Parallelizing the Linkage Tree Genetic Algorithm and Searching for the Optimal Replacement for the Linkage Tree

Master's Thesis

Roy de Bokx

Parallelizing the Linkage Tree Genetic Algorithm and Searching for the Optimal Replacement for the Linkage Tree

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Roy de Bokx
born in Vlissingen, the Netherlands

Algorithmics Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Centrum Wiskunde & Informatica
Science Park 123
Amsterdam, the Netherlands
www.cwi.nl

Parallelizing the Linkage Tree Genetic Algorithm and Searching for the Optimal Replacement for the Linkage Tree

Author: Roy de Bokx
Student id: 1515624
Email: R.deBokx@student.tudelft.nl

Abstract

The recently introduced Linkage Tree Genetic Algorithm (LTGA) has shown to exhibit excellent scalability on a variety of optimization problems. LTGA employs Linkage Trees (LTs) to identify and exploit linkage information between problem variables. In this work we present two parallel implementations of LTGA that enable us to leverage the computational power of a multi-processor architecture.

These algorithm extensions for LTGA enable us to solve a problem that previously could not be solved, being the problem of finding high-quality predetermined linkage models that result in a better performance of LTGA for intricate problems by replacing the online-learned LTs. This is done by learning high-quality LTs offline by optimizing LTGAs performance as a function of static LTs. This results in a better performance of LTGA than with online-learned LTs as the problem complexity increases. A parameter-free implementation is used to search optimal subsets of linkage sets in the offline-learned LTs. This pruning of the LT results in a further performance improvement of the LTGA by, on average, removing about 50% of the linkage sets from the offline-learned LTs. This suggests that LTs contain redundancies that may possibly still be exploited to improve the performance of LTGA with online-learned LTs.

Thesis Committee:

Chair:	Prof. Dr. C. Witteveen, Faculty EEMCS, TU Delft
University supervisor:	Prof. Dr. C. Witteveen, Faculty EEMCS, TU Delft
Company supervisor:	Dr. P.A.N. Bosman, Life Sciences, Centrum Wiskunde & Informatica
Committee Member:	Prof. Dr. C. Vuik, Faculty EEMCS, TU Delft

Preface

This report contains the final results of my Master's thesis and is the conclusion of my Computer Science study at Delft University of Technology. Centrum Wiskunde & Informatica (CWI) gave me the opportunity to investigate and conduct research on a state of the art algorithm in an academical environment, under the supervision of Peter Bosman, senior researcher at the department of Life Sciences. Aimed at improving the performance of this algorithm in order to solve larger and more complex problems, this research provided us with a faster implementation of this algorithm which enabled us to investigate more fundamental improvements to the algorithm for which the results encountered will be presented in this report.

I would like to use this opportunity to thank the people that made this possible. Foremost I would like to thank Peter Bosman for giving me this great opportunity to graduate at an academic institute and visit one of the most renown conferences, *GECCO'15*, though most of all I would like to thank him for his guidance throughout this Master's thesis and for his relentless but always accurate and constructive feedback. Second, I would like to thank Cees Witteveen for providing feedback and for guiding me through the process of graduating at Delft University of Technology, as well as Dirk Thierens for providing critical feedback on the numerous ideas that have passed and for his collaboration in writing the paper submission for the *GECCO'15* conference. Last, I would like to thank my family and friends for their support, love, critical feedback and causing the necessary distraction in the past months.

Roy de Bokx
Delft, the Netherlands
July 18, 2015

Contents

List of Figures

List of Tables

Glossary

ℓ Number of parameters in a problem encoding.

$\boldsymbol{O}$ Offspring in a Genetic Algorithm. This is a collection of solutions.

$\boldsymbol{P}$ Population in a Genetic Algorithm. This is a collection of solutions.

$\boldsymbol{d}$ A donor solution from P used during Gene-pool Optimal Mixing (GOM).

$\boldsymbol{f}$ Fitness function evaluation time.

$\boldsymbol{g}$ Number of generations required by the Linkage Tree Genetic Algorithm (LTGA).

$\boldsymbol{n}$ Population size.

$\boldsymbol{p}$ Available number of processors.

$\boldsymbol{s}$ A solution in P.

Acronyms

EA Evolutionary Algorithm.

EDA Estimation of Distribution Algorithm.

EP Embarrassingly Parallel.

EP-LTGA Embarrassingly Parallel implementation of LTGA.

FI Forced Improvement.

FOS Family of Subsets.

GA Genetic Algorithm.

GOM Gene-pool Optimal Mixing.

GOMEA Gene-pool Optimal Mixing Evolutionary Algorithm.

LT Linkage Tree.

LT$_{off}$ offline-learned LT.

LT$_{on}$ online-learned LT.

LTGA Linkage Tree Genetic Algorithm.

MI Mutual Information.

MIMatrix Mutual Information Matrix.

MPM Marginal Product Model.

MRNE Minimally Required Number of Evalutations.

MRPS Minimally Required Population Size.

NN-Chain Nearest Neighbor Chain.

OM Optimal Mixing.

PP-LTGA Perfect Parallel implementation of LTGA.

Chapter 1

Introduction

Every day we all make hundreds if not thousands of decisions that together determine the course of our lives. Of most decisions we are not aware that we make them, however as the number of options increases, we tend to find it harder to make a decision. This can vary from choosing clothes to wear to determining what route to choose on your way home at the end of day. Behind most of these decisions, a problem can be found for which multiple aspects influence the consequences of this decision. One could for instance choose the shortest way home, though the fact that other people choose the same route, or that you first have to stop by a supermarket can influence whether or not this is a good idea. Some of those problems have so many aspects, variables and constraints that we are not able to evaluate what the best solution is, which is usually also due to the vast amount of options available. Ever since the first computer was invented, computers have been deployed to support people in solving problems they cannot solve themselves. This started off with solving relatively simple mathematical problems, though as the computational power of computers increased, more complex problems could be addressed, including complex combinatorial problems. Combinatorial problems are problems that yield multiple variables for which a solution consists of a specific assignment to those variables. These variables could for instance be indicators, e.g. for whether or not you're going to stop by the supermarket. A solution for this problem can be evaluated to rank it among other solutions. Some solution might be better than others, because for instance with this solution you do all the errands you need, while also spending little time in traffic. This way, everyday problems can be solved with the use of computers, though also more complex ones such as routing and planning problems can be mapped to this structure.

As the dimensionality, i.e. the number of variables, of combinatorial problems increases, they become harder to understand for humans, which is why computers are used to search for optimal solutions for these problems. Imagine an introduction camp for first year students of Computer Science and Mathematics at the TU Delft. One of the main purposes of such an introduction camp is to mingle and meet new fellow-students. Assume that for each student we know how well they know all other students, which is expressed by a value between 0 and 10, meaning that a connection exists between every two students with a *weight* between 0 and 10 expressing their mutual acquaintance. For the next activity of the camp we need two groups, so we want to split the group of first year students such that on

average students that already know each other are separated as much as possible. To reach this goal, it is important to put students that already have a strong connection in separate groups. More specifically, the sum of weights of the connections that are cut by splitting the group of students in two has to be as big as possible. However, as we have only two groups to assign students to, this can be far from trivial, even when, as in this example, the two groups do not have to be of equal size. As an example, separating two strongly connected students might prevent cutting other strong connections, for instance if these students both have a strong connection with a third student.

This problem is also known as the *MAXCUT* problem of which we will give a more technical description in Chapter 3. For a group of 10 people this problem can still be solved by hand, however not for an introduction camp for first year Computer Science and Mathematics students, which is attended by roughly 220 students, which means that roughly 48,180 connections have to be considered. This problem is one of many complex combinatorial optimization problems for which no computer programs are known yet that are able to solve them efficiently, which is usually due to the lacking ability to identify, efficiently represent or efficiently exploit the problem's structure due to the complexity of the problem. Finding a suitable representation of the most important parts of a problem's structure is often of great importance for efficiently exploiting the structural features of such a problem, for instance when it is decomposable, meaning that it can be decomposed into smaller sub-problems. In the example of our introduction camp this is for instance the case when our group of students consist of isolated groups, meaning that students of such an isolated group know each other, but don't know any other students. In this situation, our problem can be decomposed into a sub-problem for each isolated group, meaning that an optimal division can be constructed by finding an optimal division for each isolated group and combining these optimal sub-solutions. The combined complexity of these sub-problems for a problem like *MAXCUT* is far smaller than the complexity of the overall problem, which is why by identifying and exploiting this composition, an optimal solution can be found more efficiently. In reality, however, the decomposition of a problem is often complex and hard to identify, let alone represent and exploit efficiently, which is why for a large collection of combinatorial optimization problems no algorithms are known yet that can guarantee the finding of an optimal solution within polynomial time.

Many these problems are at the basis of every-day optimization problems in large organizations. This can vary from optimizing variables for the modeling of an airplane wing to optimizing routes of package delivery companies. Finding an optimal configuration is of great importance for these companies, as for some of these problems, small alterations to variable values can have great consequences for the outcome, for instance for the integrity of such an airplane wing, or the fuel costs for a package delivery company. Although for relatively small or simple problems often structural features can be identified by hand, large organizations often face problems so large or complex that automatic detection of these structural features is required.

In order to solve these optimization problems, many algorithms have been presented so far. One of these algorithms is the Linkage Tree Genetic Algorithm (LTGA) which is a state of the art algorithm that was first presented by Bosman and Thierens [6, 56]. This algorithm has shown to be able to efficiently find optimal solutions to complex problems

due to its unique approach by using novel techniques to identify the structure of a problem and efficiently exploit this when searching for the optimal solution, which enables LTGA to outperform its competitors on most problems it has been tested on. However, still many problems of great importance remain unsolved, which is why research in combinatorial problem solving is mainly aimed at expanding the collection of problems that can be solved within reasonable time.

This Master's thesis is aimed at expanding the problem solving capabilities of LTGA by investigating possible improvements, which is done according to the following research-questions:

1. What is currently preventing us from solving more complex problems with LTGA?
2. How can we increase the potential of LTGA to solve larger and more complex problems?

This Master's thesis describes the identified potential extensions for LTGA, how these were implemented and the results that have been achieved by leveraging these extensions, including insights gathered in the methodology used by LTGA to model the structure of a problem. This will be done by first discussing the background of this project, including the implementation of LTGA and use this to formulate more concrete research questions in Chapter 2. Next, we will validate and analyze the implementation that we developed of LTGA in Java 8 in Chapter 3 after which we will present the extensions that were implemented in Chapter 4. We will use these extensions to investigate the model used by LTGA to increase its potential, which will be discussed in Chapter 5. Finally, we will conclude with a discussion in Chapter 6 and present our conclusions and suggested future work in Chapter 7 where we will also revisit our research questions. The findings of this Master's thesis have been submitted to the *Genetic and Evolutionary Computation Conference (GECCO) 2015*, of which the full paper submission can be found in Appendix E. This submission has been accepted as a poster, of which the two-page abstract can be found in Appendix F and in the ACM Digital Library[1].

[1] `http://dl.acm.org/citation.cfm?id=2739482.2764679`

Chapter 2

Background and Problem Description

As stated in the previous chapter, we wish to expand the problem-solving capabilities of the Linkage Tree Genetic Algorithm (LTGA) such that larger and more complex problems can be solved efficiently by investigating and improving this algorithm that has already shown to have great potential. In this chapter, we will give a brief description of the background of optimization algorithms in general to give insight into the various possibilities for solving combinatorial optimization problems. Subsequently, we will zoom in on a selection of optimization algorithms, which will eventually bring us to a detailed description of the implementation of LTGA. This will support us in revisiting the research questions posed in the previous chapter to present a more concrete formulation, after which related work will be discussed to complete the context in which the research in this Master's thesis was performed.

2.1 Background

For many everyday problems, no computer programs are known yet that are able to solve them efficiently, which eventually brings us back to the P=NP millennium problem [14]. This problem questions whether it holds that if a problem is in NP it is also in P and visa versa. Problems in P are problems for which the solution can be *found* in polynomial time. A trivial example would be for instance finding the maximum value in a sequence of n integers, which can simply be done by traversing this sequence in $O(n)$ time. Problems in NP are problems of which the validity of a solution can be *verified* in polynomial time. Although solutions for any problem in P can be verified in polynomial time, many problems exist for which solutions can be verified in polynomial time while no algorithm is known yet to actually *find* a solution in polynomial time. An example of a so called *NP-complete* problem, which is a problem in NP that is at least as hard as any other problem in NP, is the problem presented in the previous chapter where a group of students has to be split in two groups, which is also known as the *MAXCUT* problem. For this problem we can verify in polynomial time whether a found cut is at least larger than any other known cut, however no

algorithm is known yet that can guarantee the finding of a maximum cut within polynomial time.

Although not proven, for now it is assumed that P≠NP, which means that it is not guaranteed that solutions to problems in NP can be found in polynomial time. The most important impediment on finding optimal solutions for NP-complete problems efficiently is usually finding the problem's structure and modeling it such that it is efficiently exploitable. Note that in this work we do not consider any problems outside NP, being problems for which the solution cannot be verified within polynomial time.

In the past, a scala of possible approaches for solving combinatorial problems has been presented, all attempting to solve problems from a different perspective, resulting in differences in outcomes and applicability. For instance there is the class of *guaranteed exact optimization algorithms* that guarantee to find an optimal solution. This could for instance be done by exploring the entire solution space or by using the problem structure in order to find the optimal solution more efficiently, if this is known and sufficiently understood. These algorithms can be applied to problems in P, however assuming that P≠NP, these algorithms cannot be applied to NP-complete problems of non-trivial sizes, as they do not scale efficiently with the size of an NP-complete problem. For such problems, *guaranteed approximation algorithms* and *non-guaranteed approximation algorithms* are used, which are both aimed at finding an answer within polynomial time that is within a certain distance from the optimal solution. They differ in being either able or unable to guarantee the finding of such a solution.

2.1.1 Evolutionary Algorithms

A class of algorithms that has shown to be good at solving complex problems that are not fully understood, is the class of Evolutionary Algorithms (EAs). EAs are algorithms that use the principle of evolution in order to find the optimal solution for a given problem [2, 33]. Instead of focusing on one single solution, these algorithms consider *population P*, which is a collection of n solutions. The main characteristics of EAs is that they use *variation* and *selection* to repeatedly create *offspring O* using the information contained by this population, where O is a collection of solutions that are aimed to be of higher quality than the ones contained by the population they originated from. In this situation, *variation* is defined as the act of creating new solutions based on solutions in the population, while *selection* again decreases the number of solutions to n by choosing well performing solutions over poorly performing solutions, based on their fitness. The fitness of a solution is the rating of such a solution for the problem at hand, expressing how well the solution performs. Unless indicated otherwise, scores are to be maximized by the algorithms presented in this work.

Often starting with an initial population that consists of randomly generated solutions, creating offspring based on a population is done repeatedly and each iteration is called a *generation*. In each generation, the offspring of the previous generation will be used as population to create improved solutions from. By doing so, EAs evolve a given population such that it will contain increasingly better solutions, which should eventually result in the finding of the optimal solution. This is illustrated by the following pseudo-code:

Algorithm 2.1: Evolutionary Algorithm Outline

```
1  P = generateRandomSolutions(n)
2  while(!stopConditionMet){
3    O = performVariation(P)
4    O = performSelection(P, O, n)
5    P = O
6  }
7  return P.getBestFound()
```

The specific combination of variation and selection enables an EA to traverse a much larger solution space compared to other algorithms. The only prerequisite for an EA is that the fitness of possible solutions can be evaluated, which makes them particularly useful for finding solutions to problems that are not fully understood or to problems that cannot be solved efficiently using guaranteed optimization algorithms. The finding of the optimal solution is often not guaranteed, which is why EAs generally belong to the class of non-guaranteed approximation algorithms.

2.1.2 Genetic Algorithms

There are different types of EAs. For traditional combinatorial optimization problems, solutions can often be binary encoded, which is why often Genetic Algorithms (GAs) are used to solve such problems. GAs are EAs of which the solution space is binary or cartesian encoded. Without loss of generality, we wille only consider problems with binary variables in this work, meaning that a possible solution consists of a sequence of ℓ binary values, also called a *bitstring*. The basic principles of adaptation were first presented by J.H. Holland back in 1975 [33], which later presented a paper that first introduced GAs [34]. Like EAs, GAs start with an initial population of solutions. Next, a GA performs variation and selection for every generation. A basic example of variation that is traditionally used by GAs is *uniform cross-over*, which creates two new solutions from two parent solutions from P by means of mixing. In pseudo-code:

Algorithm 2.2: `performVariation(P)` for Uniform Cross-over

```
1  O = {}
2  for(i in 0 to |P|){
3    parent1 = P.getRandom()
4    parent2 = P.getRandom()
5    child1 = []
6    child2 = []
7
8    for(k in 0 to ℓ){
9      if(getRandomBoolean()){
10       child1[k] = parent1[k]; child2[k] = parent2[k]
11     } else {
12       child1[k] = parent2[k]; child2[k] = parent1[k]
13     }
14   }
15   O ++ child1 ++ child2
16 }
17 return O
```

Consider the following example. Assume P contains two solutions 011010 and 110110 that are chosen for uniform cross-over. This could for instance mean that, using the pseudo-code, from the solutions **011**010 and 110**110**, the children **010010** and 111110 originate. Performing uniform cross-over is traditionally performed n times, where n is the size of the population and where for each uniform cross-over operation two solutions are randomly drawn from P to serve as parents. After variation is performed, selection is used to select n solutions from all $3n$ parents and children. This can be done by simply selecting the best n solutions, however also other heuristics exist such as *tournament selection* that causes high-quality solutions to be copied to the offspring multiple times, which will discard more low-performing solutions and increase the chance of a high-quality solution to be used when variation is performed during the next generation [57]. By combining important parts of solutions present in a population in this way, GAs are able to evolve a population of binary encoded solutions such that eventually the optimal value assignment for the problem's variables can be found.

2.1.3 Linkage Learning

As shown above, uniform cross-over can be used as variation technique in GAs. However, uniform cross-over only has a limited applicability, as it performs very strong variation that does not take into account that for some problems correlations or dependencies might exist between problem variables [58]. Consider for instance a problem with $\ell = 6$ variables, where the fitness function considers the first 3 variables as one block, while the other variables are independent of each other. Consider solutions 100110 and 111001, where the assignment of 100 to the first three variables constitutes a high fitness. In this situation, copying the first 3 variables from **100**110 to a child solution constitutes a faster convergence towards the optimal solution, while uniform cross-over would most likely disrupt this so called *building block*.

This shows that uniform cross-over can be efficiently applied to relatively simple problems, however, for more complex problems, uniform cross-over does not constitute an efficient search for the optimal solution. For long, it is known that this can be caused by the *linkage problem* [61], in which *linkage* is described as follows by Yu [66]:

> *If linkage exists between two genes, recombination might result in lowly fit offspring with high probability if those two genes are not transferred together from parents to offspring.*

Here, *genes* refer to the variables in the encoding of the problem. In in this work we will only consider problems that can be binary encoded, which means that every position in a bitstring corresponds to a gene. In our example of the introduction camp, this means that linkage exists between the genes corresponding to students between which a relatively high or relatively low connection exists, as not transferring these genes together from a solution that has an assignment for these genes that constitutes a high fitness, can be detrimental for the fitness of the offspring solution. If this happens, we say that an important building block has been disrupted.

In order to perform variation effectively, linkages between genes have to be identified on forehand. Identification of linkages is called *linkage learning*, which is a field in which extensive research has already been performed, as will be discussed in section 2.4.

2.2 The Linkage Tree Genetic Algorithm

The LTGA is a state of the art linkage-learning algorithm that was first presented by Thierens in 2010 [56]. This novel algorithm distinguished itself from other algorithms by using a tree-structured model that contains all identified variable linkages, which together represent the identified problem structure. This so called linkage model is called the Linkage Tree (LT) and is defined by Thierens as follows [56]:

Definition 1. *The Linkage Tree (LT) of a population of solutions is the hierarchical cluster tree of the problem variables using an agglomerative hierarchical clustering algorithm with a distance measure D. The distance measure $D(X_1, X_2)$ measures the degree of dependency between two sets of variables X_1 and X_2.*

D measures the degree of dependency between two sets of variables X_1 and X_2, meaning that a smaller distance corresponds to stronger linkage between X_1 and X_2. This is based on linkage learning of which the implementation in LTGA will be discussed in the next subsection.

In a sense, LTGA can be placed between conventional GAs and Estimation of Distribution Algorithms (EDAs), which are algorithms that attempt to find an optimal solution by estimating the probability distribution of solutions in the solution space. LTGA uses a more novel approach than conventional GAs for exchanging information with the use of linkage learning, aimed at identifying building blocks of the problem at hand. However, although it tries to identify the structure of a problem based on statistics of solutions in a population, it is not an EDA, as it does not attempt to approximate a probability distribution of such a population.

In this section, we will discuss the internal mechanics of LTGA, of which a general outline is presented in Figure 2.2. Just like traditional EAs, LTGA starts with generating a random population P, being a collection consisting of n solutions randomly drawn from a uniform distribution. Next, LTGA will generate offspring based on P by improving its solutions for each generation. After each generation, the best found solution so far will be saved, until one of the stop criteria is met. In LTGA, the following stop criteria are implemented:

1 The maximum number of fitness function evaluations has been reached.
2 A minimal fitness value has been reached by the best solution found so far.
3 The fitness variance of P has dropped below a certain threshold. This can be used to stop LTGA upon convergence, meaning that all solutions in P have the same fitness value and no further improvement is to be expected.
4 The *maximum no improvement stretch* has been reached, meaning that the number of generations not showing any improvement has exceeded a certain threshold.

Figure 2.1: LTGA General outline.

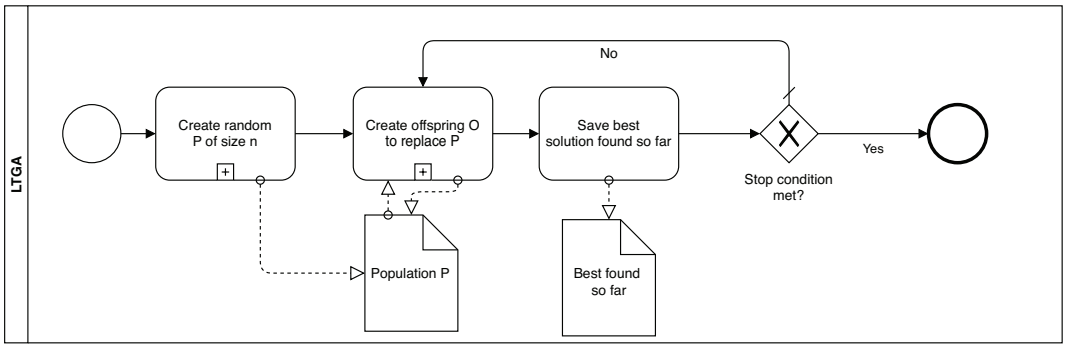
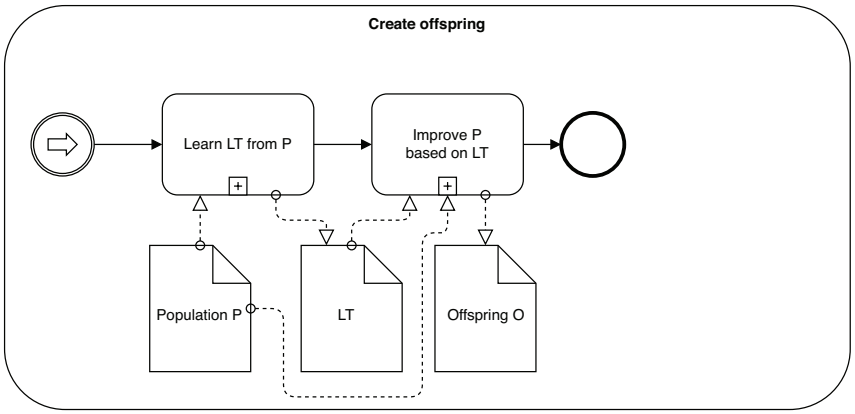


Figure 2.2: Creating Offspring.

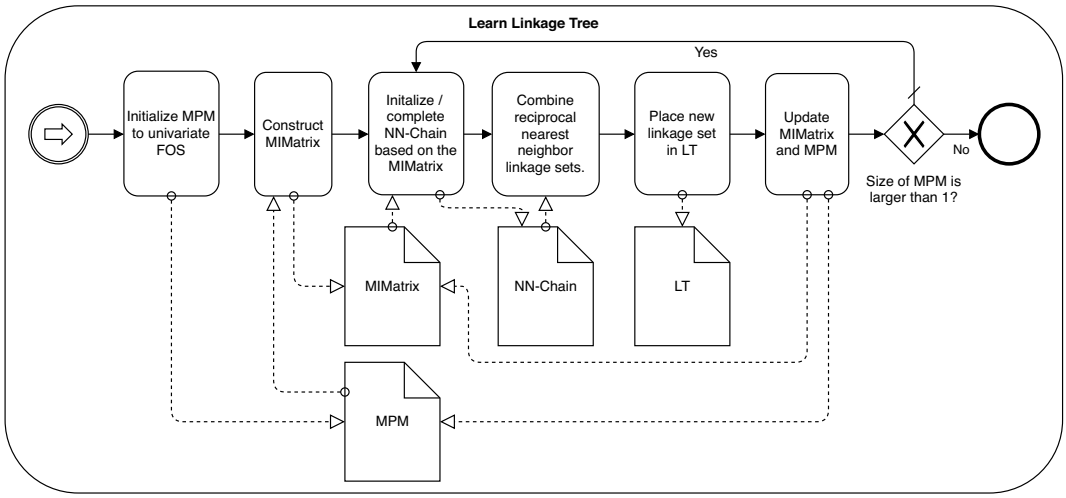


For each generation, LTGA first constructs an LT, which is discussed in the subsection below. Using this LT, Gene-pool Optimal Mixing (GOM) is performed in order to generate improved offspring based on P, which will be discussed consequently. This offspring O will replace P for the next generation and after each generation, LTGA checks for the stop criteria and will halt if any of these criteria are met, returning the best solution found so far. If none of the stop criteria are met, a subsequent generation will follow.

2.2.1 Building the Linkage Tree

Each generation starts by learning the Linkage Tree (LT), which is implemented as follows, as also illustrated in Figure 2.3. First, the Marginal Product Model (MPM) and LT are initialized to the *univariate FOS*. A Family of Subsets (FOS) is a subset of the power set of all problem encoding variables. An MPM is a FOS in which each problem variable is contained by exactly one set and the univariate FOS is a FOS that contains all variables in a singleton linkage set: $\big\{\{x_0\}, \{x_1\}, ..., \{x_{\ell-1}\}\big\}$ [57]. Based on this MPM, the *distance matrix D* is constructed, containing the distances between all clusters currently contained by in the MPM, which is defined as the difference between the joint entropy and the Mutual Information (MI) of the clusters, divided by the total information as presented by the en-

Figure 2.3: Learning the LT.



tropy, in order to normalize the metric. These metrics are defined by the definitions below, in which $p_i(X_k)$ is the chance that variables in X_k have a configuration i, which in the implementation used is approximated by the relative frequency of i for X_k in P. In other words, every cell in D quantifies the linkage between two sets of variables, based on the correlation between these variables that was found among the solutions in the current population.

Joint Entropy:

$$H(X_k) = -\sum_i p_i(X_k) \log p_i(X_k) \tag{2.1}$$

Mutual Information (MI):

$$I(X_1,\ldots,X_\ell) = \sum_{k=1}^{\ell} H(X_k) - H(X_1,\ldots,X_\ell) \tag{2.2}$$

Proximity metric:

$$D(X_1,X_2) = \frac{H(X_1,X_2) - I(X_1,X_2)}{H(X_1,X_2)} \tag{2.3}$$

Using a metric that is based on MI is beneficial to the scalability of the algorithm because of the *grouping property*, which states [56]:

Definition 2. *The mutual information between three clusters of random variables C_1, C_2 and C_3 is equal to the sum of the mutual information between two clusters C_1 and C_2, plus the mutual information between the union of the two clusters $C_1 \cup C_2$ and C_3: $I(C_1,C_2,C_3) = I(C_1,C_2) + I((C_1 \cup C_2),C_3)$*

This means that when clusters are grouped, only entries in D have to be changed that are related to these clusters. In LTGA, the negated MI is used in D, as a larger MI value indicates a stronger correlation between two clusters, indicating that the distance between

these clusters is smaller. Therefore, in LTGA, the distance matrix is called the Mutual Information Matrix (MIMatrix). According to the terminology used by Chen [12], this is the *centralized model* of the algorithm.

Next, a Nearest Neighbor Chain (NN-Chain) is initialized, which is a chain of linkage sets [26, 57]. In this chain, each element is the *nearest neighbor* of its predecessor, meaning that the linkage set X_{k+1} in the NN-Chain is the set that is closest to set X_k according to D. If indeed a strong correlation exists between the variables in X_k and X_{k+1}, as the algorithm progresses, solutions in P are likely to have a specific configuration in common for variables in X_k and X_{k+1} that constitutes a better fitness. This corresponds to a high MI which means these linkage sets will be considered as nearest neighbors. If we look back to our example of an introduction camp as presented in Chapter 1, then the students that correspond to the variables in X_k on average would have relatively strong or weak connections to students in X_{k+1}.

Initializing this NN-Chain is done by starting off with a random element from the MPM. Next, the nearest neighbor of the last element in the NN-Chain will be added to the chain repeatedly, until a loop was introduced. If such a loop is introduced, this will always mean that the last element is equal to the third-last element. This is called a *complete* NN-Chain. In pseudo-code:

Algorithm 2.3: `InitializeNNChain()`

```
1  NNChain = {}
2  NNChain.add(MPM.getRandom())
3  NNChain.makeComplete(MPM)
4  return NNChain
```

Algorithm 2.4: `NNChain.makeComplete(MPM)`

```
1  while(NNChain.size < 3)
2    NNChain.add(MPM.getNearestNeighbor(NNChain.last))
3
4  while(NNChain.last != NNChain.thirdLast)
5    NNChain.add(MPM.getNearestNeighbor(NNChain.last))
6
7  return NNChain
```

Consider for example the MIMatrix presented in Table 2.1 that contains a part of the MI values calculated from some population P with $\ell = 10$ where ℓ is the number of variables in the problem's binary encoding. To provide a clear overview of how an LT is constructed, we will only consider the variables $\{x_0,...,x_4\}$ in this example. As mentioned before, after constructing an MIMatrix, the NN-Chain is initialized with a random element, for instance $\{x_4\}$. As presented by Table 2.1, the nearest neighbor of $\{x_4\}$ is $\{x_2\}$, which will therefore be added to the chain. Adding the nearest neighbor of the last element in the chain is repeated until a loop is introduced which in this case results in the following complete NN-Chain:

$$NNChain_{complete} = \{x_4\} \rightarrow \{x_2\} \rightarrow \{x_0\} \rightarrow \{x_1\} \rightarrow \{x_0\}$$

After constructing a complete NN-Chain, the last three elements, being essentially two different sets from the MPM, are removed from the NN-Chain and MPM, merged and added

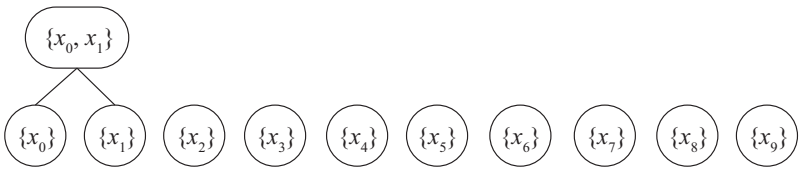
Table 2.1: Cropped example of an MIMatrix.

	$\{x_0\}$	$\{x_1\}$	$\{x_2\}$	$\{x_3\}$	$\{x_4\}$	...
$\{x_0\}$	-	0.1	0.2	0.8	0.8	...
$\{x_1\}$	0.1	-	0.4	0.4	0.9	...
$\{x_2\}$	0.2	0.4	-	0.9	0.3	...
$\{x_3\}$	0.8	0.4	0.9	-	0.4	...
$\{x_4\}$	0.8	0.9	0.3	0.4	-	...
...	...	...	...	...	...	...

Table 2.2: Cropped example of an MIMatrix after first merge.

	$\{x_0,x_1\}$	$\{x_2\}$	$\{x_3\}$	$\{x_4\}$	...
$\{x_0,x_1\}$	-	0.3	0.6	0.85	...
$\{x_2\}$	0.3	-	0.9	0.3	...
$\{x_3\}$	0.6	0.9	-	0.4	...
$\{x_4\}$	0.85	0.3	0.4	-	...
...	...	...	...	...	...

Figure 2.4: An incomplete LT after merging the mutually nearest neighbors $\{x_0\}$ and $\{x_1\}$.



to the LT and MPM again as a new linkage set. In in our example this results in the incomplete LT as shown in Figure 2.4, an MPM containing the sets $\{\{x_0,x_1\},\{x_2\},\{x_3\},\{x_4\},...\}$ and the following NN-Chain:

$$NNChain = \{x_4\} \rightarrow \{x_2\}$$

The MIMatrix is updated accordingly, meaning that the sets that were merged are removed and the MIMatrix entries for the new linkage set are generated with the use of the Unweighted Pair Group Method with Arithmetic-mean (UPGMA). This method defines distances between a cluster C_k and $(C_i \cup C_j)$ as the average distance between (C_k,C_i) and (C_k,C_j), as defined by the formula (2.4) [26]. In our example, this would result in the MIMatrix shown in Table 2.2.

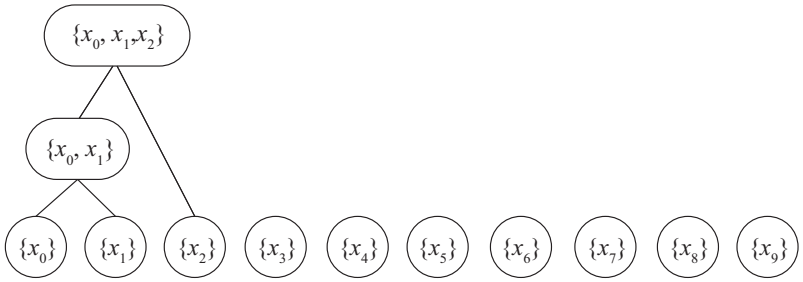
$$D(C_k,(C_i \cup C_j)) = \frac{|C_i|}{|C_i|+|C_j|}D(C_k,C_i) + \frac{|C_j|}{|C_i|+|C_j|}D(C_k,C_j) \tag{2.4}$$

Merging nearest neighbor linkage sets is done repeatedly which means that in our example in the next iteration, based on the MIMatrix shown in Table 2.2 and the incomplete NN-Chain above, the following complete NN-Chain is constructed:

$$NNChain_{complete} = \{x_4\} \rightarrow \{x_2\} \rightarrow \{x_0,x_1\} \rightarrow \{x_2\}$$

From this complete NN-Chain, the linkage sets $\{x_0,x_1\}$ and $\{x_2\}$ are removed, merged and added to the LT as shown in Figure 2.5 and the MPM and MIMatrix are updated accordingly. This merging of nearest neighbor linkage sets is repeated until only the union of all variables remains in the MPM. This linkage set will not be added, as using this linkage set during GOM would mean rather copying an entire solution, which is unwanted when performing variation as will be illustrated in the next subsection. This results in a bottom-up agglomerative hierarchical clustering mechanism that produces a tree-like linkage model

Figure 2.5: An incomplete LT after merging the mutually nearest neighbors $\{x_0, x_1\}$ and $\{x_2\}$.



which is called the Linkage Tree (LT). This LT essentially models the most important elements of the problem based on the correlations that could be identified in the current population. In our introduction camp example this could correspond to significantly strong or weak connections between students. As the solutions in P improve per generation, this LT will represent the true structure of the problem better and better, supporting the combination of important building blocks and with that an efficient convergence towards the optimal solution, as will be shown next. An example of such an LT is displayed in Figure 2.6.

Note that an advantage of using a distance metric that is based on MI, is that only one row or column of values has to be calculated, rather than recalculating the entire distance matrix, due to the grouping property [56]. Additionally, with the use of the UPGMA, only pairs of distances are considered which further increases the efficiency of updating MIMatrix as shown by Thierens, Bosman and Gronau [26, 57].

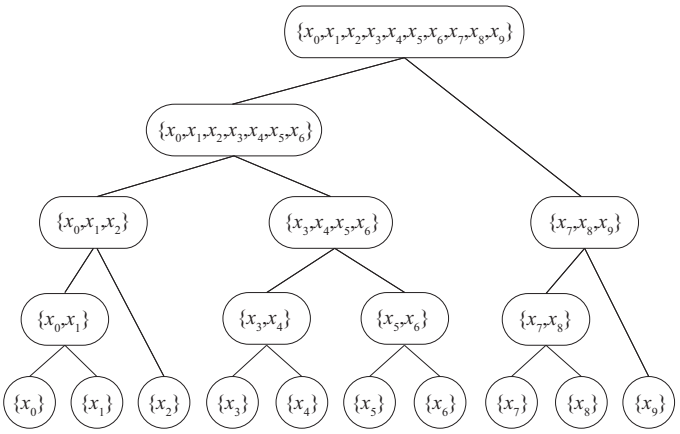
In this way, a linkage model is build by iteratively combining *locally closest pairs*, in contrast to the previous implementation of LTGA that merged *globally closest pairs*. The advantage of using locally closest pairs is that it enables us to reduce the time complexity for constructing a linkage neighbor model from $O(n * \ell^3)$ to $O(n * \ell^2)$ when using UPGMA, while the output is equivalent compared to the globally closest pair implementation as shown by Gronay and Moran [26]. This way of linkage learning is called *virtual linkage learning*, as the physical locations of the variables in the chromosome, i.e. the ordering of variables, do not affect their linkage [12].

Also note that, in contrast to the earlier version of LTGA of Bosman and Thierens [8], in this implementation the LT is not learned from a selection of P as it was found by Bosman and Thierens that this generates too much selection pressure, impeding the exploratory capabilities of LTGA.

2.2.2 Gene-pool Optimal Mixing

After the LT has been learned, the offspring collection O is to be created from the current population P. This is done by means of Gene-pool Optimal Mixing (GOM), which is aimed at improving all solutions contained by P with the use a FOS of the variables in a problem's encoding [6, 57]. In LTGA, this FOS is represented by the LT, enabling GOM to perform optimal mixing while respecting the linkages that were identified.

Figure 2.6: A complete Linkage Tree (LT).



As shown by Figure A.1, which can be found in Appendix A, the GOM phase attempts to improve a copy of each solution s in P. This is done by traversing the LT in a *random* order for each solution, as this gives on average a slightly better performance compared to a top-down traversal [8]. For each linkage set in the LT, a donor solution d is randomly picked from P and the linkage set is used as a cross-over mask to perform Optimal Mixing (OM). This means that by using the linkage set and a random donor d, s is to be improved by copying the values for the variables in the linkage set from d to s. In terms of our example of Chapter 1, this means that given some division of students, parts of some other possible division are adopted. If this does not impose a lower fitness value, the improved solution is used for further traversing the LT. Note that changes that impose no change in the fitness of the solution are also accepted. This can cause a severe performance improvement of LTGA for problems that contain plateaus in the fitness landscape, such as the *MAXSAT* problem, as this enables LTGA to move across these plateaus and explore symmetries in the solution space [8, 52]. If copying values from d to s imposes a lower fitness, the changes to s are reverted before further traversing the LT. This is done until improvement with all linkage sets in the LT is tried. Note that Figure A.1 is aimed at giving a general impression of how GOM works. Our actual implementation also uses Forced Improvement (FI), which means that GOM is again performed on s if after traversing the entire LT no changes have been made to s, however now the best solution found so far will be used as donor with each linkage set [6]. Note that FI will stop when improvement has been encountered, or when the LT is fully traversed. If this still does not not result in a better fitness, all values of the best solution found so far are copied. FI replaces the selection step traditionally done by EAs at the end of a generation as it was found that this selection step causes a selection pressure that impedes LTGA's exploration capabilities [6]. Bosman and Thierens state that FI does not continuously increase the selection pressure, but only puts a very specific convergence pressure on a solution that could otherwise not be improved, being less detrimental in terms of diversity loss. This enables efficient convergence in LTGA, even if there are multiple global optima. In this situation, with the use of FI, LTGA will choose one optimum, instead of alternating between them.

After improving *s*, it is stored in *O*. This implementation is illustrated by the following pseudo-code:

Algorithm 2.5: `generateNewSolution(Solution s)`

```
result = s.copy(), backup = s.copy()
int[] order = getRandomOrder(lt.size)
for i = 0 to lt.size -1
  donor = population.getRandom()
  result = copyForLinkageSet(donor, result, lt[order[i]])
  if(result != backup && fitness(result) >= fitness(backup))
    backup = result
  else
    result = backup

//Forced Improvement
if(result == s || noImprovementStretch > (1 + log10(n)))
  i = 0
  order = getRandomOrder(lt.size)
  while i < lt.size && result == s
    result = copyValues(bestSoFar, result, lt[order[i]])
    if(result != backup && fitness(result) > fitness(backup))
      backup = result
    else
      result = backup
    i++
if(result == s)
  result = bestSoFar

return result
```

Note that *s* is copied before any improvements are done, which means that constructing *O* does not affect *P* while GOM is performed. Also note that fitness function evaluations are only done when *s* has actually changed.

This way, LTGA is able to find the optimal solution to binary encoded combinatorial problems. Nevertheless, it is not proven that the returned solution is always the optimal solution, which makes it a non-guaranteed approximation algorithm, also called a *heuristic*. More specifically, as LTGA is not problem-specific, it is a meta-heuristic, which makes it particularly interesting to investigate due to its general applicability.

Although the performance and capabilities of this algorithm have been evaluated based on a set of problems of which we know the structure and optimal solution, note that LTGA is a black-box optimization heuristic, which means that it is aimed at operating in a context in which no problem-specific knowledge is available. Therefore, this algorithm is to be compared with other black-box optimization algorithms rather than algorithms that use problem-specific knowledge to solve problems more efficiently, though at the cost of being less generally applicable.

2.2.3 Execution Example

To provide insight into how the combination of linkage learning and GOM can support the finding of an optimal solution for a wide variety of problems, we will present an example

of a possible execution of LTGA for *Deceptive Trap* with $\ell = 10$ and $k = 5$. We will discuss this problem in detail in Chapter 3, however for now it is only important to know that in this problem linkage exists between variables $\{x_0, x_1, x_2, x_3, x_4\}$ and $\{x_5, x_6, x_7, x_8, x_9\}$ and the local optimum is 0000000000 while the global optimum is 1111111111. In our example, LTGA will terminate when the fitness variance in P has dropped below 0.0001, meaning that the population has fully converged. To maintain a clear overview of what happens during an execution of LTGA, a population size of $n = 10$ was chosen for this example, although a larger population size is needed in order to find the optimal solution to this problem with a high probability [6].

As presented in Figure 2.2, LTGA starts with initializing a population of randomly generated solutions, which can be found in Table 2.3 in the column for g_0, where every solution is accompanied by its fitness. As this population is randomly generated, no correlation to the linkage structure of the *Deceptive Trap* problem is to be expected yet.

Next, LTGA will traverse several generations until the population has fully converged, where during each generation offspring will be created by learning an LT and using this LT during GOM. For the first generation, an LT is learned from the population with randomly generated solutions, which in our case resulted in an LT that contained the linkage sets $\big\{\{x_0\}, \{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}, \{x_6\}, \{x_7\}, \{x_8\}, \{x_9\}, \{x_0, x_5\}, \{x_0, x_3, x_5\}, \{x_2, x_9\}, \{x_1, x_2, x_9\}, \{x_1, x_2, x_8, x_9\}, \{x_0, x_1, x_2, x_3, x_5, x_8, x_9\}, \{x_6, x_7\}, \{x_4, x_6, x_7\}\big\}$. Using this LT, GOM is performed in which for each solution this LT is traversed in a random order for which for each linkage set a random donor from the population is used.

We illustrate how GOM is able to construct improved solutions by investigating how the first solution of this population, being 0010101101, was transformed into 0000000001. The first linkage set used when randomly traversing the LT for this solution was $\{x_0, x_3, x_5\}$, for which the donor 0111000011 was randomly chosen. This would have copied the values 0, 1 and 0 for x_0, x_3 and x_5 respectively from the donor to our solution, however as this would cause a decrease in the fitness of this solution, this operation was reverted and no changes were made to our solution.

Next, the linkage set $\{x_6, x_7\}$ was used with donor 0011000111, which resulted in the improved solution 0010100101, entailing an increase in fitness from 0.6 to 0.8. Several linkage sets were tried after this that could not impose an improvement of the fitness, which

Table 2.3: Population contents per generation for an execution example for *Deceptive Trap*. Every solution is accompanied by its fitness.

g_0	g_1	g_2	g_3	g_4	g_5
0010101101 (0.6)	0000000001 (1.4)	0100000000 (1.4)	0000000000 (1.6)	0000011111 (1.8)	1111111111 (2.0)
0101000000 (1.2)	0000000000 (1.6)	0000000000 (1.6)	0000011111 (1.8)	1111111111 (2.0)	1111111111 (2.0)
0011000111 (0.6)	0000000000 (1.6)	0000000000 (1.6)	0000011111 (1.8)	1111111111 (2.0)	1111111111 (2.0)
1010011110 (0.4)	0010011111 (1.6)	0000011111 (1.8)	0000011111 (1.8)	1111111111 (2.0)	1111111111 (2.0)
1010010001 (0.8)	0100000000 (1.4)	0000000000 (1.6)	0000011111 (1.8)	1111111111 (2.0)	1111111111 (2.0)
0011100011 (0.6)	0000000000 (1.6)	0000000000 (1.6)	0000011111 (1.8)	1111111111 (2.0)	1111111111 (2.0)
0101111100 (0.4)	0100010000 (1.2)	0000000000 (1.6)	0000011111 (1.8)	1111111111 (2.0)	1111111111 (2.0)
0111000011 (0.6)	0100000000 (1.4)	0000000000 (1.6)	0000011111 (1.8)	1111111111 (2.0)	1111111111 (2.0)
1111101101 (1.2)	1111100001 (1.6)	1111100000 (1.8)	1111111111 (2.0)	1111111111 (2.0)	1111111111 (2.0)
0100000000 (1.4)	0000000000 (1.6)	1111100001 (1.6)	1111100000 (1.8)	1111111111 (2.0)	1111111111 (2.0)

could be due to the fact that copying variables from donors did not entail any change in the solution, or due to the fact that copying these variables imposed a worse fitness of the solution. With the use of linkage sets like $\{x_2\}$ and $\{x_4,x_6,x_7\}$ and donors 0100000000 and 0101000000 respectively, performing GOM on this solution resulted in the improved solution 0000000001 that has a fitness of 1.4, which was stored in the offspring.

Similar GOM operations are performed on all other solutions in the population, which resulted in offspring that was used as population in the next generation, presented in the column of g_1 in Table 2.3. This population already shows more resemblance with the structure of *Deceptive Trap* as for most solutions you can clearly see that the first building block of 5 variables or the second building block has converged to either the global or local optimum. This is reflected by the LT that is learned from this population, which contains the linkage sets $\big\{\{x_0\}, \{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}, \{x_6\}, \{x_7\}, \{x_8\}, \{x_9\}, \{x_2,x_9\}, \{x_3,x_4\}, \{x_0,x_3,x_4\}, \{x_0,x_2,x_3,x_4,x_9\}, \{x_0,x_1,x_2,x_3,x_4,x_9\}, \{x_6,x_7\}, \{x_6,x_7,x_8\}, \{x_5,x_6,x_7,x_8\}\big\}$. Here we see that this LT contains only few linkage sets that contain variables from both building blocks, which means that when using this LT, LTGA will attempt to find an optimal solution by copying important building blocks from high-quality solutions rather than disrupting them, in a sense solving the problems it can be decomposed to separately.

Considering the contents of this LT and the P of this generation, chances are generally small that a solution can be improved using randomly selected donors. Therefore, it is expected that often the FI mechanism is triggered, which will perform another GOM operation if a solution could not be improved, however now with the best solution of this population, where in this example 0000000000 was chosen as the best solution. Therefore, more building blocks of solutions converge towards the local optimum, however of some solutions the building blocks that were converged towards the global optimum were preserved.

This results in the population as presented in column g_2, where an even stronger pattern is shown that corresponds to the structure of the problem. With this, also the chance increases of learning an LT that contains linkage sets that essentially represent the problem structure, which for this problem would be the linkage sets $\{x_0,x_1,x_2,x_3,x_4\}$ and $\{x_5,x_6,x_7,x_8,x_9\}$. This is reflected by the LT learned for this generation, which contains the linkage sets $\big\{\{x_0\}, \{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}, \{x_6\}, \{x_7\}, \{x_8\}, \{x_9\}, \{x_0,x_4\}, \{x_2,x_3\}, \{x_0,x_2,x_3,x_4\}, \{x_0,x_1,x_2,x_3,x_4\}, \{x_5,x_7\}, \{x_6,x_8\}, \{x_5,x_6,x_7,x_8\}, \{x_5,x_6,x_7,x_8,x_9\}\big\}$. With this LT again GOM is performed. Here again it is expected that for most linkage sets in the LT no improvement can be obtained, however note that by using the linkage sets $\{x_0,x_1,x_2,x_3,x_4\}$ and $\{x_5,x_6,x_7,x_8,x_9\}$, high-quality solutions are built quite efficiently by replacing non-optimal building blocks by building blocks that have already converged to the global optimum for most solutions. This can be done by using random donors, however note that if this is not possible, FI will use the best solution of this population to copy this optimal building block to solutions that have not been improved yet.

As shown in column g_3, this enables LTGA to already find the globally optimal solution once, whereas for all other solutions a clear correlation to the problem's structure is shown. Therefore, roughly the same operations are performed in this generation compared to the previous generation, however because now the optimal solution is also the best solution of this population, all solutions that did not receive an optimal building block yet from a random donor, will receive one from this optimal solution during FI. Therefore, generation

g_3 and g_4 are merely needed to converge the population to the optimal solution by copying optimal building blocks from high-quality solutions, which eventually results in the converged population as presented in column g_5 where all solutions are identical and LTGA is terminated after returning this optimal solution.

2.3 Problem Description

We have shown how LTGA is able to find optimal solutions by means of linkage learning and Gene-pool Optimal Mixing (GOM) and why it is particularly suitable for solving complex problems for which no algorithms are known yet that can guarantee a solution within polynomial time. It is unique in its approach as it constructs a hierarchical linkage model of the dependencies that exist between the problem's variables, called a Linkage Tree (LT). It uses this model to enhance the generation of offspring, which enables it to generate offspring with solutions of higher quality. This in turn enables LTGA to converge efficiently towards the optimal solution and outperform its competitors for most problems that were tested, being other black-box optimization meta-heuristics that do not require any a-priori knowledge.

Even though LTGA shows excellent performance, there are still possibilities to improve the algorithm's applicability. First of all, the current implementation is only single-threaded, meaning that it is not able to use all computational power available when run on a multi-core architecture. At this moment, this is the biggest impediment on deploying LTGA on real-world problems as the computational demands for such problems are increasingly high. Also because nowadays computational power becomes more easily available, even for small businesses and private users, this seems to be the most promising possible extension to the algorithm for which straight forward implementations are available. Secondly, the current implementation still requires one parameter. Coming from an algorithm with at least 6 and up to 12 parameters, Bosman and Thierens spent extensive effort on simplifying the use of LTGA, ultimately aimed at developing a parameterless black-box algorithm that needs no a priori knowledge about the problem at hand. Although they constructed well-founded guidelines for most parameters, still one parameter remains, being the population size. In Chapter 3 we will present what the influence is of the population size on the performance of LTGA, however for now it is important to note that the required population size for solving a problem with a certain probability is highly dependent on the problem and problem size and no heuristics for this could be constructed so far. This has a severe impact on the usability of the algorithm and means that the algorithm can neither easily nor efficiently be used by other than experts.

This Master's thesis is focused on this state of the art algorithm due to it's potential and robustness. Our goal is to push the boundaries of optimization problem solving by expanding the potential of LTGA, though if we look at the research questions presented in Chapter 1 and take into account the details discussed in this chapter, it is clear that more detailed research questions are needed to define a more concrete approach and give our search for a more powerful optimization algorithm direction. We therefore present the following sub-goals that we aim to achieve:

1. Develop an implementation of LTGA that is able to harness the computational power of multi-processor architectures.
2. Develop an implementation of LTGA that can be executed on combinatorial problems without requiring problem-specific parameters.
3. Provide insight into LTGA's true potential.
    a) Investigate the model used by LTGA using the implementations mentioned above.
    b) Investigate alternative models such that a better performance of LTGA can be achieved.

We pursue to achieve these goals by answering the following research questions:

1. What part of LTGA has the most significant impact on the performance of the algorithm?
2. What are different ways LTGA can be redesigned in order to use the computing power available in multi-processor architectures efficiently?
    a) What are their advantages?
    b) What are their disadvantages?
    c) How does the performance of the improved implementation relate to the performance of the original sequential implementation?
3. How can we determine the population size used by LTGA such that LTGA no longer requires any parameters?
4. Can a fixed structure be found that supports a better performance of LTGA by replacing the LT?
    a) In what way does this fixed structure differ from the LT used by LTGA?

If we are able to answer the questions above, more insight can be obtained into LTGA's performance and with that, into possible improvements to the algorithm. This is an important goal as this can increase the performance of LTGA and with that it's applicability. This should ultimately expand the collection of problems solvable by LTGA, including new problems of great importance that could not be solved before.

2.4 Related Work

In this section, we will briefly discuss related publications that were studied in order to put the research conducted on LTGA into context.

2.4.1 Linkage Learning

LTGA uses *linkage learning* in order to construct an LT, which is a field in which already extensive research has been performed [11, 12, 15, 27, 28, 29, 31, 61, 67]. Starting from work presented by Thierens back in 1993 [61], linkage learning has a long history, however, in this section we will primarily focus on more recent developments in the field of linkage learning. For example, Yu et al. investigate how linkage learning can benefit situations in which a problem has to be solved with overlapping *building blocks*, which are defined as

variable sets into which a problem can be decomposed [67]. If a problem can be decomposed into building blocks, it means that the problem can be split up into sub-problems, each sub-problem solely considering the variables of a particular building block. These sub-problems can be solved independently, after which the solution to the original problem can be found by aggregating over the solutions found for the sub-problems. Yu et al. show that when trying to increase the efficiency of the cross-over operation, there is a trade-off between the amount of disruption of building blocks and the amount of mixing that occurs. They propose an algorithm that identifies the overlapping building block topology of the problem, identifies the relations between these building blocks and then partitions the graph of building blocks such that the number of disrupted building blocks is minimized. The pitfall of this algorithm is that it requires calculating an optimal partition of a graph, which itself is an NP-hard problem, indicating that this algorithm is quite costly. Using this algorithm, however, they were able to show that *detection failure* and *false linkage* both have an impact on the algorithm convergence time, which is the time that the algorithm needs before the entire population has converged towards one solution. This can be a local optimum, meaning that there might be a better solution, however ideally this should be the global optimum, meaning that no better solution exists. Yu et al. define *detection failure* as the inability of the algorithm to identify variable sets with high linkage, while *false linkage* is defined as the act of linking variables that do not have high linkage in reality. They show that for up to 60 building blocks, the negative impact of detection failure is severely larger than the impact of false linkage, while for problems with more building blocks, false linkage has a slightly bigger negative impact on the convergence time.

Linkage learning has also been studied in the context of EDAs [5, 11, 15, 22]. EDAs are able to find global optima in complex problems, by estimating the probability distribution of solutions in the solution space and using this to perform variation by drawing samples from this distribution. This is done with the use of higher-order statistics that introduce high computational complexity. Lower-order statistics require less execution time, however when using lower-order statistics, EDAs are usually not able to find global optima in complex problems, being problems in which interaction between variables exists. Emmendorfer and Pozo discuss the use of lower-order statistics combined with a clustering technique based on linkage learning in order to efficiently find global optima, even for complex problems [22]. This strategy is embodied by the φ-PBIL algorithm. The authors show that this outperforms Bayesian networks, which are considered to be very efficient in solving complex problems [32]. According to the authors this might be due to the fact that the algorithm uses an approximation of the structure of the problem, instead of calculating the exact structure.

However, also outside the field of EDAs linkage learning has been used for efficient problem solving. An example is the Linkage Learning Genetic Algorithm (LLGA) that was first presented by Harik and later investigated by Chen and Goldberg[13, 27]. Chen had already done extensive research on the scalability of the LLGA [12], where we define the scalibility of an algorithm as the ability to terminate within reasonable time as the dimensionality of a problem increases. Together with Goldberg, he presented three linkage learning techniques for GAs: *perturbation-based methods, linkage adaptation techniques* and *probabilistic model builders*. With these techniques, the LLGA is capable of solving

problems for which the complexity scales exponentially compared to the number of building blocks in linear time, while it needs an exponentially growing population size compared to the number of building blocks when the problem is uniformly scaled. This due to the fact that although it is long believed that a GA optimizes multiple building blocks simultaneously, Chen and Goldberg show that GAs first work on the most important building block, then the second most important building block, and so on. For the exponentially scaled problem this means that the most important building block is tightened first with very high probability due to the selection advantage. In contrast, building blocks in the uniformly scaled problem have the same probability of being tightened first. The authors state that in order to improve the performance of GAs, a mechanism has to be used that reduces the number of building blocks that are processed simultaneously.

Later, Duque and Goldberg also discuss the principle of linkage learning and in fact, their approach has been the basis for the linkage learning mechanism used by LTGA [21]. Duque and Goldberg present *ClusterMI*, which is a new method for linkage detection in the Extended Compact Genetic Algorithm (ECGA). This ClusterMI procedure hierarchically clusters variables by repeatedly clustering *linkage sets*, that have highest linkage, based on the MI, where *linkage sets* are sets of variables between which linkage has been identified [40]. Experiments show that an ECGA with the ClusterMI procedure needs a larger population than conventional ECGA implementations and therefore also more function evaluations. However, the model building step in the ClusterMI implementation is more efficient by one order of magnitude, therefore the ClusterMI implementation is both faster and better scalable than conventional ECGA implementations.

2.4.2 Pairwise and problem-specific metrics

Pelikan et al. [47] discuss the effects of pairwise and problem-specific metrics on the first implementation of LTGA. The authors show that a pairwise metric reduces not only the computational complexity of LTGA, but also the Minimally Required Population Size (MRPS), which causes a significant reduction in the required execution time of the algorithm. Another remarkable result that was presented by Pelikan et al., was that using problem-specific metrics does not necessarily mean that a better LT is created. In some cases, the use of problem-specific metrics causes the algorithm to identify linkage in too much detail, which means that the linkages that were defined might not be as important as other linkages or might not even be relevant for the current population. Experiments show that with the use of problem-specific distance metrics, LTGA does not always perform better.

2.4.3 Gene-pool Optimal Mixing and Forced Improvement

LTGA is a member of the Optimal Mixing Evolutionary Algorithm (OMEA) family as first presented by Thierens and Bosman [57]. In this paper, the authors investigated the efficiency of mixing in GAs and EDAs and present two implementations of Optimal Mixing (OM): the Gene-pool Optimal Mixing Evolutionary Algorithm (GOMEA) and Recombinative Optimal Mixing Algorithm (ROMEA). While then LTGA was still a ROMEA, Thierens

and Bosman show that the use of GOM often results in better performance, which is why GOM is used by LTGA from that point on.

Later, the mixing phase of LTGA was improved by combining GOM with FI [6]. The authors describe how in LTGA selection was done at two points in the algorithm by performing *tournament selection*. Tournament selection is a selection mechanism that selects n solutions from P by holding n tournaments of size t. This means that for each tournament, t solutions will be compared of which the best solutions will be placed in selection S. In LTGA, S was used to learn linkages from, which is known to be beneficial in certain EDAs [55]. Note that S was only used for *learning* the LT; the GOM phase is executed on the population that S originated from. Secondly, at the end of every generation again tournament selection was performed to converge by logistic growth of the optimal solutions over multiple generations. Bosman and Thierens find, that with the additional selection step at the end of every generation, the diversity is reduced faster than needed. They show that when removing this additional selection step, the MRPS, as well as the required number of fitness function evaluations to solve typical benchmark problems decreases. Therefore, they replace this step by FI that is applied during the GOM phase of the algorithm as described before.

2.4.4 Measuring LTGA

In 2012, Bosman and Thierens published two other papers related to LTGA. The first paper discusses the effects of different measures used for detecting linkage, as to study what it is that is needed from a measure in order for LTGA to converge to the optimal solution efficiently [7]. In this paper, they review five measures:

1 **H**: Entropy: $H(X_k) = -\sum_i p_i(X_k) \log p_i(X_k)$
2 **MI**: Mutual Information: $MI(X_1,\ldots,X_\ell) = \sum_{k=1}^{\ell} H(X_k) - H(X_1,\ldots,X_\ell)$
3 **MNI**: Mutual Normalized Information: $MNI(X,Y) = MI(X,Y)/H(X \cup Y)$
4 **VI**: Variation Information: $VI(X,Y) = H(X \cup Y) - MI(X,Y)$
5 **NVI**: Normalized Variation Information: $NVI(X,Y) = VI(X,Y)/H(X \cup Y)$

Of these metrics, using the Entropy causes a bad performance for LTGA. The other metrics show better results, although they differ only marginally, not providing enough support to prefer one of these measures over the other. The second paper they presented focused on measuring to what extend linkage learning by means of the LT contributes to converging towards the optimal solution in LTGA [58]. This was analyzed according to four measurements:

1 **Hamming Distance**: the minimum and median hamming distance to the global optimum from the solutions in the current population.
2 **Evolvability**: the evolvability is defined as the probability of success, being the probability that a new solution will have a better fitness than its parents [1].
3 **Linkage Model Evolvability**: the Linkage Model Evolvability is the evolvability as a function of the size of the masks of successive LTs.
4 **Evolvability-Based Fitness Distance Correlation (EFDC)**: This measure is considered to give the best insight in the contribution of an LT towards the

optimal solution, as it measures the correlation between the fitness value and the hamming distance towards the global optimal solution.

The authors show that of these metrics, the Hamming Distance does not give enough insight in whether LTGA is making structural progress or not, while other metrics are useful tools for obtaining insight in the characteristics of linkage model building in GAs. With these tools it is shown that LTGA is able to efficiently construct a linkage model that benefits the algorithm's performance.

2.4.5 Improving LTGA

As mentioned before, we aim to investigate the model used by LTGA in order to obtain insight into possible improvements to this model. Some research in this was already done by Thierens and Bosman.

Overlapping Linkages and Linkage Filtering

In 2013, again Bosman and Thierens published two papers that were related to LTGA. One of the papers discusses the ability of LTGA to solve problems with a hierarchical structure such as shuffled HIFF problems [63] and shuffled HTRAP [46] problems [60]. They show that for these problems, LTGA is efficient, reliable and scalable. The other paper presents results on tests that were performed with the Linkage Tree and Neighbors Genetic Algorithm (LTNGA) [8]. This algorithm combines the linkage learning Mechanisms of LTGA and the Multi-scale Linkage Neighbors Genetic Algorithm (MLNGA) [6] by using the intersect of their linkage models. This results in an algorithm that is able to combine the efficient model building of LTGA with the ability of MLNGA to represent overlapping building blocks, though still LTGA was preferred due to its robustness.

Furthermore, Bosman and Thierens attempted to improve LTGA by implementing *linkage hierarchy filtering*, which is aimed at filtering out potentially inefficient linkage sets from the linkage model. They base this on the *Linkage Strength (LS)*, which is defined as the average MI between all pairs of variables in a set F. If $LS(F^j) > LS(F^i)$, where $F^j \in F^i$, the linkages in F^j are stronger on average and combining this cluster with other variables, resulting in F^i, would result in rather disrupting larger building blocks than improving them. Therefore, after constructing the linkage model, all linkage sets for which $LS(F^i) \leq 0.99LS(F^j)$ are removed from the linkage model. This resulted in only a slight performance improvement of LTGA.

Predetermined versus Learned Linkage Models

Thierens and Bosman also investigated whether LTGA would perform better if the LT were to be replaced by a fixed structure that resembles the problem formulation structure perfectly [59]. This is done by inserting a fixed structure for the *Onemax*, *Deceptive Trap*, *NK-Landscapes* and *MAXCUT* problems. Results show that for *Onemax* and *Deceptive Trap*, LTGA indeed performs better in terms of required number of fitness function evaluations if the LT is replaced by a fixed structure that resembles the problem's formulation

structure. This is expected, as for these relatively simple problems, the formulation exactly identifies the most important building blocks of the problem. Surprisingly, however, for *NK-Landscapes* and *MAXCUT*, which are considered to be more realistic problems, LTGA using the online-learned LT not only generally performs better, but also scales significantly better compared to when the LT were to be replaced with a fixed structure. This shows that the actual problem structure of these problems is all but trivial and cannot be derived from the problem's formulation in a straight-forward way.

This raises the question whether fixed structures exist that can support a better performance for LTGA also for realistic problems. An answer to this question could indicate what the main reasons are for LTGA's excellent performance and with that enable us to leverage the true potential of LTGA, expanding the domain of problems to which LTGA can be applied.

Chapter 3

Implementation Validation and Analysis

As stated in the previous chapter, we aim to expand the potential of the Linkage Tree Genetic Algorithm (LTGA), however in order to do that, it is first important that our current implementation of LTGA is validated and analyzed as we translated the existing implementation of LTGA from the C programming language to Java 8. Often, C is preferred over Java for its unbeaten performance, however although C is indeed faster than Java, Java is generally only slower by a factor 1.5 to 2, which is not bad compared to other programming languages [19]. Additionally, for real-world problems, LTGA often has to be integrated with some other program to evaluate the fitness of possible solutions, for instance when the fitness is based on some simulation. Evaluating possible solutions for real-world problems is often costly, meaning that it requires significant CPU time. Therefore, choosing Java 8 over C will only have a limited impact on the execution time for such problems. Finally, Java 8 had shown to contain new features that would provide better support for developing a parallel implementation, such as lambda functions [44], which, together with the ExecutorService [43] would support an easy, clean and efficient implementation of a parallelized LTGA. Together with our more extensive experience with Java and the limited time available, Java 8 was therefore preferred over C.

The core of LTGA was implemented using Object Oriented Programming (OOP), which enabled us to design and organize the codebase according to best practice coding principles. Additionally, the JUnit framework was used to implement unit tests in order to verify internal functionalities [38]. The complete pseudo-code of the implementation can be found in Appendix B, together with the complexity analysis, which will also be discussed in this chapter.

To verify the soundness of our implementation a validation of our implementation of LTGA will be presented in this chapter. This will be followed by a complexity and cost analysis that will enable us to eliminate any impediments on possible algorithm improvements and determine what components of LTGA should be focused on when designing algorithm improvements or extensions.

For this project, hardware was made available by CWI for testing how the performance of LTGA could be improved. CWI recently invested in two identical 64-core servers, of

Table 3.1: 64-Core server specifications.

CPU(s)	4 x 16-core AMD Opteron(tm) Processor 6386 SE
CPU Cores	64
CPU min clockspeed	1.4 GHz
CPU max clockspeed	2.8 GHz
CPU cache	2048 KB
Memory	252 GiB

which the specifications can be found in Table 3.1. All results presented are based on this architecture, as we could use one of these servers to run our experiments.

3.1 Implementation Validation

In this section, we will first present the benchmarking problems which are not only used to validate this implementation of LTGA, but also to evaluate the performance of the improved or extended versions of LTGA presented in this work. Next, we will briefly discuss the setup and results of the experiments performed to verify that the Java implementation is sound, which are the same tests as presented by Bosman and Thierens [6, 8]. This experimental setup will also be used to verify or investigate other implementations that will be presented later on.

3.1.1 The Evaluation Problems

LTGA is considered to be a black-box meta-heuristic for combinatorial optimization problems that is able to optimize combinatorial problems without requiring any a priori knowledge about the problem structure. In previous publications about LTGA, a collection of combinatorial benchmarking problems were used that vary in difficulty and linkage structure. As these problems are considered suitable for investigating the performance of LTGA on combinatorial optimization problems, the base implementation, as well as any future improved versions of LTGA, will be verified performing experiments on these problems with varying dimensionalities. In this section, these problems will be illustrated where, without loss of generality, we will only consider binary variables. Formulas for these problems are given below and express how the fitness of a solution x will be evaluated. For all problems, an optimal solution has to be found, being a solution that corresponds to the maximum fitness value. In other words, the outcome of formulas presented below has to be maximized. Note that for testing purposes, for all benchmark problem instances used in our experiments the optimal solutions are known.

Onemax

The first and most trivial problem is the well-known *Onemax* problem, in which no significant linkage between variables exists. For this problem, the sum over all variables must

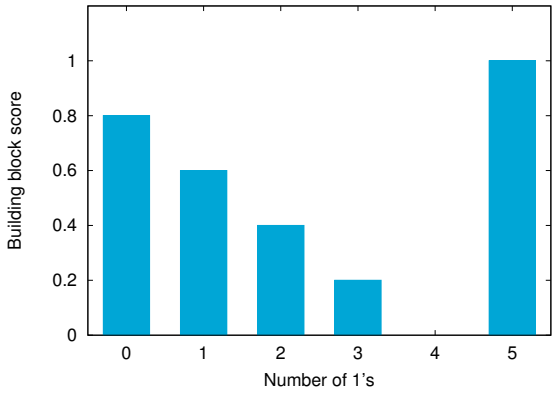
be maximized, as expressed by formula (3.1). Obviously, this means that in the optimal solution all binary variables are assigned a value of 1, resulting in the optimal score of ℓ.

$$f_{Onemax}(x) = \sum_{i=0}^{\ell-1} x_i \tag{3.1}$$

Deceptive Trap

The second problem on which the performance of LTGA is evaluated is the additively decomposable composition of k-order *Deceptive Trap* functions, which is aimed at drawing a deceivable optimization algorithm away from the global optimum, into a local optimum [17, 18, 21]. In our experiments, we used a tight encoding, meaning that the problem can be decomposed into ℓ/k non-overlapping sub-problems of size k, where each block i encapsulates the variables $i*k$ to $(i+1)k-1$. For each sub-problem, the corresponding sub-function contains a deceptive trap, meaning that this sub-function is structured in such a way that a simple algorithm would naturally be drawn towards the local optimum, instead of the global optimum.

Figure 3.1: The $f^{sub}_{Trap\text{-}k}$ function of Deceptive Trap, $k = 5$. Building block score defined by the number of 1s contained.



To illustrate, all experiments presented are performed for $k = 5$, which means that each consecutive group of 5 variables corresponds to a sub-problem. If a building block contains the bits 00000, it yields a score of 0.8 and this score decreases as the number of 1's in the block increases. If the block contains 11111, however, the solution will yields a score of 1 as illustrated in Figure 3.1. For example, the solution 01101000001101111111 is evaluated by summing the score of the 4 blocks it consists of (01101, 00000, 11011 and 11111), resulting in a score of $0.2+0.8+0+1=2$. Evidently, for the optimal solution, all consecutive blocks must consist of 11111 for $k = 5$, however, for each block, straight-forward algorithms are drawn towards the local optimum as soon as a block contains less than k 1s. This means that also for the solution as a whole, straight-forward algorithms will be drawn towards the local optimum, being a solution containing k 0s for every building block, yielding a total

score of $\frac{(k-1)\ell}{k^2}$ instead of $\frac{\ell}{k}$. The formula for this fitness evaluation function is defined as follows:

$$f_{Trap}(x) = \sum_{i=0}^{(\ell/k)-1} f^{sub}_{Trap\text{-}k}\left(\sum_{j=ki}^{ki+k-1} x_j\right) \tag{3.2}$$

where

$$f^{sub}_{Trap\text{-}k}(u) = \begin{cases} 1 & \text{if } u = k \\ \frac{k-1-u}{k} & otherwise \end{cases}$$

This function is useful for evaluating the performance of an algorithm as it contains so called deceptive traps and linkage between variables in each of the $\ell/5$ consecutive non-overlapping groups of variables. These deceptive traps are also often contained by real-world problems, which is why it is important to evaluate whether an algorithm is able to overcome these local optima and converge towards a global optimum by means of linkage learning.

NK-Landscapes

The third problem that is considered is the nearest-neighbor maximum overlapping additively decomposable composition of predetermined random sub-functions of length k, also known as the nearest-neighbors *NK-Landscapes* problem [48]. This problem is designed to be decomposible into sub-problems of length k that are maximum overlapping. Each *NK-Landscapes* problem instance therefore consists of $\ell - k + 1$ sub-problems in which the first $k-1$ variables of each sub-problem overlap with the last $k-1$ variables of the preceding sub-problem, as is visualized in Table 3.2.

For each problem instance, these f^{sub}_{NK} sub-functions are defined by predetermined randomly generated lookup tables, being tables that provide a mapping from each possible composition of variables $x_{i,i+1,\ldots,i+k}$ to a real value. An example of a problem instance for $\ell = 5, k = 3$ is presented in Table 3.3. The fitness function of a solution is then defined by the sum over the outcomes of all these sub-functions, as defined by the following formula:

$$f_{NK\text{-}S1}(x) = \sum_{i=0}^{\ell-k} f^{sub}_{NK}(x_{i,i+1,\ldots,i+k-1}) \tag{3.3}$$

Table 3.2: Visualization of variable overlap of sub-functions in *NK-Landscapes* for $k = 5$.

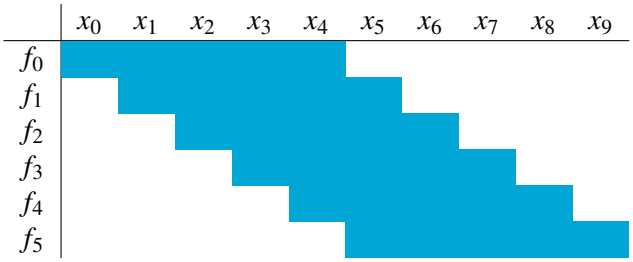


Table 3.3: An *NK-Landscapes* problem instance example. f_{NK}^{sub} functions for $\ell = 5$, $k = 3$.

$x_0x_1x_2$	score
000	0.3
001	0.3
010	0.1
011	0.9
100	0.8
101	0.0
110	0.2
111	0.4

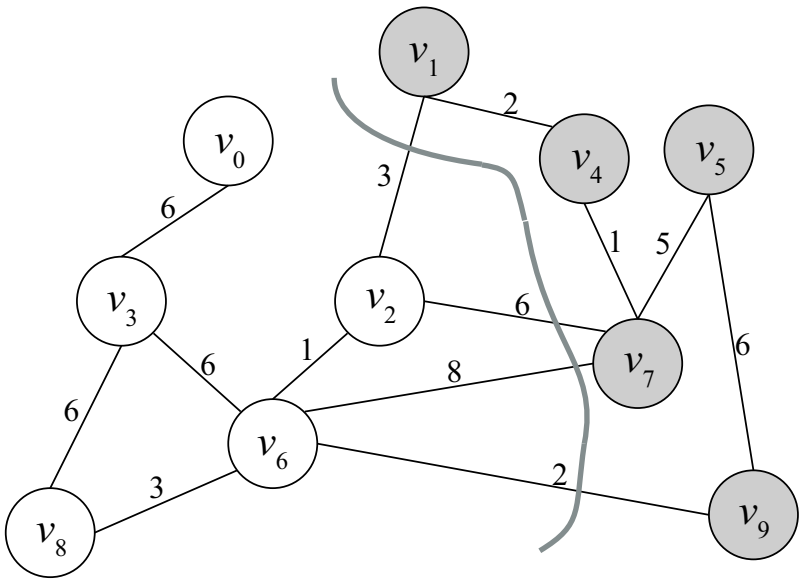
$x_1x_2x_3$	score
000	0.1
001	0.4
010	0.1
011	0.5
100	0.9
101	0.2
110	0.3
111	0.2

$x_2x_3x_4$	score
000	0.8
001	0.6
010	0.5
011	0.8
100	0.0
101	0.0
110	0.4
111	0.2

As an example, this means that for the problem instance presented in Table 3.3, the solution 11001 would be evaluated by summing the outcomes of $f_0(\mathbf{110}01)$, $f_1(1\mathbf{100}1)$ and $f_2(11\mathbf{001})$, resulting in a score of $0.2 + 0.9 + 0.6 = 1.7$. For the experiments presented in this work, the same randomly generated problem instances were used as the ones used by Bosman and Thierens [6, 7, 8, 56, 57, 58, 59, 60].

The purpose of this problem is to study the behavior of LTGA on problems that contain an intricate overlapping linkage structure. Clearly, the f_{NK}^{sub} subfunctions indirectly define the linkage between variables, however as these sub-functions are overlapping and do not contain a specific structure, the exact degree of linkage between variables, which is determined by the total composition of all f_{NK}^{sub}, is unknown.

Figure 3.2: An example of a *MAXCUT* problem instance.



Weighted MAXCUT

The last, but certainly not trivial benchmark problem that was used is the well-known NP-Complete weighted *MAXCUT* problem. This problem is defined given a weighted undirected graph with a set of ℓ vertices $V = \{v_0, v_1, ..., v_{\ell-1}\}$, a set of edges E and a weight w_{ij} for each edge $(v_i, v_j) \in E$. In this problem the sum of the weights of edges that were cut when splitting the graph in two parts is to be maximized. The division of vertices over the two parts in the graph can be encoded by binary variables, each variable indicating whether a vertex is in cut 0 or 1. To illustrate, in the *MAXCUT* instance presented in Figure 3.2, an arbitrary cut $C_0 = \{v_1, v_4, v_5, v_7, v_9\}$, $C_1 = \{v_0, v_2, v_3, v_6, v_8\}$ can be encoded with $x_1, x_4, x_5, x_7, x_9 = 0$, $x_0, x_2, x_3, x_6, x_8 = 1$. This corresponds to the solution $x = 1011001010$, of which the fitness is defined by the sum of the weights all cut edges, which in this case is 19. This means that the *MAXCUT* problem can be formulated as follows:

$$f_{weighted\ MAXCUT}(x) = \sum_{(v_i,v_j)\in E} \begin{cases} w_{ij} & \text{if } x_i \neq x_j \\ 0 & otherwise \end{cases} \tag{3.4}$$

The *MAXCUT* problem is a more realistic problem than aforementioned problems and is at the basis of a variety of real-world problems. As mentioned earlier, our example where Computer Science and Mathematics students are to be split over two groups is a practical application of this problem, however also finding an optimal partition of delivery addresses to distribute over the available trucks for a package delivery company can require solving a *MAXCUT* instance.

Linkage will exist between i and j for which w_{ij} is relatively large or small, as either having the corresponding vertices in the same part or not will have a significant influence on the outcome. However, as more edges might exist with relatively large or small weights, defining an appropriate linkage model is also for this problem non-trivial. *MAXCUT* differs from the aforementioned problems, because, in contrast to the other problems, *MAXCUT* is NP-hard. Instances with fully connected graphs for $l \in \{6, 12, 25, 50, 100\}$ that were generated using the approach presented by Rubinstein were provided [51]. These are the same instances that were used in previous experiments by Bosman and Thierens [6, 7, 8, 57, 58, 59, 60].

3.1.2 Experimental Setup

The performance of LTGA is ultimately defined by how much time the algorithm needs in order to find the optimal solution. However, it is hard to base any conclusions on time measurements, as clock speeds and architectures used for experiments are ever changing. For real-world problems, this required execution time is mainly dictated by the time required for the evaluation of the fitness function. In the example of optimizing the design of an airplane wing, as mentioned in the introduction, evaluating a solution could mean actually running aerodynamically accurate simulations with specific parameter values. Therefore, in all publications about LTGA, the performance of the algorithm has been expressed by the Minimally Required Number of Evalutations (MRNE). These results have always been

accompanied by the Minimally Required Population Size (MRPS), as the number of evaluations for a specific problem is highly dependent on the population size n.

Taking this into consideration, Bosman and Thierens determined the performance of the various versions of LTGA using the following setup, which is based the binary search approach as presented by Sastry [53]. Given a specific problem with ℓ variables, the MRPS and MRNE are determined by incrementally verifying whether this problem could be solved for a given population size n. In this setup, we consider a problem solved for a specific n if and only if at least 99 out of 100 independent runs have converged towards the optimal solution. Starting for $n = 1$, if a problem could not be solved for a specific n, it is tried for $2n$. This is done iteratively until the problem could be solved for a population size that was large enough. Next, in order to determine the exact threshold at which the algorithm could solve the given problem, a bisection search is done between $max\{n_{unsolved}\}$ and $min\{n_{solved}\}$. Bisection search is implemented as follows:

Algorithm 3.1: `doBinSearch(x, y)`

```
1  if(y - x > 1)
2    middle = (x+y)/2
3    isFound = tryForPopSize(middle)
4    if(isFound)
5      return doBinSearch(x, middle)
6    else
7      return doBinSearch(middle, y)
8  else
9    return y
```

Because the algorithm is stochastic, the outcome of such a search is subject to noise. Therefore, this entire search, consisting of an incremental search and a binary search, is performed 10 times for each experiment and the median required population size is considered as the MRPS. The MRNE is then defined as the median number of evaluations out of 100 independent runs of LTGA with n = MRPS. Note that for the *NK-Landscapes* problems, 100 random instances were provided per problem dimensionality so the aforementioned 100 independent runs can be run on separate instances. For *MAXCUT* also random problem instances were provided, however, every problem instance is considered separately as this is more typical for combinatorial problems, since the difference in hardness between problem instances can vary significantly. Note that this means that the graphs for *MAXCUT* should be interpreted differently, as the results are aggregated over 10 random instances on which 10 searches are performed per problem instance. This means that the error bars are much larger for *MAXCUT*.

3.1.3 Experimental Results

Using the presented setup, tests were performed on the Java implementation in order to verify its behavior. The results of these experimens are shown in the graphs of Figure 3.3 and 3.4. These graphs show the distribution of the results found. The line in the graphs connects average values, while the ”X” symbols mark the median of the values found. These

run within the marked boundaries, that indicate the 10th and the 90th percentile. This fashion of presentation will be used throughout this report.

The results shown in Figure 3.3 and 3.4, are practically identical to what was presented by Bosman and Thierens [6, 8]. There is a slight difference in performance compared to the results as presented by Bosman and Thierens, being that the results show that our implementation performs slightly better. This is expressed in a slightly lower MRPS and MRNE, which is due to the fact that the selection step before learning the Linkage Tree (LT) was omitted, as discussed before. These results show that the implementation performs as required, supporting further analysis and development.

Figure 3.3: MRPS for the Sequential Implementation of LTGA.

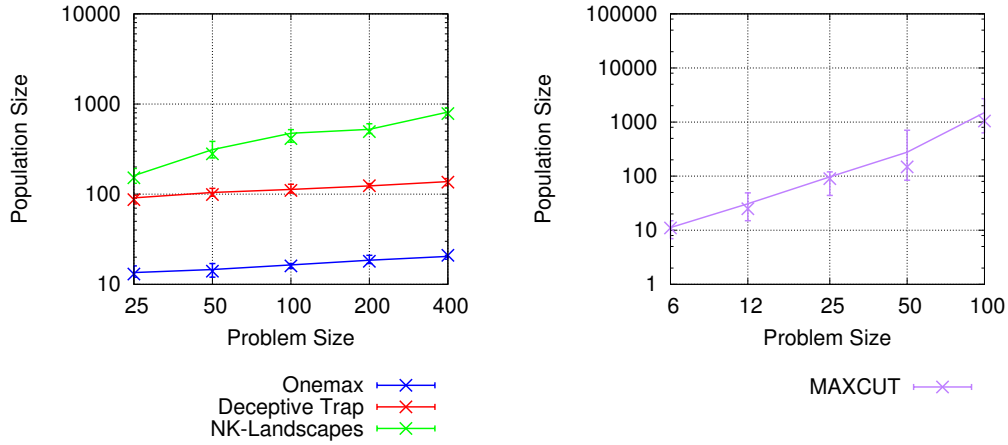
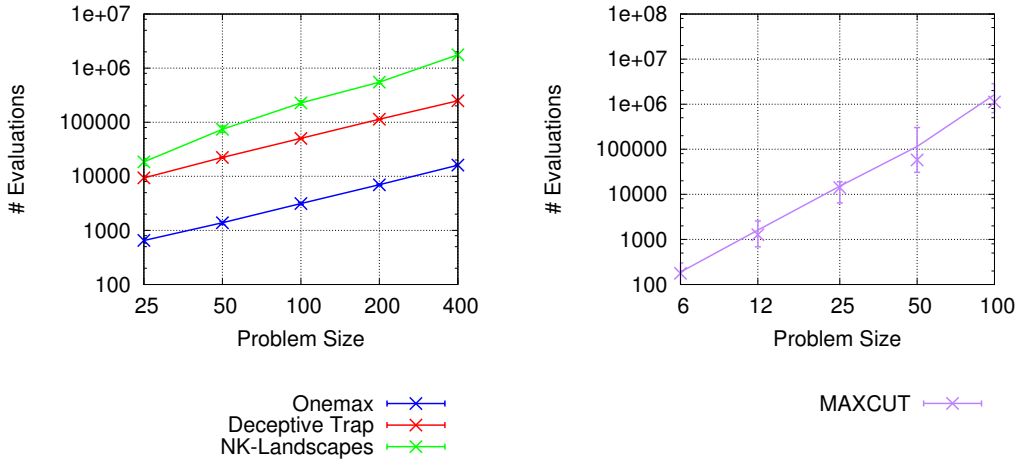


Figure 3.4: MRNE for the Sequential Implementation of LTGA.



3.2 Algorithm Analysis

In order to increase the performance of an algorithm, it is required that one first determines where possible performance gains could be achieved. In this section, an analysis of the algorithm will be given both in terms of complexity and costs, where costs are here defined as the actual execution time needed. Besides the algorithm's complexity, these costs are analyzed as they may not always comply with the theoretical complexity, for instance when a component with a low order of complexity entails costly data access or network communication. In order to implement parallelization efficiently, it is important that both poorly scalable and blocking elements in the algorithm are identified.

3.2.1 Complexity analysis

A complexity analysis is done in order to determine what elements in LTGA are most expensive in terms of computational complexity. When a component in LTGA has a high complexity, this primarily means that it is less scalable than other components, meaning that as a parameter related to the problem size increases, the execution time required by this component will increase faster than the required execution time of components with a lower computational complexity. The pseudo code of the complete implementation, as well as the complete complexity analysis can be found in Appendix B. As shown, the complexity is expressed with the use of 4 parameters:

ℓ = number of variables
n = population size
g = number of generations needed
f = fitness function evaluation time

It is important to note that these parameters are all but independent of each other as presented by Bosman and Thierens [6, 8, 56, 60]. At this point of time, the following correlations are known:

1 Between ℓ and n: As ℓ increases, the problem gets harder, meaning that the chance of finding the optimal solution with a particular value for n decreases. As n increases for a fixed ℓ, the chance of finding the optimal solution increases. Ergo, as ℓ increases, the MRPS for finding the optimal solution with a certain probability increases as well.
2 Between ℓ and f: For most problems it holds that as ℓ increases, f increases as well.
3 Between ℓ, n and $g_{optimalSolution}$, where $g_{optimalSolution}$ is the number of generations needed in order to find the optimal solution: for larger ℓ, $g_{optimalSolution}$ increases, however as n increases, $g_{optimalSolution}$ decreases.
4 Between n and $g_{convergence}$, where $g_{convergence}$ is the number of generations needed for the algorithm to converge: as n increases, $g_{convergence}$ also increases.

As no explicit formulas for these correlations are known at this point in time, the time complexity can best be expressed by means of these four parameters.

From the complexity analysis it shows that, in terms of complexity, the algorithm can be split in two main parts for each generation, being the construction of the LT and performing

Gene-pool Optimal Mixing (GOM). This is also backed by Pelikan, who stated that *"from the perspective of time complexity, the building of the LT and the evaluation of candidate solutions are the two computational bottlenecks of LTGA."* [47]

As shown, the total time complexity of building the LT is:

$$O_{BuildLT} = 3O(\ell) + O(n * \ell^2) + 2O(\ell^2) + 2O(1) = O(n * \ell^2) \tag{3.5}$$

This is because filling the distance matrix has the highest time complexity. Note that constructing an LT based on a distance matrix, as done in the block starting from line 16 in Appendix B, might seem to have a complexity of $O(\ell^3)$, which would result in a total time complexity for constructing the LT of $O(n * \ell^2 + \ell^3)$. However, as shown by Gronau and Moran, these steps have a complexity of $O(\ell * 2(\ell - 1)) = O(\ell^2)$ [26]. This is because determining the nearest neighbor and adding this cluster to the chain has $O(\ell)$ time complexity, which is done at most $O(2(\ell - 1))$ times: each cluster in the Marginal Product Model (MPM) is added at most 2 times and is removed at some point, not to be added again as it got merged with another cluster. Therefore the time complexity of constructing the LT is as stated above.

Obviously, the time complexity of the GOM phase is primarily determined by the loop that fills O with improved solutions. The complexity of this loop is given by Formula (3.6). As this loop is iterated over n times, Formula (3.7) denotes the total time complexity for GOM.

$$O_{ImproveSolution} = 4O(\ell) + 2O\big(\ell(\ell + f)\big) + 5O(1) = O\big(\ell(\ell + f)\big) \tag{3.6}$$

$$O_{GOM} = O\big(n * \ell(\ell + f)\big) \tag{3.7}$$

As shown in the pseudo code, there are more statements that have a non-trivial time complexity, however they do not influence the total time complexity of LTGA. This is shown by the formula below, which defines the total time complexity of LTGA. As can be seen, eventually the GOM phase has the highest complexity per generation and is therefore dictative for the total time complexity of LTGA.

$$O_{LTGA} = O(n*\ell) + O(n) + g\Big(O(n*\ell^2) + O\big(n*\ell(\ell+f)\big) + O(n) + O(1)\Big) = O\big(g*n*\ell(\ell+f)\big) \tag{3.8}$$

3.2.2 Costs analysis

As discussed in the previous section, the GOM phase is the primary element that affects the total computational complexity of LTGA. However, this does not always mean that this part will require the most computation time, compared to other non-trivial components. Therefore, we decided to profile LTGA's implementation.

Profiling Setup

As a start, several profilers were investigated, aimed at finding a profiler that could provide us with tools to give an insight in the required computation time of LTGA's components.

More specifically, we were looking for a profiler that could provide us with statistics such as the number of function calls and the total time spent per function. Additionally, a profiler was preferred that could be used using a command line interface, as the 64-core servers were only accessible through SSH. Finally, the overhead imposed by the profiler is also important. Profilers usually work by injecting additional code at the start and end of functions, in order to track calls to these functions. Obviously, this imposes a certain overhead, however, a profiler with minimal overhead is desired, as significant overhead will reduce the reliability of the test results.

The first profiler we therefore investigated was the profiler that comes with the Java Development Kit (JDK), called HPROF [42]. HPROF was easy to use, already installed, met all the requirements mentioned above and could save results to file. Some initial experiments showed, however, that HPROF came with overhead that was vastly greater than expected. Experiments showed that when HPROF was enabled, on average the execution time of our implementation increased by roughly a factor 20. As HPROF still based its results on the total execution time, this meant that the reliability of the returned results was diminished, as about 95% of the total execution time was imposed by HPROF itself.

Therefore, we investigated other profilers, among which were DJProf [36], Oktech-profiler [39], Perf4j [49], JRat [54], EJP [62], JConsole [45] and JIP [37, 64, 65]. Of these profilers, the Java Interactive Profiler (JIP) seemed like the most promising alternative as it was best recommended across several Java profiler comparisons and because it was well documented. It claimed to have the same abilities as HPROF, however with less overhead while providing additional features such as the ability to enable or disable the profiler while the algorithm is running. It was claimed that this does not influence the results returned, as JIP does not take execution time in account that was consumed by the profiler itself.

The overhead of the profiler is claimed usually to be around 1-3%, with a worst case scenario of 100%. Initial profiling gave unexpected results, however, showing that simply counting the frequency of variable values in the population, in order to calculate the Joint Entropy, required 46.7% of the total execution time. Additional tests showed this was merely caused by the vast amount of function calls to the corresponding `updateFrequencies` function, rather than the functionality contained by this function. As it was not expected that this vast amount of function calls would require 46.7% of the execution time, we concluded that also the overhead introduced by JIP was too large and would significantly influence our tests results, leaving us with little reliable information about possible bottlenecks in the algorithm.

As a result, we implemented a minimalistic profiling framework, aimed at simplicity and minimal overhead. This was done by simply injecting function calls at the begin and end of every function we wanted to profile, that solely saved the start and end time stamp. The differences in time between the beginning and the end of the function, as well as the call frequency were saved. Upon termination, this information was used to calculate the profiling statistics required. This should mean that the introduced overhead is minimal, also because of all the functions in the code base, probably only a handful would be interesting to profile. With some experimenting, these functions could be tracked down and functions that were trivial in terms of computation time could be omitted from profiling. On average, no significant overhead was encountered when using this profiling implementation.

This profiler was used for all time-related experiments presented. As mentioned above, only functions were tracked that showed to have a significant impact on the execution time. This means that minor discrepancies may be encountered in the figures that will be presented. These are caused by the execution time required by functions that were not tracked by the profiler, but should never be of any importance to the conclusions drawn from these experiments.

Profiling Results

As a first start, a cost analysis was done of the sequential implementation of LTGA. The average values that were found while profiling 10 executions of LTGA on 100 independent *NK-Landscape* problem instances with $\ell = 400, n = 400$ can be found in Appendix C, in Table C.2. This table shows the call count and the execution time that profiled functions consumed. Note that some functions are indented, which means that they are part of the first function above this line that is less indented, e.g. `constructMIMatrix` is called by `learnStructure`, which means that the time shown for `learnStructure` consists of the time consumed by `constructMIMatrix`, plus the time consumed by any other statements in `learnStructure`. Also note that there are 3 lines for the `constructMIMatrix` function. This is because after some initial profiling, it became clear that this was one of the bottlenecks. It was not certain, however, if this was either caused by filling the Mutual Information Matrix (MIMatrix) with entropy values, or by iterating over these values in order to calculate Mutual Information (MI) values. Therefore we traced these two functionalities separately, which means they are represented by the `constructMIMatrix - Entropy` and `constructMIMatrix - MIValues` record respectively. Last, it is important to note that the results in this table are based on the implementation of LTGA, after some initial optimizations were done, which will be discussed in the next chapter.

When performing this profiling, it became clear that there are two main bottlenecks: filling the MIMatrix with entropy values and generating new solutions by performing GOM, of which the consumed execution time was mainly imposed by evaluating the fitness of possible solutions. Note that this profiling is done with *NK-Landscapes* instances, for which the evaluation of a solution is slower than that of problems with less complex evaluation functions, such as *Onemax* or *Deceptive Trap*. When evaluating more realistic problems, however, it can be assumed that evaluating the fitness of a solution will take even more time, as for instance more data has to be processed, or, even worse, a complete simulation has to be run. This would drastically increase the required computation time for generating new solutions, increasing the impact of GOM on the total execution time even more.

To clarify, these results do not give us insight in the scalibility of the algorithm, nor does the complexity analysis give a definite answer to the question of what components require the most computation time. These two combined, however, give a good indication of what components are bottlenecks in the algorithm. As discussed in the previous section, the complexity analysis showed that there are two components in the algorithm which showed to be most complex: building the LT and generating new solutions. Eventually, generating new solutions determined the overall time complexity of LTGA. The costs analysis as discussed above also shows that generating new solutions dominates the execution time of the

algorithm, although filling the MIMatrix with entropy values, which is part of generating the LT, has shown to be also computationally intensive. As the results of these two analyses overlap, it can be concluded that filling an MIMatrix with entropy values, as well as improving solutions using GOM, have the most significant impact on LTGA's performance, which answers the the first research question posed in Section 2.3. These components of LTGA should be focal points when improving the performance of the algorithm, as they account over 99% of the execution time and form a significant scalibility impediment.

Chapter 4

Algorithm Parallelization

In the previous chapters, the internals and context of the Linkage Tree Genetic Algorithm (LTGA) have been described, as well as some of its shortcomings. In this chapter, the parallelization of LTGA will be discussed that will ultimately enable us to leverage more computational power in order to address new and more complex problems. More specifically, these algorithm extensions should support us in finding a more suitable problem structure representation, ultimately providing insight into possible improvements for the Family of Subsets (FOS) used by LTGA.

4.1 Code Optimizations

As described in the previous chapter, the LTGA was translated from C to Java 8. This was initially done by implementing a translation that was as close to the C implementation as possible, while respecting best practice coding principles and the Object Oriented Programming principle. After this implementation was finished and verified, a critical review was done to eliminate any inefficiencies in the implementation as these could impede the eventual performance gain that could be achieved. The most important optimization that was implemented, considered the implementation of the *NK-Landscapes* fitness evaluation function. This function was based on lookup tables of the provided instance files that provided a mapping of all possible compositions of variables $x_{i,i+1,\ldots,i+k}$ to double values for each f_{NK}^{sub}. The initial implementation was based on Java HashMaps as this data structure best reflected the structure of the data provided. When performing tests, however, the difference in required fitness function evaluation time for *NK-Landscapes* compared to problems such as *Onemax* and *Deceptive Trap* was significant. Obviously, a higher evaluation time was expected compared to *Onemax* and *Deceptive Trap*, as these are merely mathematical formulas that hardly need to access any data, other than the solution that is to be evaluated. However, the evaluation of solutions against the *NK-Landscape* problem seemed rather inefficient, which is why we changed the data structure used by this fitness function.

As described, the NK-Landscape lookup tables consisted of a mapping from all possible compositions of variables $x_{i,i+1,\ldots,i+k}$, stored as *bit strings*, to double values for each f_{NK}^{sub}. These bit strings are all of a fixed length k and a lookup table contains values for all possible

values of such a fixed-length bit string, meaning that it contains 2^k entries. As these bit strings can be seen as binary representations of integer values from 0 to $2^k - 1$, such lookup tables can also be saved in Arrays of doubles. Values for each f_{NK}^{sub} can then be obtained by converting the bitstring containing the variable values for $x_{i,i+1,\ldots,i+k}$ to an integer and accessing the array of a f_{NK}^{sub} at this position. This implies more efficient memory usage, however the biggest gain is achieved by now being able to skip the hashing step used by HashMaps to determine the physical memory address of the requested value [50, 41].

Apart from this optimization, several smaller improvements were implemented, although these merely consist of code simplifications, which usually are expected to be done immediately while implementing an algorithm. Together, however, these optimizations caused a significant performance improvement, as illustrated by Table C.1 and C.2. Both tables show profiling results averaged over 100 independent *NK-Landscape* instances with $\ell = 400$, $n = 400$. Table C.1 shows results for the initial implementation, in which no optimizations were performed yet, while Table C.2 shows results for the implementation on which the previously described optimizations are performed. It is clear that these optimizations cause a severe improvement in the algorithm's performance, as shown by the reduction of the total execution time by more than 90%, caused by the reduced fitness evaluation time. With this, a severe impediment in harnessing the available computational power was removed.

4.2 Parallelization Approaches

After these code optimizations were implemented, providing a sound basis for taking the next step to a more powerful algorithm, we focussed on harnessing the available multiprocessor computation power, which was the first goal presented in Section 2.3. When considering parallel algorithms, one can either focus on *implementation parallelization* or *problem parallellization*, which will both be discussed in this section, answering part of our second research question also posed in Section 2.3.

4.2.1 Implementation parallelization

We investigated the concept of *implementation parallelization* as presented by Grama et al. [25]. Implementation parallelization consists of redesigning an existing sequential implementation such that the workload of computationally expensive components can be spread over multiple processors. The reduced execution time that can be achieved by doing this is defined by Brent's scheduling principle, which states [9]:

$$T_p(n) \leq \frac{W(n)}{p} + T_s(n)$$

Where:

$T_p(n)$: Time complexity of the parallel algorithm
$W(n)$: Work complexity of the algorithm
p: Available processors
$T_s(n)$: Time complexity of the sequential algorithm

Note that throughout this work, CPU cores in the multi-core architecture used are seen as virtual processors, meaning that when the term *processors* is used, this refers to CPU cores in the architecture used.

Parallelizing an implementation starts with making a clear division between parallelizable and unparallelizable components in an implementation. Parallelizable components are components that consists of smaller elements that can be executed in parallel. This requires that these smaller elements are either independent of each other, or a decent messaging infrastructure can be implemented to exchange required information between processors. These parallelizable components are, in the field of parallelization, the components that are to be focussed on as the workload that they embody can be distributed over multiple processors, reducing the total execution time.

Cantú-Paz published an extensive study on parallel Genetic Algorithms (GAs) [10], in which he shows that implementations of GAs are generally very suitable for parallelization, because the solutions in a population can usually be evaluated and mutated completely independently. Therefore, creating new solutions can often be done in parallel. Cantú-paz divides parallel GAs in 3 different classes:

1 **Global single-population master-slave GAs**: GAs that evolve one population at a time and distribute the evaluations of fitness functions over multiple processors according to a master−slave architecture, where one master processor distributes the workload by submitting tasks to slave processors.
2 **Single-population fine-grained GAs**: GAs that evolve one population at a time, however evolving a population is divided over multiple processors.
3 **Multiple-population coarse-grained GAs**: GAs that evolve multiple populations over multiple processors, but exchange individuals occasionally, which is called *migration*. The internal logic in multiple−population coarse−grained GAs is different and therefore the behavior of this class of parallel GAs is different from the aforementioned classes.

Frameworks exist that support implementing parallel Evolutionary Algorithms (EAs). An example of such a framework is the Distributed Evolutionary Algorithms in Python (DEAP) framework that aims at rapid prototyping and testing of ideas for EAs in an explicit and transparent way [16, 23]. This is done by providing a predefined infrastructure in which the specific functionalities of an EA can be inserted. Although experiments and documentation on DEAP show that this framework makes it very easy to quickly implement an parallel EA, we decided not to use it. This was mainly done because of performance reasons, as C and Java are at this moment considerably faster than Python, but also because luxuries that come with a framework usually also come with the costs of additional overhead [20]. Therefore we felt that more performance gain could be achieved when custom parallelization was done that would suit the current implementation of LTGA best.

4.2.2 Problem Parallelization

Problem parallelization is a parallelization approach that is aimed at problems for which a straight-forward implementation for solving the problem is not suitable for implementation

parallelization. A trivial example of such a problem is generating a sequence R consisting of the cumulative sums of a sequence S. JáJá describes various ways to solve such problems in parallel [35]. However, we felt that the most computationally intensive components of LTGA were very suitable for implementation parallelization and as this was also backed by Cantú-Paz [10] and Thierens [56], no further research was done in this field as this did not seem relevant enough.

4.3 Perfect Parallel

As discussed above, implementation parallelization seemed most suitable for parallelizing LTGA. This approach, however, can again be split in two sub-approaches, which we call *internal parallelization* and *external parallelization*. The first extension that was implemented to harness the available computational power was the implementation of internal parallelization, which is aimed at redesigning the algorithm such that the work of the most computationally intensive components can be distributed over multiple processors. This implementation we call the Perfect Parallel implementation of LTGA (PP-LTGA), as it is aimed at achieving the theoretically ideal speedup of a factor p.

4.3.1 Implementation

As shown in Table C.2, the `constructMIMatrix - Entropy` entry for the Mutual Information Matrix (MIMatrix) class and the `generateNewSolution` function in the Population class are the most computationally intensive. Note that for all the function names used in this section, we refer to this Table. These two functions make up for over 99% of the execution time in these experiments and are important components in the overall complexity of the algorithm, as discussed in the previous chapter. Parallelizing these components is very feasible, as calculating entropy values for cells in the MIMatrix can be done completely independently for each cell and also solutions in a population P can be improved independently from each other. Note that problems might exist, for which the fitness of solutions might be dependent on the evaluation of other solutions, in which case more advanced parallelization techniques might be required. As this does not hold for the benchmark problems considered so far, distributing the solutions in P across the available processors is most promising for solving these problems using internal parallelization.

Implementation of internal parallelization is supported by the use of the Java ExecutorService class, which enables the scheduling of Runnables over the available processors. A clean and clear way to submit these Runnables is with the use of Lambda expressions, which are supported since Java 8. Note that also the evaluation of random solutions when initializing the first population is parallelized, as this showed to be beneficial and parts of the design and implementation for parallelizing `generateNewSolution` could be reused. After parallelizing these three functions, other functions were also paralellized that seemed suitable for workload distribution. It turned out, however, that these parallelizations were not beneficial, which will be discussed further on in this chapter. Therefore, calculating the entropy values for the MIMatrix, as well as the generation of new solutions and the evalua-

tion of solutions in the population of the first generation are the only three components that were parallelized.

In the `constructMIMatrix` function, the workload of filling cells in the MIMatrix with entropy values was distributed statically, meaning that each processor gets a static number of cells in the MIMatrix it has to fill by simply dividing the number of cells to be filled evenly over the available processors. This way, all processors have all required information available on forehand. As mentioned before, ideally a speedup of p would be achieved, which means that as the construction of the MIMatrix is the dominating term in the computational complexity of learning the Linkage Tree (LT), the complexity of learning the LT should be reduced from $O(n * \ell^2)$ to $O(\frac{n*\ell^2}{p})$.

For parallelizing the `generateNewSolution` function, a more sophisticated approach of workload distribution was needed. While filling the MIMatrix with entropy values consists of tasks that are of virtually equal costs, the costs for generating new solutions can vary significantly. This is mainly due to the stochastic nature of Gene-pool Optimal Mixing (GOM). The variance in complexity caused by this stochastic nature is further amplified by Forced Improvement (FI) that is incorporated in GOM. To illustrate, the improvement of a solution s can simply consists of copying values from random donor solutions for all linkage sets in the LT, of which only one of these operations results in a change in s. Assume that this results in a better fitness of s, then this means that only one function evaluation was required for improving s. In terms of required processor time this would be the best case when solely considering the improvement of this single solution. In contrast, in the worst case scenario all solution mixing operations result in a change in s, requiring $\ell(\ell-1)$ fitness function evaluations. Then, assume that all these changes do not result in a better fitness. This means that they are all reverted and after traversing the LT, FI is executed doing the same operations but then with the best solution found so far as a donor. Assume that all these mixing operations again result in solutions that are different from the original, but do not have an improved fitness. FI will then finish by simply copying all values from the best solution found so far. When comparing this best case and worst case scenario, it is clear that the number of required fitness function evaluations, which have a severe impact on the total required execution time, can vary from 1 to $2\ell(\ell-1)$, simply because of the stochastic nature of GOM. Additionally, for other problems than the ones presented in the previous chapter, it is possible that the fitness function evaluation time f can also vary significantly which further amplifies this variance in time required for the improvement of a single solution.

Evidently, the approach as presented for filling the MIMatrix with entropy values is not suitable for this situation, as dividing the number of solutions that have to be generated statically over the available processors will imply that severe variance of required execution time among processors is to be expected. Chances are high that there will be a significant difference in time between the termination of the first finished processor and the last finished processor, meaning that processors are idling while still tasks are available that need to be processed. This would prevent PP-LTGA from achieving an optimal speedup of a factor p, as the GOM phase is only as fast as the processor that will need most time to finish its assigned tasks.

Therefore a task pool architecture was implemented to reduce this effect. Before dis-

patching tasks to the available processors, a pool of all solutions in P is made that still need to be improved using GOM. Upon starting GOM, every processor will request a solution from this pool and improve it. When a processor is finished with improving its solution, it will store it in O and request a new solution from this pool to improve until the pool is empty. When the pool is empty and all processors are done, all solutions in the current population are improved and GOM is complete. With this implementation, in the worst case all but the largest task, being the improvement of a solution that eventually required most time, are finished at the same time, after which this largest task is submitted to one of the processors while all other processors are idling. With this approach, the possible processor idling time is significantly reduced, providing a better support for LTGA to approach an ideal speedup of a factor p. In the worst case scenario as illustrated above, the complexity of improving all solutions in P can be reduced from $O((n-1)f_{average}+f_{max})$ to $O(\frac{n-1}{p}f_{average}+f_{max})$, where $f_{average}$ is the average fitness function evaluation time of all but the most expensive fitness function evaluation and f_{max} is the fitness function evaluation time of the most expensive fitness function evaluation.

The parallelization of filling the MIMatrix and generating offspring results in a *single-population fine-grained GA*, according to Cantú-paz [10]. An important difference between the description that was given by Cantú-paz and our implementation, however, is that in the original description, individuals in a population are only allowed to mate with their neighbors, while in our GOM implementation, solutions are mixed with random donors.

4.3.2 Experimental Results

With the implementation discussed in the previous section, experiments were performed to determine the effect of internal parallelization on the required execution time. This was done for all benchmark problems presented before, for varying values for ℓ and p. Experiments were performed with optimal values for the population size n, being the Minimally Required Population Size (MRPS) as presented earlier in Figure 3.3. In the experiments performed, LTGA is run until it was fully converged. The average required execution time for 100 independent runs is saved for every problem size, for various numbers of processors used. The results found are shown in Figure 4.2. Note that for *NK-Landscapes*, the average over 100 independent randomly generated instances was taken, while for *MAXCUT* the results shown consider 100 independent runs on one specific *MAXCUT* problem instance per problem size, being the instance of which the MRPS was the median of the MRPSs of all 10 problem instances for that problem size. For future reference, these instances are displayed in Table 4.1.

The graphs in the left column of Figure 4.2 show the execution times that were found during the experiments, while the right column shows the speedup factor that was achieved compared to the sequential implementation. This provides insight into how the execution time of our parallel implementation relates to the original implementation, as questioned by the second research question in Section 2.3. As a reference, the execution times for the sequential implementation for the same experimental setup as discussed above are shown in Figure 4.1.

Table 4.1: *MAXCUT* instances of median difficulty.

ℓ	**Instance filename**
6	n0000006i01
12	n0000012i02
25	n0000025i05
50	n0000050i06
100	n0000100i02

Figure 4.1: Execution Time for the Sequential Implementation of LTGA.

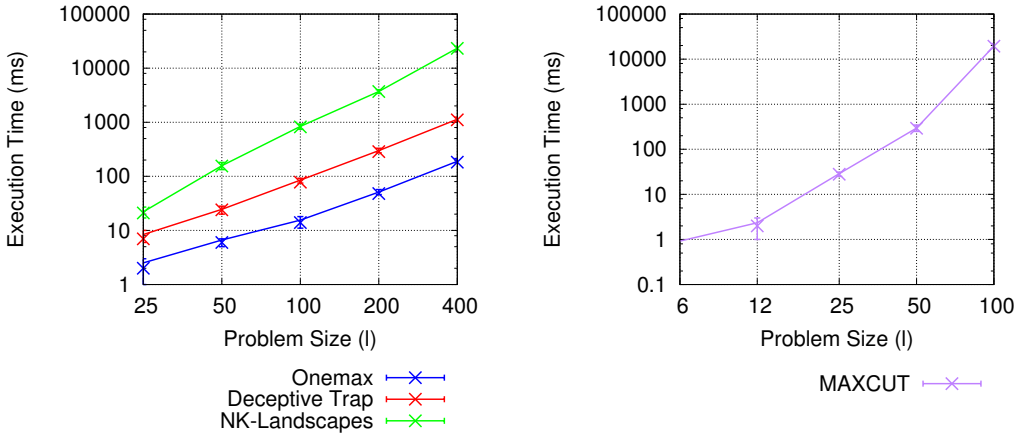


Figure 4.2 clearly shows that parallelization introduces significant overhead as even when using only one processor, PP-LTGA performs significantly worse than the sequential implementation of LTGA for small problem sizes, while apart from managing a separate thread, it is actually identical to the sequential implementation of LTGA. The effect of this overhead clearly increases as the number of processors increases since for small problem sizes the required execution time even increases, which is contrary to our expectations. For more complex problems, being for $\ell = 100$ for *MAXCUT* or for $\ell = 400$ for other problems, the results show that dividing the workload over the available processors can sometimes be beneficial, however still the speedup is far from the ideal speedup.

Further investigation was therefore done to determine what prevented us from achieving the desired speedup. Several causes were found, which will be discussed below. It is important to note, however, that the overhead introduced by the Java thread scheduler is at the root of several of these causes, which is why a short brief description of this overhead is discussed first.

Java Thread Scheduler Overhead

Experiments with the Java thread scheduler were performed on the 64-core server and show that solely the start up and termination of 1 empty thread requires 1.3 ms on average, while

starting up and terminating 64 empty threads even requires an average of 15.5 ms. This overhead of thread management is to be called significant compared to the execution time of the sequential implementation of LTGA as per generation $2p$ additional threads are used each generation and typically it holds for the number of generations g that $0 < g < 15$. As an example, *Deceptive Trap*, with $\ell = 25$ and MRPS $= 87$, typically requires 6 generations, which means that even when using only 1 processor, the PP-LTGA will need an additional 15.6 ms for thread management, while the sequential implementation of LTGA only requires a total execution time of 8.39 ms. When using 64 processors, PP-LTGA requires an impressive additional 186 ms due to thread management. We will show that causes 1, 2 and 4 are strongly related to this overhead and influence the magnitude of the impediment this overhead will be on the performance of LTGA.

Cause 1: Problem Size

For most of the problems shown in Figure 4.2, the main impediment on reaching the ideal speedup is the fact that these are toy-size problems. These problems do not entail enough work to be distributed efficiently over the available processors as the overhead introduced by the Java scheduler is too significant compared to the total time required when the problem would be solved by the sequential implementation. For instance, in the case of *Deceptive Trap* with $\ell = 25,$ MRPS $= 87$, as presented above, the sequential implementation of LTGA only requires 8.39 ms on average to terminate. In this case, the overhead as presented above is not just significant, but in fact dominating the required execution time of PP-LTGA, preventing it from achieving the optimal speedup and in fact making parallelization in this situation detrimental to the algorithm's performance.

Cause 2: Granularity

An important aspect of parallel algorithms is the granularity in which the algorithm is parallelized. We define granularity as the size of the chunks of parallel processable tasks. A chunk is a set of tasks that are processed sequentially by one single processor, meaning that multiple chunks of parallelizable tasks can be processed in parallel when using a multiprocessor architecture. The largest possible granularity is when all tasks are contained by one chunk, which is equivalent to a sequential implementation, not considering thread management. In contrast, the smallest possible granularity is when each parallel processable task is contained by a separate chunk. Ideally, the amount of parallel processable work is divided over the number of processors such that each processor gets exactly the same amount of work as only then a theoretical speedup of p can be achieved. Furthermore it is important to note that, in order to reduce the overhead of workload distribution, introduced by thread management, the granularity should be as large as possible, while still enabling an even distribution of the workload over the available processors.

Parallel processable tasks for filling the MIMatrix are divided into chunks statically as described before. Static workload division is suitable for components that consist of parallel processable tasks of which the (relative) execution time is known on forehand. In this situation, the granularity can be determined such that the workload can be distributed over the

available processors as evenly as possible, which in the case of filling the MIMatrix means that p chunks exist, each embodying the calculation of $(\ell^2 - \ell)/2p$ cells in the MIMatrix.

In contrast, the (relative) execution time of a single processable task in GOM, in our implementation consisting of the improvement of a single solution, cannot be known on forehand due to its stochastic nature and therefore, a more dynamic approach for workload division is chosen, as discussed before. In this situation, when choosing the granularity, there is a clear trade-off between workload distribution overhead and processor idling time. A larger granularity decreases the workload distribution overhead, however increases the average time that processors will be idling while still parallel processable tasks are available. Decreasing the granularity decreases the processor idling time, however also imposes a larger workload distribution overhead. In our implementation, we have chosen for the smallest possible granularity on the level of improving single solutions as for more complex problems the overhead imposed by the dynamic workload distribution will be negligible compared to the fitness function evaluation time and the level of improving single solutions is a natural point to initiate parallelization [34]. However, in this implementation, still a certain overhead is experienced which prevents PP-LTGA to achieve its optimal speedup factor. This effect decreases as the complexity of the problem increases, as also shown by our results.

Cause 3: Parallel Data Access

The third cause that was found was simultaneous data access operations of multiple processors. Experiments were performed in which all values from an array of 64.000 integers are accessed. The values read from this array are disposed immediately so the execution time in these experiments solely consists of memory access costs and thread management costs. When doing this sequentially, only 2 ms is needed. However, when 64 processors all read all values from the array in the same, consecutive order, this requires an impressive 235 ms, excluding the time needed to start up and terminate one thread per processor. When these 64 processors all access a separate part of the array, containing 1000 integers, still 12 ms is required. This shows that when data is accessed in the memory simultaneously, additional overhead can be expected, especially when the same physical memory addresses need to be accessed.

This is applicable to filling the MIMatrix in parallel as for each cell in the MIMatrix, over all solutions in P is iterated in the same order to calculate the entropy value. However, also in other parts of PP-LTGA simultaneous data access is causing overhead, such as in the fitness function for *NK-Landscapes*, which iterates over the same lookup tables for every function evaluation. Experiments were done to prevent simultaneous memory access of the same physical addresses by introducing redundancy by providing each processor with its own copy of the data it needs. Copying this data, however, requires more time then the overhead introduced by simultaneous data access. Another possible approach is advanced low-level coordination of memory access. However, this also comes with additional overhead at a low-level which is expected to generate significant overhead overall.

Cause 4: Remaining Sequential Components

As mentioned above, components were parallelized that showed to have most impact on the required execution time. Although less significant, still components remain that are executed sequentially, preventing PP-LTGA from achieving optimal speedup. Some of these components were theoretically suitable for parallelization, such as the `updateMIMatrix` and `constructNewMpm` functions, meaning that they consisted of smaller parallel processable tasks. However, experiments showed that parallelizing these components is unfruitful, as no reduction in required execution time could be achieved due to aforementioned causes.

Conclusion

The causes described above explain why the theoretical optimal speedup of a factor p cannot be achieved with the implementation of PP-LTGA as presented. Considering the time available, we found that with PP-LTGA as presented above, encountered trade-offs were carefully considered, resulting in an implementation that on average will perform well across a wide variety of situations and can support us in solving more complex problems in the future.

4.4 Embarrassingly Parallel

The second approach for harnessing the computational power available, was implemented by simply executing multiple instances of the LTGA in parallel and considering the best found solution over all these instances as the final solution found. This implementation of *external parallelization* is populairly called Embarrassingly Parallel (EP). The power of the Embarrassingly Parallel implementation of LTGA (EP-LTGA) is that, because it takes the best solution over all instances, the chance of finding the optimal solution for a single instance is allowed to be lower. This means that a smaller value for n will suffice for these instances, causing a reduction in the MRPS and therefore the required execution time. Therefore, by executing instances of the LTGA over all processors available with a smaller value for n, EP-LTGA is able to find the optimal solution while requiring less execution time. The correlation between the population size and the chance of finding the optimal solution as well as the correlation between the population size and the required execution time could not be accurately quantified so far, which means that no exact expression of the expected speedup could be constructed. Nevertheless, experimental results will also include the encountered speedup as also constructed for PP-LTGA, in order to make a good comparison between the scalability of these two parallel implementations.

Note that it is expected that there will be a limit to the reduction in MRPS and with that the execution time, because in the end a minimum population size of 1 is required in order to let the algorithm return a solution at all. In this situation the instances are not able to improve their one solution due to the lack of donors when performing GOM. This means that this limit for the MRPS will be reached when a number of processors is used that is large enough to support a 99% chance of finding the optimal solution, simply by generating p random solutions, as the MRPS represents the minimum population size for which at least

a chance of 99% exists of finding the optimal solution as discussed in Section 3.1.2. From that point on, using more processors will for sure impose a larger execution time due to increasing multi-threading overhead.

4.4.1 Implementation

As illustrated above, EP-LTGA rather consists of starting up p instances of LTGA. Because of the Object Oriented design of the code base, this only required the implementation of a wrapper class that could do this and collect the results when all instances were finished. When again looking at the terminology used by Cantú-paz, EP-LTGA can be classified as a multiple-population coarse-grained GA [10]. Again there is a slight difference between the description provided by Cantú-paz and our implementation, as in our implementation the instances operate completely independent and do not exchange individuals during the execution, because it was expected that this requires thread synchronization, which would introduce significant overhead.

4.4.2 Experimental Results

With this implementation, experiments were done in order to see what speedup the EP-LTGA could entail. However, as the behavior of this implementation is different from the implementation of the original LTGA and PP-LTGA, it was first important to determine the MRPS, as the potential of parallelization can best be measured when the algorithm parameters are optimal. Determining the MRPS and Minimally Required Number of Evalutations (MRNE) was done with the same experimental setup as described in section 3.1.2. Figure 4.3 clearly shows how the use of multiple instances of the LTGA simultaneously can significantly decrease the MRPS and MRNE. Note that this effect is the strongest for the *MAXCUT* problem, as all corresponding graphs show a stronger decreasing MRPS and MRNE as compared to other problems. Also note, however, that eventually all graphs seem to stagnate, which we think is due to the theoretical limit as discussed above, which we think it asymptotically converges towards.

In the end, however, the performance of this implementation will be measured in terms of required execution time. Note that although the decrease in MRPS and MRNE is promising, the algorithm is still as fast as the longest instance running, which poses the question how much performance increase actually can be achieved. Therefore, experiments have been performed on measuring the required execution time, of which the results are shown in Figure 4.4. The times measured again consider the average of 100 independent runs of EP-LTGA using n = MRPS.

For the smallest problem instances processed by EP-LTGA, it is shown that generally the use of Embarrassingly Parallel (EP) is not beneficial as no speedup could be achieved compared to the sequential implementation of LTGA and execution times further increase as p increases due to additional overhead. For more complex problems, however, the results show interesting behavior. It seems that that at first, the execution time decreases as more processors are used. This effect stagnates, however, as p increases, meaning that at some point, the minimum required execution time possible for this problem for EP-LTGA

is reached, after which the execution time increases again. Moreover, it seems that as the complexity of the problem increases, this minimum moves towards a higher number of used processors. This can be explained by the observation that, although decreasing, the MRPS and MRNE for EP-LTGA stagnate as p increases, as shown in Figure 4.3. With that, the reduction in required execution time also stagnates while at the same time the overhead imposed by multi-threading increases, as also discussed for PP-LTGA. Graphs in Figure 4.4 show that at some point, the decrease in MRPS and MRNE cannot compensate for the increase in overhead of using more processors, though as the size and complexity of the problem increases, this overhead becomes smaller compared to the execution time needed by the LTGA instances, meaning that this overhead will only be dominant for a larger number of processors. These findings are also reflected by the speedup that was encountered compared to the sequential implementation, as shown by the graphs in the right column of Figure 4.4.

4.5 Parallel Implementation Comparison

For both PP-LTGA and EP-LTGA it is shown that, as the complexity of the problems increases, their potential increases, showing a significant speedup compared to the sequential implementation of the LTGA. However, in order to solve the problem of finding the optimal LT replacement, as will be discussed in the next chapter, we need to investigate whether one implementation can be preferred over the other. As both implementations seem to overcome impediments of parallelization as the problem size increases, experiments were performed with the largest problem instances we currently have to our disposal that mimic more realistic problems, being problem instances for the *NK-Landscapes* problem for $\ell = 1600$. Results of these experiments are shown in Figure 4.5 that again shows execution times averaged over 100 independent runs.

These graphs clearly show that PP-LTGA is more scalable than EP-LTGA as it not only outperforms EP-LTGA, but also shows a stronger and more persistent decrease in execution time as the number of processors increases. With that, PP-LTGA is more capable of leveraging the computational power available, with which our first goal as presented in Section 2.3 is achieved. PP-LTGA will therefore be used to solve considerably complex problems in all future experiments.

Figure 4.2: Execution Time and Speedup for PP-LTGA.

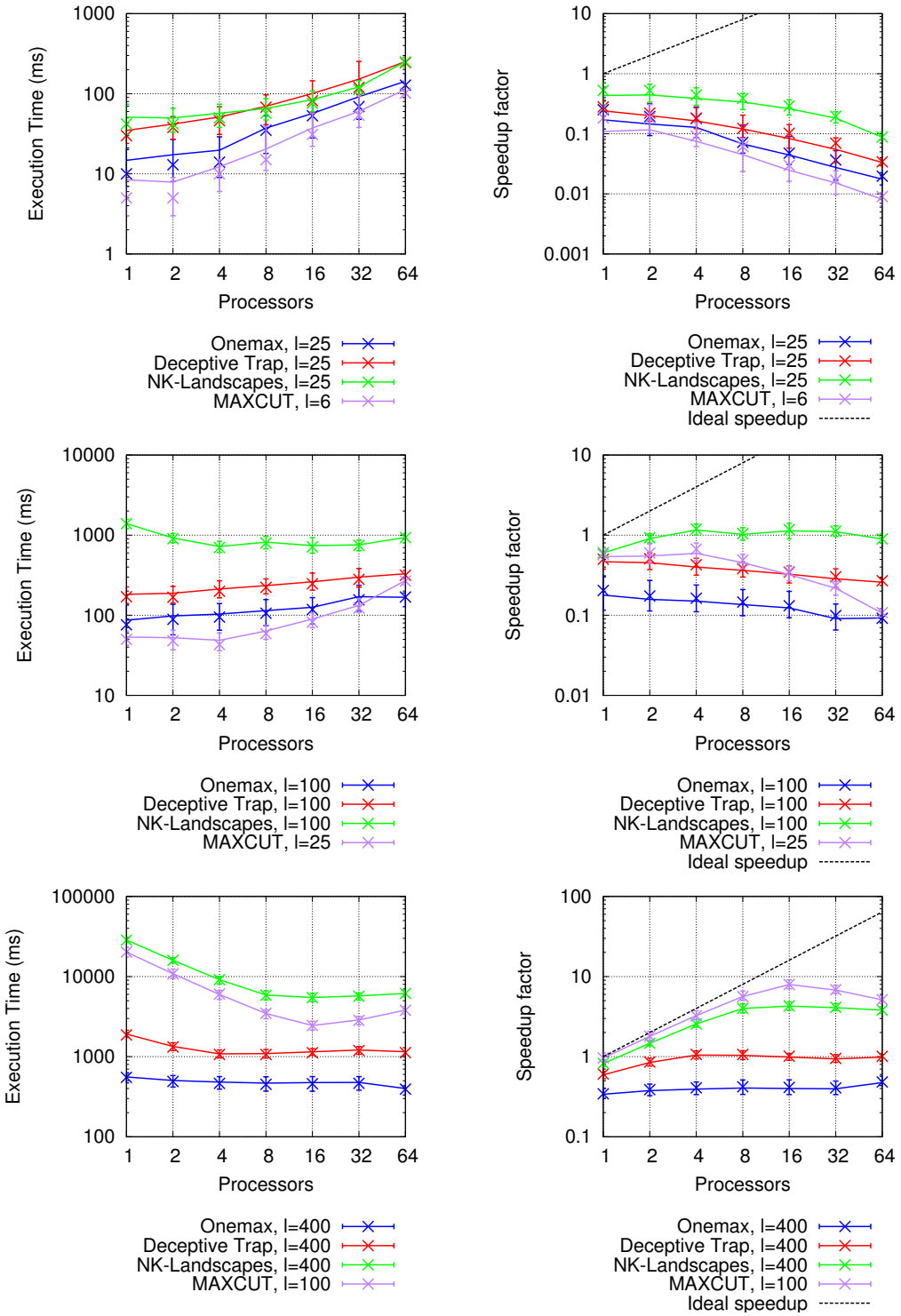


Figure 4.3: MRPS and MRNE for EP-LTGA.

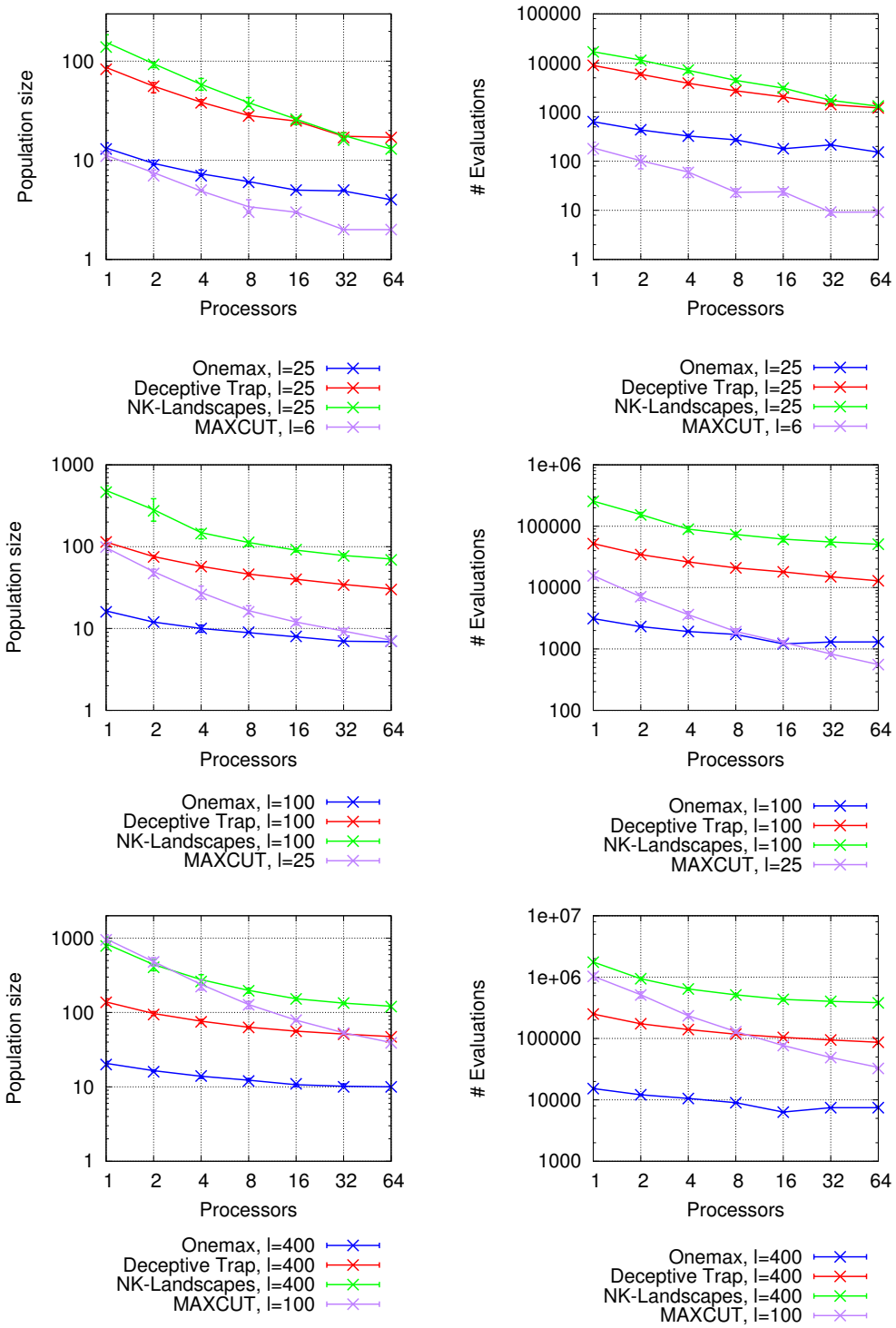


Figure 4.4: Execution Time and Speedup for EP-LTGA.

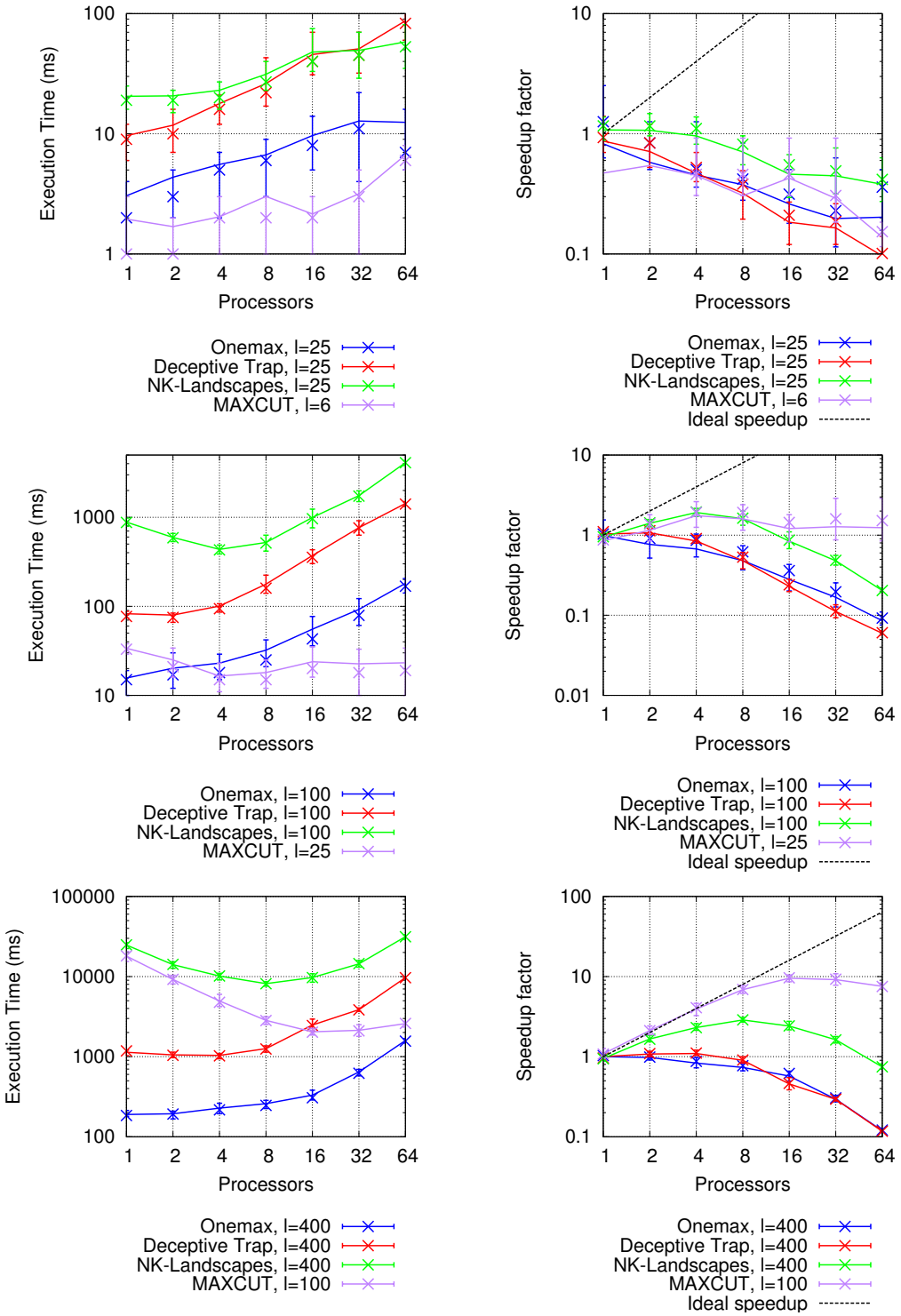
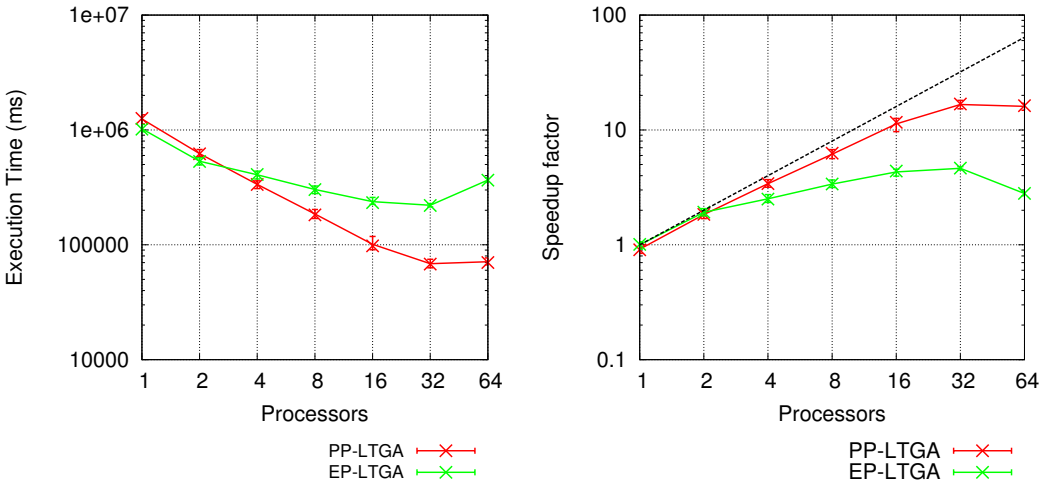


Figure 4.5: Comparison of PP-LTGA and EP-LTGA for *NK-Landscapes* with $\ell = 1600$.



Chapter 5

Searching for the Optimal Linkage Tree Replacement

In the previous chapters, the details of the Linkage Tree Genetic Algorithm (LTGA) have been discussed, as well as how the currently existing implementation has been improved. This resulted in the Perfect Parallel implementation of LTGA (PP-LTGA) that enables us to solve problems faster and, moreover, solve more complex problems that previously could not be solved within reasonable time. However, still one of the most important questions remains unanswered: why is LTGA able to outperform its competitors on most problems? As using a Linkage Tree (LT) to store important linkages is the most distinctive characteristic of LTGA, it is believed that its robustness and good performance is due to the use of this LT. However, it has never been proven that this model is optimal. Therefore, we investigated what improvements could be made to the linkage model used by LTGA, of which the results are presented in this chapter.

5.1 Background and Motivation

As stated before, LTGA outperforms its competitors, being other black-box meta-heuristics, on a variety of structured optimization problems by exploiting an LT in which important variable linkages are stored. Remarkably, however, LTGA is also able to outperform its competitors on problems for which a tree-structured model seems rather inappropriate, for instance on problems that are not hierarchically structured. This poses the question what actually is causing this robust and excellent performance of LTGA.

As discussed in Chapter 2, Bosman and Thierens have already done extensive research aimed at unveiling the true strength of the LTGA. They researched the effects of different metrics on the construction of the LT and also implemented tools that enable us to create insight into the quality of the LTs used, which showed that the Mutual Information (MI) is a suitable measure to be in the distance matrix for the construction of the LT [7, 58]. Further research was aimed at explicitly changing the structure of the model used by the LTGA. This was first done by replacing the online-learned LTs (LT_{on}s), being the LTs that are learned every generation based on the current population by LTGA, with predetermined

models that exactly resemble the problem's formulation structure [59]. As an example, this would be the univariate Family of Subsets (FOS), i.e. $\big\{\{x_0\}, \{x_1\}, ..., \{x_{\ell-1}\}\big\}$, for *Onemax* and a model containing all consecutive k-length variable clusters, i.e. $\big\{\{x_0,x_1,x_2,x_3,x_4\}, ..., \{x_{\ell-5}, x_{\ell-4}, x_{\ell-3}, x_{\ell-2}, x_{\ell-1}\}\big\}$, for *Deceptive Trap*. Although these models imposed a better performance of LTGA for *Onemax* and *Deceptive Trap*, for *NK-Landscapes* and *MAXCUT* such models imposed a worse performance and in fact scaled very poorly. Here, the performance of LTGA is measured in terms of Minimally Required Population Size (MRPS) and Minimally Required Number of Evalutations (MRNE), meaning that a better performance corresponds to a lower MRPS and MRNE. Finally, they attempted to remove superfluous linkage sets from the LT by means of *linkage hierarchy filtering*, as also discussed in Chapter 2 [8].

Although the aforementioned attempts have showed us that one of the strengths of the LTGA is the use of a hierarchically structured linkage model, none of them resulted in a clearly and consistently better performing algorithm, nor did they give a clear indication about whether this linkage model can be fundamentally improved. Since we are now able to solve more complex problems than before using the algorithm extensions implemented, perhaps it would be possible to bootstrap LTGA such that it could be used to optimize its own model. If this will indeed result in better linkage models, being models that, when replacing the LT_{on}s, cause LTGA to require less fitness function evaluations, the performance of LTGA could potentially be fundamentally improved, enabling us to solve even more complex problems within reasonable time.

5.2 Parameter-free Implementation

Apart form the extensions presented in the previous chapter, also a *parameter-free implementation* was implemented. As stated by Goldman and Punch, up until now, there has not been an implementation of LTGA that can overcome the problem of premature convergence without requiring expert knowledge about the algorithm and the problem at hand [24]. Premature convergence means that the algorithm has converged towards a non-optimal solution, which is usually due to using a value for n that is too small. As the population size used by LTGA decreases, the chance of premature convergence increases. A large value for n decreases the chance of permature convergence, however imposes the algorithm to require more execution time. Therefore, in order to increase the usability of LTGA such that problems can be solved about which only little is known, a novel implementation is needed that does not require any expert knowledge about the required population size, while it is still able to find the optimal solution within reasonable time, as also stated in Section 2.3.

Implementation

In order to achieve this, the *parameter-free implementation* is presented, inspired by the parameter-free scheme presented by Harik and Lobo [30]. The parameter-free implementation will run PP-LTGA with an ever increasing population size. It starts by executing PP-LTGA on the problem with $n_0 = 1$, after which PP-LTGA will be run for $n_i = 2n_{i-1}$ iteratively. After each iteration the best found solution of that execution is saved. This is

done until the algorithm is stopped. Note that if any knowledge is available considering the MRPS, a different value for n_0 can be used to trim redundant runs of the PP-LTGA. As will be shown later on in this chapter, this parameter-free implementation helped us in solving the problem of finding the optimal replacement for the online-learned LTs for the LTGA.

5.3 Learning LTs Offline

Initially, we aimed at finding an optimal linkage model to replace the LT by searching in the entire solution space of all possible linkage models and find an optimal replacement of the LT for every generation. Soon this turned out to be not feasible, however. First of all, we were not able to find an efficient way to traverse the total space of all possible linkage models. This space is spanned by the power set of all problem variables, meaning that it is spanned by $O(2^\ell)$ possible linkage sets, form which in total $O(2^{2^\ell})$ linkage models can be constructed. To find high-quality linkage models in this solution space efficiently, a clever mechanism is needed due to the exponential growth of this solution space as ℓ increases. Using LTGA itself in this situation showed to be hardly of any use. Using a straight-forward encoding for this solution space would provide LTGA with $O(2^\ell)$ variables, which would rapidly grow beyond LTGA's capabilities as the dimensionality of the considered problem increases. Several attempts were therefore done to construct a scalable collection of linkage sets that could be provided to LTGA to choose an optimal combination from. All these attempts resulted in collections of linkage sets that either did not contain the important linkage sets needed, meaning that LTGA was not able to construct a high-quality linkage model from this collection, or did not scale efficiently, meaning that as the problem size increased, the collection of interesting linkage sets expanded with such a rate that it soon grew beyond LTGA's capabilities.

Secondly, finding an optimal linkage model for every generation is a problem of vast complexity as what it is that makes a particular linkage model optimal to be used for generation g_i, is highly dependent on the effects of the linkage model used for generation g_{i-1}. Moreover, it is hard to estimate how these linkage models exactly are related and whether a linkage model that is optimal to be used just for a particular generation, will also support a fast convergence towards high-quality solutions in consecutive generations.

Therefore, we limited our search to finding a high-quality predetermined offline-learned LT (LT_{off}). An LT_{off} is an LT that is learned offline, i.e. on forehand, and is used as a predetermined linkage model to replace the traditionally used LT_{on}s for every generation in LTGA. Note that this means that every generation uses the same LT_{off} that was learned on forehand. Although also in this situation LTGA itself could not be used to find such LT_{off}s, we already had a mechanism to construct LTs efficiently, which can be manipulated to construct LT_{off}s as will be illustrated next.

5.3.1 Experimental Setup

Recall that the construction of the LT is based on the contents of the Nearest Neighbor Chain (NN-Chain), as discussed in Chapter 2. The contents of the NN-Chain are determined by the first element the NN-Chain is initialized with and the contents of the Mutual Information

Matrix (MIMatrix). Given the stochastic behavior that determines what element the NN-Chain is initialized with when it is empty, a certain MIMatrix can create a specific set of NN-Chains and with that a specific set of LTs. By manipulating the values in the MIMatrix, the set of LTs that can originate from this MIMatrix can be manipulated. Therefore, in a sense, the MIMatrix can be seen as an encoding of all possible LTs. We employed iAMaLGaM, a numerical optimization Estimation of Distribution Algorithm (EDA) [4], to optimize over the contents of the MIMatrix resulting in the creation of a high-quality LT_{off}. iAMaLGaM was chosen over other real-valued Evolutionary Algorithms (EAs) due to its robustness [3]. The fitness of an LT_{off} is evaluated by executing 1000 independent instances of the parameter-free implementation of LTGA with this LT_{off} replacing the traditional LT_{on}s. Each of these 1000 runs will terminate when it has found the optimal solution. The fitness of an LT_{off} is then defined as the negated average number of required function evaluations over these instances and is to be maximized. This means that an LT_{off} is considered to be better than some other LT_{off} if it supports a better performance of LTGA in terms of the number of fitness function evaluations required by the parameter-free implementation.

Note that the problem that iAMaLGaM is solving is vastly more complex than the benchmark problem considered, as for a benchmark problem with ℓ variables, $\ell(\ell-1)/2$ continuous variables in the MIMatrix have to be optimized where for each fitness function evaluation in iAMaLGaM, the benchmark problem has to be solved 1000 times. Therefore, this setup is not aimed at finding the optimal solution to this benchmark problem, but aimed at creating insight into what it is that makes an LT_{off} optimal. Because of this vast complexity, only high-quality LT_{off}s could be found for problems with $\ell \leq 25$.

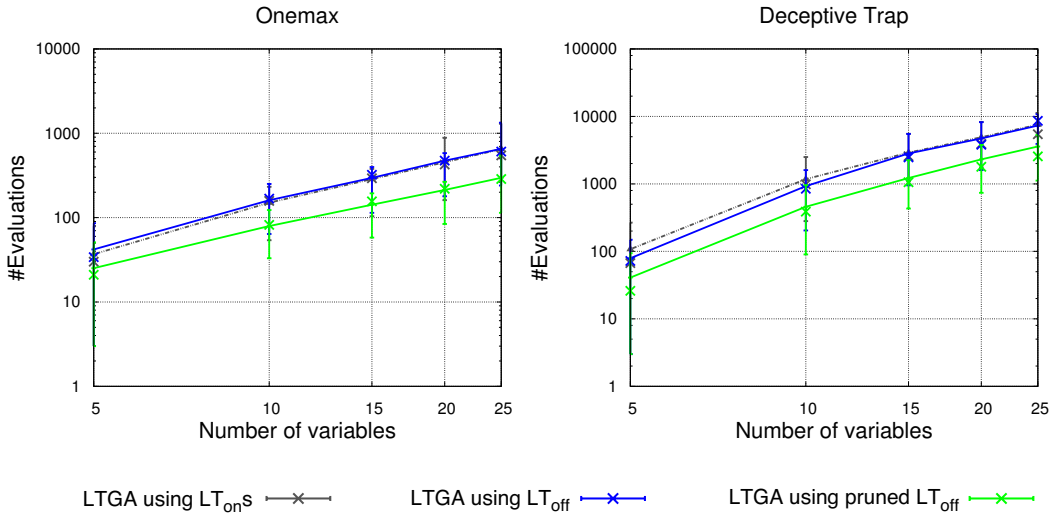
5.3.2 Experimental Results

Performance Analysis

Figures 5.1 and 5.2 show the results corresponding to the LT_{off}s found by iAMaLGaM. They show the performance of LTGA when using LT_{off}s as predetermined models and the performance of LTGA when using LT_{on}s. Note that the performance of LTGA that uses LT_{on}s is determined using the same procedure as for LT_{off}s, however now without replacing LT_{on}s with LT_{off}s. Additionally, these graphs show the performance of *pruned* LT_{off}*s*, however, these will be discussed further on.

The graphs clearly show that for *Onemax* no improvement was found, however this is rather trivial as it is known that the optimal predetermined linkage model for *Onemax* is the univariate FOS, which is always included by any LT. For *Deceptive Trap* only marginal improvement in LTGA's performance was found, however for more realistic problems, such as the *NK-Landscapes* problem and the *MAXCUT* problem indeed LT_{off}s were found that support a better performance of LTGA as the complexity of the problem increases. Moreover, these results show that the LT_{off}s found are more scalable than the LT_{on}s originally used by LTGA. Note, however, that finding such optimal LT_{off}s comes at high costs imposed by iAMaLGaM as stated before.

Figure 5.1: Required number of evaluations when using LT_{on}s and pruned LT_{off}s.



Contents Analysis

Apart from comparing the performance imposed by the LT_{on}s and LT_{off}s, also the contents of these LTs were analyzed. Results can be found in Appendix D, in which columns titled "in pruned LT_{off}" can be ignored for now. The results for *Onemax* as shown in Table D.1 are rather trivial, as stated before, however the results for *Deceptive Trap* as presented in Table D.2 are more insightful. For *Deceptive Trap* it is known that the exact variable linkage is given by the set containing all consecutive k-length blocks of variables. Results for *Deceptive Trap* are presented for $k = 5$ and $\ell = 15$ which means that the exact variable linkage is given by the FOS: $\big\{\{x_0,x_1,x_2,x_3,x_4\}, \{x_5,x_6,x_7,x_8,x_9\}, \{x_{10},x_{11},x_{12},x_{13},x_{14}\}\big\}$. Note that the optimal *linkage model* does not have to be equal to this FOS, as will be discussed later.

The results presented in Table D.2 correspond to the known variable linkage, as these linkage sets are included in the LT_{off} found. However, these results also show that LTGA aims to include these linkage sets in the LT_{on}s used. Consider the higher order linkage sets, being all but the singleton linkage sets in the LT_{off}. Results show that, while most of these higher order linkage sets do not occur in more than 25% of the LT_{on}s on average, the aforementioned important linkage sets show an average frequency of 70% to 75%.

However, if these linkage sets indeed represent the most significant variable linkages for *Deceptive Trap*, why do they not hold an average frequency of 100%? Looking at the results it is clear that this is caused by lower frequencies in the first and last generations. The same behavior is found for different problem sizes, which can be explained by a discrepancy between the population of these generations and the actual variable linkage as defined by the problem. In the first generations this is because the LTGA starts with a population of randomly generated solutions, which does not show any relation yet to the linkages as defined by the problem. As solutions improve, a clear correlation emerges between the content of these solutions and the problem's structure. This is reflected by the results found,

which show that the frequencies in the LT_{on} of the important linkage sets rapidly increase to a frequency of 100%. This shows how LTGA is able to rapidly uncover the problem's structure and therefore the overhead encountered in the first generations caused by imperfect LT_{on}s should be seen as the *costs* of learning the problem's structure online, rather than an *inefficiency*. Results of experiments aimed at decreasing these costs by performing *local search* will be discussed in the next section.

In the last generations, frequencies of these important linkage sets decrease again, which can be explained by the phenomenon that as soon as the optimal solution is present in the current population, LTGA will converge towards this solution. This means that the frequency of this optimal solution in P increases rapidly, which causes a skewed distribution of solutions and with that a skew distribution of identifiable linkages in the consecutive populations.

For *NK-Landscapes* and *MAXCUT* problem instances, as presented in Table D.3 and D.4, also particular linkage sets seem to be more dominantly represented than others, although they do not show a clear division as shown for *Deceptive Trap*. This could be due to the fact that, in contrast to *Deceptive Trap*, which has non-overlapping linkages of equal strength, these more complex problems contain overlapping linkages among which linkage strengths can vary, which can only limitedly be represented by an LT. As an example, it could be that for the *MAXCUT* instance shown in Table D.4, both $\{x_3, x_6\}$ and $\{x_3, x_9\}$ are important linkage sets, of which an LT can only include one. Assuming that stronger linkage exists between the variables in $\{x_3, x_9\}$ than those in $\{x_3, x_6\}$, one would expect that, while LTGA generally includes $\{x_3, x_9\}$ in its LT_{on}s, it will also occur that $\{x_3, x_6\}$ is included instead of $\{x_3, x_9\}$, due to LTGA's stochastic behavior. If this is indeed the case, it is expected that the frequencies of $\{x_3, x_6\}$ and $\{x_3, x_9\}$ correspond to the relative linkage strength between variables inside these linkage sets, meaning that neither of these linkage sets will have a frequency of 100% in the LT_{on}s, though both should have a relatively high frequency. This explains why a clear division as seen for *Deceptive Trap* is not encountered in the results for *NK-Landscapes* and *MAXCUT*.

In general, however, the results show that also for *NK-Landscapes* and *MAXCUT* the found LT_{off}s contain linkage sets that LTGA aims to include in its LT_{on}s, showing that with finding these high-quality LT_{off}s, indeed collections containing important linkage sets have been constructed.

5.4 Introducing Local Search

Based on the findings presented above, experiments were done aimed at decreasing the costs of online linkage learning by adding *local search* to LTGA. Local search is aimed at finding local optima using a low-cost and simple heuristic, which has often been shown to be a beneficial addition to evolutionary algorithms.

First, we attempted to reduce the costs of online linkage learning by replacing the first LT_{on}, which would be learned based on a population of randomly generated solutions, by the univariate FOS. It was expected that this would result in performance increase as, by performing Gene-pool Optimal Mixing (GOM) with this univariate FOS, a limited local

search would be done that would save evaluations spent on performing GOM with higher order linkage sets that were not aligned with the problem's structure. Results, however, showed the contrary, as this resulted in a worse performing algorithm, requiring a higher population size and more fitness function evaluations. Upon analyzing the behavior of LTGA using this implementation of local search, we discovered that this was in fact caused by the fact that using a full LT_{on}, albeit based on a population of random solutions, a stronger local search on the first generation was performed. The additional higher order linkage sets showed to have two functions in this situation. In the situation that such a linkage set could be used by GOM to improve a solution, this linkage set in a sense contributed to local search in a way comparable to singleton linkage sets. However, if this was not the case, this linkage set increased the selection pressure due to the Forced Improvement (FI) mechanism, which turned out to contribute to local search as well.

Based on these results, a stronger implementation of local search was used that was similar to the local search as presented by Goldman and Punch [24]. This implementation would, for each solution in the first generation, iterate over all variables in that solution in a random order and invert the value of a variable if and only if this imposed an improvement to the fitness of the solution. Note that, in contrast to the implementation of Goldman and Punch, our implementation will iterate over all variables in a solution only once per solution. Results for these experiments are shown in Figures 5.3 and 5.4, which show the performance of LTGA and LTGA when using this implementation of local search. Results shown in these graphs for LTGA when using only 50% of the LT_{off}s can be ignored for now as this will be discussed in the next section.

These results show that for *Onemax* this obviously imposed a severe performance improvement, as this problem is structured such that after performing this implementation of local search, all solutions in the first population are changed to the optimal solution and no further generations are needed. For *Deceptive Trap* and *NK-Landscapes* only marginal improvement could be achieved. For *MAXCUT*, it was shown that this implementation of local search could significantly decrease the initial costs of online linkage learning. However, when comparing this to the total costs of finding the optimal solution, the reduction in required number of evaluations turned out to be limited.

5.5 Pruning Offline-learned LTs

With the LT_{off}s presented in Section 5.3, collections of variable sets were found that contain important linkage sets. These same results, however, also contain linkage sets that are very poorly represented in the LT_{on}s used by LTGA. What is the role of these linkage sets and to what extend do they support an efficient search for the optimal solution? What if we take away the constraint of using a tree-structured linkage model? What if we use LTGA itself to only select the truly necessary linkage sets from the LT_{off}s that were found using iAMaLGaM? The fact that some linkage sets of the found LT_{off}s are so poorly represented in LT_{on}s, while LTGA is still able to find the solution quite efficiently, could indicate that these linkage sets are not so important and that by removing them, less fitness function evaluations would be required during GOM. On the other hand, these poorly represented linkage sets

could be the reason why the LT_{off}s found support a better performance of LTGA. Therefore, additional experiments were done aimed at pruning the found LT_{off}s in order to improve the performance of LTGA, but also to create better insight into what linkage sets are actually essential to a high quality linkage model.

5.5.1 Experimental Setup

In order to filter the essential linkage sets from the LT_{off}s that were found using iAMaLGAM, LTGA itself was used, in a sense optimizing its own linkage model. For clarification reasons, we will call this LTGA instance that is used to prune LT_{off}s the "meta-LTGA". Given an LT_{off} that was found using iAMaLGaM, each linkage set in this tree is assigned a binary variable, which encoding is then used by this meta-LTGA to find the optimal subset of this LT_{off}. This means that, when searching for a high-quality *pruned* LT_{off} for a benchmark problem with ℓ variables, the encoding used by the meta-LTGA to find such a high-quality pruned LT_{off} consist of $2\ell-2$ variables in which each variable indicates the presence of a linkage set of the LT_{off} in the pruned LT_{off}. For instance, if the LT_{off} would be $\big\{\{x_0\}, \{x_1\}, \{x_2\}, \{x_1,x_2\}, \{x_0,x_1,x_2\}\big\}$, the solution 10010 would represent the pruned LT_{off} $\big\{\{x_0\}, \{x_1,x_2\}\big\}$. The fitness of such a pruned LT_{off} will be evaluated in the same way as LT_{off}s were evaluated as described in Section 5.3.1.

By using this encoding, the meta-LTGA can be used to find hiqh-quality subsets of linkage sets of an LT_{off}, in a sense searching for the optimal linkage model within the search space spanned by this LT_{off}. Note that this approach is expected to find pruned LT_{off}s that do not impose a worse performance of LTGA than the considered LT_{off} does, as in the worst case scenario, the optimal pruned LT_{off} has the same performance as the full LT_{off} for it is always possible to select all linkage sets from this LT_{off}. To find high-quality pruned LT_{off}s, the parameter-free scheme was used for iteratively starting meta-LTGA instances with ever increasing population sizes as we have no knowledge about the ideal population size to be used to solve this problem. Execution was stopped as soon as two consecutive runs of the meta-LTGA initiated by the parameter-free scheme resulted in finding the same linkage model, as for less complex problems tested this had shown to be an indication for finding the global optimum.

5.5.2 Experimental Results

Performance Analysis

As stated before, results found for these experiments are included in Figures 5.1 and 5.2, while in Appendix D the contents of some pruned LT_{off}s are presented by the column "in pruned LT_{off}". In Figure 5.1 it is shown that for *Onemax*, linkage models are found that support LTGA in significantly outperforming its original implementation by a constant factor in terms of the required number of evaluations. To illustrate, in terms of required execution time, 1.782 milliseconds was required on average when using the a parameter-free implementation which uses the sequential implementation of LTGA with LT_{on}s for solving *Onemax* with $\ell = 25$, while only 0.626 milliseconds was required when using the pruned LT_{off}. This is as expected, as the univariate FOS is considered as the optimal representation

of the linkages in this problem, which LTGA was indeed able to find, as shown in Table D.1. Here, we see that all singleton linkage sets but one are selected from the LT_{off} for the best found linkage model, which corresponds to our expectations as this is equivalent to using the univariate FOS, on a conceptual level. Assume that a solution s exists in population P in which the value for some variable $s_i = 0$. Then assume a donor d in P exists for which $d_i = 1$. This means that s can be improved using d and the univariate FOS as a linkage model. However, on a conceptual level, this is equivalent to improving d using s as a *donor* with a linkage model consisting of all singleton linkage sets but $\{x_i\}$. The only practical difference is that, due to our implementation, this alternative linkage model will require one fitness function evaluation less on average, meaning that the linkage model that contains all but one linkage set will have a slightly better fitness. Therefore, LTGA omits one of the essential linkage sets randomly.

For *Deceptive Trap*, similar results are shown in which the pruned LT_{off} imposes a performance that is better than the original performance of LTGA by a constant factor. To illustrate, the use of the pruned LT_{off} reduced the required execution time from 7.593 milliseconds to 1.774 milliseconds for $\ell = 25$ on average. Also in this situation, this seems to be caused by the use of a pruned LT_{off} that consists of all but one of the essential linkage sets. Recall that for $\ell = 15$ for *Deceptive Trap* the best representation of the most important linkages are given by the FOS $\big\{\{x_0,x_1,x_2,x_3,x_4\},\ \{x_5,x_6,x_7,x_8,x_9\},\ \{x_{10},x_{11},x_{12},x_{13},x_{14}\}\big\}$, then indeed Table D.2 shows that the best found subset of the LT_{off} consists of two of these three linkage sets, being $\{x_0,x_1,x_2,x_3,x_4\}$ and $\{x_{10},x_{11},x_{12},x_{13},x_{14}\}$.

This already shows us that, using this approach, high-quality linkage models can be found at high costs that support a significantly better performance of LTGA when such a subset is to replace the LT_{on}s. However, aforementioned benchmark problems are rather simplistic, which is why this approach has also been applied to more realistic problems such as the *NK-Landscapes* problem and the *MAXCUT* problem, for which the exact variable linkages are unknown. The graphs in Figure 5.2 show that also for these problems, pruned LT_{off}s were found that impose a better performance than the full LT_{off}s. To illustrate, the use of the pruned LT_{off} reduced the required execution time from 6.464 milliseconds to 2.588 milliseconds for *NK-Landscapes* with $\ell = 25$. Note that therefore also these pruned LT_{off}s impose LTGA to be more scalable compared to when LT_{on}s are used.

Contents Analysis

Results presented above are as expected, however analyzing the exact contents of these pruned LT_{off}s turned out to be less insightful. Consider the results presented in Table D.3 for *NK-Landscapes* with $\ell = 20$. These results show that roughly half of the linkage sets of the found LT_{off} were omitted in order to find an optimal subset of this LT_{off}. This finding is particularly interesting because the same behavior has been found across multiple problem instances for *NK-Landscapes* and *MAXCUT*. Especially for more complex instances, roughly between 49% and 53% of the linkage sets were consistently filtered out.

Despite this clear trend in our results, no clear correlation could be found between the linkage sets that were selected. Roughly half of the singleton linkage sets are selected of the pruned LT_{off} but also among higher order sets, roughly half of the linkage sets are

selected. On first sight, no linkage sets of a particular size or content, nor linkage sets that were dominantly represented in the LT_{on}s seem to be preferred, which is contrary to our expectations. Additional experiments were performed to verify that the omitted linkage sets were not randomly selected, by using only a random 50% of the linkage sets from the LT_{on} during GOM. As shown in Figure 5.3 and 5.4 this indeed did not result in the same performance improvement, indicating that indeed some correlation should exist between the selected linkage sets.

Results for *MAXCUT*, as presented in Table D.4, were studied in detail in which also a seemingly random half of the linkage sets was selected. When comparing the pruned LT_{off}s with the problem instance data, it seemed that linkage sets were preferred that could contribute to strong *subcuts*. We define a *subcut* as a division between a subset of the vertices of the graph considered in a *MAXCUT* problem instance. For a different *MAXCUT* instance, which is presented in Figure 5.5, this could for instance be $A = \{v_1, v_4\}, B = \{v_2, v_7\}$, where A and B define the cut between the vertices $\{v_1, v_2, v_4, v_7\}$. A strong subcut is a subcut that, when combined with other strong subcuts, defines the maximum cut for a graph. In the example presented in Figure 5.5, this could for instance be $A = \{v_2, v_6\}, B = \{v_7\}$, because in this subcut, two edges with high weights are cut. In this situation, the linkage set $\{x_2, x_6, x_7\}$ would have been selected for a pruned LT_{off}.

Although the linkage sets in the pruned LT_{off}s often seem to correspond to such strong subcuts, we were not able to exploit this. Experiments were performed in which a linkage model would be built that contained linkage sets that could support the construction of strong subcuts, based on the problem instance data provided. However, results showed that this approach did not cause a significant performance increase for the LTGA. Also for *NK-Landscapes*, experiments were performed in which a linkage model would be constructed based on the variance of variables in the provided lookup tables, though also these experiments turned out to be unfruitful.

5.6 Conclusions

Results presented in this chapter have shown us that the LT_{on}s used by LTGA are not optimal and predetermined LTs exist that enable the LTGA to find optimal solutions while requiring less evaluations. Finding these LT_{off}s using iAMaLGaM comes at a high cost, which is why the presented approach is not suitable for substituting the currently used LT construction implementation. These results did, however, provide us with high-quality LT_{off}s that are expected to contain linkage sets which are important for solving the problem at hand. Additional experiments have shown that pruning these LT_{off}s results in linkage models that support an even better performance of LTGA, albeit at an even higher cost. Finding high-quality subsets of the LT_{off}s found by iAMaLGaM was expected to provide us with information that could provide insight into the true reason for LTGA's excellent performance. Although in these results consistently around 50% of the linkage sets in the LT_{off}s were omitted, no clear correlation could be found between the linkage sets that remained. Therefore we were not able to improve LTGA's linkage learning mechanism based on this information.

Note, however, that such a correlation should exist, as randomly using only 50% of the LT_{on} during GOM did not result in the same performance improvement. Understanding why these linkage sets constituted a better linkage model is of great importance as this can support us in reflecting upon the currently used linkage learning mechanism but perhaps also on other components of LTGA. Ultimately, this could support us in improving LTGA by for instance using better linkage models or more efficient variation operations, enabling LTGA to solve problems while requiring significantly less fitness function evaluations. It should be noted, however, that an important requirement for such improvements is that they can be applied without increasing the computational complexity of LTGA as only then the required execution time of LTGA can be decreased for more complex problems.

Figure 5.2: Required number of evaluations when using LT_{off}s and pruned LT_{off}s.

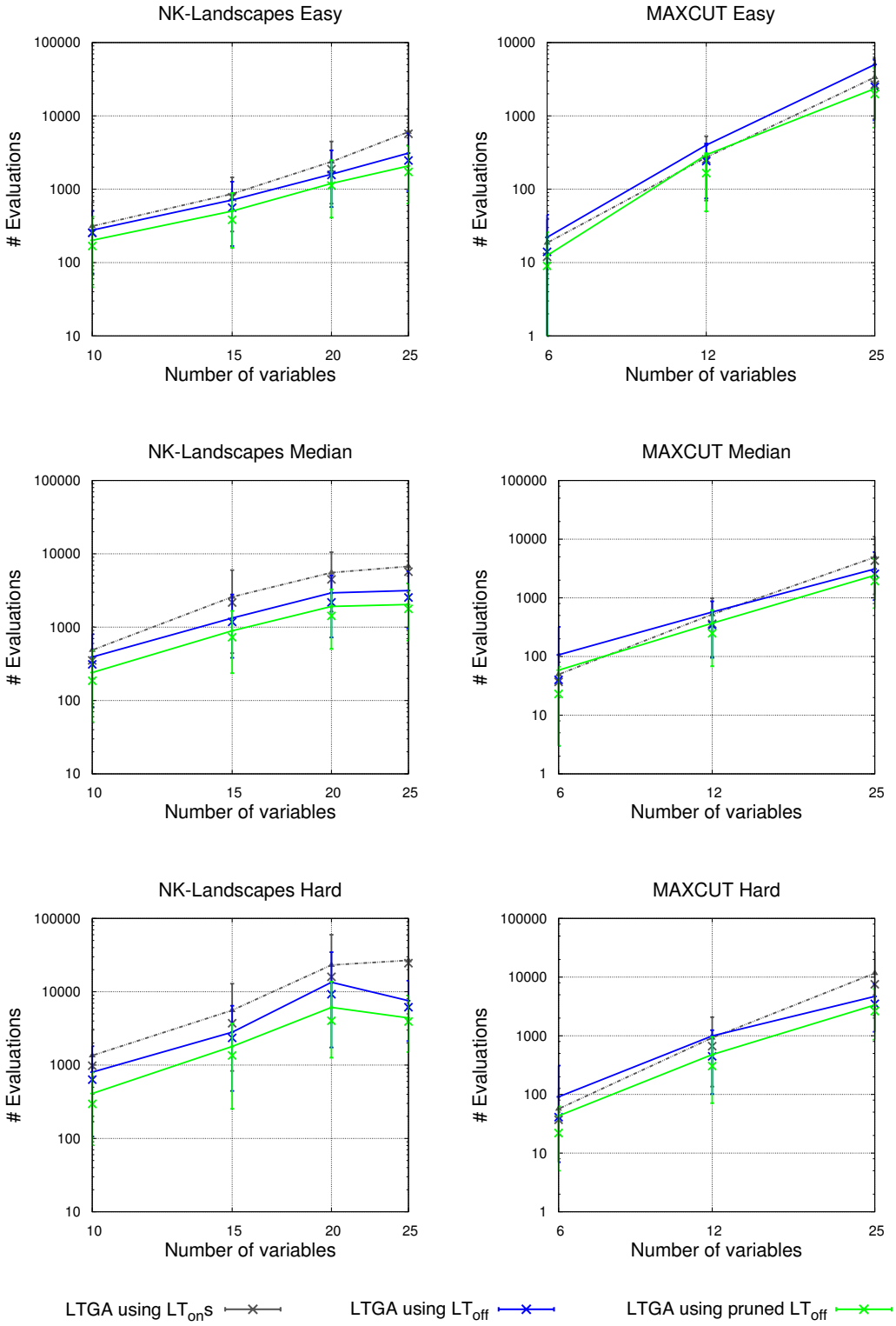


Figure 5.3: Required number of evaluations when using LTGA with local search or a randomly halved LT.

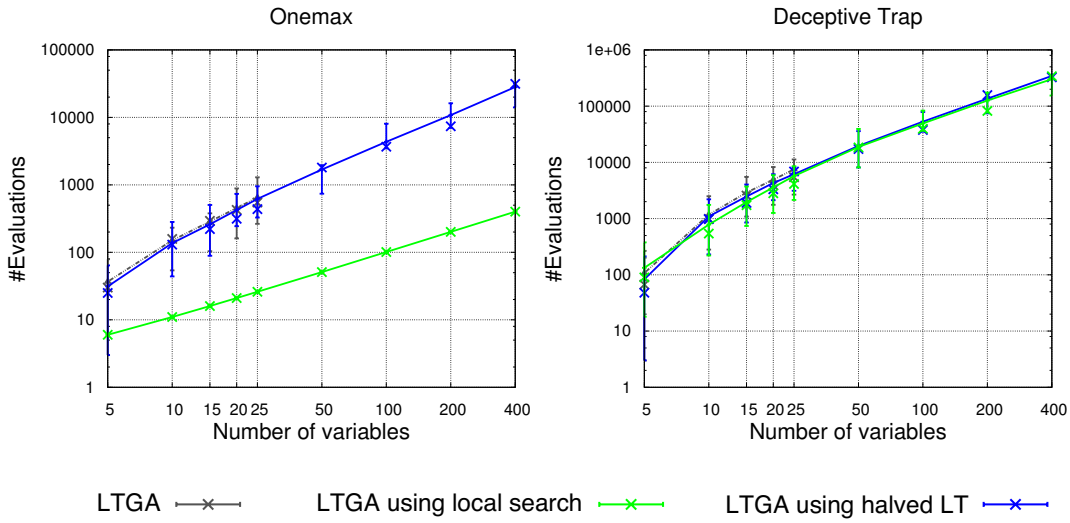


Figure 5.4: Required number of evaluations when using local search or a randomly halved LT.

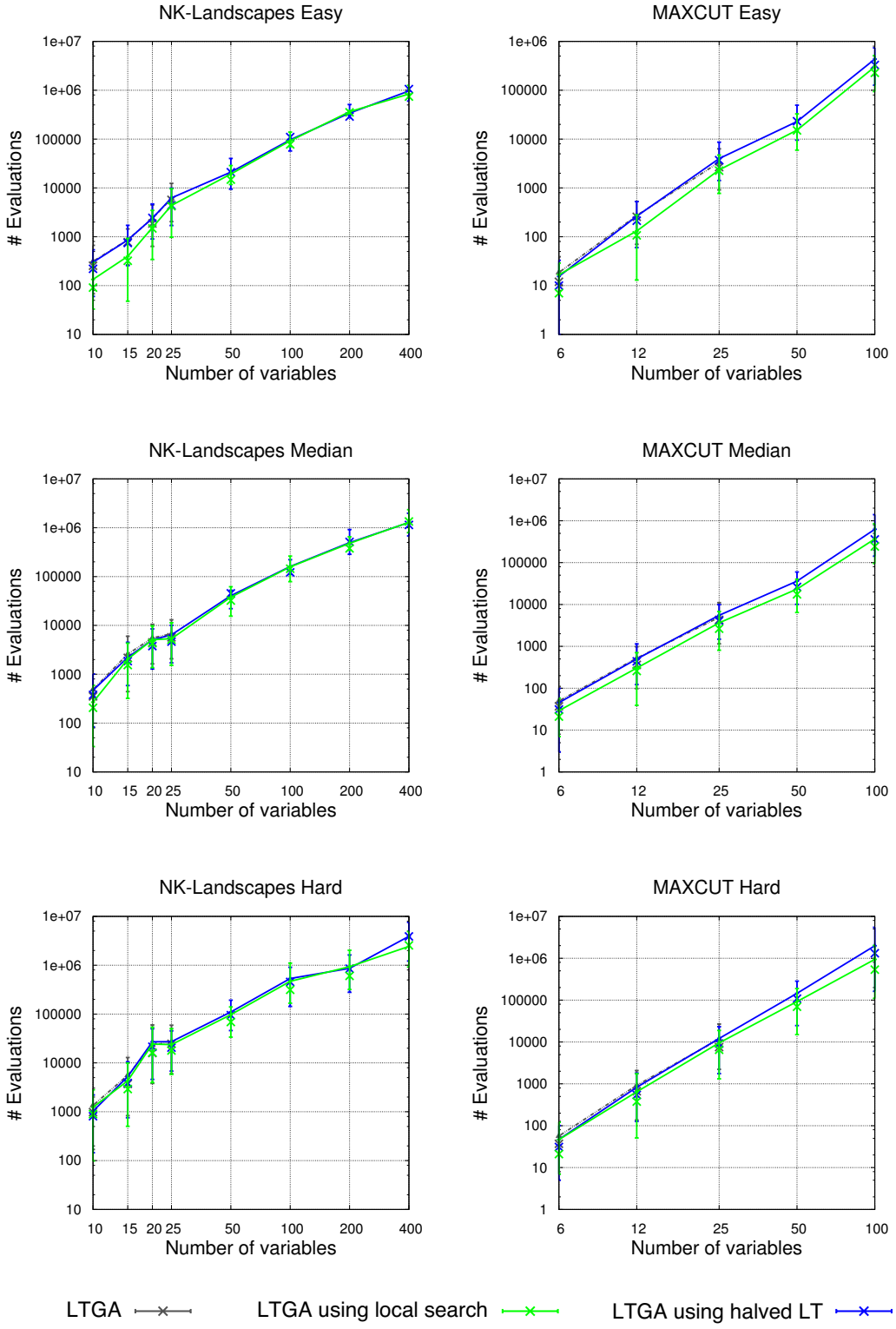
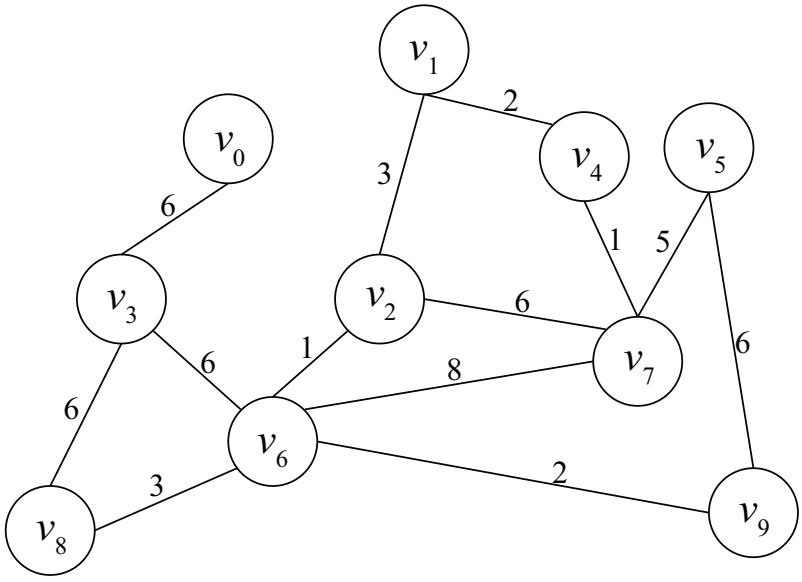


Figure 5.5: An example of a *MAXCUT* problem instance.



Chapter 6

Discussion

In this thesis, results have been presented of research done on extending the implementation of the Linkage Tree Genetic Algorithm (LTGA). With these extensions, new doors have opened, enabling us to use LTGA to find high-quality predetermined linkage models that impose a better performance of LTGA when replacing the traditionally used online-learned LTs (LT_{on}s). However, upon analyzing the results of experiments performed, it became clear that still improvements exist for LTGA. It became clear that with these new possibilities new aspects came into view that are still to be investigated. In this chapter, the most important aspects will briefly be discussed.

6.1 Algorithm Complexity and Bottlenecks

In the analysis of the LTGA in Chapter 3 it was shown that Gene-pool Optimal Mixing (GOM) dictates the complexity of the algorithm and, together with the filling of the Mutual Information Matrix (MIMatrix), determines the required execution time. Moreover, it was shown that because of this, improving the implementation of the fitness function for *NK-Landscapes* significantly reduced the time required by GOM. Inefficiencies in the evaluation of fitness functions may still exist in the currently implemented fitness function and might exist in future fitness functions. that can give room for further improvement to the performance of LTGA. A small reduction in the fitness function evaluation time can have great impact, due to the vast amount of fitness function evaluations done for complex problems.

However, on a slightly higher level, a far more promising extension can be added to GOM, being the integration of caching of fitness function evaluations to avoid redundant function evaluations. By maintaining a map that saves the fitness of every solution evaluated so far, no fitness function evaluation is required if a solution gets created during GOM that has been evaluated before. Initially this extension was not added to the algorithm as experiments were mainly performed on toy-size problems. Using caching for toy-size problems will not benefit the algorithm, as the overhead introduced by caching might be too big compared to the actual fitness function evaluation time. However, for more complex

problems such as the problem of pruning offline-learned LTs (LT_{off}s), this reduction in function evaluations is expected to cause a severe reduction in required execution time.

6.2 Parallelization Improvements

In Chapter 4, several aspects were presented that prevent the Perfect Parallel implementation of LTGA (PP-LTGA) from reaching the optimal speedup. As mentioned before, we feel that, on average, this implementation is able to efficiently leverage the computational power available, however, still some fine-tuning can be done that could increase the performance of PP-LTGA. The first subject worth further investigation could be ways to decrease the processor idling time by, for instance, trying to find more appropriate ways to estimate the workload of parallel processable tasks in order to dynamically fine-tune the used granularity or scheduling of tasks. Second, it might be worthwhile to investigate whether performance improvement can be achieved by using memory that is more suitable for parallel data access. Last, it might be interesting to investigate the optimal configuration for PP-LTGA. Experiments showed that the overhead introduced by the Java Thread Scheduler increases as the number of processors increases. As the complexity of the problem increases, the effect of this overhead decreases compared to the total execution time required. However, even for *NK-Landscapes* with $n = 1600$, using 32 processors imposed a better performance than using 64 processors. We think this is due to the fact that in reality not all 64 processors were freely available. For instance, the Java Garbage Collector is scheduled on a separate thread, meaning that when the Garbage Collector is initiated, some processors need to serve more than 1 thread, which is not optimal. We therefore think that in general for large-scale problems, an even better performance could be achieved by using $32 < p < 64$ when using a machine with 64 cores.

Also for the Embarrassingly Parallel implementation of LTGA (EP-LTGA), some aspects are still to be investigated. As stated in Chapter 4, EP-LTGA can be classified as a multiple-population coarse-grained Genetic Algorithm (GA) as described by Cantú-Paz. However, in contrast to the description provided by Cantú-Paz, in EP-LTGA no solutions are exchanged between the populations of separate instances, which is called migration. We implemented EP-LTGA this way because we expected that migration required thread synchronization, which was expected to create a bottleneck in the algorithm. It might be interesting to investigate, however, whether migration could be implemented without blocking threads and what effect this has on the performance of EP-LTGA. Recall that EP-LTGA is only as fast as the longest running instances. One would expect that this instance requires more time because it has trouble finding high-quality solutions. Exchanging high-quality solutions between populations might significantly speed up the process of finding the optimal solution for such an instance, decreasing the processor idling time and significantly reducing the required execution time for EP-LTGA.

6.3 Hardware

6.3.1 GPUs

When executing a large number of numerical operations, often GPUs are used as they are optimized to execute such operations at a higher speed than CPUs and usually contain more processor cores in order to do this in parallel. When looking at LTGA, we felt that the only component suitable for being executed on a GPU, would be the filling of the MIMatrix as this is the only component entailing vast amounts of numerical operations. As the execution time was mainly dominated by the execution of GOM, we did not further investigate this. However, when integrated appropriately, LTGA might still benefit from using GPUs for calculating entries in the MIMatrix.

6.3.2 Using large numbers of processors

As mentioned before, a server with 64 virtual processors was used for all the experiments presented, of which the specifications are presented in Table 3.1. When the experiments were performed, this was considered a very powerful machine, however parallel hardware develops fast. Already architectures are available to consumers and small businesses that contain a multitude of processors of the server used for our research and it is expected that the computational power available for customers and small businesses will continue to increase in the form of increasing numbers of processors per machine. We have seen that an increasing amount of processors implies an increasing overhead imposed by thread management. At some point this might be a severe impediment for an algorithm like PP-LTGA, which means a different approach will be needed. Recall that EP-LTGA only encountered minimal overhead imposed by the Java Thread Scheduler due to the limited amount of threads. It might be interesting to investigate the potential of a hybrid implementation, being an extension to EP-LTGA that uses PP-LTGA instances that each get assigned a cluster with a limited number of processors. Note that in order to properly investigate the potential of such a hybrid implementation, indeed an architecture is needed with a multitude of the processors used for our experiments.

6.4 Fuzzy Linkage Modeling

Last but not least, still problems are to be addressed on a more conceptual level that could have a fundamental impact on the performance of LTGA. Our research has been aimed at finding reasons why some linkage sets should be chosen over others by linkage learning mechanisms. By learning Linkage Trees (LTs) offline and pruning them, we were able to find linkage models that outperformed linkage models produced by renown learning linkage mechanisms. However, why were we able to find these high-quality linkage models, while previous attempts did not result in any models that supported a significant and consistent better performance of LTGA? What is the reason why the pruned LT_{off}s that were found supported a better performance of LTGA than the linkage models used in previous publications?

One of the main inspirations for learning LTs offline has been the research performed by Thierens and Bosman on predetermined linkage models [59]. Despite using an approach which seemed to be bound to result in a linkage model that aligns better with the reality, their linkage models did not seem to contain the right linkage sets. This can also be said for the results published by Pelikan et al. [47], which concluded that by using problem-specific distance metrics, linkages were identified in *too much detail*, resulting in linkage models that contained linkage sets in which relatively weak linkage existed between the variables contained. Though, what if Thierens and Bosman in fact were facing the same problem? For *Onemax* and *Deceptive Trap* they were able to construct linkage models that supported a better performance of LTGA in two of their publications [8, 59], however, these are problems for which an easy distinction can be made between relevant and irrelevant linkage sets. In contrast, the models they constructed for *NK-Landscapes* and *MAXCUT* did not cause a better performance; problems for which the relevance of linkage sets can be hard to determine.

Assuming that the models presented by aforementioned authors indeed lacked an appropriate level of linkage detail, this should mean that with the approach presented in this thesis, we were able to overcome this problem. Although it can be interesting to perform more experiments as presented in the previous chapter in order to find heuristics that approach this correct level of detail, there is a more fundamental problem if indeed *linkage strength* is the main property that is at the root of the performance imposed by linkage models. Currently we are using linkage learning techniques in which variable linkages are either acknowledged or not. This results in linkage models containing variable sets of which we find the linkage is relevant. However, this does not mean that variable sets that are not included in such a linkage model do not contain significant linkage, especially in a model like the LT that has only a limited ability to model overlapping linkages. We feel that this way of linkage learning does not comply with the reality, as we think that in reality linkage is not a property that is either existent between variables or not. Linkage might exist between all variables of all possible variable sets, though some linkages are stronger than others.

Therefore, we feel that a fundamentally different approach, aimed modeling this *fuzzy linkage*, is needed in order to construct a model that finds better alignment with reality. This can be done by, for instance, constructing a *linkage distribution* in which the linkage strengths of all variable sets are contained. Being able to construct such a linkage distribution can provide numerous new ways for performing variation in LTGA and perhaps GAs in general. A straight-forward way for using this linkage distribution could be by selecting a fixed amount of linkage sets from this distribution according to their linkage strength. This means that linkage sets for which strong linkage exists between the contained variables have a higher chance of being used during for instance a GOM operation compared to weaker linkage sets. This in turn means that strong linkage sets will be used more often than weak linkage sets. If a linkage distribution that aligns better with reality can be exploited appropriately, fundamental changes in the performance of LTGA can be expected, possibly fundamentally expanding the collection of problems that can be solved by LTGA.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis has been aimed at increasing the problem solving capabilities of the Linkage Tree Genetic Algorithm (LTGA). This started with an introduction into the background of LTGA, after which the details of LTGA were discussed. The most significant impediment on solving large-scale problems with LTGA was its lacking ability to use computational power present in multi-processor architectures. As we are approaching the economical limit of the maximum clock speed of a single-core CPU, current technological developments are aimed at increasing the number of CPU cores on a single chip. With that, machines with tens or even hundreds of CPU cores also become available to small businesses and even consumers. This called for a parallel implementation of LTGA to leverage the computation power in multi-processor architectures as illustrated by the goals and research questions presented in Section 2.3. We presented both an *internally parallelized* and *externally parallelized* implementation of LTGA, which were called the Perfect Parallel implementation of LTGA (PP-LTGA) and Embarrassingly Parallel implementation of LTGA (EP-LTGA) respectively. These implementation were then used to harness the computational power available to perform experiments for gathering insight into the model used by LTGA to investigate whether the potential of LTGA could be further increased.

7.1.1 Parallelizing LTGA

The first approach for developing a parallel implementation of LTGA that was investigated was *internal parallelization*, for which computationally intensive components of LTGA were to be parallelized, which resulted in the PP-LTGA. An analysis was performed aimed at identifying the most computationally intensive components of LTGA, as questioned by our first research question in Section 2.3. This analysis has shown that the execution time of LTGA is dictated by the filling of the Mutual Information Matrix (MIMatrix) and the generation of new solutions using Gene-pool Optimal Mixing (GOM). This means that LTGA is very suitable for internal parallelization as the workload of these most expensive components can be distributed over multiple processors using a straight-forward approach. This is due to the fact that cells in the MIMatrix can be filled independently of each other

and also solutions can be generated independently of each other. However, although the structure of LTGA is very suitable for applying internal parallelization, several overhead aspects impede the performance increase that can be achieved, which are mainly due to the overhead introduced by the Java Thread Scheduler. Although in our experiments a maximum speedup of 10x on 32 cores was encountered, it is expected that a larger speedup can be achieved for larger and more complex problems.

The second approach for developing a parallel implementation of LTGA that was investigated according to our second research question posed in Section 2.3 was *external parallelization* in which multiple instances of LTGA are executed simultaneously, spread over a number of processors, after which only the best solution of these runs is considered. This allows for a smaller population size per instances, causing a reduction in the execution time required by this EP-LTGA. Nevertheless, this reduction stagnates as the number of instances increases, which we think is due to a limit below which LTGA no longer has any significant problem solving capabilities. Therefore, as the number of processors increases, at some point the decrease in population size can no longer compensate for the increasing multi-threading overhead. Although this pivot point moves towards a higher number of processors as the size and complexity of the problem increases, this still impedes EP-LTGA from efficiently leveraging the computational power available.

Therefore, PP-LTGA is considered as a more scalable implementation of LTGA, of which the performance can still be improved by applying techniques like caching of function evaluations and the use of novel heuristics to better estimate the workload that is to be distributed for better job scheduling and less processor idling time. With this, we achieved our first goal as presented in Section 2.3.

Learning Linkage Trees Offline

In order for LTGA to be used to optimize its own linkage model, a limited collection of linkage sets is to be supplied to chose from, or a novel encoding for the total solution space of all possible linkage model has to be found. By manipulating the contents of the MIMatrix, the linkage learning mechanism of LTGA can be used to construct specific collections of Linkage Trees (LTs). Therefore, by optimizing over the contents of the MIMatrix, high-quality offline-learned LTs (LT_{off}s) can be found using iAMaLGaM that can support a better performance of LTGA when used as a predetermined linkage model, albeit at high costs. By doing so, LT_{off}s could be found for problems with $\ell < 25$. These high-quality LT_{off}s support a better and more scalable performance of LTGA compared to the traditionally used online-learned LTs (LT_{on}s), which means that the LT_{on}s used by LTGA are not optimal and LTGA's performance can perhaps still be improved.

These LT_{off}s also showed that LTGA is capable of quickly uncovering the problem's structure. The overhead encountered in the first generations where LT_{on}s do not align with the problem yet, should therefore not be seen as an *inefficiency*, but merely as the *costs* of learning the problem's structure online. With the use of *local search* these costs can be reduced, however the impact of this reduction is limited compared to the total costs of executing LTGA.

Pruning the Offline-learned LTs

Experiments presented in this work show that particular linkage sets contained by the LT_{off}s found by iAMaLGaM are dominantly represented in the LT_{on}s learned by LTGA, while other linkage sets are very poorly represented. Using a parameter-free implementation, the solution space spanned by an LT_{off} can be traversed efficiently to prune such an LT_{off} by removing these seemingly redundant linkage sets. This parameter-free implementation executes instances of LTGA, which we called the "meta-LTGA" with ever increasing population sizes, enabling us to solve problems efficiently for which no knowledge is available about the optimal population size to be used while overcoming the problem of *premature convergence*. With this parameter-free scheme, pruned LT_{off}s can be found that support an even better performance of LTGA, albeit at even higher costs. This answers the third research question and enabled us to achieve our second goal as presented in Section 2.3

To achieve this even better performance of LTGA, roughly 50% of the linkage sets were filtered out from LT_{off}s found by iAMaLGaM for *NK-Landscapes* and *MAXCUT* problem instances. This strongly suggests that the Linkage Tree model traditionally used by LTGA contains redundancies that, if filtered out correctly, could fundamentally improve the performance of LTGA.

Randomly removing 50% of the linkage sets in LT_{on}s does not result in a comparable performance improvement, indicating that some correlation exists between the linkage sets selected by the meta-LTGA. This means that the linkage learning mechanism that is currently used returns a linkage model that does not align with the actual linkage structure of the problem. Unfortunately, we were not able to identify this correlation, which prevented us from leveraging this new information. Constructing a fixed linkage model a priori that would contain linkage sets that support the construction of strong *subcuts*, which the pruned LT_{off}s suggested to be beneficial when solving a *MAXCUT* problem instance, did not result in a performance improvement. Also experiments performed for *NK-Landscapes* in which predetermined linkage models were constructed based on the variance of variables in the provided problem data were unfruitful.

Contrary to past research, however, we are now able to show that it is possible to find consistently better linkage models for LTGA, albeit at high costs. In the end, no definite answer could be given to the question of what is exactly causing LTGA's excellent performance and robustness, however with the linkage models that were found, initial insight has been gathered into the true potential of LTGA, with which we answered the remaining research questions and achieved the goals presented in Section 2.3. These insights enabled us to propose a variety of possible improvements for LTGA, of which perhaps the most fundamentally changing concept would be the introduction of *fuzzy linkage* into the linkage learning mechanisms that are currently used.

Finally, these insights show that creating a better understanding of high-quality linkage models is of great importance as it can give direction to enhancing the linkage learning mechanism of LTGA and to fine-tuning the algorithm so its potential can be fully harnessed, fundamentally improving the performance and with that the problem solving capabilities of LTGA.

7.2 Future work

To further improve our understanding of high-quality linkage models for LTGA, some extensions to the algorithm can be implemented and more experiments can be performed. We feel that future work should first of all be aimed at integrating the caching of fitness function evaluations into GOM in order to avoid fitness function evaluations of solutions that have already been evaluated before. We feel that this should be the first extension to be made to LTGA as straight-forward and well documented approaches for integrating caching are available and with this extension, the applicability of LTGA is extended to more complex and larger scale problems, as for such problems in particular the integration of caching is expected to have a significant impact on the required execution time.

Secondly, we feel that the most interesting aspect to focus future work on would be finding ways to model *fuzzy linkage* as discussed in the previous chapter. In order to construct a *linkage distribution*, the linkage strength between variables of all possible variable sets has to be quantified. Although Bosman and Thierens showed that Mutual Information (MI) is a suitable metric for the linkage learning mechanism used in LTGA [7], it is not known whether this metric is also suitable for constructing a linkage distribution, which is why this should be verified. Consequently, after being able to construct such a linkage distribution, possible ways to perform variation can be evaluated to investigate what variation operations could be used that fully exploit the information contained by a linkage distribution. A straight-forward implementation could be using GOM with this linkage distribution, during which for each solution a fixed amount of linkage sets are drawn from the distribution to be used as cross-over masks. Strong linkage sets then have a higher chance of being drawn from this distribution than weaker linkage sets, which means that stronger linkage sets will in general be used more often than linkage sets containing variables with weaker linkage. We believe that constructing a linkage model that is better aligned with reality and exploiting such a model accordingly can fundamentally improve the performance and the applicability of LTGA, fundamentally changing what problems it can be applied to.

Bibliography

[1] Lee Altenberg. Fitness distance correlation analysis: An instructive counterexample. In *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 57–64. Morgan Kaufmann, 1997.

[2] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–23, March 1993.

[3] Peter A. N. Bosman. On empirical memory design, faster selection of bayesian factorizations and parameter-free gaussian EDAs. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 389–396, New York, NY, USA, 2009. ACM.

[4] Peter A. N. Bosman, Jörn Grahl, and Dirk Thierens. Benchmarking parameter-free AMaLGaM on functions with and without noise. *Evolutionary Computation*, 21(3):445–469, September 2013.

[5] Peter A. N. Bosman and Dirk Thierens. Linkage information processing in distribution estimation algorithms. In *GECCO*, pages 60–67, 1999.

[6] Peter A. N. Bosman and Dirk Thierens. Linkage neighbors, optimal mixing and forced improvements in genetic algorithms. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, pages 585–592, New York, NY, USA, 2012. ACM.

[7] Peter A. N. Bosman and Dirk Thierens. On measures to build linkage trees in LTGA. In Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors, *Parallel Problem Solving from Nature - PPSN XII*, volume 7491 of *Lecture Notes in Computer Science*, pages 276–285. Springer Berlin Heidelberg, 2012.

[8] Peter A. N. Bosman and Dirk Thierens. More concise and robust linkage learning by filtering and combining linkage hierarchies. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 359–366, New York, NY, USA, 2013. ACM.

[9] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[10] Erick Cantú-Paz. A survey of parallel genetic algorithms. *Calculeurs Parallelles, Reseax et Systems Repartis*, 10, 1998.

[11] Si-Cheng Chen and Tian-Li Yu. Difficulty of linkage learning in estimation of distribution algorithms. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 397–404, New York, NY, USA, 2009. ACM.

[12] Ying-Ping Chen. *Extending the Scalability of Linkage Learning Genetic Algorithms: Theory and Practice*. PhD thesis, Champaign, IL, USA, 2004. AAI3130894.

[13] Ying-Ping Chen and David E. Goldberg. Convergence time for the linkage learning genetic algorithm. *Evol. Comput.*, 13(3):279–302, September 2005.

[14] Clay Mathematics Institute. P vs NP problem. `http://www.claymath.org/millenium-problems/p-vs-np-problem`. [Online; accessed February 14, 2014].

[15] David J. Coffin and Robert E. Smith. The limitations of distribution sampling for linkage learning. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 364–369. IEEE, 2007.

[16] François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: A python framework for evolutionary algorithms. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, pages 85–92, New York, NY, USA, 2012. ACM.

[17] Kalyanmoy Deb and David E. Goldberg. Analyzing deception in trap functions. In *FOGA*, volume 2, pages 98–108, 1992.

[18] Kalyanmoy Deb and David E. Goldberg. Sufficient conditions for deceptive and easy binary functions. *Annals of Mathematics and Artificial Intelligence*, 10(4):385–408, 1994.

[19] Debian.org. The computer language benchmarks game - Java vs C gcc. `http://benchmarksgame.alioth.debian.org/u64q/benchmark.php?test=all&lang=java&lang2=gcc&data=u64q`. [Online; accessed November 25, 2014].

[20] Debian.org. The computer language benchmarks game - Java vs Python. `http://benchmarksgame.alioth.debian.org/u64q/benchmark.php?test=all&lang=java&lang2=python3&data=u64q`. [Online; accessed November 14, 2014].

[21] Thyago S. P. C. Duque and David E. Goldberg. A new method for linkage learning in the ECGA. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 1819–1820, New York, NY, USA, 2009. ACM.

[22] Leonardo R. Emmendorfer and Aurora T. R. Pozo. Effective linkage learning using low-order statistics and clustering. *Trans. Evol. Comp*, 13(6):1233–1246, December 2009.

[23] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *J. Mach. Learn. Res.*, 13(1):2171–2175, July 2012.

[24] Brian W. Goldman and William F. Punch. Parameter-less population pyramid. In *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 785–792, New York, NY, USA, 2014. ACM.

[25] Ananth Grama. *Introduction to parallel computing*. Pearson Education, 2003.

[26] Ilan Gronau and Shlomo Moran. Optimal implementations of UPGMA and other common clustering algorithms. *Information Processing Letters*, 104(6):205 – 210, 2007.

[27] Georges R. Harik. *Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms*. PhD thesis, The University of Michigan, 1997.

[28] Georges R. Harik. Linkage learning via probabilistic modeling in the ECGA. *Urbana*, 51(61):801, 1999.

[29] Georges R. Harik and David E. Goldberg. Linkage learning through probabilistic expression. *Computer Methods in Applied Mechanics and Engineering*, 186(24):295 – 310, 2000.

[30] Georges R Harik and Fernando G Lobo. A parameter-less genetic algorithm. In *GECCO*, volume 99, pages 258–267, 1999.

[31] Georges R. Harik, Fernando G. Lobo, and Kumara Sastry. Linkage learning via probabilistic modeling in the extended compact genetic algorithm (ECGA). In *Scalable optimization via probabilistic modeling*, pages 39–61. Springer, 2006.

[32] David Heckerman, Dan Geiger, and David M. Chickering. Learning bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, 1995.

[33] John H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. Michigan Univ. Press, Ann Arbor, MI, 1975.

[34] John H. Holland. Genetic algorithms. *Scientific american*, 267(1):66–72, 1992.

[35] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[36] Java-Source.net. DJProf: an aspect-oriented java profiler. http://homepages.mcs.vuw.ac.nz/~djp/djprof/. [Online; accessed November 15, 2014].

[37] Java-Source.net. Open source profilers in java. http://java-source.net/open-source/profilers. [Online; accessed November 15, 2014].

[38] JUnit. Junit - about. http://junit.org/. [Online; accessed November 27, 2014].

[39] OKTECH-Info Kft. OKTECH profiler. https://code.google.com/p/oktech-profiler/. [Online; accessed November 15, 2014].

[40] A. Kraskov, H. Stögbauer, R. G. Andrzejak, and P. Grassberger. Hierarchical clustering using mutual information. *EPL (Europhysics Letters)*, 70(2):278, 2005.

[41] Oracle. Hashmap (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html. [Online; accessed December 2, 2014].

[42] Oracle. Hprof: A heap/CPU profiling tool. https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html. [Online; accessed November 14, 2014].

[43] Oracle. Interface executorservice. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html. [Online; accessed November 20, 2014].

[44] Oracle. Lambda expressions. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html. [Online; accessed November 20, 2014].

[45] Oracle. Using JConsole. https://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html. [Online; accessed November 15, 2014].

[46] Martin Pelikan and David E. Goldberg. Escaping hierarchical traps with competent genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, 2001.

[47] Martin Pelikan, Mark W. Hauschild, and Dirk Thierens. Pairwise and problem-specific distance metrics in the linkage tree genetic algorithm. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1005–1012, New York, NY, USA, 2011. ACM.

[48] Martin Pelikan, Kumara Sastry, David E. Goldberg, Martin V. Butz, and Mark Hauschild. Performance of evolutionary algorithms on NK Landscapes with nearest neighbor interactions and tunable overlap. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 851–858, New York, NY, USA, 2009. ACM.

[49] perf4j.org. Perf4J home. http://perf4j.codehaus.org/. [Online; accessed November 15, 2014].

[50] Aaron Kans Quentin Charatan. *Java in Two Semesters*. The McGraw-Hill Companies, second edition, 2006.

[51] Reuven Y. Rubinstein. Cross-entropy and rare events for maximal cut and partition problems. *ACM Trans. Model. Comput. Simul.*, 12(1):27–53, January 2002.

[52] Krzysztof L. Sadowski, Peter A.N. Bosman, and Dirk Thierens. On the usefulness of linkage processing for solving max-sat. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 853–860, New York, NY, USA, 2013. ACM.

[53] Kumara Sastry. *Evaluation-relaxation schemes for genetic and evolutionary algorithms*. PhD thesis, 2002.

[54] Sourceforge.net. JRat the java runtime analysis toolkit. `http://jrat.sourceforge.net/`. [Online; accessed November 15, 2014].

[55] Fabien Teytaud and Olivier Teytaud. Why one must use reweighting in estimation of distribution algorithms. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 453–460, New York, NY, USA, 2009. ACM.

[56] Dirk Thierens. The linkage tree genetic algorithm. In Robert Schaefer, Carlos Cotta, Joanna Koodziej, and Gnter Rudolph, editors, *Parallel Problem Solving from Nature, PPSN XI*, volume 6238 of *Lecture Notes in Computer Science*, pages 264–273. Springer Berlin Heidelberg, 2010.

[57] Dirk Thierens and Peter A. N. Bosman. Optimal mixing evolutionary algorithms. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 617–624, New York, NY, USA, 2011. ACM.

[58] Dirk Thierens and Peter A. N. Bosman. Evolvability analysis of the linkage tree genetic algorithm. In Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors, *Parallel Problem Solving from Nature - PPSN XII*, volume 7491 of *Lecture Notes in Computer Science*, pages 286–295. Springer Berlin Heidelberg, 2012.

[59] Dirk Thierens and Peter A. N. Bosman. Predetermined versus learned linkage models. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, pages 289–296, New York, NY, USA, 2012. ACM.

[60] Dirk Thierens and Peter A. N. Bosman. Hierarchical problem solving with the linkage tree genetic algorithm. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 877–884, New York, NY, USA, 2013. ACM.

[61] Dirk Thierens and David E Goldberg. Mixing in genetic algorithms. *Urbana*, 51:61801, 1993.

[62] Sebastien Vauclair. Extensible java profiler. `http://ejp.sourceforge.net/`. [Online; accessed November 15, 2014].

[63] Richard A. Watson and Jordan B. Pollack. A computational model of symbiotic composition in evolutionary transitions. *Biosystems*, 69(23):187 – 209, 2003.

[64] Andrew Wilcox. Jip - the java interactive profiler. `http://jiprof.sourceforge.net/`. [Online; accessed November 14, 2014].

[65] Andrew Wilcox. JIP - the java interactive profiler. an effective new configurable profiler for java. 2006.

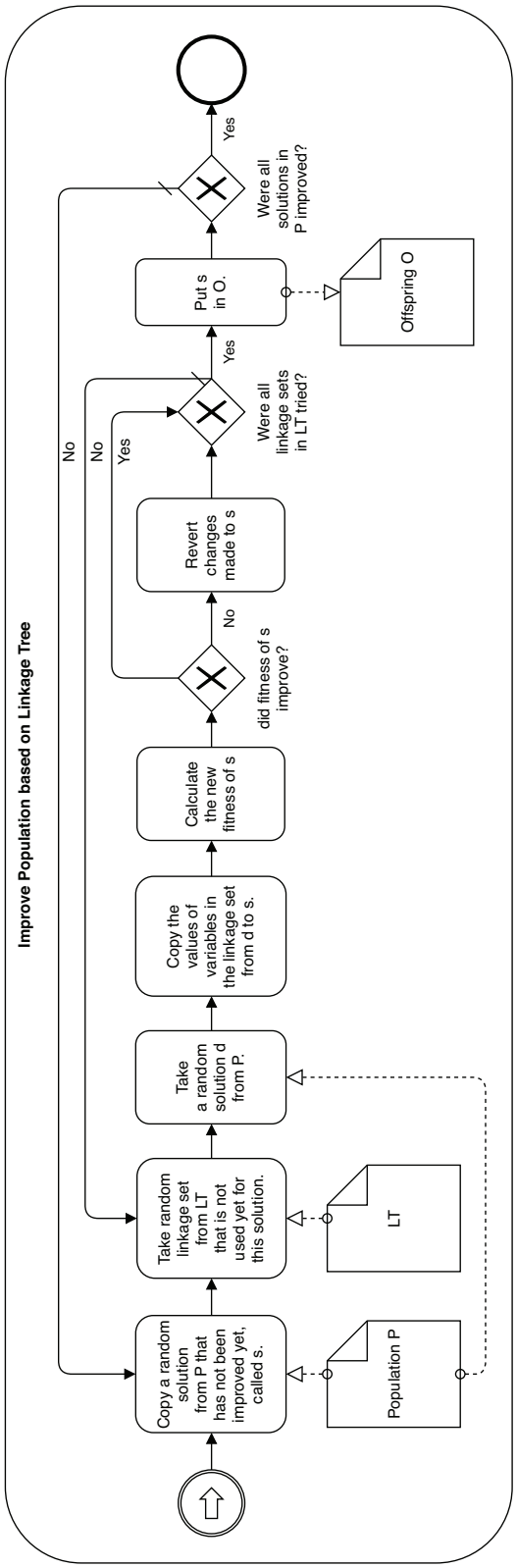
[66] Tian-Li Yu and David E. Goldberg. Toward an understanding of the quality and efficiency of model building for genetic algorithms. In Kalyanmoy Deb, editor, *Genetic and Evolutionary Computation GECCO 2004*, volume 3103 of *Lecture Notes in Computer Science*, pages 367–378. Springer Berlin Heidelberg, 2004.

[67] Tian-Li Yu, Kumara Sastry, and David E. Goldberg. Linkage learning, overlapping building blocks, and systematic strategy for scalable recombination. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1217–1224, New York, NY, USA, 2005. ACM.

Appendix A

Diagrams

Figure A.1: Gene-pool Optimal Mixing.



Appendix B

LTGA Complexity Analysis

In this complexity analysis, the complexity is expressed in terms of the following parameters:

ℓ = number of parameters
n = population size
g = amount of generations needed
f = fitness function evaluation time

```
1. population = createRandomPopulation()                          O(n*ℓ)
2. bestFound = population.getBestFound()                          O(n)
3. while(!stopConditionMet(population))                           O(g * n * ℓ(ℓ+f))
4.   //Initialize MPM and LT
5.   int[] order = getRandomOrder(ℓ)                              O(ℓ)
6.   mpm = initMPM(order)                                         O(ℓ)
7.   lt = initLT(mpm)                                             O(ℓ)
8.   MIMatrix = new MIMatrix[ℓ][ℓ]                                O(ℓ^2)
9.   for all ℓ*(ℓ+1)/2 elements in MIMatrix upper triangle        O(n*ℓ^2)
10.    dist = new Distribution(population, i, j)                  O(n)
11.    MIMatrix[i, j] = dist.getJointEntropy()                    O(1)
12.  calculateMIValues(MIMatrix)                                  O(ℓ^2)
13.
14.  //Construct LT
15.  nnChain = new NearestNeighborChain()                         O(1)
16.  while(mpm.size > 1)                                          O(ℓ^2)
17.    while(!nnChain.isComplete())                               O(ℓ)
18.      nnChain.add(getNearestNeighbor(nnChain.last))            O(ℓ)
19.    MIMatrix = updateMIMatrix(MIMatrix, nnChain.last,          O(ℓ)
       nnChain.secondLast)
20.    mpm = updateMpm(mpm, nnChain.last,                         O(ℓ)
       nnChain.secondLast)
21.    nnChain.truncate(nnChain.size - 3)                         O(1)
22.
23.  //Generate offspring
24.  offspring = []                                               O(1)
25.  for i = 0 → n                                                O(n*ℓ(ℓ+f))
26.    result = population[i].copy()                              O(ℓ)
27.    backup = population[i].copy()                              O(ℓ)
28.    order = getRandomOrder(lt.size)                            O(ℓ)
29.    for j = 0 → lt.size                                        O(ℓ(ℓ+f))
30.      donor = population.getRandom()                           O(1)
31.      result = copyForLinkageSet(result, donor,                O(ℓ)
         lt[order[j]])
32.      if(result != backup && fitness(result) >=                O(f)
         fitness(backup))
33.        backup = result                                        O(1)
34.      else
35.        result = backup                                        O(1)
36.
37.    //Forced Improvement
38.    if(result == population[i] || noImprovementStretch        O(1)
       > 1+log(n))
39.      order = getRandomOrder(lt.size)                          O(ℓ)
```

```
40.      j = 0                                                    O(1)
41.      while j < lt.size && result == population[i]             O(ℓ(ℓ+f))
42.        result = copyForLinkageSet(result, bestFound,          O(ℓ)
           lt[order[j]])
43.        if(result != backup && fitness(result) >               O(f)
           fitness(backup))
44.          backup = result                                      O(1)
45.        else
46.          result = backup                                      O(1)
47.        j++                                                    O(1)
48.    if(result == population[i])                                O(1)
49.      result = bestFound                                       O(1)
50.    offspring[i] = result                                      O(1)
51.
52.  population = offspring                                       O(1)
53.  bestFound = population.getBestFound()                        O(n)
54.return bestFound                                              O(1)
```

Appendix C

Profiling Results

Table C.1: Sequential Profiling for NK-Landscape, $\ell = 400, n = 400$. Before initial optimizations.

Average total execution time: 239.46687 seconds.

Method	**Call count**	**Time (s)**	**Time (%)**
`sequential.Population.initialize`	1.00	0.1499	0.06
`sequential.LinkageTree.learnStructure`	7.26	1.72616	0.72
`sequential.MIMatrix.constructMIMatrix`	7.26	1.69615	0.71
`sequential.MIMatrix.constructMIMatrix - Entropy`	7.26	1.69194	0.71
`sequential.MIMatrix.constructMIMatrix - MIValues`	7.26	0.00414	0.00
`shared.NearestNeighborChain.getNNTuple`	2896.74	0.00732	0.00
`sequential.MIMatrix.updateMIMatrix`	2896.74	0.00904	0.00
`sequential.LinkageTree.constructNewMpm`	2896.74	0.00574	0.00
`sequential.Population.makeOffspring`	7.26	239.1743	99.88
`sequential.Population.generateAndEvaluateNewSolutionsToFillOffspring`	7.26	237.44797	99.16
`shared.ProblemEvaluator.installedProblemEvaluation`	809612.83	235.44046	98.32

Table C.2: Sequential Profiling for NK-Landscape, $\ell = 400, n = 400$. After initial optimizations.

Average total execution time: 18.76841 seconds.

Method	**Call count**	**Time (s)**	**Time (%)**
`sequential.Population.initialize`	1.00	0.01003	0.05
`sequential.LinkageTree.learnStructure`	7.88	2.10218	11.20
`sequential.MIMatrix.constructMIMatrix`	7.88	2.06442	11.00
`sequential.MIMatrix.constructMIMatrix - Entropy`	7.88	2.05904	10.97
`sequential.MIMatrix.constructMIMatrix - MIValues`	7.88	0.00534	0.03
`shared.NearestNeighborChain.getNNTuple`	3144.12	0.00787	0.04
`sequential.MIMatrix.updateMIMatrix`	3144.12	0.01066	0.06
`sequential.LinkageTree.constructNewMpm`	3144.12	0.00684	0.04
`sequential.Population.makeOffspring`	7.88	16.58247	88.35
`sequential.Population.generateAndEvaluateNewSolutionsToFillOffspring`	7.88	16.58245	88.35
`shared.Population.generateNewSolution`	3152.00	16.57979	88.34
`shared.ProblemEvaluator.installedProblemEvaluation`	929744.51	14.55443	77.55

Table C.3: PP-LTGA Profiling for NK-Landscape, $\ell = 400$, $n = 400$, $p = 64$

Average total execution time: 3.708835 seconds.

Method	**Call count**	**Time (s)**	**Time (%)**
parallel.Population.initialize	1.00	0.01731	0.47
parallel.LinkageTree.learnStructure	7.84	0.87998	23.73
parallel.MIMatrix.constructMIMatrix	7.84	0.40308	10.87
parallel.MIMatrix.constructMIMatrix - Entropy	7.84	0.29061	7.84
parallel.MIMatrix.constructMIMatrix - MIValues	7.84	0.11165	3.01
shared.NearestNeighborChain.getNNTuple	3126.96	0.0139	0.28
parallel.MIMatrix.updateMIMatrix	3126.96	0.0194	0.52
parallel.LinkageTree.constructNewMpm	3126.96	0.40409	10.90
parallel.Population.makeOffspring	7.84	2.8103	75.77
parallel.Population.generateAndEvaluateNewSolutionsToFillOffspring	7.84	2.80975	75.76

Appendix D

Offline-learned LT Linkage Sets Frequencies

In this Appendix, percentual frequencies in LT_{on}s of linkage sets contained by LT_{off}s are shown per generation, based on 100 independent executions of the LTGA. Results are only shown for the generations encountered in these 100 executions. We use the following notations:

[i, j, k] = linkage set $\{x_i, x_j, x_k\}$
$\mathcal{F}_{g_i}(x)$ = percentual frequency of linkage set x in the LT_{on}s learned for generation i
$\mathcal{F}_{avg}(x)$ = average percentual frequency of linkage set x in the LT_{on}s over all generations

To illustrate, this means that if $\mathcal{F}_{g_2}([0,\ 2,\ 4,\ 5]) = 75\%$, linkage set $\{x_0, x_2, x_4, x_5\}$ was contained by the LT_{on} of generation 2 in 75 out the 100 executions performed.

Additionally, in the results shown for *Onemax* and *Deceptive Trap*, elements are printed in bold that are known to be part of the Family of Subsets (FOS) that perfectly represents the linkage between variables in these problems. Note that for *NK-Landscapes* and *MAXCUT* no such FOS is known. For each problem, results are shown for a specific size, and for *NK-Landscapes* and *MAXCUT* for a specific instance, that are suitable for showing behavior that is representative for other problem instances in an orderly fashion.

Table D.1: Percentual Frequencies for *Onemax*, $\ell = 10$.

Linkage Set in LT_{off}	$\mathcal{F}_{g_0}$	$\mathcal{F}_{g_1}$	$\mathcal{F}_{g_2}$	$\mathcal{F}_{g_3}$	$\mathcal{F}_{avg}$	in pruned LT_{off}
[0]	100%	100%	100%	100%	100%	1
[1]	100%	100%	100%	100%	100%	1
[2]	100%	100%	100%	100%	100%	1
[3]	100%	100%	100%	100%	100%	1
[4]	100%	100%	100%	100%	100%	1
[5]	100%	100%	100%	100%	100%	1
[6]	100%	100%	100%	100%	100%	1
[7]	100%	100%	100%	100%	100%	0
[8]	100%	100%	100%	100%	100%	1
[9]	100%	100%	100%	100%	100%	1
[4, 5]	7%	9%	8%	25%	12%	0
[4, 5, 7]	2%	1%	0%	0%	1%	0
[1, 6]	5%	7%	6%	17%	9%	0
[0, 9]	10%	10%	15%	0%	9%	0
[2, 3]	5%	9%	12%	8%	9%	0
[2, 3, 4, 5, 7]	0%	0%	0%	0%	0%	0
[0, 8, 9]	0%	0%	0%	0%	0%	0
[0, 1, 6, 8, 9]	1%	0%	0%	0%	0%	0

Table D.2: Percentual Frequencies for *Deceptive Trap*, $\ell = 15$.

Linkage Set in LT_{off}	$\mathcal{F}_{g_0}$	$\mathcal{F}_{g_1}$	$\mathcal{F}_{g_2}$	$\mathcal{F}_{g_3}$	$\mathcal{F}_{g_4}$	$\mathcal{F}_{g_5}$	$\mathcal{F}_{g_6}$	$\mathcal{F}_{avg}$	in pruned LT_{off}
[0]	100%	100%	100%	100%	100%	100%	100%	100%	0
[1]	100%	100%	100%	100%	100%	100%	100%	100%	0
[2]	100%	100%	100%	100%	100%	100%	100%	100%	0
[3]	100%	100%	100%	100%	100%	100%	100%	100%	0
[4]	100%	100%	100%	100%	100%	100%	100%	100%	0
[5]	100%	100%	100%	100%	100%	100%	100%	100%	0
[6]	100%	100%	100%	100%	100%	100%	100%	100%	0
[7]	100%	100%	100%	100%	100%	100%	100%	100%	0
[8]	100%	100%	100%	100%	100%	100%	100%	100%	0
[9]	100%	100%	100%	100%	100%	100%	100%	100%	0
[10]	100%	100%	100%	100%	100%	100%	100%	100%	0
[11]	100%	100%	100%	100%	100%	100%	100%	100%	0
[12]	100%	100%	100%	100%	100%	100%	100%	100%	0
[13]	100%	100%	100%	100%	100%	100%	100%	100%	0
[14]	100%	100%	100%	100%	100%	100%	100%	100%	0
[6, 8]	0%	11%	15%	27%	21%	17%	0%	15%	0
[5, 7]	5%	11%	14%	26%	15%	14%	100%	14%	0
[5, 6, 7, 8]	0%	10%	16%	0%	0%	0%	0%	4%	0
[0, 2]	4%	10%	16%	15%	24%	10%	0%	13%	0
[0, 2, 3]	0%	7%	6%	8%	13%	5%	0%	6%	0
[5, 6, 7, 8, 9]	0%	74%	100%	100%	99%	69%	0%	74%	0
[11, 13]	4%	15%	12%	17%	19%	17%	0%	14%	0
[0, 1, 2, 3]	0%	15%	17%	0%	0%	0%	0%	5%	0
[12, 14]	7%	14%	15%	19%	18%	12%	0%	14%	0
[11, 12, 13, 14]	0%	19%	15%	0%	0%	2%	0%	6%	0
[10, 11, 12, 13, 14]	0%	83%	99%	100%	99%	43%	0%	71%	1
[0, 1, 2, 3, 4]	1%	80%	99%	100%	98%	55%	100%	72%	1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	0%	24%	35%	34%	41%	17%	0%	25%	0

Table D.3: Percentual Frequencies for *NK-Landscapes*, $\ell = 20$, instance of median difficulty.

Linkage Set in LT_{off}	$\mathcal{F}_{g_0}$	$\mathcal{F}_{g_1}$	$\mathcal{F}_{g_2}$	$\mathcal{F}_{g_3}$	$\mathcal{F}_{g_4}$	$\mathcal{F}_{g_5}$	$\mathcal{F}_{g_6}$	$\mathcal{F}_{avg}$	in pruned LT_{off}
[0]	100%	100%	100%	100%	100%	100%	100%	100%	1
[1]	100%	100%	100%	100%	100%	100%	100%	100%	0
[2]	100%	100%	100%	100%	100%	100%	100%	100%	1
[3]	100%	100%	100%	100%	100%	100%	100%	100%	1
[4]	100%	100%	100%	100%	100%	100%	100%	100%	0
[5]	100%	100%	100%	100%	100%	100%	100%	100%	0
[6]	100%	100%	100%	100%	100%	100%	100%	100%	1
[7]	100%	100%	100%	100%	100%	100%	100%	100%	0
[8]	100%	100%	100%	100%	100%	100%	100%	100%	0
[9]	100%	100%	100%	100%	100%	100%	100%	100%	1
[10]	100%	100%	100%	100%	100%	100%	100%	100%	0
[11]	100%	100%	100%	100%	100%	100%	100%	100%	0
[12]	100%	100%	100%	100%	100%	100%	100%	100%	0
[13]	100%	100%	100%	100%	100%	100%	100%	100%	1
[14]	100%	100%	100%	100%	100%	100%	100%	100%	1
[15]	100%	100%	100%	100%	100%	100%	100%	100%	0
[16]	100%	100%	100%	100%	100%	100%	100%	100%	1
[17]	100%	100%	100%	100%	100%	100%	100%	100%	0
[18]	100%	100%	100%	100%	100%	100%	100%	100%	1
[19]	100%	100%	100%	100%	100%	100%	100%	100%	0
[9, 10]	4%	5%	1%	0%	0%	2%	0%	2%	0
[7, 8]	6%	1%	0%	0%	0%	0%	0%	1%	1
[16, 19]	9%	0%	0%	0%	0%	2%	0%	2%	0
[11, 12]	5%	100%	100%	99%	56%	9%	0%	53%	1
[14, 16, 19]	1%	0%	0%	0%	0%	0%	0%	0%	1
[1, 3]	4%	12%	3%	3%	11%	9%	0%	6%	1
[4, 6]	3%	13%	6%	2%	2%	7%	0%	5%	1
[1, 2, 3]	0%	3%	20%	60%	71%	7%	0%	23%	0
[1, 2, 3, 18]	0%	0%	0%	0%	0%	2%	0%	0%	0
[4, 5, 6]	0%	6%	7%	4%	2%	0%	0%	3%	0
[1, 2, 3, 4, 5, 6, 18]	0%	0%	0%	0%	0%	0%	0%	0%	0
[1, 2, 3, 4, 5, 6, 7, 8, 18]	0%	0%	0%	0%	0%	0%	0%	0%	1
[13, 15]	4%	35%	44%	63%	74%	32%	25%	40%	1
[9, 10, 11, 12]	0%	2%	4%	0%	0%	0%	0%	1%	1
[13, 15, 17]	0%	0%	1%	7%	13%	12%	0%	5%	0
[9, 10, 11, 12, 14, 16, 19]	0%	0%	0%	0%	0%	0%	0%	0%	1
[9, 10, 11, 12, 13, 14, 15, 16, 17, 19]	0%	0%	2%	0%	0%	0%	0%	0%	0
[0, 1, 2, 3, 4, 5, 6, 7, 8, 18]	0%	0%	0%	0%	0%	0%	0%	0%	0

Table D.4: Percentual Frequencies for *MAXCUT*, $\ell = 12$, instance of median difficulty.

Linkage Set in LT_{off}	$\mathcal{F}_{g_0}$	$\mathcal{F}_{g_1}$	$\mathcal{F}_{g_2}$	$\mathcal{F}_{g_3}$	$\mathcal{F}_{g_4}$	$\mathcal{F}_{g_5}$	$\mathcal{F}_{g_6}$	$\mathcal{F}_{avg}$	in pruned LT_{off}
[0]	100%	100%	100%	100%	100%	100%	100%	100%	1
[1]	100%	100%	100%	100%	100%	100%	100%	100%	1
[2]	100%	100%	100%	100%	100%	100%	100%	100%	0
[3]	100%	100%	100%	100%	100%	100%	100%	100%	0
[4]	100%	100%	100%	100%	100%	100%	100%	100%	1
[5]	100%	100%	100%	100%	100%	100%	100%	100%	1
[6]	100%	100%	100%	100%	100%	100%	100%	100%	0
[7]	100%	100%	100%	100%	100%	100%	100%	100%	0
[8]	100%	100%	100%	100%	100%	100%	100%	100%	1
[9]	100%	100%	100%	100%	100%	100%	100%	100%	1
[10]	100%	100%	100%	100%	100%	100%	100%	100%	1
[11]	100%	100%	100%	100%	100%	100%	100%	100%	0
[7, 10]	6%	14%	6%	5%	12%	6%	0%	7%	1
[3, 9]	5%	38%	70%	64%	24%	18%	0%	31%	1
[5, 11]	7%	27%	45%	35%	19%	29%	0%	23%	0
[1, 8]	9%	39%	56%	32%	23%	18%	0%	25%	1
[1, 5, 8, 11]	1%	9%	31%	18%	5%	6%	0%	10%	1
[0, 2]	5%	10%	2%	2%	9%	12%	0%	6%	0
[4, 6]	11%	58%	90%	94%	73%	71%	100%	71%	1
[0, 2, 3, 9]	0%	9%	8%	8%	4%	6%	0%	5%	0
[0, 2, 3, 4, 6, 9]	0%	4%	8%	7%	4%	12%	0%	5%	0
[1, 5, 7, 8, 10, 11]	0%	2%	5%	3%	5%	6%	0%	3%	0

In Search of Optimal Linkage Trees

Roy de Bokx
Delft University of Technology
Delft, The Netherlands
Rdebokx1990@gmail.com

Dirk Thierens
Utrecht University
Utrecht, The Netherlands
D.Thierens@uu.nl

Peter A.N. Bosman
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
Peter.Bosman@cwi.nl

ABSTRACT

The recently introduced Linkage Tree Genetic Algorithm (LTGA) has been shown to exhibit excellent scalability on a variety of optimization problems. LTGA employs Linkage Trees (LTs) to identify and exploit linkage information between problem variables. Much is already understood about LTGA's performance, but it is still unclear whether the LT model can be further improved upon. In this paper we analyze the results of learning LTs offline by optimizing LTGA's performance as a function of static LTs. This results in a better performance of LTGA than with online-learned LTs as problem complexity increases. Further analysis of the offline-learned LTs indicates that pruning the LT can result in a further performance improvement of the LTGA. Using a population-size-free internally parallelized version of LTGA, we found that the optimal subset of the offline-learned LT typically contains only about 50% of the nodes. This suggests that the LT contains redundancies that may possibly still be exploited to improve the performance of LTGA with online-learned LTs.

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Performance, Experimentation

Keywords

Evolutionary Computation, Parallel Computation, Genetic Algorithms, Estimation-of-Distribution Algorithms, Linkage Learning, Optimal Mixing, Linkage Tree Genetic Algorithm

1. INTRODUCTION

It is known that variable linkage is highly important in solving certain problems efficiently. In particular, if strong linkage exists between a set of variables, the chance of generating low-fitness offspring is high when these variables are not transferred to the offspring together [12]. Linkage-learning Evolutionary Algorithms (EAs) use *linkage learning* to construct a *linkage model*, which is exploited to solve problems efficiently by taking into account important linkages during variation. It has been shown that when this linkage model is aligned correctly with the structure of the problem that is to be solved, EAs are capable of solving such problems efficiently by performing variation based on this linkage model, see e.g. [1]. One algorithm in particular has shown to be very promising, which is the Linkage Tree Genetic Algorithm (LTGA). This algorithm uses a Linkage Tree (LT) as a linkage model to identify the problem's structure, enabling it to solve various problems very efficiently, requiring a smaller population size and less execution time than most well-known problem-structure exploiting Evolutionary Algorithms, such as those in the class of Estimation-of-Distribution Algorithms (EDAs). This holds for well-known benchmark problems, including *Onemax*, k-order *Deceptive Trap*, maximum overlapping nearest-neighbor *NK-Landscapes*, *MAXCUT* and hierarchically structured *HIFF* and *HTRAP* problems.

Understanding the reasons for the excellent performance of LTGA is highly valuable. Besides hierarchically structured problems, LTGA is also able to efficiently solve problems for which a tree-like linkage model seems inappropriate. Moreover, it has been shown that using a predetermined linkage model that exactly resembles a problem's formulation structure instead of the LTs that are learned from the population for every generation, not always results in a better performance, especially for more complex problems such as the *NK-Landscapes* and *MAXCUT* problem, where the results are even much worse [10].

This brings us to ask the question what in fact makes a linkage model ideal for LTGA to be used, either predetermined, or learned online. This paper pursues to answer this question by searching for an optimal replacement for the LT in LTGA within the space of binary trees and subsets of binary trees. Finding such an optimal replacement can provide great insights. Ultimately, this might even enable us to improve the linkage model used by LTGA.

The remainder of this paper is organized as follows. We first provide a basic understanding of LTGA in Section 2 to support further sections. This is followed by a brief summary of related work in Section 3. Next, experiments aimed at finding the optimal offline-learned LT are discussed in section 4. Based on the results of these experiments, addi-

GECCO'15, July 11-15, 2015, Madrid, Spain.

tional experiments are done aimed at investigating the effects of local search in Section 5 and the effects of pruning the offline-learned LTs in Section 6. Finally, conclusions are presented in Section 7.

2. LTGA

The Linkage Tree Genetic Algorithm (LTGA) is a state-of-the-art linkage learning EA that uses a Linkage Tree (LT) as linkage model to model and exploit variable linkages [1]. This LT supports efficient variation operations with the use of Optimal Mixing (OM), enabling LTGA to converge towards the optimal solution efficiently, requiring a smaller population size and less run time than most Linkage Learning Evolutionary Algorithms for all benchmark problems tested so far. A selection of these benchmark problems will also be considered in this paper. In the remainder of this section we will present a short outline of LTGA itself.

2.1 Benchmark problems

Let ℓ be the number of variables. The first problem is the well-known *Onemax* problem, which can be expressed by the following formula.

$$f_{Onemax}(x) = \sum_{i=0}^{\ell-1} x_i \tag{1}$$

The second benchmark problem is the non-overlapping additively decomposable composition of k-order *Deceptive Trap* functions, denoted by the formula below. We consider subfunctions with $k = 5$.

$$f_{Trap}(x) = \sum_{i=0}^{(\ell/k)-1} f^{sub}_{Trap\text{-}k}\left(\sum_{j=ki}^{ki+k-1} x_j \right) \tag{2}$$

where

$$f^{sub}_{Trap\text{-}k}(u) = \begin{cases} 1 & \text{if } u = k \\ \frac{k-1-u}{k} & otherwise \end{cases}$$

Thirdly, the problem of maximum overlapping nearest-neighbor *NK-Landscapes* is considered, which is a additively decomposable composition of predetermined, but completely random subfunctions of length k. Again, problem instances with $k = 5$ are considered. This problem is expressed by formula (3). For this problem, for each problem size, 100 randomly generated problem instances were used.

$$f_{NK\text{-}S1}(x) = \sum_{i=0}^{\ell-k} f^{sub}_{NK}(x_{i,i+1,\ldots,i+k-1}) \tag{3}$$

Last, the well-known NP-Complete weighted *MAXCUT* problem is used as a benchmark, which is defined given a weighted undirected graph with a set of ℓ vertices $V = \{v_0, v_1, ..., v_{\ell-1}\}$, a set of edges E between the vertices, and a weight w_{ij} for each edge $(v_i, v_j) \in E$. The goal in weighted MAXCUT is to split V into two sets such that the sum of the weights of all edges that are thereby cut, i.e. running between vertices in different sets, is maximized. By introducing a single binary variable x_i for every vertex that indicates if vertex v_i is either in set 0 or set 1, the function to be optimized can be expressed by formula (4). For this problem, for each problem size, 10 randomly generated fully connected graphs were generated. Weights were set randomly using a β distribution with parameters $\alpha = 100, \beta = 1$ and scaled to the range of [1...5].

$$f_{weighted\ MAXCUT}(x) = \sum_{(v_i,v_j)\in E} \begin{cases} w_{ij} & \text{if } x_i \neq x_j \\ 0 & otherwise \end{cases} \tag{4}$$

```
1. P = generateRandomSolutions(n)
2. while stopCriteriaNotMet()
3.   M = buildDistanceMatrix(P)
4.   LT = learnLinkageTree(M)
5.   Offspring = GOM(P, LT)
6.   bestSoFar = Offspring.getBest()
7.   P = Offspring
8. return bestSoFar
```

Figure 1: The Linkage Tree Genetic Algorithm

2.2 The Algorithm

In 2010, Thierens presented the first implementation of the Linkage Tree Genetic Algorithm (LTGA) [8]. Experiments discussed in this paper were done with the most recent version of LTGA [1] of which the code can be found online[1]. However, Tournament Selection performed before learning the LT in every generation was removed as it was found that the additional selection pressure imposed by this increases the minimally required population size, which ultimately negatively influences performance.

LTGA distinguishes itself from earlier model-based EAs by learning an LT to drive variation. This LT is learned online based on the population in each generation. This way, LTGA aims to identify important variable linkages and exploit these such that important building blocks are efficiently exchanged between candidate solutions. Since the first implementation of LTGA, several improvements were developed, such as the use of Gene-pool Optimal Mixing (GOM), Unweighted Pair Group Method with Arithmetic-mean (UPGMA) [5] and Forced Improvement.

A high-level outline of LTGA is presented in Figure 1. LTGA starts with a population P of n randomly generated solutions. Every generation, first a *distance matrix* is built, containing pair-wise distances for all variable pairs. The distances are in turn based on the frequencies of variable values in P [1]. A typical measure used is the Mutual Information (MI) between two variables, which is a statistical expression of dependency between stochastic random variables, rooted in information theory. MI is large for close neighbors, i.e. strongly dependent variables, which is why the distance matrix used by LTGA, also called the MIMatrix, contains negated MI values. Next, the LT is learned based on this MIMatrix. Starting with all singleton variable sets, also called *linkage sets*, a bottom-up agglomerative hierarchical clustering algorithm repeatedly merges linkage sets. By merging *reciprocal nearest neighbors*, based on the MIMatrix, a tree-like linkage model, called the Linkage Tree, can be constructed in only $O(n\ell^2)$ time, where ℓ is the number of variables [5]. An example of such an LT is displayed in Figure 2.

[1] `http://homepages.cwi.nl/~bosman/source_code.php`

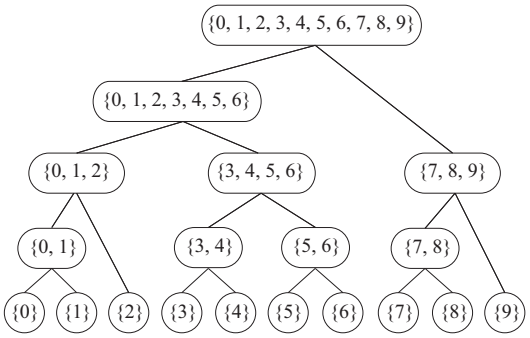


Figure 2: An example Linkage Tree.

After the LT is learned, offspring are generated by performing a procedure called Gene-pool Optimal Mixing (GOM) [1]. During GOM, for each solution, the LT is traversed in a per-solution random order, in which for each linkage set in the LT a random donor solution from P is chosen. The values of the variables contained by the linkage set are copied from the donor to the solution if and only if this results in a solution of which the fitness is at least as good as the original, which is called Optimal Mixing. After this is done, the improved solution is stored in the collection of offspring.

Conclusively, after each generation the best solution found so far is replaced by the best solution of the generated offspring and P is replaced by the generated offspring. A consecutive generation follows based on this new population if none of the stop criteria are met, otherwise the best solution found so far will be returned.

Note that any iterative process that concerns learning or variation is done in a per-case random order, meaning that the performance of LTGA, as well as the results of the experiments shown in this paper are independent of the variable ordering of problem formulations.

3. RELATED WORK

Related research has been performed on topics aimed at uncovering the reasons behind the strengths of LTGA, such as on the effects of different metrics for detecting linkage, as to study what it is that is needed from a measure in order for LTGA to converge to the optimal solution efficiently [3]. *Mutual Information* showed to have useful properties, which is used in the current implementation of LTGA. Additionally, metrics were presented aimed at evaluating the extent to which the linkage sets in an LT support convergence towards the optimal solution. The *Linkage Model Evolvability* and the *Evolvability-Based Fitness Distance Correlation* showed to be most useful for providing insight into the performance of LTGA, showing that even though LTGA is not always able to represent all (overlapping) information, it is able to capture enough of the problem's structure to solve it reliably and efficiently [9].

Ensuing these results, Thierens and Bosman studied alternatives to online-learned LTs (LT_{on}s), targeted explicitly at modeling linkage overlap [1]. This resulted in the Linkage Trees and Neighbors Genetic Algorithm (LTNGA) that constructs a linkage model by taking the intersection of the linkage models learned by LTGA and the Multi-scale Linkage Neighbors Genetic Algorithm (MLNGA) for each generation. This LTNGA performs comparable to LTGA for *Onemax* and *Deceptive Trap*, and slightly better for *NK-Landscapes* and *MAXCUT*. Additionally, research has been done aimed at constructing optimal fixed structures to replace the online-learned LTs of LTGA, which showed that for the *Onemax* and *Deceptive Trap* a straight-forward fixed structure indeed supports a better performance for LTGA [10, 11]. For problems with a more intricate structure, however, such as the *NK-Landscapes* and *MAXCUT* problems, such predetermined models were shown to be far less scalable, meaning that for larger problem instances the optimal solution could not be found within reasonable time using the predetermined linkage models, in contrast to the conventional LTGA [10]. This is contrary to what is expected and poses the question what in fact makes a linkage model ideal for LTGA to be used, either predetermined, or learned dynamically.

4. OFFLINE LINKAGE TREE LEARNING

To study the strengths and weaknesses of LTGA, we performed experiments aimed at learning LT offline to be used as a predetermined linkage model for LTGA, replacing the LT_{on}s. Contrary to conventional approaches, this is done by searching in the space of LTs and evaluating the associated performance of LTGA, as will be discussed below. It is expected that an offline-learned LT (LT_{off}) will contain linkage sets that will best support LTGA in converging towards the optimal solution. By constructing LT_{off}s in this manner and comparing them with the LT_{on}s used by LTGA, we can verify to what extent LTGA is actually able to capture the structure of the problem at hand, and moreover, whether this is actually beneficial to the performance of the algorithm.

4.1 Search Space

As discussed in Section 2.2, the LT is built based on a distance matrix by means of a hierarchical clustering algorithm that iteratively combines nearest linkage neighbors. The exact contents of the LT therefore depend on the specific contents of the distance matrix.

When searching for the optimal LT_{off}, an efficient mechanism is needed to traverse the space of possible LT_{off}s. Using a binary encoding to represent all possible LT_{off}s is non-trivial and would result in a problem with high dimensionality, of which it is questionable whether it can be solved within reasonable time. Instead, the outcome of the hierarchical clustering algorithm as discussed above can be manipulated by changing the contents of the distance matrix, resulting in LTs that represent different linkage contexts. Using the iAMaLGaM algorithm, a real-valued EA [2], the solution space of possible LT_{off}s can be traversed efficiently by finding optimal values for the continuous parameters in the upper triangle of the distance matrix. Note that the problem that iAMaLGaM is solving is far more complex than the original problem, as for a problem with ℓ variables, the upper triangle of the distance matrix consists of $\ell(\ell-1)/2$ continuous parameters that have to be optimized. Additionally, fitness function evaluations are far more expensive. Therefore, generally only LT_{off}s could be found within reasonable time for problems with $\ell \leq 25$. The fitness of a distance matrix is then defined by the performance of LTGA with the associated LT_{off}, for which a novel metric was implemented.

4.2 Fitness

In this work, a metric is used for evaluating the performance of predetermined linkage models that is less expensive than the typical use of multiple bisections to determine the minimally required population size [1]. As indicated by Goldman and Punch, no implementation of LTGA existed yet that could overcome the problem of premature convergence without first requiring information about the ideal population size to be set [4]. Therefore, we implemented a population-size-free scheme of LTGA that iteratively starts an instance of LTGA with an increasing population size n. If after the termination of an instance, starting at $n = 1$, none of the stop criteria, e.g. the optimal solution was found, were met, a new instance is started with population size $n = 2n_{previous}$.

The average number of evaluations needed by this population-size-free scheme over 1000 independent runs with fixed, but random, seeds defines the fitness of an LT_{off}, where per run the population-size-free scheme is stopped when the optimal solution was found. In this situation, the average number of evaluations is preferred over the median number of evaluations as the average contains more information, for the number of evaluations can be largely varying between independent runs. For *Onemax* and *Deceptive Trap*, 1000 independent runs are performed. For *NK-Landscapes* and *MAXCUT*, 1000 independent runs were performed for each problem size on the easiest, median and hardest problem instances. These are the instances that require the least, median and largest minimally required population size respectively for which LTGA is able to solve the problem for 99 out of 100 runs [1].

This metric is far less computationally intensive than finding the minimally required population size with multiple bisections [1] and could therefore be used by iAMaLGaM to optimize distance matrices in search for the optimal LT_{off}.

4.3 Results

We compare the performance of LTGA using LT_{off}s found by iAMaLGaM with the conventional LTGA that uses LT_{on}s learned by hierarchical clustering of the population-based MIMatrix in order to give insight into the extent to which LTGA is able to identify linkage sets that contribute to a better performance.

4.3.1 Linkage Tree Performance Comparison

LT_{off}s are compared with LT_{on}s in terms of performance of LTGA with the population-size-free scheme described above. Note that just comparing the performance of LTGA using these LTs is in a sense not a fair comparison, as the performance of the conventional LTGA includes the costs of both exploration, being the learning of the problem's structure online, and exploitation, being the traversal of the search space using this structure. For the LT_{off}s, exploration is done by iAMaLGaM and using the LT_{off}, LTGA only performs exploitation. The costs of learning LT_{off}s with iAMaLGaM are not considered as this is vastly more expensive and is solely aimed at unveiling characteristics of optimal linkage models, rather than functioning as an alternative linkage learning mechanism. However, defining the exact costs of online linkage learning is all but trivial. For *Onemax* no significant linkage learning can be done. For a problem with a clear problem structure, such as *Deceptive Trap*, a very rough indication of the costs can be made. Experiments showed that after the first, and for larger problems after the second, generation, the important linkage sets are present in the LT more than 98% of the time. However, the exact costs of online linkage learning cannot be accurately expressed in terms of generations. Moreover, for more intricate problems such as *NK-Landscapes* and *MAXCUT*, the learning process might not be as straight-forward as for *Deceptive Trap*, because there may be other important problem structures in addition to the linkage structure, such as symmetries and multi-modalities, i.e. local optima. Therefore, in order to make an accurate comparison, more research has to be done in exactly defining the costs of online linkage learning.

4.3.2 Offline-learned Linkage Tree Performance

The results in Figure 3 show that the LT_{off}s found for *Onemax* cause a slightly worse performance compared to LT_{on}s. This is rather trivial, as for *Onemax* no variable dependencies exist, meaning that the set of singleton linkage sets, ideally represents the linkage structure, which is always included by both LT_{off}s and LT_{on}s. For *Deceptive Trap*, iAMaLGaM is able to learn LT_{off}s that make LTGA perform slightly better, however also here the difference is marginal and even decreasing for larger problem sizes.

For *NK-Landscapes*, however, results show that LTGA using the LT_{off}s substantially outperforms the conventional LTGA for all problem instances tested. Moreover, results with LT_{off}s seem to be slightly better scalable, as the difference with the performance of the conventional LTGA increases with the complexity of the problem instances. This suggests that the LT_{off}s learned are intrinsically better than the LT_{on}s used by LTGA. This also applies to *MAXCUT*, for which also increasingly better performing LT_{off}s could be found as the problem complexity increases.

While earlier attempts to construct a predetermined model that could outperform the LT_{on}s were not fruitful [10], it is now shown that indeed such predetermined linkage models exist. This shows that the LT_{on}s used by LTGA might not be optimal linkage models. In order to understand why the LT_{on}s could be outperformed and what flaws these LTs have, the differences between the LT_{off}s and LT_{on}s have to be investigated in more detail, which we do next.

4.3.3 Linkage Tree Comparison

Linkage Subsets in LT_{off}s are driven by LTGA's performance with these LT_{off}s and thus in a sense must contain key linkage structures. To obtain insight into the extent to which LTGA also identifies these structures, we study the overlap between the LT_{off}s and the LT_{on}s used by LTGA. We perform 100 independent runs of LTGA with the minimally required population size, in which occurrences of the linkage sets of the LT_{off} are tracked for every LT_{on}, i.e. for every generation in every run. As an example, for *Deceptive Trap* with $\ell = 10$ the occurrence ratios are presented in Table 1 and for the easy problem instance of NK-Landscapes with $\ell = 10$ the occurrence ratios are presented in Table 2.

For *Onemax*, these results are rather trivial, as the important linkage sets of this problem consist of all singleton linkage sets, which is always contained in both LT_{off}s and LT_{on}s. This is backed by the fact that all other sets in the LT_{off} have a low occurrence ratio in the LT_{on}s found.

For *Deceptive Trap*, however, results are more insightful. Experiments were performed for $k = 5$, meaning that the

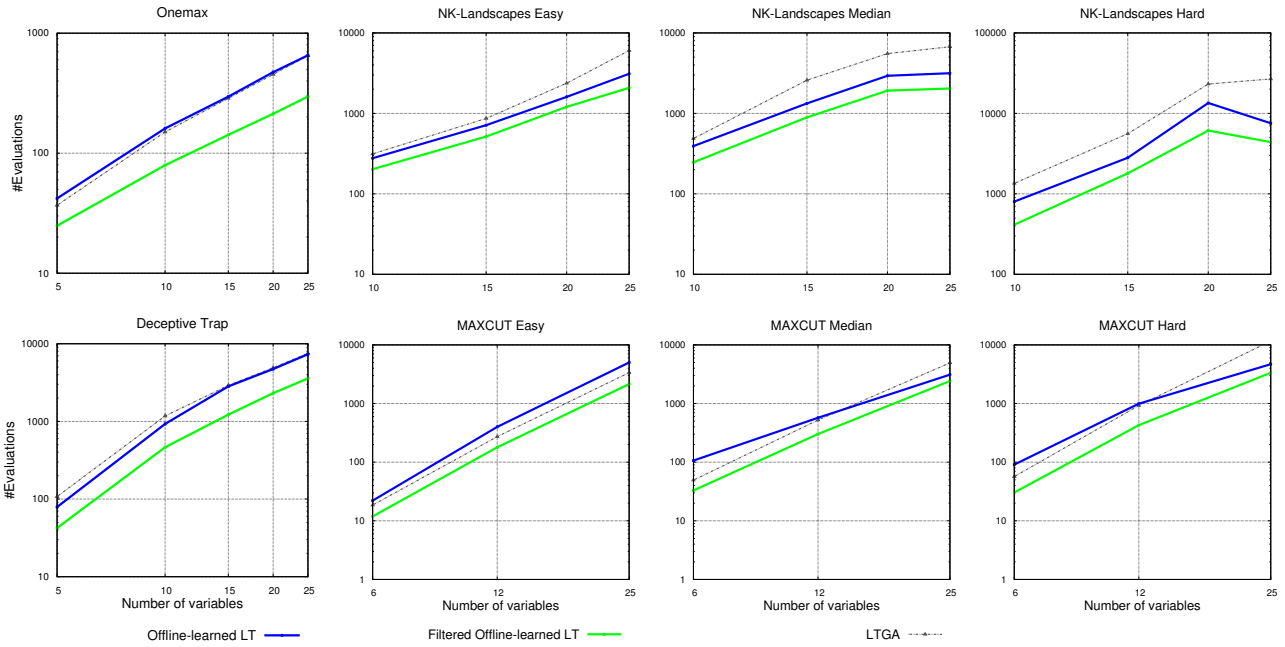


Figure 3: Experimental results of LTGA using (filtered) offline-learned LTs.

Table 1: LT_{off} linkage set average occurrence ratios over 100 LTGA runs for *Deceptive Trap*, $\ell = 10$, including average occurrence ratio. Bold sets were selected during meta-optimization.

Linkage set	g_0	g_1	g_2	g_3	g_4	Avg
{0}	100%	100%	100%	100%	100%	100%
{1}	100%	100%	100%	100%	100%	100%
{2}	100%	100%	100%	100%	100%	100%
{3}	100%	100%	100%	100%	100%	100%
{4}	100%	100%	100%	100%	100%	100%
{5}	100%	100%	100%	100%	100%	100%
{6}	100%	100%	100%	100%	100%	100%
{7}	100%	100%	100%	100%	100%	100%
{8}	100%	100%	100%	100%	100%	100%
{9}	100%	100%	100%	100%	100%	100%
{0, 2}	11%	14%	8%	26%	23%	16%
{6, 8}	6%	11%	9%	22%	18%	13%
{6, 8, 9}	2%	11%	11%	10%	3%	7%
{6, 7, 8, 9}	0%	19%	25%	0%	0%	9%
{5, 6, 7, 8, 9}	0%	77%	98%	99%	100%	75%
{0, 2, 3}	0%	12%	6%	12%	11%	8%
{0, 1, 2, 3}	0%	12%	24%	2%	0%	8%
{0, 1, 2, 3, 4}	0%	82%	99%	99%	100%	76%

building blocks of the problem are found at $\{0, 1, 2, 3, 4\}$, $\{5, 6, 7, 8, 9\}$, ... For $\ell = 10$, results show that indeed the two important linkage sets $\{0, 1, 2, 3, 4\}$ and $\{5, 6, 7, 8, 9\}$, which are both present in the LT_{off}, are also dominantly represented in the LT_{on}s, occurring in 75% and 76% of the LT_{on}s on average respectively, while other sets have an occurrence ratio of 16% or less. For $\ell = 15, 20, 25$, this clear division also occurred: all important linkage sets had an average occurrence ratio of over 60%, while all other linkage sets in the offline LT had an occurrence ratio below 20%.

A question that naturally arises is why these important linkage sets do not have an average occurrence ratio of 100%. First of all, the occurrence ratios of all linkage sets in the offline-learned LT are between 0% and 10% for the first generation, which is rather obvious as in the first generation, an LT is built based on a random population which does not contain any signs of linkage until GOM is performed. Results show however, that as generations progress, LTGA is quickly able to identify the building blocks of the *Deceptive Trap* problem, as the occurrence of the building block linkage sets rapidly increases, showing percentages between 65% and 85% for the second generation and 98% and over for subsequent generations for all problem sizes. These values decrease again for the last two generations, because by

Table 2: LT_{off} linkage set average occurrence ratios over LTGA 100 runs for *NK-Landscapes*, $\ell = 10$, median difficulty problem instance, including average occurrence ratio. Bold sets were selected during meta-optimization.

Linkage set	g_0	g_1	g_2	g_3	g_4	Avg
{0}	100%	100%	100%	100%	100%	100%
{1}	100%	100%	100%	100%	100%	100%
{2}	100%	100%	100%	100%	100%	100%
{3}	100%	100%	100%	100%	100%	100%
{4}	100%	100%	100%	100%	100%	100%
{5}	100%	100%	100%	100%	100%	100%
{6}	100%	100%	100%	100%	100%	100%
{7}	100%	100%	100%	100%	100%	100%
{8}	100%	100%	100%	100%	100%	100%
{9}	100%	100%	100%	100%	100%	100%
{3, 5}	6%	31%	33%	29%	28%	25%
{1, 3, 5}	1%	21%	38%	12%	0%	14%
{7, 8}	10%	10%	14%	16%	0%	10%
{2, 4}	11%	23%	3%	0%	0%	7%
{7, 8, 9}	1%	0%	2%	0%	0%	1%
{2, 4, 7, 8, 9}	0%	0%	0%	0%	0%	0%
{0, 1, 3, 5}	0%	6%	8%	4%	4%	4%
{0, 1, 2, 3, 4, 5, 7, 8, 9}	4%	0%	0%	0%	0%	1%

then the algorithm is mainly converging towards, i.e. copying from, the optimal solution that is already dominantly represented in the population.

Additionally, tests were performed for the *NK-Landscapes* and *MAXCUT* problem, for which the problem structure is more intricate and the optimal linkage structure is unknown. Figure 3 shows for *NK-Landscapes* that iAMaLGaM is able to learn LT_{off}s that impose a better performance for LTGA compared to the use of LT_{on}s, even showing to be more scalable for larger problem instances tested. This difference in performance is reflected by the lack of overlap between the LT_{off}s and LT_{on}s, as generally the sets contained by the LT_{off}s are only poorly represented in the LT_{on}s, as also shown in Table 2. On rare occasions, some sets have an occurrence ratio of around 40%-50%, while generally, occurrence ratios do not exceed 30%.

For *MAXCUT*, tests show that learning an LT_{off} is less fruitful for less complex problems, however as the problem size and complexity increases, iAMaLGaM was able to learn LT_{off}s that support a better performance for LTGA compared to when LT_{on}s are used. Also here the difference is backed by a general mismatch between the LT_{off}s and LT_{on}s.

The construction of LT_{on}s is subject to the exploration scheme used by LTGA and to the properties of Mutual Information. It has been shown that finding an alternative, better approach for these factors is non-trivial [10, 3]. Results presented in this paper show that although LTGA is able to capture important linkages, the resulting LT_{on}s differ significantly from optimal LT_{off}s. Moreover, LTGA performs significantly better when using these LT_{off}s instead of LT_{on}s for more complex problems, even showing to be more scalable. This suggests that these LT_{on}s might not be optimal and a more appropriate exploration scheme or distance quantity might exist.

5. INTRODUCING LOCAL SEARCH

No appropriate linkage model can be learned in the first generation of LTGA as it is based on a randomly generated population. However, in subsequent generations, LTGA is able to capture important Linkage Sets in the LT that are necessary to find the optimal solution. A natural question that arises is whether the costs of this online linkage learning can be reduced.

First, experiments were performed by replacing the LT_{on} in the first generation with only the singleton linkage sets. This however gives a worse performance of LTGA, which can be explained by the observation that an LT learned from a randomly generated population contains, besides the set of all singleton linkage sets, additional linkage sets. These linkage sets might seem inappropriate, however, they cause additional selection pressure which also contributes to unveiling the problem's structure.

Additionally, *single-pass first improvement local search* was implemented. Goldman and Punch presented a hill climbing algorithm that iteratively inverts variables of a solution in a random order if and only if this imposes an improvement of the solution [4]. While in their implementation this is repeated until no further improvement could be achieved, our implementation only performs such a cycle once for each solution in the first, randomly generated, population. Note that as in our experiments no partial evaluation is used, as this is not applicable to all benchmark problems, every bit flip invokes one function evaluation, resulting in a total of $O(n * \ell)$ evaluations. If partial evaluation can be used, this can possibly be reduced to $O(n)$.

Results of experiments with local search are shown in Figure 3, showing that generally the population-size-free scheme using this form of local search only slightly outperforms the conventional LTGA for most problems. For more complex problems, such as *MAXCUT*, local search was found to be a suitable substitute for the learning process in the first generations, significantly reducing the costs of learning the problem's structure. Note that on the total number of evaluations this still implies only marginal improvement, contrary to results presented by Goldman and Punch, suggesting there might be a subtle difference in their implementation of LTGA [4]. The only problem for which outstanding results are found, is for *Onemax*, which is rather obvious, as a hill climbing algorithm as described above is able to find the optimal solution immediately, requiring only a population size of $n = 1$ and making LTGA itself redundant.

6. PRUNING THE LINKAGE TREE

A further attempt at finding the optimal LT is the pruning of the LT. LTGA is already able to outperform other black-box optimization algorithms, such as ECGA and hBOA with the use of an LT[6, 7]. Though what if we take away the constraint of using a tree-structured linkage model? Additional overhead may exist in the LT in the form of superfluous linkage sets that require additional evaluations. In the past, both attempts to use predetermined linkage models and to filter superfluous linkage sets from the LT_{on}s were shown to meet with checkered results [1, 10]. Filtering LT_{off}s has not been tried before. Given that results so far showed that the LT_{off}s already outperform the conventional LTGA, this approach seems most promising as in the worst case, the optimal subset of an LT_{off} is the LT_{off} itself.

6.1 Meta Optimization using LTGA

Searching for an optimal subset of the linkage sets of the LT_{off} is a combinatorial problem of which the solution space can be binary encoded in a straightforward manner. For instance if the LT_{off} would be $\{\{0\}, \{1\}, \{2\}, \{1, 2\}, \{0, 1, 2\}\}$, the solution 10010 would represent the linkage model $\{\{0\}, \{1, 2\}\}$. Such a linkage model can then be evaluated as described in Section 4.2, which means that LTGA can be used to search an optimal linkage model within the space spanned by the LT_{off}. Because for this problem no knowledge is available about the complexity of the problem and the required population size, the population-size-free scheme for LTGA is used. LTGA was halted when two consecutive runs returned the same answer, as for all benchmark problems tested so far it was found that this often means that the returned solution is in fact the optimal solution.

6.2 Parallelization

Finding the optimal subset of linkage sets of LT_{off}s is far from trivial and requires far more evaluations than previously faced problems. Evaluations also require more time, as 1000 instances of the population-size-free scheme for LTGA are executed for every evaluation. Therefore, a parallelized implementation of LTGA was developed that could be run on a multi-core architecture.

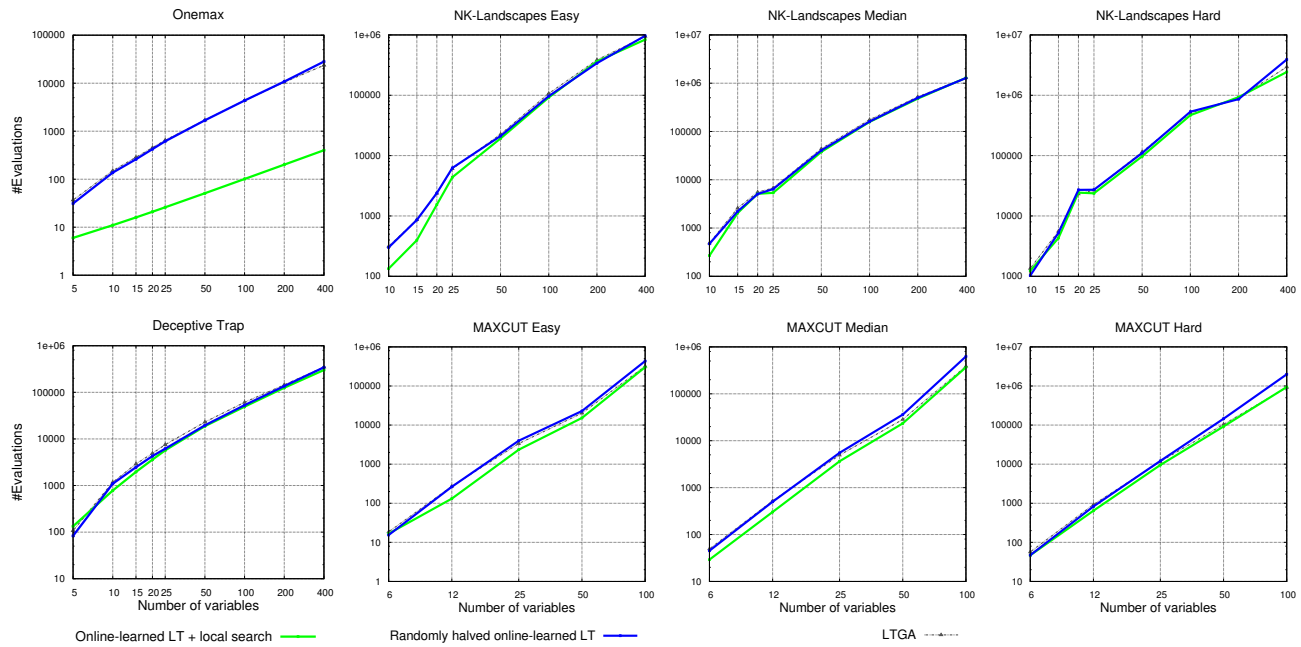


Figure 4: Experimental results of LTGA using online-learned LTs.

Parallelization can be done internally and externally. External parallelization, popularly called *Embarrasingly Parallel*, is done by running single-threaded instances of the program on multiple CPU cores simultaneously, discarding all but the best of the solutions found by these instances. As only one instance has to find the optimal solution, the chance of finding the optimal solution per instance is allowed to be less, requiring a smaller population size per instance, which in turn implies a lower required execution time.

Internal parallelization is done by parallelizing the components of the algorithm that require significant computation time. For LTGA, extensive profiling showed that this concerned the construction of the MIMatrix and the generation of new solutions, the latter mainly being caused by the time required for evaluating possible solutions. Because all cells in the MIMatrix can be calculated independently and all solutions in a population can be improved independently, LTGA was found very suitable for internal parallelization.

Parallelization comes with a certain overhead, for instance caused by a thread scheduler and the access of shared data. Results show, however, that it is almost always profitable to use the internally parallelized implementation over the externally parallelized implementation for the populations-size-free LTGA as the number of available CPU cores increased. Using a 64-core server, we were able to perform the meta optimization as described above.

6.3 Experimental Results

A first analysis of the results of the meta-optimization of LTGA is done by comparing the performance of LTGA using the offline-learned filtered LTs with the performance of the conventional LTGA that uses LT_{on}s. Also these results can be found in Figure 3. It is clear that LTGA, when using the selected subsets of the LT_{off}s as linkage models, shows an even better performance than LTGA when using the full LT_{off}s, further outperforming the conventional LTGA. This enforces our finding that the linkage models used by LTGA might not be optimal and performance improvements can still be gained by using more suitable linkage models. Note of course, that such performance improvement can only be gained if costs of constructing more suitable linkage models are comparable to the current costs of learning LTs online.

In Tables 1 and 2, the linkage sets that were selected by meta-optimization are shown in bold. For *Onemax*, the optimal subset of the LT_{off} consists of all singleton linkage sets but one. This is not exactly as expected, as the complete set of singleton linkage sets represents the structure of this problem best, as discussed before. However, assume that a solution s could be improved using a singleton linkage set k, which in the case of *Onemax* means that the value of the variable in k is set to 0 in s. If a donor d exists for which the value of the variable in k is set to 1, performing GOM with donor d on s will improve s. However, this is equivalent to applying GOM to d with s as a donor with the same linkage model minus k. The only difference is that this linkage model will require one evaluation less on average, meaning that the optimal linkage model for *Onemax* is the set of singleton linkage sets, minus one singleton linkage set. In theory, when using the minimally required population size, which implies that for every variable there is at least one solution that has a value of 1 for this variable, the initial population consists of uniformly distributed solutions, meaning that any singleton linkage set can be chosen as the one to discard, as no linkage set can be preferred of the other. Meta-optimization is therefore able construct an optimal Subset of the LT_{off} by discarding one singleton linkage set from the set of singleton linkage sets.

Comparable results were found for *Deceptive Trap*, for which an example is shown in Table 1. Also for larger problem instances all but one important linkage sets were selected. Also for *NK-Landscapes MAXCUT* results were studied, though no clear correlation nor pattern could be

found for the selected linkage sets, for which the intricate problem structure is the most important reason.

One clear pattern that was found in the results was the fact that in general only half of the linkage sets in the LT_{off}s were selected by the meta-optimization, which is likely due to the overlap between linkage sets contained in the LT. For instance in the example in Table 2, the set {7, 8} is also contained in the sets {7, 8, 9}, {2, 4, 7, 8, 9} and {0, 1, 2, 3, 4, 5, 7, 8, 9}. As no clear correlation could be found about which 50% was removed or maintained, an experiment was performed where during GOM for each solution only half of the LT was traversed, simulating the discarding of 50% of the LT randomly for every solution. Results can be found in Figure 3, showing that, although better than expected, the performance does not consistently exceed the performance of the conventional LTGA. As per solution only 50% of the evaluations are performed while the total number of evaluations is quite similar to the number of evaluations required by the conventional LTGA, these results indicate that when using only a random half of the LT, LTGA will need a larger population size or more generations to find the optimal solution. This shows that indeed a specific underlying scheme is causing half of the LTs to be redundant and a more advanced heuristic is needed to efficiently determine a more suitable linkage model for LTGA, either for fully learning the LT in a different manner of by pruning LT_{on}s.

7. CONCLUSIONS

In this paper, results were presented that were found in search of the optimal Linkage Tree to use in LTGA. To this end we learned predetermined LTs by offline optimizing the performance of LTGA. For all benchmark problems, LT_{off}s could be learned that resulted in linkage models that are better as the complexity of the problem increases, resulting in a better performance of LTGA when they were used as a predetermined linkage model, replacing the LT_{on}s. Analyzing these results, the LT_{on}s used by LTGA were found to contain inefficiencies caused by the presence of superfluous linkage sets in the LT. An internally parallelized version of LTGA was implemented, leveraging computational power of multi-core CPU architectures. Combining this with a population-size-free scheme of the LTGA enabled us to find optimal subsets of these LT_{off}s, which in turn showed a performance increase by up to a factor of 6. These results show that the LT_{on}s used by LTGA might not be optimal and performance improvement might be still be gained when more suitable linkage models can be constructed. It is important to note, however, that the magnitude of this performance improvement is subject to the costs implied by constructing such linkage models, meaning that significant performance improvement can only be achieved when they can be constructed at costs that are comparable to the current costs of learning the LTs online. Future work includes constructing a suitable metric for defining these exact costs of online linkage learning.

In general, about 50% of the linkage sets contained by the LT_{off}s were filtered out, though experiments in which per solution only a random 50% of the linkage sets in the LT were used during GOM showed not to be fruitful. As no clear correlation between the linkage sets in the pruned LT_{off}s could be found for problems other than *Onemax* and *Deceptive Trap*, future work also includes research aimed at gaining more insight into the contents of pruned LT_{off}s in order to find a method for constructing optimal linkage models. This might ultimately enable us to reduce the number of evaluations, resulting in further performance improvement for LTGA.

8. REFERENCES

[1] P. A. Bosman and D. Thierens. More concise and robust linkage learning by filtering and combining linkage hierarchies. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 359–366, New York, NY, USA, 2013. ACM.

[2] P. A. N. Bosman, J. Grahl, and D. Thierens. Benchmarking parameter-free AMaLGaM on functions with and without noise. *Evolutionary Computation*, 21(3):445–469, Sept. 2013.

[3] P. A. N. Bosman and D. Thierens. On measures to build linkage trees in LTGA. In *Parallel Problem Solving from Nature - PPSN XII*, volume 7491, pages 276–285. Springer, 2012.

[4] B. W. Goldman and W. F. Punch. Parameter-less population pyramid. In *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 785–792, New York, NY, USA, 2014. ACM.

[5] I. Gronau and S. Moran. Optimal implementations of UPGMA and other common clustering algorithms. *Inf. Process. Lett.*, 104(6):205–210, Dec. 2007.

[6] G. R. Harik, F. G. Lobo, and K. Sastry. Linkage learning via probabilistic modeling in the extended compact genetic algorithm (ECGA). In *Scalable optimization via probabilistic modeling*, pages 39–61. Springer, 2006.

[7] M. Pelikan and D. Goldberg. Hierarchical bayesian optimization algorithm. In M. Pelikan, K. Sastry, and E. Cantú-Paz, editors, *Scalable Optimization via Probabilistic Modeling*, volume 33 of *Studies in Computational Intelligence*, pages 63–90. Springer, 2006.

[8] D. Thierens. The linkage tree genetic algorithm. In R. Schaefer, C. Cotta, J. Kołodziej, and G. Rudolph, editors, *Parallel Problem Solving from Nature, PPSN XI*, volume 6238 of *Lecture Notes in Computer Science*, pages 264–273. Springer Berlin Heidelberg, 2010.

[9] D. Thierens and P. A. N. Bosman. Evolvability analysis of the linkage tree genetic algorithm. In C. A. C. Coello, V. Cutello, K. Deb, S. Forrest, G. Nicosia, and M. Pavone, editors, *Parallel Problem Solving from Nature - PPSN XII*, volume 7491 of *Lecture Notes in Computer Science*, pages 286–295. Springer Berlin Heidelberg, 2012.

[10] D. Thierens and P. A. N. Bosman. Predetermined versus learned linkage models. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, pages 289–296, New York, NY, USA, 2012. ACM.

[11] S.-M. Wang, Y.-F. Tung, and T.-L. Yu. Investigation on efficiency of optimal mixing on various linkage sets. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 2475–2482, July 2014.

[12] T.-L. Yu, K. Sastry, and D. E. Goldberg. Linkage learning, overlapping building blocks, and systematic strategy for scalable recombination. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1217–1224, New York, NY, USA, 2005. ACM.

In Search of Optimal Linkage Trees

Roy de Bokx
Delft University of Technology
Delft, The Netherlands
Rdebokx1990@gmail.com

Dirk Thierens
Utrecht University
Utrecht, The Netherlands
D.Thierens@uu.nl

Peter A.N. Bosman
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
Peter.Bosman@cwi.nl

Keywords

Evolutionary Computation; Parallel Computation; Genetic Algorithms; Estimation-of-Distribution Algorithms; Linkage Learning; Optimal Mixing; Linkage Tree Genetic Algorithm

CCS Concepts

•Mathematics of computing → Evolutionary algorithms; •Computing methodologies → Model verification and validation;

1. INTRODUCTION

Linkage-learning Evolutionary Algorithms (EAs) use *linkage learning* to construct a *linkage model*, which is exploited to solve problems efficiently by taking into account important linkages, i.e. dependencies between problem variables, during variation. It has been shown that when this linkage model is aligned correctly with the structure of the problem, these EAs are capable of solving problems efficiently by performing variation based on this linkage model [2]. The Linkage Tree Genetic Algorithm (LTGA) uses a Linkage Tree (LT) as a linkage model to identify the problem's structure hierarchically, enabling it to solve various problems very efficiently. Understanding the reasons for LTGA's excellent performance is highly valuable as LTGA is also able to efficiently solve problems for which a tree-like linkage model seems inappropriate. This brings us to ask what in fact makes a linkage model ideal for LTGA to be used.

2. OFFLINE LINKAGE TREE LEARNING

To study the strengths and weaknesses of LTGA, we performed experiments aimed at learning LTs offline to be used as predetermined linkage models for LTGA, that replace the online-learned LTs (LT_{on}s). Contrary to conventional approaches, this is done by searching in the space of LTs and evaluating the associated performance of LTGA.

GECCO '15 July 11-15, 2015, Madrid, Spain

ACM ISBN 978-1-4503-3488-4/15/07.

DOI: http://dx.doi.org/10.1145/2739482.2764679

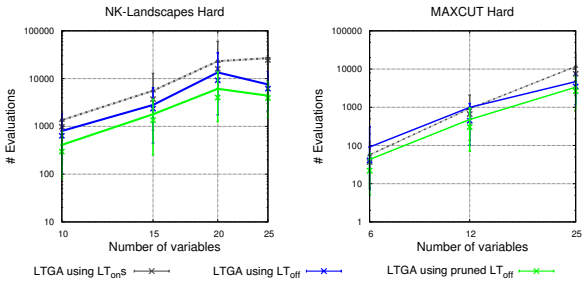


Figure 1: Experimental results of LTGA using (pruned) offline-learned LTs for the hardest *NK-Landscapes* and *MAXCUT* instances.

Using a binary encoding to represent all possible offline-learned LTs (LT_{off}s) is non-trivial and would result in a problem with high dimensionality, of which it is questionable whether it can be solved within reasonable time. Instead, the outcome of the hierarchical clustering algorithm used by LTGA to learn LT_{on}s based on a population can be manipulated by forcing different contents into the distance matrix used, resulting in LTs that represent different linkage contexts. We used the real-valued EA known as iAMaLGaM to this end, as it was found to be very robust in finding suitable contents for a distance matrix [1]. The fitness of solutions, i.e. distance matrices, evaluated by iAMaLGaM was defined as the number of evaluations needed by LTGA with a newly implemented population-size-free scheme when the LT_{on}s of LTGA are replaced by the corresponding offline-learned LT (LT_{off}), averaged over 1000 runs. This population-size-free-scheme repeatedly initiates instances of LTGA, ever doubling the population size until a stopping condition is met.

This resulted in distance matrices that support the creation of optimal LT_{off}s, being LTs that cause the largest reduction in required number of evaluations by LTGA when such an $LT_{off}$is used as a predetermined linkage model, replacing the LT_{on}s of LTGA.

2.1 Experimental Results

We compare the LT_{off}s found by iAMaLGaM with the LT_{on}s learned by the conventional LTGA in terms of LTGA's performance and by the contents of these LTs. Figure 1 shows the performance imposed by LTGA when using either LT_{off}s or LT_{on}s, which corresponds to the fitness values found by iAMaLGaM. Results for *pruned* LT_{off}*s* in Figure 1 can be ignored for now.

For *NK-Landscapes* and *MAXCUT*, experiments show that LTGA using the LT_{off}s substantially outperforms the conventional LTGA, as reflected by Figure 1. Moreover, LTGA using LT_{off}s seems to be slightly better scalable, suggesting that the LT_{off}s learned are intrinsically better than the $LT_{on}$and raising the question why this is the case.

Linkage sets in LT_{off}s are determined based on LTGA's performance when using these LT_{off}s and thus in a sense must contain key linkage structures. To obtain insight into the extent to which LTGA also identifies these structures, we study the overlap between the LT_{off}s and the LT_{on}s used by LTGA over 100 independent runs of LTGA. For *Onemax*, results of these experiments are rather trivial, however for *Deceptive Trap*, these experiments show that after the second generation all important linkage sets are present in the LT_{on} more than 98% of the time. This verifies that LTGA is able to identify important linkages and represent these in the LTs learned for problems with clear linkage structures.

While earlier attempts to construct appropriate predetermined linkage models were not fruitful for more intricate problems such as *NK-Landscapes* and *MAXCUT* [3], these results show that indeed such predetermined linkage models exist. This indicates that the LT_{on}s used by LTGA might not be optimal linkage models as iAMaLGaM is able to learn LT_{off}s that differ significantly from the LT_{on}s while containing important linkage sets and supporting a better performance of LTGA. A clear explanation for this phenomenon was not found after inspecting the differences between LT_{off}s and LT_{on}s.

3. PRUNING THE LINKAGE TREE

Experiments described above resulted in intrinsically better LTs. Though what if we take away the constraint of using a tree-structured linkage model? Additional overhead may exist in the LT in the form of superfluous linkage sets that impose additional evaluations when performing variation. Therefore, the found LT_{off}s were pruned to filter out such linkage sets.

This was done with the use of LTGA itself by encoding subsets of an LT_{off}in a straightforward manner. For instance if the LT_{off} would be $\{\{0\}, \{1\}, \{2\}, \{1, 2\}, \{0, 1, 2\}\}$, the solution 10010 would represent the linkage model $\{\{0\}, \{1, 2\}\}$. As no knowledge was available about the required population size for this problem, the population-size-free scheme was used. Moreover, an internally parallelized implementation of LTGA was used in this scheme that is able to distribute the workload of the construction of the distance matrix and the generation of new solutions over processor cores available in a multi-core architecture.

3.1 Experimental Results

Experimental results show that LTGA, when using the selected subsets of the LT_{off}s as linkage models, exhibits even better performance than LTGA when using the full LT_{off}s, further outperforming the conventional LTGA. The performance of LTGA when using the pruned LT_{off}s is included in Figure 1 for the hardest problem instances of *NK-Landscapes* and *MAXCUT* in a randomly generated test-suite of 100 instances.

An analysis of the contents of the pruned LT_{off}s show that for *Onemax* and *Deceptive Trap*, the pruned LT_{off} contains all but one of the important linkage sets, which is as expected. For *NK-Landscapes* and *MAXCUT*, however, no clear correlation nor pattern could be found among the selected linkage sets. One clear pattern that was found in general, was that only half of the linkage sets in the LT_{off}s were selected, which is likely due to the overlap between linkage sets contained in the LT. Experiments performed in which for each solution only a random half of the LT was used when performing variation, show that, although better than expected, the performance does not consistently exceed the performance of the conventional LTGA. This indicates that indeed a specific underlying scheme is causing half of the LTs to be redundant and a more advanced heuristic is needed to efficiently determine a more suitable linkage model for LTGA, either for fully learning the LT in a different manner of by pruning LT_{on}s at low costs.

4. CONCLUSIONS & FUTURE WORK

The recently introduced Linkage Tree Genetic Algorithm (LTGA) has been shown to exhibit excellent scalability on a variety of optimization problems. LTGA employs Linkage Trees (LTs) to identify and exploit linkage information between problem variables. Much is already understood about LTGA's performance, but it is still unclear whether the LT model can be further improved upon. In this work we analyzed the results of learning LTs offline by optimizing LTGA's performance as a function of static LTs. This resulted in a better performance of LTGA than with online-learned LTs as problem complexity increases. Further analysis of the offline-learned LTs indicated that pruning the LT can result in a further performance improvement of the LTGA up to a factor 6. Using a population-size-free internally parallelized version of LTGA, we found that the optimal subset of the offline-learned LT typically contains only about 50% of the nodes. This suggests that the LT model contains redundancies that may possibly still be exploited to improve the performance of LTGA with online-learned LTs. The magnitude of this performance improvement is subject to the costs implied by constructing improved linkage models, meaning that significant performance improvement can only be achieved when they can be constructed at costs that are comparable to the current costs of learning the LTs online. Future work is aimed at constructing a suitable metric for defining these exact costs of online linkage learning and gaining more insight into the contents of pruned LT_{off}s in order to find a method for constructing linkage models of higher quality. This might ultimately enable us to reduce the number of evaluations, increasing the performance of LTGA and supporting the ability to solve more complex problems.

5. REFERENCES

[1] P. A. N. Bosman. On empirical memory design, faster selection of bayesian factorizations and parameter-free gaussian EDAs. In *Proc. of the 11th Annual Conf. on Genetic and Evolutionary Computation*, GECCO '09, pages 389–396, New York, NY, USA, 2009. ACM.

[2] P. A. N. Bosman and D. Thierens. More concise and robust linkage learning by filtering and combining linkage hierarchies. In *Proc. of the 15th Annual Conf. on Genetic and Evolutionary Computation*, GECCO '13, pages 359–366, New York, USA, 2013. ACM.

[3] D. Thierens and P. A. N. Bosman. Predetermined versus learned linkage models. In *Proc. of the 14th Annual Conf. on Genetic and Evolutionary Computation*, GECCO '12, pages 289–296, New York, USA, 2012. ACM.