. The user supplies an access path consisting of a frame, a slot, and a facet.
- **FPUT** places information. As with FGET, the user supplies a frame-slot-facet access path.
- **FREMOVE** removes information. As with FGET and FPUT, the user supplies a frame-slot-facet access path.
- **FGET-V-D** fetches information. The user supplies an access path consisting only of a frame and a slot. The VALUE facet is inspected first. If nothing is found, then the DEFAULT facet is also inspected.
- **FGET-I** fetches information. The user supplies a frame-slot access path. If nothing is found in the VALUE facet, then the procedure looks in frames found in the given frame's A-KIND-OF slot, under their VALUE facet. This enables a frame to inherit information from frames it is related to by the A-KIND-OF relation.

- **FGET-N** and **FGET-Z** fetch information. They use values, defaults and value-finding procedures, not only as found in the given frame but also in the frames it is related to by the A-KIND-OF relation (slot).
- **FPUT-P** and **FREMOVE-P** activate demons. The description of the procedure consists of a list with Lispfunction calls that perform the necessary task if the value of a slot is modified.

The FGET, FPUT and FREMOVE can be considered as the basic Frame-handling procedures.

Note

In the early days, the relevance of frames has only been acknowledged in the academic world. Companies were mainly concerned with data storage in hierarchical databases and more recently relational databases using Cobol as a main application language.

Only now when they are aware of the difficult maintainability of relational databases as well as the unavoidable redundant storage of key information which is inherent to relational concepts they are considering the use of semantic databases. In our opinion semantic databases are closely related to frame networks and the transformation of a frame network to a semantic database is trivial as it is vice versa.

Very few frame-based systems are commercially available and those which are, are not meant for massive data storage but rather for the implementation of artificial-intelligence systems, so called expert systems. The storage of knowledge in frames is very attractive because it is possible to replace the rules in expert systems that use an inference engine and database to come to conclusions. The frame structure reduces the search in the rule base and inference process to a matching process, that matches a query against the frame structure containing the transformed rules. An example is given in figure 4.3.11.

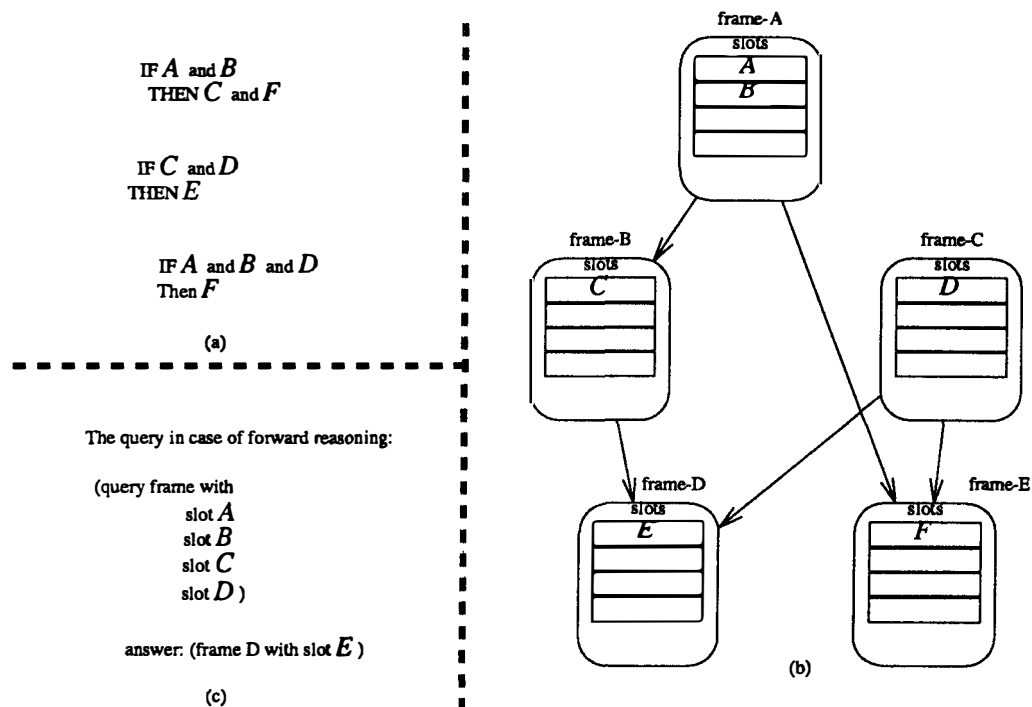


Figure 4.3.11: Example of Rules Versus Frames.(a) A Rule Base
(b) Corresponding Frame-structure, and (c) the Query.

4.3.2.2. Skeletons

Describing the shapes of alphabetic characters has been interesting for two main reasons [Cox et al., 1982]:

- (1) Researchers who wanted to make a multi-font, automatic character-recognition system, need a description of characters from which they can deduce the appropriate measurements to be made on characters in order to identify them.
- (2) Workers interested in character generation or type design, particularly when a computer is to be used in the process, need a description of characters to facilitate not only the storage of large numbers of type fonts. A description is required that separates aspects of letterness from aspects of embellishment. Where does a description of a letter leave off and a description of embellishments begin.

There are many paths one could construct between the abstract description of a letter and its physical representation as a character. A pure linguistic path is described in this part. The path consists of three parts. One part concerns the abstract description of letters in terms of elementary parts called functional attributes. These attributes can be found by investigating ambiguous characters, that is, characters whose identity is equally assigned one or two letter labels by human observers. This is the analytical approach. The second part consists out of the investigation of grammatical description of characters and the stylistic consistency among them in terms of strokes and serifs. This is a more synthetic approach. There is however a problem to both approaches; where does the description of a letter end and where starts the description of an embellishment. The latter question can be replaced by the question : "What is the formal relationship between a letter and a character?" This question can be solved by developing a new formal structure, called a skeleton between the letter and the character description. Its main goals will provide a means for incorporating: (1) Letter versus nonletter constraints of the letter description and (2) inter-character constraints on the style of the characters of an alphabet. We can say that:

- **Letter** : Abstract description in terms of functional attributes. To distinguish letters from letters.
- **Skeleton** : Semi-abstract, semi-physical description in terms of vertices spatial ordering and vertex/edge relations. To distinguish letters from non-letters.
- **Characters** : Physical description in terms of strokes and serifs. To distinguish between styles of letters.

In figure 4.3.12 the skeleton notation can be found. Please note the difference between links, meets and joins; in links one edge touches another somewhere in the middle, meets are defined as two edges which should touch each other at the end, and joins are two edges where the end of the edges must lay in the same region. In figure 4.3.13 the roman font can be found, in figure 4.3.14 the skeleton representation for this font is given [Cox et al., 1982].

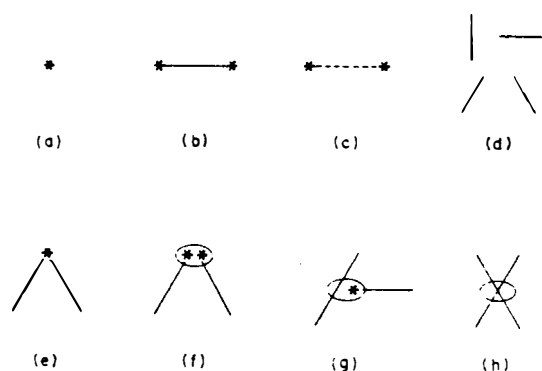


Figure 4.3.12: The Skeleton Notation: (a) Vertex, (b) Explicit Edge, (c) Implicit Edge, (d) Four-edge Orientation, (e) Meet, (f) Join, (g) Link, and (h) Cross.

ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890

Figure 4.3.13: The Roman Font.

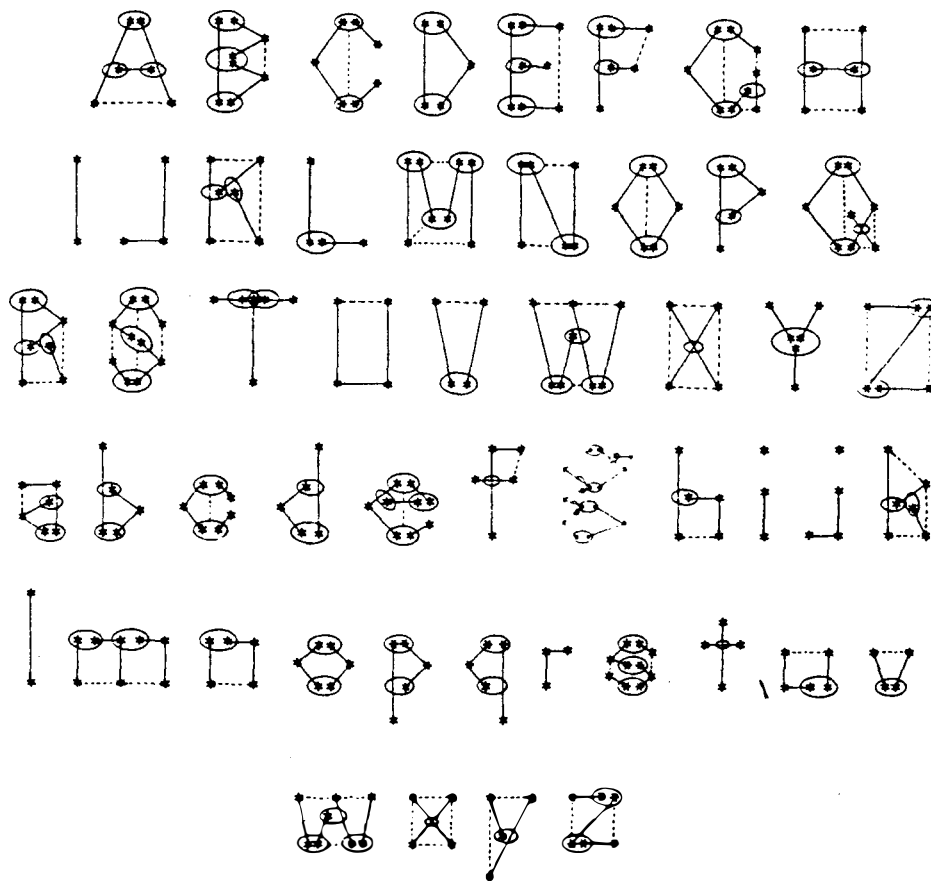


Figure 4.3.14: Skeletons for the Roman Font.

With skeletons it is possible to describe diverse characters in an unambiguous, high level manner. The process from character to skeleton is completely rule based and unique. This method could provide us with a good technique usable in Optical Character Recognition.

4.3.2.3. Trigram Analysis for Search Optimisation

Assume we have a character set of 80 characters; lower- and upper-case alphabet, numerals, brackets and read markers. Then, each time we want to recognise a character we have at least one and at most eighty checks to make (the average is forty checks a character). In case of a well-filled page, about

$40 * 3000 = 120,000$ checks are to be made. Knowing that the checks are based on symbol comparison in a LISP environment, the time needed is too long for a well-working system. There is a need for search optimisation.

Knowing that the context we are working in is, the context of the real-world words, we can make a histogram of all the trigram occurrences in the text. For instance, the *a priori* probability that a *q* will be followed by another *q* is nihil in English. The *a priori* probability for a *h* to follow a *t* is very high (*the* is the most often used word in English texts). Those *a priori* probabilities are, however, depending on the context. In Russian the combination *zz* will probably occur much more frequently than in English or in French. Even in the English language there are domain dependencies: the difference between a UNIX-Manual and Computer-Chess Related Texts (We used the International Computer Chess Association Journal texts: ICCA Journal). The combination *ply*, a term connected to the search depth in computer chess, occurs much more in ICCA Journal tables than in the UNIX-Manual tables. Even better frequencies are found in the succeeding of three letters: trigram analysis [Angell, 1983] [Zamora, 1981].

A trigram is a combination of three following characters in a word, including the spaces. The word *spelling* has the following trigrams: —s, —sp, spe, pel, ell, lli, lin, ing, ng—, g—.

In the beginning of the recognition of a word, searching is started for the characters which occur mostly at the beginning of words. When the first character is found, the character with the biggest *a priori* probability to follow the first is first to be checked, then the next, etc. Depending on the first two characters, the search is continued for the third. Using this method we can decrease search time a great deal. In table 4.3.1 and 4.3.2 a number of trigrams is shown for UNIX-Manual frequencies and ICCA-Journal frequencies respectively. †

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
aba	297	afr	2	akw	5	anv	3	ast	434	axe	6
abe	9	afs	14	ak-	39	anw	11	asu	155	axi	154
abf	14	aft	188	ala	75	any	286	asv	4	ax-	28
abi	58	afw	1	alb	39	an-	2083	asy	24	aya	6
abl	790	afz	1	alc	13	aoi	1	as-	1621	ayb	2
abo	215	af-	8	ald	8	ao-	3	ata	569	aye	141
abr	6	aga	162	ale	467	apa	100	atc	68	ayi	61
abs	100	agd	2	alf	34	ape	273	ate	1931	aym	5
abu	21	age	455	alg	130	aph	75	atg	4	ays	81
aby	2	agi	15	ali	285	api	40	ath	189	ay-	745
ab-	5	agl	1	alk	5	apl	171	ati	2142	aza	14
aca	1	agn	16	alm	24	apo	19	atk	57	aze	1
ace	284	ago	12	alo	145	apr	184	atl	18	azi	17
ach	645	agr	129	alp	26	aps	39	atm	8	az-	1
aci	61	ags	4	alr	34	apt	114	atn	8	a--	5039
ack	722	agu	5	als	319	apu	9	ato	98	bab	24
acl	14	agy	2	alt	94	apy	4	atr	75	bac	135
acm	17	ag-	42	alu	314	ap-	68	ats	24	bad	32
aco	93	ahe	9	alv	35	aqu	2	atu	151	bag	1
acq	11	ah-	1	alw	30	ara	446	aty	12	bak	7

Table 4.3.1: Some ICCA Results

† The frequencies are calculated on a GOULD Mainframe at Westmount Technology B.V., Poortweg 4, 2612 PA, P.O. Box 5063, Delft. Email gen@wmt.uucp. The texts from both the ICCA Journal and the UNIX Manual were resident on the system. With the UNIX utility DEROFF -W the words were unformatted and put in a file with one word on a line. With the UNIX utility TR the upper case letters were translated in lower case. A self-developed C program counted the frequencies. We did further analysis with the MS-DOS compatible PC programs Symphony and Reflex.

TRIGRAM FREQUENCY TABLES FOR UNIX-MANUAL											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
aba	3	age	367	amd	1	arl	17	awn	5	bew	3
abc	4	agi	18	ame	1566	arm	11	aws	6	bex	2
abe	35	agl	2	amh	8	arn	18	aw-	10	bey	4
abi	10	agn	38	ami	61	aro	17	axb	3	be-	921
abl	483	agr	27	aml	2	arp	8	axh	1	bfa	1
abm	1	ags	74	amo	34	ars	37	axi	41	bfb	1
abn	6	agt	1	amp	111	art	179	axn	4	bfi	6
abo	81	ag-	33	ams	55	ary	129	axp	2	bfr	1
abr	11	aha	1	amw	1	ar-	313	axr	2	bft	1
abs	59	ahe	1	am-	144	asc	43	axu	1	bg-	4
aby	1	ah-	4	ana	18	ase	281	ax-	202	bib	17
ab-	27	aid	2	anc	41	ash	63	aye	9	bic	1
acd	2	aig	1	and	2335	asi	59	ayl	2	bid	7
ace	582	ail	262	ane	37	ask	165	ayo	1	bif	12
ach	288	aim	10	ang	279	asm	7	ays	65	big	12
aci	44	ain	410	ani	57	aso	68	ay-	427	bih	1
ack	244	air	36	ank	54	ast	118	a--	343	bil	25
acm	1	ais	5	anl	1	asu	21	bab	2	bin	83
aco	44	ait	63	anm	4	asv	1	bac	98	bir	2
acq	1	aiv	1	ano	81	asw	23	bad	42	bit	176
acr	56	ai-	2	anp	1	asx	1	bal	43	biw	1
act	578	ajf	2	anr	1	asy	24	ban	13	bi-	1
acu	4	ajo	3	ans	146	as-	791	bar	7	bje	84

Table 3.2: Some UNIX Results

4.3.3. Postprocessing

In a number of recently developed Optical Character-Recognition Systems, the postprocessing part uses techniques to increase the recognition performance of the system. Often the performance can be increased by 10 percent using spelling-correction techniques. This can be done in several ways:

- Check the words through a complete word list .
- Check the words through a domain-dependent word list .
- Check the words using trigram analysis.

The biggest problem in using spelling correction is the enormous amount of memory occupied by a complete or even a domain-dependent word list. A second problem is the time needed to check the entire word list on the existence of a word. These problems can be solved by either a clever data structure or the use of only trigrams.

In [McIlroy, 1982] a method is presented to set up a complete word list . The word list was reduced to one third of the original size by stripping all the pre- and suffixes. Hashing discards 60 percent of the bits to remain, and data compression halves it once again. McIlroy was able to squeeze 30,000 English words into 26,000 16-bit machine words.

It should be mentioned that McIlroy faced the problem of only having 2^{15} 16-bit words at his disposal. Nowadays, there is no need for reducing dictionary sizes because of memory problems. However, the reduction McIlroy has applied also meant a great speed-up of the entire system and should therefore not be considered trivial.

4.4. MAJOR RESEARCH PROJECTS AND THE STATE OF THE ART OF OCR

To give some more insight into the specific problems of optical character recognition and to complete the relation with the chapter on pattern recognition we shall discuss some research projects on OCR. On each project we will discuss the preprocessing, the classifier, and postprocessing methods.

4.4.1. Bell Laboratory's Multi-font, Multi-size O.C.R.

At Bell laboratories, an Optical Character Recognition System is developed which can recognise characters of any font and any size [Kahan et al., 1987]. This was the first serious attempt to develop such a system. In the past, OCR Systems were developed which could recognise about 99% of the characters (even on IBM AT Systems). But as the font and the size changes, the recognition rate falls dramatically. Considering that on a full page about 3000 characters are printed, 95% recognition means 150 errors, that is about 3 per line. A minimal performance for a practical system is 99,9% correct recognition with all errors being rejections. Then we would have about three rejection per page, which is the number of errors expected from manual entry.

The system of Bell laboratories is implemented on a VAX 11/750 under UNIX. The papers are scanned by a 200 dot per inch AT&T Scanner. This device scans a page in 5 seconds. The system's overall processing speed is about 5 characters a second.

The system recognises with an accuracy of 97% on mixtures of six dissimilar fonts across a wide range of point sizes, using a single classifier with one class per character type. It was necessary to mix methodologies, and to use context information at different kind of degrees at different stages.

The Methodology

No effort is made to detect lines in advance; the program starts to detect and to recognise blobs, and then proceeds with line information. From a scanned page a Line Adjacency Graph (LAG) is constructed. The LAG has nodes that correspond to dark segments and branches that join nodes whose corresponding segments touch. Blobs are found as connected components of the LAG. The LAG is used for line thinning, which traverses the contour, and extracts various features of the characters. The LAG thinning (see figure 4.4.1) requires under 3 minutes per page of text versus 20 minutes for pixel-based thinning.

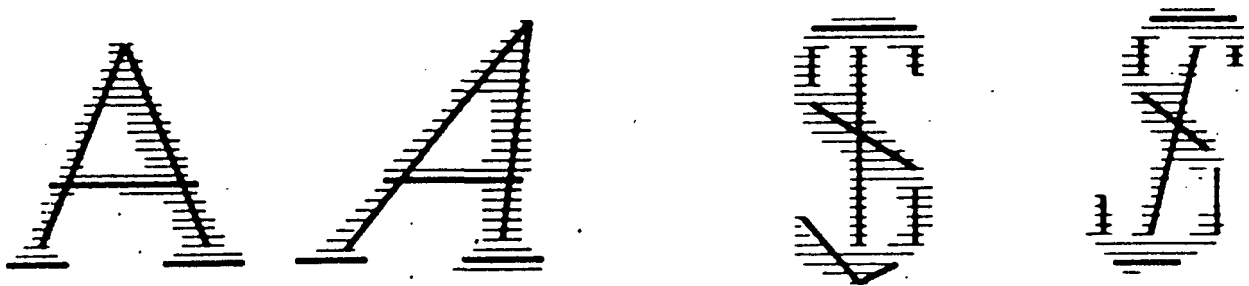


Figure 4.4.1: Examples of Thinning. The Horizontal Lines Represent the Segments at Run Time Encoding and the Heavy Lines the Strokes Extracted by the Thinning Algorithm.

Contour analysis is known as a time-consuming technique, therefore this method which provide only incidently new information, was used when a letter belongs to a confusion group. Confusion groups are:

- a, e, s, g, 8, B (having two holes)
- O, o, 0, D, Q
- 6, b
- b, h
- f, t
- [,], (,), 1, l, i

Another problem is the separation of merged characters; this can be done by contour analysis. At Bell laboratories this problem was solved by breaking the characters in different parts. The breaking decision was made with a vertical projection-based criterion function (see figure 4.4.2 and 4.4.3).

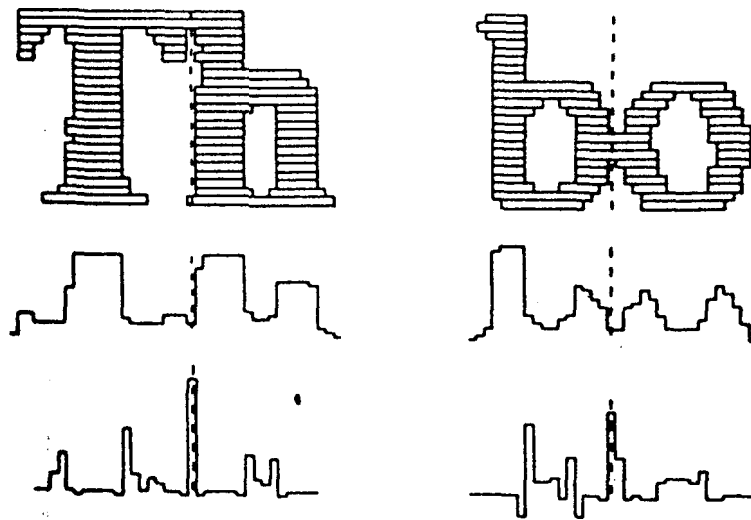


Figure 4.4.2: Illustration of Merged Characters and the Functions that are Computed for Breaking them.
 Top: A Typical Serif Join and a Typical Double-o Join.
 Middle: Vertical Projection.
 Bottom: Breakpoint Criterion Function.

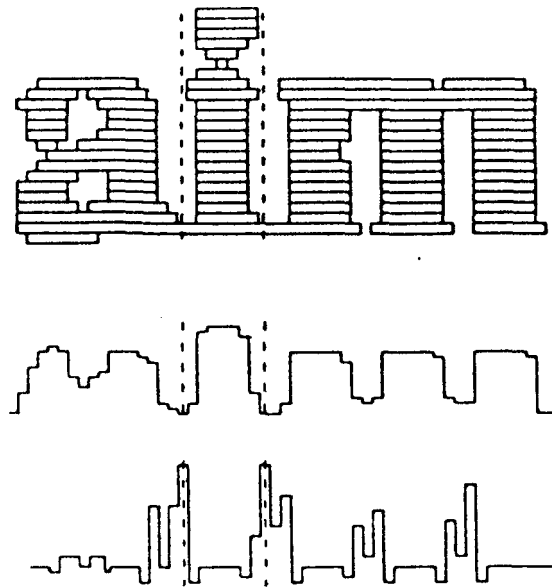


Figure 4.4.3: Top Serif Merges Due to Poor Quality Input. In the Breakpoint Criterion the Peaks are Worth to Investigate.

The classification is done by using the structural information, the LAG produced. First the character is normalised with respect to its box. In this way the problem of Multi-size characters can be solved. The Bayes classifier was selected. The reason why a statistical classifier is selected above a syntactical is because it is automatically trainable and it is faster.

After the text is put together, postprocessing using the spelling list from [McIlroy, 1982] for correction is performed to increase the recognition rate. Characters which do not pass a certain threshold are tried to be corrected by the system: e.g., *spe*]] to *spell*.

At this moment the current methods are speeded up, and other more promising methods concerning human-reading abilities, high-performance classifiers, etc. are investigated to increase the system's performance.

4.4.2. Recognition of Isolated Hand-written Numerals

One of the main problems in recognising hand-written numerals is to yield high accuracy (99%) and to isolate the various characters correctly. In this part we shall describe the results and the methodology of a research project set up to recognise isolated hand-written numerals. [Shridhar et al., 1986] The system recognises 99% of isolated numerals. When the numerals were connecting or touching an accuracy of 93% was achieved.

The method is a pure statistical one. A number of features of characters is collected from the left and right profiles of the external contours. The set of profiles contains $LP(k)$, $RP(k)$; $k=1,2,\dots$; NH where $LP(k)$ and $RP(k)$ are the left and the right profiles, NH is the height of the character and W is the width of a character. In figure 4.4.4, a typical set of those features is shown for the numeral 5.

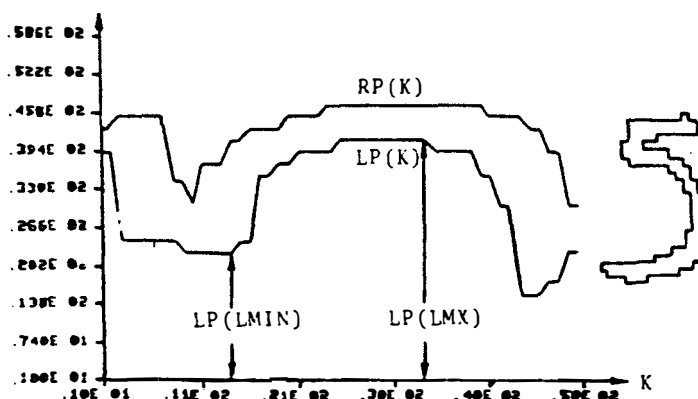


Figure 4.4.4: Left and Right Profiles of the Numeral 5.

From those features a number of new features is derived to obtain a unique set for the numeral. These are as follows:

$$\text{Character Widths: } W(k) = RP(k) - LP(k)$$

$$\text{Ratio: } R = \frac{NH}{W_{\max}}$$

$$\text{Maximum width: } W_{\max} = \max_k(W(k))$$

$$L_{\max} = \text{location of } \max_k\{LP(k)\}$$

$$R_{\max} = \text{location of } \max_k\{RP(k)\}$$

$$L_{\min} = \text{location of } \min_k\{LP(k)\}$$

$$R_{\min} = \text{location of } \min_k\{RP(k)\}$$

With those features it is possible to characterise a numeral. Special functions are necessary to calculate peak values. Using a hierarchical classification method (statistical classifiers at different levels of sub-classes, or clusters) the characters are divided among the different classes. In figure 4.4.5 a number of specimens for hand-written numerals are shown to give an insight in the complexity of the problem. This method is a variation of the method used at the Dutch Post Bank for the recognition of hand-written numerals in banking transaction.



Figure 4.4.5: Specimens of Hand-written Data

The system is implemented on a Cyber supercomputer. In the concerning article [Shridar et al., 1986] no word has been spoiled on recognition speed and scanning.

4.4.3. Benchmarks

There are different ways to interpret a benchmark. First, the speed and the recognition rate can be investigated, next the ability to recognise different fonts and sizes, etc. It was quite difficult to find benchmarks of the different Optical Character Recognition Systems. The following are the only reliable and most complete benchmarks we have been able to find. They are revealed in [Kahan et al., 1987] [Shridhar et al, 1986] [Kushnir, 1985]. In table 4.4.1 we compare the speed and recognition rates; in table 4.4.2, the speed is outlined in more detail; in table 4.4.3 different abilities are compared. The speed is expressed in seconds / page, where the scanning, pre- and postprocessing are also involved. The speed is compared in characters/second.

Name	Company	Machine	Speed char/s	Recognition Rate
Bell Labs Reading Machines	AT&T Bell	VAX 11/750	5	97%
Univ. of Windsor	-	Cyber	-	99%
Shizuoka Univ.	-	HP 2100	0.14	87%
Recognita	Recognita Inc.	IBM PC, PC/XT, PC/AT	5.7	-

Table 4.4.1: Programs and Their Speeds and Recognition Rates

Name sec/page	Scanning sec/page	Preproc. char/sec	Classification char/sec	Postproc. char/sec	Total char/sec
Bell	5	180	-	-	5
Windsor	-	-	-	-	-
Shizuoka Univ	-	-	-	-	0.14
Recognita	4	-	-	-	5.7

Table 4.4.2: Speeds in Detail.

A page consists of 3,000 characters. Under preprocessing we understand line thinning and transformations, classification is the recognition part, postprocessing is done by spelling- correction or *n*-gram analysis methods.

Name	Handwritten	Printed	Preproc	Classification	Postproc
Bell	no	yes	Line Thinning Contour Analysis Character Breaking	Bayes	Spelling Correction
Windsor	numerals	-	Statistical Analysis	Hierarchical	no
Shizuoka Univ.	Hebrew Katakana	-	Hough	Bayes	Heuristics
Recognita	no	yes	Statistical Analysis	Hierachical	no

Table 4.4.3: Abilities of the systems.

4.5. CONCLUSIONS ON OPTICAL CHARACTER RECOGNITION

During the past twenty years, substantial efforts have been devoted to the subject of optical character recognition. Recognition of hand-printed characters has advanced from the problem related to few symbols to the recognition of a full set of Chinese characters. Various techniques for preprocessing have been investigated as well as different recognition algorithms. The result reflects the importance of the quality of the data, and the feasibility of varying constraints to achieve high performance ratings. When trained subjects are used, and the shapes of the models are specified with provisions to disambiguate easily confusing characters, a remarkably high recognition rate has been obtained with the present machines. The recognition of unconstrained hand-prints, has been touched only slightly through some interesting results. Anyway, the very fine distinction and intelligent interpretation of the immense variety of handprints by human beings are acquired through a long learning process. The extraction of the essence of such knowledge and the inference into the computer will shorten the distance between human and machine intelligence in hand-printed character recognition.

When we consider the character recognition process as the three subphases: preprocessing, classification and postprocessing, discussed earlier, we can summarise as follows:

- **Preprocessing.** This phase remains necessary, even with a very sophisticated perfect classifier, because of the fact that the input data in optical character recognition is very large and contains much redundant information. Moreover, preprocessing must reshape the encoding in a more computer-friendly form for speed increasing and for a better use of structural methods. The graph-based approach [Kahan et al., 1987] seems to be one of the best at this moment, concerning a fast line thinning. The Hough transform, however, provides us with a data structure which can be used directly in a structural classifier. When the Hough transform is changed slightly, it can also be used for automatic line thinning. The thresholds can be used as a tool in this application. The Hough transform can give some problems when the input data is of poor quality, e.g., there are gaps in the lines. The Hough can be used to filter to many lines, but is can not be used to connect lines together whenever there is an empty space between them. We shall return on this subject in the discussion of our Hough transform.
- **Classification.** This is the crucial item in optical character recognition. The statistical classifiers are fast, automatically trainable, but very unflexible. The structural classifiers are flexible, more fit for human perception ideas, but are slow and can become very complex even in small problems. In practice, the statistical classifiers do not work well without reclassification of structural methods. The speed argument can be solved by using very good preprocessing techniques, parallel processing and better, more overthought structural methods. The learning in structural methods can be much better than in statistical methods, although statistical classifiers are automatically trainable. However, the system designer must design a system that is prepared for it. Another reason for using a structural classifier, is that humanly seen, structural classification is the only good classification, and speed and complexity arguments may not be the reason for no further research in this area. Thus, this study must give a better insight into the problems arising in structural classification and provide a solution for them.
- **Postprocessing.** This phase is best served with spelling-correcting techniques. The only new developments in this area are new ways to squeeze tens of thousands of words into less memory, faster entries to the spelling lists, etc.

The system we propose is one which uses after scanning a split algorithm to detect the different letters. A simple pixel-based line-thinning algorithm to reduce the data, especially in the corners of characters. Then, further line thinning and curve and line detection can be done with the Hough transform. Classification can be done with a structural classifier, using skeletons to represent the characters, and frames to store the known characters. The frames can be searched through rapidly with the aid of the trigram frequency tables. The learning process is implicitly resident in the trigram frequencies. Further, frames are written in LISP, and LISP can generate LISP. Hence, a program can be written which generates frames for a number of input characters (or in instantiations). The postprocessing can best be done with (domain-dependent) spelling lists. The system is shown in figure 4.5.1.

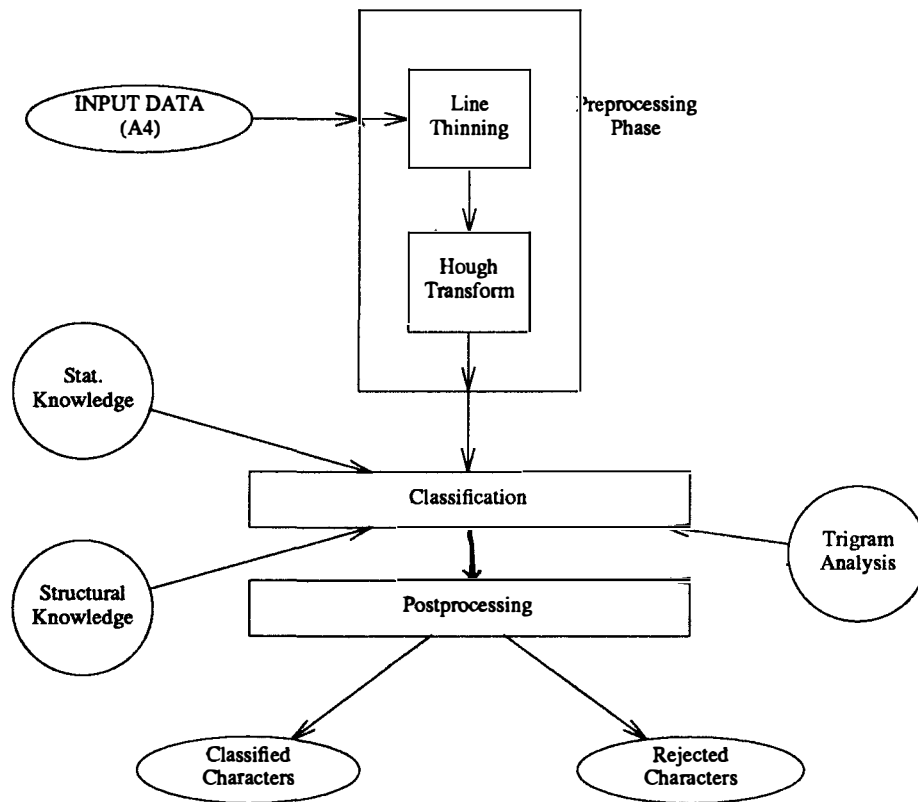


Figure 4.5.1: The Proposed Optical Character-Recognition System.

4.6. RESEARCH OUTLINE

The next question is how to parallelise such a system. There are globally three ways to do this:

- Split the data on the host and perform the line thinning, Hough transform and classification on the different processors. Once a number of characters have been recognised, the data is sent back to the host, where the spelling correction takes place.
- Idem as above, but once a character of a certain type is recognised, this type is broadcasted to the other processors to be checked first.
- The knowledge is split and distributed over the different processors; the host or a processor performs the character distribution; once a character is classified in a subclass, the result is sent to the processor containing that specific knowledge. Postprocessing, e.g., spelling correction takes place on the host.

The first solution is quite a brute force one, the second offers a good perspective, and the third fails probably on communication. We have decided to choose the second option. This concludes also our literature overview.

PART 2

DESIGN, REALIZATION and IMPLEMENTATION

- **THE PROGRAMMING ENVIRONMENT**
- **THE PREPROCESSING**
- **THE CLASSIFIER**
- **THE POSTPROCESSING**

CHAPTER 5

THE PROGRAMMING ENVIRONMENT

ABSTRACT

In this Chapter we shall discuss an important aspect of our work on the NCube, viz. the *Programming Environment*. An note worthy decision, i.e. the realization of an distributed Scheme Interpreter, resulting in the Flisp environment, will be specified. The arguments for our choice instead of parallelising one Scheme interpreter, will be presented in Part three of this report. The *first* section provides some background on Scheme and the second describes how a distributed Scheme environment, called the Flisp system, is realized on the NCube four/+, and in the third section some examples of parallel programs are presented;

5.1. BACKGROUND

The AI-classifier project was planned as a half-year work for three students. According our opinion this task could only be accomplished in such a short time if the following requirements were met :

- After the design, further development, implementation and testing phases should be performed concurrently instead of sequentially which can back fire seriously if original systems turn out to be basically wrong.
- The program should be realized as a module system to facilitate the distribution of work and to maintain an abstract and functional description of the entire system and its components.

The modular construction of an AI classifying system has already been described in Part one of this treatise. The first requirement, however, resulted in the realisation of a Scheme interpreter on the NCube processor nodes. In the following sections advantages of a symbolic manipulation language such as Scheme, implementation aspects and some examples will be discussed.

Programming in Scheme allows using rapid prototyping, e.g., the fast implementation of concepts and testing them at the same time. This ability is based on several features that are included in Scheme, or for that matter in any Lisp flavour. These basic features are :

- Symbolic manipulation supports high-level programming
- Interpreted language allows fast testing
- Memory management with garbage collection
- Dynamic programming allows programs to change themselves as if they were in a learning process.
- Knowledge representation with frames can easily be implemented in Scheme

5.2. SCHEME

The programming language Scheme originated at MIT [Sussman and Steele, 1975]. This introduction is based on an article in AICOM by Daniel Schneider [Schneider, 1987].

5.2.1. The Philosophy and History of Scheme

Scheme has several things in common with Pascal. One of them is the objective to educate computer science students. Another is the concept of "streamlinedness" emphasized in the Pascal language.

A first description of Scheme written by Guy Lewis Steel Jr. and Gerald Jay Sussman in 1975. A revised report appeared in 1978. As local dialects began to diverge a meeting was organized for representatives of the major implementations in October 1984 to work towards a better and more widely accepted standard for Scheme. Subsequently a final "Revised Revised Report" completed the language definition in 1985.

It can be said that the relation of Scheme to Lisp is what Pascal has been to the Algol languages. This emphasizes the idea that Scheme cleaned up Lisp. Other people opine that Scheme is a Algol dialect departing from the Lisp family, because it uses lexical scoping. From this point of view Scheme is a union of Lisp and Algol, "leaning" towards Lisp. From a practical point of view, Scheme is the Modula-2 of Artificial Intelligence (as opposed to Common Lisp : the ADA of AI). This, however, does not mean that Common Lisp is more powerful than Scheme; in some ways the opposite is true :

- First class procedures in Scheme prove the usefulness of static scope rules and block structures in a dynamically typed language. First class objects in scheme are objects that can be passed on as arguments to other procedures.
- Scheme uses a single lexical environment for all variables.
- The operator position in a procedure call is evaluated in the same way as an operand position, at this point procedures have become procedural objects.
- By relying entirely on procedure calls to express iteration, Scheme emphasized that Tail-recursive procedure calls are essentially goto's that pass arguments and not gosub's.
- First class escape procedures enable construction of all known sequential control structures.

A few of these innovations have recently been incorporated into Common Lisp, while others remain the be adopted.

5.2.2. The Scheme Language

A Scheme program is based on *expressions*, *literals*, *procedure-calls* and some *special-forms*. A Scheme *expression* is a construct which returns a value. A value may be a variable reference, a procedure call or a special-form. *Literals* evaluate either to themselves (such as numbers, strings or boolean constants) or they are quotations. Lists are used to build the basic datastructures in Lisp, but they also serve to define procedural abstractions or to make procedure calls. If a list is quoted it is a literal, else it is a procedure-call or a special expression. Simple *procedure-calls* are represented by (*<operator><operand>**) lists. A complex procedure-call has operands that are themselves procedure calls. The evaluation of procedure calls is done with the "evaluate the arguments and then apply operator" method, also known as the eval-apply process.

A *special-form* is interpreted according to special syntactic rules, i.e., they are treated according to their keyword which is the first symbol in the list. Basically, Scheme knows about fifteen types of special forms. We shall discuss the most important ones here.

A *lambda expression* returns a procedural object (procedure). It has the syntax (*lambda <formals><body>*). For example the form

```
(lambda (x y)
  (+ x y))
```

returns an (unnamed) procedural object which adds x and y. Normally, these procedural objects are stored somewhere for later use. The concept of procedural objects enables a Scheme program to add procedural knowledge to itself, as procedures can be created dynamically, stored in datastructures, returned as results of procedures, and so on.

Definitions bind values to variables in a certain environment (the scope). Variables (at the top level) in Scheme are defined with the *define* expression :

```
(define <var><val>)
```

The same form can be used to bind a procedural object to an identifier (the procedure-name), but there is a more convenient "syntactic-sugar form" which is displayed in the following example :

```
(define (mul x y)
  (X x y))
```

this form is syntactic sugar and is equivalent to the formal syntactic correct form

```
(define mul
  (lambda (x y) (X x y)))
```

A procedural object created by the evaluation of a lambda special form is also called a closure. The "streamlinedness" of Scheme is based on the following six ideas only; everything else can be built as syntactic sugar :

1. variables and constants (*quote*)
2. functional abstractions (*lambda*)
3. branching (*cond*)
4. side effects (*set!*)
5. continuations (*call/cc*)
6. applications (*apply*)

A *conditional expression* is a list with the first element containing the expression *cond*. The other elements in the list are clauses which are themselves lists containing a test, which, if true, computes the expressions following and returns the last expression as value of the conditional expression.

Procedural objects are first-class data and as such they can be passed on to procedures as an argument. This is in many ways very convenient because it helps us to maintain abstraction in our program as can be seen in the following example :

```
(define (accumulate e11 e12 how)
  (how e11 e12))
```

When designing this definition of *accumulate* it is not necessary to know what operations may be applied to *e11* and *e12* in future, so we have introduced an abstraction level in our definition.

Lexical scoping, or statical binding, dictates that free variables in a procedure are taken to refer to variables in enclosing procedures; that is, they are looked up in the environment in which the procedure was defined. The nesting of definitions is called block structure and originated with the programming language Algol 60. This block structure can be used to manage large programs by dividing it in tractable pieces. The following is an example of block structured programming in Scheme, based on the concept of lexical scoping. The example is taken from "Structure and Interpretation of Computer Programs" (p. 27-28) and is about square roots :

Consider the following Scheme program for calculating the square root of *x* :

```
(define (accumulate e11 e12 how)
  (how e11 e12))
```

When designing this definition of *accumulate* it is not necessary to know what operations may be applied to *e11* and *e12* in future, so we have introduced an abstraction level in our definition.

Lexical scoping, or statical binding, dictates that free variables in a procedure are taken to refer to variables in enclosing procedures; that is, they are looked up in the environment in which the procedure was defined. The nesting of definitions is called block structure and originated with the programming language Algol 60. This block structure can be used to manage large programs by dividing it in tractable pieces. The following is an example of block structured programming in Scheme, based on the concept of lexical scoping. The example is taken from "Structure and Interpretation of Computer Programs" (p. 27-28) and is about square

roots :

```
(define (sqrt x)
  (sqrt-iter 1 x))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) .001))

(define (improve guess x)
  (average guess (/ x guess)))
```

The problem here is that the only procedure important to the user is `sqrt` and that we would rather hide the other procedures from the user, but not from procedure `sqrt`. This can be accomplished with lexical scoping as can be seen in the following Scheme program :

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) .001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1))
```

The advantage of this construction is obvious: not only have we hidden the procedures *good-enough?*, *improve* and *sqrt-iter*, but we also have decreased the amount of source code by setting the scope of variable *x* to all three procedures, and so deleting the necessity of passing *x* on to these procedures as an argument.

Object-oriented programming is an important discipline in AI. In essence an object is a data class that incorporates procedural knowledge as well as "common" data. Semantic features of Scheme like procedural objects and lexical scoping enable some primitive object-oriented programming. In the next example object oriented programming is achieved with procedural objects that keep some internal state as a result of lexical scoping. The example is taken from "Structure and Analysis of Computer Programs" (p. 168-173).

```
(define (make-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance ))
```

The result of evaluating the *make-withdraw* procedure is a procedural-object with an environment that contains the "variable-value" binding of variable *balance*. Each point of execution in a Scheme program is associated with an environment. Environments are frames [†] which mirror the levels of binding constructs in a program, such as the ones produced by a procedure or lexical form. In short this means that every time procedure *withdraw* is called a new environment is created with a new variable-value binding for *amount*. The procedural object that is a result from *make-withdraw* is also called a lexical closure.

[†] The frames should not be confused with the frame-concept as proposed by Minsky [M. Minsky, 1975].

Iteration in Scheme is entirely done with recursion. Such a recursive description of an iterative computation can only be done in constant space if tail-recursion is properly handled. A procedure is tail-recursive if it is recursive and the only recursive call is at its end (the tail). In the case of tail recursion it can be proved that the recursive call is a goto that passes arguments rather than a gosub which requires a return address that is put on the stack.

The final but certainly not the least important concept that we shall discuss in this introduction are *continuations*. The course of execution of a Scheme program consists of a series of continuations, each resembling a state in the evaluation of a Scheme expression. Although this is true for all programming languages, in Scheme these continuations possess "first-class" status and can be explicitly created by the user while in other languages they are transparent.

Continuations are useful for implementing a wide variety of advance control structures, including non-local exits, backtracking, and coroutines. *Call/cc* meaning "call-with- current-continuation" allows the creation of an explicit continuation object :

```
(call/cc <procedure>)
```

The *call/cc* expression passes the current continuation as an argument to *<procedure>*. This argument represents the remainder of the computation from the *call/cc* evaluation point. The result of this evaluation is the result from the evaluation of *<procedure>*. However, the evaluation of procedure may be stopped if the argument, containing the current continuation, is invoked with any value that is then taken as the result of the evaluation. The next example shows an implementation that uses *call/cc* to implement a simple non-local exit. *Add-tree* is a procedure which adds the numbers of a tree-structure. The *S* variable is bound to the procedure body computing out addition. (*cond...*) has four clauses: The first tests whether we are dealing with a subtree and if true causes a recursive tree walk; the next clause tests the empty subtree; the third clause tests whether a leaf has an illegal argument and calls the continuation with a message with which our recursion is exited; the last clause returns the value of a leaf.

```
(define Add-tree
  (lambda (tree-of-numbers)
    (call/cc
      (lambda (exit)
        (letrec ((S (lambda (tree)
                      (cond
                        ((pair? tree)
                         (+ (S (car tree)) (S (cdr tree))))
                        ((null? tree)
                         0)
                        ((not (number? tree))
                         (exit (list "failed" tree)))
                        (else
                         tree))))))
          (S tree-of-numbers))))))
```

In the example the continuation object is used as second class escape structure. A continuation object can also be used as first class data object. The following example first prints "hello" on the user screen and stops, returning a continuation that resembles the end of the evaluation. If this continuation is invoked, the evaluation is continued and "world" is printed.

```

(define (interact)
  (call/cc
    (lambda (halt)
      (call/cc
        (lambda (world)
          (print "hallo")
          (halt world)))
      (print "world"))))

-> (define continue (interact))
hello
continue
-> (continue '())
world

```

This example ends the introductory to the Scheme language. Much remains to be said but a full introductory lies not in the scope of this report.

5.2.3. Flisp is a Scheme implementation

The Scheme interpreter that we use has been developed in C by Frank Eggink [F. Eggink, 1987] on a SUN 3 workstation at Westmount Technology. Implementations for ATARI 1040 ST and IBM PC personal computers will be available in the near future.

Roughly all functions mentioned in the Scheme definition are implemented. Only the functions concerning other data types than symbol and integer are not implemented. No macro definitions are possible, file I/O has been poorly implemented and is not according the definition. There is no compiler implementation.

The Flisp memory is defined as a pool of CELLS. A freelist of cells keeps a record of all unused cells in the pool. A cell has two pointers and *tag* byte. The two pointers respectively point to the *car* and *cdr* of the cell. The *tag* byte is used to store flags for typing and housekeeping like the printer and garbage collector. Three types of data are supported by Flisp :

- *Integers*
- *Strings*
- *Atoms*

Garbage collection takes place whenever there are no unused cells left in the memory pool of the interpreter, viz. the freelist is empty. A mark-sweep algorithm is then executed that marks all cells that under no circumstance can be accessed in future and subsequently adds all marked cells once again to the freelist (sweep). For a description of the mark and sweep garbage collection algorithm see "Structure and Interpretation of Computer Programs" (p. 498) [H. Abelson et. al., 1985].

An expression is evaluated in its own environment. An environment is represented by a list of frames. Each frame contains a list of cells, called *dotted pairs*. The *car* of such a dotted pair is a variable that is bound to a value, the *cdr* of that same pair. When a procedure is defined, the procedure stores its current environment in its data structure. When a procedure needs to be evaluated a new frame, containing bindings between local parameters and the evaluated arguments of the procedure call, is put on top of the frames in the environment that was stored at the time of the procedure definition. Variable look-up is done by examining all frames until the active binding is found. In short this is how the lexical scoping, or statical binding, is done.

A practical aspect of the interpreter implementation is the number of bytes occupied by one cell and the number of cells required to do the necessary calculations. A cell contains two pointers that both occupy four bytes. This together with the tag byte adds up to nine bytes which is increased to ten by the compiler to ensure even addresses. The interpreter itself occupies about 50 Kbytes at least, up to 100 Kbytes depending on the amount of built-in functions or even more if trigram frequency tables and bitmaps are compiled with the program. The nodes each have 512 Kbytes of RAM memory. Our estimate is that we can have a memory pool that contains 30.000 to 40.000 lisp cells which to our calculations should be enough to

perform the necessary computations.

5.3. DISTRIBUTED SCHEME

The idea was not to implement a parallel Scheme interpreter but rather to implement several interpreters that are distributed over the nodes of the NCube. There are, however, studies that examine Lisp from the point of view of parallel computation. They attempt to identify exactly where the potential for parallel execution really exists in Lisp and what constructs are useful in realizing that potential. An example of such a study is a paper from Michael van Biema at Columbia University [M. van Biema, 1987]. Van Biema presents three case studies of attempts made at augmenting Lisp with parallel constructs, like *futures*, *delays* etc. In Part Three we shall support our decision to use a distributed Scheme interpreter with arguments related to performance and local memory concepts.

Although the C program has been designed with special attention to portability, the implementation on the nodes of the NCube system presented us with an input/output problem, since the nodes on a NCube four/+ have no disk access nor terminal connection. We will first discuss some implementation aspects of the Scheme interpreter and then consider the I/O problem and user interaction with more than one computer at the same time via one ANSI typed terminal.

5.3.1. Communication

Programs that are run on a node of the NCube four/+ have no direct access to the hard disk nor to a terminal device. This is a problem for any program that uses a disk to input and output data to and that communicates with a user by means of a terminal.

We were confronted with this problem when implementing the Flisp interpreter on the NCube nodes. Originally the interpreter had been written in C on a SUN workstation under UNIX operating system 4.2 BSD. The problem was solved by programming a standard input/output library for programs that run on a node and that is almost the same as the standard I/O package as described by B. Kernighan and D. Ritchie in their book "The C programming language" [Kernighan et. al., 1978]. The advantage of this standard is that it is also supported by the UNIX version operating on the SUN workstation.

5.3.1.1. Library access

Input and output are not part of the C language, so the construction of an I/O library is not so strange as it may seem. The library contains a set of functions that are designed to provide a standard I/O system for C programs. The functions are intended to present a convenient programming interface. From the point of view of portability we decided to let the functions in the I/O library for node-to-node and node-to-host communication correspond with the functions in the standard I/O library for C programs. Only the streams have become channels and the `<stdio.h>` (standard input and output) header file has become `<nio.h>` (node input and output), containing macro definitions of some functions to avoid overhead.

5.3.1.2. Character I/O: `n_getc` and `n_putc`

The simplest input mechanism is to read one character at a time from an input channel with `n_getc(chptr)`. The argument to `n_getc` is a pointer to channel description, `chptr`. A channel is connected to the source from which the characters are coming and will be discussed in the next paragraph. The standard input channel is referred to by `stdin`.

For output, `n_putc(channel,c)`, puts the character `c` on the output channel pointed to by `chptr`. The standard output channel is referred to by `stdout`. In the header file macro definitions for `n_getchar()` and `n_putchar()` have been implemented. These are equivalent to respectively `n_getc` and `n_putc` only they are directly connected to `stdin` and `stdout`.

The standard input and output channel are connected to the host and must be initialized by the user application, by calling `n_init()`. For details on this implementation see also Appendix E.

5.3.1.3. Channels

Before a channel can be read from or written to it has to be opened by the library function *n_open*. *N_open* takes an external address, e.g. a logical node address, and returns a name which must be used in subsequent reads or writes of the channel.

This internal name is actually a pointer, called a *channel pointer*, to a structure which contains information about the channel, such as the location of a buffer, the current character position in the buffer, whether the channel is for input or output, etc. It is not necessary to know the details, because part of the node I/O definitions obtained from the `<nio.h>` header file is a structure definition called *CHANNEL*. The only declaration needed for a channel pointer is exemplified by :

```
CHANNEL *n_open(), *chptr;
```

This says that a *chptr* is a pointer to a channel, and *n_open* is a function that returns a pointer to a *CHANNEL*.

The first argument to *n_open* is an integer, the logical node address of a destination or source node which must be in the allocated hypercube. The second argument is the *mode*, a character string which indicates how one intends to use the channel. Allowable modes are read *r* or write *w*. The third argument is a user supplied *type* that is used when sending or receiving messages to or from other nodes. The exact meaning of this type will be discussed in the next paragraph.

The function *n_close* is the inverse of *n_open*; it breaks the connection between the channel pointer and the node it was connected to by *n_open*. In case an output channel is closed, the buffer is flushed to the destination before the channel is disconnected.

5.3.1.4. The Vertex System Interface

Low level input and output on the nodes is handled by the operating system Vertex. Vertex offers primary facilities for handling messages. A message is simply an array of bytes with four associated attributes: *source*, *destination*, *length* and *type*. At this level all input and output is done with *nread* and *nwrite* [NCube Handbook, version 1.1]. Vertex uses the message attributes to setup instructions for DMA interfaces, calculate routing information, divide the message in packets and maintain data security with ECC. Vertex system calls concerning message communication are :

- *nread(&buf, buflen, &source, &type, &flag);*
- *nwrite(buf, buflen, destination, type, &flag);*
- *ntest(&address, &type);* and
- *whoami(&address, &pid, &host, &dimension).*

A *&* meaning that the reference to the variable should be passed as an argument and not its value. In general a message buffer is used to store information that has been received or is to be sent. An important feature is the *type* argument. When a message is sent to the destination node it may be typed by the user with a value between 0...255. The *nread* call has references to the logical node address from where the message should be sent from and its type. The user may let *&address* and/or *&type* refer to "-1", meaning that a message is read from any source and/or type. After the message has been read the "-1" value(s) are replaced by the actual source address and/or type of the message that has been received. This assignment explains the need for variable references in the *nread* call instead of variable values which is the case in a *nwrite* call.

Nread is a blocking function, which means that it only returns if a message is sent to it that matches the source address and type. This can be inconvenient if the user is not sure from which source or of what type the message will be and thus deadlocks may occur. Therefore the function call *ntest* is supplied to the user. *Ntest* tests whether there is a message that matches the values referred to by arguments of the *ntest* call. By referring to the value "-1" the user indicates that any message succeeds, as is the case with *nread*. If no such message is available, *ntest* returns with arguments remained unchanged. Otherwise any "-1" values are replaced by the actual source address and/or type respectively and *ntest* returns.

An user application that runs on a node, does not know on forehand on which node it will be executed and neither what the dimension is of the cube its node belongs to. It is important to have this information with respect to communication with neighbours. This information is obtained by calling *whoami*. *Whoami* returns with the following information : the logical node address of the node; the process number of the process running on the host that allocated the cube the node is active in; the identifier of the channel to the host that has been assigned to that application; and the dimension of the hypercube it is active in.

5.3.1.5. Flisp communication interface

In order to be able to use the system calls mentioned in the previous paragraph we have designed ten functions that are built- in Flisp that can be directly used by any Scheme program, running in the Flisp environment. The built-in functions are, in alphabetical order :

nclose, nopen, nprint, nread, ntest, ntime, see, seen, tell, told and *whoami*.

Except for *see, seen, tell* and *told* the meaning of the functions is the same as the corresponding C functions from the standard I/O library for the nodes. Below we give a short description of each function and its syntax.

1.(*nopen arg1 list*)

The first argument is a zero for a read type channel or non zero for a write type channel. The second argument is a list consisting of two elements : the first one represents the logical node address of the node to which the channel should be connected, the second is the message type that will be used to when information is either received or sent through the channel. Take for example:

```
->(define channel (nopen 1 '(1 116)))
channel
```

that opens a channel to node 1 for writing with type 116.

2.(*nclose <channel>*)

A *<channel>* that was once returned by *nopen* is disconnected.

3.(*nprint <expression> <channel>*)

Nprint prints the evaluated expression, unless it was quoted, to *<channel>*. If the channel has not been opened for writing the message is not sent. *Print* prints its argument to the standard output. Consider the following example:

```
->(define n (nopen 1 '(1 116)))
n
->(nprint (+ 4 5) n)
t
```

The expression *(+ 4 5)* is evaluated and the result is sent to node one with type 116.

4.(*nread <channel>*)

Nread reads a lisp-structure from *<channel>*. *<Channel>* should be a read type channel, otherwise *nil* is returned. *Read* reads from the standard input.

5.(*ntest*) or (*ntest '(<node> <type>)*)

Ntest without arguments tests for any message from another node, viz. node and type refer to a "-1" value. *Ntest* with a list as an argument checks whether there is a message from *<node>* with type *<type>*. *<type>* and *<node>* may be -1. *Ntest* returns a list containing the *type* and *node* of any message that matched its arguments. If no such message was found then the list (-1-1) is returned. Consider the following example where both procedure calls are equivalent :

```

->(ntest '(-1 -1))
(-1 -1)
->(ntest)
(-1 -1)

```

In this case there was no message at all available.

6.(whoami)

Whoami returns a list containing four elements:

```
(node dimension stdin stdout)
```

- *node*, is the logical address of the node itself;
- *dimension*, is the dimension of the hypercube where the node is in;
- *stdin*, is the standard input channel; and
- *stdout*, is the standard output channel.

Four additional functions have been built for purposes of communication. The functions are :

7.(see <channel>)

See sets the standard input (by default the host, see *whoami*) to <channel> which must have been opened for input. The standard output can subsequently be connected back to the host with *seen* (8). When the standard input is redirected to <channel>, input may be read from <channel> with *read*.

8.(seen)

Seen connects the standard input back to the host. The following example exemplifies the use of *see* and *seen* :

```

->(define channel (nopen 0 '(0 116)))
channel
->(see channel)
t

;do bunch of stuff just using (read) to read from channel
;instead of (nread channel)

->(seen)
t

```

An input channel is opened for input from node 0 with type 116. The standard input is directed to this channel. Subsequently any time that input is needed from this channel, it can be read with *read* instead of using *nread*.

9.(tell <channel>)

Tell sets the standard output (by default the host, see *whoami*) to <channel> which must have been opened for output. The standard output can subsequently be redirected back to the host with *told*. When the standard output is redirected to <channel>, output may be printed to <channel> with *print*.

10.(told)

Told connects the standard output back to the host. The following example exemplifies the use of *tell* and *told* :

```

->(define channel (nopen 1 '(1 116)))
channel
->(tell channel)
t

; do bunch of stuff just using (print <expression>) to print data
; to channel instead of using (nprint <expression> channel)

->(told)
t

```

An input channel is opened for output from node 1 with type 116. The standard output is directed to this channel. Subsequently any time that output has to be written to this channel, it can be written with *print* instead of using *nprint*.

The advantage of the additional four functions is twofold:

1. they reduce the programming effort, and
2. they can be used to write channel independent programs.

5.3.2. Interaction

Besides the input/output problem, another problem had to be solved in order to develop the distributed Scheme environment. How can an user interact with more than one interpreter in a flexible manner?, viz. on each node runs one interpreter, the user is at the console, an ANSI terminal. For the run-time system without user interaction this is no problem. However, in the development phase of our project the user should be able to control all Flisp interpreters simultaneously.

5.3.2.1. The Interface System

We coped with the problem by designing an universal interface concept that runs on the host. The system consists of two subsystems: a system that dispatches output from the NCube nodes to specific displays on the terminal screen, and the other system sends user input to a user specified node, viz. a user logs on to the standard input channel of an interpreter. The first system enables the user to monitor all output of active nodes simultaneously. We implemented a simple program that is based on this concept for interaction with the Flisp interpreters.

Since all the interaction is done on the same terminal a display system was designed and a library is implemented. The library supplies the programmer with functions that offer a convenient interface to program applications with multi displays on an ANSI typed terminal. In the following paragraph the library is discussed.

The concept is very clear and can be applied to many NCube applications. However, there is one problem:

The interface program has no knowledge concerning the kind of communication that takes place between NCube application and user. For example it does not know when to receive output from the nodes and when from the user. The problem lies with the user, or rather the user terminal : the AXIS system call *read(1)* that reads input from disk or terminal is blocking, e.g. if there is no input from these devices the *read* call does not return until input is available. Hence, our interface system has to wait until the user has provided *read* with data. While the interface is waiting it cannot dispatch any output from the applications that are running on the NCube nodes, and so a deadlock may occur.

The solution to the problem is to split the interface physically into two programs that function conform to the subsystems that were described. The programs are called:

- The Node Output Dispatcher (NOD); and
- The User Input Handler (UIH).

These programs both need access to the hypercube that is allocated in the UIH. But here a new problem arises:

The channel between host and cube is allocated with the AXIS system call *nopen(1)*[†]. *Nopen* takes the dimension of the hypercube that the user wants to allocate, as an argument and on success returns a channel descriptor that the program can use to communicate with nodes in the allocated cube. This channel descriptor can not automatically be shared by two different processes.

The problem was solved because the channel descriptor turned out to be an ordinary file descriptor. File descriptors may be shared between different processes under special circumstances. A process may execute another program, its child, with the AXIS call *frun(1)*. The mother process may connect its child to a specified standard input, standard output and standard error file. Thus be by letting the UIH process execute the NOD program with the standard input of NOD connected to the channel descriptor, instead of the terminal, our problem is solved and both processes have access to the cube.

The entire structure of the interface system is in figure 5.3.1.

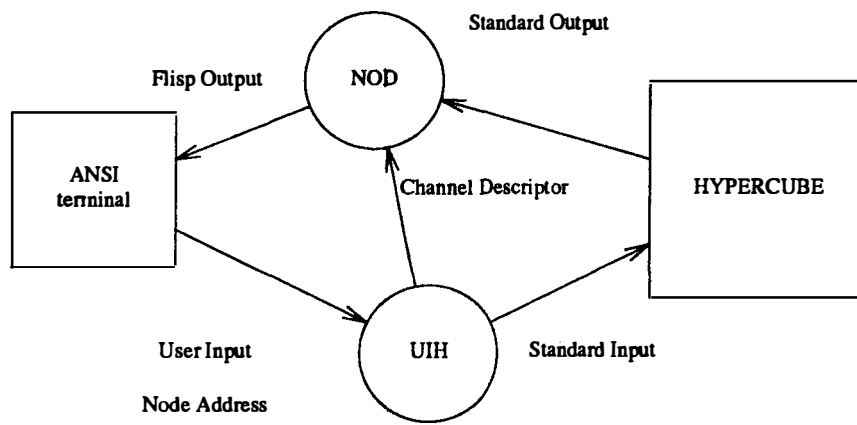


Figure 5.3.1 : The Interface System.

The UIH process allocates a cube of a dimension that has to be entered by the user. As explained it then executes the NOD program with its standard input stream connected to the allocated cube. After this has been done the lower half of the screen is allocated as a display in which the user may enter input that is sent to a interpreter running on an node. In the UIH the user may use the following four commands:

Escape-N- $\langle n \rangle$

Connect to the Flisp interpreter running on node $\langle n \rangle$; If -1 is entered the input is broadcasted to all active nodes.

Escape-R- $\langle filename \rangle$

Send the contents of file $\langle filename \rangle$ to the current Flisp interpreter;

J Match the open brackets with closing brackets and flush the result to the current Flisp interpreter; and

Control-X

Exit the Interface System, killing all processes involved.

When the NOD is initiated, it downloads the Flisp executable to the nodes in the cube. The upper part of the screen is split into 2^n displays, with n the dimension of the allocated hypercube. The output of node 0

[†] This function *nopen* should not be confused with the *n_open* library function that resides in our I/O library for communication between nodes.

to $2^n - 1$ is sent to the corresponding display.

5.3.2.2. Multi Display Management Library

Both the User Input Handler (UIH) and the Node Output Dispatcher (NOD) allocate (a) display(s) on the user screen. This led us to implement a library containing functions that offer a convenient interface for programs to realize such displays. A display is a scroll region on an ANSI terminal, e.g. displays are always 80 columns wide and can only be positioned above or below another display, no overlapping allowed. All displays have to be initialized at the same time with the function *initialize_windows*. The arguments of this function are the number of windows that have to be initialized, the top row of the first display and the bottom row of the last display. Each window gets an equal amount of lines assigned. This may sound inconvenient but we found it was better not to have too much freedom and it is always possible to change the library functions at later time. The functions *w_putchar* and *w_putstr* may be used to send characters to a specified display. In Appendix E a list of available display library functions can be found.

5.4. EXAMPLES OF PARALLEL PROGRAMS

In the other paragraphs from this chapter we have introduced the reader to a Programming Environment, consisting of Scheme, NCube communication and the Interface System. In this paragraph we would like to give the reader an impression of the power offered by such an interactive system, in which parallel concepts can be tested and implemented in an ad hoc manner.

Our first distributed Scheme program was an implementation of the Fibonacci relation, which is a recurrence relation. The relation gives the n^{th} Fibonacci number the value of the sum of the $n-1$ and $n-2$ Fibonacci number. The initial conditions chosen for the sequence of Fibonacci numbers are unimportant. We will assume $F_0=0$ and $F_1=1$. The recurrence relation then becomes :

$$F_n = \begin{cases} 0 & \text{for } n=0 \\ 1 & \text{for } n=1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

This relation can be written down in a Scheme program as follows:

```
(define (fibonacci n)
  (cond
    ((equal? n 0)
     0)
    ((equal? n 1)
     1)
    (else
     (+ (fibonacci (- n 1)) (fibonacci (- n 2))))))
```

To distribute this program over two Flisp interpreters we let one interpreter be the *master* that calculates F_n by letting the other interpreter, the *slave*, calculate F_{n-2} and self calculating F_{n-1} . We assume that both interpreters have the Scheme definition of *fibonacci* as described above. Below we describe the program for the *master* with necessary I/O added. We assume that the *master* has logical node address 0 and the *slave* has logical node address 1.

```

->(car (whoami))
0
->(define slave-read (nopen 0 '(1 116)))
slave-read
->(define slave-write (nopen 1 '(1 116)))
slave-write
->(ntest)
(1 116)
->(nread slave-read)
t
->(ntest)
(-1 -1)
->(define (distributed-fibonacci n)
  (cond
    ((equal? n 0)
     0)
    ((equal? n 1)
     1)
    (else
     (nprint (backquote (fibonacci ,(- n 1))) slave-write)
     (+
      (fibonacci (- n 2))
      (nread slave-read))))))
distributed-fibonacci
->(fibonacci 12)
144
->(distributed-fibonacci 12)
144

```

The function *backquote* is almost the same as the ordinary *quote* except that each expression in its argument that is preceded by a comma is evaluated before it is returned. This is necessary because sending *'(fibonacci (- n 1))* would not help us very much, since the *slave* interpreter does not know variable *n*. The *ntest* call has been added as an example, though it tells us an important thing namely that there already is a message from the *slave*. After reading, the message appears to be a *t*. Below we shall see that this is the acknowledge from the *slave* when it directs its standard input to the *master* after it has directed its standard output to the *master*. The *see* call returns *t* that is no longer sent to the user display as depicted below but to the *master* which explains the early *t*. The *slave* interpreter has some additional definitions and its *standardinput* and *standardoutput* are connected to the host. These additional definitions and applications for the slave are:

```

->(car (whoami))
1
->(define master-read (nopen 0 '(0 116)))
master-read
->(define master-write (nopen 1 '(0 116)))
master-write
->(tell master-write)
->(see master-read)
->->->->

```

As can be observed only the Scheme prompt is sent to the user display after standard input and output have been redirected to the *master*. The *master* is able to connect the *slave* back to the host with the following function calls:


```
->(nprint '(told) slave-write)
t
->(nprint '(seen) slave-write)
t
```

We can now benchmark the functions *fibonacci* and *distributed-fibonacci*. It turns out that if it takes time T to calculate (*fibonacci* n) then it takes $\frac{1}{2}T+C$ time to calculate (*distributed-fibonacci* n). The constant C is caused by communication time, for large n $C \ll T$. For very large n the interpreter has to collect garbage and the $\frac{1}{2}T+C$ equation is not valid anymore. In such cases (*distributed-fibonacci*) does even better than $\frac{1}{2}T$! This may sound strange for this implies a more than linear speed up. The result is caused by the fact that two interpreters together have twice as much memory CELL's at their disposal, thus fewer garbage has to be collected.

This concludes the Chapter about the programming environment. More examples are given in Part Three and in the Chapter about the Classification System.

CHAPTER 6

THE PREPROCESSOR

Functional Specifications and Algorithms

ABSTRACT

General preprocessing techniques have been described in Part one. The system's preprocessor performed in the Optical Character Classifier combines two techniques mentioned earlier connected with third one. These additional technique may be considered as a composing technique used both in the low- and in the high-level preprocessing phases. The techniques are described in this Chapter and their implementation in the Flisp environment our outlined. Furthermore, it is shown that the decomposing technique can also be used to distribute parts of the image over the nodes in the NCube.

6.1. BACKGROUND

The input data of the system is a file containing a bitmap of the image that has to be classified. This phase is called the scanning phase, and has been kept very simple by setting the following constraints :

- No digitation is done, the digital information is supposed to be available.
- There is no grey level information in the input data.
- There is no noise in the input data.

With respect to this treatise and its main objective we think that one an acceptable problem reduction is acceptable and that does not really change the nature of our problem. In figure 6.1.1 the specific phases of the preprocessing process are described.

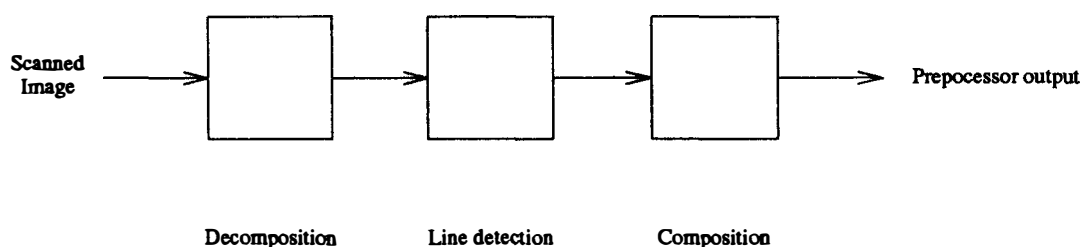


Figure 6.1.1 : Preprocessing phases

The phase that follows the scanning phase, should transform the bitmap into interpretable information for

the classifying process. We have chosen for a combination of the following three techniques:

1. A decomposition technique : The Split Algorithm;
2. An identification technique : The Hough Transformation; and
3. A composition technique : A series of Connectivity Algorithms.

Each technique can be applied singly or in combination with the others by the classifying process. This implies that decisions on the application of these techniques may be based on classification results that were obtained earlier. Such use introduces the possibility of a feed-back mechanism that allows the implementation of very smooth and "natural" classifying process. If the Classifier and the Preprocessor are integrated the preprocessing algorithms must be available from the Flisp environment, since the classification knowledge is stored in this environment. It is of no use to implement the preprocessing algorithms in Scheme since this would mean a serious slow down for all the preprocessing activities. A good compromise is made if the preprocessing functions are built into the Flisp environment, so that they can be programmed in C.

The time-consuming aspect of raster preprocessing functions is a well-known fact and many recognition systems are slowed down, e.g., by a pixel based line thinning algorithm as is described in Part one. Especially, thinning algorithms specially, are time consuming and have therefore not been included in our set of preprocessing techniques.

The identification of line-segments is also slow although the Hough transform has a good performance. In Section 6.2 we shall show that the Hough transform has a space complexity of $O(n^3)$ for line detection in a $n \times n$ plane. The time complexity is at least of the same order. For large images this is a disaster and if not dealt with it might even prove most computers being incapable of performing the necessary calculations by lack of memory. We have dealt with this problem by decomposing the input data and then applying the Hough transform separately to the image parts. The space complexity is then reduced with a factor that is the third power of the square root of the number of parts the image was decomposed to.

If the lines in the input image have been detected then there is a vector representation available. In many cases, it is desirable to have some knowledge about the structure and, the connectivity, that lies implicitly in the vector representation, which may then being considered as an undirected graph. Therefore, a series of connectivity algorithms have been implemented that can be used for instance to find all cycles in the image. This is discussed in Section 8 of this Chapter.

6.2. SPLIT : A DECOMPOSITION ALGORITHM

There are two main reasons for applying a decomposition technique. They are :

1. The Hough transform for line detection is very memory and time expensive for large images.
2. We are dealing with text, composed of lines, words and characters which can be very well decomposed.

A major problem that arises when an arbitrary image is split into parts, are lines that are then as well split into several parts although they belong together. Fortunately, this is of no concern to us because of the existence of lines, words and characters in a text image.

Before discussing the implementation of the Split algorithm we will support the first reason with some evidence :

Consider an image of $n \times n$ pixels. We can split this image in K^2 sub images as depicted in figure 6.2.1.

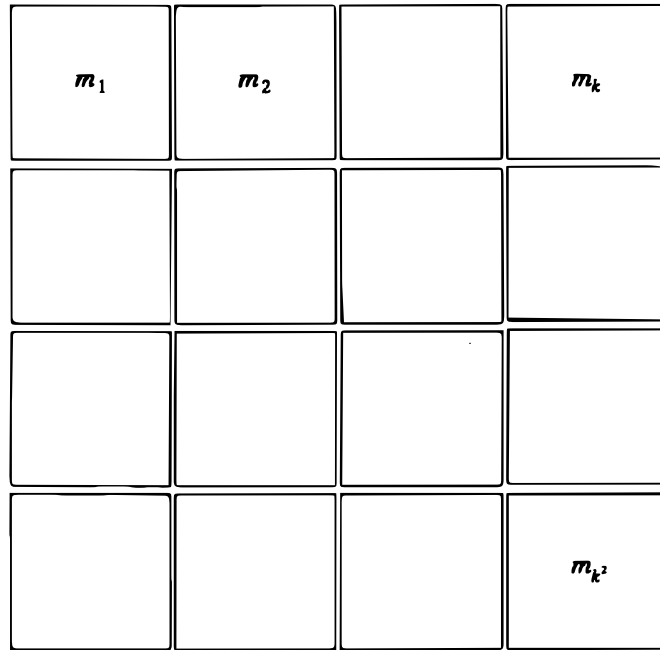


Figure 6.2.1 : A $n \times n$ image split into K^2 subimages

The space complexity of the Hough transform on input of $n \times n$ is $O(n^3)$. For each subimage this is $O(n^3/K^3)$. Since there are K^2 such subimages this would mean a poor improvement of only K . But this is not altogether true, since the workspace that is used for detecting line segments in a character can be re-used for the next character and thus the practical space complexity remains $O(n^3/K^3)$. We add that the space complexity of the Hough algorithm with a $n \times n$ input image can be reduced to $C \times O(n^2)$ with $C \ll n$, see Section 3 of this Chapter.

6.2.1. Distributed Processing of the Image

We mentioned earlier the use of distinguished processing technique as a means to distribute the input data over the nodes of the NCube. This thought has led to the implementation of a binary split algorithm recursively repeated until no split is further possible. The *split* algorithm is described in the following Section. We shall now examine algorithm 6.2.1 that uses *split* to distribute (equal) parts of the image over the nodes of the NCube.

The *split* algorithm running on a node splits the image into two halves from which one is sent to a neighbour node. This adds another functional requirement to the *split* algorithm since we must ensure that the amount of data in both halves should be equal to guarantee a good workload balancing. A neighbour node does the same with the part of the image it has received. In that way the entire image is split in 2^n (equal) subimages, each one residing on a different node. After this each node starts for himself and splits the subimage it has left in pieces until no further split is possible. These pieces are put in a Scheme list which can be accessed by the Classifying System. The procedure is described in algorithm 6.2.1 below.

Procedure *DISTRIBUTE***Method**

Let d be the dimension of the allocated hypercube

Let n be the logical address of the current node.

Let $a_0, a_1, \dots, a_{d-1}$ with a_i in $\{0,1\}$, such that $n = \sum_{i=0}^{d-1} a_i 2^i$

If $n=0$ then

 Read *image* from host

 Let k is 0

else

 Read *image* from any neighbour node m

 Let $b_0, b_1, \dots, b_{d-1}$ with $b_i \in \{0,1\}$, such that $m = \sum_{i=0}^{d-1} b_i 2^i$

 Let k be the i for which $a_i \neq b_i$

endif

For all neighbours m from n do

 Let $b_0, b_1, \dots, b_{d-1}$ with $b_i \in \{0,1\}$, such that $m = \sum_{i=0}^{d-1} b_i 2^i$

 If $i > k$ for $a_i \neq b_i$ then

 Split *image* in half 1 and half 2

 Send half 2 to node m

 Let *image* be equal to half 1

 endif

endfor

endmethod

Algorithm 6.2.1 : The procedure *DISTRIBUTE* to distribute the input image over the nodes.

This algorithm executes on every node and distributes the image correctly to the nodes the hypercube. The course of an image part from node 0 to its final destination is rather complex. The distribution of an image over the nodes in a three dimensional hypercube, a cube, is shown in figure 6.2.2 below. After this algorithm has executed a node may begin with the actual preprocessing, of the *image* that has remained after sending parts to its neighbours. The splitting which takes place is based on the same *split* algorithm that is used in algorithm 6.2.1.

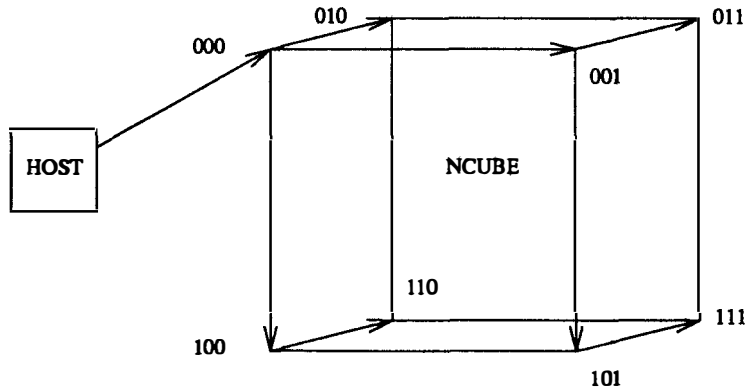


Figure 6.2.2 : Course of distribution in a three dimensional hypercube

6.2.2. The Split Algorithm

In this Section we shall examine the *split* algorithm in greater detail.

To achieve a balanced split the algorithm counts pixels in both vertical and horizontal directions and stores these in a so called horizontal and vertical profile. A profile is an array of integers, each representing the number of pixels on a vertical or horizontal line, depending on whether it concerns a vertical or a horizontal profile.

A zero pixel count in a profile entry means there are no pixels so it is possible to split the image at that place without breaking one or more lines in two separate ones. An image is split by drawing two lines in the image that have only white space, zero count's, in between them. The strip of white is vertical or horizontal depending if the zero counts were measured in respectively a vertical or a horizontal profile. By making sure that the total number of pixels T in an image is split in exactly twice $\frac{1}{2}T$, a good workload balancing is guaranteed.

The split process can be modelled by the following set of logical relations that have to be satisfied :

Let $g(x)$ be a function that describes an arbitrary profile. Suppose we would like to split the range $[a, b]$

Let T be the total pixel count of $g(x)$ in $[a, b]$: $T = \sum_{x=a}^b g(x)$

Define *split* that is a relation between $\{a, b, T, g()\}$ and $\{\alpha, \beta, U\}$. The range $[\alpha, \beta]$ denotes the strip of white space in $[a, b]$ that splits the image.

The relation *split*($a, b, T, g(), \alpha, \beta, U$) holds when α, β and U meet the following set of constraints :

Determine the smallest $p < b$ such that $\sum_{x=a}^p g(x) > \frac{1}{2}T$

p points to an equal split point in the profile described by $g()$. But we would not like to splice a character in two halves, thus we have to find an optimal p' close to p with $g(p')=0$.

$$p' = \begin{cases} k1 & \text{if } \sum_{x=k1}^p g(x) < \sum_{x=p}^{k2} g(x) \text{ and } g(k1)=0 \\ k2 & \text{if } \sum_{x=k1}^p g(x) > \sum_{x=p}^{k2} g(x) \text{ and } g(k2)=0 \end{cases}$$

The idea is not to find the p' with $g(p')=0$ that is physically closest to p but that keeps closest to the $\frac{1}{2}T$ pixel count. This may sound somewhat complicated but in figure 6.2.3 a situation is setup in which such a choice is made. Imagine the profile $g()$ as a landscape profile, see figure 6.2.3. The total area, the integration of $g()$, equals T .

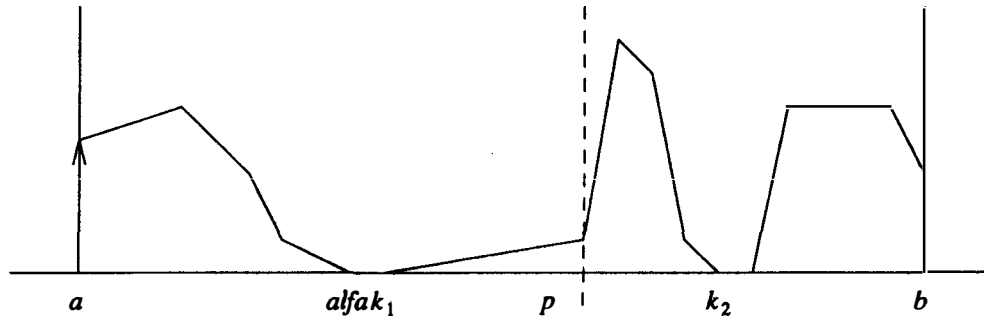


Figure 6.2.3 : A profile $g()$ and its equal split point p

From figure 6.2.3 the reader should learn that the k_1 is a better choice for p' then k_2 is, although $|p - k_1| > |p - k_2|$.

Depending on the choice between k_1 and k_2 for p' , p' is the value for β or α respectively.

In case α is known, β can be obtained from:

$$\text{largest } \beta \text{ with } \sum_{x=\alpha}^{\beta} g(x) = 0$$

else if β is known, α can be obtained from:

$$\text{smallest } \alpha \text{ with } \sum_{x=\alpha}^{\beta} g(x) = 0$$

Finally we set U equal to:

$$U = \sum_{x=\alpha}^{\alpha} g(x)$$

Further it may be noticed that the pixel count of the $[\beta, b]$ range is equal to

$$\sum_{x=\beta}^b g(x) = T - U$$

The final implementation in C doesn't differ very much from the description above.

On top of the *split* algorithm a few other functions have been implemented that efficiently split an entire text image into as many areas as possible. First word lines are separated by drawing horizontal lines, until not further possible, and then splitting each of these by drawing vertical lines. In this way the final result is a list containing area descriptions, each area containing one character. Subsequently this list is processed and on basis of the gaps between areas in the same word line they are put together in lists that represent one word.

By splitting a number of times only in one direction it is not necessary to calculate the profile arrays each time. This is only necessary when a split in an other direction is made. The calculation of the horizontal

and vertical profile is done at the same time, assuming the image as a $n \times m$ array containing only ones and zeros, respectively white and black pixels :

$$A(x, y) \text{ in } \{0, 1\}; (x, y) \in n \times m$$

$$Gh(x) = \sum_{y=0}^m A(x, y), \text{ and}$$

$$Gv(y) = \sum_{x=0}^n A(x, y)$$

The implementation of this calculation is somewhat complicated because of the way in which the bitmap, containing all the pixel information, is stored. In Section 6.3 we shall describe this datastructure in detail and methods to access pixel information are presented.

All source code concerning the split algorithm and controlling procedures are given in Appendix E.

6.2.3. The Split function description

Finally, we build the *split* algorithm in into the Flisp environment so that it may be accessed by the Classification System. The function is called *split*.

Split is called with one argument only. This argument is a list containing four elements describing the area that has to be split: $(x \ y \ a \ b)$. The x and y are the coordinates of the upper left corner of the area and the a and b are the coordinates of the lower right corner.

Split returns a list that contains all areas that resulted from the slit operation. This operation has been described in the previous paragraph. The areas are grouped into lists which are the actual elements of the result list. A group of areas contains all characters that belong to a word.

6.3. HOUGH: A LINE-SEGMENT DETECTION ALGORITHM

In Part One the Hough transform has been introduced as a method to identify line- as well as curved- segments, and later we mention that the method can be generalized to detect objects with arbitrary shape as long as they have a parametric representation. The nature of our input data from which we have foreknowledge allows us to improve on the algorithm that performs the Hough transform without changing its basic principle. These improvements and aspects of implementation are discussed in the following paragraph. The actual implementation is discussed in Section 6.3.2 and in paragraph 6.3.3 the Flisp interface to this function is described.

6.3.1. Aspects and consequences of the implementation

In this implementation we introduce the Hough transform as an algorithm to detect line segments. This means that we have chosen not to detect any other parametrised shapes with this algorithm. We support this choice with the following arguments:

- Unfortunately each parameter in a parametric representation of a shape introduces an extra dimension in the accumulator array. The role of the accumulator array is described in Part One in chapter the about Optical Character recognition. For a line this means a two dimensional accumulator array is needed, for a circle a three dimensional etc.;
- Very few curves in characters are part of a true circle which means that detection of these curves will cost a great effort;
- Many curves turn out to be very well replaceable with one or more line segments; and
- Making parametric representations of characters or character parts, which could then be directly recognized with the Hough transform, is not a sensible thing to do since:

- a. It presents no flexible method;
- b. The accumulator array will have an unpractical size; and
- c. This would be more then just preprocessing the image.

In the introduction to the The Hough transform In Part One, two parametric forms are presented that can both be used for line detection, they are:

- (1) $y = ax + b$ with parameters (a, b)
- (2) $x \cos \alpha + y \sin \alpha = r$ with parameters (r, α)

Disadvantages of one are:

- Lines with slope a , with $||a|| < 1, a \neq 0$ are not detected, and
- Vertical lines with $a = \pm\infty$ are also not detected, simply because this would mean an infinite accumulator array.

Because of these disadvantages two is then presented. But, although it lacks the disadvantages of one it has some other serious disadvantages :

- It is not possible to work without floating point calculation which slows down the system, although the NCube nodes have a floating point processor on chip;
- Floating point calculations introduce inaccuracies due to rounding errors; and
- The Accumulator array in two contains (r, α) points and updating costs much greater effort than in one because these points lie on a sinusoidal curve which is hard to calculate.

In view of these disadvantages we have chosen for representation one after having dealt with the above mentioned problems in the following manner:

Let us call the orientation of the image in normal position XY . By mirroring the image in the line $y=x$ we then obtain the YX orientation of the image. If we apply the Hough algorithm to both the XY and the YX orientation of the image both problems are solved. This can be explained as follows:

Let l be a line in the XY orientation of the image

$$l : y = ax + b$$

If $a=0$ then l reduces to

$$l : y = b$$

which is a horizontal line. When mirrored in $y=x$, this becomes a vertical line which is no longer a function in x :

$$l' : x = b$$

This reasoning holds also vice versa, a vertical line in XY orientation becomes a horizontal line in YX orientation. This means that our second problem is solved since vertical lines in the image are detected as horizontal lines in the YX orientation of the image.

else if $a \neq 0$ then mirroring i the line $y = x$ gives l' :

$$l' : y = x/a - b/a$$

l' has slope $1/a$. Thus for all lines in the image with $||a|| \leq 1$ and $a \neq 0$, the mirror line l' has slope $1/a$. Since $||1/a|| \geq 1$, l' will be detected in the YX orientation of the image, hence our first problem is solved as well. We note that lines with $||a|| = 1$ are detected in both the XY and YX orientation. By making sure that lines with $a=1$ in the YX and $a=-1$ in the XY orientation are not detected we rule out the possibility of detecting a line twice.

The value of a point (a, b) in the accumulator array is an exact measure of the number of pixels on a line. These pixels together form line segments. For the detection of these line segments it is necessary to scan the line $y = ax + b$ to detect all physical line segments that lie on that line.

We will now examine the space complexity of the accumulator array used by the Hough transform. Earlier we have pretended that this complexity is $O(n^3)$ for a $n \times n$ image. Below this result is explained:

Consider a $n \times m$ image as depicted in figure 6.3.1 below.

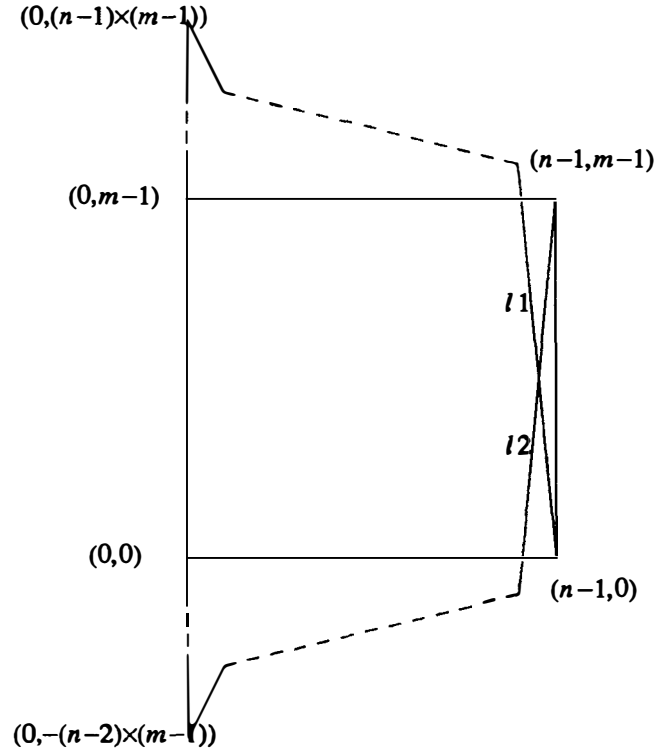


Figure 6.3.1 : A $n \times m$ image

Consider the detection of a line $y=ax+b$ in this image. The largest slope a that can be detected in this plane is $m-1$, otherwise it would be a vertical line. The highest b_1 that can be reached is the y value for $x=0$ in l_1 in figure 6.3.1 that goes through $(n-2, m-1)$ and $(n-1, 0)$. Thus the highest value for b_1 is

$$b_1 = (n-1) \times (m-1)$$

The smallest possible b_2 is the y for $x=0$ in l_2 in figure 3.1 that goes through $(n-1, m-1)$ and $(n-2, 0)$. This leaves us with a smallest b_2

$$b_2 = -(n-2) \times (m-1)$$

The pair (a, b) is a point in the accumulator array. Because we have determined the dimensions of a and b we can now determine the actual size of the accumulator array in terms of n and m , which are the dimensions of our input image. The a corresponds with the horizontal axis and the b with the vertical axis of the accumulator array. The accumulator array with its borders is depicted in figure 6.3.2.

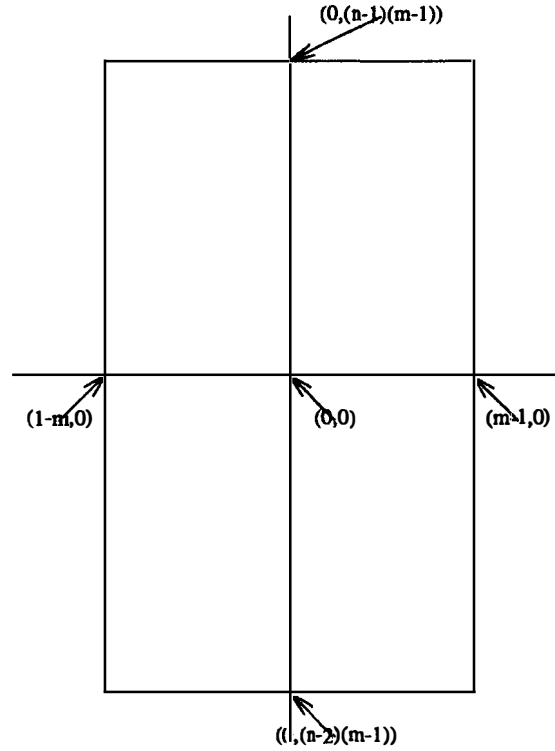


Figure 6.3.2 : The accumulator array used by the Hough transform

We can now calculate the space complexity Sh of the Hough transform:

$$Sh = O((2(m-1)+1) \times ((n-1)(m-1) + (n-2)(m-1)+1))$$

$$= O((2m-1) \times ((2n-3)(m-1)+1))$$

$$= O(m^2 \times n)$$

We note that in the case of $m = n$, Sh becomes $O(n^3)$.

This calculation is very important because it informs us on the nature of the space complexity and also on the time complexity. This time complexity Th is at least equal to the space complexity Sh , since we have to check from each point in the accumulator array if it exceeds the threshold. So

$$Th > Sh = O(n^3)$$

The representation of the accumulator array is a plane, the accumulator plane, that appears to waste much space as can be seen from the numbered areas in the accumulator array in figure 6.3.3. Areas I and III contain points (a, b) that can never lead to detection of a line $y = ax + b$ in our image.

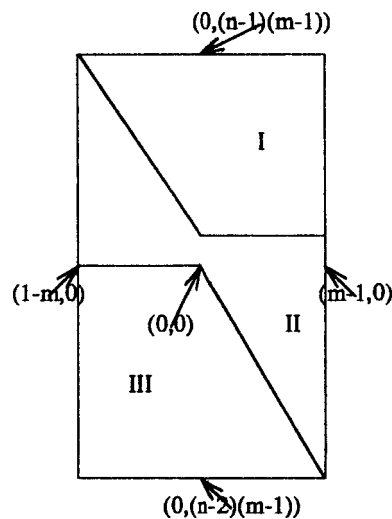


Figure 6.3.3 : Wasted space in the accumulator plane

The interpretation of figure 6.3.3 is that two out of the three pairs (a,b) are of no use. For large images the accumulator plane becomes very large, perhaps too large. In view of the amount of wasted space depicted in figure 6.3.3 this means that something has to be done.

It is possible to find a clever representation of the accumulator array that is more space economic than a plane. It is very likely, however, that the representation can not be very efficiently accessed with respect to the time involved to update points (a,b) on the line $y = ax + b$. This updating has to be performed very often and should be implemented as efficiently as possible. We think that an accumulator plane is the best representation and have therefore kept to it and instead we have tried to decrease the physical dimensions of the accumulator plane.

From test results it appeared unnecessary to detect line segments with very large slopes. In rare cases that a line with a large slope occurs it may just as well be split up in several vertical lines as can be seen in figure 6.3.4 below.

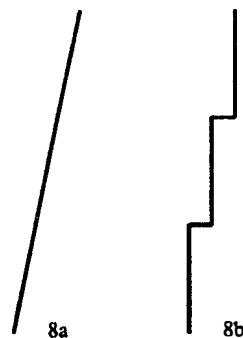


Figure 6.3.4 : A line with large slope (8a) is detected as several vertical lines (8b)

Earlier we found that the maximal slope that could be detected in a $n \times m$ image was $m-1$. Results show that detection of slopes to a maximum of 20% of $m-1$ is sufficient. For the space complexity $Sh = O(n \times m^2)$ this means a reduction of a factor 0.04.

By first decomposing the image as described in Section 6.2 of this chapter, with the Split algorithm, and subsequently applying the Hough transform separately to these areas we have small n and m , thus the accumulator plane has a reasonable size and the algorithm executes very fast.

6.3.2. The implementation of the Hough transform

The implementation of the algorithm that performs the Hough transform will be described in two parts :

1. The algorithm itself, and
2. The datastructures containing the image and the accumulator plane.

The algorithm is described in Section 6.3.2.1 and the datastructures in 6.3.2.2.

6.3.2.1. The Hough algorithm

At high abstraction level the algorithm closely resembles the algorithm that was presented in Part One about the Hough transform for line detection. This algorithm is again described as algorithm 6.3.1 below. Notice that there are two accumulator planes, one for the XY and one for the YX orientation of the image.

Procedure *HOUGH* (H, B)

Records H and B that hold coordinates of respectively the upper left corner and the bottom right corner of the area in which line segments are to be detected, the area lies in the $n \times m$ input image;

Method

Initialize the accumulator plane, its physical dimensions are $2 \times RCMAX + 1$ horizontally and $2 \times (n-1)(m-2) + 1$ vertically;

Scan the image for black pixels at (x, y) and increment all cells (a, b) in the XY accumulator that are on the line $b = -xa + y$ with a in $[-RCMAX, +RCMAX]$ and all cells (a, b) in the YX accumulator that are on the line $b = -ya + x$ with a in $[-RCMAX, +RCMAX]$;

Search the XY accumulator for cells (a, b) that exceed the *threshold* and for these pairs (a, b) check in the image if there are any line segments that have a length greater then *LLENGTH* on the line $y = ax + b$

Search the YX accumulator for cells (a, b) that exceed the *threshold* and for these pairs (a, b) check in the image if there are any line segments that have a length greater then *LLENGTH* on the line $y = x/a - b/a$ if $a \neq 0$ or on the line $x = b$ if $a = 0$.

endmethod

Algorithm 6.3.1 : The *HOUGH* procedure.

In the description of algorithm 6.3.1 two important variables *LLENGTH* and *RCMAX* are introduced. A third important variable which has been introduced earlier is the *threshold*. *RCMAX* is the maximal slope that can be detected because it is the physical dimension of the accumulator plane in horizontal direction. The *LLENGTH* denotes the minimum length that line segments must have if they are to be detected. The *threshold* is the minimal number of black pixels (x, y) that must be on the line $y = ax + b$ in the image. If the threshold is exceeded by cell (a, b) in the accumulator plane then the line $y = ax + b$ is scanned for segments and the value of cell (a, b) can be taken as the exact number of black pixels that lie on the line in the image.

These three variables are an important means to regulate the performance of the implemented Hough transform on an image. The nature of our text images causes the *threshold* variable to be of little importance. This might have been expected since characters are very compact and there exist many false

accumulator cells that exceed the value of true accumulator cells. False accumulator cells are created through pixels from different lines that together form an illusory line. It is therefore advisable to set the *threshold* variable equal to the *LLENGTH* variable.

In Section 6.3.3.1 we shall show how *RCMAX* and *LLENGTH* variable are integrated in the Hough function interface.

6.3.2.2. The Datastructures

The Hough algorithm retrieves and stores the greatest part of its information in two datastructures. The datastructures are:

1. The bitmap containing the pixel information about the image; and
2. The accumulator plane that contains cells that are accumulated which are used to detect lines.

The bitmap is implemented as an array of 32-bit words. The image overlaid by its datastructure is depicted in figure 6.3.5 below.

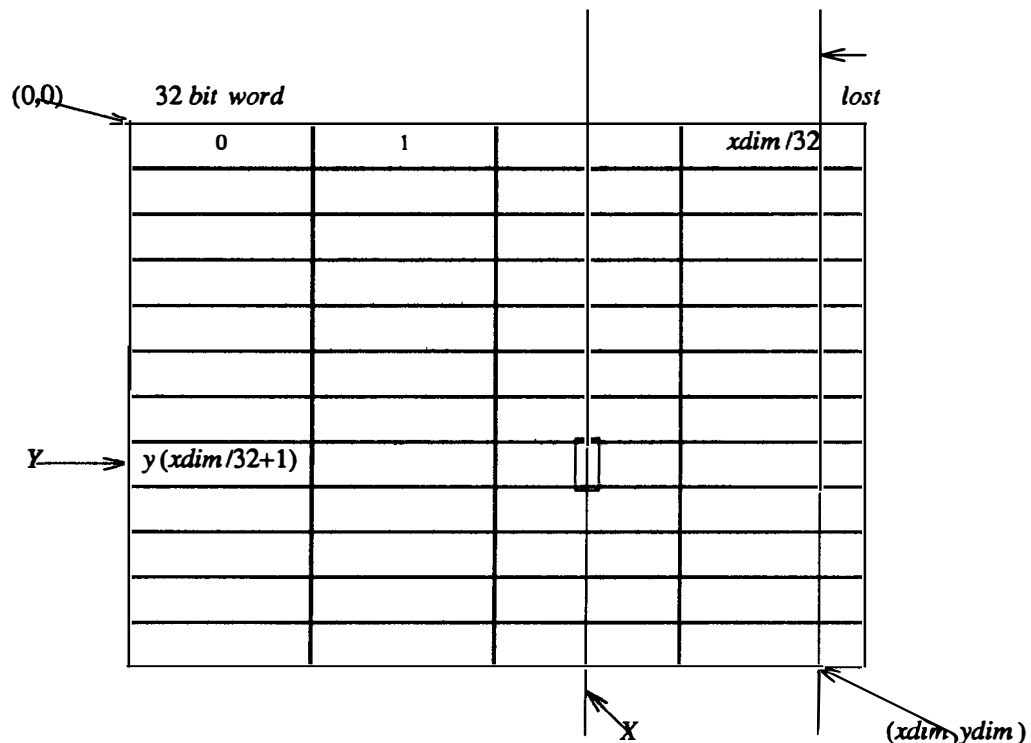


Figure 6.3.5 : The bitmap with its datastructure

In figure 6.3.5 we see the bitmap that represents a $n \times m$ input image. To facilitate further calculations we have introduced two variables *xdim* and *ydim* who have respectively values $n-1$ and $m-1$. The upper left corner is taken as the origin (0,0). The image is overlaid with a raster containing rectangles. The most right rectangle on each horizontal strip of rectangles may exceed the image somewhat on its right side. Each rectangle resembles a 32-bit word, this means that there are $xdim/32+1$ such rectangles on the horizontal axis of the image. The extra space on the right side arises when $xdim+1$ is not a multiple of 32. The number 32 is used because the node CPU has 32-bit wide registers so it is possible to store 32 bits in a register at one

time and processing of all 32 bits can be done without having to access the RAM memory.

Each 32-bit word contains 32 pixels. A pixel has coordinates (x,y) . The *address* of the word that contains the pixel information of pixel (x,y) can be calculated as follows :

$$address = y \times (xdim / 32 + 1) + x / 32$$

The $/$ operator is actually the *div* operator. To determine the bit position of the pixel (x,y) we use the $\%$ operator that is the same as the *modulo* operator. This is depicted in figure 6.3.6 below.

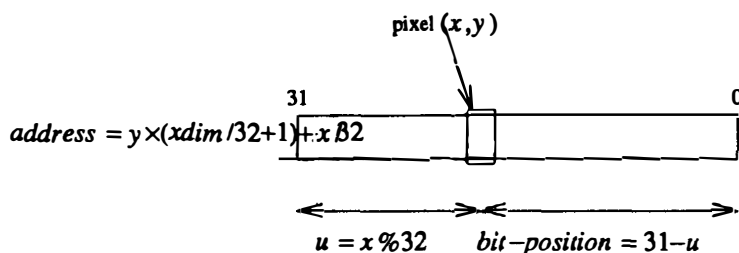


Figure 6.3.6 : Bit position of pixel (x,y) in a 32-bit word

The bit position of the pixel (x,y) in a 32-bit word is

$$bit = 31 - x \% 32$$

This calculation is, however, of no concern to us, because it suffices to rotate the word $x \% 32$ times to the left, thereby putting the relevant bit in the sign bit which we can directly test by testing if the entire word is negative or not. Viz. if the signbit is a one then the pixel is black, otherwise white.

The method described above is one in which pixel information can be randomly accessed, i.e. arbitrary pixels can be tested. In the Hough and Split algorithm it is very often not necessary to have random access but it is sufficient to have sequential access, viz. a construction as described in algorithm 6.3.2 below.

Procedure RANDOM-ACCESS ($xdim, ydim$)

Method

```

For  $y=0$  to  $ydim$  do
  For  $x=0$  to  $xdim$  do
    Let  $address = y \times (xdim / 32 + 1) + x / 32$ 
    Let register  $w$  = contents of  $address$  rotated  $x \% 32$  times to the left
    If  $w > 0$  then
      The pixel  $(x,y)$  is white
    else
      The pixel  $(x,y)$  is black
    endif
  endfor
endfor
endmethod

```

Algorithm 6.3.2: The procedure RANDOM-ACCESS scans the image by randomly accessing the bitmap structure.

The use of random access to pixels in algorithm 6.3.2 shows careless programming for each time a new coordinate is reached the entire calculation of *address* and *bit-position* is repeated. In algorithm 6.3.2 a better method is shown. That method uses all 32 bits that were accessed at the same time since they are positioned in the correct order. When these 32 bits have been processed it continues by incrementing the *address* with one so that *address* automatically points to the 32 bits that contain information about the next 32 pixels in the image.

Procedure SEQUENTIAL-ACCESS (*xdim*,*ydim*)

Method

```

    Let address=0
    For y=0 to ydim do
        For x=0 to xdim do
            Let register w= the contents of address
            Let i=0
            while register w ≠0 and x+i ≤ xdim do
                If w >0 then
                    Pixel (x+i,y) is white
                else
                    Pixel (x+i,y) is black
                endif
                Let i=i+1
                Rotate register w one place to the left
            Let address=address+1
        endfor
    endfor
endmethod

```

Algorithm 6.3.3 : Procedure SEQUENTIAL-ACCESS scans the image sequentially.

Furthermore algorithm 6.3.3 has a tremendous advantage namely that it only needs to shift the register *w* to the left until *w* has become zero which can be tested with one test. This means for example that only one test is needed to skip 32 white pixels !

We conclude this description of the bitmap datastructure by mentioning that the result of this intensive low level programming is twofold, namely:

1. The Hough and Split are both incredibly fast as can be seen in part three of this report and
2. The amount of storage that is needed to store a bitmap is good, each pixel is represented by one bit. If we discard squeezing techniques like octrees etc. this is optimal. We did not use any squeezing techniques because they slow down the access to the bitmap.

The second datastructure we shall discuss is the accumulator plane. The structure itself is not so complicated as is in the case with the bitmap but we shall focus on the size of the plane for an average character.

In figures 6.3.2 and 6.3.3 the plane is depicted. Each point refers to a cell that should be able to hold a number. The plane ranges in horizontal direction from $-RCMAX$ to $+RCMAX$ and in vertical direction from $-m - n \times RCMAX$ to $m + n \times RCMAX$. These ranges are a little oversized so that the cells in the plane can be more easily manipulated. The calculation of these ranges may come as a surprise but is roughly the same as the calculation that belongs to figures 6.3.1 and 6.3.2.

To get an idea of the amount of memory that is occupied by the accumulator planes we take an average character of say 20×30 pixels. For the *RCMAX* variable we take 2, this adds up to a total of

$$(2 \times 2 + 1) \times (2 \times (30 + 2 \times 20) + 1) =$$

$$5 \times 141 = 705$$

numbers for the *XY* accumulator. If we assume that a number fits in a two byte unsigned integer then this comes to 1.4 Kbyte. The same calculation holds for the *YX* accumulator except the 20 and 30 are swapped and comes to 1.6 Kbyte. So all together the workspace for our algorithm requires 3 Kbyte of RAM memory for processing one character.

This concludes our description of the algorithm and datastructures involved with the Hough transform. Final implementations were done in C and can be found in Appendix E.

6.3.3. The Hough function interface description

The Flisp function *hough* takes as three arguments and can be used to apply a Hough transformation to a selected area in the loaded image.

The first argument of *hough* is a list that describes an area, e.g. on of the areas that was returned by *split*.

The second argument is an integer value that is assigned to the *LLENGTH* variable that is the minimal length of a line segment if it is to be detected by the Hough transform.

The third argument is a value that is assigned to the *RCMAX* variable that denotes the maximal slope that can be detected.

The output of *hough* is a list that contains line-segments. A line segment is represented by a list containing five integers, the first two represent the coordinates of the start point, the third and fourth integer representing the coordinates of the end point and the fifth integer is the line length.

In the following example the first area that is returned by *split*, is take as the first argument to *hough*. The *hough* function call then detects all line segments in this area with minimal line length 1 and maximal slope 1.

Example 3.1 :

```
->(define area (car (split '(0 0 223 30))))
area
->area
(206 4 211 12)
->(hough area 1 1)
((5 0 5 1 1) (0 6 0 8 2) (5 1 0 6 7) (0 7 1 8 1) (4 0 5 1 1)
 (0 8 5 8 5) (0 0 5 0 5))
->
```

In figure 6.3.7a and 6.3.7b below, the original bitmap input and the vector output of the *hough* function respectively are depicted.

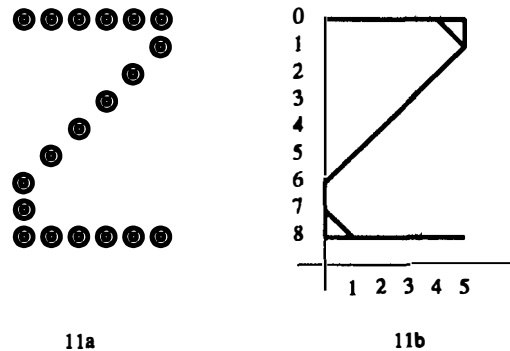


Figure 6.3.7: (a) Original bitmap corresponding with the digitised input image, and (b) The vector output of the *hough* function.

This example concludes the description of the Hough algorithm, its datastructures and the Flisp *hough* function.

6.4. GRAPH : CONNECTIVITY BASED ALGORITHMS

The idea of implementing Graph algorithms and building them into Flisp originated when the connectivity in a vector representation of a character became an important issue for classifying the characters. It is of course possible to match line segments in Scheme with each other but this done very slow since it requires recursive list traversals and many comparisons of integer pairs that represent coordinates.

In terms of a Graph we speak not of line-segments but of *edges* and not of its two endpoints but of *vertices*. A finite graph $G = (V, E)$ consists of a finite set V of *vertices* $V = \{v_1, v_2, \dots\}$ and a finite set E of *edges* $E = \{e_1, e_2, \dots\}$. To each edge there corresponds a pair of vertices: if the edge (v, w) corresponds to edge e , then e is said to be *incident* on vertices v and w , whereas vertex v and w are *adjacent* to each other. When we draw a graph G , each vertex is represented by a dot and each edge is represented by a line segment connecting its two end vertices.

The Graph algorithms that are presented in this chapter should be considered as a composing technique in the *Preprocessor*, versus Split that is clearly a decomposing technique. For example *cycles*, or loops, in a graph are *composed* of edges and as such an algorithm that detects such *cycles* is a composing algorithm.

In Section 6.4.1 the implementations of some of the graph algorithms are discussed, in Section 6.4.2 the resulting Flisp functions are described.

6.4.1. Implementation of a Graph environment

The term environment is used because we have chosen to create an environment that contains all graph information. This information can only be directly accessed by the graph functions.

The environment containing all vital graph information, should be compared with a clipboard which can only hold one graph at a time. All information about a graph is kept on the clipboard until a new graph is put on it. This construction was chosen because from the point of view of memory usage it is impractical to enable storage of an unlimited numbers of graph descriptions in RAM memory. The number of graphs may be large in our system since each area that was split from the image has a vector representation. As processing of characters on one node is done sequentially this means that storage needed for their graph representation can be used again when a new character is being processed. This method fits exactly in the clipboard concept.

To represent a graph we need a connection matrix, an array that contains coordinates of each vertex and some other datastructures that are used in the algorithms that operate on the connection matrix. Since we start with a vector representation we first need a function that transforms the vector representation into data

that can be stored in the connection matrix. This function is performed by the *edge-to-vertex* algorithm, described in algorithm 6.4.1 below.

```

Procedure EDGE-TO-VERTEX (list)
    list contains line segments, viz. a result of the hough function.
Method
    Let  $n=0$ 
    Initialize the connection matrix
    For all lines in list do
        Let  $H$  = starting point of line
        Let  $B$  = ending point of line
        If there exists an  $i$ ,  $0 \leq i < n$  such that  $H = \text{vertex}[i]$  then
            Let  $h$  = vertex number of  $H$ 
        else
            Let  $h = n$ 
            Let  $n = n+1$ 
            Let  $\text{vertex}[h] = H$ 
        endif
        If there exists an  $i$ ,  $0 \leq i < n$  such that  $B = \text{vertex}[i]$  then
            Let  $b$  = vertex number of  $B$ 
        else
            Let  $b = n$ 
            Let  $n = n+1$ 
            Let  $\text{vertex}[b] = B$ 
        endif
        Let  $\text{connection}[h,b] = \text{TRUE}$ 
        Let  $\text{connection}[b,h] = \text{FALSE}$ 
    endfor
endmethod

```

Algorithm 6.4.1 : Procedure *EDGE-TO-VERTEX* transforms a vector representation into an undirected graph.

Now we have a powerful tool with which we can analyse the structure of a graph. There seven functions that interpret the connection matrix. Six of these functions are straightforward and are not described to their full extend. We only mention them here with a short description of their function. The seventh is a more complicated algorithm that finds all cycles in a graph and will be discussed later on in this paragraph.

The six algorithms are:

Procedure *CLEAR-GRAPH*

clears the connection matrix;

Procedure LIST-ADJACENT (v)

takes as input a vertex v and returns in a list all vertices w that are *adjacent* to v , thus $connection[w, v]$ or $connection[v, w]$ is true;

Procedure GET-VERTEX (v)

takes as input a vertex v and returns in a list the coordinates of the vertex, equal to $vertex[v]$;

Procedure ADJACENT(v, w)

takes as input two vertices v and w and returns the truth value of $connection[v, w]$ or $connection[w, v]$;

Procedure STRUCTURE

has no input and returns a list $(l_0, l_1, \dots)$ with $l_i = (v_{k_0}, v_{k_1}, \dots)$ if and only if all v_{k_i} are adjacent to v_i ; and

Procedure GRAPH(l)

takes as input a list l of vertices $(v_{k_0}, v_{k_1}, \dots)$ and returns the vector representation of the subgraph G' with $V' = \{v_{k_0}, v_{k_1}, \dots\}$ and with $E' = \{e_{m_1}, e_{m_2}, \dots\}$ such that all $e_{m_i} \in E$ and $e_{m_i} = (v_i, v_j)$. The vector representation of G' is then $((vertex[w_0]vertex[w_1])(vertex[w_2] \dots))$ if and only if there exists an $e_{m_i} \in E'$ that is (w_{2n}, w_{2n+1}) .

The algorithm that finds all cycles in a graph is based on an algorithm that is described in the book "Combinatorial Algorithms" by Edward M. Reingold, Jurg Nievergelt and Narsing Deo [E.M. Reingold, et al., 1977] (pp. 348-352). The algorithm as we have implemented it in C is has remained the same with exception of two things. Firstly the algorithm that is described in the book uses a stack to keep intermediate results, we have implemented this with arrays for speed up reasons. Secondly we eliminate all cycles that contain only two vertices. Such cycles can be generated because of the fact that the algorithm operates on a directed graph. A cycles is always generated twice since it can be traversed in two directions.

The number of cycles in a directed graph can be exponential in the number of vertices. In fact, the complete directed graph of n vertices and $n(n-1)$ edges contains

$$\sum_{i=2}^n \binom{n}{i} (i-1)!$$

cycles of length i for a total of

$$\sum_{i=2}^n \binom{n}{i} \times (i-1)! > (n-1)!$$

cycles. Thus we can hardly expect an algorithm to be polynomial in the size of the input. We can, however, try to develop an algorithm that generates the cycles in time $O(|V| + |E|)(c+1)$, where V is the collection of vertices, E is the collection of edges and c is the number of cycles in our directed graph. Such an algorithm would be close to optimal, since the time taken just to print the cycles must be $|I| \times c$, where $|I|$ is the average cycle length, which may expected to satisfy $|I| = O(|V|)$.

This discussion about the complexity of the algorithm is copied from "Combinatorial Algorithms" pp. 348. We found it a valuable contribution to the problem description of finding all the cycles which we did not want to deny the reader. In the following alineas the reader is given an impression of the basic principle of the algorithm.

The cycles in a directed graph can be generated by a depth-first search in which edges are added to path until a cycle is produced. If the backtracking is carelessly done, many more paths than necessary will be explored and the efficiency of the algorithm will suffer. Careful pruning of the search is necessary in order to endure that every cycle is generated exactly once and that very little extra work is done.

In order to ensure that each cycle generated is new and not simply a permutation of a cycle generated earlier, we assume that the vertices are represented as integers from 1 to $|V|$, and will consider each cycle to be rooted at its smallest vertex. We begin the search at a vertex s and build a (directed) path $(s, v_1, v_2, \dots, v_k)$, such that $v_i > s, 1 \leq i \leq k$. A cycle is reported only when the next vertex $v_{k+1} = s, k > 1$. After generating the

cycle $(s, v_1, v_2, \dots, v_k, s)$ we explore the next edge going out of v_k . If all edges going out of v_k have been explored, we back up to the previous vertex v_{k-1} and try extending a path from it and so on. This process is continued until we finally try to back up past the starting vertex s ; all cycles containing vertex s have been generated. This process is repeated for $s = 1, 2, \dots, |V|$.

To prevent traversing cycles that originate at a vertex v_i during the search for cycles rooted at s , it is necessary to indicate that all the vertices on the current path (except s) are unavailable as extensions of that path. A vertex v will be marked as not available by setting $available[v]$ to *false* as soon as v is appended to the current path. The vertex v will remain not available at least until we have backed up past v to the previous vertex on the current path. If the current path up to vertex v did not lead to a cycle rooted at s , v will remain unavailable for a while even after we back up past it. This prevents searching for cycles in parts of the graph on which such searches have been previously unsuccessful.

As the search progresses, the vertices on the current path are stored in an array called *path*: the variable *route* is equal to the number of vertices in the current path and points to the next free array element in *path* where the next vertex that is used to extend the *path* will be stored in. The vertex that was last added to *path* can be removed by decrementing *route* with 1. In array *unavailable* [v, w] a record is kept of the predecessors of unavailable vertices that are not in the current path:

$$\begin{aligned} &unavailable[w, v] = \text{true, if and only if} \\ &\quad v \in V \text{ and } (v, w) \text{ in } E \text{ and} \\ &\quad available[v] = \text{false and there is no} \\ &\quad i : 0 \leq i < route \text{ such that } path[i] = v \end{aligned}$$

In order to make a vertex u available again after it has been unavailable, we must change $available[u]$ into *true*. However, if u becomes available, so do all its unavailable predecessors that are not on the current path, since a new cycle containing u might exist that passes through one of the previously unavailable predecessors of u . Thus all the elements $unavailable[u, w]$ that were *true* should become *false* and for all those w who are not available, $available[w] = \text{false}$ should become *true* and the unavailable predecessors of w should be unmarked in the same way and so on.

The entire algorithm is represented by three algorithms: 6.4.2, 6.4.3 and 6.4.4. Algorithm 6.4.3 is a recursive procedure to do the depth-first search for cycles. This procedure calls the *unmark* procedure that is described in algorithm 6.4.2. To complete the algorithm the search for cycles is repeated for $s = 1, 2, \dots, |V|$ which is done by the procedure described in algorithm 6.4.4 that also takes care of the necessary initializations.

Procedure UNMARK(u)

Method

```

Let available [ $u$ ] = true
For all vertices  $w \geq s$  with unavailable [ $u, w$ ] = true do
    Let unavailable [ $u, w$ ] = false
    If available [ $w$ ] = false then
        Call UNMARK( $w$ )
    endif
endfor
endmethod

```

Algorithm 6.4.2: The recursive procedure *UNMARK* to make vertices available again as extensions to the current path in *CYCLES*.

Procedure CYCLE(*v*, *flag*)**Method**

```

    Let path[route] = v
    Let route = route + 1
    Let available[v] = false
    Let flag = false
    Let result-list = empty
    For all adjacent vertices w ≥ s of v do
        If w = s then
            Add the cycle (path[0], path[1], ..., path[route - 1]) to the result-list
            Let flag = true
        else
            If available[w] = true then
                Call CYCLES(w, g)
                Let flag = flag or g
            endif
        endif
    endfor
    If flag = true then
        Call UNMARK(v)
    else
        For all adjacent vertices w ≥ s of v do
            Let unavailable[w, v] = true
        endfor
    endif
endmethod

```

Algorithm 6.4.3: The recursive procedure *CYCLE* to do depth-first search for cycles. The procedure *UNMARK* is given in algorithm 6.4.3

Procedure MAIN**Method**

```

    Let route = 0
    For s = 0 to number_of_vertices - 1 do
        1   Let all available [...] = true
        2   Let all unavailable [...,...] = false
        3   Call CYCLES(s, f)
    endfor
endmethod

```

Algorithm 6.4.4: The driver program to generate all cycles of a directed graph using *CYCLES* and *UNMARK*.

This concludes the description of the implemented graph algorithms. C source code can be found in Appendix E.

6.4.2. Graph environment interface description

In total there are eight functions in the graph environment. In this paragraph their functional interface from Flisp is described. The examples that are used to exemplify the functions are based on figure 6.4.1a and 6.4.1b. Figure 6.4.1a depicts an undirected graph and 6.4.1b depicts a subgraph of the graph depicted in 6.4.1a.

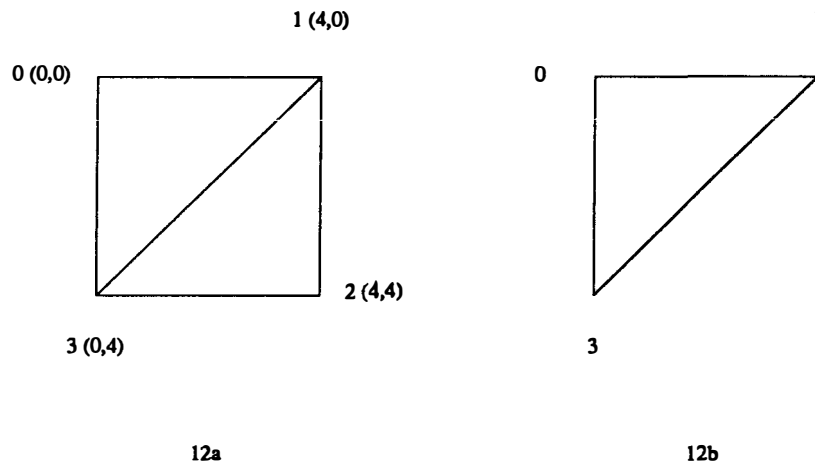


Figure 6.4.1: (a) An undirected graph G containing four vertices and five edges, and (b) the subgraph G' of G consisting of nodes 0, 1 and 2.

1. (*adjacent* $v_1 v_2$)

Adjacent returns *t* if v_1 and v_2 are adjacent vertices, otherwise *nil* is returned. For example :

```
->(adjacent 0 1)
t
->(adjacent 0 2)
nil
```

2. (*clear-graph*)

Clear-graph always returns *t* and initializes the clipboard datastructures.

3. (*cycles*)

Cycles takes no argument and returns in a list all cycles that are present in the current graph on the clipboard. For example :

```
->(cycles)
((1 3 2) (1 2 3) (0 3 2 1) (0 1 2 3) (0 3 1) (0 1 3))
```

4. (*edge-to-vertex list*)

Edge to vertex takes *list* as an argument that contains line segments, that are a vector representation of a graph *G* that will be initialized on the clipboard by this call. The number of vertices in the graph is returned. The following example initializes the graph from figure 6.4.1a on the clipboard :

```
->(edge-to-vertex ' ((0 0 4 0) (4 0 4 4) (4 4 0 4) (0 4 0 0) (4 0 0 4)))
4
```

5. (*graph list*)

Graph takes as an argument a *list* containing vertices that are supposed to be part of the current graph. A list is returned that contains all line segments that belong in the subgraph from the current graph by excluding all vertices from the current graph that are not in the argument list. The following example describes how the subgraph in figure 6.4.1b can be generated :

```
->(graph ' (0 1 3))
((0 0 4 0) (4 0 0 4) (0 4 0 0))
```

6. (*list-adjacent v*)

List-adjacent returns a list containing all vertices that are adjacent to *v*. For example :

```
->(list-adjacent 1)
(0 2 3)
```

7. (*structure*)

A list containing the entire graph structure is returned. The list contains lists $l_0, l_1, \dots$. The list l_i contains all vertices that are adjacent to vertex *i*. For example :

```
->(structure)
((1 2) (0 2 3) (1 3) (0 1 2))
```

8. (*vertex v*)

Vertex returns the coordinates of vertex *v* in a list as is shown in the following example :

```
->(vertex 1)
(4 0)
```

These descriptions conclude the paragraph about the Graph environment and thereby the chapter about Preprocessing.

CHAPTER 7

THE CLASSIFIER

Functional Specifications and Implementation

ABSTRACT

In this Chapter we shall describe the ingredients needed for a classifier. The classifier will use three sorts of structured data, called the Topological Knowledge, the Statistical Knowledge and the Contextual Knowledge. The latter two will be used for obtaining hypotheses, whereas the first will be used for deducing a hypothesis. In case the hypothesis fails, we shall start a forward reasoning process to derive the most likely character.

7.1. BACKGROUND

A classifier has always been one of the basic features of Pattern-Recognition systems. The idea behind a classifier, in this context, is that each pattern has certain properties which can be used to determine pattern sets, or in our case character classes. The actual recognition of a pattern is the classification of the pattern into predefined character classes, based upon its detected properties.

First we determine the functional specifications of a classifier in general, then we shall these general specifications with the ones needed for our Optical Character Classifier. Finally, we discuss how the classifier is designed and realised, which data structures are used, and how the classification process is controlled.

7.2. A CLASSIFIER

7.2.1. A theoretical approach to a classifier

We state that the intuitive function of a classifier is:

to take an object, look at its properties and to relate it with all known classes which have some or all properties of the object.

Clearly we want to sharpen this function description to obtain a formal description of the computing and its data structures. Before giving an exact definition of the function performed by a classifier we have to examine the classes and properties on which this function will have to operate.

7.2.1.1. Classes and Properties

We define a class, C , as the set of objects $\{O_1, O_2, \dots, O_n\}$. Further we say a class C has the properties $\{P_1, P_2, \dots, P_n\}$, if and only if all objects $\{O_1, O_2, \dots, O_n\}$ have at least all the properties $\{P_1, P_2, \dots, P_n\}$. Since we defined classes as being sets, the relational operator *improper subset*, $\subseteq$, and the standard set-operators, *intersection*, $\cap$, and *union*, $\cup$, can be applied to the classes resulting in a new set, or even in a new class.

Take for instance class C_1 having m objects and properties $\{a_1, a_2, \dots, a_x\}$, and class C_2 having n objects and properties $\{b_1, b_2, \dots, b_y\}$.

Clearly $C_1 \subseteq C_2$, if and only if for all $o_i \in C_1 : o_i \in C_2$ and o_i has at least properties $\{b_1, b_2, \dots, b_y\}$, see also figure 7.2.1.

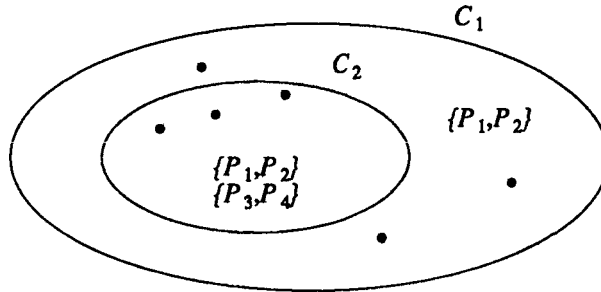


Figure 7.2.1: Class C_1, C_2 having respectively properties P_1, P_2 and P_3, P_4 .
Since $C_2 \subseteq C_1$, all objects $\in C_2$ have properties P_1, P_2, P_3, P_4 .

The class $C_3 = C_1 \cap C_2$ contains those objects of $\{C_1, C_2\}$ which all have the properties $\{a_1, a_2, \dots, a_x, b_1, b_2, \dots, b_y\}$, see also fig 7.2.2.

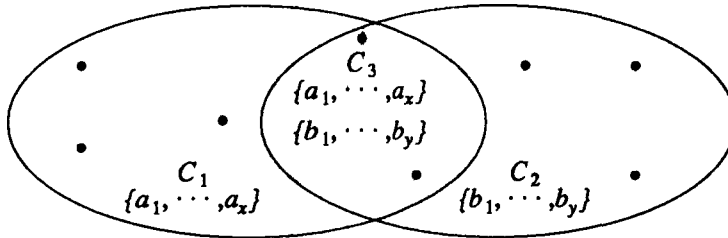


Figure 7.2.2: Class C_3 being the intersection of C_1 and C_2 .

Note that an object $\in C_3$ is also an element of C_1 and C_2 . Therefore an element belongs to a class if it has the properties of that class, but it is allowed to have additional properties.

This leaves us with the operator $\cup$. Straightforward, one could define $C_3 = C_1 \cup C_2$ as being the set of $\{C_1, C_2\}$ from which each element has the properties $\{a_1, a_2, \dots, a_x\}$ or the properties $\{b_1, b_2, \dots, b_y\}$ or both. But, then the question arises: what are the properties of C_3 ? Consider two objects $\in C_3$ each having mutually different properties, which is possible with the above definition of $\cup$, then we can not define a class, with certain properties, over these objects. So, C_3 is not a class.

An informal way to show this is to look at the intuitive function of a classifier, described above. There we see that the goal to be achieved, is to find the classes which have some or all properties of the input object. Therefore it does not make sense to allow classes of which the elements can have different properties. Hence, the operator $\cup$ applied on classes does not result in a class, the reason for excluding this operator.

7.2.1.2. The exact function of a classifier

One aspect we have not discussed yet, is the object which is the input of the function. The object has a number (possibly nil) of properties which are known. It may be possible to determine additional properties that belong to the same object. Look for example at fig. 7.2.3, where class A has the properties $\{P_1, P_2\}$, class B , being a 'subclass' of A , has the properties $\{P_1, P_2, P_3\}$, class C , also being a subclass of A , has the properties of A and $\{P_4, P_5\}$. Class D being the intersection of the classes B and C has the properties $\{P_1, P_2, P_3, P_4, P_5\}$ and finally class E , being a subclass of the intersection B and C , has properties $\{P_1, \dots, P_6\}$.

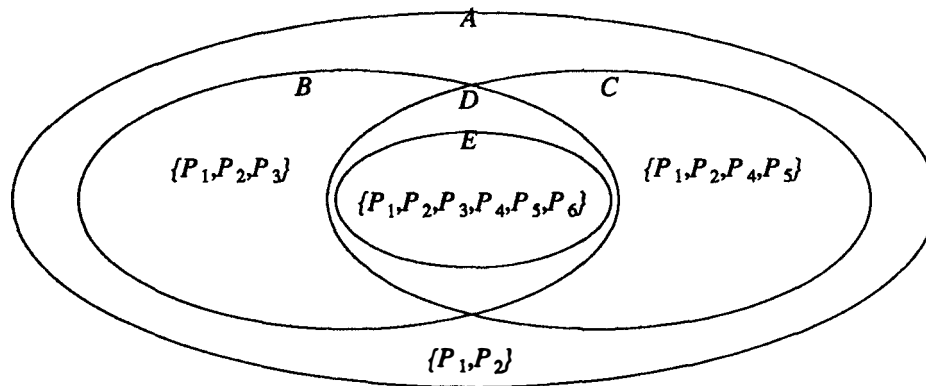


Figure 7.2.3: A structure of the sets A, B, C, D, E .

Now suppose our input is an object having the properties $\{P_1, P_2, P_3, P_4, P_5, P_7\}$, then we would expect, from our intuitive function description, that the object is classified in the class A, B, C or D , since all have some of the properties of the object.

Three remarks, about the intuitive function description, can be made from the example:

- The object has a property which is not known to the classifier using these classes, n.l. $\{P_7\}$. If we look at the definition of an object of a class we see that this object should have all the properties of the class, therefore the object is allowed to have more properties and still be an element of the same class. This explains the words *some or all* in our intuitive function description. More intuitively, one can see this as an *extra* property of the object, which effectively is an aid in describing the object itself. So, if the classifier is not acquainted with this property it is its own limitation, and therefore it can have no reasons for not letting the object be an element of the classes A, B, C, D, E .
- If we look at class E we see it has a property P_6 . The object does not have this property, that is, it is not known whether the object has this property. Therefore it seems not reasonable, in any sense, to let the object be a member of E . Thus we should expand the words *some or all*, in our informal definition, with *but none different*.
- Our last remark is a somewhat arbitrary one. If we look at the function, with the additions and restrictions suggested above, we would still get the output $\{A, B, C, D\}$. But since $(D = (B \cap C)) \subseteq A$ and we want to classify the object as strict as possible, we want the output D . So we are looking for the class which has the most properties of the object and no other ones.

So finally we can state what function a classifier algorithm should compute, with respect to the above definitions:

The function should output, on a given input, i.e. an object with the properties $\{P_1, P_2, \dots, P_n\}$, that class which has the most (i.e. the best) properties equal to $\{P_1, P_2, \dots, P_n\}$, and no other properties.

Finally we define the class *REJECT*, which has no properties at all. By this we achieve that objects, whose properties contradict every known class or are not known at all, are element of the class *REJECT*. By this definition, our function becomes total, i.e. for every input, being an object, there is an output (please read existing class).

7.2.2. Data needed for a Classifier

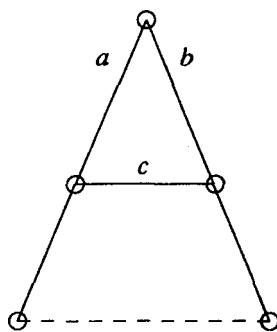
Above we have defined our classification function, now we can investigate how the input of the function is to be constructed. We discuss this in the context of an Optical Character Recognition system.

7.2.2.1. Skeleton Representation used as Knowledge

In Optical Character Recognition, we can use the *skeletons* to represent characters. With this representation, discussed in Part One, every character can be expressed in terms of spatial ordering and vertex/edge relations. The skeleton notation obtained from the input object (which is the set of vertices and edges returned by the Hough Transform), will be expressed in the following notation:

- **Explicit edges**, which are *horizontal*, *vertical*, *positive* or *negative* oriented, and have length *small*, *medium* or *large*. The length is relative to the largest and smallest line in the object.
- Every explicit edge has an internal orientation: *upper*, *middle* and *down* for vertical oriented edges, and *left*, *middle* and *right* for all other oriented edges. The *left*, *right*, *upper* and *down* orientations are called the *endpoints* of an explicit edge.
- **Implicit edges**, which are either *horizontal* or *vertical* oriented, are denoted by the endpoints (see the internal orientation of an edge) of two explicit edges.
- A **join** is denoted by the endpoints of two explicit edges.
- A **link** is denoted by the internal orientations of two explicit edges, of which exactly one is an endpoint.
- A **crossing** is denoted by two explicit edges.
- A **triple connection** is denoted by the internal orientations of three explicit edges, where each internal orientation forms a link or a join with the others.

Note that we have discarded the *meet* from the original skeleton notation. The reason for doing so is, that the input data may contain some inaccuracies and that therefore the endpoints of two lines (in the input) might differ slightly. So instead of a *meet* we use the *join* for denoting endpoint connected edges in our skeleton notation. In figure 7.2.4 an example of the notation is given for the character "A".



Explicit edges: (*long pos a*), (*long neg b*), (*medium hor c*)

Implicit edges: (*left a*, *right b*)

Links: (*middle a*, *left c*), (*middle b*, *right c*)

Joins: (*right a*, *left b*)

Tripple connetions: None

Crossings: None

Figure 7.2.4.: The Notation of the Character "A".

character, whether there is a crossing or whether there is one long line. Therefore we can supply the controller, with the notation for all characters. Obviously, by abstracting a notation from our input data, the classifier should be able to classify this notation as a character. By definition of the *skeletons*, the notation obtained should be valid for a large group of fonts, enabling the classifier to recognise a large group of fonts with just one notation.

The knowledge, about the notation of characters, which is expressed by property sets and the way classes are structured (using the properties obtained from the *skeleton* notation), will be called the **Topological Knowledge**

7.2.2.2. Structuring Topological Knowledge

The next question which arises is : "How can we combine the above mentioned properties to create distinguished classes?" The most direct way would be, to define a class for each character and give this class all the properties of the character. This would leave the classifier checking all the sets for all the properties of the given object until one set was found appropriate or else return the class *REJECT*. Clearly this is a terminating computation which satisfies our function description, but, as mentioned earlier, this may require the computation of all the properties. One can see that this is not really necessary. Suppose the character 'X' has the properties : 'one crossing, horizontal and vertical symmetry'. These properties actually describe the letter 'X', so for recognising it, we do not want to look at other properties than these three. Further more some characters share certain properties, so these properties could give us a partial solution. For example, by looking at the properties '1 vertical line, 1 surrounding' we can say that the object, having these properties is an element of the class {'b', 'd', 'p', 'q'}.

To avoid this we can store all the character properties, i.e. which characters share which properties, into a rule-based system [Hayes-Roth et al., 1983]. For example:

IF (object has property {one vertical line})
THEN (object $\in C_1$)

IF (object has property {one surrounding})
THEN (object $\in C_2$)

IF (object has property {one horizontal line})
THEN (object $\in C_3$)

IF (object has property {one crossing})
THEN (object $\in C_4$)

IF (object has property {horizontal symmetry})
THEN (object $\in C_5$)

IF (object has property {vertical symmetry})
THEN (object $\in C_6$)

IF (object $\in C_1$ and $\in C_2$)
THEN (object $\in \{ 'b', 'd', 'p', 'q' \}$)

IF (object $\in C_1$ and $\in C_3$ and not $\in C_4$)
THEN (object $\in \{ 'L' \}$)

IF (object $\in C_4$ and $\in C_5$ and $\in C_6$)
THEN (object $\in \{ 'x', 'X' \}$)

To derive new facts from such a rule-base, given a certain instance of data, we need a so-called *inference-mechanism*. We can distinguish two main *inference mechanism's* [Rich, 1983], n.l.:

- A *backward reasoning* mechanism, which takes a "THEN-part" of a rule as a hypothesis, and tries to derive this hypothesis by proving the "IF-parts" of that same rule. If the "IF-parts" are not known, they become hypotheses of a lower order. The mechanism will then have to look for a rule which has one of those lower-order hypotheses in its "THEN-part", and then try to prove the "IF-parts" of that rule. This process continues until all lower-order hypotheses are derived and the mechanism can conclude the hypothesis. If the mechanism fails to derive the lower-order hypotheses and can find no new lower-order hypotheses, it concludes that it can not derive the hypothesis.
- A *forward reasoning* mechanism, which tries to match the given data (i.e. facts), with the "IF-parts" of a rule. If all the "IF-parts" of a rule are matched we say the rule *fires*, the "THEN-parts" of that rule become facts, and the matching process starts all-over again. This continues until we either derive a final conclusion or the matching process fails, due to the lack of (new) facts.

If we subject the rule-base, given above, to an inference mechanism which tries to deduce the character 'L' by backward reasoning, we shall not have to investigate the properties {horizontal symmetry, vertical symmetry, surroundings}.

In the case of a forward reasoning mechanism, we shall have to build our rules so, that we effectively have a decision tree. In the nodes of this tree, the classifier is directed, by the values of certain properties, which class to enter next (read which properties to investigate next). Finally the classifier will enter a class which has but one object; the character.

In such a decision tree all classes which have class C_1 as a parent, effectively are subclasses of C_1 . The intersection of classes, which we defined above, can not be expressed in a decision tree, without redundant data. If we would like to use the intersection operator, we shall have to construct a graph, rather than a tree. To see this, take the example in figure 7.2.5, where C_i is the class with property P_i . In the case of a tree, class C_3 has to be denoted twice, whereas in the case of the graph only once.

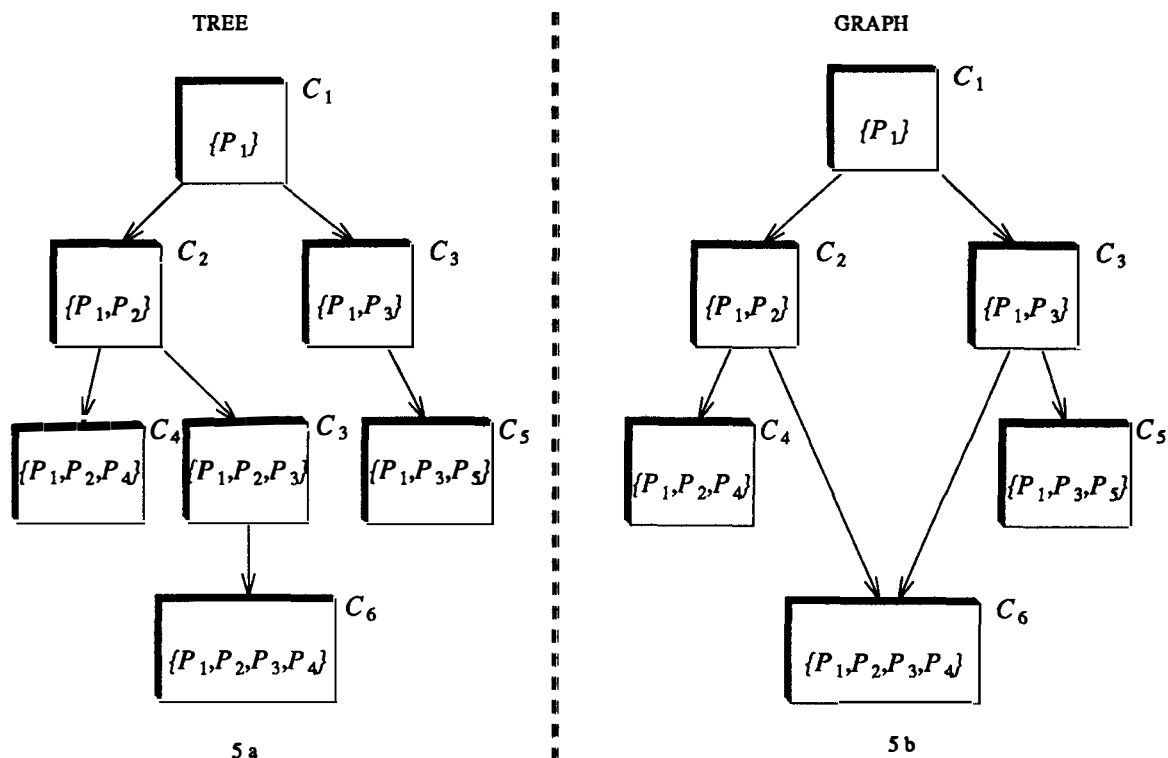


Figure 7.2.5: The intersection operator expressed in a tree (a) and a graph (b).

Once we have constructed this decision graph, the backward reasoning mechanism can find the necessary properties, by going straight up the graph, from the hypothesis to the top, and taking all the properties in the nodes it encounters on its way. Please note that there are several paths to be found by the forward- or backward reasoning process.

7.2.2.3. The Contextual Knowledge

As depicted in Section 7.2.2.2, recognising a character can be done by a forward- (which will have to look at quite some, for the solution, uninteresting properties) or by a backward reasoning mechanism (which takes the least properties for a character, but takes the risk of having a wrong hypothesis).

Clearly the data obtained by the Hough transform will invoke a forward reasoning process. This can be seen from the fact that we have to investigate the output of the Hough transform in order to derive a property. Which property this is, is determined by a forward reasoning process, and not by a backward reasoning process, since we do not have a hypothesis.

So the next question arises: "How can we obtain a hypothesis for the inference mechanism?"

Suppose the OCR system has been given a text, then the output of the Hough transform will be a set of sets of lines. The classifier then must try to classify each set of lines as a character. If we allow the classifier to keep track of its own history, i.e. what was the last character and the character before that one and so on, we can use the trigram analysis, described in Part I. For example, if the last two characters recognised were 'zz' then the next character to be recognised will almost certainly not be the character 'z', and therefore the trigram probability of the triple '(z,z,z)' will be *nil*. On the other hand, if the characters 'th' were the last to be recognised, then the trigram value of the triple '(t,h,e)' will be very high, and the hypothesis "the character to be recognised will be 'e'" seems very reliable.

Please note that we have found another source of knowledge, by using trigram frequencies, which enables us to guide the classifier in its process. We will refer to this knowledge as the **Contextual Knowledge**.

7.2.2.4. The Statistical Knowledge

In section 7.2.2, we have seen that it is possible to obtain the properties of a character from the *skeleton* notation. These properties are font independent. However, it might be useful to have some properties which are abstracted directly from the output of the Hough transform. For example the total number of lines found by the Hough transform might be different for a lot of characters, and thus indicating that the output contains a certain character.

These properties are font and character size dependent, and that we will need the values of the properties for each known font. We will refer to the values of these properties as the **statistical knowledge**.

7.2.2.5. The Interactive Use of the Preprocessor by the Classifier

Generally one can state that the task of a preprocessor is to abstract information from the input data. As discussed earlier, the Hough transform detects lines in pixel images.

One of the problems now is, how to interpret the lines, because some will be of real significance for a character, and others will be generated from the embellishments of the font.

Furthermore we can see that every character will have different kinds of embellishments.

The Hough transform (as implemented in our OCR system and as described in Chapter 6), has some parameters with which one can obtain certain subsets of the normal output. It is possible to specify the maximum slope and minimum length of the line segments that should be detected.

Hence the classifying process, might get additional information by applying the 'Hough transform' with parameters which seem to fit the class currently under investigation. Therefore we can give the classifier the possibility of applying the Hough transform interactively.

7.2.3. The Conceptual Model of a Classifier for our OCR system

We have indicated that our classifier will use four sorts of data:

- Topological knowledge,
- Contextual knowledge,
- Statistical knowledge, and
- New information about the object to be classified, by applying the 'Hough transform' with different parameters.

In order to control and use these sorts of knowledge and information, we need a controller which, led by some strategies, will combine these sources and come to a final solution.

In figure 7.2.6, the conceptual model of the classifier is shown. Together with the above defined function description, we are now ready to take a look at how all components will be constructed.

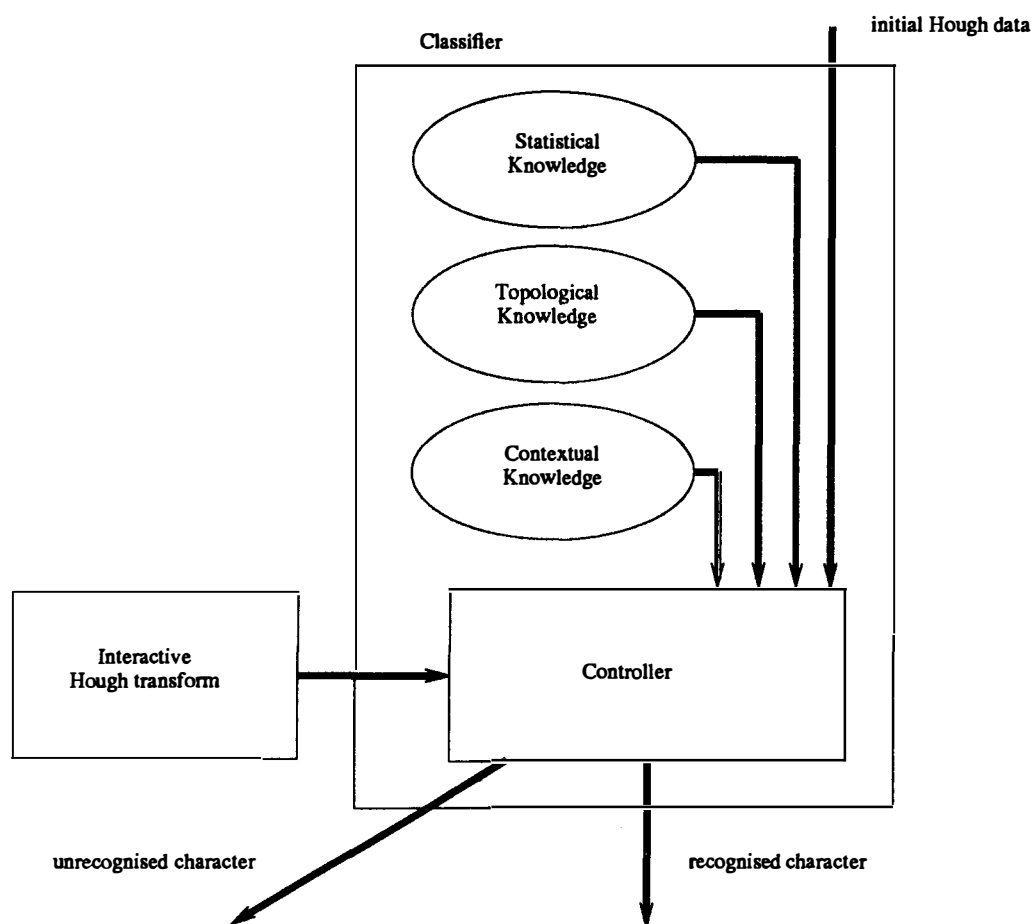


Figure 7.2.6: The conceptual model for the classifier in an OCR system.

7.3. THE TOPOLOGICAL KNOWLEDGE

The topological knowledge will be represented with the use of a decision graph. After defining the functional specifications of the decision graph, we need some standard functions which, operating on a data structure, will meet these specifications. The data structures used are the so-called *Frames*. The actual implementation of the frames and functions was done in *Scheme*; a LISP dialect.

7.3.1. Introduction

In this section we construct our knowledge base for the topological knowledge. Firstly, in Section 7.3.2, we state the functional specifications for this knowledge base. Next, we use the frame-concept, as described by Minsky [M. Minsky, 1975]. In Section 7.3.3 we investigate the implementation of these frames and the functions required for operating on them. Finally, in Section 7.3.4, we discuss the implementation of the functions, which will be used by the Controller.

7.3.2. Functional specifications

As shown above, we actually want to create a decision graph. This graph will have to perform two major functions:

- If, at a certain point, the controller has derived some properties (maybe none) of the object to be recognised, it will ask the decision graph to what class this object with these properties belongs. The decision graph, then, will search for a node, in its tree, which has the most properties, but none other, of the object (see also functional specifications of the classifier); assume that node is N_1 . Moreover, the decision tree will suggest what properties might be interesting to invest next, i.e. it enumerates the nodes (classes) of the graph, that are the children of the node N_1 . We will refer to this function as the *query* function.
- Secondly, when the Controller has a hypothesis it wants to deduce, it is up to the decision graph to enumerate all required properties. To achieve this the decision graph must find a path from the hypothesis to the top-level. There may exist several of these paths, since we are dealing with a graph, not a tree. We will refer to this function as the *trace* function.

An example of the *query* function is given in fig. 7.3.1, where the controller made a query for an object with properties P_1, P_2 . It will be suggested to either investigate P_3 or P_4 . Note that the decision graph will need some sort of *backtracking mechanism* to find the solution, and not to halt after investigating C_2 .

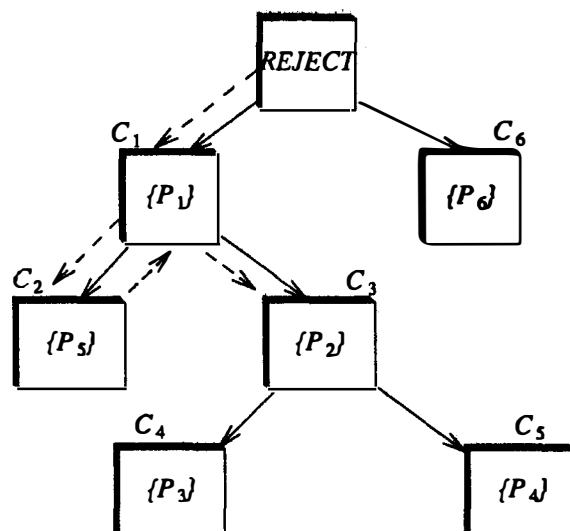


Figure 7.3.1: A query from the controller for an object with properties $\{P_1, P_2\}$.

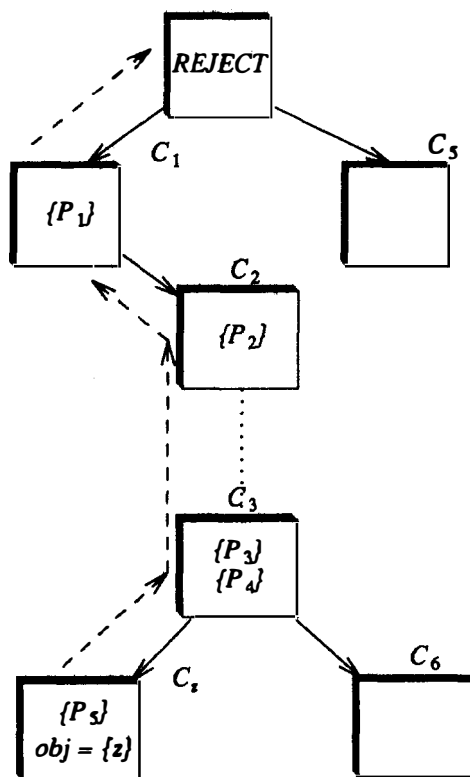


Figure 7.3.2: A trace of the character 'z'.

An example of the trace function is shown in fig. 7.3.2. The hypothesis of the controller being the character 'z', the decision tree looks for a path from the class C_z , which has but one element n.l. the character 'z', up to the class *REJECT*. The answer then is a list of all the required properties found on this path.

Obviously, it follows from these two specifications that we have to provide in some (depth-first) backtracking mechanism and therefore we shall have to make the structure of all the classes explicit. In the next section we shall look at the implementation of the decision graph with the use of *frames*.

7.3.3. Using Frames for Constructing a Decision Graph

In Part One, we have seen the basic concepts for frames as they were stated in [M. Minsky, 1975]. We repeat, for the sake of clarity, the most important features:

- Frames are data structures having the following form

```
(<frame-name>      (<slot1>  (<facet1>  <value1>)
                    <value2>
                    ...etc)
                    (<facet2>  <value1>))
                   (<slot2>  (<facet1>  ....)))
```

- Frames can be linked together in a hierarchical way by adding the slots *parents* and *childs*, which have for value a list of frame names. If we allow the slot *parents* to have but one value, the actual structure will be a tree. In the frame-concept this is called **single inheritance**. If we allow multiple parents, the structure will be a graph; **multiple inheritance**.
- One very powerful facet is the *If Added*. If a slot has this facet, and the value of this slot is changed, the function, which is indicated in the facet *If Added*, will be activated. For example this function might even alter the actual frame-structure.

7.3.3.1. The Implementation of Classes in Frames

Now, the implementation of our classes into frames is a very straightforward one. We take for each class a frame, so <frame-name> will become the name of the class. We have seen that classes consist of two things, n.l.

- Objects, and
- Properties.

Hence we give each frame representing a class, a slot *objects*, which has characters for values. So far we have not yet spoken about whether properties are predicates or have values. From the examples given earlier, we see that the most convenient method is to give properties a value which is one of the three sorts: Boolean, single valued or multiple valued. We can achieve this by introducing a facet *sort* which has one of the three following values: *Boolean*, *Multiple*, *Single*.

We give a frame, representing a class, a slot for each property of the class. These slots will have two facets, n.l. *values* and *sort*:

```
(property-name (values .....  
                (sort          Boolean/multiple/single))
```

Finally we add the slots *parents* and *childs*, where we allow *parents* to have more than one value, and thus obtaining multiple inheritance. This, actually is the allowing of the *intersection* operator, defined in 7.2.1.1.

Altogether, a class description, using frames will look like this:

```
(class          (parents  (value .....  
                        .....  
                        .....))  
                (childs   (value .....  
                        .....))  
                (objects   (value .....  
                        .....  
                        .....))  
                (property-1 (value .....  
                        (sort .....))  
                (property-2 (value .....  
                        (sort .....))  
                (etc.      )))
```

As described earlier, there is no need for explicitly denoting all the properties of a class. The only properties we want put in a node of the graph are the new ones for that class, thus not the properties of the classes of which this class is a subset. The same remark can be made for the objects of a class. So the only frames where we shall find values for the slot *objects* will be the terminal nodes of the graph. This might give the

idea of leaving out the slot *objects* for each frame being not a terminal node. We will leave them in, for the sake of uniformity.

Since we finally want to identify the object with a character, the terminal nodes of the graph become classes, inheriting quite some properties from the graph structure, with only one value for their slots *objects*, n.l. the ASCII-value of a character. Hence, there will be exactly one terminal node for every ASCII character.

In figure 7.3.3 an example of the decision graph, implemented in frames, is shown.

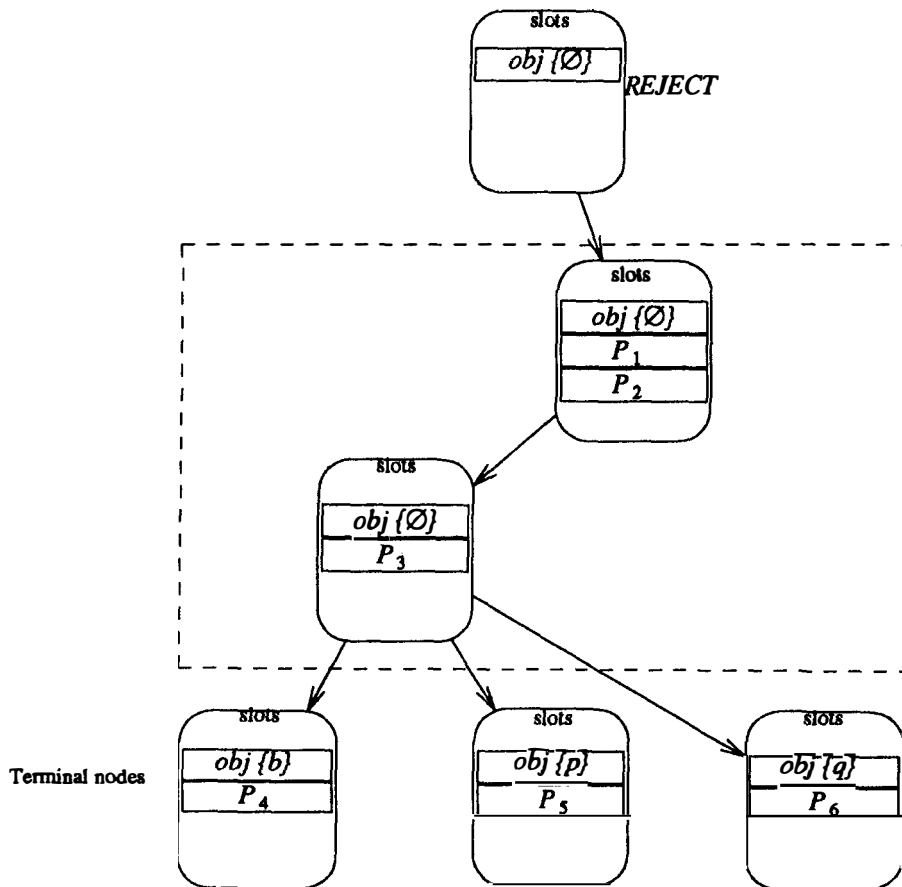


Figure 7.3.3: An example of the decision graph with terminal nodes 'p', 'b', 'q'.

In avoiding redundant data, we make use of the inheritance mechanism in two ways:

1. if we want to know all the properties of a class C_i , we take the properties of the associated frame plus all the properties found on all paths which lead from the frame up to the frame of the class *REJECT*,
2. if we want to know all objects of a class C_i we take all the paths from the associated frame which lead downwards to a terminal node. In these terminal nodes we find all the objects of C_i .

7.3.3.2. Implementation aspects and basic functions

As mentioned earlier, in [P.H. Winston et al., 1984] Winston suggests how the frame-concept can be implemented in Lisp. We have followed his suggestion by implementing our frames in Flisp; the Scheme interpreter described in Part One.

First of all, the implementation of a frame is done by using nested association lists. The required sublist can then be retrieved by supplying the standard function *ASSOC* with the key of the sublist.

Take for instance the frame *patient*, which is constructed of nested lists, and assigned to variable 'X':

```
(set x      (patient      (sex      (value      man))
                          (name      (value      Johnson))
                          (weight    (value      78)
                                   (if-added check-weight))))
```

Now the function:

```
(assoc patient x)
```

will return with the whole frame, since *patient* is the key. If we want to know something about the name of the patient, we apply the following function:

```
(cdr (assoc value (cdr (assoc name (cdr (assoc patient))))))
```

and get *Johnson* for answer. The reason for the *cdr* functions, is that an *assoc* returns the list (*key rest*). Taking the *cdr* of this list gives us (*rest*).

We will store every frame we define, as an element of the list *FRAMES*, which will be a global variable for all the functions working on the frames. So every frame will be accessible with an (*assoc frame-name FRAMES*) call. For handling the frames, we have build several functions, which are discussed below.

1. FGET_FRAME

Our first function to define is the *FGET_FRAME*. Its implementation follows directly from the above:

```
(define (FGET_FRAME frame-name)
  (cond
    ((assoc frame-name frames))
    (t (set-cdr! frames (append (cdr frames)
                                (list (list (frame-name))))
      (assoc frame-name frames))))
```

Note that the function *FGET_FRAME* creates an (empty) frame, with the name *frame-name* if the asked frame was not found.

2. FGET

We also want a general function *FGET*, which gives us the value of a certain facet of a slot of a frame. The function is implemented as follows:

```
(define (FGET frame slot facet)
  (cdr (assoc facet (cdr (assoc slot (cdr (FGET_FRAME)))))))
```

3. FEXTEND

For putting values in a facet, we shall have to ensure that the given path (*frame slot facet*) exist. Stronger, we shall create the path if it does not exist. Therefore, we need first of all, a function **FEXTEND** which has a key and a list as arguments. It first checks whether the key already is a key in the list. If not, it will assert the list (*key*) as the second argument of the list. The implementation in Scheme looks as follows:

```
(define (FEXTEND key a-list)
  (cond
    ((assoc key (cdr a-list)))
    (t (set-cdr! a-list (append (cdr a-list)
                                (list (list key)))))
    (assoc key (cdr a-list)))))
```

4. FOLLOW_PATH

Secondly we need a function **FOLLOW_PATH** which tracks down the path given and returns the list of values of the asked facet. This is done recursively until the end of the path is reached. To be sure that the given path exists, it will use the function **FEXTEND**;

```
(define (FOLLOW_PATH path a-list)
  (cond
    ((null? path) a-list)
    (t (FOLLOW_PATH (cdr path) (FEXTEND (car path) a-list)))))
```

5. FPUT

With these two functions we can obtain the list of values of a certain facet, in case the path to the facet does not exist, it is generated. Further more we do not allow a value to be inserted if it already is in the list. Note that in the implementation of the function **FPUT**, we use a local variable *value-list*. After the execution of **FPUT** this variable will become unbounded and therefore garbage. Normally we shall want to avoid the use of local variables for this reason, but if we do not add it, we shall have to call the function **FOLLOW_PATH** three times! So for the sake of speed we use a local variable. Eventually the function **FPUT** looks as follows:

```
(define (FPUT frame slot facet value)
  (let ((value-list (FOLLOW_PATH (list slot facet)
                                (FGET_FRAME frame))))
    (cond
      ((member value value-list) nil)
      (t (set-cdr! value-list (append (cdr value-list)
                                      (list value-list)))
         (value)))))
```


6. REMOVE

Finally we need two functions, one which removes a value from a facet, **REMOVE**, and the second one which will remove all the values of a facet. Naturally, if the given path does not exist we do not want to create it, so we can not use the function **FOLLOW_PATH**.

The function **REMOVE** is of the same structure as **FPUT**, where we delete the value only if the function `member` is true, i.e. if the value is in the value-list.

```
(define (REMOVE frame slot facet value)
  (let ((value-list (assoc facet (cdr (assoc slot
                                              (cdr (assoc frame frames)))))))
    (cond
     ((member value value-list)
      (set-cdr! value-list (cdr (delete value value-list))) t)
     (t nil))))
```

7. FCLEAR

The function **FCLEAR** is a relatively simple function, which sets the list of values of a facet equal to the empty list (`nil`):

```
(define (FCLEAR frame slot facet)
  (let ((value-list (assoc facet (cdr (assoc slot
                                              (cdr (assoc frame frames)))))))
    (set-cdr! value-list ())))
```

7.3.4. Implementation of the functions QUERY and TRACE

The functional specifications of the functions **QUERY** and **TRACE** were given in section 7.3.2. As we have seen, both functions use the inheritance mechanism of the frames. These functions will be the only ones used by the controller, and they are the only ones who use the basic frame handling functions, described in section 7.3.3.2.

For the sake of clarity, we shall not give the actual Scheme implementations of the functions, these can be found in the appendices, instead we provide the reader with pseudo-code.

The function **TRACE**, which returns all required properties for an object, will recursively try to find a path upwards until it has reached the class *REJECT*. Since we defined all classes to be a member of *REJECT*, the function can not get lost in a 'dead-end' as long as it keeps going upwards. While passing the frames on its way, it outputs all the slots which are properties. The 'going upwards' can be effected by entering a frame with the name that was found in the value list of the property *parents* of the previous frame. Remember that the terminal frames have but one object, so instead of doing a trace on an object you may as well give the name of the terminal frame of that object.

The pseudo-code looks as follows:

```
(TRACE frame)
IF frame = REJECT THEN RETURN nil
next := TAKE from actual frame a PARENT
RETURN the list (TAKE SLOTS PROPERTIES of actual frame,
                (TRACE next))
```

Algorithm 7.3.1: The function TRACE.

As mentioned earlier the function **QUERY** returns the frame name of the class, of which all the properties are equal to the properties known about the object. The Controller can then have a look at the required properties to go on in the graph; the properties of the childs of the returned frame.

This function will perform a depth-first search in the graph, starting at the level that is indicated in the function call. The latter is done, for not having to perform the whole search again, if a second query is made with the same properties and some new ones. The function **QUERY** will "continue" its search there where it had to stop the last time, (due to the lack of known properties of the object).

In pseudo-code the function looks as follows:

```
(QUERY frame known-properties)
TAKE all properties of the actual frame
IF all properties of the actual frame  $\in$  {known-properties}
    THEN WHILE (result = nil AND child exists)
        result := (QUERY child known-properties)
        child := TAKE NEXT child
    END_WHILE
    IF result = nil
        THEN RETURN frame
        ELSE RETURN result
    ELSE RETURN nil
```

Algorithm 7.3.2: The function **QUERY**.

7.4. THE CONTEXTUAL KNOWLEDGE

In this Section we shall describe the implementation of the trigram analysis in our optical character recognition system. Furthermore the method used to determine the frequencies is described.

7.4.1. Background

In the first part of this work, the trigram analysis was introduced in Chapter Four, "Optical Character Recognition". It has been shown that trigrams can be used to optimise the search process for new characters by first looking for characters which are most likely to appear in the scanned word. Besides this usage, the trigrams can as well be used in the postprocessing phase as a tool for error detection and spelling correction. In this part we shall outline in more detail the specific advantages, the expected search optimisation, the method to obtain the trigrams and the implementation of the trigram analysis in the Flisp interpreter as a special built-in function, written in C. Furthermore we shall describe how the trigram method can be fit into the classifier.

7.4.2. The Advantages of Trigrams

7.4.2.1. Trigrams in search optimisation

The main advantage of trigrams, is that they provide a good trigram frequency distribution in a specific kind of text, with a relatively small memory usage. On the other hand the values for the various trigram frequencies can be found immediately, whenever one uses the correct way of table building. Those were the main reasons why the trigram analysis has been selected as a method to optimise our search process. A structural classifier is implemented on each node, having only 512 kByte of memory available, to recognise approximately 3000 characters from a set that contains 80 different characters. This involves an average of $\frac{1}{2} \times 3000 \times 80 = 120,000$ symbolic manipulations on the level of character comparison. Knowing that one character comparison involves more than 10 low level symbolic manipulations, one can understand that there is a need for heuristics to optimise the search process. Building a trigram analysis table for 80 characters, we need $80^3 = 512,000$ table entries of one byte each, assuming that the maximum frequency is 255 (unsigned short or a char in C). Fitting this amount of values in 512 kByte local memory on a node leaves no more space to put anything else on the node. For this reason and because of the very low frequencies (mostly zero) of combinations of higher case letters, read markers, and brackets, we have decided to use trigram frequencies of a character set containing 27 symbols, 26 lower case letters and the space character, thus enabling us to squeeze a reliable frequency of a certain kind of text in $27^3 = 19,683$ bytes. With the use

of these trigrams we must be able to suppress the number of symbolic manipulations on the character comparison. Even when we have not yet recognised a character, we can use the trigram frequency analysis, for determining the letter with the highest probability to start a word (that is where the 27th character, the space, comes in). The specific values and determination of those values is discussed in Section 7.4.4 of this Chapter.

7.4.2.2. Trigrams Used in Spelling Correction

Trigrams can also be used for spelling correction, or better, for error detection. Suggestive evidence for the existence of a spelling error is created when a tri-letter sequence in a word has a very low frequency in the trigram frequency table. But, since our classifying system uses trigrams for classification purposes, it is of no use to detect spelling errors made by the classifier with the same trigram frequency tables. Therefore we have decided not to use trigram information as a means for spelling error detection.

7.4.3. Analysis of Trigram Frequencies

The trigram frequencies were obtained from two sources: the ICCA Journals Volume 9 and 10 (1-2), and the UNIX System Manual that was on-line. † The diverse files were put together with the use of the UNIX "cat" command:

```
cat * > words.icca
```

The file `words.icca` contains a lot of troff word processing commands. Then troff commands were deleted with the UNIX command: "deroff". The DEROFF option: "-w", puts all the words after each other on different lines.

```
deroff -w words.icca > icca.data
```

The next step is to translate the higher case letters to lower case letters, and to delete all the read markers, such as ', ', etc, because of the 27 character entries in the trigram tables. This was done with the UNIX command "tr" (translate):

```
tr A-Z a-z icca.data > icca.temp and
tr -cs a-z '\012' icca.temp > final_icca.data
```

Hence, we have a file containing 138,241 words for the ICCA Journal and a file containing 70,567 words for the UNIX System Manual, with each word standing on separate lines. The reason why we calculated the trigrams for two different text files, was to find out, whatever difference there exists between two different texts concerning computer science. One can understand that a computer-chess text recognised with computer-chess based trigrams, performs better (please read: faster) than a computer-chess text recognised with a system-manual based trigram frequency distribution.

Next the trigram frequencies have to be measured. A C program reads words from an input file, and generates the trigrams by shifting a three character large raster over the word (including the two preceding, and the two following spaces) and increasing the frequencies of the trigrams found within it. Because of the fact, that we had to update a lot of frequencies, we used the following hash table for the trigrams: a table of $27 \times 27 \times 27 = 19,683$ entries. The address of each trigram was obtained by the following formula:

† The ICCA Journals and the UNIX System Manual were on-line available on the Network File System (NFS) at Westmount Technology B.V., Poortweg 4, 2612 PA Delft, P.O. Box 5063, The Netherlands. Email gen@wmt.uucp.

$$\text{Trigram address} = \text{char}_1 + \text{char}_2 \times 27 + \text{char}_3 \times 27^2.$$

The formula assigns to each trigram a unique number and enables random access in the trigram table. After the table was filled, and the entire text was parsed, we found 904,904 trigrams. From all of the 19,683 possible trigrams to occur, there were only 4,587 not equal zero. The value of the non-zero trigrams was 198, the maximum, 28,159 (the trigram "the"), the standard deviation was 920, the variation 847. We made three output files:

- An output file that can be included in a C program in which an array containing the trigram frequencies, including the zero frequencies, is initialised. Below we shall explain how we were able to represent all frequencies with one byte.
- An output file which can be used by Symphony and Reflex, for obtaining more specialised information on the trigram distributions.
- An output file with the appropriate troff commands for a nice looking trigram frequency table in the appendices (Appendix B).

We can express the algorithm in pseudocode:

Procedure COUNT-TRIGRAMS

Method

```

While not EOF do
  Let  $\text{char}_i = \text{read\_in\_word}$  for  $0 \leq i < \text{length}(\text{word})$ 
  Let  $\text{char}_{-2} = \text{char}_{-1} = \text{SPACE}$ 
  Let  $\text{char}_{\text{length}(\text{word})} = \text{char}_{\text{length}(\text{word})+1} = \text{SPACE}$ 
  Let  $i = -2$ 
  While not End of Word do
    Let  $\text{address} = \text{char}_i + \text{char}_{i+1} \times 27^1 + \text{char}_{i+2} \times 27^2$ 
    Let  $\text{Trigram}(\text{address}) = \text{Trigram}(\text{address}) + 1$ 
    Let  $i = i + 1$ 
  endwhile
endwhile
endmethod

```

Algorithm 7.4.1 : Procedure COUNT-TRIGRAMS

Now, we shall first discuss the method used to squeeze the 16-bit integers, obtained from the frequency tables, into 8 bit chars, which can be used by the NCube system. Then we shall discuss the results obtained from the analysis with Symphony and the Data Base / Spreadsheet program Reflex.

To squeeze the 16 bit integers in a 8 bit value, we had to make a transformation from the area [0..28,159] to [0..255]. This transformation is realised by sorting all trigram values and then counting the number of different frequencies. For the ICCA Journal there turned out to be some seven hundred different frequencies. Each frequency F_i belongs to one or more trigrams if we discard the zero frequency. The index i in F_i thus ranges from 0 to about 700, for all $i: F_i < F_{i+1}$. These seven hundred different frequencies can be transformed into numbers ranging from 0 to 255 by grouping three subsequent frequencies, F_i , F_{i+1} and F_{i+2} together in group G_j with $j = i/3$ which is the new trigram frequency of F_i , F_{i+1} and F_{i+2} . The sorting of the 27^3 trigrams was done with a binary tree. The sorting algorithm is the same as is described in the Section 7.4.5 the FLisp *trigram* function.

The second problem was to extract relevant information from the trigram frequency distribution, in order to get some insight in the optimisation that can be achieved. This was done by translating the results to a Symphony internal representation. The calculations concerning maxima, average, standard deviation,

variance and the summation of the total amount of the trigrams parsed, were done in Symphony. For generation of cross reference tables, the file was also translated to a Reflex representation. Because of the relevance of letters, to start a word with, we first calculated the distribution of these frequencies for as well the ICCA-Journal as the UNIX System-Manual trigrams. The results can be found in figure 7.4.1, Starting Characters for the ICCA-Journal trigrams and in figure 7.4.2, Starting Characters for the UNIX System Manual trigrams.

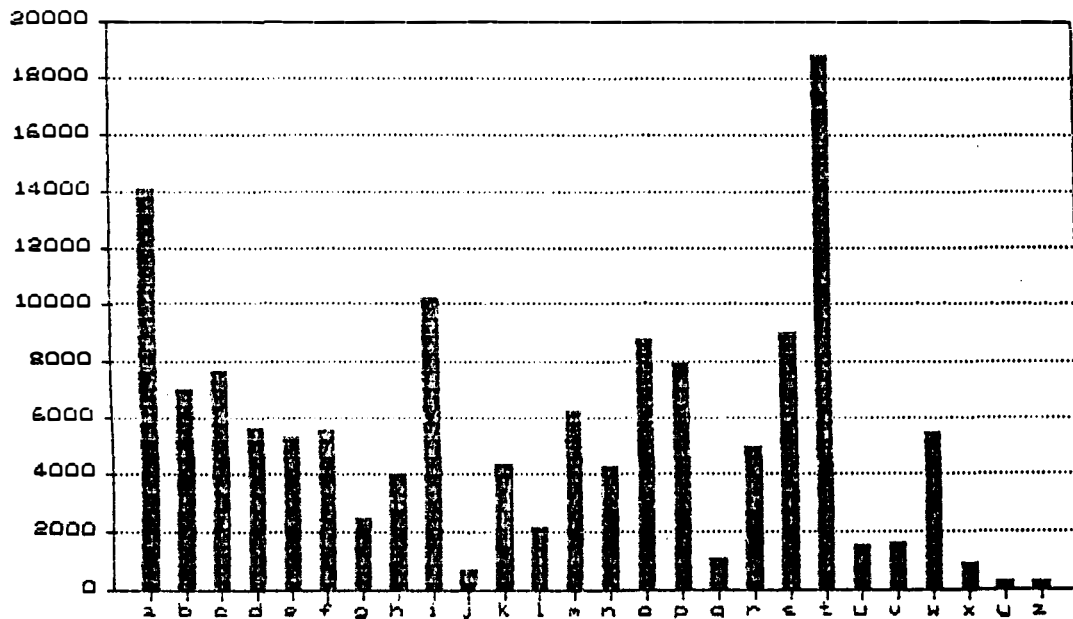


Figure 7.4.1: Starting Characters for ICCA-Journal Trigrams.

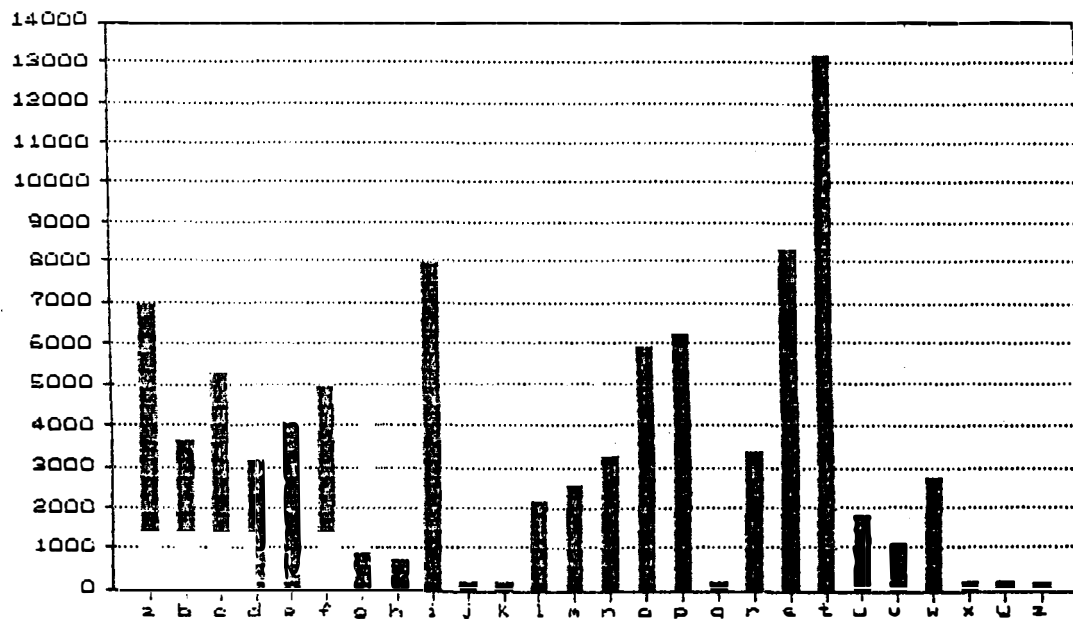


Figure 7.4.2: Starting Characters for UNIX System Manual Trigrams.

Although, we expected a difference in the trigram analysis for the different application texts, we can see that the trigrams do not differ very much in global layout. However, it must be said that there was a difference in more detailed trigrams, e.g., the trigram "ply" a strong computer chess related word, occurred in 0.05 % of all the ICCA-Journal trigrams and in only 0.001% of all the UNIX System-Manual Trigrams. In Reflex, the same analysis were made for the characters to end a word with. These results can be seen in figure 7.4.3 and 7.4.4.

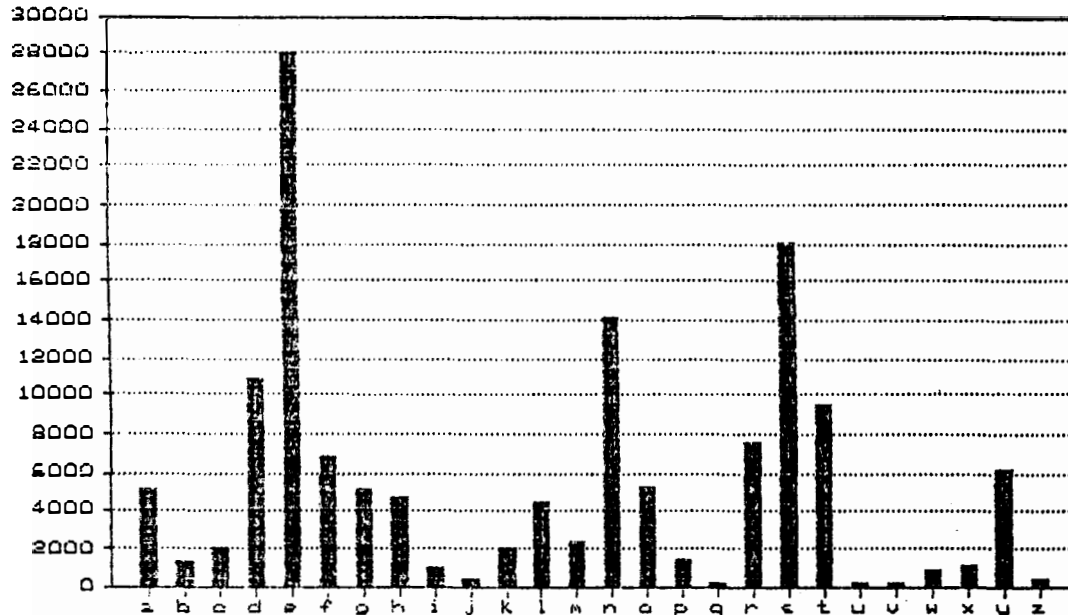


Figure 7.4.3: Ending Characters for ICCA-Journal Trigrams.

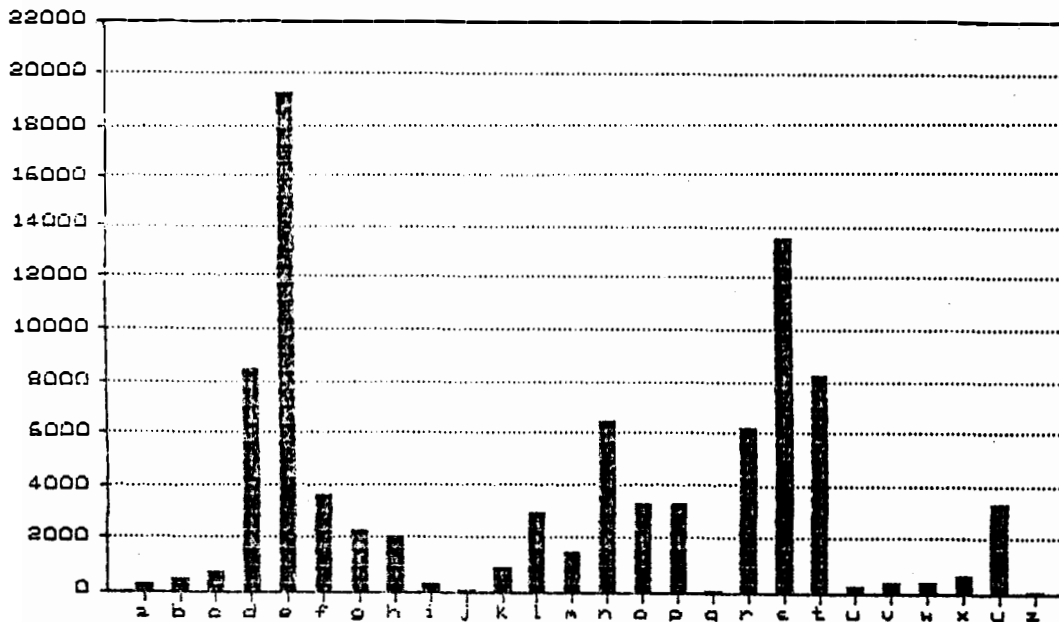


Figure 7.4.4: Ending Characters for UNIX System-Manual Trigrams.

As with the results obtained for the beginning characters, they do not differ a lot. Therefore, we made a more detailed data base study. We selected the character 'a' for further research. Suppose an 'a' is recognised, then, figure 7.4.5 and 7.4.6 show the distributions for the frequencies of the characters that can follow the character 'a'.

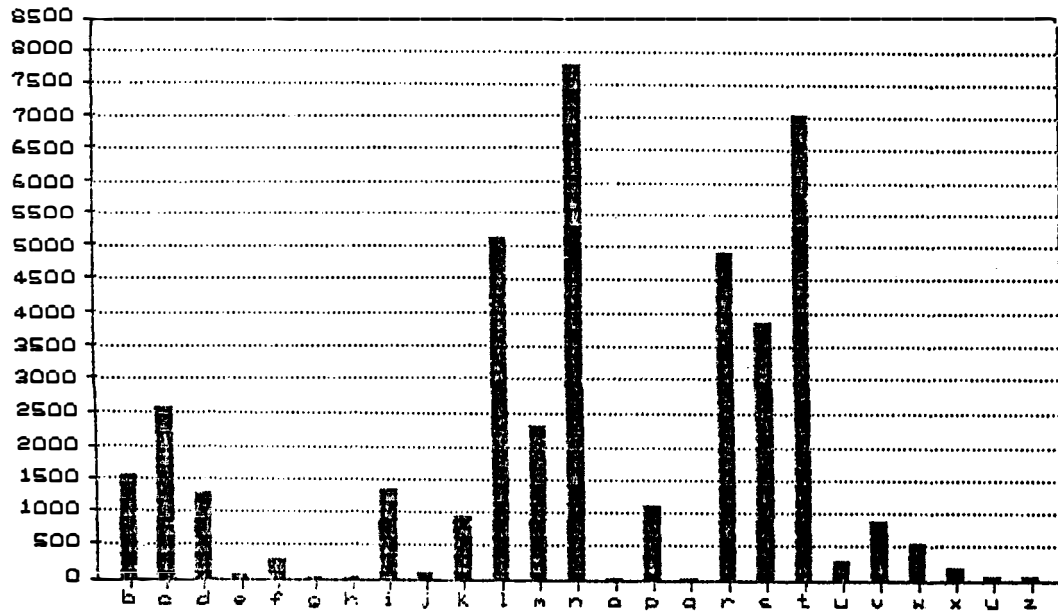


Figure 7.4.5: Characters Followed after an 'a' for ICCA-Journal Trigrams.

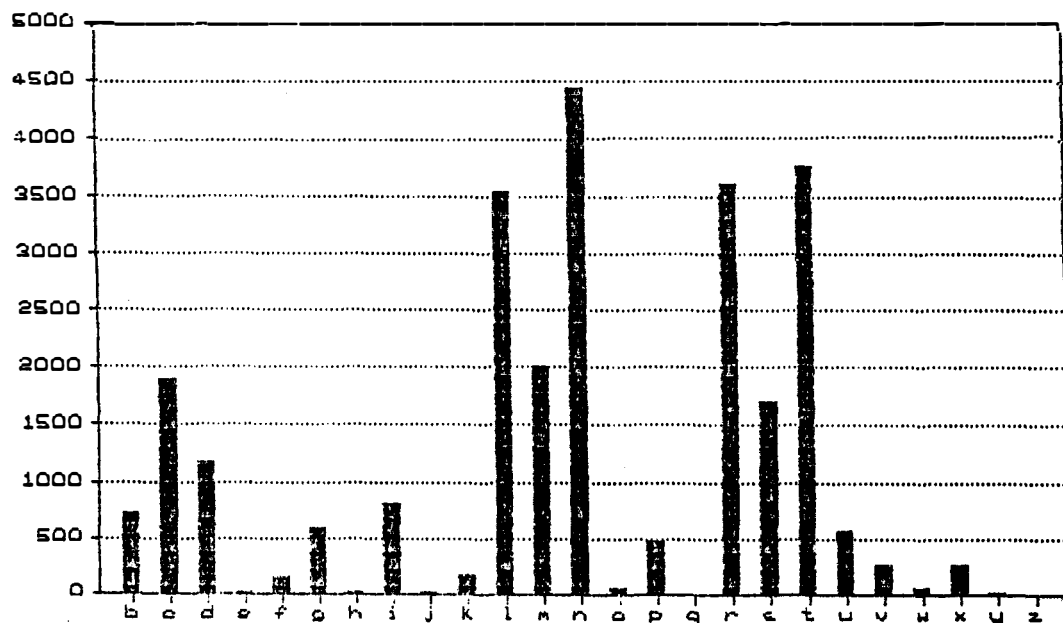


Figure 7.4.6: Characters Followed after an 'a' for UNIX System-Manual Trigrams.

Again, one can see that the differences are little. Interesting is that there are only 7 to 10 characters with a high distribution. In figure 7.4.7 and 7.4.8 the results can be seen for the distribution of the frequencies of the letters to follow on the combination 'at'. Again the difference are small, but now there are only three characters left with an interesting high frequency.

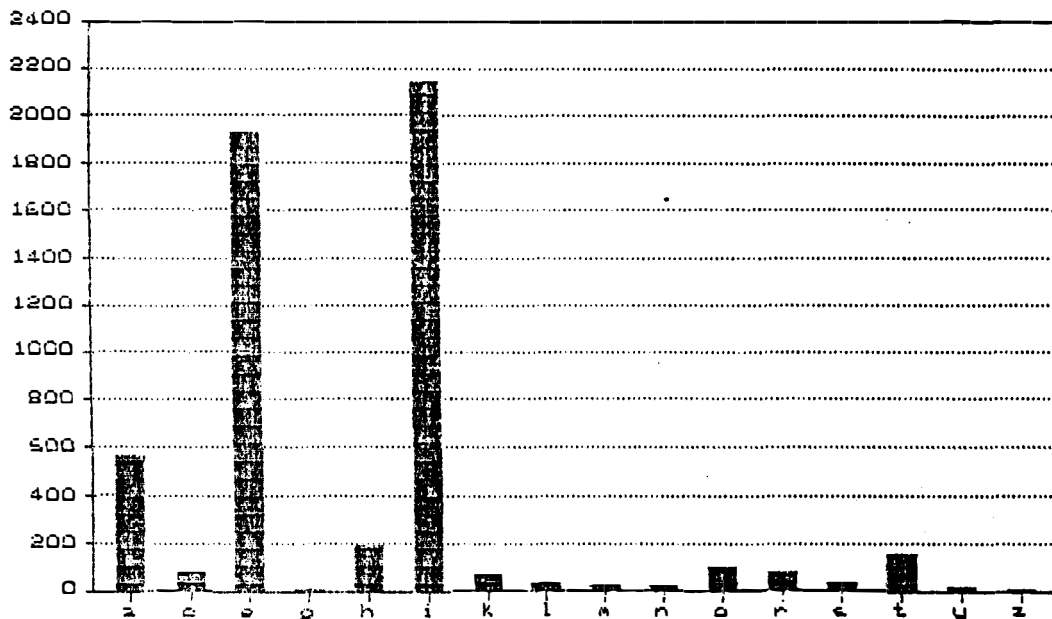


Figure 7.4.7: Characters Followed after 'at' for ICCA-Journal Trigrams.

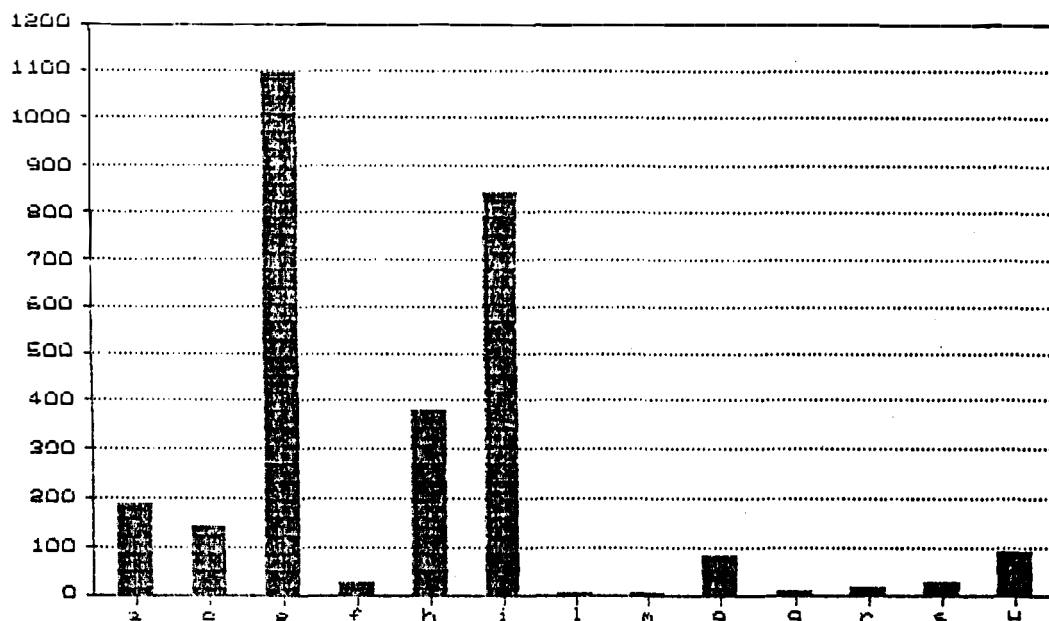


Figure 7.4.8: Characters Followed after 'at' for UNIX System-Manual Trigrams.

In the following part we shall discuss the results of the trigram analysis in search optimisation, in more detail.

7.4.4. Search Optimisation Obtained by Trigram Analysis

In the preceding parts, the use of trigram analysis in search optimisation was mentioned a number of times. In this part we will analyse the results in such a way that we have a better insight in the influence of the analysis for the search optimisation. We will split the discussion in four parts:

- The frequency of the characters to start a word with.
- The frequency of the characters to end a word with.
- The number of characters left after an 'a' recognised.
- The number of characters left after 'at' recognised.

If we keep the figures 7.4.1 to 7.4.8 in mind, we can say that the results for the ICCA-Journal and the UNIX System Manual don not differ enough to distinguish them in the NCube system. However, further research on this subject may be interesting with more different (because both trigram analysis are computer oriented) texts. Therefore we make no more difference between the two analysis.

As told above, the character to start a word with is mostly one out of 19. (the characters with a probability to appear less than 1% are left out). The cumulative percentage of these characters to appear as the first letter, is 97%. So in 97% we can speed up the process by looking for 19 characters, instead of 27. This is an average of 9.5 characters in the overall search process (see figure 7.4.1 and 7.4.2).

Now, suppose, we have recognised a word, but the last character, then there are 13 characters left to appear. The cumulative percentage of the characters is 92%. The average number of characters left to be recognised at the end of the word having a probability to appear of 92%, is roughly 6.5 (see figure 7.4.3 and 7.4.4).

In figure 7.4.5 and 7.4.6 the frequencies of the divers characters to appear after an 'a' can be seen. The characters left to be recognised is about 7. The cumulative percentage of these characters is 80%. If we add 4 characters to this group, the cumulative percentage lays around 98%. This brings the average number of characters to be a candidate back to 5.5.

The use of trigrams can be seen best in the frequencies of figure 7.4.7 and 7.4.8. This is the frequency table of the characters left to appear after 'at' recognised. The number has been brought back to three characters, with a cumulative probability of 95%. The average number of characters left be recognised after two characters is 1.5. This is summerised in table 7.4.1.

Trigram	Characters Left with Probability around 95
=	
Beginning	19
Ending	13
After 'a'	11
After 'at'	3

Table 7.4.1: Average Search

Supposing a text of 3000 characters contains 600, 5 letter words, we have a $\frac{1}{2} \times (600 \times 19 + 600 \times 11 + (600 \times 3) \times 3 + 600 \times 13) = 15,600$ comparisons left. This in contrary of the $\frac{1}{2} \times 3000 \times 26 = 39,000$ comparison needed for a search without trigrams. Hence, we can expect an search optimisation of 60%.

7.4.5. Sort and Permute Trigrams

The 27^3 trigram frequencies have been stored in a table that contains 27^3 bytes. The byte representation of a frequency was realised by squeezing the trigram frequencies which has already been described. This leaves us with the problem of interpreting the trigram frequencies and integrating this information in the classification process. The frequency table has been integrated with the classification system by implementing a *trigram* function in the Flisp environment that generates trigrams belonging to a given template, sorts these trigrams and returns the n trigrams with the highest frequencies. Functional requirements and implementation of this integration are discussed in the following Sections.

7.4.5.1. Trigram algorithm descriptions

A trigram is represented by three ordered symbols. Each symbol is either a lower case letter in the alphabet or a space. Let w be a trigram, then:

$$w = a_0 a_1 a_2 \text{ with } a_i \in Z = \{a, \dots, z, \text{space}\}$$

Let n be the index of the trigram frequency of w in the table, then:

$$w = a_0 a_1 a_2 \quad F_w = \text{table}[n], \text{ with } n = \sum_{i=0}^2 a_i 27^i$$

The Classification System needs to interpret the frequencies stored in the table and their corresponding trigram symbols. A single function that allows the Classification System to consult the frequency entry of a trigram is not very efficient. Better is a function that searches the frequency table for the most likely letter combinations which are then returned to the Classification System. Such a function would be given a *template*, which is a trigram, in which one or two symbols are unknown and that might be replaced by any symbol in Z . The fact that a symbol is unknown can be denoted with λ . A template t_w is then:

$$t_w = a_0 a_1 a_2 \text{ with } V = \{a_i \mid a_i = \lambda, 0 \leq i \leq 2\} \text{ and } |V| \leq 2 \\ \text{for all } a_i \text{ not in } V : a_i \in Z$$

By replacing the λ symbols in t_w , all valid trigrams w that match the template t_2 can be generated. For a template with $|V| = 1$ there are 27 permutations, in the case $|V| = 2$ there are $27^2 = 729$ such permutations. By sorting the generated trigrams with respect to their frequencies and deleting all trigrams that have a zero frequency we have a realistic result that gives us a good indication which symbols are most likely to be placed in the template. By returning only the n most likely trigrams that match the template, we ensure that the classifier is not overwhelmed by large lists of trigram permutations.

In algorithm 7.4.2 below the procedure *PERMUTE* is described. We assume that the elements in Z are numbered from $0 \dots |Z| - 1$. Note that $|Z| = 27$ and all a_i in t_w that are λ have number 0.

Procedure *PERMUTE* (a_0, a_1, a_2)

The template $t_w = a_0 a_1 a_2$

Method

Let $G_0 = 1, G_1 = 27, G_2 = 27^2$

Let $O_0 = 27-1, O_1 = (27-1) \times 27, O_2 = (27-1) \times 27^2$

Let $unbound[i] = \begin{cases} TRUE & \text{for all } i: a_i = \lambda \\ FALSE & \text{otherwise} \end{cases}$

Let $permutation = \sum_{i=0}^2 a_i G_i$

Label 1:

Let $f = table[permutation]$

If $f > 0$ then

Call *INSERT-IN-TREE* (*binary-tree*, $f, a_0 a_1 a_2$)

endif

For $i = 0$ to 2 do

If $unbound[i] = TRUE$ then

Let $a_i = a_i + 1$

If $a_i = 27$ then

Let $a_i = 0$

Let $permutation = permutation - O_i$

else

Let $permutation = permutation + G_i$

Goto label 1

endif

endif

endfor

endmethod

Algorithm 7.4.2 : Procedure *PERMUTE* finds all trigrams w that match template t_w and calls procedure *INSERT-IN-TREE* to insert non zero frequencies in a binary tree.

All trigrams that match the template are generated and if the trigram frequency is non zero then both the trigram and its frequency are added to a binary search tree. The binary search tree is maintained by the procedure *INSERT-IN-TREE* that is described in algorithm 7.4.3 below.

Procedure *INSERT-IN-TREE* (*tree*, *f*, $a_0a_1a_2$)

Method

```

If tree = empty then
    Let tree.frequency = f
    Let tree.trigram =  $a_0a_1a_2$ 
else
    If f > tree.frequency then
        Call INSERT-IN-TREE (tree.left, f,  $a_0a_1a_2$ )
    else
        Call INSERT-IN-TREE (tree.rigth, f,  $a_0a_1a_2$ )
    endif
endif
endmethod

```

Algorithm 7.4.3 :Procedure *INSERT-IN-TREE* inserts trigrams and their frequencies in a binary tree.

As follows from the description of algorithm 7.4.3 no trouble was taken to balance the binary tree. In the worst case $27^2 = 729$ elements have to be inserted in the tree. This implies a depth of 10 in a balanced binary tree. There are three reasons why we have left out balancing:

1. Measurements have shown that in most cases the tree depth does not exceed 15;
2. The storage mechanism used by algorithm 7.4.3 ensures that the storage of n elements in a binary tree is always the same, regardless its depth; and
3. From 2 it follows that the lack of balancing only causes search overhead. But since the sort of some hundred trigrams is performed in a split second there is no point in decreasing the search overhead.

Finally the procedure *TREE-WALK* that is described in algorithm 7.4.4 is executed. The procedure performs a recursive tree walk in depth-first order from left to right. When the n most likely trigrams have been passed, the walk is aborted. Variable n is supposed to have an initial value.

Procedure *TREE-WALK* (*tree*)

Method

```

If tree.left ≠ empty then
    Call TREE-WALK (tree.left)
endif
Let  $n = n - 1$ 
if  $n = 0$  then
    ABORT TREE-WALK
endif
Add tree.trigram to the result-list
If tree.right ≠ empty then
    Call TREE-WALK (tree.right)
endif
endmethod

```

Algorithm 7.4.4: Procedure *TREE-WALK* selects the n most likely trigram permutations that match the initial template.

Algorithm 7.4.4 concludes our discussion about the algorithms that concern the interpretation of trigram frequencies. In the following Section the Flisp functional interface of *trigram* is described.

7.4.5.2. Trigram interface description

Because no string-data type is implemented in Flisp, the arguments to *trigram* use integers that represent ASCII values of characters. A λ in a template is denoted with the 0 value.

Trigram takes two arguments. The first is the *template*, the second is n , the value that denotes the number of most likely trigrams that match the template which should be returned. If n is too large then all the permutations are returned. The template is a list that contains three integers, respectively a_0, a_1 and a_2 of the template t_w . In the following example, the seven most likely trigrams that match $th\lambda$ are returned. As can be seen the letter e is the most likely symbol to replace the λ .

```
->(trigram '(116 104 0) 7)
((101) (97) (32) (105) (111) (114) (109))
```

In the example the letters t and e have respectively ASCII values 116 and 104. The result shows that the letters $e, a, , i, o, r$ and m are the seven best hypotheses for the λ symbol in $th\lambda$.

7.4.6. Conclusions

Although, the implementation of the trigrams brought along some strange implementation aspects (the maximum INTEL 80286 segmentation length of 64 kByte, the squeezing of about 19,000 table values in about 19,000 kByte), we can say that the search optimisation was worth the trouble. In the future, further research is recommended in the dependence of the speed up on different texts and different trigram analysis. The tools which were resident on the UNIX operating system, helped us a lot and saved a lot of time, specially the "droff" command and the "-w" option were the reason why we did not had to write a parser to put the words in line. Trigram analysis seems to be a powerful, easy way to reduce search overhead.

7.5. THE STATISTICAL KNOWLEDGE

In this section we shall introduce some 'own made' properties for the output of the Hough transform which are fully, font- and size dependent. The reason for doing so is, that the output of the Hough transform for a certain character in a certain font, will be the same every time the image of the character is scanned. If the controller knows what the font and size of the printed characters are, it may use these properties for determining a hypothesis. We shall have a look at this possibility in section 6 of this Chapter.

The 8 'own made' properties for the output of the Hough transform, are

- Number of lines
- Number of horizontal lines
- Number of vertical lines
- Number of lines with a positive slope
- Number of lines with a negative slope
- The length of the smallest line(s)
- The length of the longest line(s)
- The average line length

Note that some properties are determined by others. Now suppose a font consists of 80 characters. If we determine these properties for each character in the font, it shows that almost every character has a different value for some property. Below, some results of these properties are shown.

char	#lines	#hor	#ver	#pos	#neg	aver	min	max
M	20	6	7	4	3	6	3	17
N	31	8	10	6	7	5	3	17
O	18	6	4	4	4	6	3	14
P	20	4	6	6	4	5	3	17
Q	20	6	4	4	6	6	3	16
R	27	5	9	7	6	5	3	17
S	28	8	8	4	8	4	3	8
T	8	2	2	2	2	9	3	17
U	13	4	5	2	2	7	3	15
V	19	4	8	3	4	5	3	14

Table 7.5.1: Some Characters and their 8 calculated properties for Helvetica Font.

In the following part we shall give the pseudocode for these 8 properties. Please note that they are implemented in FLisp. *LIST_LINES* is a list with lines as elements, the lines are lists too, with 5 elements: the x and y coordinate of the upperleft corner: (x,y), the x and y coordinate of the rightunder corner: (p,q), and the length of the line as fifth element: l. The *MINIMUM_LINE_LENGTH* is the minimum required length for the lines which are counted in the properties.

```

Procedure NUMBER_OF_LINES(LIST_LINES,MINIMUM_LENGTH)
Method
  If LIST_LINES not equal NULL Then
    If l > MINIMUM_LENGTH Then
      Let NUMBER_OF_LINES(CDR (LIST_LINES),MINIMUM_LENGTH)++
    Else
      Let NUMBER_OF_LINES(LIST_LINES,MINIMUM_LENGTH)
    endif
  endif
endmethod

```

Algorithm 7.5.1: Procedure *NUMBER_OF_LINES*(*LIST_LINES*,*MINIMUM_LENGTH*).

```

Procedure NUMBER_OF_HOR_LINES(LIST_LINES,MINIMUM_LENGTH)
Method
  If LIST_LINES not equal NULL Then
    If l > MINIMUM_LENGTH and CAR(LIST_LINES) = horizontal line Then
      Let NUMBER_OF_HOR_LINES(CDR (LIST_LINES),MINIMUM_LENGTH)++
    Else
      Let NUMBER_OF_HOR_LINES(LIST_LINES,MINIMUM_LENGTH)
    endif
  endif
endmethod

```

Algorithm 7.5.2: Procedure *NUMBER_OF_HOR_LINES*(*LIST_LINES*,*MINIMUM_LENGTH*).

Procedure *NUMBER_OF_VER_LINES*(*LIST_LINES*,*MINIMUM_LENGTH*)

Method

```

If LIST_LINES not equal NULL Then
  If l > MINIMUM_LENGTH and CAR(LIST_LINES) = vertical Then
    Let NUMBER_OF_VER_LINES(CDR (LIST_LINES),MINIMUM_LENGTH)++
  Else
    Let NUMBER_OF_VER_LINES(LIST_LINES,MINIMUM_LENGTH)
  endif
endif
endmethod

```

Algorithm 7.5.3: Procedure *NUMBER_OF_VER_LINES*(*LIST_LINES*,*MINIMUM_LENGTH*).

Procedure *NUMBER_OF_POS_LINES*(*LIST_LINES*,*MINIMUM_LENGTH*)

Method

```

If LIST_LINES not equal NULL Then
  If l > MINIMUM_LENGTH and CAR (LIST_LINES) = positive slope Then
    Let NUMBER_OF_POS_LINES(CDR (LIST_LINES),MINIMUM_LENGTH)++
  Else
    Let NUMBER_OF_POS_LINES(LIST_LINES,MINIMUM_LENGTH)
  endif
endif
endmethod

```

Algorithm 7.5.4: Procedure *NUMBER_OF_POS_LINES*(*LIST_LINES*,*MINIMUM_LENGTH*).

Procedure *NUMBER_OF_NEG_LINES*(*LIST_LINES*,*MINIMUM_LENGTH*)

Method

```

If LIST_LINES not equal NULL Then
  If l > MINIMUM_LENGTH and CAR (LIST_LINES) = negative slope Then
    Let NUMBER_OF_NEG_LINES(CDR (LIST_LINES),MINIMUM_LENGTH)++
  Else
    Let NUMBER_OF_NEG_LINES(LIST_LINES,MINIMUM_LENGTH)
  endif
endif
endmethod

```

Algorithm 7.5.5: Procedure *NUMBER_OF_NEG_LINES*(*LIST_LINES*,*MINIMUM_LENGTH*).

Procedure *MAXIMUM_LENGTH*(*LIST_LINES*)

Method

```

If CDR(LIST_LINES) equal NULL then
  Return length of CAR(LIST_LINES)
Else
  Let m = MAXIMUM_LENGTH(CDR(LIST_LINES))
  Let n = length of CAR(LIST_LINES)
  If n > m then
    Return n
  Else
    Return m
  endif
endif
endmethod

```

Algorithm 7.5.6: Procedure *MAXIMUM_LENGTH*(*LIST_LINES*).

Procedure *MINIMUM_LENGTH*(*LIST_LINES*)

Method

```

If CDR(LIST_LINES) equal NULL then
  Return length of CAR(LIST_LINES)
Else
  Let m = MINIMUM_LENGTH(CDR(LIST_LINES))
  Let n = length of CAR(LIST_LINES)
  If n < m then
    Return n
  Else
    Return m
  endif
endif
endmethod

```

Algorithm 7.5.7: Procedure *MINIMUM_LENGTH*(*LIST_LINES*).

Procedure *SUM_ALL_LENGTHS*(*LIST_LINES*, *MINIMUM_LENGTH*)

Method

```

If LIST_LINES not equal NULL Then
  If l > MINIMUM_LENGTH Then
    Let SUM_ALL_LENGTHS(LIST_LINES, MINIMUM_LENGTH) =
      SUM_ALL_LENGTHS(CDR(LIST_LINES), MINIMUM_LENGTH) +
      LENGTH OF CAR(LIST_LINES)
  Else
    Let SUM_ALL_LENGTHS(LIST_LINES, MINIMUM_LENGTH)
  endif
endif
endmethod

```

Algorithm 7.5.8: Procedure *SUM_ALL_LENGTHS*(*LIST_LINES*, *MINIMUM_LENGTH*).

Procedure *AVERAGE_LENGTH*(*LIST_LINES*,*MINIMUM_LENGTH*)

Method

Return *SUM_ALL_LENGTHS*(*LIST_LINES*,*MINIMUM_LENGTH*) divided by
NUMBER_LINES(*LIST_LINES*,*MINIMUM_LENGTH*)

endmethod

Algorithm 7.5.9: Procedure *AVERAGE_LENGTH*(*LIST_LINES*,*MINIMUM_LENGTH*).

In the following two graphs some insight in the distribution of the properties of the characters is given. Horizontal: some values for the properties *MINIMUM_LENGTH* (please read the minimum length of a line returned by the Hough transform) and vertical the records counted which correspond with the horizontal values (see figure 7.5.1). In figure 7.5.2 the same is done for the *AVERAGE_LENGTH*.

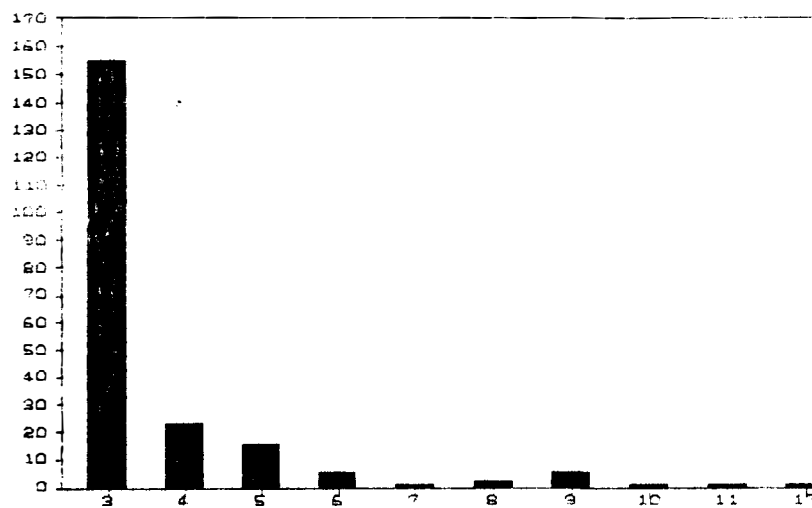


Figure 7.5.1: *MINIMUM_LENGTH* Summerised for all Characters

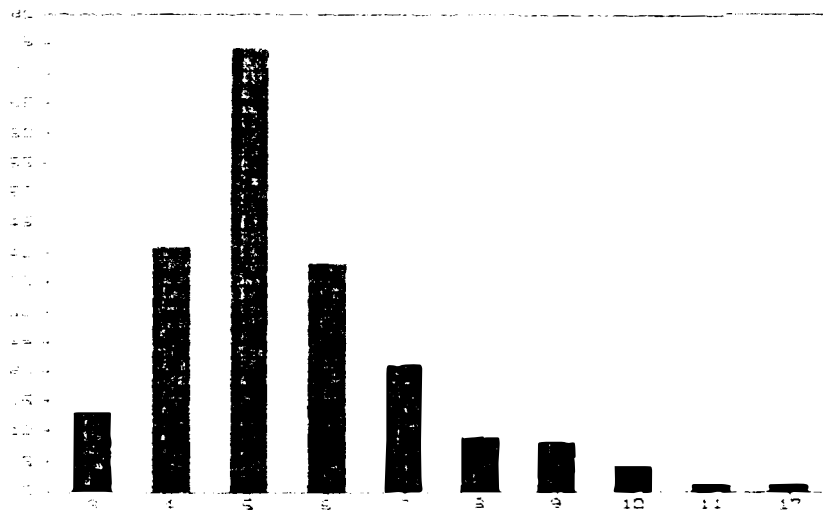


Figure 7.5.2: AVERAGE_LENGTH Summised for all Characters

It may be clear that high peaks in these graphs indicate properties with low intern variability. The more smooth the graph is, the better it can be used for the classifier. If we implement the values of these properties for each character in a table, we only need 720 Bytes (per character the ASCII value and the 8 property values). We can sort this table, using the values of the properties. These tables will be called *font tables* (see appendix A for more information on the specific values for different characters and different fonts).

7.6. THE CONTROLLER

7.6.1. Functional description

The basic function that has to be performed by the Controller, is to transform the preprocessed data into character ASCII-values.

The Hough transform can be used so that it only detects line segments with a minimum length, m . In this way it is possible to automatically discard small lines that probably are not essential for the pattern of a character. Since we do not know what the size of the object is, we are not able to state an arbitrarily value for this minimum line length. But we can say that there is some constant, c , which denotes the maximum number of characteristic lines for all characters.

Now, by applying the Hough transform repeatedly with increasing m until the number of line segments returned is less than c , we can obtain the characteristic lines, and perhaps some embellishments, for an object, independent of the size of the object.

The Controller has access to a knowledge base containing topological properties of characters, on which it can perform a **QUERY** or **TRACE**. Both these functions reply with certain properties; they advise the Controller to investigate these properties. The Controller will keep track of the properties of the object which are already known, or become known by investigating them.

We have seen that the use of a hypothesis, about the object to be recognised, will speed up the search process in the decision graph, if the hypothesis comes out to be true. So we want the hypothesis to be determined with the greatest certainty.

The use of 'trigram analysis' for obtaining a hypothesis is discussed in section 7.4. If we want to say something about the integrity of the hypothesis, obtained from the trigram analyses, we can use the font tables, discussed in the previous section. Suppose the controller has access to n font tables. We let the Controller, first of all, investigate the 8 properties for the input (which does not take much time). We can then obtain n hypotheses by looking in the n font tables, using the values found as an index, and take the associated character out of it.

This leaves the Controller with a list of hypotheses, sorted on their probabilities, obtained by 'trigram analysis', and with a list of hypotheses, obtained from a total different approach, n.l. the font tables. To obtain the most likely hypothesis, we take the intersection of the two lists, and pick, out of this intersection, the hypothesis which had the highest probability in the trigram list. If the intersection is $\emptyset$, we take the first hypothesis out of the trigram list, and we conclude that we are recognising a font of which we have no font table.

When using these font tables we are able to share the experience about the actual font with other Controllers in the parallel system. Consider the following two additional function specifications:

- Once a character is recognised, we shall take a look at the hypotheses obtained from the font tables. If none of the hypothesis was the actual character recognised, we get the strong feeling that the input is printed in a new font. If this happens several times we can start making a new font table. This is an easy job, since the controller already investigated the 8 properties for the object. We put these property values in a new font table and store the actual character recognised with it. Every time a new character is recognised this is done, and the table will grow until we have the values of all the 80 characters in a font.
Further more, in the case of a parallel system, the Controller will broadcast a message to all the other Controllers, saying that there is a new font to be considered and to start building up the new font table.
- Now suppose that the hypothesis of the font table T_{roman} turned out to be the actual character recognised. Then this strongly suggests that the input is in the roman font. If this happens several times we may assume that the font is roman. We will then be able to discard all the other font tables, and broadcast a message to the other controllers that the actual font is roman. Naturally, if one of the hypothesis of T_{roman} turns out to be false, we reject the assumption, we start looking at all the font tables again, and broadcast our rejection.

7.6.2. The Implementation

With the specifications above we come to the implementation of the controller. We will give the pseudo-code of the controller and the functions used.

Globally, the Controller works as follows:

```

(CONTROLER area)
object := (GET_HOUGH area c) {c being the maximum of lines}
known-prop :=  $\emptyset$  {known-prop is global and will be updated in CHECK_PROP}
tri-hypoth := (TRIGRAM history) {history being the last 2 characters recognised.}
check messages from other controllers
font-prop := (CALCULATE_8_PROP object)
font-hypoth := (LOOK_UP_FONT_TABLES font-prop)
hypothesis := first element of tri-hypoth that is  $\in$  font-hypoth
req-prop := (TRACE hypothesis)
result := (CHECK_PROP req-prop)
IF result = nil
THEN
    result := (GO_FORWARD)
    IF result = REJECT
    THEN
        RETURN 'unrecognised'
    ENDIF
ELSE
    result := hypothesis
ENDIF
history := history + result
compare result with font-hypoth and if needed broadcast
RETURN result
END

```

Algorithm 7.6.1: The CONTROLER.

The function GET_HOUGH will have to return the object to be recognised. As mentioned in 7.6.1 this will be done with an increasing minimum line length, until the object consists of no more than c lines. In pseudo-code:

```

(GET_HOUGH area c)

i := 1
WHILE number of elements of object > c DO
    object := (HOUGH i 1 area) {where 1 is the maximum slope of the lines}
    i := i + 1
ENDWHILE
RETURN object
END

```

Algorithm 7.6.2: The function GET_HOUGH.

The function **TRIGRAM** is discussed in section 7.4. The functional specifications of the functions **CALCULATE_8_PROP** and **LOOK_UP_FONT_TABLES** are discussed in 7.5.1. The function **TRACE** is discussed in 7.3.4.

The function **CHECK_PROP** is to investigate (or calculate) a list of properties. The properties investigated, become a member of the set *known-prop*. If one property does not hold, the function halts and returns *nil* else *true*.

```
(CHECK_PROP req-prop)

ok := true
WHILE (ok AND next-element exist) DO
    calculate property 'next-element req-prop' of object
    add 'next-element of req-prop' to known-prop
    IF property does not hold
    THEN
        ok := nil
    ENDIF
    take next element of req-prop
END_WHILE
RETURN ok
END
```

Algorithm 7.6.3: The function CHECK_PROP.

Finally we describe the function **GO_FORWARD** which uses the function **QUERY** (and the function **CHECK_PROP** for updating known-prop). This function will be activated if the hypothesis fails, and will continue investigating properties until the function **QUERY** returns with a terminal frame (the character), or with the frame *REJECT* (there is no frame which has the properties of the object). It works as follows:

```
(GO_FORWARD)

actual-frame := (QUERY REJECT known-prop)
WHILE (actual-frame is not REJECT OR a terminal) DO
    actual-frame := (QUERY actual-frame known-prop)
    (CHECK_PROP properties of the child of actual-frame)
END_WHILE
RETURN actual-frame
```

Algorithm 7.6.4: The function GO_FORWARD.

This concludes our Chapter on the design and implementation of the Classifier.

CHAPTER 8

POSTPROCESSING

A Description of the Implementation

ABSTRACT

In this Chapter we shall describe the implementation aspects of the spellings corrector which was originally planned to run on the host. But, which runs now on a ordinary mainframe because of implementation reasons. Hashing tables, permuting the wrong words, confusion groups and pre-and suffix stripping are among the subjects discussed.

8.1. BACKGROUND

Earlier research at AT&T Bell Laboratories has shown that a spelling correction system can increase the recognition rate up to 10%, with relatively less effort. On Bell, they had one advantage over us, they were provided with a good-working and broadly-tested spelling-control program: Johnson's SPELL, which has been upgrated and optimised by McIlroy. We on our NCube system, did not have a spelling checker at all. Therefore we were forced to write one ourself. This spelling checker was extended with an interface to the NCube parallel array and a permutation mechanism to generate various words within a character confusion group (e.g. i, l, 1, [,]). The program also builds the final file with recognised words. The spelling correction was implemented on the host because there was no more space left on the cube processors to squeeze another 60 kByte in the local memory. We do however take a risk in providing the system with a possible bottleneck between the host and the cube.

8.2. ADVANTAGES OF SPELLING CORRECTION

As told above, spelling correction is used to increase the performance of the system. Besides the fact that it is quite easy, whenever one has a spelling checking program, it can be said that spelling correction is in fact the most natural method to optimise the system. It uses only contextual information, and even if the system is not able to give a hint for a character not recognised exactly (or with a certain reliability) the spelling correction mechanism can provide us with a correct solution. So, by using spelling correction techniques, we can classify less exactly, because we always have a spelling correction mechanism in the final phase. The problem in tuning an optical character recognition system, is to find the perfect balance between the effort the classifier must make and the effort the spelling correction system must make, in order to get a good load-balancing.

8.3. THE METHOD USED FOR SPELLING CORRECTION

In this part we shall describe the methods used to build the spelling list, the channel walking mechanism, the generating of permutations if the word was not recognised correctly, and finally the spelling checking and the building of a ASCII-file.

8.3.1. Building the Hash-Table

In order to check a word on its presence fast, one needs to use direct adresssing. Therefore the calculated address value should be unique. On the other hand, the number of entries in the table should be restricted to a maximum, because of memory reasons. A method to do this is called Hashing [Nelson, 1985]. Each word gets a specific value, its address in a table, whenever an address is occupied, the program calculates a new

value. The maximum number of new calculations should also be restricted to a maximum of e.g. 9. Suppose we store all the 24,000 words in a table with 24,000 entries, then, using a binary search method, we have to check 15 entries. If we use hashing in a table of 50,000 entries and 9 recalculations, then we have a maximum search of 9. But whenever a table entry is found empty, one can say that the word does not exist. Because of the fact that the table was filled in alphabetical order, one can say that the word is not present whenever a word with a greater alphabetic is found. So we have an average search process of 4 table-checks. But how does one squeeze 24,000 words (consisting of lower-, uppercase and numbers) in a table of 50,000 entries? First, we calculated the ASCII-value of a word (i.e. the sum of the ASCII values of the characters in the word), and tried to store it in the hashtable. Unfortunately, the number of new calculations (because a lot of entries were already occupied) exceeded the maximum after 500 words. Then we tried to put information on the position of a character in a word, by multiplying each ASCII-value with a number, depending on the position of the character in the word. First, we tried simple numbers as 1, 2, 3, 4, ... It seemed however that this did not work quite properly. The table had an overflow after 13,000 words. The next step was to multiply the characters with prime numbers. We generated a list of 60 prime numbers: 1, 3, 5, 7, 11, 13, ... and multiplied the first character with the largest prime number of the list. The address was calculated with the following formula:

$$\text{Hash}_{\text{address}} = (\text{char}_1 \times \text{prime}_{60} + \text{char}_2 \times \text{prime}_{59} + \dots) \text{ modulo TABLESIZE}$$

$$\text{divider} = \frac{k}{\text{TABLESIZE}}$$

By calculating the initial value modulo the TABLESIZE (which is a prime number, so we exclude the influence of factors) we create a circular table. Whenever the table entry is occupied, we calculate a new one with the following formula:

$$\text{Hash}_{\text{address}} = (\text{last\_address} + \text{divider} \times \text{prime}_{\text{number of retries}}^2 + \text{prime}_{\text{number of retries}}) \text{ modulo TABLESIZE}$$

Where: $\text{prime}_{\text{number of retries}}$ is the prime number in the prime list with as index the number of retries.

We can express this algorithm in pseudocode:

Procedure FILL-HASH-TABLE**Method**

```

While not EOF
  Let Word = read_in(word)
  Let address = (char1 × prime60 + char2 × prime59 + ... ) modulo TABLESIZE
  If Hash_table(address) equal NULL then
    Let Hash_table(address) = Word
  Else
    Let FOUND = FALSE
    Let i = 0
    While i < NUMBER_OF_RETRIES and FOUND = FALSE do
      Let address = (last_address + divider × primenumber of retries2 + primenumber of retries) modulo TABLESIZE
      If Hash_table(address) = NULL then
        Let Hash_table(address) = Word
        Let FOUND = TRUE
      else
        Let i = i + 1
      endwhile
    endelse
  endwhile
endmethod

```

Algorithm 8.3.1: Procedure FILL_HASH_TABLE.

Using this hash-method, we were able to squeeze 24,000 words in a table with 48,791 entries, and maximal 9 retries. The checking of a word takes no more than 9 table comparisons, the average seemed to be about 4, using the property that the table was filled alphabetically.

8.3.2. Controlling the NCube Channels

After the Hash-table was filled, the NCube channels are checked one by one on the presence of a word, recognised by the system. The word is then loaded on the host, and checked on its correct recognition. Each word consists of three parts:

- Global Information, on the number of the word, or in other words, the place of the word in the total text.
- Characters, the recognised ASCII-values.
- A bitmap of the word, with a zero for a properly recognised character, and a one for a not properly recognised character. The correct words are stored in a file, the incorrect words are corrected. The pseudocode for the algorithm is:

```

Procedure CONTROL_WORD(Word)
Method
  For  $i = 0$  to END_OF_WORD do
    If bitmap( $i$ ) = 1 then
      Word = incorrect
      Return(Word is incorrect)
    endif
  endfor
  Word = correct
  Return(Word is correct)
endmethod

```

Algorithm 8.3.2: Procedure CONTROL_WORD.

8.3.3. The Confusion Groups

The characters can be divided into a number of confusion groups. These groups are sets of characters which look alike in many ways, for example:

- l, 1, i, (,), [,], consisting all of one vertical line.
- p, b, 6, all having a vertical line and a circle.
- Q, O, 0, characters with one circle.
- B, 8, S, two holes in big characters.
- e, s, g, a, two holes in small characters.
- d, q, consisting of a circle and a vertical line.

Because the system gives a hint for the character to be recognised, in the character list, we can permute this character within a confusion group. This is done for all the characters in a word, which were not recognised properly. Formally, we can describe the problem as follows:

We have a word W consisting of the following characters: $\{a_1, a_2, \dots, a_n\}$, in the word W a number of wrong characters exists, namely: $\{w_1, w_2, \dots, w_{\text{max\_number\_wrong}}\}$. Each wrong character w_i belongs to a certain confusion group C_i . We can describe a word as a number of correct characters and a number of wrong characters, where the wrong characters must be permuted within a confusion group c_i , e.g. $a_0a_2w_1a_4w_2a_5a_6w_3$. By replacing the w_i symbols in word W we can find all the possible words. The permuting algorithm is described in the chapter "Classifier" in the part of the trigram analysis.

8.3.3.1. Pre- and Suffix Stripping

The dictionary we have, consists of 24,000 words, one can understand that not all English words are included. For instance, the word "work" is in the dictionary, "works" and "worked" are not included. In the English language, it is however possible to strip a word off its pre- and suffixes, and then checking the remained word in the dictionary. If this word exists, then the word with pre- and suffixes exists to. The list of pre- and suffixes can be found in Appendix C. In this part we will give a little example. Suppose we have the word "preprocessing", then the prefix "pre" can be stripped according to the list of prefixes. The suffix is a little more difficult, it is clear that the suffix is "ing", these suffixes can result in a two types of words, the one without "ing" and the one with an "e" instead of "ing" (e.g. "handle" and "handling"). So we have two words left to check: "process" and "processe". Obviously, the first one exists and is found resident in the word list. So we have the following two procedures:

Procedure STRIP_PREFIX(Word)**Method**

```

Let address = ASCII-VALUE of PREFIX
If Prefix(address) equal First-letters in word then
  Return(Word without prefix)
else
  while RETRIES < MAX and NOT FOUND
    Let address = address + 1
    If Prefix(address) equal First-letters in word then
      Return(Word without prefix)
    else
      Let RETRIES = RETRIES + 1
      Let address = address + 1
    endif
    Return(Original Word)
  endwhile
endif
endmethod

```

Algorithm 8.3.3: Procedure STRIP_PREFIX**Procedure STRIP_SUFFIX(Word)****method**

```

Let address = ASCII-VALUE of SUFFIX
If Suffix(address) equal First-letters in word then
  Return(Words with replaced suffixes)
else
  while RETRIES < MAX and NOT FOUND
    Let address = address + 1
    If Suffix(address) equal First-letters in word then
      Return(Words withreplaced suffixes)
    else
      Let RETRIES = RETRIES + 1
      Let address = address + 1
    endif
    Return(Original Word)
  endwhile
endif
endmethod

```

Algorithm 8.3.4: Procedure STRIP_SUFFIX**8.3.3.2. The Final Checking**

Now, the permuted words, without pre- and suffixes, upper case letters, etc, are checked in the hash table. If one word is found, it is, after the higher case characters are restored, put in the ASCII-file, if no word is found, the word provided by the recognition system is put in the ASCII-file with a mark that it is probably wrong. We can describe the main procedure in terms of the above described procedures:

Procedure CORRECT_WORDS**Method**

```

Let Prime() = list of prime numbers
Let Prefix() = table of prefixes
Let Suffix() = table of suffixes
FILL_HASH_TABLE
While NOT EOF do
  Let Word = read_in(word)
  If CORRECT(Word) then
    Write Correct Word to file
  else
    While not all permutations done and not found do
      PERMUTE(Word)
      STRIP_PREFIX(Word)
      STRIP_SUFFIX(Word)
      For all words returned from suffix do
        CHECK Word in Hash_table
        If Word in hash_table then
          Write Corrected word in file
          Continue with new word
        endif
      endfor
    endwhile
  endif
endwhile
endmethod

```

Algorithm 8.3.5: Procedure CORRECT_WORDS.**8.3.4. Building the ASCII-File**

Finally, the words are recognised and corrected if needed, the words are standing in a ASCII-file on the Host's Hard Disk. The words are sorted so the text is back in the original form, and it can be used for all kind of purposes.

8.4. IMPLEMENTATION OF SPELLING CORRECTION ON THE HOST

During the implementation of the spelling correction mechanism on the host, we were puzzled with the declaration of large datastructures on a INTEL 80286 processor. Apparently, it is impossible to work with larger segments than 64 kB. The only way to use more than 64 kB of memory is to work with segment overlay's. Normally this is provided by the compiler one is working with. However, the C compiler, build by NCube company knows only two memory models:

- The large memory model, using two byte pointers for code-addresses and two byte pointers for data-addresses. Therefore, it can only work with a 64 kByte code segment, and a 64 kByte data segment.
- The medium memory model, using four byte pointers for the code-addresses, and two byte pointers for the data-addresses. Therefore it is possible to have more than 64 kByte program code, but it is not possible to have more than 64 kByte data.

Because of the fact, that we need already about 200 kByte for our word list, it was rather impossible to implement the program on the host. We thought about implementing it on a node, but this provided us with all kind of communication problems, another possibility was to write the program in Fortran (which knows overlays for the data addresses), time, however, was valuable at the end of the project. Therefore we

collected the data on the hard disk of the host, transferred it by modem to a mainframe, the data was then corrected (if possible) and the results were send back by modem to the host. The benchmarks, do not include the time needed for transformation of the data. Only the collection on the host, the time needed to correct the data, and the sorting into a proper text file were counted.

More details on the implementation can be expected very soon.

8.5. CONCLUSIONS

Spelling correction has proved to be a powerfull tool to include semantic information on the context of the problem. The fact that we had to implement the spelling correction program on the host, could provide us with a bottleneck. We will continue on this subject in part three: Results. In the futur it will be very interesting to investigate the perfect balance between the effort done by the classifcator to recognise a character, and the number of words that should be corrected on the host.

PART 3

RESULTS, CONCLUSIONS AND RECOMMENDATIONS

- **RESULTS**
- **CONCLUSIONS AND RECOMMENDATIONS**

CHAPTER 9

RESULTS

The Performance of the Optical Character-Recognition System

ABSTRACT

In Part Two the *designs* of the Preprocessor, Classifier and Postprocessor are described. In this Chapter the results of the *implemented* systems are presented, examples are given and preliminaries for (tentative) conclusions are made. The final conclusions presented in the next Chapter are drawn from these results.

9.1. BACKGROUND

The results of our Optical Character-Recognition System will be presented and discussed, in this Chapter. First, we shall focus on the preprocessor results concerning the distribution, splitting of the image as well as its vectorizing. Second, the results of the classifier are investigated both qualitatively and quantitatively. The trigram analysis and the recognition rate will be examined in greater detail than, e.g., the skeleton representation. Finally, we submit the output of the classifier to the postprocessor, and outline the results obtained in the last phase. In the preprocessing as well as in the classification stage, the speed-depending tests, have been performed for hypercubes of dimension 0, 1, 2, 3 and 4. The input data is presented in Section 9.2 in Figure 9.2.1.

9.2. THE PREPROCESSOR RESULTS

In this Section, we shall investigate the performance of the three algorithms: Distribute, Split and Hough Transform. The input image is *split* into areas by Split after which the areas are subsequently *distributed* over the nodes in the NCube with the Distribute algorithm. Thereafter, the distributed parts are preprocessed by further partitioning with the Split Algorithm; *line segments* are *detected* with the Hough Transform. Investigating these three algorithms for a given input image, we

1. determine the communication time needed to distribute the input image over the nodes in the allocated hypercube;
2. determine the computation time needed to split the image;
3. show the result of the split operation on the given input image;
4. relate the time needed by the Hough Transform for detecting an arbitrary area in the image containing n line segments;
5. measure the increase in performance of the Hough Transform that is obtained by first splitting the image and subsequently applying the Hough Transform to these areas separately;
6. show the line segments that are detected by the Hough Transform;

In the next three Sections we present test results that were obtained from an image containing 98 words, in total 486 characters. In Figure 9.2.1 below, the text of the image is given. The text stems from the ICCA Journal Vol. 10, No. 2.

BY CRAWLS AND CREEPS
ICCA Journal June 1987

The layman is apt to think that science progresses by leaps and bounds, where each leap and every bound is a straight successor to whatever has gone before. To the uninformed, there is a simple unbroken succession from Newton to Lorentz to Einstein. Similarly, even to the educated, Dalton his atomic hypothesis leads naturally to Rutherford his first atomic transmutation and thence to Hiroshima and beyond.

To those active in the field, nothing could be further from the truth : they are aware that science proceeds by crawls and creeps.

Figure 9.2.1: The Text Used for Obtaining Test Results.

This text, slightly amended by us, is not an exact replicate of the bitmap that was split, but it shows the word and letter sequences. The dimensions of the image are 568×341 pixels.

9.2.1. Results of the Distribute and the Split Algorithm

The measured *split* time is the time required by a node to partition the image it has left over after *distribute* has finished. The higher the dimension of the allocated hypercube, the longer it will take to distribute the image over all. At the same time, however, the parts that are sent to each node will be smaller, implying that the time to *split* the image resident on one node will be reduced. This phenomenon can directly be observed from the measurements in Table 9.2.1. In this table the *distribution* and *split* times on the nodes for respectively a 0, 1, 2, 3 and 4-dimensional hypercube configuration are presented.

Distribute and Split Times						
Dimension	Node	Image Size		Distribute <i>msec</i>	Split <i>msec</i>	Total <i>sec</i>
		<i>x</i>	<i>y</i>			
0	0	568	341	976	5241	6.2
1	0	559	166	3249	2370	5.6
	1	559	166	3161	2736	5.8
2	0	559	102	4800	1226	4.9
	1	559	78	4957	1501	6.4
	2	559	60	4673	1166	5.8
	3	559	82	4861	1209	6.1
3	0	559	81	5466	840	6.3
	1	559	39	4021	776	4.7
	2	559	39	5300	767	6.0
	3	559	39	5528	501	6.0
	4	559	18	5224	411	5.6
	5	559	36	5517	746	6.2
	6	559	18	5154	404	5.5
	7	559	40	5391	718	6.1
4	0	522	37	5132	443	5.6
	1	559	18	5361	379	5.7
	2	559	18	5043	389	5.4
	3	559	18	5152	96	5.2
	4	256	18	4872	182	5.0
	5	559	18	5346	371	5.7
	6	285	18	4864	193	5.1
	7	559	18	5269	686	5.7
	8	522	18	5058	368	5.4
	9	559	18	5286	415	5.7
	10	559	18	5002	404	5.4
	11	559	18	5110	400	5.5
	12	299	18	4837	217	5.1
	13	559	15	5283	386	5.7
	14	272	18	4815	209	5.0
	15	559	18	5208	336	5.5

Table 9.2.1: Measured Times of the Distribute and Split Algorithm

From these results the following observations have come to our attention:

- The workload is about the same for each node, hence we may argue that the Split algorithm splits the image correctly as described in Section 6.1, and
- The time lost by distributing the image is entirely recovered by the time reduction that has resulted from splitting only a part of the image on a single node, i.e., the sum of the distribution time and the split time remains constant regardless the dimension of the hypercube.

Concerning the distribution of the image we add that the bitmap consists of 24 Kbytes and for bitmaps that contain an entire page, this may be even more. Since the I/O system of the NCube does not allow any

message to exceed 28 Kbytes, the bitmap is sent in several packages. The package size used in the tests is 20,000 bytes.

In figure 9.2.2 the output of the split algorithm is presented. Each rectangle describes an area that cannot be split any further and thus holds one character.

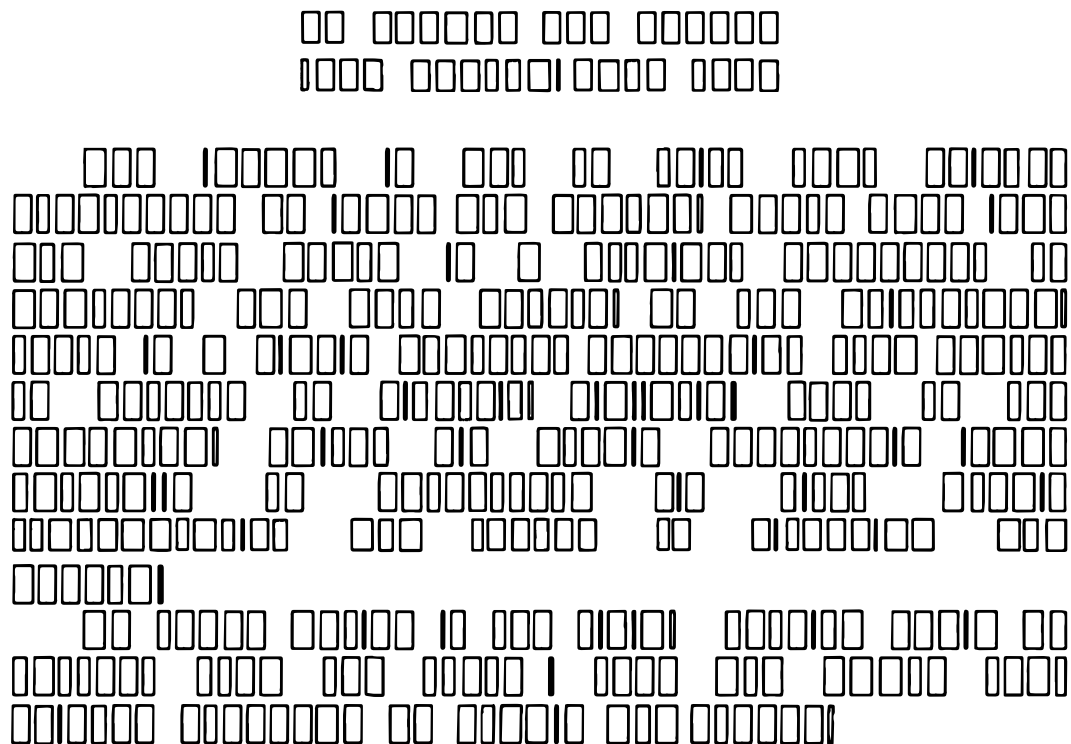


Figure 9.2.2: Output of Split for the Text Presented in Figure 9.2.1

Figure 9.2.2 gives a good impression of the split algorithm. It clearly illustrates that it is of no use to split the image by drawing vertical lines while the image contains more than one line. It also shows that areas containing a single character are not split any more by drawing horizontal lines through them, although this could be done since not all letters in a line have the same height.

9.2.2. Results of the Hough Transform

The rectangles depicted in Figure 9.2.2 in the previous Section are areas to which the Hough Transform can be applied. The line segments in the areas are then detected and returned by the Hough Transform in a vector representation. Table 9.2.2 presents the average execution times of the Hough Transform when applied to areas that contain n lines with a slope a , $|a| \leq 1$ and minimum length 2.

Hough Transform	
# Lines	Time <i>msec</i>
2	71
5	162
6	153
8	115
10	128
11	128
12	160
13	81
14	163
15	174
16	228
20	234
21	241
22	225
23	195
24	232
25	248
26	235
27	492
29	237
31	277
33	208

Table 9.2.2: Measured Times of the Hough Transform

The average times presented in Table 9.2.2 were obtained by applying the Hough Transform to all areas in the text depicted in Figure 9.2.1. In general, we have observed that the more lines exist in an area the longer it takes to detect all lines.

In Chapter 6, the space complexity of the Hough Transform has thoroughly been investigated and we claimed that the partitioning of the image would result in a reduction of the accumulator plane. Moreover, we have shown that the time complexity of the Hough Transform is at least of the same order as its space complexity. The right part of Table 9.2.3 shows the computation time for the Hough Transform when applied to a large area. The left part describes the computation times for the Hough Transform when applied to the separated areas that are obtained by splitting the area. At the bottom in the second column, the measured computation times and the *split* time are summed.

Time Complexity Hough Transform		
Split first		No Split
Sub Area	Time (msec)	
1	263	
2	237	
3	197	
4	200	
5	163	
6	264	
7	203	
8	81	
9	234	
10	196	
11	228	
12	116	
13	228	
14	228	
15	238	
16	177	
17	196	
18	77	
19	229	
20	237	
21	210	
<i>Split time</i>	225	0
<i>Total</i>	4367	21391

Table 9.2.3: Performance Increase by Splitting the Image

From the *Total* row it follows that, by first splitting the image, the lines in the large area are detected almost 5 times faster. The selected area to obtain the results shown in Table 9.2.3 and the areas that have resulted from the split operation are depicted in Figure 9.2.3. The main area is the last part of line four from the input text.

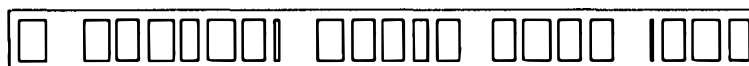


Figure 9.2.3: The split area used for Table 9.2.3

The dimensions of the split area are 298×19. The accumulators needed by the Hough Transform are roughly of the following size :

3 by 2×(298-1)×(19-1) for both the XY and YX accumulator

In total, this means that about 18 kByte is needed for the representation of the accumulator planes. For a single character this is far less, and becomes to 3×19×11 which is just a little under 1 kByte.

In Chapter 6, we showed that the time complexity of a $n \times n$ image was at least equal to its space complexity, $O(n^3)$. Subsequently, we showed that if the image was split in K^2 parts then time complexity for transforming a part would be reduced with a factor K^3 . In our case, the image in Figure 9.2.3 is split into 21 parts, thus $K = 4.9$. Hence, the Transform is speeded up by at least a factor $4.9^3 = 96$. Since there are 21 such parts the total speed up is $96/21 = 5$. This is the same factor as can be calculated from the total times

presented in Table 9.2.3.

Next, we shall consider the output of the Hough Transform. Figure 9.2.4 depicts the line segments detected in the areas from the first two lines in our input text.

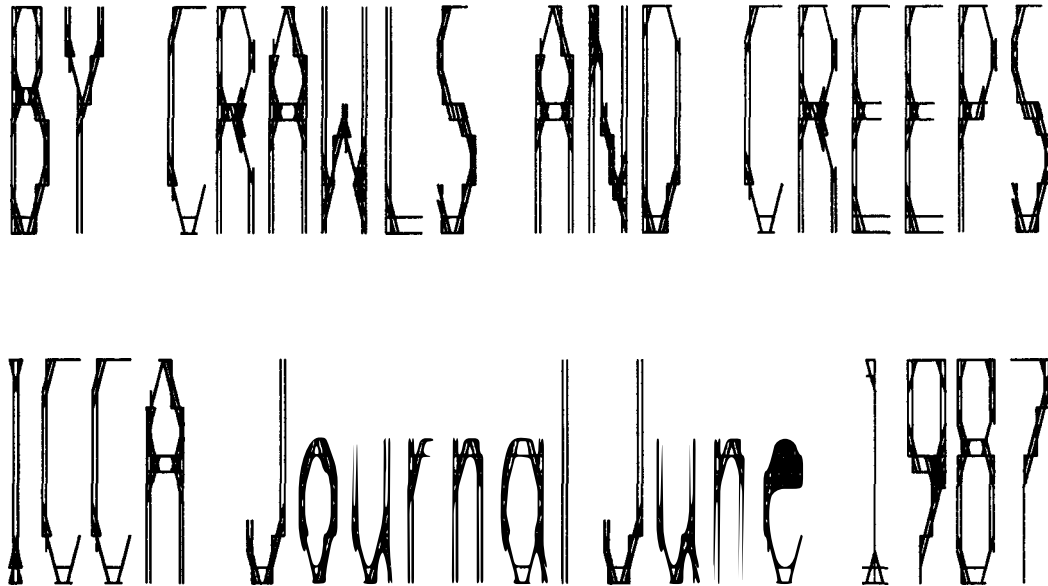


Figure 9.2.4: Output of the Hough Transform for the first Two Lines

The output is returned as a list containing vector descriptions, thereafter it will be used by the Classifier which has to determine the basic structure of the pattern contained. The results of the Classifier will be presented in the next Section.

This ends the Section about the Results of the Preprocessor.

9.3. RESULTS ON CLASSIFICATION

9.3.1. Results on Trigram Analysis

We expected the trigram analysis to decrease the search time with 60 *percent*. In this section, we shall investigate the exact influences of the trigram analysis. For a number of characters we have printed the first 10 hypotheses, provided by the trigram tables, together with the recognised character. In table 9.3.1, these results can be found, the first column holds the character recognised, the second the list with the trigrams obtained from the classifier as long as needed to find the right hypothesis, and the third column provides the reader with the order of the recognised character in the trigram list.

Recognised Character per Word	Trigram List	Order of Recognised Character in Trigram List
j	<i>taiospcmwrknhlvuqj</i>	19
u	<i>ou</i>	2
n	<i>sln</i>	3
e	<i>ditcelfskg</i>	5
t	<i>t</i>	1
h	<i>h</i>	1
e	<i>e</i>	1
l	<i>taiospcmwrknhl</i>	14
a	<i>ioea</i>	4
y	<i>rcstbny</i>	7
m	<i>-esiam</i>	6
a	<i>a</i>	1
n	<i>tn</i>	2
h	<i>taiospcmwrknh</i>	13
y	<i>aeoiy</i>	5
p	<i>dp</i>	2
o	<i>o</i>	1
t	<i>srint</i>	5
h	<i>-h</i>	2
e	<i>e</i>	1
s	<i>-rs</i>	3
i	<i>-eti</i>	4
s	<i>tnos</i>	4
l	<i>taiospcmwrknhl</i>	14
o	<i>io</i>	2
r	<i>wsgncpkatur</i>	11
e	<i>-e</i>	2
n	<i>-sn</i>	3
t	<i>t</i>	1
z	<i>all 26 lower-case characters</i>	26

Table 9.3.1: Trigram Search Optimisation

The results concerning the average number of hypotheses needed for the test text are printed against the number of a character in a word can be found in Table 9.3.2. The results are based upon the entire test text, although only 5 words are showed in Table 9.3.1.

Hypothesis	
n	Hypothesis
1	15
2	3
3	3
4	3
5	3
6	3
7	3
8	4
9	3
10	3

Table 9.3.2: Average Number of Hypothesis Needed
for n^{th} Character in a Word.

Assume that, we normally have to check $\frac{1}{2} \times 26 = 13$ characters to find the character meant. Now, with a trigram hypothesis it suffices to check only 3 characters after the first one is recognised. This implies that search optimisation of 88 percent is obtained.

9.3.2. Recognition Performance

In this Section, the recognition quality and speed are investigated. Both benchmarks are done with the text of Figure 9.3.1. The following results are based upon a classifier which uses the font table and trigram analysis only. This has been done because of the relative slowness of the decision graph. When analysing these results, we shall see that even without the decision graph (used to verify the hypothesis and which should result in a 100 percent recognition rate), the results obtained by the classifier are acceptable enough to pass the error-solving phase to the postprocessor.

The text of Figure 9.3.1 is put through the preprocessor and the classifier. The classifier came up with the following results:

BY CRAWLS AND CREEPS
ICCA Journal June 1987

The layman is apt to think that scienc{g,c,e,N,S}
progresses {d,b}y leaps an{d, b} {d, b}oun{d,b}s, where each
leap an{d,b} every {d,b}oun{d,b} is a straight successor to
whatever has gone {d,b}efore. To the uninforme{d,b},
there is a simple un{d,b}roken succession from Newton
to Lorentz to Einstein. Similarly, even to th{g,c,e,N,S}
e{d,b}ucate{d,b}, Dalton his atomic hypothesis lea{d,b}{c,s}
naturally to Rutherfor{d,b} his first atomic
transmutation an{d,b} thence to Hiroshima an{d,b}
beyon{d,b}.

To those active in the fiel{d,b}, nothing coul{d,b} {d,b}e
further from the truth : they are aware that
science procee{d,b}s by crawls and creeps.

Figure 9.3.1: Output from the Classifier

The classifier returns a number of characters, if it cannot classify the character uniquely. The possibilities

provided by the classifier can be found between the curly brackets. Totally, 27 of the 486 characters are not classified correctly. This implies that 95% of the total text is directly classified correctly. Assuming a text of 3000 characters (a normal well-filled page), 150 characters should not be classified correctly, an unacceptable high rate. Postprocessing seems to be *necessary* to obtain an economically acceptable rate.

The recognition speed that was achieved by our Optical Character-Recognition System varies from 2 characters per second on a zero-dimensional hypercube up to 17.4 characters per second for a four-dimensional hypercube. Our Optical Character-Recognition System outperforms commercial available Optical Character-Recognition Systems that achieve around 5 characters a second. Especially, considering the fact, that it is possible to run our system on a ten-dimensional hypercube, consisting of 1024 processors, without any performance loss due to communication overhead this results may be regarded hopefully. In the following table, the results will be described in more detail. The first column shows the dimension of the hypercube. The second the number of nodes, and the third the time needed to recognise the characters provided by the split and distribute algorithms. The times are given per active node.

Recognition times		
Dimension	Node	Time sec
0	0	236.6
1	0	109.2
	1	174.9
2	0	50.3
	1	67.1
	2	51.7
	3	51.7
3	0	23.4
	1	29.7
	2	40.8
	3	39.2
	4	42.1
	5	11.4
	6	23.4
	7	11.3
4	0	17.9
	1	27.9
	2	14.9
	3	3.2
	4	8.3
	5	14.1
	6	8.7
	7	14.9
	8	13.5
	9	16.5
	10	15.8
	11	15.7
	12	9.2
	13	14.4
	14	8.4
	15	13.9

Table 9.3.3: Recognition Times of Text Presented in Figure 9.1.1

The maximum time per dimension is significant for the performance of the system. In the table, different times can be found per dimension. This is due to, what we call, the load balancing of the text over the nodes. The use of trigram analysis leaves us with a choice between two distribute algorithms, because we have to keep track of the already recognised characters. In the first option, the distribute algorithm, which provides each node with the same number of characters, needs to communicate with the other classifiers which characters are recognised. In turn this would lead to an unacceptable high communication. In the second option, the distribute algorithm provides each node with a number of words, eliminating the need for communication for the trigram analysis. The partitioning of sequences of words over the nodes implies that not every node receives the same same amount of work (please read: that there is not a good work load balance). In figure 9.3.2 a graph is drawn for the number of nodes against the minimum and maximum time needed to recognise all the characters.

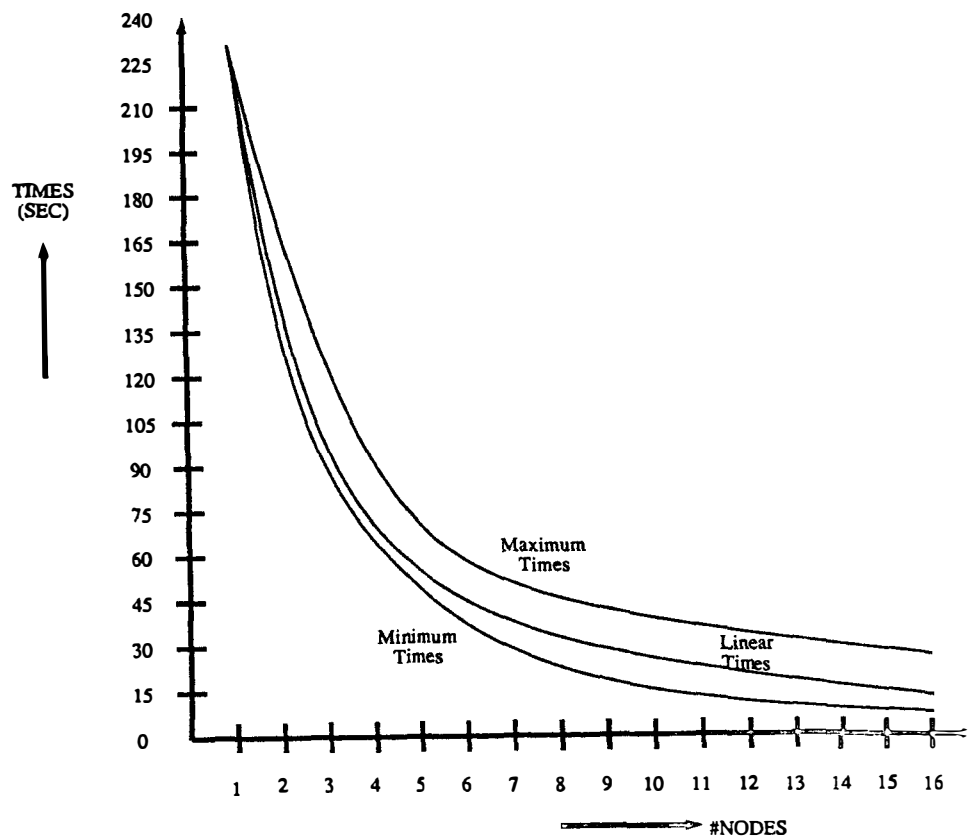


Figure 9.3.2: Maximum and Minimum times for the nodes

In the picture above, the spline "linear times" indicates the recognition time assuming that the time needed for recognition decreases linearly with the number of nodes used. However, more interesting is the number of characters recognised per second. The results can be found in table 9.3.4.

Number of characters recognised per second					
Legend	Dimension				
	0	1	2	3	4
Average	2.1	3.4	8.8	17.5	35.8
Maximum	2.1	4.4	9.7	43	153.3
Minimum	2.1	2.8	7.2	12.4	17.4

Table 9.3.4: Number of Characters Recognised per Second

The calculations are done by dividing the number of characters (i.e. 486) with the help of the maximum and minimum times of Table 9.3.3. The average number of characters per second is calculated by dividing the number of characters recognised through the average time per dimension for recognising the characters. In Figure 9.3.3, these values are presented graphically.

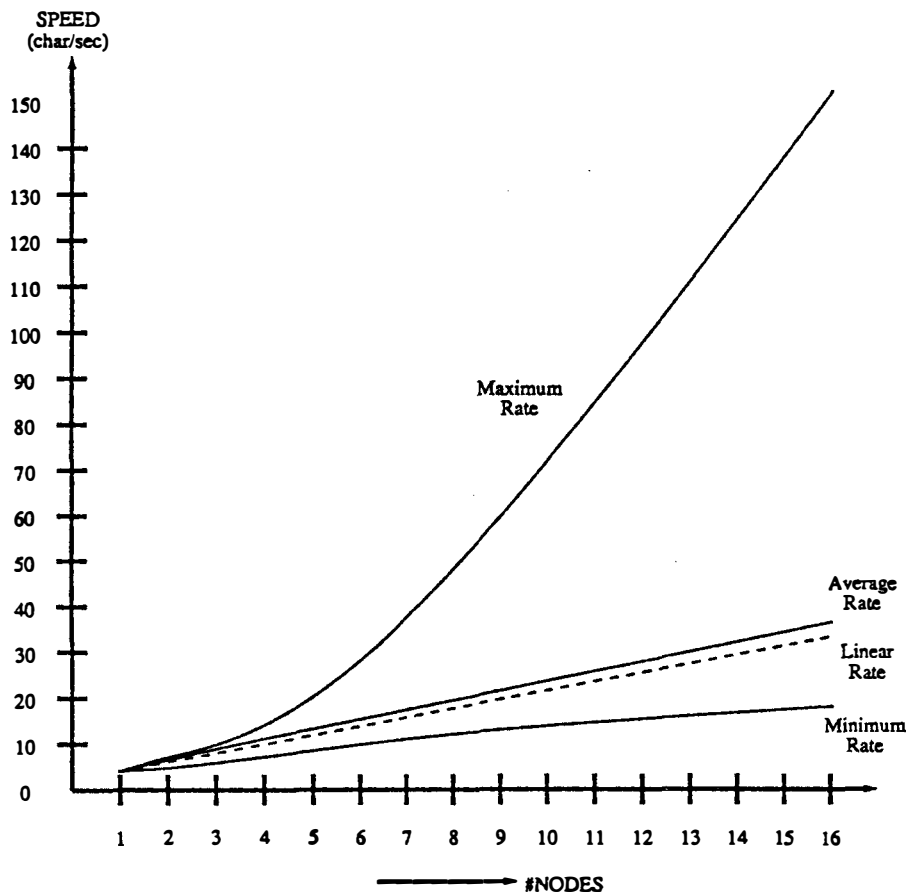


Figure 9.3.3: Maximum, Minimum and Average Characters per Second Recognised.

The Linear Rate is based upon the recognition of characters using a perfect work-load balance, (e.g., 486 characters with dimension 0; 243 characters with dimension 1, etc.) and the time achieved for the recognition on one node. The average, maximum and minimum rate are based upon the values of Table 9.3.4. We would like to call the area between the maximum and the minimum number of characters, the *load balancing dependent area*. Note that the average rate neatly follows the linear rate; moreover, it is better than the

linear rate. We might state, that the parallelisation has a more than linear speed-up. This is caused by the fact, that Lisp needs to clean its memory whenever it is full: called garbage collections. By using more nodes, more Lisp work space is available, hence, less garbage collections are needed. We can, therefore, expect that our system will outperform all existing Optical Character Recognition Systems, running on a ten-dimensional hypercube. This concludes the results on the classifier.

9.4. RESULTS ON POSTPROCESSING

In this Section, the results obtained in the postprocessing phase are presented. Unfortunately, we did not succeed in making an automatic link with the classifier. The main reason (the absence of a large memory model in the NCube-C compiler) has already been discussed in Part 3 (Chapter 8: "Postprocessing"). In the presentation of the results we have dealt with the results obtained by the classifier converted to a form the postprocessor understands. Now, we consider the postprocessor's correcting the wrong classified characters, in the ICCA-Journal text. Thereafter, the corrected words written to an output file are presented. The reader will find four tables: Table 9.4.1 to 9.4.3, giving some test input and output of the classifier, Figure 9.4.1 and 9.4.2 discuss the output of the classifier in modified form and the corrected words after the postprocessing phase. Each table or figure will be treated extensively.

9.4.1. Testing the Postprocessor

After some routine tests, concerning the working of the word list, the hashing formula and the detection of words in the word list, we seriously started testing the postprocessor's capacities on the generation of various permutations within a confusion group. Assume we have the word `being` with the first and the last character not recognised properly, hence we have: `?ein?`. The classifier provides us with some hints for these characters, e.g., the first character might be a `p`, and the last might be an `e`. It is now up to the postprocessor to generate a number of permutations within the confusion groups of the characters suggested by the classifier. In our example these are: `{p, b, 6}` and `{g, a, e, s}`. The permutations generated by the postprocessor are:

peing	peina	peine	peins
being	beina	beine	beins
6eing	6eina	6eine	6eins

After each permutation the postprocessor checks the generated word on its presence in the word list. Whenever a word is found to be correct, generating new words is interrupted. The word found is assumed to be the intended one. We are aware of the fact, that this might go wrong. However, the low probability for wrong or worst cases made us decide not checking all permutations and only returning a list with some possibilities.

In the next step in the testing process, the postprocessor's ability to strip pre- and suffixes was investigated. It looks that all the prefixes will be stripped, whenever it is possible. Moreover, all the stripped words seem to exist. In the following table some self-documenting examples are shown.

initial word	prefix stripped	resulting word
interaction	inter	action
counteraction	counter	action
preprocess	pre	process
stereoradio	stereo	radio
biolive	bio	live
unproved	un	proved
deflatable	de	flatable
electrooptical	electro	optical

Table 9.4.1: Some Stripped Prefixes

In the following phase, we had to examine the suffix stripping. Hence, an extra problem with the suffixes had to be faced, namely: the possibility of more than one suffix replacement. In Table 9.4.2 some results of the suffix-stripping tests are presented.

initial word	suffix found	resulting word	resulting word
works	s	work	
worked	ed	worke	work
processes	ses	processe	process
likely	ly	like	
potency	ncy	potent	
battlement	ment	battle	
horribly	bly	horrible	
flies	ies	fly	
identifiable	able	identify	
enumerable	able	enumere	enumer

Table 9.4.2: Some Tests on Suffix Stripping

The postprocessor stripped all the possible suffixes according to the rules. There were no suffixes found, which were not stripped. Also, no non-existing suffixes were stripped.

In the following table, the above independently tested abilities of the postprocessor can be found in a combined way. The fat printed characters in the first column are the characters which should be corrected. The second column shows the corrected words. Whenever a corrected word is not amended properly the intended word is shown in the third column. The power of pre- and suffix stripping is underlined by the fourth column, were the word found in the word list is given. Please note, that this word differs quite often from the word to be corrected. The table is split in two parts for layout reasons only.

no.	the word	the corrected word	was meant to be	the word found in table
1	interaction	interaction	copier	action
2	counteraction	counteraction		action
3	cobia	copies		copy
4	curried	curried		curry
5	flies	flies		fly
6	palatsa	palates		palate
7	woog	woos		woo
8	live[iest	liveliest		live
9	gtrongest	strongest		strong
10	axes	axes		axe
11	fixes	fixes		fixe
12	axertion	exertion		exert
13	bartly	partly		part
14	bproperly	properly		proper
15	stereoradio	stereoradio		radio
11	ladylike	ladylike		lady
11	horripoly	horribly		horrible
11	crsater	creator		create
16	works	works		work
17	worked	worked		work
18	brebrocess	preprocess		process
19	cheae	chess		chess
20	brocessing	processing		process
21	processing	processing		process
22	descripe	describe		describe
23	describe	describe		describe
24	hallo	hallo		hallo
25	more	more		more
26	NCupe	NCube		NCube
27	spell[	spell		spell
28	spell]	spell		spell
29	spell	spell		spell
30	com]ng	coming		come
31	findng	finding		find
32	walkine	walking		walk
33	beine	being	grep	be
34	gre6	gre6		-
35	intgrmgdiatg	intermediate		mediate
36	eduator	equator		equator
37	no]lity	nobility		nobel
38	ident1fiaple	identifiable		identify
39	potency	potency		potent
40	qgtection	detection		detect
41	nationgl	national		nation
42	mysticgl	mystical		mystic
43	etylistic	stylistic		stylist
44	pattlement	battlement		battle
45	su66lemental	supplemental		supplement

Table 9.4.3: Results of the Spelling Correction
fat: Characters to be Corrected
normal: Characters Recognised Correctly

We may conclude that the postprocessor corrects the words properly within a confusion group. Interesting is the third entry in the table. We intended the word *copier*, but the postprocessor found *copies* earlier as a possible correct word. The reason why the word *grep* was not found, is because it is not present in the word list. The fact that a word was corrected into a wrong word occurred only once, according to our expectation, it is due to the low probability for this case.

In Figure 9.4.1 and 9.4.2 some results obtained by the postprocessing of classified words are presented. In Figure 9.4.1 the wrong classified characters, i.e., the confusion groups the classifier provides, are printed between curly brackets. Hence, the classifier provided us with more than one possibility of a character whenever it could not prove all the required properties for a character. This set contains a number of possible characters, all having some properties of the set of required properties. Thereafter an incorrect classified word is permuted within these confusion groups. The main reason why the classifier could not recognise a character, has been bad input from the ATARI ST. Characters laying on the borders of a region are sometimes demolished. The classifier is, however, mostly able to classify these characters in a certain group and to return this group, as we can see in Figure 9.4.2.

BY CRAWLS AND CREEPS
ICCA Journal June 1987

The layman is apt to think that scienc{g,c,e,N,S}
progresses {d,b}y leaps an{d,b} {d,b}oun{d,b}s, where each leap
an{d,b} every {d,b}oun{d,b} is a straight successor to
whatever has gone {d,b}efore. To the uninforme{d,b},
there is a simple un{d,b}roken succession from Newton
to Lorentz to Einstein. Similarly, even to th{g,c,e,N,S}
e{d,b}ucate{d,b}, Dalton his atomic hypothesis lea{d,b}{c,s}
naturally to Rutherfor{d,b} his first atomic
transmutation an{d,b} thence to Hiroshima an{d,b}
beyon{d,b}.

To those active in the fiel{d, b}, nothing coul{d,b}
{d,b}e further from the truth : they are aware that
science procee{d,b}s by crawls and creeps.

Figure 9.4.1: Text Obtained by the Classifier

In this text, 95% of the characters are recognised correctly. The other 5% non-classified characters consist mostly out of {b, d}. Please, notice that most of the others error occur on the right borders of the text. We already put forward the reason therefore: a bad quality input caused by the ATARI ST screendumps.

BY CRAWLS AND CREEPS
ICCA Journal June 1987

The layman is apt to think that science progresses by leaps and bounds, where each leap and every bound is a straight successor to whatever has gone before. To the uninformed, there is a simple unbroken succession from Newton to Lorentz to Einstein. Similarly, even to the educated, Dalton his atomic hypothesis leads naturally to Rutherford his first atomic transmutation and thence to Hiroshima and beyond.

To those active in the field, nothing could be further from the truth : they are aware that science proceeds by crawls and creeps.

Figure 9.4.2: Text Corrected by the Preprocessor

We may conclude, even against our previous expectations, that the postprocessor performs excellent. The 5% wrong classified characters were all corrected rightly. Hence, the recognition rate of our system for the test text is 100%.

9.4.2. Time Performance of the Postprocessing

The postprocessor has been running on a Gould Mainframe. The time required to initialize the postprocessor is 15 seconds. In this time, 24,000 words are read into the table, the pre- and suffix tables are build, and a list of 60 prime numbers is calculated. The postprocessor then needs 1 second to permute, strip and check 50 words. This speed is caused by the hash table. For each word to be checked, an average of 3 table checks is needed. We did not investigate the speed of the postprocessor very intensively, because this phase is carried out sequentially, and thus not of interest to any parallel influences.

9.4.3. Observations on Postprocessing

After notification of the results shown, we may state that postprocessing (i.e., spelling correction) is a powerful tool in Optical Character Recognition. It is fast, and it corrected (in the test text) all the words not classified correctly by the classifier. It might be interesting to return more than one possibility after a word is permuted, stripped and corrected. This causes however an extensive searching within the permutations to be generated. But, the speed of the postprocessor seems to be high enough to handle this. This concludes the results on postprocessing.

CHAPTER 10

CONCLUSIONS AND RECOMMENDATIONS

The Interpretation of the Results

ABSTRACT

In this Chapter, we shall discuss the results of Chapter 9 in relation to the goals aimed at. For completeness reasons only, in each Section we summarize first the observations of the phase concerned, i.e., the preprocessing, the classification, and the postprocessing phase. Of course, many details are apt for enhancements. Some new ideas are augmented to the conclusions in the form of recommendations. There is no separate Section, the recommendations are included in the Sections to which they belong. Finally, we provide the reader with an evaluation of the overall results and present our conclusions.

10.1. PROLEGOMENA

The overall performance of the Optical Character-Recognition System depends on its subsystems, being the Preprocessor, Classifier and the Postprocessor. The results presented in Chapter 9 have led us to conclusions concerning the performance of a single subsystem and its coherence with the other systems. Some of these conclusions will be obvious, while others will be very surprising. The conclusions are, so to speak, the final words on the quality of the results. From our experiences, we may opine that the implemented system is relevant and even innovative in the field of Optical Character-Recognition Systems. Being so, we can conclude that the integration of knowledge with existing techniques is valuable and that it is a adequate and welcome programming approach, being also innovative in the field of software engineering and applicable in other domains as well.

10.2. CONCLUSIONS ON PREPROCESSING

The Preprocessor has no knowledge of the low-level segmentation of characters as is proposed by Levine, described in Part One. This choice has been made because we wanted to concentrate on the classification of patterns instead of the segmentation of the lower-level objects. Moreover, the choice has led to the implementation of three straight-forward algorithms that serve their purpose very well. So far, the implemented graph algorithms meant for the so-called high-level segmentation, have not yet been used in the preprocessing process.

10.2.1. The Distribute Algorithm

The distribution of an image over the nodes in an allocated hypercube appears to be essential for the workload balancing in the system. The results show that in some cases the workload balancing is far from optimal, immediately causing a performance reduction of the system. For large images we expect that the variance in the workload balancing will not be as large as is the case with the input image described in Figure 9.3.1. We recommend that in order to achieve a good workload balancing the split criterium should be revised. A new split criterium could be to split the image into to images that both contain the same amount of areas. The reason that we did not implement this split criterium at first, is that we feared for overhead in the computation needed for the split operation. The current operation does not require a complete split for the image, contrary to the new criterium.

10.2.2. The Split Algorithm

The Split algorithm plays an important role in the distribution of the image, as well as in the further splitting of the sub-images residing on each node after distribution has taken place. In this way, the splitting of the entire image is done in parallel. However, results have shown that the time needed for splitting the image is of less importance than, e.g., the time to perform the Hough Transform or Classification. Subsequently, in view of our conclusions on the image distribution and workload balancing, we recommend that the image is split entirely by one node and that equal parts of the image with respect to the number of areas in a part are distributed over the nodes.

10.2.3. The Hough Transform

The rate at which lines are detected by applying the Hough Transform is remarkable high. The performance has been increased substantially by first splitting the image, and then applying the Hough Transform. By recording the average length, the orientation and the number of lines it is possible to create an area profile, that may be used to recognise patterns that have been recognised before. By defining the "scope" of the Hough Transform the abstraction levels in the vector representation of an area can be distinguished. The scope sets the minimum length and maximum slope of the lines that are detected. It is for instance possible to detect only large, horizontal and vertical lines for determining main characteristics. In case this information does not suffice, the "scope" of the Hough Transform may be enlarged so that more lines are detected. The suspected noise sensitiveness of the Hough Transform was not confirmed since our input contained no noise.

10.3. CONCLUSIONS ON CLASSIFICATION

In this Section we shall discuss the methods used in the classification phase on their ability to serve an Optical Character-Recognition System. This yields that the methods should be flexible, fast and easy to be implemented. First, the trigram analysis is investigated; next, the use of statistical knowledge will be considered; finally, the conclusions on the skeleton representation, the frames and the controller are given.

10.3.1. Trigram Analysis

According to our opinion, the trigram analysis is an easy, fast implementable and powerful method to optimise search processes in Optical Character Recognition. We have built the trigram tables without much implementation problems, due to the structure of trigram tables, but also because of the easy usable UNIX commands, which provided us with a large amount of words, all converted to a form the trigram-counting program could understand. After having experimented with the trigram tables in the classifier we may state that the hypotheses provided by this method are mostly correct. One disadvantage, however, must be admitted: the trigram tables used by our system only knew 26 characters (i.e., a to z). Hence, we were not able to provide the system with hypotheses for the upper-case characters and the read markers. In principle, this is not a serious problem as it might seem, because most of the characters in a text are lower-case characters. However, when an upper-case character occurs in the text, the search reduction is decreased heavily. This disadvantage of trigrams is caused by the fact that a trigram table consisting of all characters, possibly to occur in a text, would occupy more than 512 kByte.

Summarized we assert that:

- Trigram tables are easy to calculate, especially on a UNIX system.
- Trigram tables suppress search time in Optical Character-Recognition Systems to a long extent.
- Complete trigram tables, containing information on all the characters possibly to occur in a text (including upper-case characters and read markers) become too large to handle.

In summary, we consider the trigram analysis as a step forward in Optical Character Recognition.

10.3.2. Statistical Knowledge

In our Optical Character-Recognition System, we have provided the classifier with statistical knowledge on the fonts types and the font sizes by means of a table consisting of eight simple calculatable properties of a character (i.e., line lengths, number of horizontal lines, number of lines with a positive slope, etc.). With these eight properties we have been succesful in classifying the characters in a text. However, if the system does not know the font or the size, the classification based only upon statistical knowledge only failed whenever the types and sizes of the font differed too much. In our system, the structural classifier could then support the statistical knowledge classifier. The tables, with the statistical knowledge, contains 720 bytes each. Even with little memory, we can store a large amount of information on common-used fonts.

A classifier based upon structural methods only, has shown to be much time-consuming in an Optical Character-Recognition System. Whenever a font type is recognised and the size is determined, statistical methods could provide the classifier with sharp hypotheses, in order to speed-up the systems performance. From our point of view, we do not consider statistical knowledge (or methods) and structural knowledge to be mutually exclusive. They should be combined as much as possible, making the system flexible enough to recognise more fonts and sizes, and providing the system with enough fast computable knowledge to enhance the performance.

In short: the addition of statistical knowledge to the classifier did increase the performance of our Optical Character-Recognition System, without the need for a large amount of memory.

10.3.3. Skeleton Representation

For the description of characters we used the skeleton representation. The primary reason for it was that they could be a helpful tool in describing characters in a unique way and, for a large class of fonts as well as for all the sizes of a font.

For the size-independency problem, we have introduced a length indication for all *explicit edges*, which was relative to the longest and smallest lines in the output of the Hough Transform. We have defined the indications *small* and *long* as a single value and *medium* as a range between these values. This feature worked out well for different sizes of characters.

For the font-independency problem, the concept of skeletons helped us well. One small remark should be made: the skeleton concept suggested to be unique for some large classes of fonts; as it turned out, the representation was almost unique for the classes used; e.i., 3 fonts could be grouped into one class with unique representation, with the exception of a single character. So, font independency has been achieved by allowing one character to have different notations.

An other limitation for font independency is the use of the four possible line orientations. If a line, in a character is, for instance, labelled vertical, then the problem arises whether the line still is labelled vertical in the case of another font (think, for example of the *italic* print out of characters).

A major problem using the skeleton representation for characters, arises when building the graph structure, employed by the classifier. The graph we were able to construct contained but two plies.

The first reason for not obtaining a graph with more plies is:

- The skeleton representation of its own consists of only 7 descriptions and properties of lines. We discarded the *meet* since the output of the Hough transform might contain some inaccuracies. The functions for investigating own-defined properties, such as *surroundings* turned out to be very time-expensive. As an example we take the own-defined property *surrounding* and the character "B", which has two *surroundings* as a property. When using the skeleton representation, these surroundings will not easy be found, due to the fact that in the skeleton notation it is not expressed that there are explicit edges between the two vertices of the long vertical line and the left vertex of the middle horizontal line.

Hence, we defined only 6 properties for the characters. Naturally, this departure from the very best representation was known from the beginning, and was taken for granted. But there is another second, far more important, reason why we encountered problems in constructing the graph:

- All properties used, turned out to have a large amount of (different) values for all characters. It was therefore nearly impossible to define subclasses of characters with the same values for some properties. It is especially the non-existence of these subclasses, which is responsible for the lack of structure in the graph.

All in all, we want to emphasize that the use of skeletons as a font-independent description of characters, is an adequate method to gain insight into the intricacies of the pattern recognition problem, and to structure the problem of character representation for different fonts and sizes. An Optical Character-Recognition System, which has knowledge of the skeleton representation, can try to match a description of a character with the output of the Hough transform. If the input represents a legal character, one of the descriptions will match.

The combination of the skeleton representation and the decision graph, used for speeding up the matching process, did not meet our expectations, due to the problems arising when interpreting the skeleton representation, caused by the nature of a decision graph.

10.3.4. Frames

The frames are used for representing topological knowledge that is employed by the Classifier. The Frames and Frame handling functions have been implemented in LISP.

10.3.4.1. Frames for Representing Knowledge

Frames are data structures supporting *object-oriented* programming in a natural and direct way. A frame can be seen as an object with a number of properties (slots), each property having its own different aspects (facets). Especially, the inheritance mechanism supports the expression of dependency between two or more objects.

One of the major problems of incorporating knowledge into a computer system is how to represent the knowledge in a uniform and logical way. A well-known method to represent knowledge, is the **IF ... THEN ...** rule. The structure of the knowledge is then represented in the way the rules can be linked together by an inference mechanism.

To our opinion, the frame concept is a more powerful data structure for representing knowledge about objects. The inheritance mechanism can be used to make the structure of the knowledge explicit. Explicitly defining the structure in this way, gives more insight into the problems to be solved by the use of knowledge. A disadvantage of this representation of knowledge is, that when adding new knowledge one might have to revise the whole structure.

10.3.4.2. Frames Implemented in LISP

The implementation of frames in LISP, is straightforward when using the `assoc` function, but it is certainly not the most economic one with respect to memory usage.

Handling the frames can be done by defining a few small functions. This clearly enhances the idea of a structure of frames, as being an independent semantic database, which is a powerful concept we used eagerly.

10.3.5. The Classifier

The Classifier used three sorts of knowledge to identify the output of the Hough transform with a character:

- statistical knowledge
- contextual knowledge
- topological knowledge

The first two were used to determine hypotheses and the latter for verifying the hypotheses or classifying the input in the case there were no hypotheses available.

The combination of hypotheses of the trigram analysis and the hypotheses of the font tables turned out to be a highly relevant concept. It turned out to be so powerful that the combined hypotheses were true in almost all cases.

The classification of the input, using skeleton notation, was a very slow process in comparison with the determination of the hypothesis. The reasons for this slowness are : the implementation of the decision graph in LISP (the hypothesis determination was implemented in C) and the lack of structure in the decision graph (see Section 10.3.3).

Thus, we can state that, due to the slowness of the decision graph and the reliability of the obtained hypothesis, it is not intelligent to verify each hypothesis. The decision graph should still be included, in case the right font table was not known or no hypothesis from the trigram analyses could be obtained. Here we encounter the problem of balancing statistical and structural pattern recognition, as describes in Section 10.3.2.

The balance between the effort made by the classifier to recognise a character and the effort made by the postprocessor, which might correct a character wrongly classified, is worth to be investigated.

10.4. CONCLUSIONS ON POSTPROCESSING

As expected, spelling correction as a method used in the postprocessing phase, turned out to be well-suited to unravel the problems not solved by the classifier. In the classifier we used statistical knowledge on the various fonts, contextual knowledge within a character represented by the skeleton notation, and contextual knowledge about the sequence of characters by means of trigram analysis. If a character could still not be recognised with these types of knowledge, then the postprocessor can use syntactical knowledge, retrieved from an English Dictionary, to classify the character. The hashing of the word list and the pre- and suffix stripping can increase the performance of the spelling corrector to a large extent.

In our system, the postprocessor corrected all the characters (in the test text) not classified by the classifier, within acceptable time limits (i.e. 200 words per second). We conclude that spelling correction is a well-working addition to an Optical Character-Recognition System.

10.5. OVERALL CONCLUSIONS

The goals aimed at in the introduction of this treatise are being evaluated in this Section. We integrate the conclusions put forward in the previous Sections in order to obtain a number of valuable suggestions and conclusions in the fields of Parallel Processing, Software Engineering and Pattern Recognition with respect to Artificial Intelligence. Finally, the NCube as a whole will be evaluated.

10.5.1. Parallel Processing

On the subject of parallel processing we have separated our conclusions for respectively the Hypercube structure, the Pipeline and the Methodology followed to develop our Optical Character-Recognition System. These conclusions are:

- **The Hypercube Structure**

One of the main advantages of a hypercube structure is the slowly increasing maximum distance between the processors (the dimension) while the number of nodes increases linearly. We exploited this advantage by designing an algorithm that distributes the input data over the nodes in the hypercube structure. This, together with our low communication rate, led to a system having a linear speed-up with respect to the number of nodes that are active.

- **Pipeline Structure**

The output of the classifier is continuous, e.g., each node reports the characters that have already been recognised to the postprocessor, while the classification process is continuing. This interaction can be implemented as a pipeline; the front-end being the classifiers, the output-side is the connection to the postprocessor, viz. the second element in the pipe. The actual pipeline consists of the communication channels between nodes and host. Whenever one intends to exploit the pipeline structure, the input should be continuous and it should have a broad band-width in order to keep the pipe saturated. The continuous input is guaranteed since there is input from several nodes, which is not very likely to stagnate.

- **Methodology**

The distribution of the input image over the nodes, rather than the distribution of the classification knowledge can be considered as a well-working method in our Optical Character-Recognition System in view of the results presented in Chapter 9. Moreover, we like to put forward the following statement in relation to the parallelisation of Knowledge-Based Systems:

If, on the one hand, the amount of input data surpasses the amount of reasoning knowledge, the knowledge-based system is best parallelised by distributing the input data over several knowledge-based systems that all have the same knowledge. On the other hand, whenever the amount of knowledge surpasses the input data it is better to distribute the knowledge.

10.5.2. Software Engineering

Parallel computers are usually equipped with low-level programming languages, e.g., Fortran, C and Assembler. Good Software Engineering is achieved through a combination of a good programming environment (supporting tools, editors, compilers, debuggers etc.) and a supporting methodology. In our system both the methodology and the environment did not meet these requirements. The only languages supplied are C and Fortran. Although the AXIS operating system was a UNIX look-a-like, this did not mean that there were proper tools available to maintain programs and documentation, e.g., there was no *MAKE* nor an *SCCS* facility. The absence of a good methodology led us to intuitively develop our own method for parallelising a problem (see statement above). In order to provide ourselves with a proper parallel-programming workbench, we have designed and realized a distributed Lisp environment that can be controlled interactively. Interaction with the Lisp interpreters, is realized through a multi-window monitor system. The use of Lisp during the programming development phase, made it possible for us to enhance new ideas and prototypes rapidly. Whenever a prototype was accepted, it was implemented in C as a built-in Lisp function for speed-up reasons.

10.5.3. Pattern Recognition

A goal aimed at in the project was to investigate the relevance of artificial-intelligence techniques in the field of pattern recognition. Our system, was basically designed to work only with structural methods, optimised by trigram analysis. During the project, structural knowledge seemed to be slowing down the system. Therefore, statistical methods were combined through knowledge. The statistics consisted out of trigram analysis and feature descriptions of detected areas. The knowledge contained skeleton representation and spelling correction features. At the end of the project, recognition was achieved by statistics only. This resulted in a system, recognising 95% of the characters correctly within recognition times that are currently the state of the art. Whenever, a character cannot be recognised with statistical methods, the system switches automatically to structural methods in order to classify the character, and record new statistics. This resulted in a fast system, with a slower automatic-learning mode. The trigram analysis turned out to be a major contribution to optical character recognition. Therefore, we would like to recall that we do not consider statistical knowledge (or methods) and structural knowledge to be mutually exclusive. They should be combined as much as possible, making the system flexible enough to recognise more fonts and sizes, and providing the system with enough fast computable knowledge to enhance the performance.

10.5.4. The NCube

All the systems implementations were programmed in C. We did not use Fortran. Therefore, our evaluation of the NCube will be restricted to our own experiences with the hypercube communication functions as supported by the NCube-C Compiler. These functions are on a very low-level and are not very appropriate for programming large systems that contain a substantial part communication code. For such systems a programmer needs a library containing functions that offer a convenient interface to the lower level communication functions. We acknowledge this need and therefore have designed a node I/O library that maintains a set of logical channels. This library enabled the fast implementation of the Lisp interpreter on the nodes of the NCube. As a whole we find the NCube fit for research in the field of parallel processing, although the system still suffers from badly-documented childly diseases.

This concludes the last Chapter of this treatise.

PART 4

REFERENCES and APPENDICES

- **REFERENCES**

- **A: STATISTICAL KNOWLEDGE FOR TWO FONTS**
- **B: TRIGRAM TABLES**
- **C: PREFIXES AND SUFFIXES**
- **D: FRAMES**
- **E: SOURCE CODES**
- **F: FLISP IMPLEMENTATION DOCUMENTATION**

REFERENCES

- (1987). NCube 4+ Second Supercomputer of Delft University of Technology (in Dutch). *Automatiserings Gids*, 8 July.
- (1987). Delft University of Technology Installs NCube Supercomputer (in Dutch). *Computable*, 10 July.
- (1987). Starting with Parallel Processing (in Dutch). *DELTA*, TU-Delft, 10 October.
- Abelson, H. and Sussmann, G.J., (1985). *Structure and*
- Angell, R.C., Freund, G.E., Willett, P., (1983). *Automatic Spelling Correction Using A Trigram Similarity Measure. Information Processing and Management*, Vol. 19, No. 4, pp. 255-261.
- Asveld, P.R.J., Gaag, L.C. van der, Sommerhalder, R. (1984). *Theoretical Informatics (in Dutch)*. Classnotes, Department of Mathematics and Informatics, Delft University of Technology.
- Backer, E. (1981). Introduction in the Pattern Recognition (in Dutch). *Tijdschrift van het Nederlands Electronica en Radiogenootschap*. Vol. 46, no. 24, pp. 61-68.
- Ballard, D. (1981). Generalizing the Hough Transform to Detect Arbitrary Shapes. *Pattern Recognition*, Vol. 13, pp. 111-122.
- Ballard, D. and Brown, C. (1982). *Computer Vision*. Prentice Hall International.
- Barhen, J. and Palmer, J.F. (1986). The Hypercube in Robotics and Machine Intelligence. *Computers in Mechanical Engineering*.
- Ben-Ari, M. (1982). *Principles of Concurrent Programming*. Prentice Hall International, Englewood Cliffs, N.J.; London.
- Cohen, M.A.H. and Houck, M.E. (1986). Simulation of a Centralized-Control Parallel Knowledge-Based System. *Internal report*, Delft University of Technology. (in Dutch)
- Cox, C.H., Coueignoux, P., Blessler, B. and Eden, M. (1982). Skeletons: A Link Between Theoretical and Physical Letter Descriptions. *Pattern Recognition*, Vol. 15, pp. 11-22.
- Devijfer, P.A. and Kittler, J. (1982). *Pattern Recognition A Statistical Approach*, Prentice Hall International.
- Duda, R.O. and Hart, P.E. (1972). Use of the Hough Transform to Detect Lines and Curves in Pictures. *Communications of the ACM*, Vol. 15, pp. 11-15.
- Duda, R.O. and Hart, P.E. (1973). *Pattern Classification and Scene Analysis*, John Wiley and Sons Inc.
- Fu, K.S. (editor) (1977). *Syntactic Pattern Recognition, Applications*. Springer-Verlag. Groen, A., Herik,

- H.J. van den, Hofland, A.G., Kerckhoffs, E.J.H., Koppelaar, H., Stoop, J.C. and Varkevisser, P.R. (1985). Performance Model of a Parallel Knowledge-Based System. *Expert Systemen, special edition 'Systemica'*, Vol. 5, Nos. 4, 5 and 6, pp. 145-169.
- Groen A., Herik, H.J. van den, Hofland, A.G., Kerckhoffs, E.J.H., Stoop, J.C. and Varkevisser, P.R. (1986). The Integration of Simulation with Knowledge-based Systems, in *AI applied to simulation* (Edited by E.J.H. Kerckhoffs, G.C. Vansteenkiste and B.P. Zeigler), *Simulation Series*, Vol. 18, No. 1, pp. 189-197.
- Harris, J.F., Kittler, J., Llewellyn, B. and Preston, G. (1982). A Modulair System for Interpreting Binary Pixel Representations of Line-Structured Data. *Pattern Recognition, Theory and Applications*, J. Kittler, K.S. Fu and L.F. Pau (eds), D. Reidel Publishing Company.
- Henseler, J., Herik, H.J. van den and Verhoest, C.R.J. (1986). Strategieën in HYDRA, een real-time parallel expertsysteem (in Dutch). *Proceedings van NGI-SION 1986 Symposium Stimulerende Informatica*, pp. 359-369.
- Herik, H.J. van den, Stoop, J.C. and Varkevisser, P.R. (1987). Heuristics in the Abbot-Monk Problem. *Expert Systems and Artificial Intelligence in Decision Support Systems* (eds. H.G. Sol, C.A.Th. Takkenberg and P. de Vries Robbe), pp. 175-195. Reidel Publishing Company, Dordrecht, The Netherlands.
- Herik, H.J. van den, Henseler, J. and Verhoest C.R.J. (1987). Strategies in HYDRA, a Real-Time Parallel Expert System. *Internal Report*, Delft University of Technology).
- Herik, H.J. van der, Henseler, J., Hofland, A.G. and Verhoest, C.R.J. (1987). Parallel Processing in The Knowledge-Based System Hydra. *Preprints of the International Symposium on AI, Exper Systems and Languages in Modelling and Simulation*. IMACS, Barcelona, 2-4 June.
- Houck, M.E., (1987). *Graduation Report*. Department of Mathematics and Informatics, Delft University of Technology.
- Hayes, J.P., Mudge, T. and Stout, Q.F. (1986). A Microprocessor-based Hypercube Supercomputer. *IEEE Micro*, October, pp. 6-17.
- Hayes-Roth, F., Waterman, D.A. and Lenat, D. (1983). *Building Expert Systems*. Addison-Wesley Publishing Company.
- Hofland, A.G. (1987). *The Hydra Systems (in Dutch)*. Internal Report Department of Mathematics and Informatics, Delft University of Technology.
- Hossfeld, F. (1983). *Parallele Algorithmen*. Springer-Verlag.
- Hills, W.D. and Steele, G.L. (1986). Data Parallel Algorithms, *Communications of the ACM*, Vol. 29, pp. 1170-1183.
- INMos (1986). *The Transputer Family*. INMos Inc.
- Japan Travel, (1985). *Preliminary Report*. Department of Mathematics and Informatics, TU-Delft, Internal Report.
- Jurasek, D., Richardson, W. and Wilde, D. (1986). A Multiprocessor Design in Custom VLSI. *VLSI Systems Design*, June pp. 26-30.
- Kahan, S., Pavlidis, T. and Baird, H.S. (1987). On the Recognition of Printed Characters of Any Font and

- Size. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-9, pp. 274-288.
- Kerckhoffs, E.J.H. (1987). *CONVEX C-1 System Documentation (in Dutch)*, Classnotes, Delft University of Technology, Department of Mathematics and Informatics.
- Kernighan, B.W. and Ritchie, D.M. (1978). *The C Programming Language*. Prentice-Hall International.
- Kimme, C., Ballard, D. and Sklansky, J. (1975). Finding Circles by an Array of Accumulators, *Communications of the ACM*, Vol. 18, pp. 120-122.
- Kowalik J.S. (Editor) (1986). *Coupling Symbolic and Numerical Computing in Expert Systems*. North Holland Publishing Company.
- Krall, E.J. and McGeheraty, P.F. (1986). A Case Study of Parallel Execution of a Rule-Based Expert System. *International Journal of Parallel Programming*, Vol. 15, No. 1., pp. 5-32.
- Kushnir, M., Abe, K. and Matsumoto, K. (1985). Recognition of Hand-printed Hebrew Characters Using Features Selected in the Hough Transform Space, *Pattern Recognition*, Vol. 18, pp. 103-114.
- Landy, M.S. and Cohen, Y. (1985). Vectorgraph Coding, Efficient Coding of Line Drawings. *Computer Vision, Graphics and Image Processing*, Vol. 30, pp. 331-344.
- Mantas, J. (1986). An Overview of Character Recognition Methodologies. *Pattern Recognition*, Vol. 19, pp. 425-430.
- May, D. (1986). Occam 2 product definition (Preliminary), *Internal Report Inmos Inc.*.
- McIlroy, M.D. (1982). Development of a Spelling List. *IEEE Transactions on Communications*, Vol. COM-30, pp. 91-99.
- Minsky, M. (1975). A Framework for Representing Knowledge. *The Psychology of Computer Vision*. (Ed. P.H. Winston) McGraw-Hill Book Company.
- Mokhoff, N. (1986). Hypercube Architecture leads the Way for Commercial Supercomputers in Scientific Applications. *Computer Design*, May, pp. 28-30.
- Mudge, T.N. and Abdel-Rahman, T.S. (1987). Vision Algorithms for Hypercube Machines. *Journal of Parallel and Distributed Computing*, Vol. 4, pp. 79-94.
- Nazif, A.M. and Levine, M.D., (1984). Low Level Segmentation: An Expert System. *IEEE Transactions on Pattern Recognition and Machine Analysis*, Vol. PAMI-6, no. 5, September pp. 555-577.
- Nelson, H.L. (1985). Hash Tables in Cray Blitz. *ICCA-Journal*. Vol 8, No. 1, March, pp. 3-13.
- NCube Company (1986). NCube Handbook, version 1.1. *NCube Company Internal Report*.
- Reingold, E.M., Nievergelt, J. and Deo, N. (1977). *Combinatorial Algorithms, Theory and Practice*. Prentice Hall International.
- Rich, E. (1983). *Artificial Intelligence*. McGraw-Hill Book Co.
- Pavlidis, T. (1986). A Vectorizer and Feature Extractor for Document Recognition. *Computer Vision, Graphics and Image Processing*, Vol. 35, pp. 111-127.

- Pavlidis, T. (1977). *Structural Pattern Recognition*, Springer-Verlag.
- Pountain, D. (1986). Personal Supercomputers (Transputers). *Byte*, July, pp. 363-368.
- Pountain, D. (1986). Turbocharging Mandelbrot (Transputers and Occam). *Byte*, September, pp. 359-371.
- Schneider, D. (1987). The Programming Language Scheme. *AICOM*, Vol. 0, No. 1, August, pp. 17-27.
- Shridar, M. and Badrelin, A. (1986). Recognition of isolated and simply connected handwritten numerals. *Pattern Recognition*, Vol 19, pp. 1-12.
- Stanfill, C. and Kahle, B. (1986). Parallel Free-Text Search on the Connection Machine System. *Communications of the ACM*, Vol. 29, pp. 1229-1239.
- Stanfill, C. and Waltz, D. (1986). Toward Memory Based Reasoning. *Communications of the ACM*, Vol. 29, pp. 1213-1228.
- Stolfo, J.S. and Miranker, D.P. (1986). The DADO Production System Machine. *Journal of Parallel and Distributed Computing*, Vol. 3, pp. 269-296.
- Stoop, J.C. and Varkevisser, P.R. (1985). The implementatieon of HYDRA on a Parallel Computer. *Internal Report*, Delft University of Technology. (in Dutch)
- Stout, Q.F. (1987). Supporting Divide-and-Conquer Algorithms for Image Processing. *Journal of Parallel and Distributed Computing*, Vol. 4, pp. 95-115.
- Suen, C.Y., Berthod, M. and Mori, S. (1980). Automatic Recognition of Handprinted Characters The State of the Art. *Proceedings of the IEEE*, Vol. 68, pp. 469-487.
- Uhr, L. (1987). *Multi-Computer Architectures for Artificial Intelligence: Towards Fast, Robust, Parallel Systems*. John Wiley and Sons.
- Varkevisser, P.R. (1986). Simulation of a Asynchronous Parallel Computer. *Internal Report*, Delft University of Technology. (in Dutch)
- Weisbin, C.R. (1987). Intelligent-Machine Research at Ceasar. *The AI Magazine*, Spring, pp. 62-74.
- Winston, P.H. (1984). *Artificial Intelligence*. Addison-Wesley Publishing Company.
- Winston, P.H. (1975). Learning Structural Descriptions from Examples. *The Psychology of Computer Vision*. (Ed. P.H. Winston) McGraw-Hill Book Company.
- Winston, P.H., Horn, B.K.P. (1984). *Lisp*. Addison-Wesley Publishing Company.
- Wu, A.Y. (1985). Embedding of Tree Networks into Hypercubes. *Journal of Parallel and Distributed Computing*, Vol. 2, pp. 238-249.
- Zamora, E.M., Pollock, J.J., Zamora, A. (1981). The Use of Trigram Analysis for Spelling Error Detection, *Information Processing and Management*. Vol. 17, No. 6, pp. 305-316.

STATISTICAL KNOWLEDGE FOR TWO FONTS

APPENDIX A

STATISTICAL KNOWLEDGE FOR ATARI-ST SYSTEM FONT								
char	#l	#h	#v	#p	#n	aver	min	max
A	23	5	6	6	6	6	3	14
B	30	7	7	8	8	5	3	17
C	14	5	2	4	3	5	3	14
D	19	5	6	4	4	7	3	17
E	16	6	2	4	4	7	3	17
F	12	4	2	2	4	7	3	17
G	21	7	4	6	4	5	3	14
H	14	2	4	4	4	8	3	17
I	14	4	2	4	4	5	3	17
J	12	3	5	2	2	6	3	14
K	16	6	6	2	2	6	3	17
L	6	2	2	2	0	10	3	17
M	20	6	7	3	4	6	3	17
N	31	8	10	7	6	5	3	17
O	18	6	4	4	4	6	3	14
P	20	4	6	4	6	5	3	17
Q	20	6	4	6	4	6	3	16
R	27	5	9	6	7	5	3	17
S	28	8	8	8	4	4	3	8
T	8	2	2	2	2	9	3	17
U	13	4	5	2	2	7	3	15
V	19	4	8	4	3	5	3	14
W	22	5	8	5	4	6	3	17
X	20	5	11	2	2	5	3	11
Y	15	3	7	3	2	5	3	11
Z	19	7	6	4	2	6	3	11
0	18	6	4	4	4	5	3	14
1	14	4	2	4	4	6	3	17
2	20	6	5	4	5	5	3	11
3	22	6	6	5	5	5	3	11
4	26	6	6	8	6	5	3	17
5	20	6	5	6	3	6	3	11
6	23	6	4	8	5	6	3	17
7	13	4	5	2	2	6	3	11
8	27	6	7	8	6	5	3	17
9	24	6	6	6	6	5	3	11

CONTINUED STATISTICAL KNOWLEDGE FOR ATARI-ST SYSTEM FONT								
char	#l	#h	#v	#p	#n	aver	min	max
a	22	8	4	5	5	5	3	11
b	24	8	4	6	6	5	3	17
c	14	5	2	4	3	5	3	10
d	23	8	4	5	6	5	3	15
e	25	7	5	6	7	5	3	11
f	19	4	3	6	6	5	3	16
g	28	10	4	8	6	5	3	13
h	14	3	5	4	2	6	3	17
i	2	0	2	0	0	10	10	10
j	10	2	4	2	2	5	3	11
k	13	4	5	2	2	6	3	17
l	2	0	2	0	0	17	17	17
m	20	4	7	5	4	5	3	12
n	12	3	4	3	2	6	3	11
o	20	6	6	4	4	4	3	8
p	25	8	6	5	6	5	3	15
q	27	8	6	7	6	5	3	15
r	8	3	2	1	2	6	3	11
s	16	6	4	3	3	5	3	11
t	16	4	2	6	4	5	3	17
u	16	5	5	4	2	5	3	11
v	11	2	6	2	1	5	3	8
w	19	5	7	4	3	5	3	11
x	13	5	5	1	2	5	3	11
y	21	7	6	4	4	5	3	12
z	17	6	5	4	2	6	3	11
(	8	2	2	2	2	6	3	14
)	9	2	4	1	2	5	3	11
[	50	17	3	15	15	4	3	17
]	49	17	4	14	14	5	4	17
+	12	2	2	4	4	6	3	11
-	2	2	0	0	0	9	9	9
<	13	5	5	2	1	4	3	9
>	13	5	5	1	2	4	3	9
=	4	4	0	0	0	9	9	9
,	14	4	3	3	4	3	3	7
!	4	0	4	0	0	7	3	12
?	16	4	5	2	5	4	3	11
;	21	7	6	4	4	3	3	6
,	14	4	3	3	4	3	3	7
:	16	6	6	2	2	3	3	3
&	39	11	12	8	8	4	3	11
%	41	12	12	8	9	4	3	12
\$	50	6	9	17	18	5	3	17
*	31	7	7	9	8	6	3	14

STATISTICAL KNOWLEDGE FOR HELVETICA FONT								
char	#l	#h	#v	#p	#n	aver	min	max
A	39	11	13	8	7	5	4	11
B	30	6	6	10	8	5	3	14
C	26	8	8	5	5	5	3	11
D	21	5	4	6	6	6	3	14
E	34	6	8	10	10	5	3	14
F	30	6	6	10	8	5	3	14
G	29	8	8	8	5	4	3	11
H	38	10	4	12	12	5	3	14
I	14	4	2	4	4	5	3	14
J	21	7	5	5	4	4	3	11
K	44	15	11	9	9	5	3	14
L	18	4	4	4	6	5	3	14
M	37	13	14	5	5	6	5	14
N	47	15	11	10	11	5	3	14
O	18	6	4	4	4	5	3	11
P	25	6	4	8	7	5	3	14
Q	30	10	7	7	6	5	3	11
R	35	8	8	12	7	4	3	14
S	33	10	9	11	3	4	3	11
T	22	4	6	6	6	5	3	14
U	25	8	5	6	6	5	3	12
V	34	8	10	8	8	4	3	8
W	43	11	14	9	9	5	4	12
X	45	19	14	6	6	6	4	13
Y	34	12	11	5	6	4	3	11
Z	43	14	16	6	7	5	3	12
0	18	6	4	4	4	5	3	11
1	13	4	2	4	3	5	3	14
2	26	7	7	4	8	4	3	9
3	23	6	5	6	6	4	3	14
4	25	6	5	8	6	5	3	14
5	24	8	6	6	4	5	3	9
6	28	9	5	8	6	4	3	11
7	23	5	8	2	8	4	3	9
8	28	6	6	8	8	5	3	14
9	26	8	5	6	7	5	3	12

CONTINUED STATISTICAL KNOWLEDGE FOR HELVETICA FONT								
char	#l	#h	#v	#p	#n	aver	min	max
a	28	7	6	8	7	4	3	10
b	24	7	5	7	5	5	3	14
c	13	4	2	4	3	5	3	11
d	25	7	5	7	6	5	3	14
e	23	6	4	6	7	4	3	11
f	23	5	3	7	8	4	3	13
g	39	9	8	10	12	4	3	9
h	30	9	5	9	7	5	3	14
i	12	4	2	4	2	4	3	10
j	12	2	3	4	3	4	3	11
k	26	9	5	7	5	5	3	14
l	13	4	2	4	3	5	3	14
m	39	9	7	13	10	4	3	12
n	30	7	5	10	8	4	3	11
o	16	4	4	4	4	6	3	11
p	35	10	6	10	9	4	3	14
q	35	10	6	9	10	4	3	14
r	19	4	4	6	5	4	3	11
s	22	6	6	7	3	4	3	8
t	16	2	3	5	6	4	3	12
u	18	6	4	6	2	4	3	11
v	21	4	9	4	4	4	3	8
w	34	4	14	8	8	4	3	10
x	20	5	9	3	3	3	3	8
y	26	4	11	5	6	3	3	6
z	17	6	5	4	2	5	3	9
(	8	2	2	2	2	6	3	14
)	9	2	4	1	2	5	3	11
[	41	14	3	12	12	4	3	14
]	41	14	3	12	12	4	3	14
+	12	2	2	4	4	6	3	11
-	2	2	0	0	0	8	8	8
<	21	7	5	5	4	4	3	7
>	21	7	5	4	5	4	3	7
=	4	4	0	0	0	9	9	9
'	14	4	3	3	4	3	3	7
!	2	0	2	0	0	11	11	11
?	13	3	5	1	4	4	3	6
;	25	8	6	5	6	3	3	7
,	14	4	3	3	4	3	3	7
:	22	8	6	4	4	3	3	4
&	45	14	14	6	11	5	3	11
\$	34	11	12	4	7	5	4	14
*	48	8	8	18	14	5	3	17

TRIGRAM ANALYSIS

APPENDIX B

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
aba	297	afr	2	akw	5	anv	3	ast	434	axe	6
abe	9	afs	14	ak-	39	anw	11	asu	155	axi	154
abf	14	aft	188	ala	75	any	286	asv	4	ax-	28
abi	58	afw	1	alb	39	an-	2083	asy	24	aya	6
abl	790	afz	1	alc	13	aoi	1	as-	1621	ayb	2
abo	215	af-	8	ald	8	ao-	3	ata	569	aye	141
abr	6	aga	162	ale	467	apa	100	atc	68	ayi	61
abs	100	agd	2	alf	34	ape	273	ate	1931	aym	5
abu	21	age	455	alg	130	aph	75	atg	4	ays	81
aby	2	agi	15	ali	285	api	40	ath	189	ay-	745
ab-	5	agl	1	alk	5	apl	171	ati	2142	aza	14
aca	1	agn	16	alm	24	apo	19	atk	57	aze	1
ace	284	ago	12	alo	145	apr	184	atl	18	azi	17
ach	645	ag-	129	ip	2	aps	59	atm	8	az-	1
aci	61	ags	4	alr	34	apt	114	atn	8	a--	5039
ack	722	agu	5	als	319	apu	9	ato	98	bab	24
acl	14	agy	2	alt	94	apy	4	atr	75	bac	135
acm	17	ag-	42	alu	314	ap-	68	ats	24	bad	32
aco	93	ahe	9	alv	35	aqu	2	atu	151	bag	1
acq	11	ah-	1	alw	30	ara	446	aty	12	bak	7
acr	10	aia	1	aly	596	arb	46	atz	1	bal	30
acs	1	aic	4	al-	2455	arc	651	at-	1691	ban	42
act	609	aid	62	ama	8	ard	397	aud	8	bar	34
acu	57	aie	3	amb	22	are	1144	auf	8	bas	596
acy	14	aig	4	ame	1139	arg	101	aug	16	bat	23
ac-	22	ail	157	amh	6	ari	317	aul	4	bau	4
ada	43	aim	44	ami	118	ark	105	aum	2	ba-	10
ade	208	ain	798	amk	1	arl	143	aun	6	bco	36
adi	204	aio	2	aml	1	arm	17	aus	135	bc-	9
adj	14	air	86	amo	101	arn	160	aut	104	bdo	13
adl	13	ais	52	amp	222	aro	59	auw	1	bd-	29
adm	48	ait	72	amr	2	arp	12	au-	10	bea	39
ado	20	aiv	15	ams	158	ars	134	ava	93	beb	15
adr	31	ai-	42	amu	31	art	562	ave	568	bec	119
ads	34	aja	3	amv	4	aru	2	avi	135	bed	72
adt	16	aje	4	am-	489	arv	3	avo	60	bef	64
adu	2	ajo	40	ana	306	arw	2	avr	6	beg	37
adv	237	ajr	11	anc	527	arx	6	avs	1	beh	43
ady	38	aka	1	and	2787	ary	202	avy	1	bei	114
ad-	342	akb	2	ane	58	ar-	404	av-	4	bek	5
aef	28	ake	224	ang	226	asa	27	awa	30	bel	165
aer	5	akg	2	ani	154	asb	3	awb	10	ben	347
aes	1	akh	18	ank	201	asc	11	awh	1	bep	1
ae-	7	aki	33	anl	3	ase	974	awi	4	ber	684
afa	3	akm	1	anm	1	asg	23	awl	5	bes	125
afe	49	akn	8	ano	179	ash	26	awn	394	bet	226
afg	4	aks	1	anq	1	asi	304	aws	25	beu	1
afh	1	akt	16	ans	343	ask	87	aw-	72	bey	50
afi	3	aku	6	ant	538	aso	161	axb	3	bez	10
afo	9	akv	3	anu	65	asp	37	axd	1	be-	1239

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
bf-	25	boi	1	buy	1	cha	473	clo	47	ctu	225
bga	7	bok	44	bu-	4	chb	66	cls	5	ctv	3
bgo	3	bol	23	bvi	35	che	1131	clu	203	cty	1
bg-	23	bon	40	bv-	5	chi	530	cmu	2	ct-	499
bh-	11	bor	68	bxa	1	chk	15	cm-	17	cui	3
bia	9	bos	35	bxb	3	chl	2	cnt	3	cul	171
bib	3	bot	112	bxc	14	chm	7	coa	4	cum	33
bid	6	bou	131	bxd	5	chn	170	cod	46	cun	3
bie	18	bov	93	bxe	11	cho	83	cof	2	cuo	6
big	24	box	7	bxf	17	chr	42	cog	59	cup	30
bij	51	bo-	49	bxg	11	chs	9	coh	11	cur	261
bil	168	bpr	8	bXH	5	cht	26	coi	1	cus	83
bin	87	bp-	9	bxq	6	chu	17	cok	1	cut	59
bio	6	bq-	10	bx-	3	chv	1	col	82	cva	7
bip	1	bra	132	byg	2	chy	2	com	1219	cxb	5
bis	90	bre	27	byt	93	ch-	1828	con	2121	cxd	9
bit	116	bri	111	by-	1083	cia	266	cop	53	cx-	5
bi-	12	bro	12	b--	1323	cib	4	cor	387	cyb	12
bje	55	bru	69	cab	47	cid	48	cos	25	cyp	1
bkb	3	bry	1	cac	14	cie	182	cot	8	cyr	3
bkd	2	br-	77	cad	9	cif	86	cou	270	cy-	79
bke	1	bsc	9	cai	7	cil	38	cov	36	cz-	1
bkn	83	bse	94	cal	473	cim	4	cow	22	c--	1942
bko	12	bsi	2	cam	13	cin	65	co-	29	dab	12
bkq	4	bso	11	can	588	cio	8	cph	12	dac	5
bkr	14	bsp	4	cap	139	cip	75	cpr	4	dag	8
bks	6	bst	110	car	67	cir	20	cpu	12	daj	3
bKx	21	bsu	1	cas	194	cis	77	cp-	8	dal	19
bk-	106	bs-	11	cat	331	cit	48	cqu	11	dam	55
bla	379	bta	69	cau	106	ciz	2	cra	38	dan	59
ble	1095	bte	1	cav	1	ci-	23	cre	208	dap	36
bli	215	btf	3	ca-	186	cka	13	cri	184	dar	18
blj	6	btl	11	cb-	3	cke	62	cro	92	das	1
blk	3	btm	4	cd-	3	ckf	1	cru	14	dat	484
blo	95	bto	9	ceb	1	ckg	8	cry	7	dau	5
blt	1	btr	11	ced	355	ckh	29	cr-	6	dav	45
blu	14	bts	6	cef	1	cki	42	csa	3	day	21
blx	6	bt-	4	cei	63	ckl	24	csi	2	da-	84
bly	96	buc	7	cek	5	ckm	20	csr	1	dba	5
bl-	25	bud	4	cel	111	ckn	26	cs-	191	dbl	9
bmi	17	bue	2	cem	47	cko	2	cta	13	dbm	21
bm-	27	buf	6	cen	322	cks	84	ctb	5	dbo	3
bne	1	bug	3	cep	126	ckt	28	cte	247	dbu	5
bng	1	bui	55	cer	109	ckv	1	cti	990	db-	48
bnk	9	bul	20	ces	1036	ckw	40	ctl	43	dch	8
bn-	8	bun	5	cet	11	cky	1	ctm	2	dco	2
boa	85	buq	1	ce-	1458	ck-	683	ctn	3	dc-	2
bob	4	bur	28	cf-	25	cla	112	cto	118	dea	104
bod	11	bus	13	cgi	1	cle	135	ctr	26	deb	12
boe	2	but	349	cgr	8	cli	14	cts	104	dec	183

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
ded	338	dja	3	ds-	355	ebe	49	ege	77	elg	1
def	520	djo	1	dth	45	ebi	14	egf	5	elh	5
deg	17	dju	10	dti	1	ebl	1	egi	106	eli	399
dek	14	dko	1	dt-	25	ebo	3	egl	29	elj	6
del	379	dle	107	dua	28	ebr	29	egn	6	elk	9
dem	32	dli	10	dub	3	ebt	1	ego	17	elm	9
den	525	dlu	1	duc	283	eby	8	egr	28	eln	27
deo	1	dly	35	due	78	eca	98	egt	1	elo	166
dep	319	dli	16	dui	1	ece	382	egu	13	elp	66
deq	27	dma	82	dul	75	ech	259	egy	66	els	77
der	703	dme	1	dun	8	eci	313	eg-	42	elt	27
des	321	dmi	48	duo	1	eck	137	eha	29	elu	8
det	103	dmo	18	dup	39	ecl	3	ehe	15	elv	32
dev	92	dm-	8	dur	107	eco	296	ehi	9	elw	1
dew	6	dne	8	dus	16	ecr	31	eho	3	ely	370
dex	28	dob	4	dut	37	ect	939	ehr	5	elz	2
dez	10	doc	26	dva	176	ecu	66	ehu	1	el-	592
de-	795	doe	115	dve	11	ecy	3	eh-	1	ema	132
dfa	1	doi	7	dvi	44	ec-	45	eid	47	emb	103
dfo	3	dol	8	dvo	10	eda	2	eig	114	emc	1
dfu	3	dom	116	dv-	24	edb	3	eih	3	emd	24
df-	13	don	97	dwa	46	ede	132	eil	5	eme	428
dga	397	dop	16	dwe	2	edf	3	eim	5	emi	56
dge	459	dor	48	dwi	17	edg	456	ein	173	emo	71
dgi	2	dos	6	dwo	3	edi	278	eir	177	emp	97
dgm	6	dot	3	dwr	1	edk	1	eis	11	ems	220
dg-	28	dou	44	dx-	9	edl	9	eit	116	emt	3
dha	4	dov	5	dxi	6	edm	18	eiv	59	emy	13
dhe	1	dow	28	dyk	1	edn	1	ei-	8	em-	443
dh-	31	doz	2	dyn	14	edo	9	eje	2	ena	47
dia	194	do-	78	dy-	98	edr	4	ejo	1	enb	20
dib	1	dpo	1	dza	8	eds	68	eka	1	enc	791
dic	96	dpr	1	d--	10858	edt	5	ekb	3	end	1020
did	35	dp-	15	eab	6	edu	189	eke	34	ene	366
die	140	dq-	9	eac	257	edy	2	ekg	1	enf	1
dif	238	dra	358	ead	274	ed-	4628	eki	9	eng	196
dig	25	dre	62	eaf	11	efa	8	ekl	1	enh	15
dij	3	dri	38	eag	3	efe	235	ekn	1	eni	142
dil	8	dro	28	eak	87	efi	216	eko	8	enj	4
dim	14	drs	4	eal	157	efk	1	ekp	1	enk	14
din	483	dru	2	eam	25	efl	30	ekr	1	enl	5
dio	15	dry	2	ean	122	efn	2	eks	12	enm	1
dip	4	dr-	95	eap	7	efo	163	ekt	11	eno	99
dir	87	dsc	5	ear	892	efr	4	ek-	42	enr	7
dis	395	dse	2	eas	480	efs	5	ela	328	ens	236
dit	279	dsh	9	eat	215	eft	24	elb	3	ent	2638
diu	13	dsi	1	eau	2	efu	78	elc	3	enu	38
div	67	dso	6	eav	11	ef-	343	eld	136	env	17
dix	106	dsp	12	ea-	61	ega	134	ele	537	enx	1
diz	3	dst	12	eba	16	egd	2	elf	159	eny	3

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
enz	13	esc	164	evo	31	fas	39	fnu	2	gat	54
en-	2512	esd	3	evr	2	fat	3	foc	12	gau	5
eof	1	ese	978	evs	6	fau	1	fol	220	gav	5
eog	1	esf	17	evy	34	fav	15	fom	1	gaz	10
eom	1	esh	17	ev-	8	fa-	1	fon	89	gba	5
eon	4	esi	452	ewa	6	fde	4	for	2141	gde	1
eop	6	esk	19	ewb	28	fdz	1	fot	6	gd-	6
eor	96	esl	10	ewc	3	fd-	17	fou	135	gea	8
eou	29	esm	7	ewe	125	fea	63	fo-	7	geb	31
eo-	38	esn	3	ewf	1	feb	8	fra	34	gec	3
epa	175	eso	149	ewh	35	fec	169	fre	97	ged	76
epe	143	esp	145	ewi	14	fed	15	frg	1	geg	9
eph	136	est	824	ewl	3	fei	6	fri	13	geh	4
epi	25	esu	412	ewo	17	fel	67	fro	647	gei	6
epl	17	es-	5115	ewp	11	fen	68	fru	7	gek	3
epo	94	eta	118	ewr	13	fer	489	fsc	1	gel	45
epr	108	etc	41	ews	25	fes	40	fst	16	gem	39
eps	10	ete	260	ewt	1	fet	3	fs-	45	gen	546
ept	244	etf	5	ew-	216	few	67	fte	209	geo	6
epu	11	etg	3	exa	243	fe-	32	fth	19	gep	5
ep-	60	eth	244	exc	299	fge	4	fts	2	ger	160
eqn	5	eti	153	exd	8	fg-	1	ftw	53	ges	150
equ	419	etj	5	exe	53	fha	1	fty	2	get	76
eq-	12	etl	8	exf	8	fh-	1	ft-	180	gev	30
era	577	etm	2	exh	22	fia	7	fug	2	gew	11
erb	29	eto	104	exi	109	fic	377	ful	198	ge-	908
erc	140	etr	73	exp	577	fid	79	fun	178	gfo	1
erd	99	ets	70	ext	229	fie	136	fur	79	gfr	5
ere	1252	etu	13	exu	1	fif	25	fus	5	gfu	10
erf	337	etw	131	ex-	87	fig	231	fut	82	gf-	32
erg	110	ety	34	eyd	4	fil	73	fuz	30	ghb	8
erh	125	etz	11	eye	1	fin	303	fu-	1	ghe	38
eri	678	et-	618	eyf	3	fip	4	fwe	1	ghi	6
erj	4	eud	5	eyi	6	fir	331	fwi	1	ghl	32
erk	50	eue	4	eyn	5	fis	7	fxe	4	ghn	1
erl	196	eul	4	eyo	50	fit	26	fxg	2	gho	25
erm	243	eum	20	eys	3	fiv	48	fyi	21	ghs	3
ern	229	eun	4	eyw	3	fix	14	fy-	24	ght	390
ero	94	eup	1	ey-	176	fiz	16	fzo	1	gh-	179
erp	66	eur	68	eze	24	fjc	1	f--	6826	gib	6
ers	1109	eus	1	ezi	1	fke	1	gab	2	gic	36
ert	464	eut	1	ez-	1	fla	77	gag	3	gid	1
eru	13	euw	8	e--	28159	fle	41	gah	2	gie	51
erv	118	euz	2	fac	268	fli	3	gai	178	gif	1
erw	39	eu-	1	fai	99	flo	22	gal	52	gil	1
ery	146	eva	224	fak	6	flu	22	gam	802	gin	181
erz	65	eve	759	fal	23	fly	10	gan	86	gio	7
er-	3889	evi	235	fam	24	fl-	18	gap	1	gir	1
esa	84	evm	9	fan	17	fma	3	gar	81	gis	27
esb	4	evn	2	far	59	fne	2	gas	4	git	9

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
giu	1	gt-	9	heg	7	hni	84	hve	1	idl	81
giv	182	gua	20	heh	1	hno	86	hwa	5	idm	1
giz	6	gue	31	hei	225	hn-	7	hwe	6	ido	3
gi-	5	gui	74	hek	3	hoc	35	hwh	7	idr	6
gla	107	gul	13	hel	93	hod	84	hxg	5	ids	8
gle	100	gum	13	hem	121	hoe	46	hya	10	idt	33
gli	24	gun	1	hen	482	hof	95	hyb	2	idu	28
glo	3	guo	4	heo	81	hog	28	hyd	184	idv	6
gly	45	gup	15	heq	3	hoi	15	hym	5	id-	190
gl-	1	gur	217	her	1374	hol	83	hyp	93	iec	125
gma	11	gus	2	hes	1224	hom	144	hys	3	ied	167
gmb	1	gwa	1	het	178	hon	16	hy-	28	ief	25
gme	12	gxe	1	heu	42	hop	94	hz-	323	ieg	5
gms	1	gxf	10	hev	7	hor	197	h--	4718	iek	5
gm-	8	gxh	2	hex	5	hos	129	iab	37	iel	115
gna	26	gya	2	hey	129	hot	36	iag	104	iem	10
gne	78	gym	3	he-	8512	hou	442	ial	589	ien	247
gni	102	gyu	1	hfl	8	hov	2	iam	16	ieo	5
gno	21	gy-	157	hf-	9	how	356	ian	74	ier	175
gns	4	g--	5101	hgh	1	ho-	49	iar	14	ies	421
gn-	57	hac	5	hg-	2	hpi	9	ias	9	iet	52
goa	53	had	96	hia	5	hra	9	iat	181	ieu	6
god	81	hae	28	hib	16	hrd	1	iav	1	iev	90
goe	4	haf	9	hic	557	hre	184	ia-	53	iew	73
goi	4	hag	2	hid	6	hri	13	ibe	98	ie-	224
gon	10	hai	31	hie	173	hrm	9	ibi	94	ifa	1
gor	136	hak	14	hif	1	hro	84	ibl	219	ife	171
gos	6	hal	110	hig	62	hrs	6	ibm	6	ifi	321
got	1	ham	68	hij	8	hru	1	ibn	1	ifo	12
gou	13	han	500	hik	28	hr-	11	ibr	21	ifs	6
gov	10	hap	149	hil	147	hsc	9	ibu	85	ift	31
gow	24	har	191	him	28	hsh	1	ib-	2	ify	39
go-	30	has	525	hin	379	hs-	39	ica	885	if-	300
gra	715	hat	1215	hio	5	hte	15	ice	195	iga	45
gre	165	hau	12	hip	72	htf	8	ich	673	igd	1
gri	9	hav	486	hir	21	hth	1	ici	248	ige	147
gro	149	hay	21	his	1118	hti	10	ick	29	igh	408
gru	3	ha-	47	hit	459	htl	6	icl	47	igi	39
gr-	1	hbe	65	hi-	32	hts	56	ico	10	igl	7
gsa	9	hbo	9	hk-	15	htw	3	icr	35	igm	9
gsc	1	hbr	1	hle	11	ht-	319	ics	138	ign	199
gse	3	hch	9	hli	1	hug	3	ict	92	igo	2
gsi	1	hdc	6	hly	33	hui	2	icu	115	igs	1
gsp	2	hdr	1	hl-	2	huj	1	icz	1	igt	2
gsr	2	hd-	1	hma	18	hul	15	ic-	242	igu	226
gst	7	hea	30	hme	7	hum	144	ida	31	igy	3
gs-	139	heb	6	hms	29	hun	37	ide	760	ig-	35
gta	3	hec	121	hm-	75	hur	2	idg	4	ihe	3
gth	49	hed	242	hna	5	hus	67	idh	6	iho	1
gto	2	hef	3	hne	1	hu-	2	idi	123	ija	3

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
ijc	6	inc	450	irl	7	ive	978	jl-	21	kd-	297
ijd	13	ind	376	irm	31	ivi	97	jnd	1	ked	93
ijf	1	ine	855	irn	50	ivo	1	jnl	3	kef	3
ijg	4	inf	271	iro	45	ivy	1	jn-	48	keg	1
ijh	1	ing	3802	irs	310	iv-	45	job	15	kel	44
ijk	77	inh	8	irt	7	ixe	12	joe	1	kem	3
ijl	21	ini	412	ir-	228	ixi	1	joh	7	ken	157
ijn	52	ink	38	isa	61	ixt	7	joi	7	kep	35
ijp	1	inl	29	isb	8	ix-	147	jon	23	ker	131
ijs	5	inm	3	isc	172	iya	4	jor	40	kes	41
ijv	10	ino	42	ise	274	iza	103	jos	7	ket	20
ijz	3	inp	28	isf	14	ize	252	jou	150	keu	3
ij-	31	ins	480	isg	21	izi	25	joy	6	kev	35
ika	5	int	1035	ish	332	izk	12	jp-	1	kew	69
ike	100	inu	63	isi	136	izm	4	jr-	13	key	8
iki	26	inv	129	isk	35	izo	18	js-	5	ke-	587
iks	2	iny	3	isl	4	iz-	36	jub	6	kfo	1
ikt	6	in-	3706	ism	87	i--	920	jud	17	kf-	398
iku	5	ioc	8	isn	3	jac	4	jul	21	kge	1
ik-	191	iod	7	iso	18	jah	2	jum	3	kgr	10
ila	143	iog	3	isp	40	jan	20	jun	19	kg-	337
ild	65	iol	25	isq	10	jap	9	jus	43	khe	7
ile	214	iom	6	isr	20	jar	14	jvi	8	khm	11
ilf	1	ion	4513	ist	684	ja-	12	jvo	2	kho	30
ili	294	ior	154	isu	63	jca	6	jzo	3	kh-	367
ilj	4	ios	2	isv	4	jc-	1	j--	406	kie	1
ilk	5	iot	4	isy	2	jda	2	kab	17	kil	33
ilm	2	iou	191	is-	3437	jde	2	kac	6	kin	358
iln	1	iow	1	ita	86	jdr	7	kad	1	kir	3
ilo	17	io-	26	itc	8	jds	1	kag	4	kis	10
ils	34	ipa	32	ite	809	jd-	1	kai	12	kit	2
ilt	19	ipd	6	itf	11	jec	141	kak	1	ki-	22
ilu	17	ipe	16	itg	4	jed	1	kan	6	kle	8
ilv	2	iph	9	ith	1023	jek	6	kar	13	kli	25
ily	63	ipi	10	iti	1629	jeo	2	kas	12	klo	4
il-	434	ipl	47	itl	37	jer	5	kat	5	kly	11
ima	486	ipm	4	itm	20	je-	3	kau	3	kl-	1
imb	4	ipr	10	ito	82	jft	1	kay	10	kma	22
ime	603	ips	17	itp	4	jgb	2	ka-	222	km-	1
imi	312	ipt	35	itr	39	jge	1	kbe	4	kna	6
imo	20	ipu	18	its	429	jgt	1	kbi	1	knc	6
imp	489	ip-	73	itu	125	jho	1	kbk	108	kne	9
ims	14	iqu	68	itw	4	jij	1	kbl	3	kni	69
imt	1	ira	18	ity	590	ji-	4	kbn	14	knk	70
imu	189	irb	1	itz	56	jke	34	kbp	5	kno	553
imx	9	irc	22	it-	946	jkh	7	kby	6	knu	5
imz	3	ird	15	ium	16	jki	2	kb-	281	knw	4
im-	46	ire	242	ius	10	jks	1	kch	9	kn-	99
ina	264	iri	17	iva	95	jkt	2	kct	6	koj	4
inb	20	irk	3	ivc	1	jk-	31	kc-	311	kok	5

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
kol	1	kvi	3	ldo	4	lih	1	lpb	12	lvo	26
kom	36	kv-	1	ldr	8	lij	49	lpf	7	lwa	30
kon	40	kwa	40	lds	26	lik	78	lph	31	lwe	1
kop	47	kwe	5	ldt	4	lim	100	lpm	10	lxw	6
kor	5	kxb	9	ldw	1	lin	358	lpr	14	lya	9
kos	3	kxc	8	ldy	1	lio	14	lps	1	lyb	6
kou	2	kxd	2	ld-	696	lip	9	lp-	31	lyh	3
kov	7	kxe	3	lea	380	lis	335	lre	33	lyi	26
koz	2	kxf	2	leb	5	lit	346	lri	1	lym	43
ko-	53	kxg	6	lec	196	liv	12	lse	29	lyn	3
kpk	20	kxh	1	led	634	liz	187	lsh	3	lys	143
kpo	3	kxw	12	lef	23	li-	7	lsi	13	lyt	15
kpu	1	kyb	9	leg	116	lja	10	lsk	3	lyz	16
kp-	77	kyi	4	leh	5	lju	6	lsl	1	ly-	1970
kqb	4	kyo	1	lei	42	lj-	6	lsm	5	lza	1
kqk	66	ky-	32	lek	4	lka	1	lsn	2	lz-	1
kqn	4	kza	1	lel	268	lke	11	lso	189	l--	4372
kqp	22	k--	1982	lem	375	lki	4	lsr	2	mab	1
kqr	8	lab	127	len	269	lko	1	lst	13	mac	162
kq-	38	lac	473	lep	4	lky	1	lsw	1	mad	59
kra	6	lad	11	ler	156	lk-	9	ls-	228	maf	1
krb	14	lae	2	les	455	lma	4	lta	30	mag	34
kre	5	laf	3	let	158	lme	4	ltc	1	mai	222
kri	3	lag	27	leu	3	lmi	4	ltd	7	maj	40
krk	30	lai	63	lev	197	lmo	18	lte	75	mak	81
krm	4	lam	2	lew	6	lms	4	lth	47	mal	460
kro	4	lan	376	lex	111	lmt	5	lti	105	mam	1
krp	13	lap	4	ley	8	lmu	4	ltm	1	man	568
krx	4	lar	244	lez	1	lm-	9	lto	1	map	22
kr-	67	las	160	le-	2024	lne	30	ltr	13	mar	188
ksc	2	lat	595	lfd	2	lny	1	lts	259	mas	224
ksh	5	lau	15	lfi	5	loa	38	ltu	2	mat	1060
ksk	6	lav	12	lfo	1	lob	3	lty	29	max	160
kso	2	law	30	lfs	1	loc	131	ltz	2	may	249
ksp	7	lax	6	lft	91	loe	3	lt-	168	maz	1
kst	1	lay	501	lfu	1	log	172	lua	169	ma-	7
ksv	2	laz	14	lf-	102	loi	19	luc	7	mba	3
ks-	220	la-	77	lga	1	lok	42	lud	127	mbe	456
kte	2	lbe	42	lge	23	lom	7	lue	159	mbh	1
kth	3	lbs	1	lgi	1	lon	146	lum	11	mbi	87
kti	13	lb-	99	lgo	117	lop	74	lun	13	mbi	7
kto	2	lca	2	lgt	3	loq	2	luo	8	mbo	18
ktr	30	lch	3	lha	5	lor	19	lur	14	mbr	1
kt-	24	lco	3	lia	60	los	215	lus	96	mbs	1
kud	1	lcu	13	lib	34	lot	37	lut	111	mbu	4
kul	23	lc-	3	lic	228	lou	36	lux	1	mb-	9
kum	1	ldc	2	lid	30	lov	3	lu-	3	mcg	9
kun	22	lde	24	lie	191	low	407	lva	3	mck	10
kup	2	ldi	37	lif	35	loy	12	lve	109	mco	6
kve	1	ldm	6	lig	127	lo-	35	lvi	52	mcs	1

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
mcv	7	mni	14	mu-	8	ndi	380	ngi	115	nk-	160
mc-	15	mns	1	mve	5	ndl	49	ngl	133	nla	12
mde	21	mn-	3	mv-	4	ndm	76	ngm	11	nle	7
md-	5	mob	40	mxy	2	ndn	7	ngo	3	nli	34
mea	136	mod	113	mx-	11	ndo	88	ngr	18	nly	258
meb	3	moe	9	mys	6	ndp	2	ngs	148	nl-	10
mec	42	mof	1	my-	82	ndr	17	ngt	54	nma	4
med	195	mog	11	mzo	3	nds	159	ngu	25	nme	19
meg	4	mol	8	m--	2408	ndt	1	ngx	1	nmo	1
mei	7	mom	17	nab	47	ndu	80	ng-	3763	noa	2
mek	1	mon	180	nac	21	ndv	18	nha	11	noc	5
mel	47	mor	368	nad	42	ndw	4	nhe	4	nod	106
mem	70	mos	183	nag	34	ndy	2	nhf	9	noe	28
men	953	mot	48	nai	32	nd-	3209	nhi	5	nof	2
mep	124	mou	155	nal	719	nea	34	nho	6	nog	13
mer	162	mov	1230	nam	145	nec	95	nh-	54	noi	5
mes	431	moz	6	nan	8	ned	406	nia	16	nol	87
met	200	mo-	2	nap	8	nef	22	nib	5	nom	46
meu	4	mpa	114	nar	31	neg	57	nic	143	non	141
mew	36	mpb	2	nas	8	nei	30	nie	32	nop	2
mex	1	mpe	43	nat	369	nej	1	nif	42	nor	141
mey	1	mph	12	nau	8	nel	47	nig	81	nos	14
me-	1360	mpi	73	nav	10	nem	17	nij	1	not	941
mfo	5	mpl	619	naz	6	nen	135	nik	26	nou	38
mfr	4	mpo	149	na-	69	neo	26	nim	30	nov	57
mfu	2	mpr	119	nba	4	neq	1	nin	537	now	685
mge	10	mps	128	nbe	18	ner	424	nio	37	noy	2
mg-	18	mpt	67	nbi	1	nes	248	nip	17	no-	300
mhz	323	mpu	511	nbu	18	net	124	niq	66	npa	2
mh-	8	mp-	7	nb-	93	neu	22	nis	142	npe	1
mia	4	aro	2	nca	7	nev	76	nit	214	npl	1
mic	170	mrs	1	ncd	1	new	211	niu	2	npo	2
mid	58	mr-	18	nce	1410	nex	71	niv	146	npr	10
mie	2	msc	2	nch	154	ney	3	nix	8	npu	18
mig	66	mse	25	nci	72	ne-	935	niy	4	nqu	1
mij	2	msk	2	nck	6	nfa	10	niz	79	nre	9
mik	14	mso	1	ncl	180	nfd	8	ni-	2	nri	6
mil	96	msq	12	nco	71	nfe	129	nja	4	nro	2
min	532	mst	39	ncr	129	nfh	1	nje	10	nsa	7
mir	5	ms-	489	nct	193	nfi	121	njo	4	nsb	2
mis	121	mt-	19	ncu	31	nfl	15	nju	4	nsc	30
mit	146	muc	59	ncy	62	nfo	180	nke	17	nsd	1
mix	1	mue	17	nc-	166	nfr	13	nki	30	nse	210
miz	10	mug	9	nda	129	nfu	3	nkkn	22	nsf	51
mi-	12	mul	296	ndb	8	nf-	176	nko	7	nsh	48
mko	1	mum	35	ndc	2	nga	9	nkp	70	nsi	289
mle	1	mun	89	nde	541	ngb	3	nkq	4	nsk	1
mli	2	mur	4	ndf	14	nge	282	nkr	5	nsl	37
mly	5	mus	89	ndg	394	ngf	43	nks	55	nso	11
mna	3	mut	20	ndh	23	ngn	1	nkt	1	nsp	55

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
nst	731	nxh	1	oda	4	oia	1	one	579	orn	45
nsu	39	nxg	2	ode	185	oic	14	onf	174	oro	9
nsw	35	nxh	1	odf	1	oid	39	ong	191	orp	21
nsy	1	nx-	1	odg	1	oil	7	oni	120	ors	222
ns-	1632	nya	1	odi	58	oin	198	onj	16	ort	476
nta	440	nyb	1	odo	12	ois	12	onk	9	oru	3
ntb	3	nyc	4	ods	17	oit	30	onl	239	orv	1
nre	918	nyi	3	odu	182	oi-	1	onm	16	orw	78
ntf	2	nym	6	ody	7	oje	77	ono	39	ory	152
nth	79	nyo	5	odz	7	oj-	4	onp	1	orz	4
nti	488	nyt	1	od-	198	oka	1	onr	1	or-	2142
ntl	139	nyu	1	oea	1	oke	62	ons	2007	osa	6
ntm	4	nyw	4	oed	5	oki	15	ont	675	osc	23
ntn	23	ny-	284	oef	1	okm	1	onu	37	ose	354
nto	271	nza	6	oeg	12	oko	9	onv	175	osh	10
ntp	38	nze	3	oei	1	oks	31	onw	2	osi	1202
ntr	596	nz-	6	oek	41	oku	2	ony	13	osl	7
nts	479	n--	14091	oel	17	oky	1	onz	2	osm	4
ntu	81	oac	128	oem	24	ok-	137	on-	4469	oso	7
ntw	9	oad	50	oen	38	ola	69	opa	9	osp	14
ntx	3	oal	56	oep	32	old	77	opb	12	ost	300
nty	19	oam	2	oes	140	ole	92	opc	1	osu	3
nt-	1560	oar	84	oet	9	olg	14	ope	288	os-	74
nua	47	oas	2	oe-	23	oli	23	opg	6	ota	112
nuc	2	oat	2	ofd	1	olm	4	oph	14	otd	3
nue	28	oa-	1	ofe	68	olo	373	opi	52	ote	218
nuf	22	oba	22	ofi	25	ols	26	opl	13	oth	367
nui	12	obe	23	ofi	57	olt	1	opm	22	oti	210
nul	11	obi	47	ofu	1	olu	118	opn	1	otl	35
num	408	obj	14	ofr	2	olv	120	opo	157	otn	13
nun	5	obl	218	ofs	36	oly	4	opr	24	oto	40
nuo	1	obo	4	oft	130	ol-	354	ops	28	otp	1
nus	44	obr	2	ofw	1	oma	131	opt	404	ots	12
nut	23	obs	45	of-	4643	omb	66	opu	4	otv	15
nu-	5	obt	69	oga	8	omd	1	opy	34	oty	3
nva	6	obv	35	oge	39	ome	481	op-	232	ot-	713
nve	216	oby	1	ogi	46	omf	5	oqs	9	oub	39
nvi	41	ob-	24	ogl	7	omg	10	oqu	2	ouc	8
nvl	1	oca	57	ogn	67	omi	109	ora	109	oud	15
nvo	60	oce	611	ogo	28	oml	2	orb	8	oug	248
nwa	3	och	11	ogr	446	omn	14	orc	120	oul	453
nwe	2	oci	55	ogs	1	omo	80	ord	386	oun	349
nwh	9	ock	118	ogu	6	omp	999	ore	868	oup	42
nwi	1	oco	23	ogw	1	oms	23	orf	11	our	659
nwy	4	ocp	4	ogy	88	omu	87	org	85	ous	345
nw-	8	ocr	2	og-	25	omy	7	ori	392	out	366
nxb	1	ocs	10	ohe	11	om-	689	orj	8	ouv	1
nxc	3	oct	10	ohi	1	ona	305	ork	157	ouw	22
nxd	7	ocu	106	ohn	12	onc	333	orl	76	ou-	29
nxe	17	oc-	46	ohr	4	ond	447	orm	514	ova	22

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
ove	1725	pb-	9	pkp	3	pto	6	qur	1	rcr	3
ovi	128	pco	6	pkq	12	pts	39	qxa	10	rds	16
ovl	6	pcr	1	pkp	11	ptu	115	qxb	5	rct	11
ovo	2	pcs	31	pk-	16	pty	17	qxc	7	rcu	25
ov-	42	pc-	1	pla	613	pt-	96	qxd	10	rc-	46
owa	17	pda	18	ple	611	pub	113	qxe	8	rda	45
owe	253	pdp	4	pli	300	pul	27	qxf	9	rdb	2
owi	175	pd-	6	plo	58	pun	26	qyg	5	rde	291
owl	454	pea	75	plu	5	pup	1	qxh	4	rdg	4
own	235	pec	333	ply	210	pur	65	qxm	6	rdi	130
owp	2	ped	164	pl-	9	pus	11	q--	149	rdl	14
ows	109	pef	2	pma	7	put	594	rab	56	rdo	1
owt	7	pek	4	pme	26	puz	6	rac	259	rdr	7
ow-	443	pel	60	pmi	3	pu-	3	rad	88	rds	61
oxe	1	pen	296	pm-	1	pvq	3	rae	5	rdt	12
oxf	3	peo	4	pna	1	pvs	1	raf	11	rdw	55
oxi	15	per	1256	pog	5	pwa	2	rag	107	rd-	301
oxo	1	pes	30	poi	186	pyi	1	rai	72	rea	748
ox-	8	pet	53	pol	16	pyr	7	raj	3	reb	24
oya	1	pew	1	pom	9	py-	37	rak	6	rec	457
oyc	52	pe-	104	pon	174	p--	1434	ral	525	red	585
oye	10	pfu	7	pop	50	qam	41	ram	584	ref	224
oyi	1	pf-	9	por	292	qar	1	ran	598	reg	79
oym	1	pga	1	pos	1325	qa-	57	rap	102	reh	8
oys	5	pge	5	pot	137	qbk	4	rar	84	rei	29
oy-	8	pha	124	pov	4	qb-	44	ras	24	rej	2
oze	10	phd	1	pow	61	qc-	37	rat	889	rek	18
o--	5156	phe	40	po-	10	qd-	54	rau	1	rel	222
pab	24	phi	152	pra	58	qe-	72	rav	8	rem	138
pac	111	pho	45	pre	701	qf-	42	raw	165	ren	602
pad	1	phr	5	prg	1	qg-	47	rax	1	reo	46
pag	18	phs	4	pri	342	qh-	39	ray	39	rep	280
pai	50	pht	2	pro	2155	qkb	16	raz	2	req	128
pak	4	phy	5	pru	17	qkn	12	ra-	290	rer	36
pal	16	ph-	78	pr-	7	qkq	4	rba	20	res	1631
pam	1	pia	1	pse	1	qkr	18	rbd	1	ret	129
pan	60	pic	43	psd	6	qk-	16	rbe	18	reu	1
pao	3	pid	4	pse	16	qnk	4	rbi	41	rev	170
pap	83	pie	168	psh	3	qn-	5	rbk	14	rew	6
paq	1	pig	3	pso	129	qpk	12	rbo	89	rey	40
par	770	pil	17	psr	5	qpv	3	rbr	4	rez	1
pas	95	pin	78	psu	1	qp-	10	rby	1	re-	2878
pat	201	pio	67	psv	4	qrk	8	rb-	37	rfa	35
pau	2	pip	7	psw	2	qrt	6	rca	1	rfd	1
paw	312	pir	52	psy	9	qsq	6	rce	217	rfe	112
pay	10	pis	6	ps-	242	qs-	9	rch	590	rfg	1
pa-	11	pit	21	pta	24	qua	309	rci	28	rfl	8
pba	12	pi-	40	pte	58	que	542	rck	30	rfo	183
pbe	2	pj-	6	pth	81	qui	218	rcl	2	rfu	4
pbo	12	pkb	19	pti	463	quo	23	rco	11	rf-	26

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
rga	57	rkx	6	rov	283	ruc	284	r--	7536	ser	345
rgd	2	rkz	1	row	34	rud	1	sab	13	ses	539
rge	117	rk-	159	rox	16	rue	43	sac	11	set	317
rgh	18	rla	86	roy	58	ruh	2	sad	21	seu	10
rgi	2	rld	72	ro-	136	rui	20	saf	41	sev	59
rgo	3	rle	3	rpa	4	ruk	2	sag	8	sew	12
rgr	3	rli	104	rpe	9	rul	664	sai	28	sey	4
rgu	20	rlo	33	rpi	6	rum	9	sak	6	se-	1563
rg-	125	rls	15	rpj	6	run	83	sal	20	sfa	7
rha	35	rly	128	rpk	8	ruo	1	sam	131	sfe	42
rhe	14	rl-	17	rpl	13	rup	13	san	32	sfi	1
rhg	2	rma	337	rpo	51	rus	31	sap	3	sfo	17
rho	55	rme	96	rpr	65	rut	45	sar	98	sfu	17
rhu	3	rmf	2	rps	145	ruy	8	sas	2	sfy	6
rh-	33	rmi	124	rp-	79	ru-	3	sat	28	sge	2
ria	265	rml	5	rsa	17	rva	32	sau	1	sgo	23
rib	174	rmo	33	rsc	73	rve	63	sav	84	sgu	21
ric	145	rms	65	rsd	6	rvi	22	saw	3	sha	118
rid	89	rmt	10	rse	121	rvo	1	say	35	shc	9
rie	220	rmu	28	rsh	19	rv-	18	sa-	12	she	164
rif	14	rmy	6	rsi	291	rwa	54	sba	9	shi	111
rig	78	rm-	134	rsk	6	rwe	10	sbe	6	sho	521
rij	23	ma	315	rsl	48	rwh	2	sbi	2	shp	9
rik	159	me	117	rsm	1	rwi	38	sbo	5	shr	1
ril	36	mg	1	rso	45	rwo	11	sbu	2	shu	2
rim	137	rni	140	rsp	5	rwr	3	sca	44	sh-	105
rin	301	rnk	4	rst	403	rw-	1	sce	26	sia	10
rio	225	rmo	4	rsu	55	rxa	2	scf	17	sib	246
rip	46	rms	54	rsv	2	rxb	5	sch	310	sic	31
ris	207	rmu	1	rsz	2	rxc	2	sci	81	sid	305
rit	434	m-	140	rs-	829	rd	5	sco	209	sie	25
riu	1	roa	140	rta	209	rx	15	scr	137	sif	40
riv	98	rob	249	rtb	1	rx	1	scu	60	sig	217
rix	22	roc	639	rtc	8	rxg	2	sc-	20	sil	23
riz	62	rod	169	rt	150	rxh	4	sda	4	sim	310
ri-	39	roe	41	rtg	2	rxm	6	sdf	2	sin	417
rja	8	rof	164	rth	155	rxp	4	sdi	6	sio	385
rji	4	rog	444	rti	333	rx-	8	sdo	5	sip	1
rka	16	roh	6	rtl	11	rya	1	sd-	1	sir	18
rkb	8	roi	12	rtm	53	ryb	1	sea	506	sis	315
rke	92	roj	77	rt	4	ryi	5	sec	246	sit	1190
rkh	1	rok	109	rto	9	ryo	4	sod	456	siu	9
rki	25	rol	289	rts	124	ryp	7	sef	32	siv	63
rkl	4	rom	708	rtu	82	ryt	2	sei	12	six	16
rkn	8	ron	158	rtw	4	ry-	587	sek	5	siz	52
rkq	4	rop	155	rt	13	rza	7	sel	269	si-	14
rkr	15	ror	28	rtz	2	rze	18	sem	99	ska	7
rks	21	ros	29	rt-	452	rzi	3	sen	580	skd	6
rkt	9	rot	130	rua	8	rzo	37	sep	42	ske	83
rku	3	rou	223	rub	3	rzv	4	seq	169	ski	40

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
skr	16	sru	4	szl	2	tep	36	tju	4	tr-	14
sks	32	sr-	2	sz-	3	ter	2126	tki	8	tsb	2
sku	12	sta	770	s--	18020	tes	419	tko	49	tsc	4
sky	14	stb	9	tab	702	teu	7	tla	6	tse	45
sk-	69	std	3	tac	183	tev	10	tle	79	tsf	8
sla	124	ste	1124	tad	15	tex	77	tli	41	tsg	2
sle	14	stf	2	tae	2	te-	1730	tlm	5	tsh	13
sli	50	stg	3	taf	5	tfa	2	tly	227	tsi	19
slo	39	sth	9	tag	226	tfd	3	tl-	6	tsl	14
slu	1	sti	760	tai	322	tfo	10	tma	31	tst	10
sly	67	stl	43	tak	119	tfu	16	tme	39	tsy	42
sl-	1	stm	14	tal	170	tga	2	tmi	1	ts-	1548
sma	64	sto	341	tam	2	tge	9	tmo	14	tua	113
sme	9	stp	5	tan	394	tgi	5	tms	1	tub	1
smo	5	str	737	tao	1	tgr	1	tmu	1	tud	174
sms	20	sts	84	tap	39	tha	1340	tm-	53	tue	7
sm-	63	stu	148	tar	185	thb	1	tna	14	tug	1
sna	2	stv	4	tas	73	thd	1	tne	8	tui	68
sne	13	stw	3	tat	374	the	10352	tno	17	tul	5
sno	2	sty	8	tau	3	thi	935	tnu	8	tun	41
snt	1	st-	1284	tav	1	thl	2	tny	4	tur	667
sn-	3	sua	29	taw	1	thm	111	tob	11	tus	24
soc	61	sub	281	tay	4	tho	585	toc	8	tut	80
sof	57	suc	267	ta-	216	thr	261	tod	12	tu-	3
sol	206	sud	1	tbe	6	ths	33	toe	56	tvi	15
som	305	sue	69	tbl	6	thu	70	tof	4	tvo	11
son	331	suf	63	tbo	7	thw	17	tog	16	twa	56
sop	17	sug	43	tbr	1	thy	5	toi	5	twe	149
sor	146	sui	15	tbu	8	th-	1265	tok	19	twi	13
sos	2	sul	414	tca	2	tia	197	tol	28	two	251
sou	28	sum	172	tch	103	tib	10	tom	52	tw-	10
so-	351	sun	6	tco	29	tic	601	ton	83	txt	3
spa	80	sup	271	tc-	40	tid	4	top	58	tya	1
spc	6	sur	105	tda	2	tie	225	toq	9	tye	4
spe	397	sus	46	tde	3	tif	89	tor	437	tyl	8
sph	3	su-	2	tdi	2	tig	58	tos	9	tym	1
spi	53	sve	5	tdo	2	tij	7	tot	102	typ	85
spl	31	svr	4	td-	7	tik	14	tou	76	ty-	710
spo	112	sv-	7	tea	73	til	134	tow	16	tze	2
spr	40	swe	75	teb	6	tim	913	to-	3626	tzi	2
spu	3	swi	12	tec	277	tin	901	tpl	2	tzk	20
sp-	9	swo	1	ted	1172	tio	4028	tpo	43	tzs	4
sqa	1	syc	9	tef	14	tip	29	tpr	1	tz-	44
sqr	6	syl	1	teg	154	tiq	2	tpu	7	t--	9524
squ	161	sym	55	tei	42	tir	13	tra	667	uab	20
sq-	24	syn	50	tek	1	tis	71	tre	172	uad	1
sra	2	sys	384	tel	351	tit	172	tri	353	uag	3
srd	1	syt	2	tem	466	tiv	497	tro	522	uai	2
sre	17	sy-	19	ten	483	ti-	26	tru	339	ual	222
sro	5	sza	2	teo	5	tje	1	try	92	uan	22

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
uar	186	ufm	3	umu	11	urm	10	uyt	2	vm-	9
uat	267	uga	1	umv	5	urn	285	uze	2	vna	2
uba	1	uge	53	um-	81	uro	15	uzi	7	vni	4
ubd	13	ugh	247	una	61	urp	43	uzl	6	voc	12
ube	2	ugl	9	unb	1	urs	134	uzy	13	voe	4
ubg	10	ugm	6	unc	205	urt	94	uz-	10	vog	9
ubi	1	ugo	6	und	373	urv	14	u--	146	voi	39
ubj	41	ugs	1	une	33	urx	5	vab	17	vok	17
ubl	157	ugu	2	unf	29	ury	5	vad	3	vol	180
ubm	16	ug-	6	ung	14	ur-	339	vag	17	von	33
ubp	8	uhe	2	unh	3	usa	18	vai	74	vor	100
ubr	4	uic	17	uni	355	usb	1	vak	3	vot	3
ubs	95	uid	24	unj	2	usc	22	val	409	vou	4
ubt	35	uie	36	unk	22	use	500	van	634	vo-	31
ubu	1	uij	1	unl	30	ush	3	var	195	vq-	3
ub-	63	uik	9	unm	3	usi	178	vas	11	vra	8
uce	161	uil	53	uno	4	usk	1	vat	78	vri	5
uch	277	uin	16	unp	5	usl	61	vax	7	vr-	3
uci	46	uip	6	unr	8	usm	2	va-	6	vsk	7
uck	12	uir	104	uns	24	usn	5	vce	1	vsp	1
ucl	2	uis	25	unt	229	uso	1	vdh	7	vs-	7
ucp	7	uit	155	unu	6	usp	7	vea	14	vul	1
uct	419	uiv	23	unw	2	ust	213	ved	236	vu-	2
ucu	1	uiz	3	un-	60	usu	28	veg	6	vy-	36
uc-	1	ui-	32	uon	1	usy	5	veh	1	v--	192
uda	4	uja	1	uot	23	us-	480	vei	10	wac	3
ude	168	uk-	3	uou	20	uta	31	vel	274	wad	1
udg	14	ula	322	upa	9	utc	48	vem	68	wag	6
udi	67	ulb	1	upd	18	utd	3	ven	457	wai	71
udl	1	ulc	5	upe	173	ute	680	ver	1187	wak	1
udo	17	uld	451	upi	13	utg	1	ves	630	wal	9
udr	1	ule	751	upl	52	uth	94	vet	67	wan	8
uds	1	ulf	6	upo	68	uti	279	vey	7	war	261
udw	1	uli	30	upr	3	utl	13	ve-	1796	was	279
udy	54	ulk	2	ups	13	utm	1	via	42	wat	10
ud-	25	ulm	1	upt	28	uto	47	vic	93	way	149
ueb	1	uln	3	upw	2	utp	5	vid	201	wa-	6
ued	19	ulo	3	upy	7	utr	2	vie	89	wba	10
uei	1	ulp	6	up-	240	uts	18	vig	13	wbc	1
uel	27	uls	6	uqu	1	utu	32	vil	15	wbo	28
uen	342	ult	585	ura	174	uty	4	vin	193	wb-	11
uer	5	uly	61	urb	89	ut-	656	vio	128	wco	3
ues	245	ul-	216	urc	20	uve	1	vir	18	wc-	1
ueu	4	uma	203	urd	4	uwe	12	vis	108	wea	61
uez	3	umb	373	ure	887	uwi	1	vit	59	wed	116
ue-	333	ume	130	urf	1	uwm	1	viz	30	weg	19
ufa	22	umi	35	urg	30	uw-	17	vi-	13	wei	51
ufe	11	umn	4	uri	129	ux-	1	vlo	1	wek	5
ufi	58	ump	20	urk	10	uyi	1	vls	7	wel	142
ufi	5	ums	5	url	6	uyn	6	vl-	6	wen	135

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
wep	14	wp-	57	xms	12	ygo	2	yzi	10	zwe	5
wer	363	wq-	1	xon	1	yho	3	y--	6179	zwo	10
wes	37	wri	62	xpa	3	yie	27	zab	2	zy-	13
wet	16	wro	11	xpe	425	yin	131	zag	2	z--	431
wev	91	wr-	8	xpi	1	yi-	4	zak	13	-ab	355
wey	1	wsl	8	xpl	99	yke	1	zal	4	-ac	487
wez	1	ws-	151	xpo	4	yle	7	zam	3	-ad	546
we-	555	wth	8	xpr	45	yli	1	zar	14	-ae	4
wfa	1	wtn	31	xp-	6	ylv	1	zas	1	-af	202
wfr	4	wto	7	xqs	6	yma	48	zat	103	-ag	235
wha	103	wvo	2	xr-	3	ymb	14	za-	10	-ah	6
whe	365	wvp	2	xs-	7	ymc	6	zed	141	-ai	78
whi	999	wy-	4	xta	1	yme	29	zeg	2	-aj	12
who	60	w--	830	xtb	4	ymi	1	zei	3	-ak	11
why	19	xac	29	xte	62	ymn	3	zek	1	-al	1065
wic	8	xag	3	xth	4	ymo	6	zel	29	-am	182
wid	58	xam	205	xti	1	ymp	12	zen	5	-an	3557
wie	12	xas	6	xtr	33	yna	14	zer	72	-ap	494
wif	2	xa-	15	xts	1	ync	16	zes	9	-aq	1
wig	3	xbi	6	xtu	3	yno	5	zet	17	-ar	1187
wij	26	xbq	9	xty	1	ynt	33	zev	9	-as	1146
wik	6	xb-	22	xt-	135	yn-	10	ze-	118	-at	549
wil	265	xce	150	xub	1	yon	59	zic	1	-au	126
win	414	xch	127	xwp	18	yor	26	zie	10	-av	189
wip	6	xci	2	xy-	2	you	46	zig	2	-aw	30
wis	43	xcl	20	x--	1108	yo-	1	zij	50	-ax	18
wit	868	xco	8	yab	8	ypa	1	zil	1	-ay	2
wi-	1	xcr	6	yai	1	ype	64	zin	52	-a-	3573
wka	11	xc-	37	yan	1	ypi	16	zio	6	-ba	528
wkc	16	xd-	47	yar	2	ypo	97	zi-	1	-bc	44
wkn	12	xec	44	yat	10	ypr	1	zku	12	-bd	28
wko	6	xed	22	ya-	13	ypt	7	zky	20	-be	2535
wkp	3	xem	6	yba	6	yri	7	zle	5	-bf	11
wk-	77	xer	4	ybe	23	yru	3	zli	1	-bg	23
wle	447	xe-	57	ybi	6	yse	45	zl-	2	-bh	10
wly	15	xfo	3	ybo	2	ysi	95	zma	4	-bi	233
wma	1	xf-	69	ybr	2	yst	392	zoa	2	-bk	126
wmc	11	xg-	33	ych	13	ys-	93	zob	2	-bl	525
wne	5	xha	12	yco	8	yte	98	zod	1	-bm	1
wnl	3	xhi	10	ycr	52	yth	3	zoe	34	-bn	4
wns	31	xh-	23	yde	4	yti	2	zon	23	-bo	310
wn-	602	xib	16	ydr	184	ytu	12	zor	2	-bp	4
woa	1	xic	1	yea	66	yug	6	zov	3	-bq	1
wod	17	xim	155	yed	82	yul	1	zow	1	-br	358
won	155	xin	2	yel	2	yus	2	zo-	2	-bt	13
wor	286	xio	6	yer	48	yu-	4	zso	4	-bu	354
wou	185	xis	47	yes	34	ywa	3	zt-	2	-bv	5
wo-	257	xit	52	yet	61	ywh	1	zui	4	-bx	76
wpa	8	xi-	3	ye-	4	ywo	3	zul	4	-by	1160
wpo	5	xl-	1	yfu	3	yze	6	zvo	4	-b-	573

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
-ca	1072	-er	53	-if	308	-lm	7	-op	729	-rn	3
-cb	3	-es	196	-ig	16	-lo	395	-or	770	-ro	362
-cd	2	-et	73	-ij	6	-lp	8	-os	14	-rp	231
-ce	208	-eu	16	-ik	1	-lt	13	-ot	151	-rs	3
-cf	8	-ev	444	-il	38	-lu	16	-ou	358	-rt	3
-ch	1211	-ex	1343	-im	436	-l-	185	-ov	154	-ru	743
-ci	40	-e-	842	-in	5057	-ma	1944	-ow	60	-rx	39
-cl	248	-fa	465	-io	1	-mb	4	-ox	5	-r-	209
-cm	2	-fd	7	-ir	60	-mc	30	-o-	81	-sa	345
-cn	3	-fe	210	-is	2206	-md	1	-pa	1236	-sb	4
-co	3562	-fi	930	-it	1088	-me	754	-pb	9	-sc	375
-cp	25	-fj	1	-iv	8	-mf	4	-pc	32	-sd	2
-cr	143	-fl	67	-i-	577	-mg	18	-pd	4	-se	1595
-cu	175	-fn	2	-ja	29	-mh	325	-pe	550	-sh	551
-cx	19	-fo	2022	-je	9	-mi	411	-pf	9	-si	702
-cy	16	-fr	765	-ji	1	-mo	2004	-ph	164	-sk	112
-c-	897	-fu	431	-jo	210	-mp	5	-pi	208	-sl	106
-da	571	-fx	6	-jr	2	-mr	19	-pk	21	-sm	55
-db	78	-f-	606	-ju	85	-ms	18	-pl	741	-sn	8
-dc	2	-ga	461	-j-	359	-mu	273	-pm	1	-so	770
-de	2185	-ge	432	-ka	256	-mx	4	-po	1463	-sp	405
-dg	19	-gh	7	-kb	371	-my	62	-pr	2644	-sq	164
-di	921	-gi	183	-kc	310	-m-	255	-ps	37	-sr	1
-dl	2	-gl	94	-kd	289	-na	186	-pu	252	-st	1162
-dm	8	-gm	1	-ke	508	-nb	93	-pv	1	-su	1027
-do	429	-go	156	-kf	398	-nc	154	-p-	452	-sw	50
-dp	11	-gr	399	-kg	337	-nd	199	-qa	98	-sy	437
-dq	9	-gs	9	-kh	360	-ne	822	-qb	44	-sz	5
-dr	323	-gt	2	-ki	195	-nf	181	-qc	37	-s-	1053
-ds	27	-gu	72	-kl	20	-ng	178	-qd	54	-ta	562
-du	195	-gx	12	-km	1	-nh	63	-qe	72	-tb	6
-dv	4	-gy	4	-kn	649	-ni	34	-qf	42	-te	592
-dw	8	-g-	574	-ko	109	-nl	9	-qg	47	-th	12558
-dx	9	-ha	1254	-kp	24	-no	1564	-qh	39	-ti	412
-dy	15	-hd	6	-kq	114	-nr	1	-qp	3	-to	3490
-d-	795	-he	733	-kr	70	-nu	383	-qu	428	-tr	524
-ea	272	-hf	8	-kt	1	-nw	8	-qx	64	-tu	195
-eb	9	-hg	1	-ku	9	-nx	38	-q-	52	-tv	4
-ec	21	-hi	359	-kv	1	-n-	279	-ra	568	-tw	272
-ed	216	-hl	11	-kx	16	-ob	176	-rb	38	-ty	76
-ef	147	-ho	433	-ky	9	-oc	116	-rc	27	-t-	124
-eg	1	-hu	180	-k-	229	-od	25	-rd	20	-uc	9
-eh	1	-hw	1	-la	289	-oe	1	-re	2409	-ui	32
-ei	76	-hx	5	-lb	99	-of	4637	-rf	21	-uk	1
-ek	4	-hy	294	-lc	3	-og	1	-rg	19	-ul	29
-el	189	-h-	694	-le	708	-oh	4	-rh	17	-un	574
-em	76	-ib	11	-lf	3	-ok	22	-ri	95	-up	212
-en	1197	-ic	145	-li	345	-ol	19	-rk	9	-ur	17
-ep	5	-id	184	-lj	6	-om	37	-rl	30	-us	462
-eq	108	-ie	4	-lk	3	-on	1411	-rm	16	-ut	43

FREQUENCIES FOR ICCA-JOURNAL TRIGRAMS			
tri	freq	tri	freq
-u-	66	-ze	88
-va	796	-zi	58
-vd	7	-zo	8
-ve	311	-zt	2
-vi	187	-zu	8
-vl	7	-zw	15
-vn	4	-z-	5
-vo	224	--a	14055
-vr	4	--b	6922
-vs	7	--c	7634
-vu	3	--d	5611
-v-	34	--e	5289
-wa	511	--f	5512
-wb	12	--g	2406
-wc	1	--h	3979
-we	961	--i	10146
-wf	4	--j	695
-wh	1491	--k	4276
-wi	1436	--l	2080
-wk	125	--m	6131
-wm	11	--n	4192
-wn	12	--o	8771
-wo	605	--p	7824
-wp	39	--q	980
-wq	1	--r	4862
-wr	64	--s	8929
-wt	38	--t	18815
-wv	4	--u	1445
-w-	63	--v	1584
-xa	2	--w	5374
-xb	6	--x	870
-xc	8	--y	270
-xe	2	--z	194
-xf	15		0
-xh	6		0
-xi	3		0
-xl	1		0
-xp	2		0
-xr	3		0
-xs	7		0
-xt	6		0
-x-	809		0
-yb	6		0
-yc	8		0
-ye	141		0
-yi	27		0
-yo	72		0
-yu	11		0
-y-	5		0
-za	10		0

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
aba	3	age	367	amd	1	arl	17	awn	5	bew	3
abc	4	agi	18	ame	1566	arm	11	aws	6	bex	2
abe	35	agl	2	amh	8	arn	18	aw-	10	bey	4
abi	10	agn	38	ami	61	aro	17	axb	3	be-	921
abl	483	agr	27	aml	2	arp	8	axh	1	bfa	1
abm	1	ags	74	amo	34	ars	37	axi	41	bfh	1
abn	6	agt	1	amp	111	art	179	axn	4	bfi	6
abo	81	ag-	33	ams	55	ary	129	axp	2	bfr	1
abr	11	aha	1	amw	1	ar-	313	axr	2	bft	1
abs	59	ahe	1	am-	144	asc	43	axu	1	bg-	4
aby	1	ah-	4	ana	18	ase	281	ax-	202	bib	17
ab-	27	aid	2	anc	41	ash	63	aye	9	bic	1
acd	2	aig	1	and	2335	asi	59	ayl	2	bid	7
ace	582	ail	262	ane	37	ask	165	ayo	1	bif	12
ach	288	aim	10	ang	279	asm	7	ays	65	big	12
aci	44	ain	410	ani	57	aso	68	ay-	427	bih	1
ack	244	air	36	ank	54	ast	118	a--	343	bil	25
acm	1	ais	5	anl	1	asu	21	bab	2	bin	83
aco	44	ait	63	anm	4	asv	1	bac	98	bir	2
acq	1	aiv	1	ano	81	asw	23	bad	42	bit	176
acr	56	ai-	2	anp	1	asx	1	bal	43	biw	1
act	578	ajf	2	anr	1	asy	24	ban	13	bi-	1
acu	4	ajo	3	ans	146	as-	791	bar	7	bje	84
acz	1	akd	1	ant	96	ata	189	bas	71	bjf	23
ac-	11	ake	94	anu	35	atc	143	bat	1	bla	48
ada	6	aki	19	any	245	ate	1093	bau	13	ble	631
adb	30	akp	18	an-	989	atf	26	bav	1	bli	25
adc	8	aks	6	aof	1	ath	381	ba-	1	blk	10
ade	134	ak-	21	aon	3	ati	843	bb-	1	blo	166
adf	37	ala	13	aos	3	atl	1	bca	1	bly	22
adi	113	alb	1	aou	29	atm	3	bcd	11	bl-	14
adj	16	alc	15	apa	17	ato	79	bcf	1	bmi	1
adl	18	ale	130	ape	146	atq	4	bci	2	bms	2
adm	17	alf	39	aph	36	atr	16	bct	1	bm-	8
adn	5	alg	9	api	27	ats	24	bc-	16	bno	5
ado	5	alh	13	apl	66	atu	91	bde	1	bn-	2
adr	361	ali	214	apm	2	at-	849	bdi	9	boa	8
ads	41	alk	1	apo	10	aud	25	bea	1	bok	20
adv	18	alm	3	apr	43	aug	17	bec	78	bol	109
ady	48	alo	152	aps	37	aul	247	bed	32	bor	14
ad-	297	alp	11	apt	9	aur	2	bef	85	bos	13
ael	1	alr	45	ap-	65	aus	214	beg	82	bot	70
aem	13	als	406	ara	687	aut	68	beh	11	bou	46
ae-	2	alt	45	arb	13	ava	65	bei	79	bov	35
afe	26	alu	485	arc	158	ave	149	bel	77	box	10
afi	12	alw	35	ard	249	avi	25	ben	67	bpr	16
afn	3	aly	244	are	1030	avm	1	bep	1	bpw	2
aft	91	al-	1670	arg	360	avo	9	ber	372	bp-	29
af-	1	ama	3	ari	320	awa	8	bes	9	bra	60
aga	24	amb	7	ark	51	awk	6	bet	50	bre	56

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
bri	8	cd-	20	ckn	25	crt	20	dat	246	dit	138
brk	19	cea	2	cko	25	cry	8	dav	3	div	21
bro	27	ced	348	cks	115	cr-	10	day	32	dix	14
br-	7	cef	5	ckt	5	csa	1	dba	1	di-	8
bsc	8	ceh	5	cku	2	csd	1	dbi	6	dja	3
bsd	6	cei	52	ckw	13	csk	53	dbl	10	djt	8
bse	33	cel	12	cky	14	csp	1	dbu	2	dju	5
bsh	4	cem	21	ck-	389	csw	5	dbx	14	dk-	6
bsi	1	cen	39	cla	46	cs-	82	db-	27	dle	53
bsc	12	cep	108	cle	45	cta	33	dca	12	dli	23
bst	105	cer	21	cli	5	ctd	2	dci	1	dlk	2
bsy	6	ces	1067	clo	91	cte	488	dcm	1	dlo	10
bs-	98	ce-	640	clr	2	cti	566	dco	2	dls	2
bta	15	cfi	2	clu	168	ctl	46	dc-	16	dly	2
bte	3	cf-	1	clw	1	cto	392	dea	30	dma	2
btl	1	cgp	1	cl-	4	ctr	2	deb	33	dmb	1
btr	20	cha	686	cma	5	ct-	43	dec	88	dmd	1
buf	183	chd	13	cmb	1	ctu	108	ded	259	dmh	1
bug	36	che	244	cmc	1	ct-	293	def	375	dmi	2
bui	25	chf	5	cmd	19	cub	1	del	143	dmp	1
bur	1	chg	7	cmh	1	cul	38	dem	7	dms	5
bus	16	chi	194	cmp	20	cum	32	den	184	dmw	1
but	106	chl	8	cmr	1	cup	4	dep	35	dm-	16
bu-	1	chm	48	cmt	1	cur	526	der	291	dnc	1
bvi	1	chn	3	cmw	1	cus	5	des	583	dne	6
bv-	1	cho	63	cm-	3	cut	279	det	58	dnh	1
bx-	15	chp	2	cne	3	cu-	1	dev	117	dni	2
bye	1	chr	32	cnt	33	cvb	2	dex	17	dnl	1
byp	3	chs	6	cob	1	cve	41	de-	463	dnp	1
byt	140	ch-	746	cod	86	cvf	5	dfd	6	dnt	1
by-	666	cia	258	cog	22	cvm	5	dfi	3	dn-	9
b--	439	cib	1	col	172	cv-	18	dfq	1	doc	20
cab	3	cid	7	com	1136	cwc	2	dfw	1	doe	157
cac	4	cie	26	con	963	cwd	3	df-	38	doi	3
cad	8	cif	746	cop	77	cw-	1	dge	1	dol	5
cal	585	cil	14	cor	168	cyc	7	dgr	8	dom	71
cam	4	cim	43	cos	7	cyf	1	dhm	1	don	41
can	270	cin	33	cou	155	cyl	2	dhu	1	dot	32
cap	48	cip	4	cov	5	cy-	8	dh-	4	dou	69
car	37	cir	11	cpc	1	cz-	1	dia	85	dow	39
cas	94	cis	21	cpi	2	c--	696	dic	203	doz	1
cat	435	cit	15	cpt	1	dab	3	did	10	do-	58
cau	192	ci-	40	cpu	29	dac	1	die	1	dpa	3
ca-	1	cka	18	cp-	41	dae	14	dif	198	dpo	4
cbs	5	ckd	1	cqu	1	dai	1	dig	24	dp-	6
cbt	3	cke	338	cra	10	dam	4	dim	1	dqb	3
cb-	12	ckf	6	cre	233	dan	16	din	245	dqu	10
cde	2	ckg	14	cri	428	dap	11	dio	1	dra	15
cdi	10	cki	36	crm	1	dar	171	dir	530	dre	312
cdp	3	ckl	10	cro	62	das	3	dis	173	drf	2

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
dri	14	eam	23	efh	3	elt	13	epr	39	etd	40
drl	3	ean	49	efi	237	elv	6	eps	2	ete	368
drm	1	eap	1	efl	13	ely	89	ept	115	etf	10
drn	3	ear	220	efo	90	el-	377	ep-	67	etg	24
dro	2	eas	59	efs	12	ema	71	eqn	20	eth	83
drs	3	eat	245	eft	62	emb	59	eqp	4	eti	170
dr-	59	eau	1	efu	49	eme	156	equ	245	etj	3
dsa	1	eav	9	ef-	29	emf	8	eq-	10	etl	7
dsc	5	ea-	21	ega	61	emi	9	era	347	etm	8
dse	31	eba	30	ege	53	eml	1	erb	16	eto	32
dst	4	ebc	10	egi	222	emo	262	erc	15	etp	65
dsw	6	ebo	17	egm	35	emp	211	erd	1	etq	8
ds-	552	ebu	45	ego	2	ems	39	ere	494	etr	55
dta	11	eby	3	egr	26	emt	2	erf	58	ets	177
dtc	1	eb-	5	egs	1	emu	1	erg	2	etu	465
dte	8	eca	54	egu	74	em-	414	eri	119	etw	62
dth	9	ece	149	egv	5	ena	147	erk	1	etx	6
dto	5	ech	62	eg-	50	enc	257	erl	33	ety	4
dua	3	eci	1016	eha	13	end	382	erm	476	et-	786
duc	81	eck	76	ehe	1	ene	85	ern	222	eud	1
due	11	ecl	41	ehi	2	enf	7	ero	534	eue	20
dul	13	ecn	1	eho	2	eng	71	erp	69	eui	14
dum	10	eco	186	eh-	25	eni	51	ers	667	eup	3
dup	38	ecp	1	eig	5	enj	1	ert	102	eur	1
dur	76	ecr	14	eil	1	enl	2	eru	70	eus	7
du-	15	ecs	2	ein	148	enm	2	erv	87	eva	34
dva	3	ect	915	eio	34	eno	124	erw	90	eve	249
dvf	2	ecu	322	eip	2	enq	1	ery	44	evi	151
dvi	9	ecv	67	eir	51	ens	37	er-	2582	evn	1
dvm	3	ecw	1	eis	18	ent	1746	esa	177	evo	2
dvr	3	ec-	101	eit	82	enu	24	esc	416	ev-	42
dv-	8	eda	1	eiv	47	env	68	ese	401	ewa	2
dwa	12	edb	1	ei-	5	enw	3	esf	51	ewd	19
dwi	6	edc	1	eje	2	enx	6	esg	4	ewe	4
dwr	3	ede	148	eka	3	en-	1104	esh	20	ewh	3
dw-	2	edg	1	eke	1	eof	25	esi	285	ewi	6
dx-	1	edh	1	eki	7	eom	4	esk	3	ewl	52
dx-	7	edi	129	ekn	1	eon	2	esl	1	ewo	43
dya	4	edl	2	eks	2	eop	6	esn	2	ewr	3
dyn	7	edo	6	ek-	28	eor	2	eso	102	ews	44
dy-	56	edq	10	ela	34	eot	2	esp	67	ew-	141
d--	8457	edr	9	elb	1	eou	23	esr	9	exa	145
eab	3	eds	39	elc	1	eo-	2	est	324	exc	166
eac	246	edt	8	eld	72	epa	82	esu	93	exd	3
ead	425	edu	58	ele	173	epe	107	esw	1	exe	343
eae	1	ed-	4421	elf	19	epf	1	esy	12	exh	19
eaf	5	efa	232	eli	101	epi	12	es-	4057	exi	224
eag	3	efb	2	elo	88	epl	60	eta	50	exm	1
eak	47	efd	5	elp	12	epo	42	etb	1	exo	1
eal	67	efe	270	els	52	epq	1	etc	58	exp	334

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
exr	2	fha	1	fsz	2	get	271	go-	7	hai	3
exs	3	fhn	1	fs-	91	gex	5	gpa	8	hal	48
ext	283	fh-	5	fta	1	ge-	356	gpg	1	ham	2
exu	3	fia	1	fte	103	gfi	5	gpi	3	han	433
ex-	52	fic	99	fth	1	gfo	1	gpl	2	hao	3
eyb	4	fie	838	fti	2	gfp	2	gpr	15	hap	7
eye	1	fif	5	ftm	2	gfr	3	gp-	5	har	521
eyl	2	fig	15	fto	1	gfu	1	gqu	1	has	143
eyo	3	fil	1958	ftp	44	gha	4	gra	239	hat	449
eys	7	fin	201	ftt	4	ghb	1	grc	2	hau	18
eyw	13	fir	146	fts	6	ghd	5	gre	90	hav	106
ey-	89	fit	1	ftw	10	ghe	6	gri	2	ha-	2
eze	3	fiv	2	fty	1	ghl	1	gro	236	hbbo	1
ezo	5	fix	105	ft-	127	gho	2	grp	36	hda	2
e--	19231	fla	84	ful	131	ght	78	gr-	1	hdi	13
fac	34	fle	8	fun	119	ghu	2	gse	13	hdo	1
fai	97	fli	5	fur	24	gh-	92	gsf	1	hdr	5
fal	10	fln	1	fus	8	gic	23	gsi	4	hd-	5
fam	17	flo	55	fut	11	gid	47	gsn	2	hea	25
fap	1	flt	6	fva	7	gig	1	gst	14	hec	76
far	11	flu	3	fvc	1	gil	4	gsy	1	hed	90
fas	8	fl-	19	fwo	21	gin	243	gs-	153	hei	52
fat	1	fma	2	fw-	8	gio	10	gta	1	hel	259
fau	245	fmt	4	fyi	25	gis	124	gte	1	hem	43
fav	2	fnd	3	fy-	55	git	27	gth	66	hen	485
fa-	2	fno	4	f--	3649	giv	138	gti	3	her	630
fbi	4	fn-	5	gab	1	gki	6	gto	3	hes	221
fbl	1	fol	204	gag	1	gld	1	gtr	4	het	29
fbr	2	fou	20	gai	18	gle	96	gts	2	heu	1
fch	9	for	1306	gal	18	gli	1	gt-	3	hew	1
fcn	26	fos	1	gam	2	glo	49	gua	38	hex	22
fc-	24	fot	1	gan	3	gma	18	gui	11	hey	67
fda	12	fou	40	gar	12	gme	35	gul	75	he-	7165
fde	25	fo-	14	gas	1	gmo	8	gum	250	hfn	5
fdi	1	fpe	2	gat	45	gmt	2	gun	1	hgr	7
fdo	1	fp-	1	gav	2	gna	266	guo	8	hib	5
fds	34	fq-	1	gbl	11	gne	49	gup	12	hic	324
fd-	81	fra	24	gbo	4	gni	30	gur	14	hid	3
fea	4	fre	42	gbu	2	gnm	17	gus	9	hie	6
feb	5	fri	2	gch	3	gno	107	gve	34	hif	35
fec	118	fro	373	gco	22	gns	8	gvt	2	hig	41
fed	11	fr-	2	gcr	1	gnu	5	gv-	25	hil	151
fel	1	fsc	1	gc-	10	gn-	28	gwi	1	him	1
fen	2	fse	28	gde	3	goa	1	gxc	2	hin	150
fer	404	fsh	2	gec	4	god	3	gxf	2	hio	1
fet	1	fsi	14	ged	100	goi	3	gy-	1	hip	3
few	7	fso	1	gem	2	gor	11	g--	2336	hir	9
fe-	4	fsp	1	gen	67	got	3	hab	15	his	478
fgr	10	fst	17	ger	84	gou	27	had	16	hit	7
fg-	4	fsy	7	ges	176	gov	2	hae	1	hiv	33

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
hi-	2	hu-	1	iel	71	ile	2059	ipi	2	itu	95
hld	2	hyp	13	ien	36	ili	72	ipl	45	itw	8
hle	2	hys	6	ier	61	ilo	12	ipr	1	ity	99
hli	11	hy-	2	ies	422	ilp	10	ips	2	it-	844
hly	1	h--	2050	iet	3	ils	45	ipt	387	ium	1
hma	2	iab	214	iev	7	ilt	29	ipu	20	iu-	3
hme	25	iac	1	iew	5	ilu	10	ip-	13	iva	30
hmo	46	iag	43	iex	2	ily	27	iqu	29	ivc	2
hm-	10	ial	312	ie-	38	il-	453	iq-	7	ive	509
hna	160	ian	3	ifb	1	ima	86	ira	4	ivi	58
hnh	1	iar	1	ifc	1	ime	417	irc	11	iv-	4
hni	2	ias	58	ifd	26	imi	231	ird	9	iwo	1
hnl	1	iat	127	ife	76	imp	92	ire	553	iwr	2
hn-	1	ia-	14	ifh	1	ims	8	irg	1	iw-	1
hod	3	iba	2	ifi	854	imu	54	irh	1	ixd	1
hoe	7	ibb	1	ifl	1	im-	20	iri	2	ixe	30
hoi	4	ibc	2	ifm	1	ina	365	irm	1	ixi	1
hok	3	ibe	33	ifn	1	inb	3	iro	53	ixl	2
hol	18	ibi	22	ifo	4	inc	240	irs	162	ixr	2
hom	36	ibl	86	ifr	1	ind	271	irt	24	ixt	1
hon	8	ibm	4	ifs	4	ine	942	ir-	174	ixw	1
hop	2	ibp	1	ift	37	inf	98	isa	36	ix-	125
hor	27	ibr	20	ify	79	ing	2124	isc	32	iza	20
hos	131	ibs	4	if-	882	inh	14	isd	6	ize	251
hot	6	ibu	9	iga	3	ini	241	ise	112	izi	1
hou	172	ib-	61	igb	13	ink	138	isf	1	izo	2
how	103	ica	349	igc	27	inl	4	isg	3	iz-	9
ho-	64	ice	124	ige	9	inm	3	ish	39	i--	305
hpa	2	ich	324	igf	7	ino	34	isi	136	jac	4
hra	18	ici	36	igh	114	inp	234	isj	1	jan	21
hrc	11	ick	30	igi	56	inq	2	isk	51	jec	86
hre	18	icl	5	igk	6	ins	313	ism	29	jfi	23
hri	1	ico	8	igm	9	int	1277	isn	3	jfl	2
hro	67	icr	5	ign	451	inu	78	iso	21	jmp	3
hr-	1	ics	67	igp	15	inv	153	isp	35	job	106
hsh	6	ict	60	igq	1	in-	1558	isr	3	joe	3
hst	2	icu	20	igr	8	ioc	23	ist	632	joi	6
hs-	10	ic-	225	igs	28	iod	15	isu	23	jor	3
hte	2	ida	7	igt	13	iog	5	isv	3	jou	1
htl	5	ide	327	igu	25	ioh	3	is-	3076	joy	1
htm	2	idi	10	igv	36	iol	7	ita	25	jp-	4
ht-	9	idl	8	igw	1	ion	2195	itc	34	jti	8
huc	62	idn	2	igx	4	ior	39	ite	322	jud	4
huf	1	idr	2	ig-	36	iot	2	itg	2	jul	2
hug	2	ids	26	ih-	1	iou	71	ith	794	jum	1
huh	3	idt	9	ike	32	iov	56	iti	343	jun	14
hun	1	idu	3	iki	2	io-	67	itl	11	jus	19
hup	9	id-	430	ila	93	ipa	1	ito	47	jwc	1
hus	34	iec	3	ilb	1	ipc	4	itr	12	j--	2
hut	12	ied	562	ild	116	ipe	68	its	257	kab	3

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
kad	10	lad	2	le-	2273	lou	3	lyz	4	mfi	8
kag	7	lag	83	lfb	1	low	375	ly-	831	mfl	6
kar	1	lai	24	lfi	5	loy	1	l--	2982	mge	5
kbi	1	lam	2	lfo	1	lo-	6	mac	103	mh-	20
kby	1	lan	89	lfw	21	lpg	10	mad	67	mic	24
kde	1	lap	5	lf-	33	lph	10	mag	36	mid	7
kdi	8	lar	196	lgo	9	lpi	2	mai	202	mig	5
kdo	1	las	133	lg-	5	lpo	2	maj	5	mil	48
ked	150	lat	130	lh-	19	lpq	1	mak	54	mim	1
kel	3	lau	5	lia	64	lpr	2	mal	159	min	357
ken	58	lay	47	lib	73	lps	3	man	745	mis	124
kep	11	la-	11	lic	108	lp-	20	map	41	mit	243
keq	4	lbe	1	lid	72	lq-	4	mar	85	mix	1
ker	25	lbo	3	lie	63	lre	40	mas	157	miz	6
kes	35	lb-	2	lif	16	lrm	10	mat	357	mkd	8
ket	272	lca	2	lig	21	lr-	2	max	59	mkn	10
kew	3	lch	1	lik	30	lsc	3	may	304	mle	7
key	47	lco	1	lil	1	lse	44	ma-	8	mli	6
ke-	69	lcr	11	lim	203	lso	247	mbe	368	ml-	1
kfi	1	lcs	1	lin	848	lst	6	mbi	17	mne	10
kf-	5	lcu	15	lio	5	ls-	286	mbf	30	mns	11
kgr	14	lc-	4	lis	255	lta	23	mbo	107	mn-	25
kil	61	ldb	10	lit	51	ltd	1	mbv	1	mod	269
kin	104	ldc	2	liv	33	lte	37	mb-	15	mom	1
kip	9	lde	48	liz	43	lth	6	mca	9	mon	79
klo	6	ldi	5	li-	1	lti	66	mck	1	mor	242
kly	4	ldl	2	lks	1	lto	1	mcl	1	mos	40
kme	4	ldm	1	lk-	12	lts	39	mco	1	mot	51
kna	9	ldn	1	lma	1	ltw	1	mcr	1	mou	130
kne	6	ldr	20	lmb	1	lt-	305	mc-	2	mov	114
kno	34	lds	44	lmd	1	lua	10	mde	1	mo-	3
knp	1	ldw	1	lmh	1	lud	146	mdh	1	mpa	71
knr	11	ld-	310	lmo	3	lue	465	mdi	6	mpe	9
kop	25	lea	78	lmw	1	lug	4	md-	29	mpf	15
kpo	18	lec	58	lnb	1	lum	41	mea	42	mpg	3
ksi	3	led	115	lnd	1	lun	3	mec	24	mpi	63
ksl	25	lef	62	lnh	1	lup	11	med	213	mpl	261
ksp	14	leg	26	lnk	1	lur	7	meg	1	mpm	3
ksw	4	leh	6	lnw	1	lus	33	meh	11	mpo	136
ks-	182	lem	70	ln-	4	lut	15	mel	37	mpq	1
кта	1	len	262	loa	72	luv	1	mem	148	mpr	59
ktn	1	leo	1	lob	64	lve	16	men	570	mps	4
ktr	3	lep	13	loc	456	lvi	10	meo	29	mpt	149
kup	5	ler	118	log	184	lwa	35	mer	88	mpu	21
kus	1	les	570	lok	21	lwc	1	mes	328	mp-	132
kwa	13	let	164	lon	117	lwo	2	met	222	mr-	1
ky-	14	lev	55	lop	50	lw-	2	mev	16	mse	7
k--	903	lew	32	lor	7	lyi	11	mew	2	msg	55
lab	91	lex	15	los	84	lys	3	mez	5	mst	9
lac	155	ley	1	lot	2	lyx	1	me-	1443	ms-	127

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
mta	1	ndi	295	nha	4	not	931	nut	11	oda	3
mte	1	ndl	48	nhc	9	nou	28	nu-	2	ode	270
mti	3	ndm	6	nhi	4	nov	8	nva	86	odi	76
mt-	10	ndn	1	nhu	1	now	39	nvc	2	ods	2
muc	10	ndo	17	nh-	3	no-	258	nve	41	odu	81
mul	62	ndp	4	nic	52	npa	1	nvi	54	od-	80
mum	42	nds	284	nie	43	npr	11	nvo	63	oed	7
mun	33	ndt	5	nif	22	npt	1	nvp	8	oel	1
mus	142	ndu	2	nig	6	npu	233	nv-	20	oen	30
mut	2	ndx	1	nim	6	np-	1	nwa	3	oer	1
mv-	3	ndy	1	nin	220	nqe	1	nwi	5	oes	157
mw-	13	nd-	2237	nio	11	nqu	3	nw-	1	oex	2
myh	2	nea	8	nip	11	nre	20	nxi	6	oe-	3
my-	2	nec	149	niq	36	nro	38	nx-	3	ofb	2
m--	1440	ned	438	nis	32	nr-	14	nym	8	ofd	13
nab	40	nef	4	nit	111	nsa	1	nyn	2	ofe	2
nac	11	neg	32	niv	8	nse	89	nyo	4	ofi	61
nad	2	nei	16	nix	34	nsf	47	nyt	2	off	1
naf	2	nel	24	niz	31	nsh	6	nyw	5	ofm	1
nag	2	nem	12	njo	1	nsi	198	ny-	235	ofs	48
nai	1	nen	108	nju	4	nsl	32	nze	20	oft	27
nal	551	neo	10	nj-	2	nsm	13	n--	6479	of-	2337
nam	1328	neq	10	nke	6	nso	4	oad	46	ogd	3
nan	1	ner	114	nkn	3	nsp	6	oar	8	oge	27
nap	5	nes	231	nks	72	nsr	1	oat	34	ogf	3
nar	25	net	51	nk-	118	nst	147	oa-	1	ogi	104
nas	2	neu	2	nle	46	nsu	30	oba	47	ogl	16
nat	161	nev	27	nli	29	nsw	2	obe	7	ogm	9
nba	2	new	250	nlo	5	ns-	891	obi	5	ogn	20
nbl	12	nex	59	nly	232	nta	435	obj	100	ogo	12
nbp	2	ne-	886	nl-	1	ntc	1	obl	8	ogr	175
nbr	1	nfd	6	nma	9	ntd	1	obo	2	ogs	3
nbu	4	nfe	1	nmb	1	nte	729	obs	44	ogy	1
nby	16	nfi	19	nmd	1	ntf	7	obt	15	og-	40
nb-	4	nfl	7	nme	65	nth	30	obu	8	oha	5
nca	36	nfo	95	nmh	1	nti	334	obv	1	ohu	5
nce	255	nfs	16	nmo	18	ntl	86	oby	12	oh-	6
nch	62	nfu	2	nmw	1	nto	107	ob-	77	oic	3
nci	6	nf-	4	nm-	2	ntr	236	oca	144	oid	11
ncl	169	nga	1	nob	10	nts	448	oce	684	oin	256
nco	69	nge	236	noc	3	ntu	3	ocg	1	okb	1
ncr	38	ngf	1	nod	52	nt-	1708	oci	53	oke	69
nct	127	ngi	22	noe	31	nua	28	ock	657	oki	12
ncu	1	ngl	81	nog	2	nue	47	ocl	4	oks	6
ncy	8	ngn	1	noh	12	nui	4	oco	71	oku	4
nc-	31	ngr	6	nom	7	nul	62	ocs	3	ok-	29
nda	188	ngs	66	non	131	num	386	oct	43	ola	17
ndb	2	ngt	66	nop	6	nuc	1	ocu	181	olc	11
ndc	2	ngu	51	nor	177	nup	2	ocw	1	old	46
nde	222	ng-	2059	nos	56	nus	22	oc-	2	ole	40

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
oli	70	opr	28	oun	334	pea	86	pno	6	put	499
olo	248	ops	26	oup	206	pec	979	poi	236	pu-	31
olr	5	opt	348	our	193	ped	87	pol	21	pwb	1
ols	34	opu	1	ous	112	pef	1	pon	207	pwd	11
olu	59	opy	31	out	539	pek	3	pop	16	pw-	2
olv	20	op-	145	ou-	92	pel	38	por	179	pyi	4
ol-	209	ora	32	ova	6	pen	239	pos	179	py-	29
oma	725	orb	6	ovc	7	per	569	pot	4	p--	3313
omb	12	orc	23	ove	220	pes	67	pou	2	qbl	3
omd	1	ord	444	ovi	44	pew	2	pov	1	qev	1
ome	161	ore	455	ovl	7	pe-	234	pow	7	qli	1
omh	1	orf	12	ovw	1	pfd	5	po-	1	qnc	1
omi	37	org	3	ov-	38	pfi	2	pqp	1	qn-	19
oml	4	ori	123	owa	3	pfn	1	pqr	1	qpa	4
omo	21	ork	107	owe	119	pfw	1	pqu	1	qpl	1
omp	434	orl	3	owi	137	pf-	17	pra	3	qrt	5
omu	29	orm	391	owm	1	pgr	31	prc	1	qua	50
omw	1	oro	4	own	144	pg-	15	pre	637	que	217
om-	390	orp	5	owo	1	pha	14	prf	2	qui	91
ona	78	ors	209	owr	1	phe	16	pri	429	quo	142
onb	10	ort	219	ows	74	phi	5	pro	1260	qv-	1
onc	32	oru	2	owt	6	pho	13	prs	1	q--	32
ond	170	orw	20	ow-	174	phr	18	pr-	58	rab	6
one	532	ory	499	oxe	1	phs	6	psc	1	rac	529
onf	18	or-	2239	oxi	5	phy	8	psd	2	rad	18
ong	109	ose	175	oxy	4	ph-	23	pse	3	rag	25
onh	1	osi	132	ox-	18	pic	13	psi	4	rah	1
oni	42	osm	1	oye	6	pid	34	pso	1	rai	31
onj	4	osn	11	oy-	2	pie	41	pst	2	ral	40
onl	232	osp	9	oze	2	pil	73	psw	2	ram	351
onm	45	ost	170	o--	3326	pin	52	ps-	116	ran	201
ono	30	osu	9	pab	14	pio	1	ptc	4	rap	80
onp	5	os-	50	pac	196	pip	64	pte	53	rar	50
onr	4	ota	118	pad	4	pir	9	ptf	9	ras	43
ons	780	otb	5	pag	98	pit	3	pth	1	rat	395
ont	466	otc	3	pai	36	pi-	6	pti	508	rau	1
onu	1	otd	31	pal	2	pla	205	ptl	9	rav	5
onv	51	ote	159	pan	69	plc	2	ptn	7	raw	18
ony	8	oth	284	pap	4	ple	264	pto	150	ray	42
onz	20	oti	40	par	462	pli	105	ptr	13	ra-	12
on-	2306	otn	1	pas	70	plo	6	pts	72	rba	1
opa	10	oto	94	pat	428	plp	1	ptu	4	rbe	4
opc	2	otr	2	pau	11	plu	11	ptv	11	rbi	18
opd	2	ots	26	paw	3	ply	68	pty	22	rbl	1
ope	308	oty	11	pa-	5	pma	2	pt-	245	rbo	13
oph	3	ot-	842	pco	3	pmb	1	pub	6	rbs	1
opi	42	oub	65	pc-	21	pme	1	pul	12	rb-	8
opm	1	ouc	18	pda	15	pmh	1	pun	1	rca	3
opn	6	oug	62	pdp	3	pmw	1	pur	20	rce	173
opo	13	oul	194	pd-	2	pm-	3	pus	16	rch	163

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
rci	3	rf-	2	rmo	2	rta	26	sap	1	sgh	5
rcl	3	rga	2	rms	25	rtb	1	sar	35	sgi	3
rcm	2	rgc	3	rmw	2	rtc	2	sas	4	sgr	9
rco	2	rge	44	rm-	180	rte	73	sat	1	sgs	5
rcp	2	rgi	9	rna	55	rtf	1	sau	2	sg-	48
rct	1	rgp	5	rnd	1	rth	34	sav	39	sha	33
rcu	11	rgs	8	rne	218	rti	85	say	6	shd	6
rcv	4	rgu	251	rmi	25	rtm	3	sa-	6	she	286
rc-	29	rgv	20	mm	2	rtn	1	sbm	1	shi	41
rda	1	rg-	33	rno	76	rto	3	sbr	10	shm	1
rde	90	rh-	1	rns	105	rtr	9	sca	86	shn	1
rdi	43	ria	253	rn-	243	rts	72	sce	17	sho	133
rdl	14	rib	47	roa	8	rtu	27	sch	6	shr	12
rdm	3	ric	55	rob	11	rtx	1	sci	41	shs	1
rdo	1	rid	9	roc	686	rtz	3	sck	1	shu	12
rds	96	rie	104	rod	73	rt-	2	scn	2	sh-	174
rdw	20	rif	7	rof	176	rt-	234	sco	21	sib	77
rdy	3	rig	65	rog	172	rub	1	scp	10	sic	8
rd-	442	rik	4	rok	14	ruc	223	scr	452	sid	185
rea	771	ril	15	rol	102	rue	43	scs	1	sie	6
reb	22	rim	10	rom	427	rui	4	sct	1	sif	1
rec	681	rin	599	ron	85	rul	12	scu	3	sig	591
red	413	rio	75	rop	63	run	108	sc-	22	sil	7
ref	258	rip	392	ror	397	rup	59	sdb	1	sim	65
reg	227	ris	21	ros	37	rus	34	sd-	1	sin	236
reh	5	rit	371	rot	132	ru-	34	sdi	6	sio	353
rei	17	riv	36	rou	337	rva	29	sd-	8	sip	1
rej	2	rix	2	rov	44	rve	52	sea	106	sir	15
rek	1	riz	8	row	29	rvi	7	sec	192	sis	35
rel	61	ri-	3	rox	18	rv-	1	sed	486	sit	72
rem	215	rke	26	roy	6	rwa	20	sef	38	siu	1
ren	580	rki	17	roz	1	rwc	1	seg	40	siv	32
reo	2	rks	20	ro-	128	rwi	86	sek	32	six	5
rep	217	rk-	115	rpa	8	rwr	4	sel	79	siz	193
req	146	rla	5	rpi	2	rw-	4	sem	41	si-	7
rer	7	rld	3	rpo	20	ryi	17	sen	225	sju	1
res	1030	rle	5	rpr	67	ryp	3	sep	75	ske	32
ret	533	rli	62	rps	1	ryt	8	seq	60	ski	11
reu	10	rlo	1	rp-	36	ryu	5	ser	504	skn	1
rev	113	rlp	7	rq-	4	ry-	1	ses	311	sks	6
rew	2	rly	30	rsa	15	r--	735	set	759	sk-	179
rez	1	rl-	3	rsc	12	sab	6180	seu	1	sla	77
re-	1630	rma	255	rse	34	sac	25	sev	10	slc	1
rfa	11	rmb	2	rsh	3	saf	2	se-	1005	sld	1
rfd	4	rmc	9	rsi	44	sag	1	sfe	44	sle	15
rfi	12	rmd	8	rso	43	sai	183	sfi	1	sli	6
rfl	4	rme	33	rst	159	sal	1	sfo	3	slo	29
rfo	43	rmh	2	rsv	1	sam	9	sfu	51	sly	20
rfr	4	rmi	385	rsw	1	san	89	sfy	1	sl-	2
rfr	4	rml	1	rs-	862			sf-	5	sma	23

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
smc	1	str	480	tao	4	tha	546	tne	2	tsu	6
smi	13	sts	74	tap	5	thd	2	tno	2	ts-	1183
smo	5	stu	8	tar	109	the	8131	tn-	1	ttv	2
sms	2	stv	1	tas	25	thi	503	tob	3	tua	61
sm-	31	stw	3	tat	310	thl	2	toc	71	tud	1
sna	8	sty	11	tav	3	thm	33	tod	6	tue	2
snd	4	st-	924	tax	193	thn	160	toe	2	tug	1
sne	3	sua	37	ta-	228	tho	130	tof	3	tui	15
sn-	5	sub	207	tbi	1	thr	54	tog	28	tun	5
soc	395	suc	158	tbl	21	ths	4	toi	7	tup	6
soe	1	sue	12	tbm	1	thu	34	tok	13	tur	579
sof	16	suf	41	tbr	1	th-	924	tol	37	tus	80
sol	29	sug	1	tbs	5	tia	88	tom	45	tut	102
som	65	sui	11	tb-	1	tib	15	ton	9	tuv	1
son	40	sul	87	tch	182	tic	235	top	136	tuw	1
sor	102	sum	68	tco	9	tid	16	tor	777	tva	11
sou	149	sun	2	tcp	5	tie	22	tos	1	tvp	6
so-	312	sup	238	tc-	66	tif	87	tot	33	tvx	1
spa	189	sur	20	tdh	2	tig	5	tou	23	tv-	4
spc	9	sus	10	tdi	44	til	61	tov	1	twa	6
spe	1019	su-	1	tdo	33	tim	396	tow	5	twe	40
spg	1	svc	12	tdt	11	tin	578	to-	2335	twi	14
spi	2	svt	3	td-	6	tio	1835	tpa	15	two	90
spl	46	sv-	18	tea	21	tip	41	tpe	6	tw-	6
spo	63	swa	34	tec	25	tir	51	tpg	15	txb	1
spr	4	swd	13	ted	1119	tis	38	tpi	12	txt	5
spu	1	swe	1	tef	6	tit	128	tpr	20	tx-	48
sp-	40	swi	30	teg	57	tiv	232	tpu	212	tyl	4
sqr	4	swo	11	teh	1	tiw	1	tp-	52	typ	265
squ	5	swr	1	tel	49	tiz	2	tql	1	tyt	9
sra	2	swt	1	tem	624	tj-	11	tqu	7	tyx	1
src	16	sw-	17	ten	206	tjm	3	tq-	4	ty-	166
srl	2	sx-	1	tep	14	tj-	19	tra	288	tzp	7
sro	6	sym	120	ter	1715	tjh	5	trc	9	tz-	4
srs	2	syn	255	tes	406	tli	1	tre	68	t--	8286
sru	2	sys	496	teu	4	tlo	7	tri	259	uad	2
sr-	87	sy-	23	tev	6	tlp	2	trl	24	uag	31
sta	678	sza	1	tex	177	tly	95	trm	6	ual	142
stb	1	sz-	2	te-	683	tl-	52	trn	2	uan	14
stc	6	s--	13603	tfa	1	tma	16	tro	160	uar	19
std	11	tab	170	tfb	1	tmb	1	trs	2	uat	19
ste	682	tac	136	tfd	10	tmd	1	tru	301	uba	1
stf	5	tad	1	tfl	7	tme	4	try	72	ubd	10
sth	1	tag	56	tfl	5	tmh	1	tr-	18	ube	2
sti	291	tah	4	tfo	2	tmo	2	tsc	1	ubf	2
stl	3	tai	247	tfs	24	tmp	24	tse	49	ubi	1
stm	6	tak	56	tf-	8	tmw	1	tsi	42	ubj	7
stn	13	tal	77	tgi	9	tm-	1	tsl	16	ubl	71
sto	216	tam	2	tgo	2	tna	19	tso	49	ubm	1
stp	5	tan	203	tgr	17	tnd	2	tst	9	ubp	16

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
ubr	4	uin	6	uot	142	uti	254	vir	76	wev	25
ubs	135	uio	2	uou	9	utl	1	vis	23	we-	11
ubt	21	uir	49	upd	15	utm	4	vit	9	wha	36
ub-	9	uis	7	upe	118	uto	74	vi-	22	whe	393
uca	2	uit	21	upf	4	utp	212	vle	7	whi	399
ucb	5	uiv	18	upg	2	uts	45	vmb	1	who	85
uce	166	ui-	1	upi	7	utu	13	vmh	1	wic	6
uch	88	ula	142	upl	43	utx	42	vms	11	wid	9
uci	3	ulb	1	upo	104	uty	4	vmu	3	wil	201
uck	2	uld	193	upr	32	ut-	795	vmw	1	win	151
ucp	2	ule	19	ups	48	uvx	1	vm-	1	wir	7
uct	232	uli	12	upt	59	uv-	1	vne	1	wis	105
udc	1	ulo	5	upy	2	uwv	1	vn-	2	wit	706
ude	127	uls	1	up-	286	ux-	2	voc	4	wk-	6
udi	30	ult	384	ura	8	u--	183	voi	11	wle	1
udo	1	ulw	2	urb	5	vai	66	vok	45	wli	46
udp	3	uly	10	urc	145	val	684	vol	25	wly	5
udr	2	ul-	175	ure	574	van	7	vor	2	wma	1
uds	1	uma	32	urf	2	vap	1	vp-	14	wm-	2
udy	4	umb	343	urg	6	var	230	vrf	2	wne	55
ud-	12	umd	1	uri	48	vat	10	vr-	1	wno	5
ued	16	ume	332	url	1	vax	3	vs-	1	wns	1
uei	1	umf	6	urm	1	vbu	2	vta	2	wn-	93
uel	4	umh	1	urn	479	vcn	7	vtx	3	wod	2
uen	60	umi	4	uro	5	vcs	4	vt-	5	wop	1
uer	4	umn	36	urp	13	vc-	13	vw-	1	wor	369
ues	212	ump	11	urs	81	vec	61	vx-	3	wou	71
ueu	20	ums	3	urt	27	ved	102	v--	365	wo-	68
uez	2	umu	3	uru	2	vef	2	wab	3	wq-	1
ue-	499	umw	1	urv	1	veg	1	wai	63	wra	7
ufd	1	um-	84	ur-	67	veh	2	wak	1	wri	274
ufe	110	una	27	usa	42	vel	88	wal	5	wro	7
ufi	37	unb	6	use	1094	vem	1	wan	10	wr-	3
ufm	2	unc	179	usg	1	ven	207	wap	33	wsg	7
ufs	23	und	143	ush	18	ver	337	war	70	ws-	1
uft	1	une	18	usi	107	ves	89	was	139	wsr	2
ufw	1	ung	1	usl	20	ve-	490	way	73	ws-	117
uf-	57	unh	3	usp	10	vfd	1	wa-	2	wth	1
ugb	4	uni	161	usr	86	vfo	14	wba	1	wtm	5
uge	17	unk	6	ust	191	vfr	5	wcs	7	wto	5
ugh	71	unl	71	usu	30	vfs	1	wcw	1	wt-	1
ugi	15	unm	21	usy	13	vfw	1	wc-	6	wun	3
ugo	2	unp	4	us-	232	vha	5	wdi	12	wvx	1
ugs	2	unr	16	uta	19	via	24	wd-	34	wxr	2
ugu	7	uns	55	utb	2	vic	88	wed	62	wxs	1
ug-	7	unt	307	utc	3	vid	56	wek	5	wx-	1
uic	6	unu	7	utd	9	vie	5	wel	17	w--	399
uid	58	unw	5	ute	227	vil	9	wen	41	xac	8
uie	2	unx	1	utg	2	vin	51	wer	76	xad	17
uil	25	un-	47	uth	3	vio	71	wes	10	xam	120

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
xbi	4	xtr	11	ysl	1	-at	329	-dn	13	-fu	190
xbs	1	xtu	2	yst	388	-au	85	-do	430	-fv	8
xce	135	xt-	222	ys-	170	-av	79	-dq	3	-f-	230
xch	5	xum	3	yte	140	-aw	14	-dr	15	-ga	12
xcl	26	xup	1	yth	7	-ax	1	-ds	32	-gc	6
xcp	2	xu-	2	yty	9	-a-	29	-dt	1	-ge	306
xcr	2	xw-	1	yus	1	-ba	204	-du	103	-gf	1
xde	3	xx-	1	ywh	5	-bc	16	-dv	10	-gh	1
xd-	1	xy-	9	ywo	13	-be	1424	-dw	2	-gi	159
xec	341	x--	587	yx-	2	-bf	8	-dx	7	-gl	48
xed	13	yac	1	yze	4	-bg	4	-dy	11	-gm	10
xer	9	yad	4	y--	3361	-bi	249	-d-	42	-go	30
xes	19	yan	6	zab	1	-bl	179	-ea	240	-gp	13
xex	1	ybo	4	zar	1	-bm	2	-eb	50	-gr	268
xfs	2	ycl	7	zat	19	-bn	1	-ec	45	-gs	1
xha	18	ydh	2	zbm	1	-bo	93	-ed	147	-gt	3
xhi	1	yea	11	zbr	1	-bp	28	-ef	168	-gu	10
xho	1	yed	15	zca	4	-br	127	-eg	63	-g-	2
xic	4	yel	2	zed	50	-bs	19	-eh	2	-ha	401
xim	46	yer	1	zen	2	-bu	296	-ei	176	-he	79
xin	4	yet	4	zeo	5	-bx	1	-ek	1	-hi	100
xio	6	ye-	1	zer	97	-by	777	-el	71	-hl	3
xis	140	yfo	1	zes	21	-b-	190	-em	36	-ho	155
xit	77	yhl	2	ze-	179	-ca	979	-en	650	-hs	1
xli	2	yie	18	zin	1	-cb	15	-eo	19	-ht	2
xmp	6	yin	43	ziv	2	-cd	22	-ep	54	-hu	8
xm-	1	yle	6	zma	1	-ce	35	-eq	59	-hy	13
xna	4	yli	4	zmb	1	-cf	2	-er	501	-h-	3
xny	2	ymb	101	zmd	1	-ch	864	-es	61	-ib	10
xn-	1	ymd	1	zmh	1	-ci	14	-et	65	-ic	1
xof	1	yml	5	zmw	1	-cl	106	-eu	8	-id	291
xpa	64	ymo	2	zon	12	-cm	49	-ev	102	-ie	35
xpe	13	ymp	1	zp-	7	-co	2353	-ew	10	-if	840
xpi	9	ym-	19	zr-	1	-cp	52	-ex	1277	-ig	70
xpl	22	yna	7	z--	18	-cr	192	-e-	236	-il	11
xpo	19	ync	50	-ab	127	-cs	57	-fa	183	-im	126
xpr	199	yno	10	-ac	327	-ct	15	-fc	58	-in	3073
xp-	10	ynq	1	-ad	490	-cu	329	-fd	77	-io	78
xrs	4	ynt	195	-af	125	-cw	3	-fe	28	-ip	16
xrw	2	yn-	2	-ag	16	-cy	10	-fg	14	-ir	3
xr-	2	yon	7	-ah	1	-c-	97	-fh	2	-is	2643
xse	3	you	108	-ai	2	-da	270	-fi	2079	-it	592
xst	2	ypa	3	-ak	1	-db	16	-fl	139	-iu	3
xsy	2	ype	247	-al	779	-dc	17	-fm	2	-iw	2
xs-	2	yph	13	-am	49	-de	1281	-fn	2	-ix	2
xtb	5	ypi	19	-an	2259	-df	12	-fo	1332	-i-	196
xte	47	ypt	8	-ao	29	-dg	8	-fp	1	-ja	22
xti	1	ysc	8	-ap	215	-di	842	-fr	433	-jo	116
xtn	2	ysh	1	-ar	1326	-dk	6	-fs	34	-jp	4
xto	1	ysi	9	-as	655	-dm	3	-ft	83	-ju	30

TRIGRAM FREQUENCIES TABLES FOR THE UNIX MANUAL											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
-jw	1	-ni	12	-q-	1	-ti	325	-wr	277	--s	8299
-kb	1	-nj	2	-ra	104	-tl	11	-ws	3	--t	13155
-ke	79	-nl	1	-rb	6	-tm	18	-wt	5	--u	1803
-ki	64	-nm	4	-rc	6	-tn	1	-wu	3	--v	1105
-km	4	-no	1233	-rd	10	-to	2325	-w-	2	--w	2787
-kn	21	-np	1	-re	2322	-tp	6	-xc	2	--x	142
-la	209	-nr	40	-rf	2	-tr	342	-xe	9	--y	165
-lb	2	-ns	11	-rg	3	-ts	61	-xm	6	--z	95
-lc	5	-nt	4	-ri	47	-tt	2	-xn	3		
-ld	10	-nu	415	-rl	33	-tu	38	-xs	5		
-le	293	-nv	2	-rm	6	-tv	8	-xt	2		
-lf	3	-nx	2	-rn	1	-tw	72	-xu	2		
-lg	5	-ny	1	-ro	95	-tx	2	-xx	1		
-lh	1	-n-	35	-rp	3	-ty	281	-xy	5		
-li	1086	-ob	131	-rq	4	-tz	9	-x-	107		
-lm	5	-oc	186	-rs	6	-t-	31	-ya	7		
-ln	8	-od	3	-rt	1	-uc	9	-yd	2		
-lo	512	-of	2270	-ru	141	-ud	4	-ye	17		
-lp	7	-oh	1	-rv	1	-ue	1	-yi	18		
-lq	4	-ok	5	-rw	20	-uf	6	-ym	1		
-ls	46	-ol	30	-r-	538	-ug	1	-yn	1		
-lt	2	-om	24	-sa	139	-ui	20	-yo	107		
-lw	2	-on	1025	-sb	11	-ul	9	-yp	1		
-ly	1	-op	584	-sc	133	-um	23	-y-	11		
-l-	2	-or	974	-sd	1	-un	454	-zb	2		
-ma	1128	-os	5	-se	1567	-up	126	-zc	4		
-mb	6	-ot	205	-sf	5	-ur	2	-ze	74		
-mc	4	-ou	322	-sh	485	-us	1081	-zi	2		
-md	1	-ov	57	-si	851	-ut	65	-zm	5		
-me	380	-ow	63	-sk	10	-ux	2	-zo	5		
-mg	5	-o-	3	-sl	56	-va	730	-zr	1		
-mi	73	-pa	924	-sm	25	-ve	69	-z-	2		
-mk	18	-pc	8	-sn	2	-vf	15	--a	6938		
-ml	1	-pd	3	-so	561	-vh	5	--b	3618		
-mn	10	-pe	262	-sp	1231	-vi	75	--c	5194		
-mo	526	-pf	6	-sq	9	-vm	9	--d	3124		
-mp	13	-pg	14	-sr	16	-vn	2	--e	4041		
-ms	48	-ph	41	-st	1120	-vo	13	--f	4895		
-mt	6	-pi	92	-su	681	-vs	1	--g	870		
-mu	201	-pl	116	-sv	28	-vt	5	--h	765		
-mv	3	-pm	3	-sw	64	-v-	181	--i	7992		
-my	4	-po	422	-sy	829	-wa	263	--j	173		
-m-	111	-pr	1855	-sz	1	-wc	9	--k	169		
-na	968	-ps	8	-s-	474	-we	83	--l	2203		
-nb	21	-pt	31	-ta	201	-wh	905	--m	2538		
-nc	1	-pu	68	-tb	15	-wi	900	--n	3253		
-nd	3	-pw	12	-tc	7	-wm	2	--o	5888		
-ne	464	-p-	2339	-td	1	-wn	5	--p	6204		
-nf	27	-qu	198	-te	577	-wo	329	--q	200		
-ng	6	-qv	1	-th	8822	-wq	1	--r	3349		

TRIGRAM FREQUENCIES TABLES FOR THESE MSc THESES

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
aba	11	afz	1	alk	3	apl	76	atg	1	azi	9
abc	1	af-	3	alm	10	apo	1	ath	52	az-	1
abd	1	aga	14	alo	34	apr	37	ati	886	a--	320
abe	29	agd	1	alp	10	aps	11	atk	1	bab	20
abf	1	age	320	alr	8	apt	58	atl	5	bac	44
abi	51	agi	5	als	105	apu	1	atm	1	bad	5
abl	210	agl	2	alt	26	apy	1	atn	1	bag	7
abm	1	agn	7	alu	98	ap-	22	ato	83	bai	1
abn	1	ago	2	alv	6	aqu	1	atr	46	bak	1
abo	88	agr	36	alw	16	ara	557	ats	3	bal	32
abr	2	ags	3	aly	137	arb	14	atu	67	ban	13
abs	25	agt	1	al-	1059	arc	120	aty	2	bar	5
abu	1	agu	1	ama	8	ard	118	at-	645	bas	117
aby	2	agy	1	amb	14	are	571	aue	1	bay	7
ab-	6	ag-	17	amd	1	arg	96	aug	3	ba-	2
aca	2	aha	5	ame	193	arh	3	aul	7	bcl	1
acd	1	ahe	3	amh	1	ari	142	aus	73	bcu	7
ace	169	ahl	2	ami	81	ark	40	aut	17	bc-	1
ach	267	ahm	1	aml	3	arl	19	au-	1	bda	11
aci	27	ah-	2	amo	45	arm	1	ava	38	bde	1
ack	84	aic	3	amp	91	arn	12	ave	182	bea	3
acl	1	aid	14	ams	48	aro	9	avi	20	bec	80
acm	7	aig	9	amw	1	arp	1	avl	4	bed	51
aco	19	ail	61	am-	132	ars	17	avo	6	bef	6
acq	10	aim	3	ana	50	art	173	avy	1	beg	9
acr	5	ain	229	anc	117	aru	1	awa	4	beh	5
acs	1	air	19	and	1086	ary	95	awi	4	bei	12
act	405	ais	2	ane	153	arz	2	awn	3	bel	65
acu	57	ait	4	anf	5	ar-	92	aws	2	ben	98
acy	3	aiv	1	ang	82	asa	3	aw-	10	ber	135
ac-	3	ai-	12	ani	63	asc	17	axb	1	bes	57
ada	4	ajf	1	ank	9	ase	233	axe	1	bet	97
adc	3	ajo	7	anl	2	ash	25	axh	1	bew	1
ade	56	aka	1	anm	1	asi	192	axi	67	bex	1
adi	32	ake	63	ano	37	ask	13	axn	1	bey	2
adj	13	aki	10	anp	1	asm	1	axp	1	be-	738
adl	4	akn	1	anr	1	asn	2	axr	1	bfa	1
adm	1	akp	2	ans	156	aso	40	axu	1	bfh	1
ado	12	aks	3	ant	130	asp	17	ax-	14	bfi	1
adr	59	akt	1	anu	10	ast	108	aya	1	bfr	2
ads	14	akw	1	anv	1	asu	43	ayb	1	bft	1
adu	1	ak-	6	anw	1	asv	3	aye	15	bf-	1
adv	43	ala	23	any	87	asw	1	ayi	8	bgr	5
ady	9	alb	1	an-	689	asx	1	ayl	1	bg-	1
ad-	106	alc	48	aoi	1	asy	9	aym	1	bia	4
ael	1	ald	3	ao-	1	as-	614	ayo	1	bib	2
afr	1	ale	221	apa	16	ata	184	ays	49	bic	1
afs	1	alf	4	ape	55	atc	19	ay-	205	bid	1
aft	32	alg	158	aph	104	ate	518	aza	1	bie	3
afw	1	ali	72	api	10	atf	1	aze	1	bif	1

TRIGRAM FREQUENCIES TABLES FOR THESE MSc THESES

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
big	15	bte	1	cho	22	cor	119	dav	2	dlo	3
bih	1	btr	2	chp	5	cos	12	day	6	dly	11
bil	69	buf	12	chr	11	cou	51	da-	21	dma	5
bim	5	bug	8	chs	2	cov	9	dbo	2	dme	13
bin	63	bui	33	cht	2	cox	4	dca	3	dmi	1
bio	2	bul	1	chu	1	co-	4	dea	26	dne	2
bir	5	bun	2	ch-	461	cpu	6	deb	3	doc	10
bit	58	bur	1	cia	88	cp-	1	dec	63	doe	21
biv	2	bus	8	cib	1	cqu	8	ded	75	doi	2
biw	1	but	131	cid	7	cq-	2	def	76	dol	17
bi-	4	buy	1	cie	50	cra	7	deg	4	dom	19
bje	52	bu-	1	cif	24	cre	61	dek	3	don	49
bla	19	bvi	3	cil	13	cri	151	del	74	dop	1
ble	425	byt	51	cim	2	cro	20	dem	12	dor	2
bli	15	by-	275	cin	10	crt	1	den	91	dos	1
blo	17	b--	23	cip	12	cru	2	deo	2	dot	3
blu	1	cab	2	cir	19	cr-	10	dep	62	dou	7
bly	13	cac	1	cis	22	cs-	40	deq	3	dow	9
bl-	2	cad	6	cit	14	cta	11	der	147	do-	56
bma	2	cal	472	ci-	26	cte	367	des	281	dpo	4
bm-	13	cam	23	cka	6	cti	478	det	84	dpr	2
bn-	1	can	379	cke	16	ctl	20	dev	62	dra	32
boa	22	cap	13	ckg	19	ctn	1	dex	3	dre	63
boc	1	car	29	ckh	12	cto	81	de-	208	dri	2
bod	3	cas	40	cki	12	ctr	8	dfo	12	dry	2
bok	11	cat	249	ckl	1	cts	55	dge	103	dr-	17
bol	20	cau	67	ckn	4	ctu	181	dha	2	dsc	2
bon	14	cav	2	ckq	1	ct-	159	dia	13	dso	4
bor	8	ca-	17	cks	12	cub	255	dib	1	ds-	200
bot	46	cdr	3	ckt	2	cui	6	dic	29	dth	13
bou	83	cd-	1	ckw	2	cul	75	did	7	dti	1
bov	25	cea	2	ck-	88	cum	63	die	5	dua	5
box	4	cec	1	cla	159	cup	8	dif	117	duc	91
bo-	2	ced	182	cle	70	cur	109	dig	7	dud	8
bph	3	cei	10	cli	12	cus	53	dil	5	due	6
bpr	1	cel	37	clo	26	cut	37	dim	57	dul	11
bra	28	cem	1	clu	51	cu-	2	din	125	dun	5
bre	19	cen	48	cly	3	cyb	3	dio	3	dup	2
bri	12	cep	49	cl-	4	cyc	45	dir	51	dur	121
bro	15	cer	44	cmo	1	cy-	20	dis	192	dut	16
bru	2	ces	585	cm-	7	c--	242	dit	37	dva	42
br-	2	cet	14	coa	1	dab	2	diu	3	dvi	1
bsd	2	ce-	394	cob	1	dad	11	div	22	dwa	18
bse	22	cge	1	cod	24	dai	1	dix	12	dwh	6
bso	2	cgr	4	cog	236	dal	2	diz	1	dwi	10
bsp	1	cha	419	coh	5	dam	1	dja	11	dwr	7
bst	21	che	158	col	22	dan	4	dje	1	dyn	8
bsy	5	chi	207	com	507	dap	4	dju	1	dy-	27
bs-	8	chm	15	con	600	dar	49	dle	10	d--	3432
bta	20	chn	98	cop	18	dat	185	dli	4	eab	3

TRIGRAM FREQUENCIES TABLES FOR THESE MSc THESES

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
eac	104	ego	1	ene	93	esa	61	eyw	4	fps	2
ead	121	egr	10	eng	37	esc	144	ey-	71	fpu	4
eaf	2	egs	1	enh	2	ese	323	eze	5	fra	77
eai	1	egu	1	eni	8	esf	5	ezi	2	fre	52
eak	17	egy	5	enk	3	esh	6	e--	9461	fri	2
eal	32	eg-	20	eno	17	esi	211	fac	82	fro	135
eam	27	eha	3	ens	127	esk	1	fai	3	fru	2
ean	55	ehe	1	ent	930	esl	5	fal	5	fr-	2
ear	126	ehi	4	env	46	esn	1	fam	9	fse	1
eas	135	eho	1	eny	1	eso	224	far	1	fs-	17
eat	82	eid	7	en-	549	esp	53	fas	30	fte	49
eav	5	eig	34	eof	2	est	184	fau	7	fth	2
ea-	35	eil	1	eom	3	esu	72	fa-	1	fti	2
ebr	2	ein	13	eon	2	es-	1522	fca	1	ftw	21
ebt	1	eir	30	eop	1	eta	25	fea	37	ft-	52
ebu	2	eis	3	eor	16	etc	42	fec	13	ful	37
eby	3	eit	17	eou	12	ete	194	fed	2	fun	98
eca	64	eiv	10	eov	6	eth	172	fel	6	fur	20
ece	61	eje	4	eo-	2	eti	38	fen	1	fus	14
ech	119	eke	4	epa	25	eto	32	fer	149	fut	9
eci	93	ekn	2	epe	44	etr	30	fet	8	fuz	5
eck	45	ek-	1	eph	1	ets	20	few	14	fu-	2
ecl	3	ela	45	epi	17	etu	52	fge	8	fw-	1
eco	319	elb	1	epl	12	etw	94	fh-	1	fyi	8
ecr	7	elc	1	epo	23	ety	4	fib	14	fy-	9
ect	574	eld	16	epr	136	et-	208	fic	152	fz-	1
ecu	58	ele	107	eps	4	eud	2	fie	67	f--	1632
ec-	12	elf	38	ept	60	eum	5	fif	3	gag	1
ede	32	eli	199	ep-	17	eun	1	fig	148	gai	12
edg	99	elk	1	equ	129	eur	10	fil	68	gal	1
edi	36	elo	83	eq-	12	eus	3	fin	124	gam	1
edl	2	elp	6	era	232	eva	33	fir	84	gan	12
edn	2	els	127	erc	164	eve	224	fis	1	gap	4
edo	1	elt	3	ere	349	evi	48	fit	16	gar	16
eds	18	elv	5	erf	112	evo	1	fiv	4	gat	18
edu	146	ely	67	erg	16	ewe	6	fix	43	gaz	1
edw	1	el-	388	erh	11	ewh	4	fi-	2	ga-	1
ed-	1597	ema	39	eri	100	ewi	1	fla	14	gby	1
efa	8	emb	30	erl	33	ewo	4	fle	25	gd-	1
efe	37	eme	281	erm	85	ew-	65	fli	30	gea	2
efi	100	emi	5	em	225	exa	102	flo	37	ged	12
efl	8	emo	99	ero	28	exc	22	flu	6	geh	1
efo	41	emp	35	erp	58	exe	42	fly	3	gel	4
eft	19	ems	148	ers	357	exh	1	foc	6	gem	6
efu	10	emt	1	ert	169	exi	89	fol	79	gen	95
ef-	5	em-	329	eru	3	exp	97	fom	1	geo	3
ega	7	ena	21	erv	20	ext	129	fon	29	ger	23
ege	13	enb	2	erw	8	ex-	66	for	746	ges	97
egi	39	enc	188	ery	70	eye	3	fou	80	get	34
egm	75	end	173	er-	1247	eyo	1	fo-	1	ge-	402

TRIGRAM FREQUENCIES TABLES FOR THESE MSc THESES

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
gfu	2	gth	14	hic	163	hta	1	idl	6	ilr	7
gha	3	gt-	1	hid	2	htf	3	ids	2	ils	8
ghb	22	gua	49	hie	31	htl	2	idt	13	ilt	15
ghe	5	gue	19	hif	10	htn	2	idu	4	ilu	5
ghl	4	gui	13	hig	39	hty	1	id-	37	ily	21
gho	1	gul	3	hil	38	ht-	38	iec	8	il-	138
ghp	2	gum	43	him	2	hum	14	ied	52	ima	164
ghs	1	guo	2	hin	198	hun	3	ief	7	imd	22
ght	44	gup	1	hip	12	hus	18	iel	13	ime	158
gh-	135	gur	148	hir	15	hya	1	iem	3	imi	44
gic	34	gus	1	his	375	hyd	18	ien	73	imp	213
gid	3	guy	1	hit	53	hyp	110	ieo	2	ims	3
gie	14	gu-	1	hiz	3	hys	14	ier	63	imu	67
gin	54	gy-	48	hle	2	hy-	13	ies	143	im-	6
gio	12	g--	1195	hly	4	hz-	3	iet	4	ina	151
gis	11	hab	9	hma	16	h--	1222	iev	23	inc	110
git	8	had	6	hme	11	iab	58	iew	14	ind	92
giv	46	hae	1	hmi	2	iac	1	ie-	11	ine	480
gi-	4	hai	4	hms	38	iag	3	ifc	1	inf	100
gle	42	hal	35	hm-	113	ial	142	ife	82	ing	1140
gli	7	ham	1	hne	4	iam	3	ifi	190	inh	7
glo	12	han	215	hni	70	ian	14	ifo	1	ini	148
gly	2	hap	68	hno	27	iar	3	ifs	4	ink	21
gl-	2	har	295	hns	1	ias	2	ift	15	inl	4
gme	78	has	204	hn-	3	iat	36	ifu	2	inm	5
gna	8	hat	516	hoa	10	ia-	7	ify	14	ino	3
gnb	1	hav	164	hoc	1	ibe	72	if-	144	inp	94
gne	29	hay	4	hod	149	ibi	22	iga	16	ins	165
gni	251	ha-	1	hoe	5	ibl	83	ige	48	int	484
gnm	2	hbo	22	hof	24	ibm	12	igh	99	inu	44
gno	4	hdr	3	hog	1	ibo	14	igi	33	inv	32
gns	2	hea	8	hoi	5	ibr	20	igl	2	in-	1132
gn-	26	heb	2	hol	30	ibu	60	ign	80	iod	3
goa	7	hec	33	hom	1	ib-	1	igr	27	iog	2
god	30	hed	16	hon	3	ica	532	igu	151	iol	3
goe	3	hei	31	hop	1	ice	77	ig-	5	ion	1715
goi	3	hel	12	hor	40	ich	150	ihi	1	ior	13
gol	6	hem	107	hos	95	ici	62	iho	6	iou	42
gon	2	hen	200	hot	6	ick	5	ih-	8	io-	5
gor	153	heo	16	hou	171	icl	12	ijd	1	ipa	1
gos	2	hep	1	how	98	ico	7	ijf	3	ipb	7
got	2	her	390	ho-	7	icr	8	ijv	2	ipe	65
gou	1	hes	136	hpt	5	ics	38	ike	32	iph	1
go-	3	het	18	hpu	2	ict	31	ik-	17	ipi	4
gra	301	heu	6	hp-	1	icu	26	ila	53	ipk	2
gre	35	hey	57	hre	50	ic-	127	ild	24	ipl	49
gri	4	he-	3320	hri	5	ida	6	ile	106	ipm	2
gro	59	hfu	1	hro	37	ide	173	ili	88	ipo	1
gr-	2	hia	1	hr-	1	idh	2	ilm	1	ipr	9
gs-	35	hib	1	hs-	18	idi	18	ilo	5	ips	3

TRIGRAM FREQUENCIES TABLES FOR THESE MSc THESES											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
ipt	70	ixt	2	key	7	lcu	47	lkn	1	lus	33
ipu	8	ix-	101	ke-	48	lc-	1	lk-	2	lut	15
ip-	22	iza	5	kgr	19	lde	2	lme	3	lu-	2
iqu	68	ize	68	kho	15	ldi	10	lmo	7	lva	1
ira	2	izi	3	kil	1	lds	12	lm-	2	lve	43
irc	23	izo	23	kim	2	ld-	171	lne	1	lvi	2
ird	16	izu	3	kin	59	lea	39	loa	24	lv-	1
ire	103	iz-	7	kip	1	lec	40	lob	14	lwa	15
iro	51	i--	192	kis	3	led	160	loc	77	lw-	1
irs	89	jac	11	kit	3	lef	19	lod	1	lyg	1
irt	3	jap	8	ki-	2	leg	1	loe	1	lyi	15
iry	1	jay	1	kje	2	lcl	154	log	91	lyn	2
ir-	45	jds	1	kla	1	lem	288	loi	7	lys	37
isa	36	jec	70	klo	2	len	34	lok	9	lyt	1
isb	3	jer	1	kly	1	lep	1	lom	1	ly-	440
isc	58	jet	2	kne	2	ler	40	lon	27	l--	1747
isd	1	je-	1	kno	130	les	165	lop	66	mab	3
ise	136	jfe	3	kop	3	let	149	lor	4	mac	115
isf	6	jf-	1	kow	5	leu	2	los	25	mad	21
ish	30	joh	4	kpo	2	lev	55	lot	32	mag	124
isi	82	joi	9	kqu	1	lew	4	lou	2	mai	67
isk	8	jor	6	kra	1	lex	76	low	170	maj	6
ism	25	jou	15	ksh	1	ley	5	loy	7	mak	24
iso	15	jo-	1	ksp	2	le-	541	lo-	5	mal	70
isp	83	jr-	1	kst	4	lfi	1	lph	9	man	155
ist	320	jul	3	ks-	56	lfo	1	lps	1	map	22
isu	11	jum	2	kth	1	lft	24	lp-	6	mar	41
isy	6	jun	3	ktr	2	lf-	17	lre	7	mas	18
is-	1434	jur	2	kus	3	lga	1	lro	7	mat	202
ita	22	jus	12	kw-	2	lgo	157	lr-	1	max	45
itc	7	jve	2	ky-	4	lg-	1	lse	25	may	67
ite	197	j--	73	k--	232	lia	13	lsi	9	ma-	10
ith	577	kab	1	lab	57	lib	21	lsk	2	mbd	11
iti	350	kag	6	lac	43	lic	92	lso	58	mbe	136
itl	19	kah	6	lag	13	lid	5	ls-	163	mbi	35
itm	17	kan	1	lai	11	lie	47	lta	9	mbf	12
ito	7	kar	1	lam	11	lif	7	lte	14	mbo	17
itr	8	kas	2	lan	106	lig	45	lth	10	mby	9
its	113	kat	1	lap	3	lih	6	lti	78	mcb	5
itu	5	ka-	3	lar	99	lik	30	ltm	2	mci	7
ity	150	kby	23	las	166	lim	12	lto	1	md-	23
itz	1	kb-	2	lat	193	lin	327	lts	32	mea	71
it-	359	ked	17	lau	6	lio	3	lty	2	mec	18
ium	6	kel	26	lav	13	lip	7	ltz	2	med	35
iva	12	ken	16	lay	33	lis	212	lt-	68	meh	1
ive	186	kep	15	la-	5	lit	171	lua	26	mel	14
ivi	38	ker	31	lbr	1	liv	1	lud	28	mem	94
iv-	7	kes	29	lb-	1	liz	23	lue	72	men	484
iw-	1	ket	5	lco	1	li-	1	lum	12	mer	49
ixe	111	kev	11			lki	1	lur	1	mes	103

TRIGRAM FREQUENCIES TABLES FOR THESE MSc THESES

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
met	205	mt-	1	nea	11	nit	267	nst	150	obt	20
mew	7	muc	25	nec	137	niv	28	nsu	9	obu	3
me-	340	mud	4	ned	186	nix	27	nsw	4	obv	3
mfi	1	mul	158	nef	2	niz	3	ns-	440	oca	69
mfl	3	mum	41	neg	1	ni-	3	nta	317	oce	545
mhz	3	mun	75	nei	28	nke	1	nte	413	och	2
mh-	1	mus	37	nel	90	nki	6	nth	2	oci	9
mia	1	mut	14	nen	17	nkj	2	nti	172	ock	25
mic	25	mw-	1	neo	12	nkn	4	ntl	45	oco	4
mid	3	m--	921	ner	98	nks	6	nto	61	ocr	9
mig	4	nab	22	nes	132	nk-	21	ntr	118	oct	4
mil	27	nac	16	net	39	nle	3	nts	179	ocu	35
mim	13	nag	11	neu	8	nli	1	ntu	1	oc-	6
min	138	nai	1	nev	20	nlo	1	nty	2	oda	3
mip	1	nal	236	new	38	nly	74	nt-	599	ode	218
mir	6	nam	32	nex	35	nl-	1	nua	23	odi	12
mis	23	nan	1	ne-	510	nma	1	nuc	2	odo	16
mit	16	nar	28	nfe	12	nme	48	nue	16	ods	49
mix	2	nat	98	nfi	14	nmo	5	nuf	1	odu	70
mi-	15	nav	9	nfl	6	nm-	1	nui	1	ody	2
mle	2	naz	6	nfo	90	noc	1	nul	3	od-	127
mli	2	na-	6	nfr	5	nod	159	num	133	oen	7
mly	3	nbe	2	nfu	13	noi	13	nuo	3	oes	29
ml-	1	nbi	2	nge	34	nol	31	nus	5	oev	1
ms	2	nca	4	ngf	2	nom	4	nut	8	ofe	6
mn-	2	nce	338	ngg	1	non	18	nve	40	ofi	17
mod	63	nch	36	ngi	18	nop	14	nvi	46	ofl	9
mok	3	nci	28	ngl	49	nor	24	nvo	15	ofo	2
mom	7	ncl	48	ngo	1	nos	2	nv-	1	ofs	13
mon	32	nco	17	ngr	1	not	194	nwa	1	oft	34
mor	185	ncr	36	ngs	28	nou	18	nwr	4	of-	1447
mos	56	nct	101	ngt	14	nov	1	nw-	1	oga	2
mot	7	ncu	128	ngu	59	now	149	nxn	3	oge	15
mou	45	ncy	17	ng-	1092	no-	64	nym	2	ogi	42
mov	14	nc-	6	nha	2	npr	9	nyw	1	ogn	236
mo-	1	nda	52	nhe	7	npu	94	ny-	87	ogo	1
mpa	47	ndb	2	nib	1	np-	1	n--	4398	ogr	152
mpe	1	nde	87	nic	89	nqu	1	oac	28	ogs	1
mph	9	ndf	12	nie	15	nre	15	oad	19	ogu	1
mpi	18	ndi	106	nif	7	nr-	1	oal	7	ogy	42
mpl	320	ndl	15	nig	2	nsa	5	oam	10	og-	3
mpo	84	ndm	13	nih	1	nse	20	oar	23	ohe	5
mpr	21	ndo	19	nil	1	nsf	59	oat	13	ohn	4
mpt	13	ndp	6	nim	12	nsh	1	oba	30	oic	5
mpu	171	ndr	1	nin	158	nsi	178	obe	6	oid	8
mp-	4	nds	47	nio	5	nsk	6	obi	1	oil	2
mse	6	ndu	3	nip	10	nsl	2	obj	40	oin	112
mst	3	ndw	23	niq	68	nsm	2	obl	137	ois	13
ms-	263	ndy	1	nir	3	nso	14	obo	7	oit	6
mty	1	nd-	1164	nis	64	nsp	44	obs	8	oi-	1

TRIGRAM FREQUENCIES TABLES FOR THESE MSc THESES											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
oje	14	opm	20	ous	79	pfu	1	psy	4	rbi	8
oka	3	opo	19	out	177	pgr	1	ps-	34	rbo	1
oke	11	opr	12	oux	1	pha	47	pta	5	rca	1
okh	3	ops	6	ou-	3	phe	2	pte	62	rce	34
oks	3	opt	74	ova	7	phi	15	pth	4	rch	122
ok-	15	opu	5	ove	111	pho	9	pti	140	rci	8
ola	9	op-	27	ovi	67	phs	8	pto	8	rck	12
old	35	ora	34	ov-	2	phy	20	ptr	5	rcl	9
ole	33	orc	3	owa	10	ph-	90	pts	10	rco	25
olf	1	ord	223	owe	65	pic	36	ptu	3	rds	1
oli	12	ore	242	owi	63	pid	7	pty	3	rcu	115
olm	1	org	11	owl	76	pie	18	pt-	44	rde	47
olo	143	ori	267	own	83	pik	3	pub	8	rdi	32
ols	8	ork	88	ows	37	pil	18	pul	13	rdl	1
olu	20	orl	9	ow-	133	pin	26	pur	7	rdr	2
olv	36	orm	270	oxi	1	pio	1	pus	1	rds	63
oly	2	orn	9	ox-	8	pip	64	put	376	rdw	17
ol-	93	oro	3	oye	1	pit	2	puz	2	rd-	195
oma	33	orp	4	oyi	1	pix	91	pu-	7	rea	300
omb	30	ors	130	oys	3	pi-	1	pwa	1	reb	3
ome	125	ort	136	oy-	10	pko	2	pyr	1	rec	409
omf	1	orw	6	o--	1418	pla	77	py-	2	red	143
omi	11	ory	102	pab	6	ple	318	p--	306	ref	76
oml	3	or-	870	pac	62	pli	170	qrt	10	reg	48
omo	19	osa	5	pag	13	plo	19	qua	53	reh	1
omp	340	osb	1	pai	12	ply	25	que	145	rei	6
omu	74	ose	79	pal	4	pl-	1	qui	61	rej	4
om-	142	osf	4	pam	2	pma	1	quo	4	rek	2
ona	115	osi	114	pan	32	pme	21	qu-	1	rel	72
onc	111	oso	5	pap	9	poc	1	q--	16	rem	53
ond	74	ost	144	par	368	poi	99	rab	2	ren	172
one	341	osu	4	pas	36	pok	1	rac	309	reo	6
onf	24	os-	15	pat	195	pol	16	rad	9	rep	178
ong	38	ota	22	pau	1	pon	42	rag	52	req	63
oni	20	ote	26	pav	4	pop	5	rah	1	rer	1
onk	2	oth	141	pa-	2	por	92	rai	30	res	528
onl	70	oti	8	pbo	7	pos	165	ral	282	ret	122
onm	46	otl	12	pc-	9	pot	11	ram	302	reu	2
ono	12	oto	15	pda	3	pou	4	ran	171	rev	17
onq	1	ots	19	pea	22	pow	11	rap	109	rew	2
ons	415	oty	3	pec	127	po-	13	rar	43	rey	2
ont	246	ot-	178	ped	64	pra	17	ras	20	re-	1265
onv	23	oub	3	pel	96	pre	306	rat	206	rfa	33
ony	2	ouc	10	pen	87	pri	96	rau	1	rfe	5
on-	1727	oue	1	peo	1	pro	1088	rav	5	rfl	1
opa	1	oug	116	pep	1	prs	1	raw	16	rfo	67
ope	183	oul	109	per	413	pru	1	ray	93	rfr	1
oph	10	oun	147	pes	44	pr-	1	raz	1	rfu	5
opi	17	oup	31	pet	1	pse	2	ra-	27	rga	10
opl	1	our	154	pe-	114	psu	1	rba	6	rge	63

TRIGRAM FREQUENCIES TABLES FOR THESE MSc THESES

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
rgi	1	rn-	158	rul	52	sel	43	sk-	15	suc	81
rgr	3	roa	36	run	31	sem	36	sla	14	sue	10
rgu	43	rob	167	rup	3	sen	228	sle	4	suf	32
rg-	4	roc	547	rus	2	sep	19	sli	6	sug	8
rha	3	rod	59	rut	3	seq	35	slo	56	sui	11
rhe	5	roe	7	rva	4	ser	121	sly	17	sul	71
rho	5	rof	26	rve	32	ses	139	sma	30	sum	35
rhu	1	rog	150	rvi	9	set	60	smi	2	sun	3
ria	84	roj	14	rwa	6	seu	2	smo	3	sup	69
rib	134	rok	6	rwi	8	sev	16	sms	2	sur	37
ric	42	rol	56	ryb	1	sey	1	sm-	23	sus	11
rid	8	rom	137	ryi	1	se-	474	sno	2	su-	2
rie	90	ron	81	rys	4	sfe	10	snt	1	sve	1
rif	6	rop	47	ryt	1	sfi	3	sob	1	sv-	2
rig	63	ror	16	ry-	275	sfo	53	soc	9	swa	1
rik	16	ros	9	rze	2	sfu	5	sof	21	swe	6
ril	3	rot	22	r--	2423	sfy	3	soi	2	swi	6
rim	17	rou	103	sab	7	sha	42	sol	50	sw-	1
rin	104	rov	88	sac	4	she	11	som	76	sx-	1
rio	45	row	9	sad	12	shi	28	son	56	syc	4
rip	85	rox	1	sag	38	shm	3	sop	10	sym	20
ris	51	roy	8	sai	11	shn	3	sor	241	syn	38
rit	253	ro-	20	sal	7	sho	110	sou	20	sys	294
riv	14	rpa	3	sam	49	shr	4	so-	114	sy-	10
rix	33	rpo	8	san	4	sht	1	spa	58	s--	6068
riz	29	rpr	58	sar	25	sh-	37	spe	193	tab	81
ri-	27	rp-	1	sas	1	sia	4	spf	1	tac	29
rka	1	rsa	4	sat	26	sib	76	spi	4	tag	60
rke	20	rse	19	say	12	sic	35	spl	74	tai	136
rki	11	rsh	2	sa-	3	sid	51	spo	26	tak	44
rkl	2	rsi	60	sba	1	sie	1	spr	4	tal	44
rks	35	rso	5	sbi	3	sif	107	spu	35	tan	145
rk-	59	rsp	1	sca	61	sig	78	sp-	55	tar	26
rla	20	rst	88	sce	6	sil	12	sqr	10	tas	32
rld	8	rsu	7	sch	81	sim	74	squ	14	tat	276
rli	10	rs-	456	sci	28	sin	278	sta	223	tau	1
rly	22	rta	54	sco	19	sio	178	std	6	tax	2
rl-	1	rte	56	scr	150	sir	2	ste	397	ta-	159
rma	156	rth	29	scu	45	sis	85	sti	180	tch	53
rme	16	rti	135	sc-	6	sit	78	stl	8	tc-	35
rmi	46	rtm	6	sdr	1	siu	3	stm	5	tdi	3
rmo	8	rtr	15	sdt	1	siv	38	sto	92	tdo	3
rms	29	rts	34	sd-	3	six	6	stp	22	tea	19
rmu	20	rtu	13	sea	68	siz	39	str	365	tec	191
rm-	81	rtw	1	sec	57	si-	18	st-	54	ted	481
rna	48	rt-	3	sed	317	ske	21	stu	26	tef	1
rne	29	rt-	168	sef	6	ski	5	sty	5	teg	30
rni	26	ruc	203	seg	75	skl	1	st-	458	tel	76
rno	3	rue	15	sei	1	sks	4	sua	5	tem	319
rns	54	rui	1	sek	2	sky	4	sub	71	ten	108

TRIGRAM FREQUENCIES TABLES FOR THESE MSc THESES

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
tep	9	tob	3	ty-	168	uga	3	unr	2	uti	84
ter	1110	toc	5	tz-	3	uge	6	uns	12	utl	4
tes	116	tod	4	t--	3394	ugh	117	unt	75	uto	16
tex	88	tog	15	uab	4	ugi	1	unu	3	utp	54
te-	281	tol	17	uag	45	ugm	1	unw	1	uts	3
tfo	3	tom	21	uai	1	ugs	2	un-	14	utu	10
tfu	1	ton	36	ual	77	ugu	1	uok	3	ut-	337
tf-	3	top	30	uan	2	ug-	4	uot	4	ux-	1
tge	1	tor	245	uar	9	uhr	1	uou	5	uy-	2
tg-	1	tos	1	uas	1	uic	2	upd	3	uzl	2
tha	534	tot	14	uat	51	uid	2	upe	33	uzy	5
thd	3	tou	14	ubc	8	uih	7	upg	1	u--	29
the	3966	tow	5	ube	257	uil	33	upi	7	vai	38
thf	1	to-	1002	ubf	1	uin	1	upl	15	val	101
thi	429	tpe	1	ubg	5	uip	3	upo	39	van	78
thm	161	tpi	22	ubi	5	uir	31	upr	1	var	104
tho	203	tpu	53	ubj	11	uis	18	ups	8	vat	10
thr	77	tra	248	ubl	11	uit	33	upt	4	vax	6
ths	7	tre	85	ubm	2	uiv	4	upw	1	vea	1
thu	18	tri	213	ubp	4	ula	155	upy	1	vec	67
th-	443	tro	97	ubr	4	uld	109	up-	74	ved	54
tia	52	tru	217	ubs	26	ule	60	ura	64	vef	1
tib	3	try	12	ubt	2	ulg	1	urb	1	veh	1
tic	321	tr-	8	ub-	3	uln	1	urc	17	vel	107
tie	51	tsc	1	uca	2	ulo	1	ure	532	vem	9
tif	47	tse	7	uce	53	ult	171	urg	1	ven	73
tig	14	tsi	1	uch	102	ulu	1	uri	24	ver	381
tij	1	tst	3	uci	4	uly	6	urn	72	ves	51
til	21	tsu	1	uck	4	ul-	38	uro	3	vex	2
tim	133	tsy	2	ucl	2	uma	24	urp	7	vey	3
tin	250	ts-	512	ucp	1	umb	120	urs	48	ve-	318
tio	1517	tua	28	uct	246	ume	88	urt	22	via	8
tip	48	tub	2	uda	8	umi	8	urv	25	vic	15
tir	17	tuc	1	ude	29	umn	4	ur-	97	vid	86
tis	61	tud	25	udg	4	umo	1	usa	8	vie	14
tit	6	tue	1	udi	9	ump	4	use	357	vig	2
tiv	71	tuf	2	udo	2	ums	3	ush	6	vij	5
tiz	4	tun	10	udy	15	umu	52	usi	73	vin	23
ti-	22	tup	2	ued	5	um-	50	usl	16	vio	17
tk-	1	tur	302	uei	1	una	20	usm	3	vir	49
tle	28	tus	2	uen	63	unb	1	uso	3	vis	43
tli	4	tu-	3	uer	6	unc	108	usp	1	vit	16
tly	82	tv-	1	ues	93	und	125	ust	63	viz	7
tl-	1	twa	21	uez	7	une	10	usu	3	vi-	1
tma	17	twe	67	ue-	111	unf	11	us-	140	vli	4
tme	6	twi	4	ufa	1	uni	167	uta	25	vls	9
tmo	3	two	141	ufe	12	unk	5	utc	20	voi	5
tm-	5	tyl	5	ufi	27	unl	3	ute	249	vok	2
tne	3	typ	93	ufl	2	unm	1	utg	1	vol	44
tn-	1	tyu	1	uf-	5	unp	2	uth	2	von	5

TRIGRAM FREQUENCIES TABLES FOR THESE MSc THESES

tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
vot	1	wro	7	yes	15	zle	2	-cm	1	-gi	47
vou	1	ws-	39	yet	3	zon	23	-co	1202	-gl	12
vri	2	wu-	3	ye-	2	zui	1	-cp	6	-gm	2
vy-	1	w--	217	ygo	1	zuo	3	-cr	37	-go	49
v--	17	xac	11	yie	5	zy-	5	-cu	90	-gr	143
wab	1	xam	91	yin	32	z--	15	-cy	48	-gu	24
wad	1	xb-	1	yi-	1	-ab	152	-c-	36	-g-	47
wai	2	xce	19	yle	4	-ac	201	-da	201	-ha	441
wal	7	xcl	3	yli	1	-ad	146	-de	634	-he	63
wan	10	xec	37	yl-	1	-af	37	-di	429	-hi	77
wap	1	xed	7	ymb	16	-ag	37	-dm	5	-ho	247
war	58	xel	91	ymo	2	-ah	6	-do	144	-hp	1
was	96	xem	4	ymp	4	-ai	26	-dr	29	-hu	15
wat	2	xes	14	ym-	1	-aj	2	-du	41	-hy	129
wav	1	xe-	1	yna	8	-ak	2	-dy	8	-h-	43
way	46	xhi	1	ync	9	-al	486	-d-	7	-ia	1
wea	2	xh-	1	yno	2	-am	53	-ea	128	-ib	12
wed	9	xib	18	ynt	28	-an	1284	-ec	4	-ic	14
weg	2	xic	12	yn-	1	-ao	2	-ed	42	-id	29
wei	3	xim	43	yon	1	-ap	151	-ef	34	-ie	5
wel	38	xin	1	yor	1	-aq	1	-eg	2	-if	118
wen	66	xis	51	you	3	-ar	734	-ei	21	-ig	1
wep	3	xit	30	yo-	1	-as	422	-el	68	-il	7
wer	57	xi-	2	ype	180	-at	150	-em	35	-im	302
wes	6	xn-	4	ypi	18	-au	19	-en	234	-in	1796
wev	42	xpa	1	ypo	5	-av	46	-eo	2	-ip	1
we-	276	xpe	44	yra	1	-aw	7	-ep	1	-ir	2
wha	22	xpl	24	yse	6	-ax	35	-eq	40	-is	1015
whe	127	xpo	5	ysi	44	-ay	13	-er	13	-it	341
whi	181	xpr	23	yst	296	-az	4	-es	42	-iv	2
who	28	xp-	1	ys-	55	-a-	18	-et	66	-i-	49
why	7	xr-	1	yte	51	-ba	206	-eu	1	-ja	9
wic	5	xte	17	yth	1	-be	855	-ev	59	-je	1
wid	21	xti	1	yti	1	-bf	5	-ex	343	-jo	28
wil	91	xtr	30	yur	1	-bg	1	-ey	2	-jr	1
win	84	xts	5	ywa	1	-bi	110	-e-	61	-ju	21
wis	9	xtu	7	ywo	4	-bj	1	-fa	95	-j-	73
wit	403	xt-	73	y--	1675	-bl	39	-fe	63	-ka	10
wle	73	xu-	1	zam	3	-bo	80	-fg	8	-kb	25
wly	3	x--	192	zat	5	-br	49	-fi	372	-ke	37
wmt	1	yat	1	za-	1	-bs	2	-fl	86	-ki	26
wne	1	ya-	1	zed	21	-bu	132	-fo	672	-kn	120
wnl	1	ybe	3	zei	2	-by	288	-fp	6	-ko	6
wn-	84	ybo	1	zen	2	-b-	8	-fr	261	-kr	1
wod	2	yb-	1	zer	15	-ca	561	-ft	1	-ku	3
wof	2	ych	4	zes	7	-cd	3	-fu	145	-k-	7
wor	243	ycl	45	ze-	45	-ce	65	-f-	6	-la	170
wou	24	ydr	18	zif	6	-ch	465	-ga	14	-le	238
wo-	110	yea	6	zin	7	-ci	22	-gb	1	-li	386
wri	64	yed	3	zi-	1	-cl	201	-ge	67	-lo	124

TRIGRAM FREQUENCIES TABLES FOR THESE MSc THESES											
tri	freq	tri	freq	tri	freq	tri	freq	tri	freq	tri	freq
-lt	3	-pl	39	-us	318	--x	3		0		0
-lu	7	-po	243	-ut	2	--y	22		0		0
-l-	17	-pr	1175	-va	200	--z	20		0		0
-ma	494	-ps	6	-ve	247		0		0		0
-mb	10	-pu	35	-vi	52		0		0		0
-mc	12	-py	1	-vl	9		0		0		0
-me	380	-p-	74	-vo	36		0		0		0
-mf	3	-qu	42	-vr	2		0		0		0
-mh	3	-q-	2	-v-	3		0		0		0
-mi	74	-ra	78	-wa	145		0		0		0
-ml	2	-re	985	-we	360		0		0		0
-mo	220	-ri	20	-wh	355		0		0		0
-mp	1	-ro	50	-wi	517		0		0		0
-ms	1	-ru	82	-wm	1		0		0		0
-mu	141	-r-	25	-wo	232		0		0		0
-m-	16	-sa	77	-wr	60		0		0		0
-na	45	-sc	174	-wu	3		0		0		0
-nc	116	-sd	2	-w-	2		0		0		0
-nd	1	-se	464	-xt	2		0		0		0
-ne	216	-sh	168	-x-	1		0		0		0
-ng	2	-si	199	-ye	10		0		0		0
-ni	11	-sk	20	-yi	5		0		0		0
-no	455	-sl	74	-yo	4		0		0		0
-np	7	-sm	29	-y-	3		0		0		0
-nr	13	-so	230	-za	3		0		0		0
-nt	15	-sp	248	-ze	16		0		0		0
-nu	138	-sq	24	-zu	1		0		0		0
-nw	4	-st	457	--a	4034		0		0		0
-nx	3	-su	295	--b	1776		0		0		0
-n-	7	-sw	9	--c	2737		0		0		0
-ob	69	-sy	340	--d	1498		0		0		0
-oc	50	-s-	7	--e	1198		0		0		0
-of	1453	-ta	117	--f	1715		0		0		0
-oi	1	-te	218	--g	406		0		0		0
-ol	2	-tf	3	--h	1016		0		0		0
-om	1	-th	4822	--i	3695		0		0		0
-on	532	-ti	85	--j	133		0		0		0
-op	179	-to	1016	--k	235		0		0		0
-or	253	-tr	251	--l	945		0		0		0
-os	3	-tu	13	--m	1357		0		0		0
-ot	82	-tv	1	--n	1033		0		0		0
-ou	130	-tw	117	--o	2816		0		0		0
-ov	44	-ty	91	--p	2573		0		0		0
-ow	11	-t-	6	--q	44		0		0		0
-o-	6	-ua	1	--r	1240		0		0		0
-pa	594	-uc	1	--s	2817		0		0		0
-pc	9	-uh	1	--t	6740		0		0		0
-pe	171	-ui	7	--u	583		0		0		0
-ph	54	-un	182	--v	549		0		0		0
-pi	172	-up	71	--w	1675		0		0		0

FRAMES

APPENDIX D

8/11/03
20:48:18

skeleton

1

(define frames '(

```
(char-A (parent (value frames))
        (child (value))
        (edges (value)
                (constraint (a longvert)
                           (b longvert)
                           (c medpos)
                           (d medneg)
                           (e medhor)))

        (impl (value)
               (constraint (down a down b)))

        (link (value)
               (constraint (mid a left e)
                           (mid b right e)))

        (join (value)
               (constraint (upper a left c)
                           (upper b right d)
                           (right c left d)))

        (trip (value)
               (constraint))

        (cros (value)
               (constraint)))
```

```
(char-B (parent (value frames))
        (child (value))
        (edges (value)
                (constraint (a longvert)
                           (b medhor)
                           (c medhor)
                           (d medhor)
                           (e smallvert)
                           (f smallneg)
                           (g smallvert)
                           (h smallpos)))

        (impl (value)
               (constraint))

        (link (value)
               (constraint (mid a left c)))

        (join (value)
               (constraint (upper a left b)
                           (right b upper e)
                           (down a left d)
                           (right d left h)
                           (right h down g)
                           (upper g right f)))

        (trip (value)
               (constraint (right c down e left f)))

        (cros (value)
               (constraint)))
```

```
(char-C (parent (value frames))
        (child (value))
        (edges (value)
                (constraint (a longvert)
                           (d medhor)
                           (e medhor)
                           (b smallpos)
                           (c smallneg)))

        (impl (value)
               (constraint (right d right e)))

        (link (value)
               (constraint))

        (join (value)
               (constraint (upper a left b)
                           (right b left d)
                           (down a left c)
                           (left e right c)))

        (trip (value)
               (constraint))

        (cros (value)
               (constraint)))
```

```
(char-D (parent (value frames))
        (child (value))
        (edges (value)
                (constraint (a longvert)
                           (b medpos)
                           (c medpos)
                           (d smallneg)
                           (e smallpos)
                           (f medhor)))

        (impl (value)
               (constraint))

        (link (value)
               (constraint))

        (join (value)
               (constraint (upper a left b)
                           (right b left d)
                           (right d upper f)
                           (down f right e)
                           (left e right c)
                           (left c down a)))

        (trip (value)
               (constraint))

        (cros (value)
               (constraint)))
```

```

(char-E      (parent (value frames))
              (child  (value))
              (edges  (value)
                      (constraint (a longvert)
                                   (b medhor)
                                   (c medhor)
                                   (d medhor)))

              (impl   (value)
                      (constraint (right b right d)))

              (link    (value)
                      (constraint (left c mid a)))

              (join     (value)
                      (constraint (left b upper a)
                                   (down a left d)))

              (trip     (value)
                      (constraint))

              (cros     (value)
                      (constraint)))

```

```

(char-F      (parent (value frames))
              (child  (value))
              (edges  (value)
                      (constraint (a longvert)
                                   (b medhor)
                                   (c medhor)))

              (impl   (value)
                      (constraint))

              (link    (value)
                      (constraint (left c mid a)))

              (join     (value)
                      (constraint (left b upper a)))

              (trip     (value)
                      (constraint))

              (cros     (value)
                      (constraint)))

```

```

(char-G      (parent (value frames))
              (child  (value))
              (edges  (value)
                      (constraint (a smallneg)
                                   (b longhor)
                                   (c smallpos)
                                   (d longvert)

```

```

                      (e smallneg)
                      (f longhor)
                      (g medvert)
                      (h medhor)))

              (impl   (value)
                      (constraint))

              (link    (value)
                      (constraint))

              (join     (value)
                      (constraint (left a right b)
                                   (left b right c)
                                   (left c upper d)
                                   (down d left e)
                                   (right e left f)
                                   (right f down g)
                                   (upper g right h)))

              (trip     (value)
                      (constraint))

              (cros     (value)
                      (constraint)))

```

```

(char-H      (parent (value frames))
              (child  (value))
              (edges  (value)
                      (constraint (a longvert)
                                   (b longvert)
                                   (c medhor)))

              (impl   (value)
                      (constraint (upper a upper b)
                                   (down a down b)))

              (link    (value)
                      (constraint (mid a left c)
                                   (mid b right c)))

              (join     (value)
                      (constraint))

              (trip     (value)
                      (constraint))

              (cros     (value)
                      (constraint)))

```

```

(char-I      (parent (value frames))
              (child  (value))
              (edges  (value)
                      (constraint (a longvert)
                                   (b smallhor)
                                   (c smallhor)))

```

87/11/04
20:49:18

skeleton

3

```
(impl (value)
      (constraint))

(link (value)
      (constraint (mid b upper a)
                  (mid c down a)))

(join (value)
      (constraint))

(trip (value)
      (constraint))

(cros (value)
      (constraint)))

(char-J (parent (value frames))
        (child (value))
        (edges (value)
                (constraint (a longvert)
                            (b smallpos)
                            (c medhor)
                            (d smallvert))))

(impl (value)
      (constraint))

(link (value)
      (constraint))

(join (value)
      (constraint (down a right b)
                  (left b right c)
                  (left c down d)))

(trip (value)
      (constraint))

(cros (value)
      (constraint)))

(char-K (parent (value frames))
        (child (value))
        (edges (value)
                (constraint (a longvert)
                            (b medpos)
                            (c medneg))))

(impl (value)
      (constraint (upper a right b)
                  (right b right c)
                  (down a right c)))

(link (value)
      (constraint))
```

```
(join (value)
      (constraint))

(trip (value)
      (constraint (mid a left b left c)))

(cros (value)
      (constraint)))

(char-L (parent (value frames))
        (child (value))
        (edges (value)
                (constraint (a longvert)
                            (b smallhor))))

(impl (value)
      (constraint))

(link (value)
      (constraint))

(join (value)
      (constraint (down a left b)))

(trip (value)
      (constraint))

(cros (value)
      (constraint)))

(char-M (parent (value frames))
        (child (value))
        (edges (value)
                (constraint (a longvert)
                            (b medneg)
                            (c medpos)
                            (d longvert))))

(impl (value)
      (constraint (down a down d)
                  (upper a upper d)))

(link (value)
      (constraint))

(join (value)
      (constraint (upper a left b)
                  (right b left c)
                  (right c upper d)))

(trip (value)
      (constraint))

(cros (value)
      (constraint)))
```

```
(char-N
  (parent (value frames))
  (child (value))
  (edges (value)
    (constraint
      (a longvert)
      (b longneg)
      (c longvert)))

  (impl (value)
    (constraint
      (down a down c)
      (upper a upper c)))

  (link (value)
    (constraint))

  (join (value)
    (constraint
      (upper a left b)
      (right b down c)))

  (trip (value)
    (constraint))

  (cros (value)
    (constraint)))
```

```
(char-O
  (parent (value frames))
  (child (value))
  (edges (value)
    (constraint
      (a longvert)
      (b longvert)
      (c medhor)
      (d medhor)
      (e smallpos)
      (f smallneg)
      (g smallneg)
      (h smallpos)))

  (impl (value)
    (constraint))

  (link (value)
    (constraint))

  (join (value)
    (constraint
      (upper a left e)
      (right e left c)
      (right c left f)
      (right f upper b)
      (down b right h)
      (left h right d)
      (left d right g)
      (left g down a)))

  (trip (value)
    (constraint)))
```

```
(cros (value)
  (constraint)))
```

```
(char-P
  (parent (value frames))
  (child (value))
  (edges (value)
    (constraint
      (a longvert)
      (b medhor)
      (c medhor)
      (d smallneg)
      (e smallvert)
      (f smallpos)))

  (impl (value)
    (constraint))

  (link (value)
    (constraint
      (mid a left b)))

  (join (value)
    (constraint
      (upper a left c)
      (right d upper e)
      (right c left d)
      (down e right f)
      (left f right b)))

  (trip (value)
    (constraint))

  (cros (value)
    (constraint)))
```

```
(char-Q
  (parent (value frames))
  (child (value))
  (edges (value)
    (constraint
      (a longvert)
      (b longvert)
      (c medhor)
      (d medhor)
      (e smallpos)
      (f smallneg)
      (g smallneg)
      (h smallpos)
      (i smallneg)))

  (impl (value)
    (constraint))

  (link (value)
    (constraint
      (right d mid i)
      (down c mid i)))

  (join (value)
    (constraint
      (upper a left e)
      (right e left c)
      (right c left f)
      (right f upper b)))
```

8/7/11/04
20:49:18

skeleton

5

```

      (down b right h)
      (left h right d)
      (left d right g)
      (left g down a)))

(char-R (parent (value frames))
        (child (value))
        (edges (value)
                 (constraint
                  (a longvert)
                  (b medhor)
                  (c medhor)
                  (d medneg)
                  (e smallvert)
                  (f medpos)
                  (g medneg)
                  (h smallvert))))

      (impl (value)
             (constraint (down a down h)))

      (link (value)
             (constraint (mid a left b)))

      (join (value)
             (constraint
              (upper a left c)
              (right c left d)
              (right d upper e)
              (down e right f)
              (right g upper h)))

      (trip (value)
             (constraint (right b left f left g)))

      (cros (value)
             (constraint)))

(char-S (parent (value frames))
        (child (value))
        (edges (value)
                 (constraint
                  (a smallneg)
                  (b longhor)
                  (c smallpos)
                  (d smallvert)
                  (e smallneg)
                  (f longhor)
                  (g smallneg)
                  (h smallvert)
                  (i smallpos)
                  (j longhor)
                  (k smallneg))))

```

```

      (impl (value)
             (constraint))

      (link (value)
             (constraint))

      (join (value)
             (constraint
              (left a right b)
              (left b right c)
              (left c upper d)
              (down d left e)
              (left f right e)
              (right f left g)
              (right g upper h)
              (down a right i)
              (left i right j)
              (left j right k)))

      (trip (value)
             (constraint))

      (cros (value)
             (constraint)))

```

```

(char-T (parent (value frames))
        (child (value))
        (edges (value)
                 (constraint
                  (a longhor)
                  (b longvert))))

      (impl (value)
             (constraint))

      (link (value)
             (constraint (mid a upper b)))

      (join (value)
             (constraint))

      (trip (value)
             (constraint))

      (cros (value)
             (constraint)))

```

```

(char-U (parent (value frames))
        (child (value))
        (edges (value)
                 (constraint
                  (a longvert)
                  (e longvert)
                  (c medhor)
                  (b smallneg)
                  (d smallpos))))

      (impl (value)
             (constraint)))

```

```

                (constraint (upper a upper e)))

(link (value)
  (constraint))

(join (value)
  (constraint (down a left b)
    (right b left c)
    (right c left d)
    (right d down e)))

(trip (value)
  (constraint))

(cros (value)
  (constraint)))

(char-V (parent (value frames))
  (child (value))
  (edges (value)
    (constraint (a longvert)
      (d longvert)
      (b medneg)
      (c medpos))))

(impl (value)
  (constraint (upper a upper d)))

(link (value)
  (constraint))

(join (value)
  (constraint (down a left b)
    (right b left c)
    (right c down d)))

(trip (value)
  (constraint))

(cros (value)
  (constraint)))

(char-W (parent (value frames))
  (child (value))
  (edges (value)
    (constraint (a longvert)
      (b longvert)
      (c medneg)
      (d medpos))))

(impl (value)
  (constraint (down a down b)
    (upper a upper b)))

(link (value)
  (constraint))

```

```

(join (value)
  (constraint (down a left d)
    (right d left c)
    (right c down b)))

(trip (value)
  (constraint))

(cros (value)
  (constraint)))

(char-X (parent (value frames))
  (child (value))
  (edges (value)
    (constraint (a smallvert)
      (b smallvert)
      (c smallvert)
      (d smallvert)
      (e longpos)
      (f longneg))))

(impl (value)
  (constraint (left f right e)
    (left e right f)
    (left e left f)
    (right e right f)))

(link (value)
  (constraint))

(join (value)
  (constraint (down a left f)
    (down b right e)
    (upper c left e)
    (upper d right f)))

(trip (value)
  (constraint))

(cros (value)
  (constraint (e f)))

(char-Y (parent (value frames))
  (child (value))
  (edges (value)
    (constraint (a smallvert)
      (b longneg)
      (c longpos)
      (d smallvert)
      (e longvert))))

(impl (value)
  (constraint (left b right c)))

(link (value)

```


87/11/04
20:49:18

skeleton

7

```

                (constraint))

(join (value)
      (constraint (down a left b)
                  (down d right c)))

(trip (value)
      (constraint (right b left c upper e)))

(cros (value)
      (constraint))

(char-z (parent (value frames))
        (child (value))
        (edges (value)
                (constraint (a longhor)
                            (b smallvert)
                            (c longpos)
                            (d smallvert)
                            (e longhor))))

(impl (value)
      (constraint (left a left e)
                  (right a right e)))

(link (value)
      (constraint))

(join (value)
      (constraint (right a upper b)
                  (down b right c)
                  (left c upper d)
                  (down d left e)))

(trip (value)
      (constraint))

(cros (value)
      (constraint))

(cros (value)
      (constraint))

(char-a (child (value))
        (parent (value frames))
        (edges (value)
                (constraint (a longvert)
                            (b smallneg)
                            (c medhor)
                            (d smallpos)
                            (e medvert)
                            (f smallneg)
                            (g medhor)
                            (h smallpos))))

(impl (value)
      (constraint))

```

```

(link (value)
      (constraint (right b mid a)))

(join (value)
      (constraint (left b right c)
                  (left c right d)
                  (left d upper e)
                  (down e left f)
                  (right f left g)
                  (right g left h)
                  (right h down a)))

(trip (value)
      (constraint))

(cross (value)
        (constraint))

(point (value)
        (constraint))

(char-b (child (value))
        (parent (value frames))
        (edges (value)
                (constraint (a longvert)
                            (b smallneg)
                            (c medhor)
                            (d smallpos)
                            (e medvert)
                            (f smallneg)
                            (g medhor)
                            (h smallpos))))

(impl (value)
      (constraint))

(link (value)
      (constraint (mid a left b)
                  (mid a left h)))

(join (value)
      (constraint (right b left c)
                  (right c left d)
                  (right d down e)
                  (upper e right f)
                  (left f right h)
                  (left g right h)))

(trip (value)
      (constraint))

(cross (value)
        (constraint))

(point (value)
        (constraint))

(char-c (child (value))
        (parent (value frames))
        (edges (value)
                (constraint (a longhor)
                            (b smallpos)
                            (c medvert)
                            (d smallneg)
                            (e longhor))))

(impl (value)
      (constraint (right a right e)))

(link (value)
      (constraint))

(join (value)
      (constraint (left a right b)
                  (left b upper c)
                  (down c left d)))

```

```

(trip      (value)      (right d left e)))
(constraint))
(cross     (value)
(constraint))
(point     (value)
(constraint)))

(char-d    (child      (value))
(parent    (value      frames))
(edges     (value)
(constraint (a longvert)
(b smallneg)
(c medhor)
(d smallpos)
(e medvert)
(f smallneg)
(g medhor)
(h smallpos)))

(impl      (value)
(constraint))
(link      (value)
(constraint (right b mid a)
(right h mid a)))

(join      (value)
(constraint (left b right c)
(left c right d)
(left d upper e)
(down e left f)
(right f left g)
(right g left h)))

(trip      (value)
(constraint))
(cross     (value)
(constraint))
(point     (value)
(constraint)))

(char-e    (child      (value))
(parent    (value      frames))
(edges     (value)
(constraint (a medhor)
(b longhor)
(c smallpos)
(d smallneg)
(e medvert)
(f smallneg)
(g medhor)))

(impl      (value)
(constraint))
(link      (value)
(constraint))
(join      (value)
(constraint (right c left a)
(right a left d)
(right d right b)
(down e left f)
(right f left g)))

(trip      (value)
(constraint (left c upper e left b)))
(cross     (value)
(constraint))
(point     (value)
(constraint)))

```

```

(char-f    (child      (value))
(parent    (value      frames))
(edges     (value)
(constraint (a medhor)
(b longvert)
(c medhor)))

(impl      (value)
(constraint))
(link      (value)
(constraint))
(join      (value)
(constraint (left a upper b)))

(trip      (value)
(constraint))
(cross     (value)
(constraint (c b)))
(point     (value)
(constraint)))

(char-g    (child      (value))
(parent    (value      frames))
(edges     (value)
(constraint (a medhor)
(b smallpos)
(c longvert)
(d smallneg)
(e medhor)
(f smallpos)
(g medvert)
(h smallneg)
(i medhor)
(j smallpos)))

(impl      (value)
(constraint))
(link      (value)
(constraint (right j mid c)))
(join      (value)
(constraint (right a left b)
(right b down c)
(upper c right d)
(left d right e)
(left e right f)
(left f upper g)
(down g left h)
(right h left i)
(right i left j)))

(trip      (value)
(constraint))
(cross     (value)
(constraint))
(point     (value)
(constraint)))

(char-h    (child      (value))
(parent    (value      frames))
(edges     (value)
(constraint (a longvert)
(b smallpos)
(c smallhor)
(d smallneg)
(e medvert)))

(impl      (value)
(constraint (down a down e)))

```

```

(link      (value)
  (constraint (mid a left b)))
(join      (value)
  (constraint (right b left c)
              (right c left d)
              (right d upper e)))

(trip      (value)
  (constraint))
(cross     (value)
  (constraint))
(point     (value)
  (constraint))

(char-i    (child      (value))
  (parent    (value frames))
  (edges     (value)
    (constraint (a longvert)))
  (impl      (value)
    (constraint))
  (link      (value)
    (constraint))
  (join      (value)
    (constraint))
  (trip      (value)
    (constraint))
  (cross     (value)
    (constraint))
  (point     (value)
    (constraint (mid upper))))

(char-j    (child      (value))
  (parent    (value frames))
  (edges     (value)
    (constraint (a smallneg)
                (b smallhor)
                (c smallpos)
                (d longvert)))
  (impl      (value)
    (constraint))
  (link      (value)
    (constraint))
  (join      (value)
    (constraint (right a left b)
                (right b left c)
                (right c down d)))
  (trip      (value)
    (constraint))
  (cross     (value)
    (constraint))
  (point     (value)
    (constraint (right upper))))

(char-k    (child      (value))
  (parent    (value frames))
  (edges     (value)
    (constraint (a long vert)
                (b medneg)
                (c medpos)))
  (impl      (value)
    (constraint (down a right b)))
  (link      (value)
    (constraint))
  (join      (value)
    (constraint))

```

```

(trip      (value)
  (constraint (mid a left b left c)))
(cross     (value)
  (constraint))
(point     (value)
  (constraint))

(char-l    (child      (value))
  (parent    (value frames))
  (edges     (value)
    (constraint (a long vert)))
  (impl      (value)
    (constraint))
  (link      (value)
    (constraint))
  (join      (value)
    (constraint))
  (trip      (value)
    (constraint))
  (cross     (value)
    (constraint))
  (point     (value)
    (constraint))

(char-m    (child      (value))
  (parent    (value frames))
  (edges     (value)
    (constraint (a longvert)
                (b longvert)
                (c longvert)
                (d medhor)
                (e medhor)))
  (impl      (value)
    (constraint (down a down b)
                (down b down c)))
  (link      (value)
    (constraint))
  (join      (value)
    (constraint (upper a left d)
                (right e upper c)))
  (trip      (value)
    (constraint (right d upper b left e)))
  (cross     (value)
    (constraint))
  (point     (value)
    (constraint))

(char-n    (child      (value))
  (parent    (value frames))
  (edges     (value)
    (constraint (a longneg)
                (b medhor)
                (c longneg)))
  (impl      (value)
    (constraint (down a down c)))
  (link      (value)
    (constraint))
  (join      (value)
    (constraint (upper a left b)
                (right b upper c)))
  (trip      (value)
    (constraint))
  (cross     (value)
    (constraint))

```

```

(point      (value)
 (constraint))

(char-o      (child      (value))
 (parent     (value     frames))
 (edges      (value)
 (constraint (a longhor)
              (b longpos)
              (c longvert)
              (d longneg)
              (e longhor)
              (f longpos)
              (g longvert)
              (h longneg)))

(impl       (value)
 (constraint))

(link       (value)
 (constraint))

(join       (value)
 (constraint (left a right b)
              (left b upper c)
              (down c left d)
              (right d left e)
              (right left f)
              (right f down g)
              (upper g right h)
              (left h right a)))

(trip       (value)
 (constraint))

(cross      (value)
 (constraint))

(point      (value)
 (constraint))

(char-p      (child      (value))
 (parent     (value     frames))
 (edges      (value)
 (constraint (a longvert)
              (b smallpos)
              (c medhor)
              (d smallneg)
              (e medvert)
              (f smallpos)
              (g medhor)
              (h smallneg)))

(impl       (value)
 (constraint))

(link       (value)
 (constraint (mid a right b)
              (mid a left h)))

(join       (value)
 (constraint (right h left g)
              (right g left f)
              (right f down e)
              (upper e right d)
              (left d right c)
              (left c right b)))

(trip       (value)
 (constraint))

(cross      (value)
 (constraint))

(point      (value)
 (constraint))

```

```

(char-q      (child      (value))
 (parent     (value     frames))
 (edges      (value)
 (constraint (a longvert)
              (b smallneg)
              (c medhor)
              (d smallpos)
              (e medvert)
              (f smallneg)
              (g medhor)
              (h smallpos)))

(impl       (value)
 (constraint))

(link       (value)
 (constraint (right b mid a)
              (right h mid a)))

(join       (value)
 (constraint (left b right c)
              (left c right d)
              (left d upper e)
              (down e left f)
              (right f left g)
              (right g left h)))

(trip       (value)
 (constraint))

(cross      (value)
 (constraint))

(point      (value)
 (constraint))

(char-r      (child      (value))
 (parent     (value     frames))
 (edges      (value)
 (constraint (a longvert)
              (b medhor)))

(impl       (value)
 (constraint))

(link       (value)
 (constraint))

(join       (value)
 (constraint (upper a left b)))

(trip       (value)
 (constraint))

(cross      (value)
 (constraint))

(point      (value)
 (constraint))

(char-s      (child      (value))
 (parent     (value     frames))
 (edges      (value)
 (constraint (a longhor)
              (b smallvert)
              (c longhor)
              (d smallvert)
              (e longhor)))

(impl       (value)
 (constraint (right a upper d)
              (down b left e)))

(link       (value)
 (constraint))

(join       (value)
 (constraint (left a upper b)
              (down b left e)))

```

87/11/04
20:49:18

skeleton

11

```

                (right c upper d)
                (down d right e)))
(trip      (value)
            (constraint))
(cross     (value)
            (constraint))
(point     (value)
            (constraint)))

(char-t     (child      (value))
            (parent     (value frames))
            (edges      (value)
                        (constraint (a longvert)
                                   (b medhor)
                                   (c smallhor))))

(impl      (value)
            (constraint))
(link      (value)
            (constraint))
(join      (value)
            (constraint (down a left c)))
(trip      (value)
            (constraint))
(cross     (value)
            (constraint (a b)))
(point     (value)
            (constraint)))

(char-u     (child      (value))
            (parent     (value frames))
            (edges      (value)
                        (constraint (a longvert)
                                   (b smallneg)
                                   (c medhor)
                                   (d smallpos)
                                   (e longvert))))

(impl      (value)
            (constraint (upper a upper e)))
(link      (value)
            (constraint))
(join      (value)
            (constraint (down a left b)
                        (right b left c)
                        (right c left d)
                        (right d down e)))

(trip      (value)
            (constraint))
(cross     (value)
            (constraint))
(point     (value)
            (constraint)))

(char-v     (child      (value))
            (parent     (value frames))
            (edges      (value)
                        (constraint (a longhor)
                                   (b longneg)
                                   (c longpos)
                                   (d longvert))))

(impl      (value)
            (constraint (upper a upper d)))
(link      (value)
            (constraint))
(join      (value)
            (value))

```

```

                (constraint (down a left b)
                            (right b left c)
                            (right c down d)))
(trip      (value)
            (constraint))
(cross     (value)
            (constraint))
(point     (value)
            (constraint)))

(char-w     (child      (value))
            (parent     (value frames))
            (edges      (value)
                        (constraint (a longvert)
                                   (b smallhor)
                                   (c smallpos)
                                   (d smallneg)
                                   (e smallhor)
                                   (f longvert))))

(impl      (value)
            (constraint (upper a upper f)))
(link      (value)
            (constraint))
(join      (value)
            (constraint (down a left b)
                        (right b left c)
                        (right c left d)
                        (right d left e)
                        (right e down f)))

(trip      (value)
            (constraint))
(cross     (value)
            (constraint))
(point     (value)
            (constraint)))

(char-x     (child      (value))
            (parent     (value frames))
            (edges      (value)
                        (constraint (a longpos)
                                   (b longneg))))

(impl      (value)
            (constraint (left a right b)
                        (right b right a)))
(link      (value)
            (constraint))
(join      (value)
            (constraint))
(trip      (value)
            (constraint))
(cross     (value)
            (constraint (a b)))
(point     (value)
            (constraint)))

(char-y     (child      (value))
            (parent     (value frames))
            (edges      (value)
                        (constraint (a longvert)
                                   (b smallneg)
                                   (c medhor)
                                   (d smallpos)
                                   (e longvert)
                                   (f smallpos))))

```

```

(g medhor)))
(impl (value)
      (constraint (upper a upper e)))
(link (value)
      (constraint (right d mid e)))
(join (value)
      (constraint (down a left b)
                  (right b left c)
                  (right c left d)
                  (down e right f)
                  (left f right g)))

(trip (value)
      (constraint))
(cross (value)
        (constraint))
(point (value)
        (constraint))

(char-z (child (value))
        (parent (value frames))
        (edges (value)
                (constraint (a longhor)
                            (b longpos)
                            (c longhor))))

(impl (value)
      (constraint (left a left c)
                  (right a right c)))
(link (value)
      (constraint))
(join (value)
      (constraint (right a right b)
                  (left b left c)))
(trip (value)
      (constraint))
(cross (value)
        (constraint))
(point (value)
        (constraint))

(char-0 (child (value))
        (parent (value frames))
        (edges (value)
                (constraint (a longvert)
                            (b smallpos)
                            (c smallhor)
                            (d smallneg)
                            (e longvert)
                            (f smallpos)
                            (g smallhor)
                            (h smallneg))))

(impl (value)
      (constraint))
(link (value)
      (constraint))
(join (value)
      (constraint (upper a left b)
                  (right b left c)
                  (right c left d)
                  (right d upper e)
                  (down e right f)
                  (left f right g)
                  (left g right h)
                  (left h down a)))

(trip (value)

```

```

(constraint))
(cross (value)
        (constraint))
(point (value)
        (constraint))

(char-1 (child (value))
        (parent (value frames))
        (edges (value)
                (constraint (a smallpos)
                            (b longvert)
                            (c smallhor))))

(impl (value)
      (constraint))
(link (value)
      (constraint (mid c down b)))
(join (value)
      (constraint (right a upper b)))
(trip (value)
      (constraint))
(cross (value)
        (constraint))
(point (value)
        (constraint))

(char-2 (child (value))
        (parent (value frames))
        (edges (value)
                (constraint (b longhor)
                            (c medvert)
                            (d longhor)
                            (e smallpos)
                            (f medvert)
                            (g longhor))))

(impl (value)
      (constraint))
(link (value)
      (constraint))
(join (value)
      (constraint (right b upper c)
                  (down c right d)
                  (left d right e)
                  (left e upper f)
                  (down f left g)))

(trip (value)
      (constraint))
(cross (value)
        (constraint))
(point (value)
        (constraint))

(char-3 (child (value))
        (parent (value frames))
        (edges (value)
                (constraint (a longhor)
                            (b longvert)
                            (c longhor)
                            (d longvert)
                            (e longhor))))

(impl (value)
      (constraint (left a left e)))
(link (value)
      (constraint))
(join (value)

```

```

(char-3
  (trip      (value)      (constraint (right a upper b)
                                   (down d right e)))
  (cross     (value)      (constraint (right c down b upper d)))
  (point     (value)      (constraint)))

(char-4
  (child     (value))
  (parent    (value      frames))
  (edges     (value)
             (constraint (a medpos)
                          (b medhor)
                          (c longvert)))
  (impl      (value)
             (constraint))
  (link      (value)
             (constraint))
  (join      (value)
             (constraint (left a left b)
                          (right a upper c)))
  (trip      (value)
             (constraint))
  (cross     (value)
             (constraint (b c)))
  (point     (value)
             (constraint)))

(char-5
  (child     (value))
  (parent    (value      frames))
  (edges     (value)
             (constraint (a longhor)
                          (b medvert)
                          (c longhor)
                          (d medvert)
                          (e longhor)))
  (impl      (value)
             (constraint (left e left c)))
  (link      (value)
             (constraint))
  (join      (value)
             (constraint (left a upper b)
                          (down b left c)
                          (right c upper d)
                          (down d right e)))
  (trip      (value)
             (constraint))
  (cross     (value)
             (constraint))
  (point     (value)
             (constraint)))

(char-6
  (child     (value))
  (parent    (value      frames))
  (edges     (value)
             (constraint (a medhor)
                          (b longvert)
                          (c medhor)
                          (d medvert)
                          (e medhor)))
  (impl      (value)
             (constraint))
  (link      (value)
             (constraint)))

```

```

(char-7
  (child      (value))
  (parent     (value frames))
  (edges      (value)
              (constraint (a longhor)
                           (b medvert)
                           (c longpos)
                           (d medhor)))
  (impl       (value)
              (constraint))
  (link       (value)
              (constraint))
  (join       (value)
              (constraint (right a upper b)
                           (down b right c)
                           (left c upper d)))
  (trip       (value)
              (constraint))
  (cross      (value)
              (constraint))
  (point      (value)
              (constraint)))

(char-8
  (child      (value))
  (parent     (value frames))
  (edges      (value)
              (constraint (a medvert)
                           (b longhor)
                           (c medvert)
                           (d longhor)
                           (e longvert)
                           (f longhor)
                           (g longvert)))
  (impl       (value)
              (constraint))
  (link       (value)
              (constraint))
  (join       (value)
              (constraint (upper a left b)
                           (right b upper c)
                           (down e left f)
                           (right f down g)))
  (trip       (value)
              (constraint (down a left d upper e)
                           (down c right d upper g)))
  (cross      (value)
              (constraint))
  (point      (value)
              (constraint)))

(char-9
  (child      (value))
  (parent     (value frames))

```

```

                (constraint      (a medhor)
                                (b longvert)
                                (c medhor)))

    (impl      (value)
                (constraint      (right a right c)))
    (link      (value)
                (constraint))
    (join      (value)
                (constraint      (left a upper b)
                                (left c down b)))
    (trip      (value)
                (constraint))
    (cross     (value)
                (constraint))
    (point     (value)
                (constraint))

(char-""]"    (child      (value))
              (parent     (value      frames))
              (edges      (value)
                          (constraint      (a medhor)
                                            (b longvert)
                                            (c medhor)))

    (impl      (value)
                (constraint      (left a left c)))
    (link      (value)
                (constraint))
    (join      (value)
                (constraint      (right a upper b)
                                (right c down b)))
    (trip      (value)
                (constraint))
    (cross     (value)
                (constraint))
    (point     (value)
                (constraint))

(char-"+"    (child      (value))
              (parent     (value      frames))
              (edges      (value)
                          (constraint      (a longvert)
                                            (b longhor)))

    (impl      (value)
                (constraint))
    (link      (value)
                (constraint))
    (join      (value)
                (constraint))
    (trip      (value)
                (constraint))
    (cross     (value)
                (constraint      (a b)))
    (point     (value)
                (constraint))

(char-"--"   (child      (value))
              (parent     (value      frames))
              (edges      (value)
                          (constraint      (a longhor)))

    (impl      (value)
                (constraint))
    (link      (value)
                (constraint))
    (join      (value)
                (constraint))

```



```

(trip      (constraint))
           (value)
(cross     (constraint))
           (value)
(point     (constraint))
           (value)
           (constraint)))

(char-<"   (child      (value))
           (parent     (value frames))
           (edges      (value)
                     (constraint (a longpos)
                               (b longneg))))

(impl      (value)
           (constraint (right a right b)))
(link      (value)
           (constraint))
(join      (value)
           (constraint (left a left b)))
(trip      (value)
           (constraint))
(cross     (value)
           (constraint))
(point     (value)
           (constraint)))

(char->"   (child      (value))
           (parent     (value frames))
           (edges      (value)
                     (constraint (a longneg)
                               (b longpos))))

(impl      (value)
           (constraint (left a left b)))
(link      (value)
           (constraint))
(join      (value)
           (constraint (right a right b)))
(trip      (value)
           (constraint))
(cross     (value)
           (constraint))
(point     (value)
           (constraint)))

(char="    (child      (value))
           (parent     (value frames))
           (edges      (value)
                     (constraint (a longhor)
                               (b longhor))))

(impl      (value)
           (constraint (left a left b)
                     (right a right b)))
(link      (value)
           (constraint))
(join      (value)
           (constraint))
(trip      (value)
           (constraint))
(cross     (value)
           (constraint))
(point     (value)
           (constraint)))

(char-"    (child      (value))

```

```

           (parent     (value frames))
           (edges      (value)
                     (constraint (a smallpos)))
(impl      (value)
           (constraint))
(link      (value)
           (constraint))
(join      (value)
           (constraint))
(trip      (value)
           (constraint))
(cross     (value)
           (constraint))
(point     (value)
           (constraint (mid upper)))

(char="!"  (child      (value))
           (parent     (value frames))
           (edges      (value)
                     (constraint (a longvert)))
(impl      (value)
           (constraint))
(link      (value)
           (constraint))
(join      (value)
           (constraint))
(trip      (value)
           (constraint))
(cross     (value)
           (constraint))
(point     (value)
           (constraint (mid down)))

(char="?"  (child      (value))
           (parent     (value frames))
           (edges      (value)
                     (constraint (a longhor)
                               (b medvert)
                               (c longpos)
                               (d medvert))))

(impl      (value)
           (constraint))
(link      (value)
           (constraint))
(join      (value)
           (constraint (right a upper b)
                     (down b right c)
                     (left c upperd)))

(trip      (value)
           (constraint))
(cross     (value)
           (constraint))
(point     (value)
           (constraint (mid down)))

(char=";"  (child      (value))
           (parent     (value frames))
           (edges      (value)
                     (constraint (a longpos)))
(impl      (value)
           (constraint))
(link      (value)
           (constraint))
(join      (value)

```

```

      (constraint))
      (value)
      (constraint))
      (value)
      (constraint))
      (value)
      (constraint (mid mid)
        (mid down)))

(char-"," (child      (value))
          (parent    (value frames))
          (edges      (value)
            (constraint))
          (impl       (value)
            (constraint))
          (link       (value)
            (constraint))
          (join       (value)
            (constraint))
          (trip       (value)
            (constraint))
          (cross      (value)
            (constraint))
          (point      (value)
            (constraint (mid down))))

(char-"," (child      (value))
          (parent    (value frames))
          (edges      (value)
            (constraint (a longpos)))
          (impl       (value)
            (constraint))
          (link       (value)
            (constraint))
          (join       (value)
            (constraint))
          (trip       (value)
            (constraint))
          (cross      (value)
            (constraint))
          (point      (value)
            (constraint (mid down))))

(char-": " (child      (value))
          (parent    (value frames))
          (edges      (value)
            (constraint))
          (impl       (value)
            (constraint))
          (link       (value)
            (constraint))
          (join       (value)
            (constraint))
          (trip       (value)
            (constraint))
          (cross      (value)
            (constraint))
          (point      (value)
            (constraint (mid mid)
              (mid down))))

(char-"&" (child      (value))
          (parent    (value frames))
          (edges      (value)

```

```

      (constraint (a smallvert)
        (b smallhor)
        (c smallvert)
        (d medpos)
        (e medvert)
        (f longneg)
        (g smallhor)
        (h medpos)))

      (impl      (value)
        (constraint))
      (link      (value)
        (constraint))
      (join      (value)
        (constraint (down a left f)
          (upper a left b)
          (right b upper c)
          (down c right d)
          (left d upper e)
          (down e left g)
          (right g left h)))

      (trip      (value)
        (constraint))
      (cross     (value)
        (constraint (h f)
          (d f)))
      (point     (value)
        (constraint))

(char-"$ " (child      (value))
          (parent    (value frames))
          (edges      (value)
            (constraint (a medhor)
              (b smallvert)
              (c medhor)
              (d smallvert)
              (e medhor)
              (f longvert)
              (g longvert))))

      (impl      (value)
        (constraint (right a upper d)
          (down b left e)))
      (link      (value)
        (constraint))
      (join      (value)
        (constraint (left a upper b)
          (down b left e)
          (right c upper d)
          (down d right e)))

      (trip      (value)
        (constraint))
      (cross     (value)
        (constraint (a f)
          (a g)
          (c f)
          (c g)
          (e f)
          (e g)))

      (point     (value)
        (constraint))

(char-"*" (child      (value))
          (parent    (value frames))
          (edges      (value)
            (constraint (a longvert)

```

87/11/04
20:49:18

skeleton

17

```

                                (b longpos)
                                (c longhor)
                                (d longneg)))
(impl      (value)
            (constraint))
(link      (value)
            (constraint))
(join      (value)
            (constraint))
(trip      (value)
            (constraint))
(cross     (value)
            (constraint (a b)
                        (a c)
                        (a d)
                        (b c)
                        (b d)
                        (c d)))
(point     (value)
            (constraint))
))
(load)
```


SOURCE CODES

APPENDIX E

87/11/04
00:21:06

down.c

1

```
/*
 * down.c                                27/10/1987
 *
 * down.c takes a nodes executable as an argument and
 * downloads it to node 0 of a zero dimensional hypercube.
 * This node is then scanned for output to the host, which
 * is displayed on the user screen.
 */

#include "stdio.h"

main(argc,argv)
int argc;
char *argv[];
{
    int chan, size, node = 0, type;
    char buf[512], *ptr;

    if (argc != 2) {
        printf("Usage : %s <program name>\n",argv[0]);
        exit(1);
    }
    printf("Downloading %s to ncube array.\n",argv[1]);
    chan = nopen( 0 );
    nload(chan,argv[1],node,buf,512);
    for(;;) {
        type = -1;
        size = nread(chan, buf, 512, &node, &type);
        for(ptr = buf;ptr < buf + size;ptr++)
            putchar(*ptr);
    }
}
```

```

/*
 * boot.c
 *
 * 27/8/1987
 *
 * Boot allocates a pipe and connects one end to the standard
 * output of a 'cat' process whose standard input is connected
 * to the terminal. The standard input of this program is closed.
 * This procedure is needed because it was the only way that
 * I could find to not have the characters typed in by the user
 * echoed directly on the screen. A hypercube with arbitrary
 * dimension is allocated and the handle, which is in fact a
 * file pointer, is connected to the standard input of the master
 * process. Further a lower part of the screen is reserved for
 * input from the user which is sent to a selected hypercube node.
 *
 * Commands:
 *
 * ESC-n-<n>
 *
 *         select node <n> as node to which the user input is
 *         sent to.
 * ESC-r-<filename>
 *
 *         send the contents of a file to the current node.
 * ]
 *
 *         match the brackets and send the message with the
 *         number of missing brackets attached to it.
 *
 * Delft University of Technology
 */

#include "window.h"
#include <power.h>

#define PMODE 0644
#define PACKAGE_SIZE 20000 /* estimate of max image size */
#define ESC '\033'
#define EOF (-1)
#define CTRL_X '\030'
#define NULL 0
#define stdin 0
#define stdout 1
#define stderr 2

char *BOOT_NR = "Boot version 3.0";
char *image_file = "/usr/hahe/images/\0";
char *master = "/usr/hahe/ncube/bin/master";
char *lisp_exec = "/usr/hahe/local/bin/nlisp";

char error[50];
char buf[1024];
int input;
int output;
int dimension;

void read_string();
void initializeNodes();

main(argc,argv)
int argc;
char *argv[];
{
    int chan,priority,state,pid_master,pid_cat,wp;
    long sysp;
    char program[100], response[10];

```

```

/*
 * Connect /bin/cat to the user terminal for this program
 * somehow (?) manages to disable the character echo to the user
 * screen. By connecting this program to the standard
 * output of cat with a pipe we still can read the user input
 * without having the characters echoed on the screen unless we
 * want them to.
 */

state = 0;
priority = 95;

if (n_mkpipe(&wp,&input) < 0) {
    perror("Can not create pipe ( no rights ? )\n");
    exit(1);
}

pid_cat = n_frun(0,95,0,wp,2,"/bin/cat");

/*
 * Initialize a multi display screen
 */

initialize_windows(1,17,24);
message(MIDDLE,BOOT_NR,0);
message(RIGHT,"LispNode 0",0);
message(LEFT,"",0);
select_keypad();

/*
 * If no image file is indicated then take "normal"
 * as an image file. Otherwise the name of the image file
 * is stored in argv[1].
 */

if (argc == 2) {
    strcat( image_file, argv[1] );
} else {
    strcat( image_file, "normal" );
}
strcat( image_file, ".img" );

/*
 * Ask the user to enter the dimension of the hypercube he
 * wants to be using.
 */

w_putstr("Dimension : ",0);
read_string(response, input);

/*
 * Allocate the hypercube. A filedescriptor (channel descriptor),
 * is returned that can be used in nread and nwrite calls.
 */

sprintf(program,"%s %s",master,response);
output = nopen(dimension = atoi(response));

/*
 * Send the Lisp executable object file to all the nodes in the
 * allocated hypercube. Subsequently the image data is sent to
 * node 0.
 */

```


87/11/04
00:19:17

boot.c

2

```
InitializeNodes();

/*
 * Execute a program that dispatches all output from the nodes
 * to the upper part of the screen as well as in a log file
 * called flisp.log. The program is executed with frun and its
 * standard input is connected to the NCube channel descriptor.
 */

sprintf(program,"%s %s",master,response);
pid_master = n_frun(state,priority,output,stdout,stderr,program);
if (pid_master < 0) {
    perror(sprintf(error,"Boot error : can not execute %s\n",program));
    exit(1);
}

/*
 * Now we can start the interaction between the user and the
 * Lisp interpreters running on the nodes. The interaction may
 * be finished by typing CTRL X.
 */

REPloop();

/*
 * When the interaction is terminated, the terminal is reseted and
 * both the /bin/cat and dispatcher program are killed before this
 * program finishes.
 */

printf("Exiting");

reset_terminal();

n_getid(&sysp);
n_psend(pid_master,sysp,0);
n_psend(pid_cat,sysp,0);
}

void InitializeNodes()
{
    int fd, type = 115, nnodes;
    long bytes_read;
    char *image;

    /*
     * Determine the number of nodes that are in the allocated
     * hypercube
     */

    nnodes = power2(dimension);

    /*
     * Before we sent the image to node 0, we first send the
     * program object.
     */

    nloadm( output, lisp_exec, (-1)*nnodes,0,buf, 1024 );

    /*
     * The binary file containing the image is stored in binfile. To

```

```

 * enable transmission of data we need a large buffer. The buffer
 * is allocated with calloc(). When ready this buffer is free-ed
 * again.
 */

if ((image = (char *) calloc(PACKAGE_SIZE,1)) == NULL) {
    printf("BOOT : Cannot allocate buffer\n");
    exit( -1 );
}

if ((fd = open(image_file,0)) == -1) {
    printf("BOOT : cannot open image file %s\n",image_file);
    exit(-1);
}

/*
 * The image is read from the imgfile and is sent to node 0
 * in chunks that are PACKAGE_SIZE bytes big. If one chunk
 * is not enough, another is sent.
 */

do {
    bytes_read = read(fd,image,PACKAGE_SIZE);
    nwrite( output, image, bytes_read, 0, type );
} while( bytes_read == PACKAGE_SIZE );

/*
 * Throw buffer away and close the file
 */

free( image );
close( fd );
}

char next_char(iop)
int iop;
{
    char c;

    return((read(iop,&c,1) == 1) ? ((c == '\015')?' \n':c) : EOF);
}

void read_string(s,iop)
char *s;
int iop;
{
    register char c,*ptr = s;

    while((c = next_char(iop)) != EOF && c != '\n') {
        w_putchar(c,0);
        if (c == '\b') {
            if (ptr > s) ptr--;
        } else
            *ptr++ = c;
    }
    *ptr = '\0';
}

void read_file(iop)
int *iop;
{
    char *ptr,name[50];

    if (*iop == input) {

```

```

        message(LEFT,"Enter filename",0);
        read_string(name,*iop);
        if ((*iop = open(name,0)) < 0) {
            sprintf(error,"\\nerror : (open) %s\\n",name);
            w_putstr(error,0);
            *iop = input;
        }
    }

int match_brackets(buf,s,n)
char *buf,**s;
int *r;
{
    char *ptr;
    int balance;

    for(ptr = buf,balance = 0;ptr < *s;ptr++)
        switch(*ptr) {
            case '(':
                balance++;
                break;

            case ')':
                balance--;
                break;

            default :
                break;
        }

    while(balance-- > 0) {
        *(*s)++ = '(';
        (*n)++;
        w_putchar(' ',0);
    }
}

void escape_mode(iop,node)
int *iop,*node;
{
    char c,string[20];

    message(LEFT,"ESCAPE ",0);
    c = next_char(*iop);
    switch(c) {

        case 'r' :
            read_file(iop);
            return;

        case 'n' :
            message(LEFT,"Enter lispNode",0);
            read_string(string,*iop);
            sscanf(string,"%d",node);
            sprintf(string,"Lispnode %d",*node);
            message(RIGHT,string,0);
            break;

        default :
            break;
    }
}

short REPloop()

```

```

register char c;
char buf[BUFSIZE],*s,*ptr,prompt[3];
int type = 116,iop = input,balance,node = 0,n = 0;

s = buf;
FOREVER {

    switch(c = next_char(iop)) {

        case '\\b' :
            if (n > 0) {
                n--;
                s--;
            }
            *s = '\\0';
            w_putchar('\\b',0);
            break;

        case '\\}' :
            match_brackets(buf,&s,&n);

            /* fall through to flush buf */

        case '\\n' :
            w_putchar('\\n',0);
            *s++ = '\\n';
            nwrite(output,buf,n+1,node,type);
            s = buf;
            n = 0;
            break;

        case ESC :
            escape_mode(&iop,&node);
            break;

        case EOF :
        case CTRL_X :
            nwrite(output,buf,n,node,type);
            s = buf;
            n = 0;
            if (iop == input) {
                nwrite(output,"(exit)\\n",8,node,type);
                return;
            } else {
                close(iop);
                iop = input;
            }
            break;

        default :
            w_putchar(c,0);
            if (n++ < BUFSIZE)
                *s++ = c;
            else {
                nwrite(output,buf,BUFSIZE,node,type);
                n = 1;
                s = buf;
                *s++ = c;
            }
            break;
    }
}

```

8/7/11/04
00:19:17

4

boot.c

```
/*
 * split.h                27/10/1987
 */

extern CELL *get_current_window();
extern CELL *get_origin();
extern CELL *distribute();
extern CELL *do_split();

extern long X_DIM;
extern long Y_DIM;

/* struct definition */

struct _bmap {
    long orx;
    long ory;
    long xdim;
    long ydim;
    long image;
};

#define BITMAP struct _bmap
```

87/11/04
00:13:30

split.c

1

```
/*
 * split.c
 *
 * split.c contains functions for executing a pixel based
 * image splitting algorithm. The algorithm counts pixels in
 * both vertical and horizontal directions and stores these
 * profiles in resp. hprofile[] and vprofile[]. A zero count
 * means there are no pixels so it's possible to split the
 * image at that place. By making sure that the total count (T)
 * is exactly split in twice T/2, a good load balancing can
 * be realized.
 */

/*
 * FLISP include files
 */

#include "limits.h"
#include "local.h"
#include "data.h"
#include "frwd.h"
#include "macrodefs.h"
#include "stdio.h"

/*
 * Include Files
 */

#include "hough.h"
#include "split.h"

/*
 * Constants
 */

#define MAXINT 0x7fff

/*
 * Macro's
 */

#define report(s)      string_out(s,stdout);newline(stderr)
#define report_int(s,g) string_out(sprintf(report_msg,"%s : %d\n",s,g),stderr)

#define add_area(x,y,a,b) {register CELL *tmp;\
                          \
                          Push(Cons(PutNumber(b),NIL));\
                          Swap(Cons(PutNumber(a),Top()));\
                          Swap(Cons(PutNumber(y),Top()));\
                          Swap(Cons(PutNumber(x),Top()));\
                          tmp = Pop();\
                          Swap(Cons(tmp,Top()));}

#define START_LIST      Push( NIL )
#define INSERT_LIST     {register CELL *tmp = Pop();\
                          Swap(Cons(tmp,Top()));}

#define LIST            (Pop())

/*
 * Global Variables
 */

static WORD    *hprofile;    /* contains Y_DIM WORDS */
static WORD    *vprofile;    /* contains X_DIM WORDS */
```

```
static char    report_msg[256];

POINT          origin;

ELEMENT *SplitList = NULL;

static long     sumgaps;
static long     numgaps;
static long     meangap;

/*
 * External variables
 */

/* hough.c */

extern long     *image;

/*
 * C Code
 */

/*
 * Memory allocation functions
 */

BOOLEAN allocate_area( xdim, ydim, area, area_size )
register long **area, *area_size, xdim, ydim;
{

    *area_size = ((xdim-1)/LONG+1)*ydim + 4;
    if ((*area = (long *) calloc(*area_size,sizeof(long))) == NULL) {
        report("ALLOCATE AREA : calloc failed");
        return( FALSE );
    }
    *area_size *= 4;
    return( TRUE );
}

BOOLEAN allocate_profiles()
{
    if ((hprofile = (WORD *) calloc(Y_DIM,sizeof(WORD))) == NULL) {
        report("ALLOCATE HPROFILE : calloc failed");
        return( FALSE );
    }
    if ((vprofile = (WORD *) calloc(X_DIM,sizeof(WORD))) == NULL) {
        report("ALLOCATE VPROFILE : calloc failed");
        return( FALSE );
    }
    return( TRUE );
}

void free_profiles()
{
    free( hprofile );
    free( vprofile );
}

ELEMENT *LinkLists(L1,L2)
register ELEMENT *L1, *L2;
```

```

{
register ELEMENT *p;

    for(p = Ll;p->next != NULL;p = p->next);

    p->next = L2;
    return(Ll);
}

ELEMENT *PutElement( Hu, Bb, T )
register POINT *Hu, *Bb;
register int T;
{
register ELEMENT *p;

    if ((p = (ELEMENT *) calloc(1,sizeof(ELEMENT))) == 0) {
        exit(1);
    }
    p->Hu = *Hu;
    p->Bb = *Bb;
    p->cnt = T;
    p->next = NULL;
    return(p);
}

FreeElements(Ss)
register ELEMENT *Ss;
{
register ELEMENT *TMP;

    while( Ss != NULL) {
        TMP = Ss->next;
        free( Ss );
        Ss = TMP;
    }
}

/*
 * Actual code that performs necessary functions
 */

void Profile( Hu, Bb, T )
/*
 * Profile calculates the vertical and horizontal profile of a
 * selected area in image and stores the result vprofile and
 * hprofile. The selected area has top-left corner H and
 * bottom-right corner Bb.
 */
POINT *Hu, *Bb;
WORD *T;
{
register long v, mask;
register unsigned long w;
register int *hp, *vp;
register int x, u;
WORD y, d, total = 0, bx;
long *xp, *yp;

    x = u = Hu->x % LONG;
    d = Hu->x - u;
    for(clear(mask);x--;)
        mask = (mask<<1)|1;

```

```

for(vp = &vprofile[Hu->x];vp <= &vprofile[Bb->x];vp++)
    *vp = 0;
for(hp = &hprofile[Hu->y];hp <= &hprofile[Bb->y];hp++)
    *hp = 0;

xp = yp = image + Hu->y*X_BYTE + d/LONG;
for(hp = &hprofile[y = Hu->y]; y <= Bb->y; y++, xp = yp += X_BYTE,hp++) {

    for(bx = x = Hu->x, w = *xp++; x <= Bb->x; x = bx +=LONG) {

        v = w<<u;
        w = *xp++;
        v |= (w>>(LONG-u))&mask;
        vp = &vprofile[x];
        while(v && x <= Bb->x) {
            if (v < 0) {
                (*hp)++;
                (*vp)++;
                total++;
            }
            v <<= 1;
            vp++;
            x++;
        }
    }

    *T = total;

    if (T == 0)
        return;

    /*
     * Scratch white space on borders.
     */

    while(vprofile[Hu->x] == 0)
        (Hu->x)++;
    while(vprofile[Bb->x] == 0)
        (Bb->x)--;
    while(hprofile[Hu->y] == 0)
        (Hu->y)++;
    while(hprofile[Bb->y] == 0)
        (Bb->y)--;
}

BOOLEAN Split( g, base, top, T, bound1, bound2, Tn)
/*
 * Split
 */
register WORD *g; /* pointer to profile */
register WORD base; /* start point in profile (g) */
register WORD top; /* end point in g */
WORD T, /* T = g[base + ... + g[top] */
    *bound1,
    *bound2,
    *Tn; /* Tn = g[base] + ... + g[bound1] */
{
register WORD halfT = T/2,

    k1, /* boundary1 */
    k2, /* boundary2 */
    p,
    sum,

```

87/11/04
00:13:30

split.c

3

```

sum1,
sum2;

if (T == 0) {
    return (FALSE);
}

/*
 * determine smallest p : g[0] + ... + g[p] > T/2
 */

for( sum = 0, p = base; p <= top; p++ )
    if ((sum += g[p]) > halfT)
        break;

/*
 * determine k1, k2 :
 *
 * (1) base <= k1 <= p and p ,= k2 <= top and
 * (2) g[k1+1] + ... + g[p] < g[p] + ... + g[k2-1] and g[k1] = 0, or
 *     g[k1+1] + ... + g[p] > g[p] + ... + g[k2-1] and g[k2] = 0
 */

for( k1 = p, sum1 = 0; g[k1] && k1 >= base; k1-- )
    sum1 += g[k1];
for( k2 = p, sum2 = 0; g[k2] && k2 <= top; k2++ )
    sum2 += g[k2];

if (k1 < base) {
    if (k2 > top) {
        return (FALSE);
    }
    sum1 = MAXINT;
} else if (k2 > top)
    sum2 = MAXINT;

if (sum1 < sum2) {
    *bound2 = k1 + 1;
    while(g[k1] == 0 && k1 >= base)
        k1--;
    *bound1 = (k1 < base) ? base : k1;
    *Tn = sum - sum1;
} else {
    *bound1 = k2 - 1;
    while(g[k2] == 0 && k2 <= top)
        k2++;
    *bound2 = (k2 > top) ? top : k2;
    *Tn = sum + sum2 - g[p];
}

return (TRUE);
}

ELEMENT *Vertical( Ss )
register ELEMENT *Ss;
{
    register ELEMENT *Result, *tmp;
    POINT P1, P2;
    WORD Tn;

    Result = NULL;
    while( Ss != NULL ) {

```

```

P1.y = Ss->Bb.y;
P2.y = Ss->Hu.y;

if (Split( vprofile, Ss->Hu.x, Ss->Bb.x, Ss->cnt, &(P1.x), &(P2.x), &Tn )

    /*
     * Substitute first element in Ss by it's two halves.
     */

    sumgaps += P2.x - P1.x;
    numgaps++;
    tmp = PutElement( &(Ss->Hu), &P1, Tn );
    Ss->Hu = P2;
    Ss->cnt = Ss->cnt - Tn;
    tmp->next = Ss;
    Ss = tmp;

} else {

    /*
     * Add element to result list since it can not be splitted
     * and remove it from Ss.
     */

    tmp = Ss;
    Ss = Ss->next;
    tmp->next = Result;
    Result = tmp;

}

return (Result);
}

ELEMENT *Horizontal( Ss )
register ELEMENT *Ss;
{
    register ELEMENT *Result, *tmp;
    POINT P1, P2;
    WORD Tn;

    Result = NULL;
    while( Ss != NULL ) {

        P1.x = Ss->Bb.x;
        P2.x = Ss->Hu.x;

        if (Split( hprofile, Ss->Hu.y, Ss->Bb.y, Ss->cnt, &(P1.y), &(P2.y), &Tn ))

            /*
             * Substitute first element in Ss by it's two halves.
             */

            tmp = PutElement( &P2, &(Ss->Bb), Ss->cnt-Tn );
            Ss->Bb = P1;
            Ss->cnt = Tn;
            tmp->next = Ss;
            Ss = tmp;

        } else {

            /*
             * Add element to result list since it can not be splitted

```

87/11/04
00:13:30

```
        * and remove it from Ss.
        */

    tmp = Ss;
    Ss = Ss->next;
    tmp->next = Result;
    Result = tmp;

}

return(Result);
}

ELEMENT *MainSplit( Hu, Bb )
register POINT *Hu, *Bb;
{
    register ELEMENT *Ss, *Result = NULL, *tmp;
    WORD T;

    /*
     * Allocate space to store profile information
     */

    if (!allocate_profiles()) {
        return( NIL );
    }

    /*
     * Initialize gap information for word detection
     */

    sumgaps = 0;
    numgaps = 0;

    /*
     * Calculate profiles
     */

    Profile( Hu, Bb, &T );

    Ss = Horizontal( PutElement( Hu, Bb, T ) );

    while( Ss != NULL ) {

        Profile( &(Ss->Hu), &(Ss->Bb), &T );

        /*
         * Isolate the first element in Ss.
         */

        tmp = Ss->next;
        Ss->next = NULL;

        /*
         * Try to split vertically.
         */

        Result = LinkLists( Vertical( Ss ), Result );
        Ss = tmp;
    }

    free_profiles();
}
```

split.c

```
    meangap = sumgaps/numgaps;

    return(Result);
}

void get_area( Hu, Bb, area_pointer )
/*
 * get_area selects the area from Hu to Bb from image and stores it at area
 */
POINT *Hu, *Bb;
register long *area_pointer;
{
    register long v, mask, *xp;
    register unsigned long w;
    register int x, u;
    WORD y, d, total = 0, bx;
    long *yp;

    x = u = Hu->x % LONG;
    d = Hu->y - u;
    for(clear(mask);x--;)
        mask = (mask<<1)|1;

    xp = yp = image + Hu->y*X_BYTE + d/LONG;

    for( y = Hu->y; y <= Bb->y; y++, xp = yp += X_BYTE ) {

        for( bx = x = Hu->x, w = *xp++; x <= Bb->x; x = bx +=LONG ) {

            v = w<<u;
            w = *xp++;
            v |= (w>>(LONG-u))&mask;
            *area_pointer++ = v;

        }
    }
}

CELL *distribute()
{
    register int k, mask, w, i;
    register char *p, *q;
    int n, m, pid, host, d, flag, T, Tn;
    int length, type = -1, bytes_read, package = 0, nr_of_packages;
    POINT P1, P2, hwindow, bwindow;
    long xdim, ydim;
    char *buf, *part;

    whoami( &n, &pid, &host, &d );

    if (n == 0) {

        while((length = ntest( &host, &type )) < 16)
            type = -1;

        k = 0;
        mask = 1;
        m = host;

    } else {

        m = -1;
        while((length = ntest( &m, &type )) < 16)
            m = type = -1;
    }
}
```



```

        w = m ^ n;
        mask = w << 1;
        for( k = 1; k < 32 && w > 1; k++ ) {
            w >>= 1;
        }

    buf = (char *) calloc( length, 1 );
    bytes_read = nread( buf, length, &m, &type, &flag );

    origin.x = ((BITMAP *) buf)->orx;
    origin.y = ((BITMAP *) buf)->ory;
    X_DIM = ((BITMAP *) buf)->xdim;
    Y_DIM = ((BITMAP *) buf)->ydim;

    /*
     * If the bitmap is not yet complete then there will be
     * coming more.
     */

    if (bytes_read < 4*((X_DIM-1)/LONG+1)*Y_DIM + 16) {

        /*
         * Allocate right amount of memory for entire bitmap
         */

        if (!allocate_area( X_DIM, Y_DIM, &part, &length )) {
            return( NIL );
        }

        /*
         * copy the buffer that has been read in the first time to
         * the new buffer that is able to hold the entire image
         */

        for( p = buf, q = part; p < buf + bytes_read; p++, q++) {
            *q = *p;
        }

        /*
         * Now we can throw away the temporary buffer
         */

        free( buf );

        /*
         * The rest the image can be read and stored in the newly
         * allocated buffer
         */

        buf = part;

        /*
         * The buffer already contains bytes_read number of bytes
         */

        p = part + bytes_read;
        while( p < buf + length ) {

            bytes_read = nread( p, PACKAGE_SIZE, &m, &type, &flag );
            p += bytes_read;
        }
    }

```

```

    image = &(((BITMAP *) buf)->image);

    if (d == k) {
        return( T );
    }

    hwindow.x = 0;
    hwindow.y = 0;
    bwindow.x = X_DIM - 1;
    bwindow.y = Y_DIM - 1;

    P1 = bwindow;

    if (!allocate_profiles()) {
        return( NIL );
    }

    for( i = k; i < d; i++ ) {

        /*
         * Determine neighgour of n in higher dimension
         */

        m = n ^ mask;
        mask <<= 1;

        /*
         * Split window (hwindow,bwindow) into two halves, either
         * vertically or horizontally
         *
         * First calculate the Profiles in both directions
         */

        Profile( &hwindow, &bwindow, &T );

        /*
         * Try to split horizontally
         */

        P1.x = bwindow.x;
        P2.x = hwindow.x;

        if (Split( hprofile, hwindow.y, bwindow.y, T, &(P1.y), &(P2.y), &Tn )) {

            /*
             * Horizontal split succeeded, areas are :
             *
             * (hwindow,P1) with count Tn, and
             * (P2,bwindow) with count T-Tn.
             */

        } else {

            /*
             * Try to split vertically
             */

            P1.y = bwindow.y;
            P2.y = hwindow.y;

            if (Split( vprofile, hwindow.x, bwindow.x, T, &(P1.x), &(P2.x), &Tn )) {

                /*
                 * Vertical split succeeded, areas are :
                 *

```

```

        * (hwindow,P1) with count Tn, and
        * (P2,bwindow) with count T-Tn.
        */

    } else {

        /*
        * No further split possible, set P1 and P2
        * to bwindow
        */

        P1 = P2 = bwindow;
        i = d;

    }

    /*
    * We're going to send the second half, that is (P2,bwindow),
    * to neighbour m, first we have to get the information
    * out of our image.
    */

    xdim = bwindow.x - P2.x + 1;
    ydim = bwindow.y - P2.y + 1;

    if (!allocate_area( xdim, ydim, &part, &length )) {
        return( NIL );
    }

    ((BITMAP *) part)->orx = origin.x + P2.x;
    ((BITMAP *) part)->ory = origin.y + P2.y;
    ((BITMAP *) part)->xdim = xdim;
    ((BITMAP *) part)->ydim = ydim;
    get_area( &P2, &bwindow, &((BITMAP *) part)->image );

    /*
    * The area has been retrieved and is pointed to by half.
    * Now we have to sent the coordinates of the origin of
    * half with respect to (0,0) ! , then we
    * we send the x and y dimensions of the new half. At
    * last we send the entire bitmap
    */

    if (length < PACKAGE_SIZE) {

        /*
        * The bitmap fits in one package
        */

        nwrite( part, length, m, type, &flag );

    } else {

        /*
        * If the bitmap is too large then transmit it
        * in packages
        */

        nr_of_packages = length/PACKAGE_SIZE;
        for( i = 0; i < nr_of_packages; i++ ) {

            nwrite( &part[i * PACKAGE_SIZE],
                    PACKAGE_SIZE, m, type, &flag );
        }
    }
}

```

```

    }

    nwrite( &part[i * PACKAGE_SIZE],
            length % PACKAGE_SIZE, m, type, &flag );

    free( part );

    bwindow = P1;

}

free_profiles();

/*
* Now we can throw away the parts of the image that have been
* sent to the neighbours.
*/

xdim = P1.x - hwindow.x + 1;
ydim = P1.y - hwindow.y + 1;

if (!allocate_area( xdim, ydim, &part, &length )) {
    return( NIL );
}

get_area( &hwindow, &P1, part );

free( buf );

image = part;
origin.x += hwindow.x;
origin.y += hwindow.y;
X_DIM = xdim;
Y_DIM = ydim;

return( T );
}

CELL *do_split( Hu, Bb )
register POINT *Hu, *Bb;
{
    register ELEMENT *P;
    register int x1, x2, y1, y2;

    P = SplitList = MainSplit( Hu, Bb );
    START_LIST;

nextword:
    START_LIST;
    do {

        add_area( P->Hu.x, P->Hu.y, P->Bb.x, P->Bb.y );

        x1 = P->Hu.x;
        y1 = P->Hu.y;

        if ((P = P->next) == NULL) {
            goto endwords;
        }

        x2 = P->Bb.x;
        y2 = P->Hu.y;
    } while (1);
}

```

87/11/04
00:13:30

split.c

7

```
    } while(y1 == y2 && (x1 - x2) <= meangap);

    INSERT_LIST;
    goto nextword;

endwords:
    INSERT_LIST;
    FreeElements(SplitList);
    return( LIST );
}

CELL *get_current_window()
{
    Push(Cons(PutNumber(Y_DIM-1),NIL));
    Swap(Cons(PutNumber(X_DIM-1),Top()));
    Swap(Cons(PutNumber(0),Top()));
    Swap(Cons(PutNumber(0),Top()));
    return( Pop() );
}

CELL *get_origin()
{
    Push(Cons(PutNumber(origin.y),NIL));
    Swap(Cons(PutNumber(origin.x),Top()));
    return( Pop() );
}
```

87/11/02
19:09:41

graph.h

1

```
/*  
 * graph.h  
 */
```

25/9/1987

```
extern WORD edge_to_vertex();  
extern void clear_graph();  
extern CELL *all_cycles();  
extern CELL *list_vertices();  
extern CELL *get_vertex();  
extern CELL *is_connected();  
extern CELL *get_adj();  
extern CELL *get_subgraph();  
extern void unmark();  
extern void cycles();
```

87/10/31
16:14:48

graph.c

1

```
/*
 * graph.c
 */

#include "limits.h"
#include "local.h"
#include "data.h"
#include "frwr.h"
#include "macrodefs.h"
#include "hough.h"
#include "graph.h"

#define MAX_VERTEX 20
#define vertex vrtx /* silly isn't it */

static BOOLEAN connection[MAX_VERTEX][MAX_VERTEX];
static BOOLEAN available[MAX_VERTEX];
static BOOLEAN unavailable[MAX_VERTEX][MAX_VERTEX];
static POINT vertex[MAX_VERTEX];
static WORD path[MAX_VERTEX];

static WORD nr_of_vertices, route, s;

#define adjacent(v,w) (connection[w][v])
#define foreign(v) (v < 0 || v >= nr_of_vertices)
#define legal(v) (0 <= v || v < nr_of_vertices)
#define START_LIST Push( NIL )
#define ADD_ELEMENT(v) (Swap(Cons( PutNumber( v ), Top())))
#define INSERT_LIST {register CELL *tmp = Pop();\
                      Swap(Cons(tmp,Top()));}
#define LIST (Pop())

WORD edge_to_vertex( edge_list )
register CELL *edge_list;
{
register WORD h, b;
POINT Hu, Bb;
register CELL *edge;

clear_graph();
while( edge_list != NIL && nr_of_vertices < MAX_VERTEX ) {

    edge = Car( edge_list );
    edge_list = Cdr( edge_list );

    Hu.x = GetNumber( Car( edge ) );
    Hu.y = GetNumber( Cadr( edge ) );
    Bb.x = GetNumber( Car( Cddr( edge ) ) );
    Bb.y = GetNumber( Cadr( Cddr( edge ) ) );

    for( h = 0; h < nr_of_vertices; h++ )
        if (vertex[h].x == Hu.x && vertex[h].y == Hu.y)
            goto H_found;
    nr_of_vertices++;
    vertex[h] = Hu;

H_found:
    for( b = 0; b < nr_of_vertices; b++ )
        if (vertex[b].x == Bb.x && vertex[b].y == Bb.y)
            goto B_found;
    nr_of_vertices++;
    vertex[b] = Bb;

B_found:
    connection[h][b] = TRUE;
```

```
        connection[b][h] = TRUE;
    }
    return(nr_of_vertices);
}

void clear_graph()
{
register WORD i,j;

    for( i = 0; i < MAX_VERTEX; i++ ) {
        available[i] = TRUE;
        for( j = 0; j < MAX_VERTEX; j++ ) {
            connection[i][j] = unavailable[i][j] = FALSE;
        }
    }
    nr_of_vertices = 0;
}

void cycles( v, flag )
register WORD v;
register BOOLEAN *flag;
{
register WORD w, i;
register CELL *tmp;
BOOLEAN g;

    path[route++] = v;
    available[v] = FALSE;
    *flag = FALSE;
    for ( w = s; w < nr_of_vertices; w++ ) {
        if adjacent(v,w) {
            if (w == s) {
                START_LIST;
                for ( i = route-1; i >= 0; i-- ) {
                    ADD_ELEMENT(path[i]);
                }
                INSERT_LIST;
                *flag = TRUE;
            } else if (available[w]) {
                cycles( w, &g );
                *flag |= g;
            }
        }
    }
    if (*flag) {
        unmark( v );
    } else {
        for ( w = s; w < nr_of_vertices; w++ ) {
            if adjacent(v,w) {
                unavailable[w][v] = TRUE;
            }
        }
        route--;
    }
}

void unmark( u )
register WORD u;
{
register WORD w;

    available[u] = TRUE;
    for ( w = s; w < nr_of_vertices; w++ ) {
        if (unavailable[u][w]) {
```

graph.c

```

        unavailable[u][w] = FALSE;
        if (!available[w]) {
            unmark( w );
        }
    }
    return;
}

CELL *all_cycles()
{
    register WORD v, w;
    BOOLEAN f;

    route = 0;
    START_LIST;
    for( s = 0; s < nr_of_vertices-1; s++ ) {
        for( v = s; v < nr_of_vertices; v++ ) {
            available[v] = TRUE;
            for( w = s; w < nr_of_vertices; w++ ) {
                unavailable[v][w] = FALSE;
            }
            cycles( s, &f );
        }
    }
    return( LIST );
}

CELL *list_vertices()
{
    register WORD v, w;
    register CELL *tmp;

    START_LIST;
    for( v = nr_of_vertices-1; v >= 0; v-- ) {
        START_LIST;
        for( w = 0; w < nr_of_vertices; w++ ) {
            if adjacent(v,w) {
                ADD_ELEMENT(w);
            }
        }
        INSERT_LIST;
    }
    return( LIST );
}

CELL *get_vertex( v )
{
    register WORD v;
    register CELL *tmp;

    if foreign(v) {
        return( NIL );
    }
    START_LIST;
    ADD_ELEMENT(vertex[v].y);
    ADD_ELEMENT(vertex[v].x);
    return( LIST );
}

CELL *is_connected( v, w )
{
    register WORD v, w;
    {
        if (foreign(v) || foreign(w))
    }
}

```

```

        return( NIL );
    }
    return( adjacent(v,w) ? T : NIL );
}

CELL *get_adj( v )
/*
 * Return the vertices that are adjacent to v
 */
{
    register WORD v;
    register WORD w;

    START_LIST;
    for( w = 0; w < nr_of_vertices; w++ ) {
        if adjacent(v,w) {
            ADD_ELEMENT(w);
        }
    }
    return( LIST );
}

CELL *get_subgraph( containing_vertices )
{
    register CELL *containing_vertices;
    {
        register WORD v, w;

        for( v = 0; v < nr_of_vertices; v++ ) {
            available[v] = FALSE;
        }
        while( containing_vertices != NIL ) {
            v = GetNumber( Car( containing_vertices ) );
            if legal(v) {
                available[v] = TRUE;
            }
            containing_vertices = Cdr( containing_vertices );
        }
        START_LIST;
        for( v = 0; v < nr_of_vertices; v++ ) {
            if (available[v]) {
                available[v] = FALSE;
                for( w = 0; w < nr_of_vertices; w++ ) {
                    if (available[w] && adjacent(v,w)) {
                        START_LIST;
                        ADD_ELEMENT(vertex[w].y);
                        ADD_ELEMENT(vertex[w].x);
                        ADD_ELEMENT(vertex[v].y);
                        ADD_ELEMENT(vertex[v].x);
                        INSERT_LIST;
                    }
                }
            }
        }
        return( LIST );
    }
}

```

87/10/31
16:21:14

hough.h

1

```
/*
 * hough.h
 */

/* Constants */

#ifndef HOUGH
#define HOUGH

#define WORD    int                /* 2 byte storage unit */
#define LONG    (sizeof(long)*8)
#define X_BYTE  ((X_DIM-1)/LONG+1)

extern long X_DIM;
extern long Y_DIM;

/* Structures */

struct point {
    WORD x,
        y;
} POINT;

struct element {
    struct point    Hu;                /* Upper-left corner */
    struct point    Bb;                /* Bottom-right corner */
    WORD            cnt;                /* pixel count in square */
    struct element  *next;              /* link to next element */
};

struct point_list {
    WORD x,y;
    struct point_list *next;
};

#define POINT struct point
#define POINT_LIST struct point_list
#define ELEMENT struct element

/* Macro's */

#define abs(x) ((x) > 0) ? (x) : (-x)
#define max(x,y) (((x) > (y)) ? (x) : (y))
#define min(x,y) (((x) < (y)) ? (x) : (y))
#define clear(x) ((x) ^= (x))

/* external variables that are defined in hough.c */

extern WORD LENGTH;
extern long RC_MAX;
extern long *image;

/* functions in hough.c */

extern CELL *perform_hough();
extern CELL *do_split();

#endif
```



```

/*
 * C code
 */

/*
 * Basic pattern scanning functions
 */

look_for_pixels( Hu, Bb )
register POINT *Hu, *Bb;
{
    register long v, w, mask;
    register WORD x, u;
    WORD y, d, bx, xsize, ysize;
    register long *xp, *yp;

    x = u = Hu->x % LONG;
    d = Hu->x - u;
    for(clear(mask);x--;)
        mask = (mask<<1)|1;
    xp = yp = image + Hu->y*X_BYTE + d/LONG;

    xsize = Bb->x - Hu->x + 1;
    ysize = Bb->y - Hu->y + 1;

    for(y = 0; y < ysize; y++, xp = yp += X_BYTE) {

        for(bx = x = 0, w = *xp++; x < xsize; x = bx +=LONG) {

            v = w<<u;
            w = *xp++;
            v |= (w>>(LONG-u))&mask;
            while(v && x < xsize) {
                if (v < 0) {
                    Hough( (long)-x, (long)y, RC_MAX, xy_height, xy_plane
                        Hough( (long)-y, (long)x, RC_MAX, yx_height, yx_plane
                }
                v <<= 1;
                x++;
            }
        }
    }
}

Workspace(width,height,hh,plane,plane_size)
long width,height;
WORD *plane[];
long *plane_size,*hh;
{
    if (*plane != NULL){
        free(*plane);
    }
    *plane_size = (2*width+1)*(2*height+1);
    if((*plane = (WORD *) calloc(*plane_size, sizeof(WORD))) == NULL) {
        exit(1);
    }
    *hh = height;
}

sweep_plane ()
{

```


87/11/02
19:02:24

hough.c

2

```

register WORD *p;

    for(p = xy_plane; p < xy_plane + xy_size; p++)
        *p = 0;
    for(p = yx_plane; p < yx_plane + yx_size; p++)
        *p = 0;
}

Hough( a, b, width, height, hough)
register long a,b;
long height, width;
WORD *hough;
{
    register WORD i,*p,*base = hough;
    register long x,y,bound;
    long hw;

    if (a == 0) {          /* horizontal line */

        base = &hough[offset(-width,b)];

        for(p = base; p < base + 2*RC_MAX + 1; p++)
            (*p)++;

    } else {

        for(y = b, x = 0; x >= -width; x--) {
            i = -a;
            while (i--) {
                hough[offset(x,y)]++;
                if (++y > height)
                    goto down_part;
            }
        }

    down_part:
        for(y = b - 1, x = 1; x <= width; x++) {
            i = -a;
            while (i--) {
                hough[offset(x,y)]++;
                if (--y < -height)
                    goto done_line;
            }
        }

    done_line:
        return;

    }

xy_line( a, b )
register WORD a, b;
{
    register long w;
    register WORD x,y,i;
    register BOOLEAN already_line = FALSE;
    WORD x_start, y_start, x_end, y_end, n;

    if (a > 0) {
        for(y = b, x = 0; x < hxdim; x++) {
            i = a;
            while (i--) {
                if (pixel_on(x,y) && y >= 0) {
                    if (already_line) {

```

```

                        x_end = x;
                        y_end = y;
                        n++;
                    } else {
                        already_line = TRUE;
                        x_end = x_start = x;
                        y_end = y_start = y;
                        n = 1;
                    }
                } else if (already_line) {
                    already_line = FALSE;
                    if (n > LLENGTH) {
                        add_segment(x_start,y_start,x_end,y_end);
                    }
                }
            }
            if (++y == hydim)
                goto xy_end;
        }
    }
} else if (a < 0) {
    for(y = b, x = 0; x < hxdim; x++) {
        i = -a;
        while (i--) {
            if (pixel_on(x,y) && y < hydim) {
                if (already_line) {
                    x_end = x;
                    y_end = y;
                    n++;
                } else {
                    already_line = TRUE;
                    x_end = x_start = x;
                    y_end = y_start = y;
                    n = 1;
                }
            } else if (already_line) {
                already_line = FALSE;
                if (n > LLENGTH) {
                    add_segment(x_start,y_start,x_end,y_end);
                }
            }
            if (--y < 0)
                goto xy_end;
        }
    }
} else if (b < 0 || b >= hydim) {
    return;
} else {
    /* a = 0 */
    for(x = 0; x < hxdim; x++) {
        if (pixel_on(x,b)) {
            if (already_line) {
                x_end = x;
                n++;
            } else {
                already_line = TRUE;
                x_end = x_start = x;
                n = 1;
            }
        }
    }
} else if (already_line) {
    already_line = FALSE;
    if (n > LLENGTH) {
        add_segment(x_start,b,x_end,0);
    }
}
}

```

```

    }
    y_end = y_start = b;
xy_end:
    if (already_line && (n > LLENGTH)) {
        add_segment(x_start, y_start, x_end, y_end);
    }
}

yx_line( a, b )
register WORD a, b;
{
    register long w;
    register WORD x, y, i;
    register BOOLEAN already_line = FALSE;
    WORD x_start, y_start, x_end, y_end, n;

    if (a > 0) {
        for(x = b, y = 0; y < hydim; y++) {
            i = a;
            while (i-- > 0) {
                if (pixel_on(x, y) && x >= 0) {
                    if (already_line) {
                        x_end = x;
                        y_end = y;
                        n++;
                    } else {
                        already_line = TRUE;
                        x_end = x_start = x;
                        y_end = y_start = y;
                        n = 1;
                    }
                } else if (already_line) {
                    already_line = FALSE;
                    if (n > LLENGTH) {
                        add_segment(x_start, y_start, x_end, y_end);
                    }
                }
                if (++x == hxdim)
                    goto xy_end;
            }
        }
    } else if (a < 0) {
        for(x = b, y = 0; y < hydim; y++) {
            i = -a;
            while (i-- > 0) {
                if (pixel_on(x, y) && x < hxdim) {
                    if (already_line) {
                        x_end = x;
                        y_end = y;
                        n++;
                    } else {
                        already_line = TRUE;
                        x_end = x_start = x;
                        y_end = y_start = y;
                        n = 1;
                    }
                } else if (already_line) {
                    already_line = FALSE;
                    if (n > LLENGTH) {
                        add_segment(x_start, y_start, x_end, y_end);
                    }
                }
            }
        }
    }
}

```

```

        if (--x < 0)
            goto yx_end;
    }
} else if (b < 0 || b >= hxdim) {
    return;
} else {
    /* a = 0 */
    for(y = 0; y < hydim; y++) {
        if (pixel_on(b, y)) {
            if (already_line) {
                y_end = y;
                n++;
            } else {
                already_line = TRUE;
                y_end = y_start = y;
                n = 1;
            }
        } else if (already_line) {
            already_line = FALSE;
            if (n > LLENGTH) {
                add_segment(b, y_start, b, y_end);
            }
        }
    }
    x_end = x_start = b;
}
xy_end:
    if (already_line && (n > LLENGTH)) {
        add_segment(x_start, y_start, x_end, y_end);
    }
}

Hough_climb( width, height, hough, threshold, XY )
register long width, height;
register WORD *hough, threshold;
BOOLEAN XY;
{
    register WORD x, y;

    for(x = -width; x <= width; x++) {
        if (x != (XY ? -1 : 1)) {
            for(y = -height; y <= height; y++) {
                if (hough[offset(x, y)] > threshold) {
                    if (XY)
                        xy_line( x, y );
                    else
                        yx_line( x, y );
                }
            }
        }
    }
}

/*
 * Coordination of basic pattern scanning functions (see code above)
 */
CELL *perform_hough( Hu, Bb )

```

87/11/02
19:02:24

hough.c

4

```
register POINT *Hu, *Bb;
{
register CELL *Result;

/*
 * Initialize paramters for tuning hough procedure
 */

hhough = *Hu;
bhough = *Bb;

/*
 * Record size of actual image area to be processed
 */

hxdim = (long) bhough.x - hhough.x + 1;
hydim = (long) bhough.y - hhough.y + 1;

/*
 * Allocate necessary workspace in memory
 */

Workspace(RC_MAX, (hydim + RC_MAX*hxdim), &xy_height, &xy_plane, &xy_size);
Workspace(RC_MAX, (hxdim + RC_MAX*hydim), &yx_height, &yx_plane, &yx_size);

/*
 * Initialize workspace to zero
 */

sweep_plane();

/*
 * Part one :
 *
 * Search area for pixels (a,b) and increment hough
 * values of pixels (x,y) that are points on the line  $y = ax+b$ 
 * in the xy-hough plane. This is done twice, the second time
 * the image is imaginary rotated over an angle of 90 degrees
 * with actually means that all points (x,y) on the line
 *  $y = -bx+a$  WORD yx-hough plane are incremented.
 */

look_for_pixels( Hu, Bb );

/*
 * Part two :
 *
 * Now select all points (a,b) in resp. xy- and yx hough plane
 * that have a hough value greater than threshold. The value is an
 * exact measure of the number of pixels that together form line
 * pieces on the line  $y = ax+b$  in the original image.
 */

Push( NIL );
Hill_climb( RC_MAX, xy_height, xy_plane, xyhold, TRUE );
Hill_climb( RC_MAX, yx_height, yx_plane, yxhold, FALSE );
return( Pop() );
}
```

8/10/31
16:15:59

master.c

1

```
/*
 * master.c
 *
 * 27/8/1987
 *
 * master.c initializes the displays on the screen using the
 * windows library and downloads a lispexecutable to the
 * allocated hypercube with nloadm(). Master is executed by
 * boot which ensures that the standard input of master is
 * connected to the hypercube handle returned by nopen() that
 * was called by the boot program.
 * After that the master program receives all output from the
 * active nodes and dispatches the messages to the proper
 * display belonging to a certain node.
 */

#include "window.h"
#include <power.h>

#define PMODE 0644

char *MSTR_NR = "Master version 3.0";
char *log_file = "flisp.log";
int fd;

main(argc,argv)
int argc;
char *argv[];
{
    int chan = 0, node, size, type, previous, nnodes, i, dimension;
    char buf[BUFSIZE], *ptr, msg[80];

    nnodes = power2((dimension = atoi(argv[1])));

    /*
     * It is nice to have a log file since Flisp doesn't
     * have any file I/O. The name of the logfile is stored
     * in log_file
     */

    if ((fd = open(log_file,1)) == -1) {
        if ((fd = creat(log_file,PMODE)) == -1) {
            printf("Cannot create logfile %s\n", log_file);
            exit(-1);
        }
    }

    initialize_windows(nnodes,1,16);

    for(node = 0; node < nnodes; node++) {
        sprintf(msg, "LispNode %d", node);
        message(RIGHT, msg, node);
        message(MIDDLE, MSTR_NR, node);
        message(LEFT, "", node);
    }

    /*
     * Start dispatching output from the nodes forever. This
     * process only halts if it is killed
     */

    FOREVER {
        node = -1;
        type = -1;
```

```
size = nread(0, buf, BUFSIZE, &node, &type);
for(ptr = buf; ptr < buf + size; ptr++) {
    w_putchar(*ptr, node);
}
write(fd, buf, size);
}
```

87/11/02
19:10:08

ownlib.h

1

```
/*  
 * ownlib.h  
 */  
  
extern void n_puts();  
extern char *n_gets();  
extern int _n_fillbuf();  
extern int _n_flushbuf();  
extern CHAN *n_open();  
extern void n_close();
```

```
/* nio.h                26/7/1987
 *
 * _NCHAN is the number of files that can be handled.
 * nio.h contains constants and macros that are used in the
 * standard host/node communication. At this time it is
 * not possible to maintain multiple communication channels
 * to the host or a single node.
 *
 * J. Henseler, J.C. Scholtes and C.R.J. Verhoest
 * Delft University of Technology
 */

#ifndef HOST

#define HOST
#define NODE

#define _NCHAN 25

#define NULL 0
#define EOF (-1)
#define BUFSIZE 1024
#define BUFSIZ 1024

#define _READ 0x01
#define _WRITE 0x02
#define _UNBUF 0x04
#define _BIGBUF 0x08
#define _EOF 0x10
#define _ERR 0x20

struct _chanctrlblk {
    int _node; /* node number */
    int _procp; /* process number on the host */
    int _chd; /* communication channel to the host */
    int _dim; /* dimension of the allocated hypercube */
    int _cflag; /* information returned by several vertex systemcalls*/
    char *_ptr; /* next character position */
    int _cnt; /* number of characters left */
    char *_base; /* location of the buffer */
    int _flag; /* status of the channel */
    char _type; /* message type; user defined at n_open() */
};

#define CHAN struct _chanctrlblk

#define __xn(p) (--(p)->_cnt >= 0)

#define n_getc(p) (__xn(p) ? *(p)->_ptr++ & 0377 : _n_fillbuf(p))
#define n_getchar() n_getc(stdin)

#define __yn(c,p) (int) (*(p)->_ptr++ = (c))
#define __zn(c,p) _n_flushbuf((c), (p))

#define n_putc(c,p) _n_flushbuf((c), (p))
/* (__xn(p) && c != '\n' ? __yn(c,p) : __zn(c,p)) */

#define n_putchar(x) n_putc((x), stdout)

/* Extern variabels */

extern void n_init();
extern void n_puts();
extern char *n_gets();
```

```
extern int _n_fillbuf();
extern int _n_flushbuf();
extern CHAN *n_open();
extern void n_close();
extern unsigned char n_error();
extern unsigned char n_tell();
extern unsigned char n_see();
extern void n_told();
extern void n_seen();

extern CHAN _chcb[];
extern CHAN *stdin;
extern CHAN *stdout;
extern CHAN *stderr;

#endif
```

87/10/31
16:16:44

ownlib.c

1

```
/*
 * ownlib.c                                27/8/1987
 *
 * J. Henseler, J.C. Scholtes and C.R.J. Verhoest
 *
 * Delft University of Technology
 * Julianalaan 132
 * Delft
 * The Netherlands
 *
 * Phone : 015-785245
 *
 * NOTE:
 *
 * Ownlib is a C library that serves as a standard node input/output
 * package for communication on the ncube system.
 * Reason for the development of this package was the need for a flexible
 * way to implement existing software, written in C, on the node processors.
 * This reason should not keep you from using the package for your own
 * 'to be developed' programs, for it is easily understood due to it's
 * resemblance with the standard input/output package as described by
 * Kerningham and Richie in their book 'The C programming language'.
 *
 * A hypercube-structure based program that uses the package can be tested
 * in a restricted way. The way to do this is to change the _fillbuf and
 * _flushbuf functions so that they perform a monitoring function that
 * reports to the user about the communication that's going on in the
 * program. After this change the package should be compiled for the host
 * as well as the hypercube-structured program.
 *
 * Because the hypercube is a distributed-memory system and
 * not a shared-memory one, a hypercube-system simulator can easily be
 * implemented if when the monitoring functions are extended with
 * scheduling functions.
 */

#include "stdio.h"

CHAN _chcb[_NCHAN], *stdin, *stdout, *stderr;

void n_puts(s, chp)      /* put string s to chan */
register char *s;
register CHAN *chp;
{
    register int c;

    while(c = *s++)
        n_putc(c, chp);
}

char *n_gets(s, n, chp)  /* get at most n chars from chan */
register char *s;
register int n;
register CHAN *chp;
{
    register int c;
    register char *cs;

    cs = s;
    while(--n > 0 && (c = n_getc(chp)) != EOF)
        if((*cs++ = c) == '\n' || c == '\0')
            break;
    *cs = '\0';
    return((c == EOF && cs == s) ? NULL : s);
}
```

```

}

int n_fillbuf(chp)      /* allocate and fill input buffer */
register CHAN *chp;
{
    if(((chp->_flag & _READ) == 0 || chp->_flag & (_EOF | _ERR)) != 0)
        return(EOF);

    chp->_ptr = chp->_base;
    chp->_cnt = nread(chp->_ptr,
        (chp->_flag & _UNBUF) ? 1 : BUFSIZE,
        &(chp->_chd,
        &(chp->_type,
        &(chp->_cflag));
    if (--chp->_cnt < 0) {
        chp->_flag = _EOF;
        chp->_cnt = 0;
        return(EOF);
    }
    return(*(chp->_ptr++ & 0377)); /* make char positive */
}

int n_flushbuf(c, chp)  /* fill and flush output buffer */
register char c;
register CHAN *chp;
{
    if(((chp->_flag) & _WRITE) == 0)
        return(EOF);

    *(chp->_ptr = c;      /* save last char */

    chp->_ptr = chp->_base;

    chp->_cnt = nwrite(chp->_ptr,
        (chp->_flag & _UNBUF) ? 1 : BUFSIZE - chp->_cnt,
        chp->_chd,
        chp->_type,
        &(chp->_cflag));
    if (chp->_cnt < 0) {
        chp->_flag = _EOF;
        chp->_cnt = 0;
        return(EOF);
    } else
        chp->_cnt = (chp->_flag & _UNBUF) ? 1 : BUFSIZE;
    return(c);
}

CHAN *n_open(name, mode, type)
register int name;
register char mode, type;
{
    static char smallbuf[_NCHAN]; /* for unbuffered i/o */
    char *calloc();
    register CHAN *chp;

    if(mode != 'r' && mode != 'w')
        return(NULL);

    if (n_error(name))
        return(NULL);
}
```

```

for(chp = _chcb; chp < _chcb + _NCHAN; chp++)
    if ((chp->flag & (_READ | _WRITE)) == 0)
        break; /* found free slot */

if (chp >= _chcb + _NCHAN) /* no free slots */
    return(NULL);

chp->_chd = name;
if (mode == 'w')
    chp->_flag |= _UNBUF; /* due to response problem */

chp->_base = NULL;
while(chp->_base == NULL) /* find buffer space */
    if ((chp->_flag & _UNBUF)) /* unbuffered */
        chp->_base = &smallbuf[chp->_chd];
    else if ((chp->_base = calloc(BUFSIZE, 1)) == NULL)
        chp->_flag |= _UNBUF; /* can't get big buf */
    else
        chp->_flag |= _BIGBUF; /* got big one */

if (mode == 'w')
    chp->_cnt = (chp->_flag & _UNBUF) ? 1 : BUFSIZE;
else
    chp->_cnt = 0;
chp->_ptr = chp->_base;
chp->_flag &= ~(_READ | _WRITE);
chp->_flag |= (mode == 'r') ? _READ : _WRITE;
chp->_type = type;
return(chp);

void n_close(chp)
register CHAN *chp;
{
    _n_flushbuf(EOF, chp);
    free(chp->_base);
    chp->_node = 0;
    chp->_procp = 0;
    chp->_chd = 0;
    chp->_cnt = 0;
    chp->_base = NULL;
    chp->_ptr = NULL;
    chp->_flag = 0;
    chp->_cflag = 0;
}

void n_init()
/*
 * n_init initializes channels to the host
 */
{
    static char c;
    CHAN *chp;

    whoami( &(((_chcb[0])->_node), /* *node number */
            &(((_chcb[0])->_procp), /* *process id on the chan */
            &(((_chcb[0])->_chd), /* *io channel to chan */
            &(((_chcb[0])->_dim)); /* *dimension of the hypercube*/

    _chcb[1] = _chcb[0];

    stdin = &_chcb[0];

```

```

stdout = stderr = &_chcb[1];

_chcb[0]._flag = _chcb[1]._flag = 0;

_chcb[0]._flag |= _READ;
_chcb[1]._flag |= (_WRITE | _UNBUF);

for(chp = _chcb; chp < _chcb + 2; chp++) {

    chp->_base = NULL;
    while(chp->_base == NULL) /* find buffer space */
        if ((chp->_flag & _UNBUF)) /* unbuffered */
            chp->_base = &c;
        else if ((chp->_base = calloc(BUFSIZE, 1)) == NULL)
            chp->_flag |= _UNBUF; /* can't get big buf */
        else
            chp->_flag |= _BIGBUF; /* got big one */

    if (chp->_flag & _WRITE)
        chp->_cnt = (chp->_flag & _UNBUF) ? 1 : BUFSIZE;
    else
        chp->_cnt = 0;

    chp->_ptr = chp->_base;
    chp->_type = 't';
}

int power2(n)
int n;
{
    switch(n) {

        case 0 : return(0x00000001);
        case 1 : return(0x00000002);
        case 2 : return(0x00000004);
        case 3 : return(0x00000008);
        case 4 : return(0x00000010);
        case 5 : return(0x00000020);
        case 6 : return(0x00000040);
        case 7 : return(0x00000080);
        case 8 : return(0x00000100);
        case 9 : return(0x00000200);
        case 10 : return(0x00000400);
        case 11 : return(0x00000800);
        case 12 : return(0x00001000);
        case 13 : return(0x00002000);
        case 14 : return(0x00004000);
        case 15 : return(0x00008000);
        case 16 : return(0x00010000);
        case 17 : return(0x00020000);
        case 18 : return(0x00040000);
        case 19 : return(0x00080000);
        case 20 : return(0x00100000);
        case 21 : return(0x00200000);
        case 22 : return(0x00400000);
        case 23 : return(0x00800000);
        case 24 : return(0x01000000);
        case 25 : return(0x02000000);
        case 26 : return(0x04000000);
        case 27 : return(0x08000000);
        case 28 : return(0x10000000);
        case 29 : return(0x20000000);

```


8/7/10/31
16:16:44

ownlib.c

3

```
        case 30 : return(0x40000000);
        case 31 : return(0x80000000);
        default : return(-1);
    }
}

unsigned char n_error(n)
int n;
{
    if ((_chcb[0]._flag&_READ) == 0)
        return(1);

    return(n < -1 || n >= power2(_chcb[0]._dim));
}

unsigned char n_see(chp)
CHAN *chp;
{
    if ((chp->_flag&_READ) == 0)
        return(0);

    stdin = chp;
    return(1);
}

void n_seen()
{
    stdin = &_amp;_chcb[0];
}

unsigned char n_tell(chp)
CHAN *chp;
{
    if ((chp->_flag&_WRITE) == 0)
        return(0);

    stdout = chp;
    return(1);
}

void n_told()
{
    stdout = &_amp;_chcb[1];
}
```

```
/*
 * table.h
 *
 * 9/10/1987
 */

#define TABLE_SIZE (27*27*27)
#define FALSE 0
#define TRUE 1
#define FOREVER for(;;)
#define TREE_SIZE 5000
#define TYPE 111
#define OK 116
#define NR_OF_PACKETS ((TABLE_SIZE/PACKET_SIZE) + 1)
#define PACKET_SIZE 512
#ifdef vertex
#define BUF_SIZE (PACKET_SIZE*sizeof(int))
#else
#define BUF_SIZE (PACKET_SIZE*sizeof(short))
#endif

typedef char BOOLEAN;

/* structures */

struct node {
    int left;
    int right;
    short f;
    short entry;
};
```

87/10/31
16:17:19

sort.c

1

```
/*
 * sort.c
 *
 * 9/10/1987
 */

#include "table.h"
#include "nio.h"

/* Global variables */

struct node bintree[TREE_SIZE];
short table[TABLE_SIZE];
int bottom = 0; /* start of freelist in bintree */
int magic_number = 0; /* new number for squeezed entries */
int nr_of_classes = 0; /* count of different numbers in table */
int group_size = 0;
int class_nr = 0;
short previous = 0;
char msg[100];

/* C code */

void insert_in_tree( tree, f, entry )
register struct node *tree;
register int entry;
register short f;
{
    FOREVER {
        if (f < tree->f) {
            if (tree->left == 0) {
                tree = &bintree[tree->left = bottom++];
                tree->entry = entry;
                tree->left = 0;
                tree->right = 0;
                tree->f = f;
                return;
            } else {
                tree = &bintree[tree->left];
            }
        } else {
            if (tree->right == 0) {
                tree = &bintree[tree->right = bottom++];
                tree->entry = entry;
                tree->left = 0;
                tree->right = 0;
                tree->f = f;
                return;
            } else {
                tree = &bintree[tree->right];
            }
        }
    }
}

void sort_table()
{
    register int i;

    bintree->f = 127;
    bintree->left = 0;
    bintree->right = 0;
    for(i = 0; i < TABLE_SIZE; i++) {
        if (table[i] > 0) {
```

```
/*
        n_puts(sprintf(msg, "%d Sorting %d\n", i, table[i]), stdout);
    */
    insert_in_tree( bintree, table[i], i );
}

void squeeze( tree, last )
register struct node *tree;
register short last;
{
    if (tree->left > 0) {
        squeeze( &bintree[tree->left], last );
    }
    if (table[tree->entry] != last) {
        last = table[tree->entry];
        if (++class_nr > group_size) {
            n_puts(sprintf(msg, "range [%d,%d] squeezed to %d\n", previous, last,
                previous = last;
                magic_number++;
                class_nr = 1;
            )
        }
        table[tree->entry] = magic_number;
        if (tree->right > 0) {
            squeeze( &bintree[tree->right], last );
        }
    }
}

void count ( tree, last )
register struct node *tree;
register short last;
{
    if (tree->left > 0) {
        count( &bintree[tree->left], last );
    }
    if (table[tree->entry] != last) {
        last = table[tree->entry];
        nr_of_classes++;
        n_puts(sprintf(msg, "class %d : %d\n", nr_of_classes, last), stdout);
    }
    if (tree->right > 0) {
        count( &bintree[tree->right], last );
    }
}

main()
{
    int node, pid, host, dimension, type = TYPE, flag, ok, packet;
    char z;
    int n;

    n_init();

    whoami( &node, &pid, &host, &dimension);

    for(packet = 0; packet < NR_OF_PACKETS; packet++) {
        nread( &table[packet*PACKET_SIZE], BUF_SIZE, &host, &type, &flag );
        nwrite( &z, 1, host, ok, &flag );
    }

    n_puts("Sorting\n", stdout);
    sort_table();
}
```

```
n_puts("Counting the classes\n",stdout);
nr_of_classes = 0;
count( &bintree[bintree->left], (short) 0);
n = nr_of_classes;
nr_of_classes = 0;
count( &bintree[bintree->right], (short) 0);
nr_of_classes += n;
group_size = (nr_of_classes / 256) + 1;

n_puts(sprintf(msg,"nr of classes is %d",nr_of_classes),stdout);
n_puts(sprintf(msg,"suggested nr of clases in one group is %d\n",group_size), std

n_puts("Squeezing\n",stdout);
squeeze( &bintree[bintree->left], (short) 0 );
squeeze( &bintree[bintree->right], (short) 0 );

n_puts("ready\n",stdout);
type = TYPE;
for(packet = 0;packet < NR_OF_PACKETS;packet++) {
    nwrite( &table[packet*PACKET_SIZE], BUF_SIZE, host, type, &flag );
    ok = -1;
    nread( &z, 1, &host, &ok, &flag );
}
}
```

87/11/05
01:02:07

squeeze.c

1

```
/*
 * squeeze.c
 *
 * This program squeezes 16 bit values into 8 bit values.
 */

#include <stdio.h>
#include "table.h"

/* external functions */

extern void sort_table();
extern void squeeze();

/* Global variables */

char data_file[] = "/usr/hahe/hough/src/tri.data";
char output_file[] = "/usr/hahe/hough/src/tri.sq";
char prog_name[] = "/usr/hahe/hough/bin/sort";
char buf[BUF_SIZE];

FILE *fd;
int table[TABLE_SIZE]; /* 16 bit values stored in table */

void fill_table()
{
    int i, entry;
    char line[256], c;

    if ((fd = fopen(data_file, "r")) == NULL) {
        printf("Can not read %s\n", data_file);
        exit(-1);
    }
    entry = 0;
    while (fscanf(fd, "%d", &table[entry++]) != EOF && entry < TABLE_SIZE)
        ;
    printf("Table size is %d, entry read is %d\n", TABLE_SIZE, entry);
    fclose(fd);
}

void store_squeezed_table()
{
    register int i;
    register int tab;

    while ((fd = fopen(output_file, "w")) == NULL) {
        printf("Can not open %s for output, enter new name : ", output_file);
        scanf("%s", output_file);
    }
    fprintf(fd, "/*\n * trigram.h\t\t\t13/10/1987\n * \n */\n\nchar table[] = {\n");
    for (i = 0; i < TABLE_SIZE; i++) {
        fprintf(fd, "\t");
        for (tab = 0; tab < 10 && i < TABLE_SIZE; tab++) {
            fprintf(fd, "%3d, ", (unsigned char) table[i++]);
        }
        fprintf(fd, "\n");
    }
    fprintf(fd, "\n};");
    fclose(fd);
}

main()
{
    int node = 0, type = TYPE, packet, *p = table, ok = OK;
```

```
int i, chan, n, size;
char c;

printf("Downloading %s to ncube array.\n", prog_name);
chan = nopen(0);
nload(chan, prog_name, node, buf, BUF_SIZE);

printf("Reading input file %s\n", data_file);
fill_table();

printf("Sending table to node\n");
printf("nr of packets is %d, packet size %d, buf size %d\n", NR_OF_PACKETS, PACKET_SIZE, BUF_SIZE);

for (packet = 0; packet < NR_OF_PACKETS; packet++) {
    nwrite(chan, &table[packet*PACKET_SIZE], BUF_SIZE, node, type);
    printf("Packet %d sent\n", packet);
    n = ok = -1;
    nread(chan, &c, 1, &n, &ok);
    printf("Packet %d received by node\n", packet);
}

printf("Squeezing in process\n");
while (ok != TYPE) {
    ok = -1;
    n = -1;
    ntest(chan, &n, &ok);
    if (ok != -1 && ok != TYPE) {
        size = nread(chan, buf, BUF_SIZE, &n, &ok);
        for (i = 0; i < size; i++) {
            putchar(buf[i]);
        }
    }
}

for (packet = 0; packet < NR_OF_PACKETS; packet++) {
    node = -1; type = -1;
    nread(chan, &table[packet*PACKET_SIZE], BUF_SIZE, &node, &type);
    printf("Packet %d received\n", packet);
    nwrite(chan, &c, 1, node, ok);
}

printf("Saving results in %s\n", output_file);
store_squeezed_table();
printf("squeeze succeeded\n");
}
```

87/11/02
19:08.08

trigram.c

1

```
/*
 * trigram.c
 */

/* Include files */

#ifdef FLISP
#include "limits.h"
#include "local.h"
#include "data.h"
#include "frwd.h"
#include "macrodefs.h"
#include "nio.h"
#include <setjmp.h>
#else
#include <stdio.h>
#endif

#include "tri.sq"

/* Constants */

#define TRI 3 /* number of symbols in a permutation */
#define ORD_V 27 /* number of symbols in alfabet */
#define ORD_H ORD_V*ORD_V*ORD_V /* size of histogram */
#define TRUE 1
#define FALSE 0

/* Macro's */

#define trans(c) (((c)=='-')? 0 : ((c) == ' ')? 26 : (c) - 'a')
#define letter(c) (((int) c == 26) ? ' ': c + 'a')
#define prompt putchar('&')
#define TOP (&bintree[0])
#define FOREVER for(;;)

/* Lisp list manipulation macro's */

#ifdef FLISP
#define START_LIST Push( NIL )
#define ADD_ELEMENT(v) (Swap(Cons( PutNumber( v ), Top())))
#define INSERT_LIST {register CELL *tmp = Pop();\
                     Swap(Cons(tmp,Top()));}

#define LIST (Pop())
#endif

/* Types */

#ifdef FLISP
typedef unsigned char BOOLEAN;
#endif

/* Global variables */

char msg[256];

struct node {
    int left;
    int right;
    FREQ f;
    char p[TRI];
} bintree[ORD_V*ORD_V];

int bottom = 0;
```

```
int count = 0;
long G[TRI] = { 1, ORD_V, ORD_V*ORD_V };
long O[TRI] = { ORD_V-1, (ORD_V-1)*ORD_V, (ORD_V-1)*ORD_V*ORD_V };
BOOLEAN unbound[TRI];
#ifdef FLISP
jmp_buf back;
#endif
/* Tables for sorting and storage of trigram histogram */

FREQ scratch[ORD_V*ORD_V]; /* maximal two symbols unbound */

/* C code */

BOOLEAN process_permutations( p )
/*
 * p is a pointer to a TRI letter word. Symbols in the word that are
 * marked with '-' are called unbound. Permutations are generated by
 * assigning all possible values in the alfabet to unbound symbols.
 */
char *p;
{
    register long permutation, i;
    char w[TRI];
    register FREQ f;
    void insert_in_tree();
    register int n = 0;

    for( i = 0; i < 3; i++) {
        w[i] = trans(p[2-i]);
        if (unbound[i] = (p[2-i] == '-')) {
            if (++n > 2) {
                return( FALSE );
            }
        }
    }
    permutation = w[0]*G[0] + w[1]*G[1] + w[2]*G[2];

    ALL_PERMUTATIONS :

    if ((f = histo[permutation]) > 0) {
        insert_in_tree( TOP, f, w );
    }
    for( i = 0; i < TRI; i++) {
        if (unbound[i]) {
            if (++w[i] == ORD_V) {
                w[i] = 0;
                permutation -= O[i];
            } else {
                permutation += G[i];
                goto ALL_PERMUTATIONS;
            }
        }
    }
    return( TRUE );
}

void insert_in_tree( tree, f, p )
register struct node *tree;
register char *p;
register FREQ f;
{
    if (bottom == 0) {
        tree->f = f;
```

87/11/02
19:08:08

trigram.c

2

```

    tree->left = 0;
    tree->right = 0;
    tree->p[0] = p[0];
    tree->p[1] = p[1];
    tree->p[2] = p[2];
    bottom++;
    return;
}
FOREVER {
    if (f > (tree->f)) {
        if (tree->left == 0) {
            tree = &bintree[tree->left = bottom++];
            tree->p[0] = p[0];
            tree->p[1] = p[1];
            tree->p[2] = p[2];
            tree->left = 0;
            tree->right = 0;
            tree->f = f;
            return;
        } else {
            tree = &bintree[tree->left];
        }
    } else {
        if (tree->right == 0) {
            tree = &bintree[tree->right = bottom++];
            tree->p[0] = p[0];
            tree->p[1] = p[1];
            tree->p[2] = p[2];
            tree->left = 0;
            tree->right = 0;
            tree->f = f;
            return;
        } else {
            tree = &bintree[tree->right];
        }
    }
}

print_tree( tree )
register struct node *tree;
{
    register int i;

    if (tree->left > 0) {
        print_tree( &bintree[tree->left] );
    }
#ifdef FLISP
    if (count == 0) {
        longjmp( back, TRUE );
    }
    count--;
    START_LIST;
    for( i = 0; i < 3; i++) {
        if (unbound[i]) {
            ADD_ELEMENT( (int) letter(tree->p[i]) );
        }
    }
    INSERT_LIST;
#else
    printf("%c%c%c %d\n", tree->p[2]+'a', tree->p[1]+'a', tree->p[0]+'a', tree->f);
#endif
    if (tree->right > 0) {
        print_tree( &bintree[tree->right] );
    }
}

```

```

    }
}

#ifdef FLISP
CELL *Trigram( p, k )
register char *p;
register int k;
{
    register CELL *reverse, *previous, *tmp;

    bottom = 0;
    if (process_permutations( p )) {
        count = k;
        START_LIST;
        if (!setjmp( back ))
            print_tree( TOP );
        reverse = LIST;
        previous = NIL;
        while (reverse != NIL) {
            tmp = Cdr( reverse );
            SetCdr( reverse, previous );
            previous = reverse;
            reverse = tmp;
        }
        return( previous );
    }
    return( NIL );
}

#else
main()
{
    char w[10];
    register struct node *tree;
    register long permutation;

    prompt;
    while(gets(w) != NULL) {
        if (*w == '@') {
            permutation = trans(w[1])*G[0] + trans(w[2])*G[1] + trans(w[3])*G[2];
            printf("%ld %d\n", permutation, histo[permutation]);
            prompt;
            continue;
        }
        process_permutations(w);
        print_tree( TOP );
        tree = TOP;
        while(tree->left) {
            tree = &bintree[tree->left];
        }
        printf("Max is %c%c%c %d\n", tree->p[2]+'a', tree->p[1]+'a', tree->p[0]+'a', bottom);
        printf("Number of elements in tree is %d\n", bottom);
        nr_unbound = bottom = 0;
        prompt;
    }
}
#endif

```

87/11/05
00:06:40

```

/*
 * dbase.c
 *
 * 2/11/1987
 *
 * Statistics concerning specific atari font are initialized in a header file.
 * Table can be accessed by a lisp function
 */

/*
 * FLISP include files
 */

#include "limits.h"
#include "local.h"
#include "data.h"
#include "frwd.h"
#include "macrodefs.h"
#include "stdio.h"

/*
 * Include Files
 */

#include "system.h"

#define NUM_ENTRIES 70
#define RECORD_SIZE (9*sizeof(char))

/*
 * Macro's
 */

#define report(s) string_out(s,stdout);newline(stderr)
#define report_int(s,g) string_out(sprintf(report_msg,"%s : %d\n",s,g),stderr)

#define START_LIST Push( NIL )
#define ADD_ELEMENT(v) (Swap(Cons( PutNumber( v ), Top())) )
#define INSERT_LIST {register CELL *tmp = Pop();\
                    (tmp != NIL) ? Swap(Cons(tmp,Top())) : Top();}

#define LIST {Pop()}
#define CORRECT {Pop();Push( NIL );}

#define abs(a) ((a < 0) ? (-a) : a )

/*
 * Global Variables
 */

char variation[9] = {
    0, /* ascii value */
    0, /* nr of lines */
    0, /* horizontal */
    0, /* vertical */
    0, /* positive */
    0, /* negative */
    0, /* average */
    0, /* minimum */
    0, /* maximum */
};

/*
 * C Code
 */

```

dbase.c

```

*/

CELL *scan_font(type, template)
register int type;
register char *template;
{
    register int i, d;
    register char *font, *p, *t, *v;

    /*
     * Determine font type
     */

    switch (type) {
        case 0:
            font = system;
            break;
        default:
            /* nothing else yet */
            return (NIL);
    }

    /*
     * Walk font table, filling in unbound entries in template
     */

    START_LIST;

    for (i = 0; i < NUM_ENTRIES; i++) {
        p = &font[i * RECORD_SIZE] + 8;

        START_LIST;

        for (t = template + 8, v = variation + 8;
             t >= template;
             t--, p--, v--) {
            /*
             * Check if template entry is unbound
             */

            if (*t == -1) {
                /*
                 * If template field is unbound then add table entry value to
                 * result list
                 */

                ADD_ELEMENT(*p);
            } else {
                /*
                 * If template field was not unbound then check if the field
                 * is within allowable variation of the table value
                 */

                if (*p != *t) {
                    FORGET;
                    break;
                }
            }
        }
    }
}

```


87/11/05
00:06:40

dbase.c

2

```
    }  
    INSERT_LIST;  
}  
return (LIST);  
}
```

87/11/02
19:12:35

window.h

1

```
/*
 * window.h      version 1.0      8/7/1987
 *
 * window.h contains macro's and constants that are used by the window.o
 * library. The control codes are based on the ansi-standard.
 *
 * J. Henseler, J.C. Scholtes and C.R.J. Verhoest
 * Delft University of Technology
 */

#define FOREVER for(;;)
#define BUFSIZE 1024

#define row(x)      ((0 < (x) && (x) < 26))
#define column(y)   ((0 < (y) && (y) < 81))
#define w_putchar(c,w) w_putc((c), &_window[w])
#define w_putstr(s,w) w_puts((s), &_window[w])
#define video_attribute(x,w) vid_attr((x), &_window[w])
#define message(p,x,w) w_msg(p, (x), &_window[w])
#define window(x)   (&_window[x])

#define BLANK      "
"
#define MAX_W      16

#define NORMAL      0
#define BOLD        1
#define UNDERLINE   4
#define BLINKING    5
#define REVERSE     7

#define RIGHT       0
#define LEFT        1
#define MIDDLE      2
#define ALL         3

struct _wndwctrblk {
    int _upper;      /* number of top row in scroll region */
    int _lower;      /* number of bottom row in scroll region */
    int _place;      /* cursor position on bottom line of the window */
    int _attri;      /* video attribute for window */
};

#define WINDOW struct _wndwctrblk

WINDOW _window[MAX_W], *crrntw;
```

87/10/31
16:20:10

window.c

1

```
/*
 * window.c
 *
 * Delft University of Technology
 * Julianalaan 132
 * 2628 BL Delft
 * phone : 015-785245
 *
 * Window.c contains functions that can be used to create a multidisplay
 * screen. To use these functions link your programobject with
 * window.o and make sure that window.h is included.
 */

#include "window.h"

char *WNDW_NR = "Multi Display System version 1.0 27/8/1987";

WINDOW _window[MAX_W];

char code[50];

void w_puts(),
w_putc();

void execute(p)
char *p;
{
    int s;
    char *ptr;

    for(s = 0, ptr = p; *ptr != '\0'; ptr++, s++);
    write(1, p, s);
}

void reset_terminal()
{
    write(1, "\033[r\033[f\033[J", 9);
}

void select_keypad()
{
    write(1, "\033=", 1);
}

void set_ansi_mode()
{
    write(1, "\033{?2h", 5);
}

void set_scrolling_region(n, m)
int n, m;
{
    if (!row(n) || !row(m))
        return;
    sprintf(code, "\033[%d;%dr", n, m);
    execute(code);
}

void move_cursor_to(y, x)
int x, y;
{
    if (!row(x) || !column(y))
        return;
    sprintf(code, "\033[%d;%df", x, y);
}
```

```
execute(code);
}

void erase_display()
{
    move_cursor_to(1, 1);
    write(1, "\033[J", 4);
}

void vid_attr(att, wptr)
WINDOW *wptr;
{
    wptr->_attri = att;
}

void initialize_windows(n, tr, br)
int n, tr, br;
{
    WINDOW *wptr;
    int size, upper;

    if (!row(tr) || !row(br) || n > MAX_W || tr > br)
        return;

    size = (br - tr + 1) / n;
    for(upper = tr, wptr = _window; wptr < _window + n; wptr++) {
        wptr->_upper = upper;
        wptr->_lower = upper + size - 2;
        wptr->_place = 1;
        wptr->_attri = 0;
        upper += size;
    }
    w_putstring(WNDW_NR, 0);
    w_putchar('\n', 0);
}

void w_putc(c, wptr)
char c;
WINDOW *wptr;
{
    if (c == '\b') {
        if (wptr->_place > 0) {
            move_cursor_to(--(wptr->_place), wptr->_lower);
            write(1, " ", 1);
            move_cursor_to(wptr->_place, wptr->_lower);
        }
        return;
    }

    sprintf(code, "\033[%dm", wptr->_attri);
    write(1, code, 4);

    set_scrolling_region(wptr->_upper, wptr->_lower);
    move_cursor_to(wptr->_place, wptr->_lower);
    if (c == '\n')
        wptr->_place = 1;
    else if (wptr->_place == 80) {
        write(1, "\n", 1);
        wptr->_place = 1;
    } else
        wptr->_place++;

    write(1, &c, 1);
    return;
}
```

```
void w_puts(s,wptr)
char *s;
WINDOW *wptr;
{
    while(*s != '\0')
        w_putc(*s++,wptr);
}

void w_msg(z,s,wptr)
char *s;
WINDOW *wptr;
int z;
{
    move_cursor_to((z == 2)?28:(z == 1)?1:55,wptr->_lower + 1);
    sprintf(code, "\033[7m%27s\033[m", s);
    execute(code);
}
```

87/10/31
16:21:53

usrgraph.h

1

```
/*  
 * Usrgraph.h  
 */  
  
#include "hough.h"  
#include "graph.h"  
  
static CELL *v, *w;  
  
CELL *UserEdgtover( arg )  
register CELL *arg;  
{  
    if (GetArguments( 1, arg, &v ) != 0)  
        return( NIL );  
  
    return(PutNumber((int)edge_to_vertex( v )));  
}  
  
CELL *UserSubgraph( arg )  
register CELL *arg;  
{  
    if (GetArguments( 1, arg, &v ) != 0)  
        return( NIL );  
  
    return(get_subgraph( v ));  
}  
  
CELL *UserGclear()  
{  
    clear_graph();  
    return( T );  
}  
  
CELL *UserAllcycles()  
{  
    return(all_cycles());  
}  
  
CELL *UserLstver()  
{  
    return(list_vertices());  
}  
  
CELL *UserGetver( arg )  
register CELL *arg;  
{  
    if (GetArguments( 1, arg, &v ) != 0)  
        return( NIL );  
  
    return(get_vertex( GetNumber( v )));  
}  
  
CELL *UserAdj( args )  
register CELL *args;  
{  
    if (GetArguments( 2, args, &v, &w ) != 0)  
        return( NIL );  
  
    return(is_connected( GetNumber( v ), GetNumber( w )));  
}  
  
CELL *UserLstadj( arg )  
register CELL *arg;  
{
```

```
    if (GetArguments( 1, arg, &v ) != 0)  
        return( NIL );  
  
    return(get_adj( GetNumber( v )));  
}
```

87/10/31
16:22:01

usrhough.h

1

```
/*
 * usrhough.h                21/9/1987
 *
 * Performs argument handling for implemented functions :
 */

#include "hough.h"

#define inside(Hu,Bb) ((0 <= Hu.x)&&(Hu.x < Bb.x)&&(Bb.x < X_DIM)&&(0 <= Hu.y)&&(Hu.y < B

CELL *UserHough( args )
register CELL *args;
{
  CELL *area, *ll, *rc;
  POINT Hu, Bb;

  if(GetArguments( 3, args, &area, &ll, &rc) != 0) {
    return( NIL );
  }
  LLENGTH = GetNumber( ll );
  RC_MAX = (long) GetNumber( rc );
  Hu.x = GetNumber(Car(area));
  Hu.y = GetNumber(Cadr(area));
  Bb.x = GetNumber(Car(Cddr(area)));
  Bb.y = GetNumber(Cadr(Cddr(area)));

  if (!inside( Hu, Bb ))
    return( NIL );

  return(perform_hough( &Hu, &Bb ));
}
```

87/10/31
16:22:08

usrsplit.h

1

```
/*
 * usrsplit.h                27/10/1987
 */

#include "split.h"

#define inside(Hu,Bb) ((0 <= Hu.x)&&(Hu.x < Bb.x)&&(Bb.x < X_DIM)&&(0 <= Hu.y)&&(Hu.y < B

CELL *UserSplit( args )
register CELL *args;
{
    CELL *area;
    POINT Hu, Bb;

    if(GetArguments( 1, args, &area ) != 0){
        return( NIL );
    }
    Hu.x = GetNumber( Car( area ) );
    Hu.y = GetNumber( Cadr( area ) );
    Bb.x = GetNumber( Car( Cddr( area ) ) );
    Bb.y = GetNumber( Cadr( Cddr( area ) ) );

    if ( !inside( Hu, Bb ) )
        return( NIL );

    return( do_split( &Hu, &Bb ) );
}

CELL *UserGetWindow( arguments )
CELL *arguments;
{
    return( get_current_window() );
}

CELL *UserOrigin( arguments )
CELL *arguments;
{
    return( get_origin() );
}
```

87/10/31
16:22:19

usrtri.h

1

```
/*
 * usrtri.h                      26/10/1987
 *
 * This file should be included by user.c
 */

/* Macro definitions */

#define invalid(c)    ((c < 'a' || c > 'z') && c != '-' && c != ' ')

/* External functions */

CELL *Trigram();

/* Variables */

static char perm[4];

/* C Code */

CELL *UserTrigram( args )
register CELL *args;
{
    register int n, i;
    CELL *pattern_list, *nr_of_elements;
    int pattern;

    if(GetArguments( 2, args, &pattern_list, &nr_of_elements ) != 0) {
        string_out("TRIGRAM : Need two arguments\n",stderr);
        return( NIL );
    }
    if (!(TestFlag(L,pattern_list) && TestFlag(N,nr_of_elements))) {
        string_out("TRIGRAM : Wrong arguments\n",stderr);
        return( NIL );
    }
    for( i = 0; i < 3 && pattern_list != NIL; i++ ) {
        if (!TestFlag(N,Car(pattern_list))) {
            string_out("TRIGRAM : Pattern contains false element\n",stderr);
            return( NIL );
        }
        pattern = GetNumber(Car( pattern_list ));
        perm[i] = (pattern == 0) ? '-' : (char) pattern;
        if invalid(perm[i]) {
            string_out("TRIGRAM : Invalid character\n",stderr);
            return( NIL );
        }
        pattern_list = Cdr( pattern_list );
    }

    n = GetNumber( nr_of_elements );

#ifdef DEBUG
    perm[3] = '\0';
    string_out( perm, stderr );
#endif

    return( Trigram( perm, n ));
}
```


87/11/02
19:17:06

spelcor.c

1

```
/*
 *
 *
 * PROGRAM TO CORRECT THE SPELLING OF WORDS RECEIVED FROM THE NCube
 *
 * J.C. Scholtes
 *
 * september 1987
 */

/*
 *
 * The program first reads the words of the file words, these are put into
 * a hash-table, then the words to be corrected are read from the file
 * check.words, if the word is correct nothing happens, else, the word
 * is permuted within a confusion group, the prefixes are stripped, the
 * suffixes are stripped, and the resulting stamwords are checked.
 * If a word is found, the original word is put into the file with good
 * words, if the word is not found, it is put in the final file with a
 * asterix after it.
 */

#include <stdio.h>
#include "malloc.h"

#define WORD_LENGTH 20 /* length of a string */
#define TABLE_SIZE 58791 /* the hash-table size = prime number */
#define MAX_PRIME 59 /* length of the list of prime numbers */
#define HASH_OVERFLOW 10 /* number of recalculations in hash table fill */
#define PERMUTATION 6 /* */
#define NUMBER_PERMUTATION 4 /* maximum number of characters to be permuted */

/*
 * this is the structure recieved ot the cube, a string and a bitmap of
 * the word
 */

typedef struct cube {
    char *s;
    int bitmap[WORD_LENGTH];
} CUBE;

CUBE input_cube;

/*
 * this is a string, used in various variables
 */

typedef struct words {
    char *s;
} WORDS;

WORDS hash[TABLE_SIZE];

WORDS suffix_string[2];

typedef struct pre {
    char *s;
```

```

} PRE;

PRE prefix[208];

typedef struct suf {
    char *s;
} SUF;

SUF suffix[260][3];

int prime[MAX_PRIME];
int word_number;
int max_ascii;

FILE *file_ptr,*write_ptr;

char *string;
char *prefix_string;

/*
 * in this procedure the initial hashvalue of a word is calculated
 * the word is common known as 'string', the value is returned.
 */

int ascii(string)
char *string;
{
    int i,value;

    i=0;
    value =0;

    while(string[i] != '\0'){
        value += ((string[i++]-'0')*prime[i]);
    }

    if (value > max_ascii){
        max_ascii = value;
    }

    return(value);
}

/*
 * This procedure fills the hash-table. The string to be filled is in string,
 * if the first address is full, a new one is calculated in the second
 * part of the procedure. If now place is left after 10 retries the
 * function returns a -1 and the program is stopped
 */

int fill_hash(string)
char *string;
{
    char *t;
    int deler,som,i,hash_value,word_hash,length;

    hash_value = (som = ascii(string)) % TABLE_SIZE;
    deler = som / TABLE_SIZE;
```

```

    if((t = hash[hash_value].s) == NULL){
        t = (malloc(strlen(string)));
        for(i=1;i<strlen(string);i++) *t++ = '\0';
        t = t - (i);
        length = 0;
        while(*t++ = *string++) length++;
        hash[hash_value].s = t-(length+1);
        return(hash_value);
    }
    else{
        for(i=1;i<HASH_OVERFLOW;i++){
            hash_value = (deler*prime[i]*prime[i]) + prime[i]
            if((t = hash[hash_value].s) == NULL){
                t = (malloc(strlen(string)));
                for(i=1;i<strlen(string);i++) *t++ = '\0';
                t = t - (i);
                length = 0;
                while(*t++ = *string++) length++;
                hash[hash_value].s = t-(length+1);
                return(hash_value);
            }
        }
        return(-1);
    }
}

/*
 * In this procedure a string is checked to be in the hash-table, if the
 * string is found, a 0 is returned, otherwise a 1 is returned
 */

int check_hash(string)

char *string;
{
    int i, hash_value, deler, som;

    hash_value = (som = ascii(string)) % TABLE_SIZE;
    deler = som / TABLE_SIZE;

    if(hash[hash_value].s == NULL) {
        return(-1);
    }
    else{
        if(strcmp(hash[hash_value].s, string) == 0){
            return(0);
        }
        else{
            for(i=1;i<HASH_OVERFLOW;i++){
                hash_value = (hash_value + (deler*prime[i]*prime[i]) + pr
                if(hash[hash_value].s == NULL){
                    return(-1);
                }
                else{
                    if(strcmp(hash[hash_value].s, string) == 0){
                        return(0);
                    }
                }
            }
            return(-1);
        }
    }
}

```

```

/*
 * here, a list of primenumbers is generated, starting with a 1 in the
 * highest table entry. The numbers go as follows: 1 3 5 7 11 .....
 * By using this method, the first letter of a word is multiplied
 * with the highest prime number in the function ascii(string).
 */

int int_prime()
{
    int found, prime_number, i, p;

    prime_number = 4;
    prime[MAX_PRIME-1] = 1;
    prime[MAX_PRIME-2] = 3;

    for(i=(MAX_PRIME-3);i>=0;i--){
        found = 0;
        prime_number++;
        while(found == 0){
            p = 2;
            while(((prime_number % p) != 0) && (p < prime_number)) p++;
            if(p == prime_number){
                prime[i] = prime_number;
                found = 1;
            }
            else{
                prime_number++;
            }
        }
    }
    return(0);
}

/*
 * This function reads the string from the file words, it is also used
 * to read the pre and suffixes from the corresponding files.
 * In case of EOF, a -1 is returned. The string read in is put in
 * the variable string.
 */

int read_string()
{
    int i;
    char c;

    for(i=0;(i<WORD_LENGTH) && (string[i] != '\0');i++) string[i] = '\0';

    i = 0;
    while(((c=getc(file_ptr)) != EOF) && (c != '\012')){
        string[i++] = c;
    }
    if (c == EOF) {
        return(-1);
    }
    else{
        return(0);
    }
}

```

87/11/02
19:17:06

spelcor.c

3

```

/*
 * This function determines if a word is correct recognised on the
 * cube (e.g. all the values in the bitmap are 0). If it is correct
 * a zero is returned, otherwise a -1 is returned.
 */

int word_good()
{
    int i;

    for(i=1;i<WORD_LENGTH;i++){
        if(input_cube.bitmap[i] == 1) return(-1);
    }
    return(0);
}

/*
 * This function generates a number of permutations within a certain
 * confusion group, once a permutation is found, the pre- and suffixes
 * are stripped, and the words are checked. If a good word is found,
 * the function return a 0 (This does not mean that the word found is the
 * only possible good word. If the word is wrong, the same procedure
 * is repeated for the following permutations. If no good word is
 * found at all, the function returns -1. The word to be corrected
 * is found in input_cube.s, if it is coorrected, the word is places in the
 * same variable.
 */

int correct_word()
{
    int old, m, i, j, k, l, number_permuted;
    char word[WORD_LENGTH][PERMUTATION];
    int wrong[WORD_LENGTH];
    char check[WORD_LENGTH];
    char c[WORD_LENGTH];
    int perm[NUMBER_PERMUTATION];

    for(i=0;i<NUMBER_PERMUTATION;i++) perm[i] = 0;

    for(i=0;i<WORD_LENGTH;i++){
        wrong[i] = 0;
    }

    for(i=0;i<WORD_LENGTH;i++){
        for(j=0;j<PERMUTATION;j++){
            word[i][j] = '\0';
        }
    }

    for(i=0;i<WORD_LENGTH;i++){
        check[i] = input_cube.s[i];
    }

    for(i=0;i<WORD_LENGTH;i++){
        string[i] = '\0';
        c[i] = '\0';
    }
    j = 0;
    for(i=0;i<WORD_LENGTH;i++){

```

```

    if(input_cube.bitmap[i] == 1){
        c[i] = input_cube.s[i];
        wrong[j++] = i;
        if(j>NUMBER_PERMUTATION) return(-1);
    }
}

number_permuted = old =0;

for(i=0;i<WORD_LENGTH;i++){
    if(c[i] == '[' || c[i] == 'i' || c[i] == ']' || c[i] == 'l' || c[i] == '1')
        word[i][0] = 'l';
        word[i][1] = 'i';
        number_permuted++;
    }
    if(c[i] == 'b' || c[i] == 'p' || c[i] == '6'){
        word[i][0] = 'b';
        word[i][1] = 'p';
        number_permuted++;
    }
    if(c[i] == '8' || c[i] == 'B' || c[i] == 'S'){
        word[i][0] = 'B';
        word[i][1] = 'S';
        number_permuted++;
    }
    if(c[i] == 's' || c[i] == 'e' || c[i] == 'g' || c[i] == 'a'){
        word[i][0] = 's';
        word[i][1] = 'e';
        word[i][2] = 'g';
        word[i][3] = 'a';
        number_permuted++;
    }
    if(c[i] == 'd' || c[i] == 'q'){
        word[i][0] = 'd';
        word[i][1] = 'q';
        number_permuted++;
    }
    if(c[i] == 'Q' || c[i] == 'O' || c[i] == '0'){
        word[i][0] = 'Q';
        word[i][1] = 'O';
        number_permuted++;
    }
    if((c[i] != '\0') && (number_permuted == old)){
        word[i][0] = c[i];
        number_permuted++;
        old = number_permuted;
    }
    else{
        old = number_permuted;
    }
}

i = 0;

while(i<number_permuted){
    for(i=0;i<number_permuted;i++){
        check[wrong[i]] = word[wrong[i]][perm[i]];
    }
    strip_prefix(check);
    strip_suffix(prefix_string);
    for(j=0;j<2;j++){
        if(check_hash(suffix_string[j].s) == 0){
            for(m=0;m<WORD_LENGTH;m++){
                input_cube.s[m] = check[m];
            }

```

```

        return(0);
    }
    i = 0;
carry:
    perm[i]++;
    if (word[wrong[i]][perm[i]] == '\0') {
        perm[i++] = 0;
        if (i < number_permuted) {
            goto carry;
        }
    }
    return(-1);
}

/*
 * In this function the suffixes are stripped, they are address in the
 * table suffix, hashed on the starting character.
 * The variable is found in prefix_string, and put into suffix_string
 * Because of the fact that it is possible to have more then one
 * word resulting, but no more then 2, this variable has 2 entries.
 * The word without a possible prefixes is stored in the variable
 * suffix_string.
 */

int strip_suffix(prefix_string)
char *prefix_string;
{
    int temp, temp2, length, suffix_length, i, j, address, wrong, found;
    char *revstr;

    for (i=0; i<2; i++) {
        for (j=0; j<WORD_LENGTH; j++) {
            suffix_string[i].s[j] = '\0';
        }
    }

    revstr = malloc(WORD_LENGTH);
    for (i=0; i<WORD_LENGTH; i++) revstr[i] = '\0';

    length = strlen(prefix_string);
    for (i=0; i<length; i++) revstr[length-i-1] = prefix_string[i];

    address = (revstr[0] - 'a') * 10;

    if (suffix[address][0].s == NULL) {
        for (i=0; i<WORD_LENGTH; i++) suffix_string[0].s[i] = prefix_string[i];
        return(0);
    }
    else {
        while ((suffix[address][0].s != NULL) && (suffix[address][0].s[0] == revst
            i = 0;
            wrong = 0;
            while ((suffix[address][0].s[i] != '\0') && (!wrong)) {
                if (revstr[i] == suffix[address][0].s[i]) {
                    i++;
                }
                else {
                    wrong = 1;
                }
            }
        }
    }
}

```

```

        }
        if (wrong == 0) {
            j = 1;
            suffix_length = 1;
            while ((j<3) && (suffix[address][j].s != NULL)) {
                temp = length - suffix_length;
                for (i=0; i<temp; i++) {
                    suffix_string[j-1].s[i] = prefix_string[i]
                }
                if (strcmp(suffix[address][j].s, "&") != 0) {
                    temp = length - suffix_length;
                    temp2 = temp + strlen(suffix[address][j].s);
                    for (i=temp; i<temp2; i++) {
                        suffix_string[j-1].s[i] = suffix[
                    }
                }
                j++;
            }
            return(0);
        }
        address++;
    }
    for (i=0; i<WORD_LENGTH; i++) {
        suffix_string[0].s[i] = prefix_string[i];
    }
    return(0);
}

/*
 * in this procedure a prefix is stripped, the word to be stripped
 * is found in the variable check, the stripped word is put into
 * the variable prefix_string
 */

int strip_prefix(check)
char *check;
{
    int i, j, address, wrong;

    address = (check[0] - 'a') * 8;

    for (i=0; i<WORD_LENGTH; i++) prefix_string[i] = '\0';

    if (prefix[address].s == NULL) {
        for (i=0; i<WORD_LENGTH; i++) prefix_string[i] = check[i];
        return(0);
    }
    else {
        while ((prefix[address].s != NULL) && (prefix[address].s[0] == check[0]) {
            i = 0;
            wrong = 0;
            while ((prefix[address].s[i] != '\0') && (!wrong)) {
                if (check[i] == prefix[address].s[i]) {
                    i++;
                }
                else {
                    wrong = 1;
                }
            }
            if (prefix[address].s[i] == '\0') {
                j = 0;
                while (check[j] != '\0') {

```

8/11/02
19:17:06

5

spelcor.c

```

        prefix_string[j++] = check[i++];
    }
    return(C);
}

for(i=0;i<WORD_LENGTH;i++) prefix_string[i] = check[i];
return(0);
}

/*
 * This function reads the input from the cube. It reads a string
 * first, and then a bitmap. The values are stored in input_cube.s
 * and input_cube.bitmap
 * In case of EOF, a -1 is returned.
 */

int read_cube()
{
    int i;
    char c;

    for(i=0;i<WORD_LENGTH;i++) {
        input_cube.s[i] = '\0';
        input_cube.bitmap[i] = 0;
    }

    i = 0;
    while(((c=getc(file_ptr)) != EOF) && (c != '\012')){
        input_cube.s[i++] = c;
    }
    i = 0;
    if (c!= EOF){
        while(((c=getc(file_ptr)) != EOF)&&(c!='\012')){
            input_cube.bitmap[i++] = c-48;
        }
    }
    if (c == EOF) {
        return(-1);
    }
    else{
        return(0);
    }
}

/*
 * This procedure reads the prefixes from the file prefix, and
 * then stores them in the table prefix. The words are hashed on
 * the first character.
 */

int fill_prefix(string)
char *string;
{
    int found,j, length, i,address;
    char *t;

    found = j = 0;
    address = (string[0] - 'a')*8;

    while(j++ < 8){
        if((t = prefix[address].s) == NULL){

```

```

            t = malloc(strlen(string));
            for(i=0;i<strlen(string);i++) *t++ = '\0';
            t = t - i;
            length = 0;
            while(*t++ = *string++) length++;
            prefix[address].s = t - (length+1);
            return(address);
        }
        else{
            address++;
        }
    }
    return(-1);
}

/*
 * This procedure fills the suffix table by reading them from the
 * file suffix. The suffix is places in entry 0, the first possible
 * solution in entry 1 and if there is a second option, this is placed
 * in entry 2. An & is put in the table if the suffix got to be
 * stripped.
 */

int fill_suffix()
{
    int i,number, length, address,found,retry;
    char *revstr;

    revstr = malloc(WORD_LENGTH);

    while(read_string() != -1){
        number = 0;
        length = strlen(string);
        for(i=0;i<WORD_LENGTH;i++) revstr[i] = '\0';
        for(i=0;i<length;i++) revstr[length-i-1] = string[i];
        address = (revstr[0] - 'a')*10;
        retry = 0;
        found = 0;
        while(!found) && (retry < 10){
            if(suffix[address][0].s != NULL){
                address++;
                retry++;
            }
            else{
                found = 1;
            }
        }
        if(found == 0) {
            return(-1);
        }

        for(i=0;i<strlen(string);i++){
            string[i] = revstr[i];
        }

        while(strcmp(string,"") != 0){
            suffix[address][number].s = malloc(WORD_LENGTH);
            for(i=0;i<WORD_LENGTH;i++){
                suffix[address][number].s[i] = '\0';
            }
            for(i=0;i<WORD_LENGTH;i++){
                suffix[address][number].s[i] = string[i];
            }
            number++;
        }
    }
}

```

87/11/02
19:17:06

spelcor.c

6

```
        read_string();
    }
    return(0);
}

/*
 * The program first reads the words of the file words, these are put into
 * a hash-table, then the words to be corrected are read from the file
 * check.words, if the word is correct nothing happens, else, the word
 * is permuted within a confusion group, the prefixes are stripped, the
 * suffixes are stripped, and the resulting stamwords are checked.
 * If a word is found, the original word is put into the file with good
 * words, if the word is not found, it is put in the final file with a
 * asterix after it.
 */

main()
{
    int i, hash_value, j;
    char c;
    char *t;

    string = malloc(WORD_LENGTH);
    input_cube.s = malloc(WORD_LENGTH);
    prefix_string = malloc(WORD_LENGTH);

    for(i=0; i<2; i++){
        suffix_string[i].s = malloc(WORD_LENGTH);
    }

    max_ascii = 0;

    word_number = 0;

    int_prime();

    if((file_ptr = fopen("prefix", "r")) == NULL){
        printf("File for prefixes not found\n");
        exit(1);
    }

    printf("Reading prefixes\n");

    while(read_string() != -1){
        if(fill_prefix(string) == -1){
            printf("Error occurred during prefix table fill.\n");
            exit(1);
        }
    }

    fclose(file_ptr);

    if((file_ptr = fopen("suffix", "r")) == NULL){
        printf("File for suffixes not found.\n");
        exit(1);
    }
}
```

```
printf("Reading suffixes.\n");

if(fill_suffix() == -1){
    printf("Error occurred during suffix table fill.\n");
    exit(1);
}

fclose(file_ptr);

if((file_ptr = fopen("words", "r")) == NULL){
    printf("File with words not found\n");
    exit(1);
}

printf("Reading words into hash table.\n");

while(read_string() != -1) {
    if( (fill_hash(string)) == -1){
        printf("error occurred during table fill\n");
        printf("at word : %d with final hashvalue %d\n", word_number, ascii);

        printf("%d = max ascii\n", max_ascii);

        exit(1);
    }
    word_number++;
}

printf("Table is filled successfully with %d words.\n", word_number);
fclose(file_ptr);

if((file_ptr = fopen("check.words", "r")) == NULL){
    printf("file to be checked not found\n");
    exit(1);
}

write_ptr = fopen("good.words", "w");

word_number = 0;

while(read_cube() != -1) {
    if(word_good() == 0){
        fprintf(write_ptr, "%s\n", input_cube.s);
    }
    else{
        if(correct_word() == 0){
            fprintf(write_ptr, "%s\n", input_cube.s);
        }
        else{
            fprintf(write_ptr, "%s * \n", input_cube.s);
        }
    }
    word_number++;
}

fclose(file_ptr);
fclose(write_ptr);

printf(" %d words checked succesfully \n", word_number);

/*

for(i=0; i<TABLE_SIZE; i++){
    if((t = hash[i].s) != NULL){
        printf("%d %s ", i, hash[i].s);
    }
}
```

87/11/02
19:17:06

```
printf("\n");
```

```
}
```

```
}
```

```
*/
```

spelcor.c

7

87/10/28
19:13:32

tricube.c

1

```
#include <stdio.h>
/*
 * PROGRAM TO CALCULATE TRIGRAM FREQUENCIES AND MAKING
 * OUTPUT FIT FOR THE NCUBE COMPUTER
 * ALL TABEL ENTRIES ARE GENERATED, ALSO THE ONES EQUAL 0
 *
 * J.C. Scholtes
 *
 * september 1987
 */

/*
 * The program first reads a word from a file, the trigram
 * values are determined by the formula: address = char 1 *
 * 27 **2 + char2 * 27 + char3 . The table has room for
 * 27 characters, including the space. The two spaces in front
 * and the two spaces at the end are also included.
 * After the table generation, the required output is generated
 * into the standard output channel.
 */

#define NR_CHAR 27
#define MAX_TABLE (NR_CHAR*NR_CHAR*NR_CHAR)
#define PROB_NOT_NIL 6000
#define SPACE 'z'+1
#define MAX_VALUE 28159

int table[MAX_TABLE];

main()
{
    int i, p, j, regel, breedte;
    int index;
    char c, s[256], tempchar[PROB_NOT_NIL][3];

    for(i=0; i<MAX_TABLE; i++) table[i] = 0;

    while( (c= getchar()) != EOF){
        s[0] = SPACE;
        s[1] = SPACE;
        s[2] = c;

        i = 3;

        while( (c = getchar()) != '\012' && c != EOF){
            s[i] = c;
            i++;
        }
        s[i] = SPACE;
        s[i+1] = SPACE;
        s[i+2] = '\0';
        i = 0;

        while(s[i+2] != '\0'){
            index = ((s[i]- 'a')*NR_CHAR*NR_CHAR)+((s[i+1] - 'a')*NR_CHAR)+(s[
                (table[index])++;

            i++;
        }
    }
}
```

```
/*
 * Here the table is printed
 */
breedte = 0;
for(i=0; i<MAX_TABLE; i++){
    printf("%d\n", table[i]);
}
}
```


87/10/28
19:16:35

trilodus.c

1

```
#include <stdio.h>
/*
 * PROGRAM TO CALCULATE TRIGRAM FREQUENCIES AND MAKING
 * OUTPUT FIT FOR THE LOTUS SPREADSHEET
 * TABLE ENTRIES ARE GENERATED WHICH ARE UNEQUAL 0
 *
 * J.C. Scholtes
 *
 * september 1987
 */

/*
 * The program first reads a word from a file, the trigram
 * values are determined by the formula: address = char 1 *
 * 27 **2 + char2 * 27 + char3 . The table has room for
 * 27 characters, including the space. The two spaces in front
 * and the two spaces at the end are also included.
 * After the table generation, the required output is generated
 * into the standard output channel.
 */

#define NR_CHAR 27
#define MAX_TABLE (NR_CHAR*NR_CHAR*NR_CHAR)
#define PROB_NOT_NIL 6000
#define SPACE 'z'+1

int table[MAX_TABLE];
int temp[PROB_NOT_NIL];

main()
{
    int i, p, j, regel, breedte;
    int index;
    char c, s[256], tempchar[PROB_NOT_NIL][3];

    for(i=0; i<MAX_TABLE; i++) table[i] = 0;

    while( (c= getchar()) != EOF) {
        s[0] = SPACE;
        s[1] = SPACE;
        s[2] = c;

        i = 3;

        while( (c = getchar()) != '\012' && c != EOF) {
            s[i] = c;
            i++;
        }
        s[i] = SPACE;
        s[i+1] = SPACE;
        s[i+2] = '\0';
        i = 0;

        while(s[i+2] != '\0') {
            index = ((s[i]- 'a') * NR_CHAR * NR_CHAR) + ((s[i+1] - 'a') * NR_CHAR) + (s[
                (table[index])++;
            }
        }
    }
}
```

```
        i++;
    }
}

/*
 * generation of the output
 */

printf("ICCA TRIGRAMVERDELINGEN VOOR LOTUS INPUT\n\n\n");
s[0]=s[1]=s[2]='a';
for(i=0; i<MAX_TABLE; i++) {
    if(table[i] != 0) {
        printf("\'%c%c%c\' %d\n", s[0], s[1], s[2], table[i]);
    }
    if(s[2] != 'z'+1) {
        s[2]++;
    }
    else {
        s[2] = 'a';
        if(s[1] != 'z'+1) {
            s[1]++;
        }
        else {
            s[1] = 'a';
            if(s[0] != 'z'+1) {
                s[0]++;
            }
        }
    }
}
```

87/10/28
19:16:19

tritbl.c

1

```

/*
 * PROGRAM TO CALCULATE TRIGRAM FREQUENCIES AND MAKING
 * OUTPUT FIT FOR TROFF TBL OUTPUT
 * ONLY TABEL ENTRIES ARE GENERATED UNEQUAL 0
 *
 * J.C. Scholtes
 *
 * september 1987
 */

/*
 * The program first reads a word from a file, the trigram
 * values are determined by the formula: address = char 1 *
 * 27 **2 + char2 * 27 + char3 . The table has room for
 * 27 characters, including the space. The two spaces in front
 * and the two spaces at the end are also included.
 * After the table generation, the required output is generated
 * into the standard output channel.
 */

#include <stdio.h>

#define NR_CHAR 27
#define MAX_TABLE (NR_CHAR*NR_CHAR*NR_CHAR)
#define PROB_NOT_NIL 6000
#define SPACE 'z'+1

int table[MAX_TABLE];
int temp[PROB_NOT_NIL];

main()
{
    int i, p, j, regel, breedte;
    int index;
    char c, s[256], tempchar[PROB_NOT_NIL][3];

    for(i=0; i<MAX_TABLE; i++) table[i] = 0;
    for(i=0; i<PROB_NOT_NIL; i++) temp[i] = 0;

    while( (c = getchar()) != EOF){
        s[0] = SPACE;
        s[1] = SPACE;
        s[2] = c;

        i = 3;

        while( (c = getchar()) != '\012' && c != EOF){
            s[i] = c;
            i++;

            s[i] = SPACE;
            s[i+1] = SPACE;
            s[i+2] = '\0';
            i = 0;

            while(s[i+2] != '\0'){
                index = ((s[i]- 'a')*NR_CHAR*NR_CHAR) + ((s[i+1] - 'a')*NR_CHAR) + (s[i+2] - 'a');
                table[index]++;
            }
        }
    }

    p = 0;
    s[0] = s[1] = s[2] = 'a';
    for(i=0; i<MAX_TABLE; i++){
        if(table[i] != 0){
            tempchar[p][0] = s[0];
            tempchar[p][1] = s[1];
            tempchar[p][2] = s[2];
            temp[p] = table[i];
            p++;
        }
        if(s[2] != 'z'+1){
            s[2]++;
        }
        else{
            s[2] = 'a';
            if(s[1] != 'z'+1){
                s[1]++;
            }
            else{
                s[1] = 'a';
                if(s[0] != 'z'+1){
                    s[0]++;
                }
            }
        }
    }

    /*
     * generating the required output
     */

    i=0;
    while(i<(p-1)){
        printf(".TS\n");
        printf("tab(//)center box ;\n");
        printf("c s s s s s s s s s s\n");
        printf("c | c | c | c | c | c | c | c | c | c | c | c | c\n");
        printf("c | r | c | r | c | r | c | r | c | r | c | r | c | r.\n");
        printf("TRIGRAM FREQUENTION TABLES FOR UNIX-MANUAL\n");
        printf("\n");
        printf("tri/freq/tri/freq/tri/freq/tri/freq/tri/freq\n");
        printf("\n");
        for(regel=0; regel<50; regel++){
            for(breedte=0; breedte<6; breedte++){
                printf("%c%c%c / %d /", tempchar[i+(50*breedte)][0], tempchar[i+(50*breedte)][1], tempchar[i+(50*breedte)][2], temp[i]);
                printf("\n");
            }
            i++;
        }
        printf(".TE\n.bp\n");
        i=i+250;
    }
}

```

```

        (table[index])++;

        i++;
    }
}

p = 0;
s[0] = s[1] = s[2] = 'a';
for(i=0; i<MAX_TABLE; i++){
    if(table[i] != 0){
        tempchar[p][0] = s[0];
        tempchar[p][1] = s[1];
        tempchar[p][2] = s[2];
        temp[p] = table[i];
        p++;
    }
    if(s[2] != 'z'+1){
        s[2]++;
    }
    else{
        s[2] = 'a';
        if(s[1] != 'z'+1){
            s[1]++;
        }
        else{
            s[1] = 'a';
            if(s[0] != 'z'+1){
                s[0]++;
            }
        }
    }
}

/*
 * generating the required output
 */

i=0;
while(i<(p-1)){
    printf(".TS\n");
    printf("tab(//)center box ;\n");
    printf("c s s s s s s s s s s\n");
    printf("c | c | c | c | c | c | c | c | c | c | c | c | c\n");
    printf("c | r | c | r | c | r | c | r | c | r | c | r | c | r.\n");
    printf("TRIGRAM FREQUENTION TABLES FOR UNIX-MANUAL\n");
    printf("\n");
    printf("tri/freq/tri/freq/tri/freq/tri/freq/tri/freq\n");
    printf("\n");
    for(regel=0; regel<50; regel++){
        for(breedte=0; breedte<6; breedte++){
            printf("%c%c%c / %d /", tempchar[i+(50*breedte)][0], tempchar[i+(50*breedte)][1], tempchar[i+(50*breedte)][2], temp[i]);
            printf("\n");
        }
        i++;
    }
    printf(".TE\n.bp\n");
    i=i+250;
}
}

```

87/10/31
16:09:53

lispfun

1

```
;; Standaard en 'handige' lisp functies

(define (count lijst)
  (cond
    ((equal? lijst nil) 0)
    (t (+ 1 (count (cdr lijst)))))

; element gives the number-th element of lijst

(define (element number lijst)
  (cond ((< number 1) nil))
  (do ((dum lijst dum) (aantal number aantal))
    ((or (equal? dum nil)
         (eqv? aantal 1))
     (cond ((equal? dum nil) nil)
           ((eqv? aantal 1) (car dum))))
    (set! aantal (- aantal 1))
    (set! dum (cdr dum))))

(define (show l)
  (cond
    ((eq? l nil) t)
    (t (print (car l))
        (show (cdr l)))))

; (define (append l1 l2)
;   (cond
;     ((eq? l1 ()) l2)
;     ((eq? l2 ()) l1)
;     (t (cons (car l1) (append (cdr l1) l2)))))
; system has its own primitive

(define (delete a l)
  (cond
    ((eqv? l nil) ())
    ((eqv? (car l) a) (delete a (cdr l)))
    (t (cons (car l) (delete a (cdr l)))))

; system has its own, somewhat different implementation
; (define (member el lijst)
;   (cond
;     ((eq? lijst ()) nil)
;     ((equal? (car lijst) el) t)
;     (t (member el (cdr lijst)))))

(define (or arg1 arg2)
  (cond
    (arg1 t)
    (arg2 t)
    (t nil)))

(define (and arg1 arg2)
  (cond
    (arg1 (cond
            (arg2 t)
            (t nil)))
    (t nil)))

(define (abs arg)
  (cond
    ((< arg 0) (-arg))
    (t arg)))

(define (not arg)
  (cond
```

```
((eq? arg t) nil)
(t t)))

(load)
```

87/11/02
17:11:47

function

1

```
;; ADJUSTED VERSION TO NEW HOUGH INFORMATION :  
;; NOW THE FIFTH ELEMENT OF HOUGH LIST EQUAKS THE  
;; LINE_LENGTH....., THE FUNCTIONS ARE UPDATED  
;;  
;; linelength can be found with (car (cddr (cddr (car list-lines))))  
;; or with (list-ref list-lines 4)  
;;  
;; In this file you find all the functions wich calculate  
;; the values for the constraints of a specific set of lines  
;; produced by the hough function.  
;;  
;; the two following procedures detect a hor or ver line  
;; Note that a line of length 0 a hor and a ver line is:
```

```
(define (ver-line? line)  
  (eqv? (car line) (list-ref line 2)))
```

```
(define (hor-line? line)  
  (eqv? (list-ref line 1) (list-ref line 3)))
```

```
(define (pos-line? line)  
  (cond ((< (car line) (list-ref line 2))  
        (> (list-ref line 1) (list-ref line 3)))  
        ((> (car line) (list-ref line 2))  
        (< (list-ref line 1) (list-ref line 3)))  
        (t nil)))
```

```
(define (neg-line? line)  
  (cond ((< (car line) (list-ref line 2))  
        (< (list-ref line 1) (list-ref line 3)))  
        ((> (car line) (list-ref line 2))  
        (> (list-ref line 1) (list-ref line 3)))  
        (t nil)))
```

```
; the two following functions give the number of hor resp ver  
; lines in a list of lines. One should add in the call the  
; minimum length of a line to let it be counted  
;
```

```
(define (number-hor-line list-lines lte)  
  (cond  
    ((equal? list-lines nil) 0)  
    (t (cond  
        ((and (hor-line? (car list-lines))  
              (> (list-ref (car list-lines) 4) lte))  
          (+ 1 (number-hor-line (cdr list-lines) lte)))  
        (t (number-hor-line (cdr list-lines) lte))))))
```

```
(define (number-ver-line list-lines lengte)  
  (cond  
    ((equal? list-lines nil) 0)  
    (t (cond  
        ((and (ver-line? (car list-lines))  
              (> (list-ref (car list-lines) 4) lengte))  
          (+ 1 (number-ver-line (cdr list-lines) lengte)))  
        (t (number-ver-line (cdr list-lines) lengte))))))
```

```
(define (number-pos-line list-lines lengte)  
  (cond  
    ((equal? list-lines nil) 0)  
    (t (cond
```

```
((and (pos-line? (car list-lines))  
      (> (list-ref (car list-lines) 4) lengte))  
      (+ 1 (number-pos-line (cdr list-lines) lengte)))  
    (t (number-pos-line (cdr list-lines) lengte))))))
```

```
(define (number-neg-line list-lines lengte)  
  (cond  
    ((equal? list-lines nil) 0)  
    (t (cond  
        ((and (neg-line? (car list-lines))  
              (> (list-ref (car list-lines) 4) lengte))  
          (+ 1 (number-neg-line (cdr list-lines) lengte)))  
        (t (number-neg-line (cdr list-lines) lengte))))))
```

```
;  
; this function calculates the average line-length in a character  
;
```

```
(define (mean-length list-lines lengte)  
  (cond  
    ((equal? list-lines nil) 0)  
    (t (cond  
        ((> (list-ref (car list-lines) 4) lengte)  
          (+  
            (list-ref (car list-lines) 4)  
            (mean-length (cdr list-lines) lengte)))  
        (t (mean-length (cdr list-lines) lengte))))))
```

```
;  
; this function calculates the maximum linelength in a character  
;
```

```
(define (max list-lines)  
  (cond  
    ((equal? nil (cdr list-lines))  
     (list-ref (car list-lines) 4))  
    (t (let  
          ((m (max (cdr list-lines)))  
           (n (list-ref (car list-lines) 4)))  
          (cond  
            ((> n m)  
             n)  
            (t m))))))
```

```
;  
; this function calculates the minimum line length in a character  
;
```

```
(define (min list-lines)  
  (cond  
    ((equal? nil (cdr list-lines))  
     (list-ref (car list-lines) 4))  
    (t (let  
          ((m (min (cdr list-lines)))  
           (n (list-ref (car list-lines) 4)))  
          (cond  
            ((< n m)  
             n)
```

87/11/02
17:11:47

function

2

```
(t m)))))
```

```
; following function takes all the characters and counts the  
; number of hor and ver lines with a minimum length of 2
```

```
(define (generate)  
  (let ((areas (split '(0 0 343 241))))  
    (print (car areas))  
    (perform areas)))
```

```
(define (perform areas)  
  (let ((uit nil))  
    (do ((i 0 1))  
      ((> i 209) t)  
      (set! uit (hough (list-ref areas i) 2 1))  
      (do ((j 3 j))  
        ((> j 10) t)  
        (cond  
          ((to-many-lines? uit 0)  
           (set! uit (hough (list-ref areas i) j 1))  
           (set! j (+ 1 j)))  
          (t  
           (set! j 11)  
           (print  
            (list  
              i  
              (number-of-lines uit 2)  
              (number-hor-line uit 2)  
              (number-ver-line uit 2)  
              (number-pos-line uit 2)  
              (number-neg-line uit 2)  
              (/ (mean-length uit 2)  
                (number-of-lines uit 2))  
              (min uit)  
              (max uit))))))  
          (set! i (+ 1 i))))))
```

```
;;  
;;  
;; following function determines if there are too many lines > 50  
;;  
;;
```

```
(define (to-many-lines? list-lines number)  
  (cond  
    ((equal? list-lines nil) nil)  
    (t  
     (cond  
       ((> number 50) t)  
       (t  
        (to-many-lines? (cdr list-lines) (+ 1 number))))))
```

```
;; following function gives the number of lines longer  
;; than the minimum length
```

```
(define (number-of-lines list-lines lengte)  
  (cond
```

```
((equal? list-lines nil) 0)  
  (t (cond  
      ((> (list-ref (car list-lines) 4) lengte)  
       (+ 1 (number-of-lines (cdr list-lines) lengte)))  
      (t (number-of-lines (cdr list-lines) lengte))))))
```

```
; function klas-num-lines returns all the characters which have  
; 'aantal' or more lines
```

```
(define (klas-num-lines lijst aantal)  
  (cond  
    ((not (equal? lijst nil))  
     (cond  
       ((equal? (cadr (car lijst)) aantal)  
        (print (car (car lijst)))  
        (klas-num-lines (cdr lijst) aantal))  
       (t (klas-num-lines (cdr lijst) aantal))))  
    (t nil)))
```

```
(load)
```

87/10/31
16:09:11

framefun

1

```
(define (fget
  frame slot facet)
  (cdr (assoc facet (cdr (assoc slot (cdr (assoc frame frames)))))))

(define (fextend
  key a-list)
  (cond ((assoc key (cdr a-list)))
        (t (set-cdr! a-list (append (cdr a-list)
                                      (list (list key))))
          (assoc key (cdr a-list)))))

(define (ffollow-path
  path a-list)
  (cond ((null? path) a-list)
        (t (ffollow-path (cdr path) (fextend (car path) a-list)))))

(define (fget-frame
  frame)
  (cond ((assoc frame frames)
        (t (set-cdr! frames (append (cdr frames)
                                      (list (list frame))))
          (assoc frame frames)))))

(define (fput
  frame slot facet value)
  (let ((value-list (ffollow-path (list slot facet)
                                   (fget-frame frame))))
    (cond ((member value value-list) nil)
          (t (set-cdr! value-list (append (cdr value-list)
                                           (list value)))
            (value))))))

(define (fremove frame slot facet value)
  (let ((value-list (assoc facet (cdr (assoc slot (cdr
                                                    (assoc frame frames)))))))
    (cond ((member value value-list)
          (set-cdr! value-list (cdr (delete value value-list)))
          t)
          (t nil))))

(define (fclear frame slot facet)
  (let ((value-list (assoc facet (cdr (assoc slot (cdr
                                                    (assoc frame frames)))))))
    (set-cdr! value-list ())))

;; From here are the framestructure query functions defined

(define (query frame eisen)
  (cond ( (match frame eisen)

          (do
            ( (result nil result)
              (lijst (fget frame 'child 'value) lijst))
```

```
          ((or (not (eq? result nil))
                (eq? lijst nil))
           (cond
            ((not (eq? result nil)) result)
            (t frame)))

          (set! result (query (car lijst) eisen))
          (set! lijst (cdr lijst))))

  (t nil)))

;; note that by defining match as a separate procedure one can
;; easily modify and enhance the matching process

(define (match frame eisen)
  (do
    ((constraints (cddr (cdr (fget-frame frame))) constraints)
     (failure nil failure))

    ((or (eq? failure t)
          (eq? constraints nil))
     (cond
      ((eq? failure t) nil)
      (t t)))
    (set! failure (not (print (take-con (print (car constraints)) eisen))))
    (set! constraints (cdr constraints))))

;; Remind eisen = ( (slot value) ...) and
;; beperking = (slot (value) (constraint waarde waarde ..))

(define (take-con beperking eisen)
  (let ( (eis (assoc (car beperking) eisen)) )
    (cond (eis (set! eis (cadr eis)))
          (t nil))
    (cond
     ((member eis
              (cdr (assoc 'constraint beperking))) t)
     (t nil))))

;; function trace returns the list of constraints wich are in the
;; tree above the given frame until the level top-frame is reached

(define (trace frame)
  (cond
   ((equal? (fget frame 'parent 'value) '(frames))
    (cdr (cddr (fget-frame frame))))
   (t
    (cond
     ((cdr (cddr (fget-frame frame)))
      (append
       (cdr (cddr (fget-frame frame)))
       (trace (car (fget frame 'parent 'value))))))
     (t (trace (car (fget frame 'parent 'value))))))
```

(load)

FLISP IMPLEMENTATION DOCUMENTATION

APPENDIX F

1. Introduction

This document describes how Flisp is implemented. It is assumed that the reader has some knowledge about the implementation of a Lisp interpreter.

Flisp is a lisp dialect. The language which can be evaluated is a subset of Scheme. The language has a block structure like Algol a.s.o.

Roughly all functions of the Scheme definition [1] are implemented. Only the functions concerning other data types as symbol and integer are not implemented.

No macro definitions are possible, and file I/O is poorly and not according to the definition. No compiler is implemented.

2. Data Representation

2.1. The Cell

A cell is basically a structure with a flag byte and two pointers to another cell. A cell can contain other data like numbers or characters. See next paragraphs.

```
typedef struct cell {
    struct cell *car;
    struct cell *cdr;
    flags      flag;
} CELL;

typedef unsigned char flags;
```

MBOTTOM and MTOP stand for the lowest and the highest LISP memory address respectively and determine the LISP memory size (in cells).

The following flags are defined for a memory cell:

M = garbage-tor Mark-bit
R = Reserved for future use
L = List
A = Atom , also used by Print() for marking (see "The Printer")
N = Number
S = String
V = garbage-collector reVersed tag
B = Reserved for future use

A unsigned char contains all per cell flags. The flag is used for all kinds of marking.



A byte with eight flags

For further extensions it is useful to know that currently the flags B and R are not used.

If the time comes even that is not sufficient, the flags L, A, N and S are exclusive. A cell can only be of one type at the time. So in these four bits sixteen types can be defined.

2.2. The different types

This section covers the different types implemented in Flisp.

2.2.1. Numbers

In Flisp only one numeric type is available, integer number. A number is stored in a cell. The car part contains the integer, the cdr part is set to NIL, and the flag is N.

A compiler dependency is that $(\text{sizeof int}) \leq (\text{sizeof CELL} *)$. For most C-compilers this shouldn't be any problem.

Flags	Car	Cdr
S	123	nil

Flisp representation of the integer 123

There are only two procedure's which access the raw number implementation described above :

1. PutNumber()
2. GetNumber()

These procedure are defined in the file "gc.c", where all storage functions are defined.

2.2.2. Strings and Numbers

Strings are stored in cells. A string has a maximum length of STRLEN defined in file data.h. The car part of a string contains the first number of characters, the cdr part contains a pointer to the rest of the string, and the flag is S. The number of characters stored in the car part equals $(\text{sizeof CELL} *)$, the size of a cell pointer.

When no more character are to be stored, the remaining positions in the car part are filled with ' ', and the pointer in cdr is set to NIL.

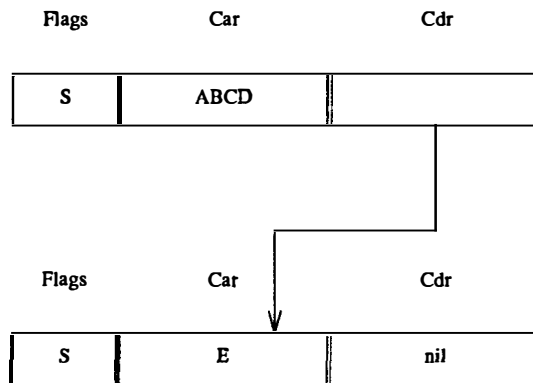
There are only two procedure's which access the raw string implementation described above :

1. PutString()
2. GetString()

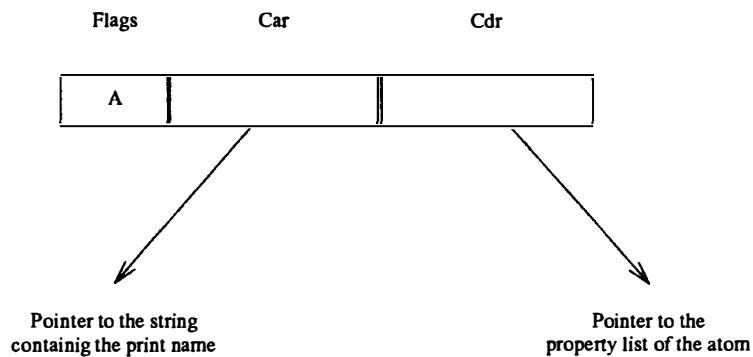
These procedure are defined in the file "gc.c", where all storage functions are defined.

2.2.3. Atoms

An atom is a basic Flisp data type. An atom has an print name, a property list and a value. An atom consists of 1 cell with a pointer to a string, the car, and a pointer to the property list, the cdr. The current value of an atom is stored in the current environment. (see section about evaluation or section 3 in the user manual.)



Flisp representation of the string ABCDE



Flisp representation of an atom

Predefined atoms are placed in the LISP memory just as any other atom. The only difference is that they are located in the lowest part of memory and they never change their positions (even if compacting garbage collection algorithm should be used).

Atoms are assigned constant values in the program. This is done by the procedure `GetInitEnv()` in the file `env.c`. The constant value assigned to a predefined atom corresponds to the LISP cell space address of the atom. Built-in functions take the locations between the addresses `MBOTTOM` and `PRIMITIVE_PROCEDURE`.

2.3. Storage

Sos uses two different storage spaces. A stack space for intermediate results, and a heap. It is possible to implement the stack within the heap. This approach is less efficient, because it uses more CPU-time and more memory space. The only disadvantage is that there are now two different space's which can overflow.

2.3.1. The Stack

The stack is implemented as an array of pointers to CELL of size STACKSIZE, defined in data.h. The operations on the stack are :

```
StackInit()  
Push(item)  
Item = Pop()  
ClearStack()
```

All these stack functions are defined in gc.c.

2.3.2. The Heap

The LISP heap space is implemented as an array of CELL's. The array is divided in two parts :

1. The free memory list
2. What's left

Every time NewCell() is called a cell is removed from the free list and is passed to some function which needs storage facility.

There only 4 functions which call NewCell() for allocating new data :

1. Cons()
2. PutNumber()
3. PutString()
4. CreateAtom()

No function is available to free allocated cells. After some time the free list will be emptied and no free cells are available any more. That's when the garbage collector is activated.

2.3.3. Garbage Collection

Every time a Lisp cell is needed, the function NewCell() is called. NewCell() first checks if a free cell is available. In the present implementation, free cells are placed in a linear list pointed to by the variable free list. There are no free cells in case free list points to NIL.

When no free memory cells are available, the garbage collection is started (function GarbageCollector()). GarbageCollector() consists of two phases:

- Mark phase
- Sweep phase

A skeleton of the GarbageCollector() is shown in Figure 5.1.

```
procedure GarbageCollector;  
  procedure Mark(list: pointer);  
  ...  
  procedure Sweep;  
  ...  
begin  
  Mark(oblist);  
  Mark(alist);  
  ...  
  Sweep;  
  if freelist = NIL then FatalError;  
end;
```

Figure 5.1 Skeleton of the GarbageCollector

The point is that when you want to reclaim cells, you only want to reclaim cell you are sure of they never will be used again. These cells are called "Garbage".

So now we have three kinds of cells :

1. Free cells
2. Garbage cells
3. Cells possibly to be used sometime

When we start garbage collection we know there are no free cells, so all cell are in [2] or [3]. If no cell or temporary storage space, intermediate variables within the evaluator, contains the address of cell X, cell X can never be used again. If this condition is satisfied cell X is garbage.

This is why all cells which can be reached from within the system are marked. All other cells are garbage, and can be collected.

The implementation of the garbage collector is copied from an earlier implementation of Shallow Binding Lisp written by Gerard van Bochoven, Marek Druzdzel and Paul van Niekerk.

They implemented a non-recursive mark algorithm (see [Knuth]). The reason for this that a recursive version does not use constant data space, with the danger of running out stack space if the level of recursion gets too deep.

Sweep() is the collector, and is implemented by means of a simple non compacting algorithm. All cells, which have not been marked, are added to the list of free cells.

3. The Input Parser

Scanning is done by lex, parsing is done by yacc. It is not difficult to code by hand an input scanner and an input parser for Scheme. I have chosen for lex and yacc because in this way it would be easier to make additions to the input parser. A second advantage is that yacc generates a non-recursive parser, although it uses a stack.

In the yacc-code intermediate results are pushed and popped on the stack for intermediate results. This is inefficient. Yacc keeps track of these result also. The choice to implement it this way is for portability, time, and maintainability.

All identifiers are stored in alphabetical order in the OBLIST. When an input token is recognised as an identifier, the function PutOblist() is called to search in the OBLIST for that identifier. If it isn't already in the OBLIST, the identifier is added. The function PutOblist returns the address where the name of the identifier is stored.

Because this address is unique, it is possible to compute within the Scheme system with this address instead of the name of the identifier. The name is only used when printing the identifier.

The result of the function Read() is a parsed expression. This expression is stored on the heap, and is passed to the evaluator.

No macro definitions are possible yet.

4. Evaluation

After parsing the input the expression can be evaluated. The function Eval() evaluates an expression. According to the type of the expression the work is delegated to other functions. All evaluation is handled by this file except for the primitive user functions, User....(), these are in the file "user.c".

All expressions are evaluated within an environment. Functions handling environments are in the file env.c.

An environment is a list of frames. Each frame contains a list of dotted pairs. The Car of such a pair is the variable which is bound to the Cdr of that same pair.

```
((x.1) (y.2)) (<frame_n-1>) .... (frame_1))
```

When a procedure is defined, the procedure stores the current environment in its data structure. When a procedure needs to be evaluated, the arguments will be evaluated and bound to the parameters of the procedure first. This binding implies that a new frame is made out of the value's of the arguments and the parameters. This frame is put on top of the frames of the environment which was stored at the time of the definition of the procedure.

```
((arg_n.val_n) ... (arg_1.val_1)) ((x.1) (y.2)) (<frame_n-1>) .... (frame_1))
```

This is how the lexical binding is done. The problem for the interpreter is that every time when a variable has to be evaluated, the environment has to be scanned for the first binding-instance of the variable. This is called the stack-scan. This takes relatively a lot of time, for every variable value has to be resolved this way.

Compilers on the other hand can use lexical addressing to find the binding of a variable. At compile time the compiler calculates in which frame the first binding is, and the number of the binding. So when the value of an variable has to be changed or read, the runtime system knows almost exactly where to look.

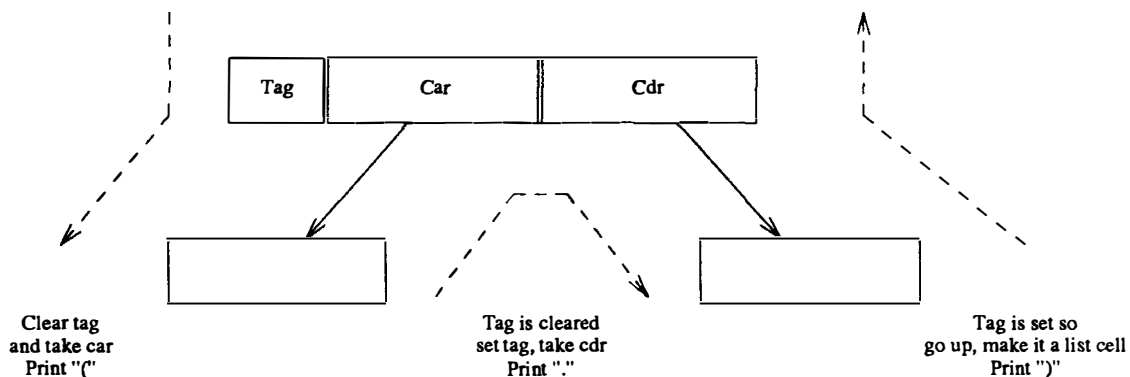
No compiler is implemented yet.

5. The Printer

The implementation of the printer is copied from an earlier implementation of Shallow Binding Lisp written by Gerard van Bochoven, Marek Druzdzal and Paul van Niekerk.

The printer is non-recursive. Print uses the method of link reversal, and therefore it does not need additional storage.

It does an attempt to Print an expression in list notation, returning to dot notation when necessary. The algorithm needs a Tag bit, but due to the nature of the algorithm, the Atom bit can be used for this purpose. The need is in "up" mode, to determine whether you came from the car or from the cdr direction. The following figure shows the order of printing a (sub)tree using the Tag.



Conversion to list notation is done as follows:

1. The sequence "(\$" is never printed. A " " is printed instead.
2. The sequence ".NIL" is left out.
3. A ")" is not printed when the previous cell has the Tag set.

6. What where to find

This section describes what the contents is of the different files.

data.h	This files contains all the global variable declarations and all the #define's specific for the Flisp implementation.
frwr.h	This files contains all the exported functions, the functions which are to be used in another file as where they are defined.
limits.h	This files contains machine limits. Not all limits are used.
local.h	This files contains some #define's and typedef's which are commonly used.
macrodefs.h	This file contains the macro definitions which replace functions when everything is compiled with FAST defined.
continuation.c	This is a machine dependent file. Here the continuations fetched by setjmp() are stored in the heap. Because every machine can have another size of the jmp_buff, this file has to be adapted when porting to a new machine.
env.c	This file contains all functions concerning environments
error.c	This file manages error handling.
evap.c	In here are the different evaluation functions.
gc.c	This files contains the memory management functions.
init.c	This file contains the initialisation routines.
io.c	All output is redirected through routines in this file. This is was necessary for the port to the N-Cube.
lisp.c	The main routine is in here.

print.c	Contains the printer
shell.c	This file is a shell around the cell structure. All access to a cell structure, except access to the continuations, is done by functions in this file.
user.c	All primitive functions, functions defined when initialising, are defined here.
util.c	This file contains some basic utilities.
lex.l	Contains the input scanner.
yacc.y	This file contains the input parser, and the the functions Read() and PutOblist().

