

Delft University of Technology
M.Sc. Thesis in Computer Science

A Genetic Approach to Programming Large-Scale Systems

Victor-Lucian Spiridon



A Genetic Approach to Programming Large-Scale Systems

M.Sc. Thesis in Computer Science



Embedded Software Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Victor-Lucian Spiridon
victor.lucian.spiridon@gmail.com

December 2nd, 2013

Author

Victor-Lucian Spiridon (victor.lucian.spiridon@gmail.com)

Title

A Genetic Approach to Programming Large-Scale Systems

M.Sc. Presentation

December 12th, 2013

Graduation Committee

Prof. Dr. K. G. Langendoen (Chair)	Delft University of Technology
A.S. Pruteanu, Ph.D. (Daily Supervisor)	Delft University of Technology
Dr. P.A.N. Bosman (External Member)	Centrum Wiskunde & Informatica
Prof. Dr. E. Visser (Core Member)	Delft University of Technology

Abstract

In recent years, there is an increasing interest in the scientific community for development of algorithms targeting large-scale networks. In addition to their sheer size, these networks also exhibit various topology dynamics – nodes join and exit at high rates (churn), are mobile and are not always reliable. Such extreme properties make centralized algorithms unsuitable in a lot of practical situations.

Our goal is to find algorithms that run continuously and are based on local rules only (i.e., decisions based on information exchanged through 1-hop interactions). If designed correctly, these algorithms are highly scalable and robust. However, identifying the local rules is a very difficult task with no clear discovery methodology.

The heuristics we introduce in this thesis represent a so-called "global-to-local compiler" – designed to search for a mapping between global system requirements and local interaction rules. We test search methods such as Hill-Climbing, Simulated Annealing and Tabu Search. Additionally, we develop custom heuristics such as GG (Generation Gap) Tabu Hill-Climbing, GG Tabu Simulated Annealing and an implementation of GOMEA, named GOMGP. We perform thorough benchmarking in order to find the optimal parameter configuration. For the identified parameter combinations, the heuristics vastly outperform state-of-the-art heuristics.

We discover new algorithms for the problems: "distributed maximal independent set formation" and "event localization using graph spectral theory" with good results and insights. Finally, we improve existing algorithms for the "node failure rate estimation" (FailDetect) and "churn rate estimation" (ChurnDetect). The improvements consist of lower estimation variance at network nodes.

Keywords: Genetic Programming, Large-Scale Systems, Distributed Algorithms, Algorithm Discovery, Scalability, Wireless Sensor Networks, Optimization Heuristics

“Think like a man of action, act like a man of thought.” – Henri Bergson

Preface

A good *friend* of mine, in the company of whom I have spent many evenings, wrote: “*It’s a dangerous business going out your door. You step onto the road, and if you don’t keep your feet, there’s no knowing where you might be swept off to.*” The same playfulness and curiosity followed me during the two years of scientific adventure, here in Delft with the Embedded Software Group. And it is this curiosity that brought me to the topic of designing algorithms for Large-Scale Systems by using Genetic Programming techniques. Bridging the gap between these two fields represents, simply, the greatest challenge I have ever faced. What seemed at first as a weakness became my greatest strength as being armed with a fresh eye helped me to identify things never tried before and thus create the scientific contribution represented by this thesis. And here I am, at the end of the road, with my eyes fixed on the horizon, ready to seek new *adventures*, knowing one thing for certain: the greater the struggle, the greater the reward.

Firstly, I would like to thank my brothers Dani, Costi and Cătă who represented a great support both morally and financially. A great deal of appreciation goes to Andrei Pruteanu, Ștefan Dulman, Peter Bosman and Andreas Loukas for guiding me during the last 10 months, and showing me what it is to be a rigorous researcher. Many thanks go to Koen Langendoen for hosting this thesis, and giving me essential comments and feedback for making this work relevant for the Embedded Software Group. The *Experiments* section would have been much shorter if not for the help of Emil Slușanschi who granted me access to the HPC cluster at Politehnica University of Bucharest, and to whom I am so much obliged. I would also like to acknowledge the financial support through scholarships offered by the *Dinu Patriciu Foundation* and the *Romanian-American Foundation*. Finally, I am so grateful to my girlfriend, Roxana, for her constant love and support even from far, far away. This thesis is dedicated to her.

Victor-Lucian Spiridon

Delft, The Netherlands
December 2nd, 2013

Contents

Preface	vii
1 Introduction	1
1.1 Problem Statement	2
1.2 Methodology	2
1.3 Thesis Outline	3
2 Background and Related Work	4
2.1 Genetic Programming	4
2.1.1 Performance Criteria in Genetic Programming	5
2.1.2 Linear Program Representations in Genetic Programming	6
2.1.3 Steady-State Genetic Programming	7
2.1.4 Competent Genetic Algorithms and Linkage Learning via Estimation of Distribution	7
2.1.5 Genetic Programming Results	8
2.2 Simulated Annealing	9
2.3 Classic Distributed Algorithms	10
2.4 History of the MetaCompiler Project	12
3 Genetic Heuristics	13
3.1 Problem Definition	13
3.2 Classic Genetic Programming	14
3.2.1 Program Representation and Generation	14
3.2.2 Fitness Function	14
3.2.3 Genetic Operators and Selection	15
3.3 Conditional Whole-Population Heuristics	16
3.3.1 Hill Climber	16
3.3.2 Simulated Annealing	16
3.3.3 Tabu Search	17
3.3.4 GOMGP – GP with GOMEA Inspired Generations	17
3.4 Conditional Generation Gap Heuristics	19
3.4.1 GATLE – Generation Gap Tabu Hill Climber	19

3.4.2	GATSE – Generation Gap Tabu Simulated Annealing	19
4	MetaCompiler Design	21
4.1	Agent-Based Simulator – NetLogo	21
4.1.1	Code Injection	22
4.1.2	Concurrency of Execution	23
4.2	MetaCompiler	24
4.2.1	Execution Flow	24
4.2.2	Agent Program Skeleton and Building Blocks	26
4.2.3	Discovering Algorithmic Building Blocks	27
5	Benchmarking of the Genetic Heuristics	29
5.1	The Test Problems	29
5.1.1	Artificial Ant on the Santa Fe Trail	30
5.1.2	Symbolic Regression	30
5.1.3	Synthesis of an 11-bit Multiplexer	31
5.2	Benchmarking Experiments	31
5.2.1	Genetic Operations in HC and SA Heuristics	32
5.2.2	Generation Gap in GATLE and GATSE	33
5.2.3	Genetic Operations and Acceptance Probability in GATLE and GATSE	34
5.2.4	Tree Linearization and Node Selection Strategy in GOMGP	36
5.2.5	Genetic Operations and Individual Improvement Conditions in GOMGP	36
5.2.6	Tournament Size in GOMGP	37
5.2.7	Comparison of the Best Parameter Combinations	38
5.2.8	Comparison with State-of-the-Art Heuristics	42
5.3	Discussion	43
5.3.1	Parameters Common to All Heuristics	43
5.3.2	Parameters in GATLE and GATSE	43
5.3.3	Advantages of GATLE and GATSE	44
5.3.4	Tree Linearization and Node Selection Strategies in GOMGP	44
5.3.5	GOMGP Halting Properties	44
5.3.6	Advantages of GOMGP	46
5.4	Choosing the Best-Suited Heuristic	46
6	Algorithm Discovery Case Studies	47
6.1	Maximal Independent Set Formation	48
6.1.1	Problem Description	48
6.1.2	Experimental Setup	48
6.1.3	Fitness Function	50
6.1.4	Found Solutions	50

6.1.5	Discussion	53
6.2	Event Localization Using Graph Spectral Theory	54
6.2.1	Problem Description	54
6.2.2	Experimental Setup	56
6.2.3	Fitness Function	58
6.2.4	Found Solutions	59
6.2.5	Discussion	62
6.3	FailDetect Algorithm	63
6.3.1	Problem and Algorithm Description	63
6.3.2	Experimental Setup	63
6.3.3	Fitness Function	64
6.3.4	Found Programs	65
6.3.5	Discussion	66
6.4	ChurnDetect Algorithm	67
6.4.1	Problem and Algorithm Description	67
6.4.2	Experimental Setup	67
6.4.3	Fitness Function	68
6.4.4	Found Programs	69
6.4.5	Discussion	70
7	Discussion	71
7.1	Using Problem Knowledge	71
7.2	Fitness Function Design – Avoiding Trivial Solutions	72
7.3	Effect of the Generic Building Blocks	73
7.3.1	Hiding the Intermediary States of an Algorithm	73
7.4	Choosing the Right Set of Building Blocks	73
7.4.1	Examples of Choosing Building Blocks	76
8	Conclusion and Future Work	77
8.1	Conclusion	77
8.2	Future Work	78
	Bibliography	81
	A Pseudocode	89
	B Sample Configuration File	95

Chapter 1

Introduction

Soon everything around us will be interconnected via massively large-scale networks. With size there also comes the problem of efficiently managing such extreme systems. Algorithms tailored for them are difficult to invent due to the distributed nature of the network. Our goal is to tackle this challenging problem with genetic programming-based techniques, in order to automate the discovery of distributed algorithms.

A large-scale system is a distributed collection of devices (nodes) that is perceived as a single entity by its users. Nodes are characterized by *autonomy* (take decisions independently, without central control), *cooperation* (work together toward a common goal) and *interaction* (exchange information among each other) [21]. In practice, devices that form such systems have limited capabilities (e.g., the links between nodes suffer from communication delays and very often there is packet loss; computing power is limited due to power and space constraints). At the same time, the scale of the systems can vary from tens to thousands of (heterogeneous) devices and are characterized by dynamic topologies, meaning that nodes are either continuously joining and leaving the network or are simply mobile. Examples of such large-scale distributed systems are wireless sensor networks, vehicular networks, mobile ad-hoc networks, electrical power grids, social networks, etc.

The properties of large-scale distributed systems make centralized algorithms unsuited for many practical reasons such as limited scalability, reduced responsiveness, high communication complexity, etc. Moreover, for dynamic systems, runtime adaptation is of major importance. Approaching the design of algorithms with a bottom-up approach is a tedious effort and does not guarantee any level of success. Out of simple, local interactions it is very difficult to predict complex, system-wide (global) behaviour. A well-known characteristic of large-scale systems is that above a certain scale, certain phenomena occur somewhat unexpectedly out of these very simple local rules. In other words, simple rules can create complex behaviour. In

a very small number of cases this phenomenon, also called *emergent behavior*, is a positive property and can be put to good use by, for instance, synchronization algorithms [94], clustering schemes [78], distributed feedback mechanisms [9], etc. There are cases though where it has a negative effect [67].

1.1 Problem Statement

Our approach to overcome the challenge of inventing distributed algorithms is to use Genetic Programming (GP) for realizing a global-to-local compiler [4]. GP is a type of evolutionary algorithm, inspired from biology, which aims at developing computer programs (solutions) that solve user-defined tasks. The quality of a program is evaluated by a fitness function that assigns a score describing how “fit” a program is to solve the given task. The heuristics showcased in this thesis are designed to automatically search for local rules that generate complex behaviour at global level.

The first research objective is to create a flexible and efficient software framework. For this, it is necessary to:

Objective 1: Have a domain-specific agent behaviour representation format that is able to express solutions to our search in terms of computer programs. It also has to be able to characterize solutions in terms of various computational and communication costs.

Objective 2: Have a fast and efficient heuristic with respect to execution time and computational resources. Due to the fact that classic genetic programming heuristics do not perform well enough for this set of problems [93], we have to come up with novel genetic search methods.

Provided with the GP-based search tools, a second set of objectives are to:

Objective 3: Discover new algorithms for distributed large-scale systems

Objective 4: Improve existing algorithms for distributed large-scale systems

1.2 Methodology

We call our GP software platform “MetaCompiler”. Candidate solutions are represented by computational building blocks such as functions, procedures, and variables. Ideally, we aim to have specific building blocks that create a small search space. However, this makes it hard to find solutions since the set of building blocks can be insufficient to fully describe a solution. The certainty that a set of specific building blocks is sufficient comes with prior knowledge about the solution, which is not always available. In order to meet Objective 1, we extend the generative capabilities of the MetaCompiler to

include more generic building blocks. These enable us to search for more complex algorithms with less domain knowledge.

Given a fitness function that defines the problem and a set of building blocks, the MetaCompiler uses an optimization heuristic to find the best fitness score. At the core of our approach is the classic Genetic Programming (CGP). However, in order to achieve Objective 2, we added and tested many others:

- Four heuristics inspired from literature (see Section 3.3):
 - *Hill-Climbing* [58] (HC)
 - *Simulated Annealing* [13] (SA)
 - *Tabu Search* [26] (TS)
 - A partial implementation of GOMEA [7], which is, to our knowledge, applied for the first time to GP. We named it GOMGP.
- Two heuristics developed by us (see Section 3.4):
 - *Generation gAp Tabu hilL climbEr* (GATLE)
 - *Generation gAp Tabu Simulated annEaling* (GATSE)

To accomplish Objective 3, we experimented with discovering algorithms for the following problems:

- Given a set of dependency relationships between nodes in a graph, identify a maximal independent set of nodes in a distributed manner
- Event localization with graph spectral theory in a sensor network

Finally, to attain Objective 4, we experimented with improving the following algorithms from literature:

- FailDetect [80] (estimation of the communication fail rate in a network)
- ChurnDetect [79] (estimation of the churn rate in a network)

1.3 Thesis Outline

The thesis is organized as follows. We first describe related work in Chapter 2. We then introduce the heuristics and the MetaCompiler with an emphasis on our contributions in chapters 3 and 4. These are followed by a first set of experiments consisting of benchmarking the heuristics in Chapter 5. A second set of experiments representing case studies of improving and discovering algorithms with the MetaCompiler is showcased in Chapter 6. We summarize the insights gathered with this work in Chapter 7. Finally, we conclude and state proposals for future work in Chapter 8.

Chapter 2

Background and Related Work

In this chapter, we give background information related to the major topic of this thesis: Genetic Programming. We also present other closely related optimization techniques that served as inspiration for the development of the set of heuristics in this thesis. Additionally, we briefly analyse a set of well-known classic distributed algorithms, to identify how suited are they for large-scale dynamic networks. Finally, we present the history of the MetaCompiler project and the starting point of the research in this thesis.

2.1 Genetic Programming

The original concepts of Genetic Algorithms (GAs) were developed by J. Holland [34, 35] and further refined by J.R. Koza [42–46]. Coming from these concepts, the aim of Genetic Programming (GP) is to automatically generate and evolve computer programs. A set of programs is called a population. First, a formal grammar has to be defined in order to generate the programs being searched for. It is a generative grammar for randomly creating the initial population of programs (e.g. 100 individuals), all representing candidates to the solution. Each candidate is evaluated using a user-defined fitness function. From that initial population, the genetic search process is divided in generations. In each generation, the old set of candidates (the parents), has to be replaced by a new set of candidates (the children). This is accomplished by performing genetic operations (e.g., Crossover and Mutation) on the parents, thus creating new candidates. Each new candidate is evaluated and when a good enough solution is found (which is up to the user to define), the process stops. Otherwise, it continues until a specified number of generations is reached or a computational budget is consumed. Genetic Programming is a heuristic that traverses the solution space given by the formal grammar. The goal is to find a program that minimizes the

fitness function. Therefore, the search problem maps onto an optimization problem.

2.1.1 Performance Criteria in Genetic Programming

The performance of a GP-based search framework can be assessed by several criteria: the size of the individual programs in a population (parsimony), population diversity and, the fitness value.

The first criterion is related to the problem of code bloat: an increase in the size of programs in a population, not accompanied by an improvement in fitness. Thus, the growth does not contribute to an iterative construction of the solution. There are several explanations for bloat. McPhee and Miller [65] argue that larger individuals have a higher chance of producing offspring with the same fitness. This is because larger individuals have a higher chance to form semantically equivalent children via the Crossover operator. With the *removal bias theory*, Soule and Foster [89] categorize the code into active and inactive code. They have found that the inactive code usually resides deeper in the trees, thus containing small subtrees. Then, with Crossover, there is a high chance that the inactive code receives a larger subtree (than the one donated), thus enlarging the inactive code's size, but still keeping the fitness of the parent. These two theories explain why and how code bloat happens. Langdon and Poli [48–51] argue, more generally, that variable length representations are another cause of bloat, because there is an unlimited number of ways of expressing the same solution candidate (due to inactive / identity code). Poli proposes the Tarpeian method [75], in which large fitness values are assigned to large individuals. Another basic method was proposed by Koza [43] in which individuals above a certain code size, are simply rejected. Other methods have “parsimony pressure” [59, 61, 76, 77], which introduces parametric penalizations (depending on code size) to the fitness. More generally, there are methods that use the reduction of code size as a separate objective, and then use multi-objective techniques such as Pareto fronts [5, 15, 16, 19] to promote shorter solutions. Luke and Panait [60] found that combining methods, such as the ones presented above, can yield even better results.

Improving population diversity can also be tackled as a separate objective. However, the effectiveness of a given method largely depends on the way diversity is measured. This task is made harder by the active and inactive code separation. Several surveys have been written that study various types of measurements [10–12, 20]. Population diversity is important, because the lack of it can lead to suboptimal solutions. Diversity can be lost when too much selective pressure is applied, or in other words, when only the fittest individuals get to reproduce into the following generations.

Parsimony and diversity are important for the success of a search. With too much bloat, the individuals can grow so large that they consume all

the resources, thus preventing the continuous improvement of the solutions. Without diversity, the search process can get stuck into local optima. However, finding a balance between these objectives is not trivial and most of the times it is problem dependant [60].

2.1.2 Linear Program Representations in Genetic Programming

When the MetaCompiler was designed, a linear representation named Grammatical Evolution (GE) [72] was argued to be the best option for program representation [92]. Given a linear representation, programs are encoded as arrays (chromosomes) of numbers (genes) or as strings, even if semantically it has a tree structure when it gets executed. Because arrays in GE can have variable lengths, GE is considered a variable length representation. The generative grammar is given in Backus-Naur form. Genes are used to pick productions from the set of derivation rules in order to replace non-terminals. If there are insufficient genes, the array is reused from the start (wrapping). One disadvantage of GE is that, with an unlucky combination of numbers, the non-terminal replacement process can never end.

In Gene Expression Programming (GEP) [22] the array is split into a head (terminals and non-terminals) and a tail (only terminals). The split ensures that all the non-terminals are replaced. The head length h is chosen by the user and the tail length is computed as the maximum number of terminals that can be needed considering h and the grammar. On one hand, GEP also generates syntactically correct programs, like GE. On the other hand, because it uses fixed length arrays, it makes parameter tuning difficult — most short solutions cannot be represented by long chromosomes and vice versa.

The next class is making representations using strings. Linear Genetic Programming (LGP) [8] encodes programs as sequences of instructions of an imperative language (usually C). LGP uses registers for input, output and intermediary results. Choosing an improper number of registers can lead to poor performance. However, evolving programs in a low-level language means that programs can be executed very fast (compared to using an interpreter). Cartesian Genetic Programming (CGP) [66] encodes programs as chained modules (e.g., a function with 2 inputs and 1 output) forming a graph. The graph is more general than the tree from standard GP, thus able to evolve more complex programs with more than just one output. However, choosing an improper shape of the graph (the number of rows and columns) can lead to poor results.

2.1.3 Steady-State Genetic Programming

In generational GP, program generations do not overlap, because in each generation the population gets replaced by a new population. In contrast, in steady-state GP, the generations of programs overlap, because in each generation only a percentage (the least fit individuals) of the population gets replaced by new individuals. This population percentage is called a *generation gap*. Initially, Ryan et al. designed GE with a generational mechanism [83] but they have later shown in [84] that a steady-state GE can find solutions faster. This is due to the higher selective pressure applied when children compete directly with their parents and also to the survival of the fittest in future generations.

However, it was shown in [17, 18, 81, 90], by studying the generation gap size, that a steady-state genetic algorithm can lose population diversity at twice the rate of the generational variant, which can lead to a local optimum trap. To overcome this, the basic solution is to use a larger population size. A different solution, proposed by Ghosh et al. [25], is to track the age of every individual and penalize the fitness of the older ones. In [56], Lozano et al. use the contribution to the population diversity of an individual as a secondary objective of the selection process.

2.1.4 Competent Genetic Algorithms and Linkage Learning via Estimation of Distribution

Competent Genetic Algorithms are GAs that are able to solve hard problems [29]. In GAs problems can be represented by variables and their values (genes). Thus, the solving of hard problems is done by explicitly learning correlations between good combinations of genes that improve the fitness. This is called linkage learning.

An example of a simple problem is to maximize the 1s out of a fixed size string of 1s and 0s e.g.: $F_1(0101)=2$ with 4 the ideal fitness value¹. For a GA, it is easy to discover that the more 1s the better the fitness. A hard problem is represented by a deceptive function F_2 identical to F_1 with one exception: $F_2(0000)=5$. Finding the 0000 solution is much harder because the search converges to 1111. A partially deceptive function F_3 accepting longer strings as input, can be the sum of applying F_2 on consecutive groups of 4 digits [32]. These problems are called additively decomposable problems because each gene or group of genes, by itself, improves the fitness.

One important branch of competent genetic algorithms is based on representing linkages through a distribution of probability giving the likelihood of the existence of a gene at a certain position within the individual. The distribution is learned from the genes of the fittest individuals. It is then sampled in order to generate a part of the next population. It was found that

¹In this example, a higher fitness is better.

this method scales better with problem size, compared to evolving populations (classic GA). Some early algorithms are cGA [31] and PBIL [3], which are order-1 algorithms, meaning that they can learn probabilities only for single genes. Order-n algorithms such as eCGA [32] form marginal product models that can describe the probabilities of partitions of genes, thus forming linkages. GOMEA [7] creates hierarchical clusters of linked genes using the mutual information similarity measure and then uses these clusters as parts for the Crossover operator. Crossover is applied multiple times using various sized clusters until an improvement is made to the individual (described in detail in Section 3.3.4). GOMEA halts when improvements are no longer made.

The idea of linkage learning by estimating a distribution of probability had also been applied to GP, although in smaller extent than to GA. One classic example is PIPE [85] in which the probabilities are encoded in a variable size Probabilistic Prototype Tree (PPT). However, PIPE cannot encode multivariate dependencies. ECGP [87] is based on the marginal product model of eCGA which is also encoded in a PPT similar to the one in PIPE. ECGP can model dependencies between multiple graph nodes, located anywhere in the tree. On the downside, the template tree is of fixed size (a wrong tree size can prevent the finding of the solution). In GT-EDA² [6], Bosman and Jong encode the probability distribution for the space of trees through grammar transformations i.e., the set of grammar productions is modified and extended such that a bias is introduced toward better solutions. However, the distribution learning time grows fast with the number of non-terminals. Moreover, the usage of multiple data-types complicates the encoding of probabilities in GT-EDA, and can make the PPT generate invalid programs. Finally, all these methods are best suited for additively decomposable problems, a property which is yet to be proven for the problems we are tackling.

Considering the non-convincing results of PIPE, eCGP and GT-EDA compared to classical GP, and their drawbacks when a complex grammars is used, we choose not to proceed further in our search with this type of genetic heuristics. However, the greedy Crossover and the halting criterion of GOMEA are used as inspiration for our GOMGP heuristic, presented in Section 3.3.4.

2.1.5 Genetic Programming Results

Genetic Programming has been used successfully for a large series of problems, ranging from electronic circuit design, robot control algorithms³, prediction and classification, image processing, economy, medicine [69] and even music composition [36, 39].

²Our abbreviation.

³<http://www.genetic-programming.com/humancompetitive.html>

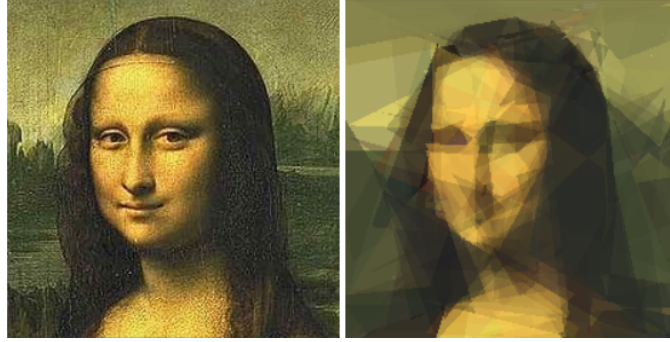


Figure 2.1: Mona Lisa approximated by 50 semi-transparent polygons [58].

One remarkable result was obtained by Winstein and Balakrishnan [96] who developed, using GP, a version of the TCP protocol that outperforms the standard TCP implementations in simulations. This is achieved by developing customized congestion control mechanisms for each network node. The input consists of prior knowledge or assumptions about the network and an objective e.g., high throughput, low latency. The algorithms are developed offline and it generally takes a few hours for them to be discovered.

Another interesting result with respect to software engineering is the use of GP for automated software repair [24, 52, 53]. This is done by providing a set of test cases that should succeed and the one test that exposes the bug in the program. Then by tracing the program’s execution, changes are made to the code, which is then re-evaluated using the tests provided.

One final result, by a genetic heuristic similar to the Hill-Climber presented in Section 3.3.1, is a cropped part of the Mona Lisa painting (see Fig. 2.1) approximated by 50 semi-transparent polygons.

2.2 Simulated Annealing

Simulated Annealing (SA) [13, 41] differs from classical GAs in one fundamental way: new individuals have to fulfill a condition in order to be accepted. If the fitness is better than its parent’s, then the individual is accepted. If it is worse, then the acceptance is given by a probability depending on the “annealing temperature”. The evolution of the annealing temperature over time is called a “temperature schedule”. The “temperature” is high at the beginning of the search, making worse individuals easily acceptable, and as the temperature lowers, worse individuals have smaller chances of being accepted. Another difference is that an SA search usually has a population of 1 individual. Across the literature, there are many proofs of the convergence of SA (e.g., in [30]) and most of them emphasize on the need of a very slowly decreasing temperature. However, the proofs largely depend on the

criterion based on which the new individuals are selected and on the nature of the problem’s search space.

SA and GAs have been compared in papers such as [38, 47, 63] and the performance of one over the other is largely problem dependant. Attempts of combining the two heuristics, such as the ones in [14, 23, 86, 91, 97] managed to combine the population and recombinative power of GAs with the local search and the ability to avoid local optima of SA, thus obtaining better results over GA and SA taken individually.

2.3 Classic Distributed Algorithms

In order to better understand the shortcomings of classic distributed algorithms in large-scale dynamic networks (LSDNs), we have studied some of the basic ones from literature. The findings are summarized in Table 2.1. For each algorithm, we state its goals and the most important prerequisites. We also state the network properties and events to which an algorithm is robust / vulnerable.

One of the main challenges brought by LSDNs is the fact that the network structure changes, therefore classic distributed algorithms have to be restarted. The restart is not always feasible, therefore, in LSDNs, algorithms that run continuously are needed. While studying the algorithms, we also thought about whether they are suitable, *off-the-shelf*, for LSDNs, and, if not, whether we can adapt them or find a replacement algorithm. These findings are summarized in the last column of Table 2.1.

In case of “Termination Detection” and “Dining Philosophers” (resource allocation) the algorithms need little changes. The “Gamma Synchronizer” can be replaced with “Firefly Synchronization” [94]; “Leader Election” can be implemented by broadcasting (using gossiping) the maximum node ID; and “BFS” can be implemented by broadcasting (using gossiping) the route (to the initiator node) having the best score, with the route score aging over time. The original “Maximal Independent Set” (MIS) algorithm was not suitable for LSDNs. Therefore, we used this problem as an algorithm discovery case study. As a result we discovered a decentralized algorithm that tells each node of the network if it is part of the MIS or not.

We implemented the replacement algorithms in the NetLogo agent-based simulator. We did this by organizing the algorithms into simple algorithmic building blocks mapped onto specific tasks, e.g., communication, computation and decision taking. Some of the building blocks could be reused for different algorithms, while others were specific to a single algorithm. This exercise was beneficial for the organization of the set of building blocks for our algorithm discovery case studies (in Chapter 6). Based on this exercise and also on the case studies, we were able to define a methodology for algorithmic building block set definition, which is presented in Section 7.4.

Name	Goals	Prerequisites	Robust to	Vulnerable to	Suitable for LSDN / Replacement
Gamma Syn- chronizer [2]	Nodes agree to a common clock	Neighbourhood ⁴ knowledge	Asynchro- nicity	Packet loss	No. Firefly synchronization [94]
Leader Election [1]	Elect a leader, everyone acknow- ledges the leader	Knowledge of the IDs of all nodes	Unsynced start	Node failures, Packet loss	No. Maximum node ID gossiping
BFS [62]	Everyone knows the path toward an <i>initiator</i> node	Neighbourhood knowledge	Comm. delays, packet loss	Node failure	No. Gossiping of the best route, with route score aging over time.
Termination Detection [37, 64]	Detect the ending of a distributed al- gorithm	Synchronization, leader elected	Comm. delays	Node failures, Packet loss	Yes. Little changes
Dining Philosophers [54]	Non-blocking, fair & efficient resource allocation	Synchronization, random number generator	Node het- erogeneity	Failure of the allocated resource	Yes. Little changes
Maximal Independent Set [57]	Obtain one MIS of the network graph	Synchronization, random number generator	Node failure	Packet loss	No. MetaCompiler case study

Table 2.1: A Selection of Classic Algorithms for Distributed Systems.

⁴The neighbors are the set of nodes one node can connect to.

2.4 History of the MetaCompiler Project

This thesis represents a continuation of the work started by S. van Berkel in his master thesis project [92]. It is part of a bigger project named Snowdrop⁵, which aims at developing tools for creating and making use of large-scale networks. The main part of the project is represented by the MetaCompiler, a tool for automated discovery of algorithms for large-scale networks using Genetic Programming (GP). The heuristics are implemented using the EpochX framework [74], which is further extended with several components representing the main contributions of S. van Berkel’s thesis:

1. Among the three main program representations of GP implemented in EpochX, namely: Grammatical Evolution GP [72], Context-free Grammar GP [95] and Strongly Typed GP [68], a hybrid between the first one (for population initialization) and the second one (for genetic operator application) is found to be performing the best.
2. The population initialization process is improved, in order to create more varied programs in length, by using a fitness function that scores the length of the program, in the first few generations. In some cases, a population of longer (and more complex) programs is shown to find complex solutions faster.
3. Two new genetic operators are added to the framework: “LHS Crossover” and “LHS-like Mutation” inspired from [33]. These operators can preserve a program’s structure better than the operators applied directly on the bit sequence specific to Grammatical Evolution GP, as shown in [82]. This is achieved by parsing each program into a tree, using the grammar and the bit sequence, and then applying the operators on the tree.
4. A pattern detection scheme is proposed to protect the subtrees common to the fittest programs of the population. The scheme identifies pairs of child-parent tree nodes from the program’s tree, which appear often in the fittest programs and reinforces them such that they are less likely to get split by the operators.

S. van Berkel’s work shows promising initial results by rediscovering simple existing algorithms. In this thesis, we extend the prior work in the following ways: we implement multiple genetic heuristics; we perform a thorough benchmarking using standard GP problems; we extend the generative capabilities of the MetaCompiler; we discover two new algorithms and we improve two existing algorithms for large-scale networks.

⁵<http://code.google.com/p/snowdrop>

Chapter 3

Genetic Heuristics

In this chapter we describe in detail the various GP-based heuristics that we implemented for the MetaCompiler. Genetic heuristics represent the core of the MetaCompiler, because they perform the program discovery process. We are searching for the best suited heuristic for our algorithm search i.e., algorithms for large-scale networks. Therefore, we implemented several heuristics that will be further evaluated and compared in a separate benchmarking chapter.

We start by defining the general problem that a Genetic heuristic solves, then briefly present its major components: the program representation and generation, the fitness function and the genetic operators. Afterwards, we present four heuristics inspired from the state-of-the-art related work that we implemented (Hill Climber, Simulated Annealing, Tabu Search and GOMGP) and two heuristics developed by us (GATLE and GATSE). The full pseudocode of heuristics is detailed in [Appendix A](#).

3.1 Problem Definition

Given a subset $\mathcal{P}$, of programs p , from the infinite set of programs defined by a context-free grammar that contains:

- non-terminal symbols: branching statements, variable assignments, functions and procedures that accept parameters;
- terminal symbols: integer and boolean constants, variables, functions and procedures that do not accept parameters;

and given a function $F(p)$ that maps such a program to a fitness value $f_p \geq 0$, then the problem is to find a p for which f_p is either 0 or as small as possible (i.e., to minimize f_p).

3.2 Classic Genetic Programming

In the classic GP, generations of programs are generated and evaluated. We keep the organization in generations for all of the other heuristics developed. After the initial population, new programs are generated by applying genetic operators (i.e., Crossover and Mutation) on existing programs, from the current generation. Programs compete with each other through their fitness values in order to be selected as input to the operators. Thus, programs with a low fitness have a higher chance of contributing to future generations. The process ends when a perfect algorithm (with respect to the fitness function) is found or the maximum number of generations is reached, or the computational budget is consumed. The pseudocode for classic GP is listed in Appendix A.1.

3.2.1 Program Representation and Generation

Program representation is very important for the effectiveness of genetic programming. The representation ensures that genetic operators applied on the candidates produce syntactically correct programs. In our GP implementation, the grammar is designed in such a way that it always generates syntactically correct programs for NetLogo (and possible other target languages) with correctly used data types. Currently, the only data types supported are integers, floats, booleans and strings.

In the MetaCompiler, the programs are represented using the Grammatical Evolution method [72]. Each program is encoded as an array of numbers (codons) that are used to interpret the grammar in Backus-Naur form, by selecting production rules in order to replace the non-terminals with terminal symbols. The traversal is done in a depth-first manner. When the program consists of only terminal symbols, the translation is completed and an executable¹ candidate is obtained. In this way, programs can be randomly created by generating arrays of random numbers. Grammatical Evolution and its trade-offs are discussed at length in [70–73, 83, 92].

3.2.2 Fitness Function

The fitness function is a user-defined function that evaluates a program and assigns a score (fitness) to it. The better the outcome the lower the score, thus creating a minimization problem. The perfect outcome receives a score of zero, meaning that the solution was found. Special care has to be taken when designing a fitness function, such that it penalizes trivial solutions that do not define good algorithms. This can also be overcome by setting up the simulation environment (on which the programs are tested) using

¹An executable program has a correct syntax and correctly used names of functions, procedures, and variables and correctly used data types.

random initial conditions from simulation to simulation, in order to achieve different results. In this way, the ideal program is not a *value*, but a *behavior*. However, this can still be insufficient due to the fact that a program can get *lucky* in finding the result for that specific set of initial conditions. Therefore, we run the simulation for a program several times (e.g., 10) and aggregate the set of scores using one of the functions: **average**, **median**, **minimum**, **maximum**, thus checking the fitness of the program for consistency.

3.2.3 Genetic Operators and Selection

With genetic operators, one or two parents can spawn one, respectively two children. The basic operators are:

- *Mutation* – replaces a part of the parent (the subtree of the selected node) with a randomly generated subtree, having the same non-terminal symbol as root as the original subtree.
- *Crossover* – creates two children by selecting two parts (one subtree from every parent) and interchanging them.

The point in the program where a genetic operator is applied is selected at random. However, in our implementation, in order to preserve the structure and executability of a program, it gets parsed into a tree using the codons and the generative grammar. Using the tree, Mutation will generate a random subtree having as root the same non-terminal as the one used for deriving the original subtree. Crossover will try to find two compatible subtrees, for example two conditional statements. This is achieved by interchanging subtrees with root nodes corresponding to the same non-terminal. Following this procedure the executability of the children is guaranteed. Crossover and Mutation can be applied in two ways:

1. Koza [42] genetic operations: use Crossover 90% of the times and Mutation 10% of the times.
2. Mixed steady-state GP genetic operations²: apply Crossover and then Mutation on the children.

The selection of a parent is performed via *Tournament selection* by picking a number (the tournament size) of random programs from the population and then selecting the one most fit out of them. In this way selection pressure is applied, meaning that fit programs reproduce more often than unfit programs. The higher the tournament size, the higher the selective pressure. A size of 1 is equivalent to random selection. At the other extreme, a size equal to the population size always selects the best individual from the population. The default tournament size is set to 7, because it is a common value in genetic programming literature [60].

²From now on referred to as “mixed genetic operations”

3.3 Conditional Whole-Population Heuristics

Classic Genetic Programming is a whole-population heuristic meaning that it changes all the individuals from one generation to the next. Without a condition to enter the population, every new individual is accepted. This allows for a lot of bad individuals (having a high fitness) join the population, which will spawn more bad individuals in the following generation. Because of this, the good programs are harder to find. In the *conditional* whole-population heuristics, children compete directly with their parents and get accepted into the next generation only if they fulfill a condition. In addition, these heuristics iterate over the whole population and attempt to *improve* each individual. Thus, every individual will reproduce, which differs from classic GP, which prevents the reproduction of the worst individuals through Tournament selection.

3.3.1 Hill Climber

In the *Hill Climber* (HC) heuristic, the parent's and child's fitnesses are compared and the one with the lower fitness enters the next generation. For the case of equality, the child is accepted in order to enable the traversal of plateaus in the fitness landscape. Thus, the HC heuristic gradually improves the mean population fitness over time. We implemented three variations with different genetic operations. The first uses only Mutation. The second variation uses Koza genetic operations. The third uses mixed genetic operations. For Crossover, the second parent is chosen through a tournament, and it only returns the first child³. We test each of these variations in Chapter 5. The pseudocode is listed in appendices A.2 and A.3.

3.3.2 Simulated Annealing

Due to the high selective pressure, the HC heuristic can get stuck in a local minimum. Furthermore, due to the nature of the search space, a less fit child can lead to much fitter subsequent children in the following generations. In order to enable the acceptance of less fit children, the acceptance condition is replaced with a *simulated annealing* (SA) process [13]. A child that is less fit than its parent, still has a probability of being accepted given by P_{sa} from Equation 6.6. f_p and f_c are the fitness values of the parent and the child, respectively; pde makes the probability decrease faster as fitness grows (by default it is set to 2). G_c is the current generation number, while G_{max} is the maximum number of generations. T_{sa} from Equation 3.2 represents the temperature for a certain generation number (the temperature schedule).

³The first child is the one having the first parent's tree root.

$$P_{sa} = \begin{cases} 1 & , f_c < f_p \\ T_{sa} \times \left(\frac{f_p}{f_c}\right)^{pde} & , f_c \geq f_p \end{cases} \quad (3.1)$$

$$T_{sa} = \left(1 - \frac{G_c}{G_{max}}\right) \quad (3.2)$$

At high temperatures, the search process is in an exploration phase, in which worse children are easily accepted. As the temperature decreases, the process enters an exploitation phase, in which the search focuses on improving the mean population fitness. The fact that this heuristic initially accepts worse children is found to be beneficial in avoiding local minima [13]. We enforce that the best individual will never be replaced by a worse individual. The same three genetic operations variations (as for HC) are tested. The pseudocode is listed in appendices A.2 and A.4.

3.3.3 Tabu Search

In the *Tabu Search* (TS) heuristic, every created program is stored in a list called a *tabu list*. A child is accepted if it does not exist in the tabu list, thus ignoring the fitness value. This heuristic also has been shown to be effective in avoiding local minima by not allowing the already known programs to be accepted. Furthermore, it prevents the search to cycle between the same set of programs and thus TS is able to find solutions with less computational effort than a *history-unaware* heuristic (such as CGP) [27, 28]. However, because the search space is infinite, and the lack of influence from the fitness, TS becomes almost like a random search, so it remains to be explored how effective it is when applied on our type of problems. Nevertheless, the hint to not use tabu search alone, given by Glover in [26], is clear: “Tabu search may be viewed as a ‘meta-heuristic’ superimposed on another heuristic.” . The pseudocode is listed in appendices A.2 and A.5.

3.3.4 GOMGP – GP with GOMEA Inspired Generations

GOMGP represents an adaption of GOMEA (as presented in [7]) for Genetic Programming. In our implementation, we focus on GOMEA’s three-phase generation, the greedy acceptance condition and the halting criterion, which are described below. This heuristic halts when no improvement is made to the best individual over a certain number of generations. The best individual (even if not perfect) is returned. On one hand, this avoids the case in which computational budget gets consumed when the search gets stuck into local minima. On the other hand, it also avoids the case when the best individual can still be improved, but the preset number of generations has been reached.

GOMGP uses more complex versions of Mutation and Crossover than the ones presented in Section 3.2.3. The original operators, are applied at random positions in an individual’s tree. The operators for GOMGP are applied at a specified tree-node number from the tree of the program. The tree-node numbers are identified by traversing and linearizing the tree. We experimented with tree-node lists created through breadth-first search (BFS), depth-first search (DFS) and sorting the tree-node list based on descending subtree size (STS) order. Therefore, the Mutation operator is applied exactly at the specified tree-node number. Crossover is applied between the specified tree-node of the first parent and one of the compatible nodes (without caring about its node number) from the second parent, and finally only the child corresponding to the first parent is returned.

The heuristic goes as follows. Each generation has three phases and we keep track of the improvements made to the individuals in each generation.

1. In the first phase we attempt to make **only one improvement** to each individual by applying both Mutation and Crossover at specific tree-node numbers. During Crossover, the second parent is selected with the Tournament method. We experimented with two tree-node number selection strategies. The first is to have fixed numbers e.g., 2 to 10. The second is to pick the node number for the current attempt depending on the size of each program’s tree-node list e.g., the tree-node at 10%, at 20%, ... at 100% in the node list.
2. The second phase is similar to the first, with the difference that we try to improve the yet unimproved individuals, having the best known individual as the second parent for Crossover.
3. In the third phase, the unimproved individuals get replaced altogether by the best individual.

If the number of generations in which the best known individual did not change is larger than the length of the no-improvement stretch (NIS⁴), then in the second phase, every individual (i.e., the improved and the unimproved) gets the best known individual as the second parent. At the end of the generation the improvement statistics get reset.

Finally, as a result of the third phase, it can happen that all the population ends up consisting of the same individual, in which case the search halts and the individual (which is also the best known individual) is returned together with its fitness. In Chapter 5 we search for the best parameter combination by varying: the tree linearizing method, the node number selection strategy, the Tournament size, the improvement condition (“ $f_c < f_p$ ” vs. “ $f_c \leq f_p$ ” vs. “ $f_c \leq f_p$ and $c \notin \text{tabu.list}$ ”) and mixed vs. Koza genetic operations. The pseudocode is listed in Appendix A.6.

⁴In the GOMEA paper [7] $\text{NIS} = 1 + \lfloor \log_{10}(P_{\max}) \rfloor$; i.e., allow an extra generation for every order of magnitude (factor 10) that the population size is larger.

3.4 Conditional Generation Gap Heuristics

In this section, we present two new population-based heuristics, developed by us, which are inspired by the steady-state GP [58,90] with generation gaps and also by the HC, SA and TS heuristics. The generation gap represents the proportion of the population that has to be replaced. We chose to always replace the individuals with the worst fitness.

3.4.1 GATLE – Generation Gap Tabu Hill Climber

GATLE (**G**eneration **gAp** **T**abu **hilL** **climbEr**) is built on the steady-state GP algorithm with a configurable generation gap size (e.g., it replaces $\delta = 10\%$ of the programs from one generation to the next). In our experiments δ ranges from 1% to 30%. We had chosen a fairly harsh selection mechanism, by replacing the δ most unfit programs from the current generation. The new programs, which are candidates for filling the gap, get generated by selecting parents through tournaments from the whole current generation, and applying genetic operations. The candidates get accepted if they fulfill an acceptance condition, which is described further. From the current generation, we define f_{worst} as the fitness of the most unfit individual, and f_{goal} as the fitness of the most unfit individual that gets to enter the next generation with respect to δ . Moreover, we store all the programs generated in a tabu list. Then the acceptance condition for a new program c to enter the next generation is that $c \notin \text{tabu.list}$ and its fitness f_c satisfies $f_c < f_{goal}$. Let P_{max} be the maximum population size. Then let the minimum population size $P_{min} = P_{max} - P_{gap}$, where $P_{gap} = \lfloor P_{max} \times \delta \rfloor$. If $\lfloor P_{max} \times \delta \rfloor = 0$ then $P_{gap} = 1$. The next generation is formed as follows:

1. The current generation of programs gets sorted ascending based on the fitness score
2. The first P_{min} programs get copied into the next generation
3. Try⁵ to find P_{gap} programs in order to have P_{max} programs in the next generation

In Chapter 5 we experiment with various generation gap sizes and mixed vs. Koza genetic operations. The pseudocode is listed in Appendix A.7.

3.4.2 GATSE – Generation Gap Tabu Simulated Annealing

GATSE (**G**eneration **gAp** **T**abu **S**imulated **annE**aling) is a version of GATLE with a leaner acceptance condition. In GATSE, if $f_c \geq f_{goal}$, the child has a second chance of entering the next population generation with the probability P_{gatse} . The probability is calculated following the same intuition from

⁵Have a limited number of attempts in order to avoid getting stuck at this generation.

the SA heuristic, see Equation 6.6. We keep the same notations from Section 3.3.2. In the first G_{T1} generations, $T_{gatse} = 1$, and in the last G_{T0} generations, $T_{gatse} = 0$. In the rest of the generations, the temperature decreases linearly (see Fig. 3.1). The value of T_{gatse} is given by Equation 3.3. We do this in order to have multiple pure exploration generations, respectively multiple pure exploitation generations.

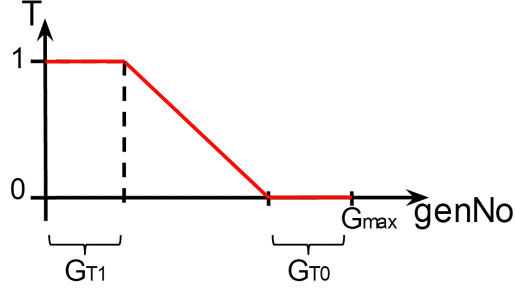


Figure 3.1: The temperature schedule

$$T_{gatse} = \begin{cases} 1 & , G_c < G_{T1} \\ 1 - \frac{G_c - G_{T1}}{G_{max} - G_{T1} - G_{T0}} & , G_c \in [G_{T1}, G_{max} - G_{T0}] \\ 0 & , G_c \geq G_{max} - G_{T0} \end{cases} \quad (3.3)$$

$$P_{gatse} = \begin{cases} 1 & , f_c < f_{goal} \\ T_{gatse} \times \left(1 - \frac{f_c - f_{goal}}{1.01 \times f_{worst} - f_{goal}}\right)^{pde} & , f_c \in [f_{goal}, f_{worst}] \\ 0 & , f_c > f_{worst} \end{cases} \quad (3.4)$$

In Equation 3.4 the probability of a worse child to enter the population is computed with respect to how much better f_c is compared to f_{worst} , rather than how much worse f_c is compared to f_{goal} (as for P_{sa} in Equation 6.6). In Chapter 5 we also test the P_{sa} acceptance condition (we still keep T_{gatse} for obtaining the temperature). The pseudocode of GATSE is listed in appendices A.7 and A.8.

Chapter 4

MetaCompiler Design

This chapter builds on the information about the GP-based heuristics (the core of the search methodologies) introduced in the previous chapter by describing the software architecture of the MetaCompiler. Here we describe the elements that support the algorithm discovery: the agent-based simulator and agent programming language, the skeleton of an algorithm and the execution flow of a MetaCompiler experiment. We also describe a new type of generic algorithmic building blocks developed by us, which are helpful in discovering solutions for problems where we have very little problem knowledge. The information presented in this chapter is essential for setting up experiments targeted at searching for specific algorithms.

4.1 Agent-Based Simulator – NetLogo

NetLogo is a multi-agent programmable modeling environment developed at Northwestern University¹, currently at the 5.0.4 version. It is written for the Java virtual machine, so it can run on multiple computer platforms.

Simulations in NetLogo feature a 2D world in which agents can move. We use NetLogo for simulating networks in which the agents represent network nodes able to create links to the other agents (their neighbors) with respect to the Euclidean distance between them (the transmission range). Because agents can be made mobile and continuously change their set of neighbors, NetLogo makes a very handy platform for simulating Mobile Ad-Hoc Networks (MANETs). The simulation proceeds in *ticks*, which represent the most granular units of time. A NetLogo program, in its most general form, can be structured as in Listing 4.1.

In the “**setup**” procedure, the agents are set into an initial state (e.g., position, links to neighbors, color), while with every execution of the “**go**” procedure, the agent behavior is updated. Standard statements, such as

¹<http://ccl.northwestern.edu/netlogo/>

Listing 4.1: General structure of a NetLogo program.

```
1 global variable declaration
2 agent variable declaration
3 to setup
4     <statements>
5 end
6 while (ticks < tick_limit)
7     to go
8         <statements>
9         ask (agents) [ <statements> ]
10        <statements>
11    end
```

procedure calls, assignments and conditionals, as well as functions, arithmetic and logical expressions, can be used as **<statements>**. There are two scopes of execution: the global scope and the agent scope. In the global scope, statements have access to information from any agent. We use it to compute the fitness function using aggregates such as: **min**, **max**, **mean**, **count**, etc. In the agent scope, the statements at line 9 in Listing 4.1 have access to information from the agent executing the code, and from the neighbouring agents. The MetaCompiler generates code containing building blocks that are executed only in the agent scope. Making agents execute statements is done through the “ask” keyword applied to a set of agents. Sets of agents are returned by primitives, such as: **agents** (all the agents) and **link-neighbors** (neighbouring agents). Filters can be applied on these sets using the “with” primitive followed by a condition (a boolean expression). Agents remain in the set **with** is applied to, if they fulfill the condition. The condition is executed in every agent’s scope using their local variables values.

4.1.1 Code Injection

We use code injection to define more general building blocks. One example is a function that returns the number of neighbors that satisfy a given condition. However, we do not have enough problem knowledge to define the condition in the function, so we let the MetaCompiler discover it. The statement that counts the neighbors has to be written inside a function, therefore the condition has to be given as an argument when calling the respective function. In order to execute the condition in each neighbor’s scope it needs to be injected in the argument position of the “with” primitive. Below we give an example to clarify the need and usage of code injection

In Listing 4.2, we define a function that returns the number of neighbors having their local variable **Y** equal to 2. It is important to note that **Y = 2**

Listing 4.2: A function that counts neighbors with $Y = 2$

```
1 to-report CountY2 []
2   report count link-neighbors with [Y = 2]
3 end
```

Listing 4.3: An incorrect function that should count neighbors with condition as argument

```
1 to CountFiltWrong [filt_]
2   report count link-neighbors with [filt_]
3 end
```

gets executed in the agent scope of every agent from the `link-neighbors` set. However we do not know for certain that “Y” is the correct variable to consider and furthermore, whether it should be equal to “2”. In order to let the MetaCompiler discover the condition we need to rewrite the function with the condition as a parameter, see Listing 4.3. Unfortunately, this will not work as expected, because when the procedure is called: `CountFiltWrong (Z > 4)`, the argument gets evaluated in the agent scope of the caller. To overcome this, we use the code injection capability of NetLogo, see Listing 4.4. “`sfilt_`” is a string, so when the procedure is called: `CountFiltOK ("Z > 4")` the condition gets executed in the neighbor’s scope.

We will come back to writing more general building blocks in the last part of this chapter.

4.1.2 Concurrency of Execution

Ideally, when simulating a network, agents execute their code independently from one another, concurrently. However, using “`ask`”, the first agent executes its code block, then the next one and so on. In order to simulate concurrency, NetLogo provides the “`ask-concurrent`” primitive. But still, in order to avoid race conditions, agents use a turn taking mechanism to execute code. The first agent takes a turn, then the second, until every agent has finished executing their statements. An agent’s “turn” ends when

Listing 4.4: A function that counts neighbors with condition as argument

```
1 to CountFiltOK [sfilt_]
2   report count link-neighbors with [run-result sfilt_]
3 end
```

it performs an action that can affect other agents such as moving or sending a value to a neighbor. Changing a local variable does not count as a turn.

4.2 MetaCompiler

MetaCompiler is the name of the whole software suite for discovering new algorithms. It is written mostly in Java and Scala, and extends the EpochX genetic framework.

4.2.1 Execution Flow

In Fig. 4.1 the major components of the work and execution flow of a MetaCompiler experiment are presented. The blocks represent major modules of the flow. The arrows represent temporal precedence except for the dotted lines which specify the input of a block. The dashed lines are optional depending on the genetic heuristic used.

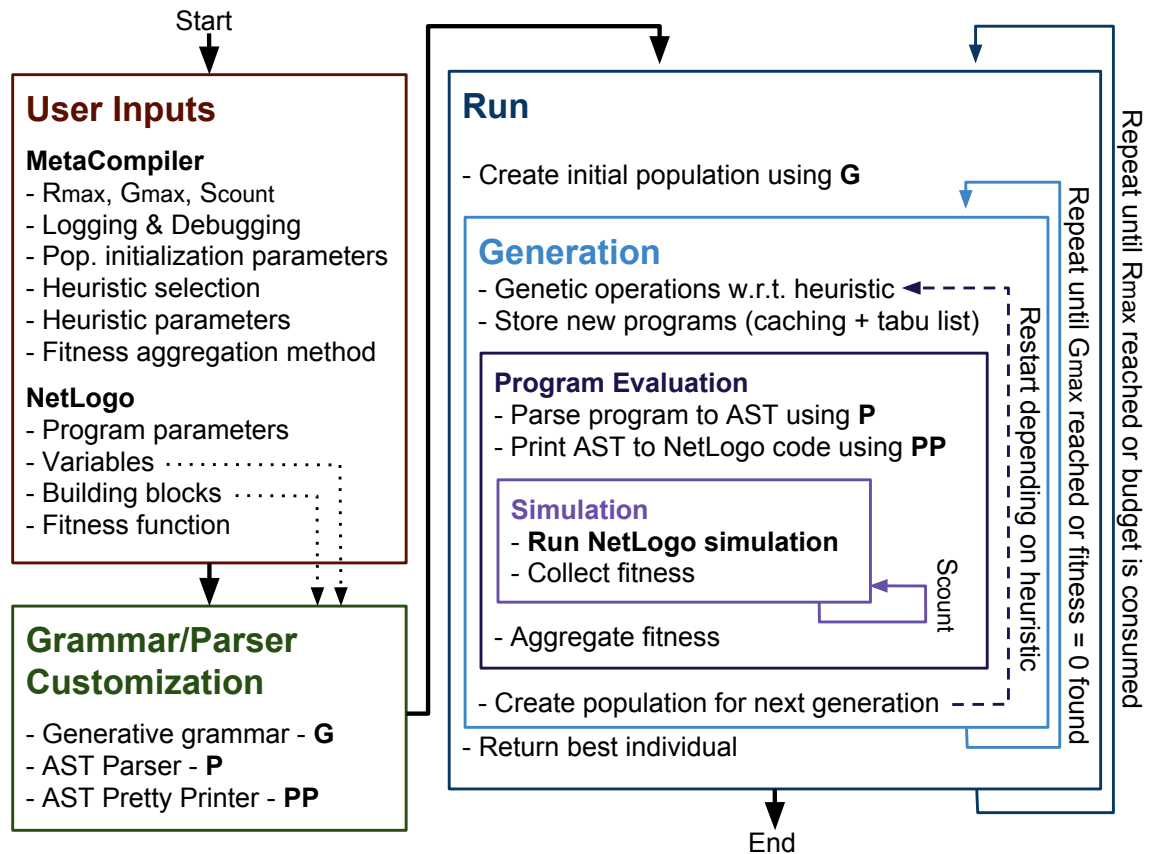


Figure 4.1: MetaCompiler execution flow.

The user provides the inputs necessary for running an experiment. The “MetaCompiler User Inputs” configures the various modules of the MetaCompiler, including the genetic heuristic. The “NetLogo User Inputs” configures the NetLogo programs that are going to be generated. Program parameters such as world size, number of agents, number of ticks, define the NetLogo simulation environment. The fitness function, the variables and the algorithmic building blocks (functions and procedures) are also specified by the user via a library file. The latter two also act as input for the “Grammar/Parser Customization” module.

After reading the user inputs, the MetaCompiler will customize the generative grammar and the two parsers (written in ANTLR) with respect to the variable names, building blocks and their declared types. This step is necessary because for every specific problem, we try to discover a program that solves it using the specified building blocks and the variables we provided. So these have to be included in the generative grammar, and the “Grammar/Parser Customization” module does this in an automatic way. Without it, the user would have to modify the grammar and parsers by hand, such that they contain the correct set of variables and building blocks. To accomplish this, the grammar and parsers are specified using the Velocity² templating language. At runtime they get customized using VelocityEngine and then compiled into Java classes.

After customizing the grammar and parsers, the main execution loop starts. A MetaCompiler experiment consists of multiple (R_{max}) runs executed sequentially. It stops when all the runs have been executed or the computational budget had been consumed. Within each run, an initial population of programs is randomly generated using the generative grammar. After executing G_{max} generations the best individual of the run is returned. A run ends when all the generations have been executed, or a program with a perfect fitness had been found, or the heuristic had converged (in case of GOMGP). Newly generated programs get stored for caching purposes and to populate the tabu list (when needed), then they are evaluated. After the evaluations, depending on the heuristic (and the quality of the programs), the population of the next generation is created. Evaluating a program consists of transforming it into a NetLogo program, and then running it S_{count} times using the NetLogo library³. The multiple results get aggregated into the program’s fitness (the resulting fitness gets stored in the cache) using one of the functions: **average**, **median**, **minimum**, **maximum**. In the Program Evaluation step, all the NetLogo simulations get added into a working pool, which can be executed in parallel in a multi-threaded environment.

²<http://velocity.apache.org/>

³A Java library provided with NetLogo that can be used to simulate NetLogo programs from Java code.

Listing 4.5: Program skeleton

```

1 Setup:
2   Initialization of global & local variables
3   Initialization procedures
4 Go: While (maxIterations not reached)
5   StartGlobal static procedures
6   Agent methods:
7     StartLocal static procedures
8     !!!Combination of generated statements!!!
9     EndLocal static procedures
10  EndGlobal static procedures
11 Finally: evaluate Fitness function

```

4.2.2 Agent Program Skeleton and Building Blocks

Every program produced during the generation step, parsing and printing process, has the structure in Listing 4.5. All the parts, except for the 8th are static (i.e., every generated program from a certain experiment contains

Name	Data type	NetLogo syntax
Procedure (statement when called)	void	to <i>name</i> [<i>param</i> [*]] <i>statement</i> [*] end
Function (expression when called)	number	to-report <i>name</i> [<i>param</i> [*]] <i>statement</i> [*] report <i>expression</i> end
Assignment statement	void	set <i>var</i> <i>expression</i>
Conditional statement simple	void	if <i>condition</i> [<i>statement</i> [*]]
Conditional statement double	void	ifelse <i>condition</i> [<i>statement</i> [*]] [<i>statement</i> [*]]
Number constants / exp	number	1 ... 100; 0.01 ... 0.99
Boolean constants / exp	boolean	true; false
Relational operators / exp	boolean	!=; ==; <; >; <=; >=
Logical operators / exp	boolean	and; or; not

Table 4.1: Procedures, Functions and built-in statements. “*” means zero, one or multiple repetitions. A *condition* is an expression returning a boolean.

them in the exact same way). The `StartGlobal`, `EndGlobal`, `StartLocal`, `EndLocal` static procedures and the Fitness function as well as the global and local variables and their properties are defined by the user in the configuration file. Arguments that can be constants or variable names, can be passed to the static procedures and functions.

The 8th line in Listing 4.5 is where the GP creates the mix of building blocks, function and procedure calls, presented in Table 4.1. The functions and procedures get declared beforehand by the user in the supplied library file. The passed arguments are specified in the configuration file. An argument can be static⁴: case in which it is a constant or a variable name, or dynamic (being an expression generated by the GP) case in which it is specified according to its data type: integer, float, number⁵, boolean, “number as string” or “boolean as string”. Note that arithmetic operators are not built-in, because they are not always necessary. When needed, they can be implemented with functions. On the other hand, the conditional statements and the constants are built-in but can be disabled from the configuration file.

4.2.3 Discovering Algorithmic Building Blocks

Programs made out of ifs, assignments, procedures and functions without any parameters can solve a limited number of problems. Solutions to problems can be expressed by a chain of statements e.g., the Santa Fe Trail problem [43], provided that the set of possible statements is fairly small, and all of them are known beforehand (e.g., `turnLeft()`, `turnRight()`, `moveForward()`). However, in many cases we do not know the complete set of statements that is sufficient for finding the solution. For problems such as Symbolic Regression and the 11-bit Multiplexer [43] defining a sufficient set of statements is not possible, hence we need more abstract building blocks. These are developed by receiving input parameters, e.g., `add(a,b)`, `mul(a,b)`, etc. Whenever the generative process discovers a structure such as `mul(mul(X,X),X)` we say that a building block, in this case the one which computes X^3 , was discovered.

The ability to discover building blocks is important, because it simplifies the definition of the building blocks set. Writing more abstract building blocks (e.g., a function computing the product of two numbers given as parameter vs. function computing “ X^3 ”) requires less prior problem knowledge. Ideally, this allows the user to just specify the generic structure of a building block and leave the GP process to fill in the specificities. Going back to the example in Section 4.1.1, it is unfeasible to write a version of `CountY2` for every integer. Furthermore, because the condition given as a filter should

⁴This is useful when we want to customize a generic building block, for example: `pow(..., 2)`, but we do not want to change its code in the library.

⁵Integer or float.

Listing 4.6: A function that returns the maximum result of an expression run by all neighbors.

```
1 to-report maxValueOfNeigh [exp_]
2   report max runresult ("[" + exp_ + "]" of link-neighbors")
3 end
```

be executed in the scope of the neighboring agent, the condition has to be passed as a symbolic argument in the form of a string.

We have extended the generative grammar to allow more generic building blocks such as `CountFilterOK`. The main distinction is that arguments are passed as symbols (i.e., strings). The symbols can have a specific data type (i.e., boolean or number). We enforce that a function that gets a parameter as string and returns a number, cannot appear in the code generated for a dynamic argument that has to have the data type number as string or boolean as string. The reason for this is that the code would feature a string within a string, which is not supported in NetLogo.

An example of a function with a boolean as string parameter can be considered the one in Listing 4.4 from Section 4.1.1. An example when it is required to pass an argument of the type number as string can be the one in Listing 4.6. This building block computes the maximum out of the set of results returned by an expression that is executed by every neighbor, be it: `"X"`, `"mul(X,Y)"` or the `"count-neighbors()"` function. Therefore, by adding this new capability we have added more versatility in specifying generic building blocks, thus requiring less prior problem knowledge. An example of generic building blocks usage is shown in the “Maximal Independent Set” case study in Section 6.1.

Chapter 5

Benchmarking of the Genetic Heuristics

In this chapter we present the benchmarking results of the various heuristics that we introduced in Chapter 3. This step is essential for identifying what heuristic to choose (together with the optimal parameter combination) for discovering algorithms for large-scale networks. Having a high performing heuristic is essential for the success of algorithm discovery.

The performance of a heuristic is described as the fitness of the best program found during an experiment (a MetaCompiler run, see Fig. 4.1). However, since the best found program varies from one experiment to another, we define the performance as the mean best fitness averaged over 50 experiments (or runs). We call a solution, a program with a fitness of 0.

In the first part of this chapter we describe the three test problems used for benchmarking. These are followed by the experiments consisting of varying several important parameters and measuring the effect of each variation. We also compare the performance of our heuristics with state-of-the-art genetic frameworks that used the same test problems. Finally, we discuss the observed influence on performance of the various tested parameters. The discussion ends with a recommendation (based on the observed results) of the best suited heuristics for our types of algorithms i.e., algorithms for large-scale networks.

5.1 The Test Problems

In order to test the heuristics developed, we had chosen three problems that are often used in literature: the “Artificial Ant on the Santa Fe Trail”, “Symbolic Regression” and “Synthesis of an 11-bit Multiplexer” problems (see Table 5.1). Because these problems are very simple with respect to computing the fitness of an individual, we had chosen not to execute NetLogo code due to the large overhead introduced by the NetLogo library. Instead,

Name	Function Set	Fitness function
Santa Fe Trail	moveForward(); turnLeft(); turnRight(); if(foodAhead()) ... else ...	Given the Santa Fe Trail [43] 32×32 grid with 89 food pellets; $F(p)$ = food left after 400 moves.
Symbolic Regression	add(i,j); sub(i,j); mul(i,j); div(i,j); sin(i); cos(i); exp(i); log(i); x	$F(p) = \sum_{i=1}^{20} p(x_i) - Y(x_i) $ where $Y(x)=x^4+x^3+x^2+x$ and $x_i \in \{-1, -0.9, \dots -0.1, 0.1, \dots 1\}$
11-bit Multiplexer	and(i,j); or(i,j); not(i); if(...) ... else ... ; $a_0 - a_2$; $d_0 - d_7$	$F(p)$ = Number of cases with an incorrect output (for the address $\bar{a}_i$ and the input $\bar{d}_i$) out of 2048 in total

Table 5.1: Test problems building blocks.

the programs are evaluated through an interpreter, which we implemented in Scala, designed specifically for the three problems. However, it is important to note that the generative grammar featured the same customization process and retained, essentially, the same generative properties as for the NetLogo programs.

5.1.1 Artificial Ant on the Santa Fe Trail

The Artificial Ant problem consists of an ant-like agent that is in search of food pellets on the Santa Fe trail¹ in a 32×32 grid world with periodic boundaries. The program with which an ant is controlled consists of a series of statements shown in Table 5.1, each one representing a move, except for foodAhead(). After all the statements are executed, the execution is restarted, until all of the 400 moves budget is consumed. Programs that get stuck, for example: if(foodAhead()) turnLeft() else turnRight() in an area without any food, are stopped without all of the moves budget being consumed.

Solutions for this problem tend to be short and use only a few conditional statements. We expect this problem to be easily solved. However, due to the conditionals, inactive code tends to be large, which may prevent finding a solution. The fitness values are integers between 0 and 89.

5.1.2 Symbolic Regression

Symbolic Regression problems consist of discovering programs able to compute the output of functions such as $Y(x)=x^4+x^3+x^2+x$ with x given as input. The programs are made out of building blocks showcased in Table 5.1. Note that some functions are completely useless in trying to discover $Y(x)$.

¹<http://upload.wikimedia.org/wikipedia/commons/b/ba/SantaFeTrail.gif>

Their presence is necessary for the case when the expression for $Y(x)$ is not known, yet there is a way of computing it (e.g., by some *black-box* behavior). The values for x come from the set $\{-1, -0.9, \dots -0.1, 0.1, \dots 1\}$.

Solutions for this problem tend to be short i.e., most of the solutions look as $Y(x)$ expressed with functions from Table 5.1. We expect this problem to be very easily solved. The fitness can take any value from 0 to infinity.

5.1.3 Synthesis of an 11-bit Multiplexer

An 11-bit Multiplexer gets as input 3 address bits and 8 data bits, and its task is to output the correct data bit from the address encoded in the address bits. Therefore, there are 2048 possible combinations of bits that have to be tested. The goal of the 11-bit Multiplexer problem is to discover a program composed of boolean functions, described in Table 5.1, that correctly decode 3-bit addresses.

Solutions for the 11-bit Multiplexer tend to be very long, with a lot of nested conditionals, which create large portions of inactive code. Therefore, we expect this problem to be the hardest to solve out of the three benchmarking problems. The fitness function ranges from 1024 to 0 and the values usually can be divided by powers of 2, e.g. 32, 64. This means that a lot of programs will have the same fitness, thus making program selection less relevant.

5.2 Benchmarking Experiments

In Table 5.2, we present the various common parameters used during experiments. For each heuristic, the number of program evaluations in a generation differs. Therefore, in order to compare the heuristics fairly, we configure an unlimited number of generations, while having a fixed computational budget. This means that at the end of each generation if the number of evaluated programs exceeds the budget, the execution stops. Also, in order to speed-up execution, we cache the program's fitnesses, meaning that each program gets evaluated only once. Since a program for any of the three problems is deterministic, we execute it only once. In case of the heuristics based on Simulated Annealing, the temperature is now computed with respect to the computational budget and not to the generation number. Also in GATSE, G_{T1} and G_{T0} are defined with respect to the budget. In GOMGP, the default Tournament size is 1, i.e., in the first phase the second parent is selected at random from the population. For practical purposes we limit the tree size to 500 nodes. That means if Crossover or Mutation create a program bigger than the limit, it gets rejected (in CGP, GATLE and GATSE), or it gets replaced with its parent (in HC, SA and GOMGP). Finally, for each parameter combination, we execute 50 experiments in order to have a reliable assessment. In most of the benchmarks we plot the performance of

Parameter	Value
Population size	4, 8, 16, ... 16384
Tournament size	7
Crossover / Mutation probabilities	90%/10%
Experiments per parameter combination	50
Budget per experiment	50,000 program executions
Tree size limit	500 nodes
GATLE, GATSE maximum attempts	10
GATSE G_{T1} / G_{T0}	20% / 20% of the budget
GOMGP improvement attempts	10

Table 5.2: Common parameters of the heuristics during the benchmarking experiments.

a certain heuristic per parameter combination as population size vs. mean best fitness. We choose to always vary the population size, because it is considered in literature to be the parameter with the largest influence on performance. Note that in some plots, large mean best fitness values are intentionally left outside the plot in order to focus on the most relevant part of the fitness interval.

We define the following metrics:

- M1 = mean best fitness, averaged over 50 experiments (always referred to as “performance”)
- M2 = standard deviation of the best fitness
- M3 = number of found solutions, grouped by population size
- M4 = mean computational budget consumed per experiment (percentage of 50.000)

In the next sections we study the performance (mean best fitness) of the developed heuristics, with respect to their important parameters. Then, in Section 5.2.7 we choose the best parameter combinations for each heuristic and compare them using all four metrics defined above. Finally, we make a comparison with state-of-the-art heuristics.

5.2.1 Genetic Operations in HC and SA Heuristics

In this set of experiments we test the performance of the SA and HC heuristics for various genetic operations variants: Mutation alone, Koza and

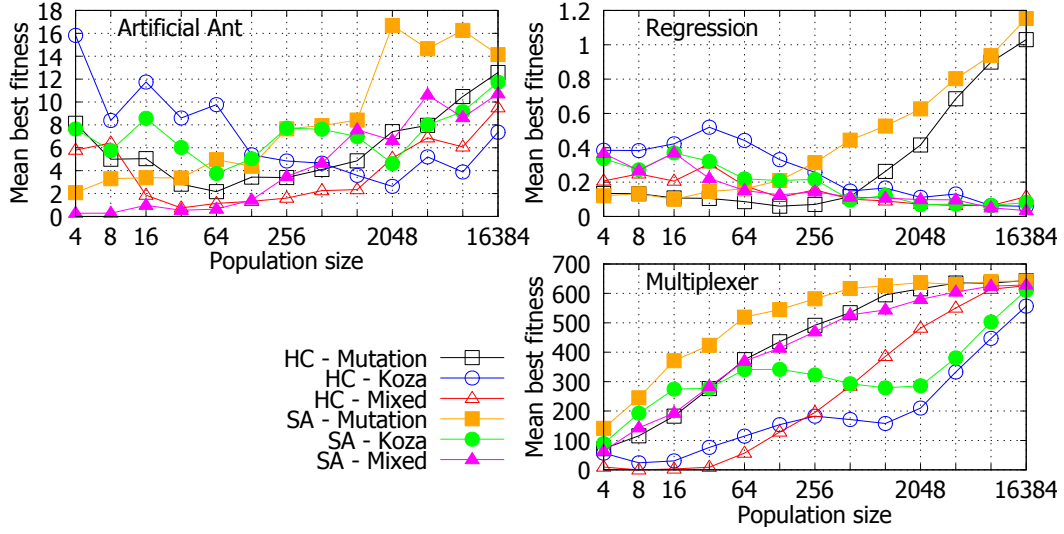


Figure 5.1: SA and HC: Mean best fitness across various population sizes and for different genetic operations.

Mixed genetic operations (for the description of the genetic operations, see Section 3.2.3).

In Fig. 5.1, we observe very good performance for the Mixed operator in the Artificial Ant and the 11-bit Multiplexer experiments. For the Symbolic Regression experiments, Mutation alone is better for small population sizes, while operators with Crossover are better for large population sizes. We cannot declare any clear winner between HC and SA: HC is better than SA in the 11-bit Multiplexer but worse for the Artificial Ant. One important characteristic that stands out is the good performance of these heuristics for small population sizes. Also note the smoothness of the curves for the Multiplexer experiments.

5.2.2 Generation Gap in GATLE and GATSE

In these experiments we vary the generation gap percentages. This parameter is important in order to determine how many individuals need to be accepted using the current generation as a gene pool.

We notice in Fig. 5.2 that overall, a larger generation gap improves the performance for both GATLE and GATSE. However, this parameter does not change the general performance trend. This is because, it alters only the *granularity* of the generations (how many better individuals we have to find per generation). In general, GATLE and GATSE perform better for larger population sizes between 2048 and 8192. We also observe that overall, GATSE has a better performance than GATLE, which is most clearly shown in the 11-bit Multiplexer experiments.

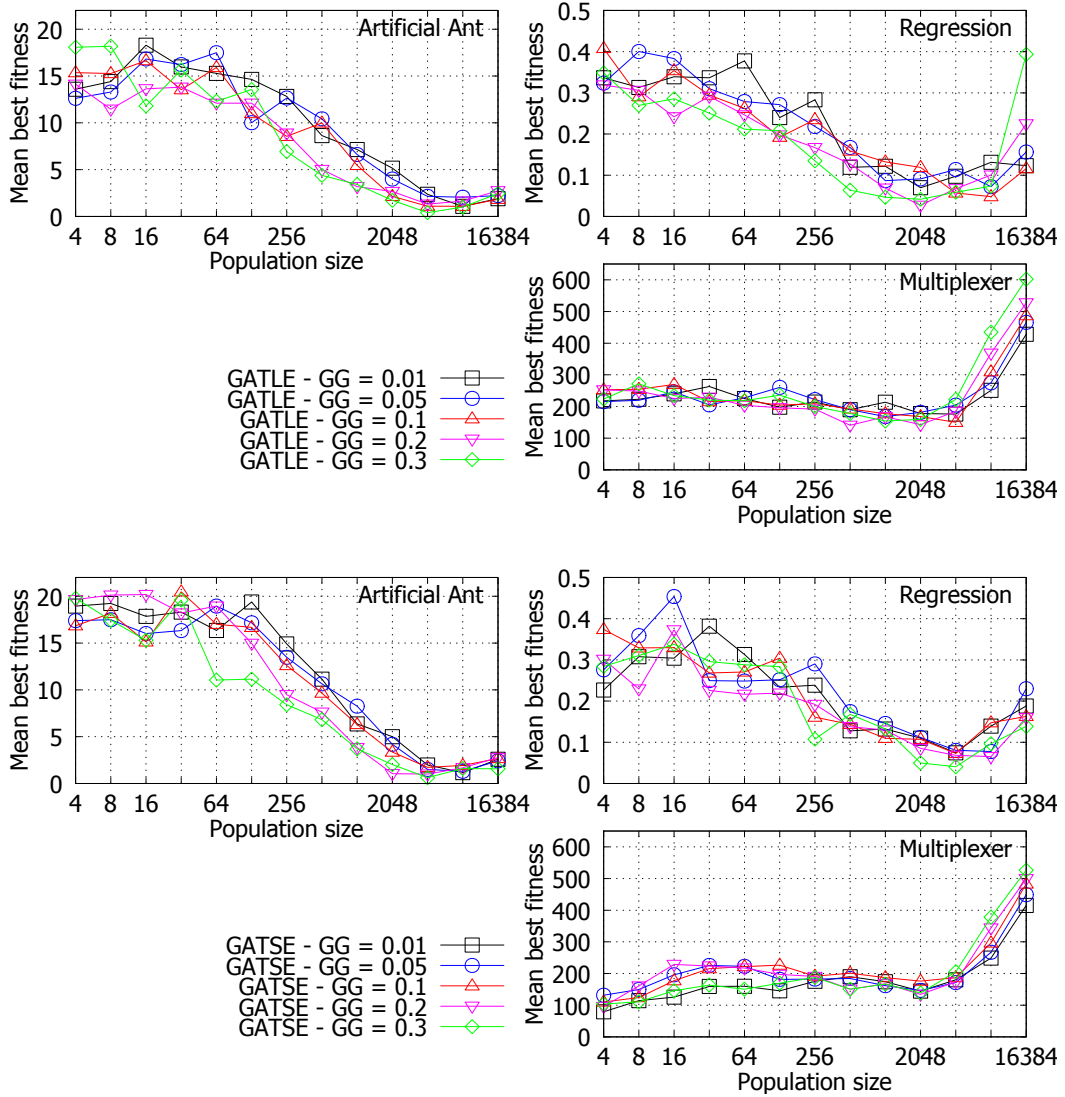


Figure 5.2: GATLE and GATSE: Mean best fitness across various population sizes and for different generation gap sizes.

5.2.3 Genetic Operations and Acceptance Probability in GATLE and GATSE

In this section we test the influence on performance of the two genetic operations we have used for the HC and SA heuristics (i.e., Koza and Mixed operators). For GATSE we also test a simpler acceptance probability, which is based on P_{sa} (see Equation 6.6). The results are shown in Fig. 5.3.

Firstly, we observe that in most the cases the Mixed operations have

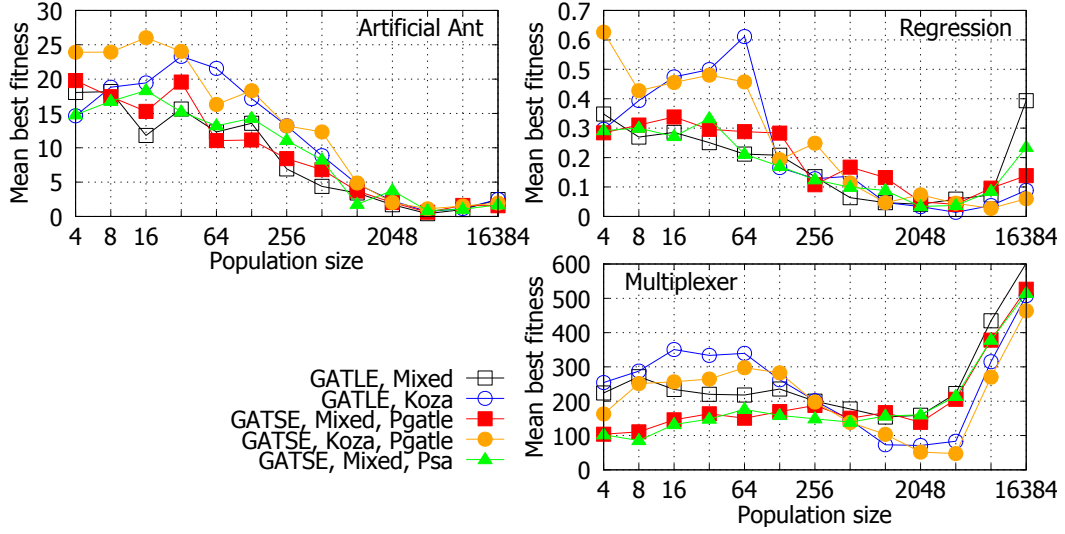


Figure 5.3: GATLE and GATSE: Mean best fitness across various population sizes and for different genetic operations and different acceptance probabilities in GATSE algorithm.

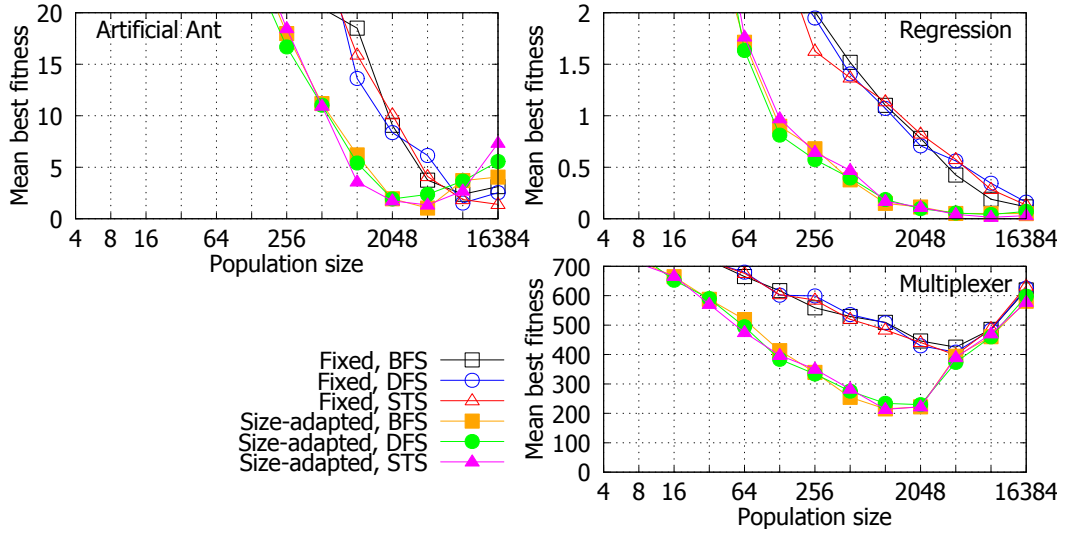


Figure 5.4: GOMGP: Mean best fitness across various population sizes and for different tree linearization and node selection strategies.

better performance than the Koza operations. Exceptions are represented by large population sizes (≥ 1024) in the 11-bit Multiplexer experiments. The performance of the Koza operations seems much more sensitive to population size. With regards to the acceptance probability used to allow worse children (see sections 3.3.2 and 3.4.2), we cannot say clearly which one of P_{gatse} and

P_{sa} is better. The first is better for some population sizes, especially in the Artificial Ant experiments, while the second is better, again, only for some population sizes in the Symbolic Regression experiments. In case of the 11-bit Multiplexer the two have similar performance. The main purpose of allowing worse children is to facilitate population diversity. Both P_{gatse} and P_{sa} manage to fulfill that purpose

5.2.4 Tree Linearization and Node Selection Strategy in GOMGP

In this set of experiments we test the three program-tree linearization strategies: breadth-first search (BFS), depth-first search (DFS) and sub-tree size sort (STS), which were proposed in Section 3.3.4. We also test two node selection strategies: fixed (node 1, 2, ... 10) and tree-node list size-adapted (node at 10%, 20%, ... 100% in the node list). The Koza genetic operations are used (we will compare Koza vs. Mixed in the next section). The results are shown in Fig. 5.4.

We observe that tree linearization has very little impact on GOMGP’s performance. This means that it does not matter if the changes are applied to big subtrees first and to small subtrees after. This is consistent with the original GOMEA paper [7] where the program parts to be crossed over are not ordered by size. Even so, BFS and STS are used in the parameter combinations that have the best performance. On the other hand, node selection has a big impact on performance, as we cannot fail to notice that size-adapted node selection has better performance for smaller population sizes. The reason for this is that the *fixed* strategy only modifies tree nodes close to the root of the tree, while the *size-adapted* can modify tree-nodes deeper in the tree. Being able to also make small changes to programs increases the chances of finding solutions.

5.2.5 Genetic Operations and Individual Improvement Conditions in GOMGP

With these experiments we test three improvement conditions (see Section 3.3.4) for both the Koza and the Mixed genetic operations. In “Lt” an individual is considered improved if his fitness is strictly smaller than his parent; in “Leq”, the individual can also have an equal fitness with his parent; and additionally, in “Leq, Tabu” the individual has to be all new. The “subtree size” tree linearization and “size-adapted” node selection strategies are used.

We observed that the “Lt” condition makes GOMGP converge really fast for small population sizes, thus not making use of the entire budget. The convergence happens too early such that good programs have no time to develop. This explains the very low performance for low population sizes.

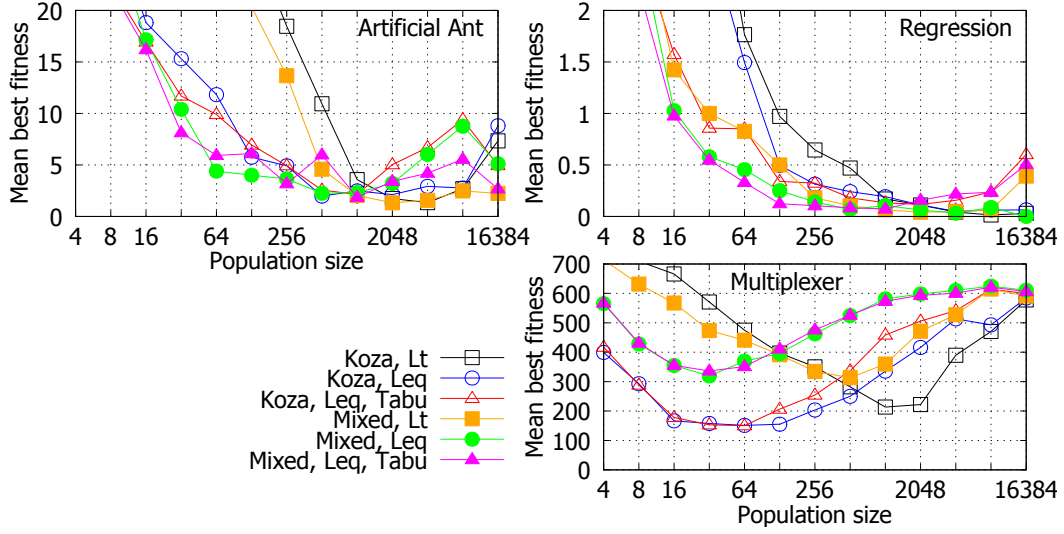


Figure 5.5: GOMGP: Mean best fitness across various population sizes and for different genetic operations and improvement conditions.

On the other hand, “Leq” impedes the convergence almost every time, because finding new individuals with the same fitness is easy due to inactive code. Adding the Tabu list aims to reduce this effect but it is only minimally successful, with the added negative effect of slightly reducing performance.

In Fig. 5.5, we observe that in general the larger the population size, the better the performance (except for “Leq” in the 11-bit Multiplexer). In most of the cases, having Mixed genetic operations results in better performance, than using Koza operations. For the Symbolic Regression problem with the “Mixed, Leq” variant and $P_{max} = 16384$ we managed to find a solution in every experiment.

5.2.6 Tournament Size in GOMGP

In this set of experiments, we want to observe the influence of the Tournament size on performance for both the Koza and Mixed operations. We use the “Leq” acceptance condition, with the “subtree size” tree linearization and “size-adapted” node selection strategies. Results are shown in Fig. 5.6.

In general, we observe that the larger the population size the better the performance of large Tournament sizes. This is expected because with a Tournament having a larger size, it is easier to find good programs in a large population. However, for small population sizes, the same set of (good) programs is used too often by the genetic operations, resulting in lower population diversity, hence lower performance.

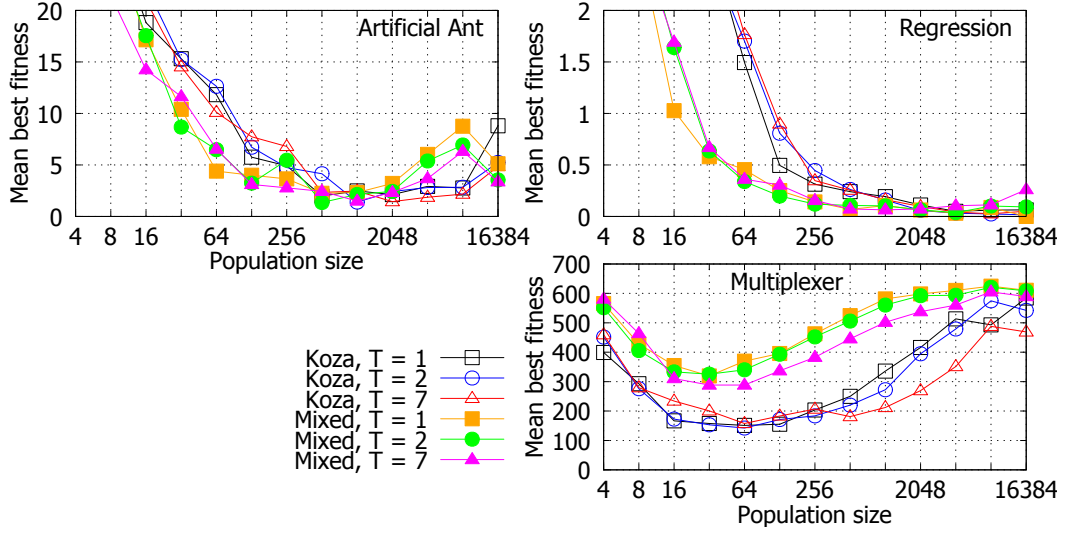


Figure 5.6: GOMGP: Mean best fitness across various population size and for various genetic operations and Tournament sizes.

5.2.7 Comparison of the Best Parameter Combinations

In this section we compare the best parameter combinations for all heuristics. The parameters are determined per experiment and are listed in Table 5.3. For reference we also plot the performance of Classic GP. We have also tested Tabu Search alone, but the performance is very low. We continue to vary the population size, since we observed that it has the most influence on performance.

Heuristic	Artificial Ant	SF	Symbolic Regression	SF	11-bit Multiplexer	SF
CGP	Koza operations	159	Koza operations	300	Koza operations	106
HC	Mixed operations	246	Mutation alone	295	Mixed operations	216
SA	Mixed operations	240	Mutation alone	246	Mixed operations	38
GATLE	Mixed, $\delta=0.3$	269	Mixed, $\delta=0.3$	374	Mixed, $\delta=0.3$	41
GATSE	Mixed, $\delta=0.3$, P_{gatse}	225	Mixed, $\delta=0.3$, P_{gatse}	359	Mixed, $\delta=0.3$, P_{gatse}	112
GOMGP	Percentual, STS, Mixed, Leq, T = 7	103	Percentual, STS, Mixed, Leq, T = 1	329	Percentual, STS, Koza, Leq, T = 1	16

Table 5.3: Best parameter combinations for all heuristics, determined per experiment. The SF column contains the number of solutions found, out of the total of 650 (13×50) experiments for every parameter combination.

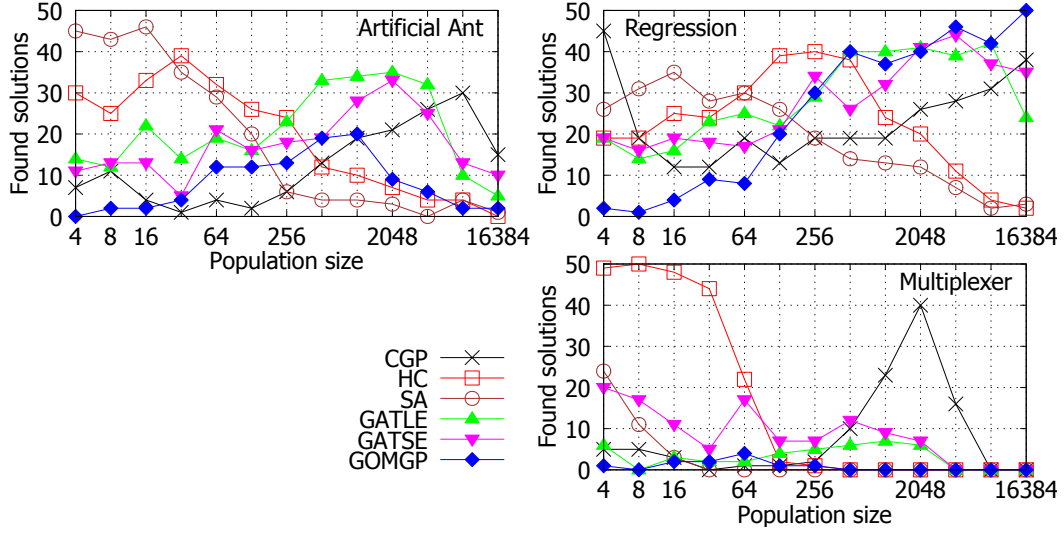


Figure 5.7: Number of solutions found across various population sizes for best parameter combinations for all heuristics, determined per experiment.

Number of Solutions Found

In Fig. 5.7 we plot the number of solutions found by each heuristic. The results are aggregated in Table 5.3 in the SF columns. We observe that for most of the heuristics, finding solutions largely depends on the population size. However, the behaviour of GATLE and GATSE is much more stable with respect to population size. This means that the success of these two heuristics is less dependent on the population size parameter. Except for the 11-bit Multiplexer experiments, GATLE and GATSE manage to find the largest number of solutions.

Mean Best Fitness and Standard Deviation of Best Fitness

The mean best fitness and standard deviation of best fitness are plotted in Fig. 5.8. We observe that in most of the cases, CGP has a good performance only for large population sizes. Conversely, HC and SA showcase very good performance for small population sizes. GATLE and GATSE, to some extent, mix the two behaviours, although with weaker performance for the 11-bit Multiplexer experiment. GOMGP has very good performance for the Symbolic Regression experiments with large population sizes. All around, the HC heuristic has the best performance with respect to the mean best fitness and also to the number of solutions found.

The standard deviation plots show that the HC and SA heuristics have the lowest best fitness variance. CGP, GATLE and GATSE show almost the same trend in variance except for the Symbolic Regression experiments where CGP has a higher variance. GOMGP has a high variance for small

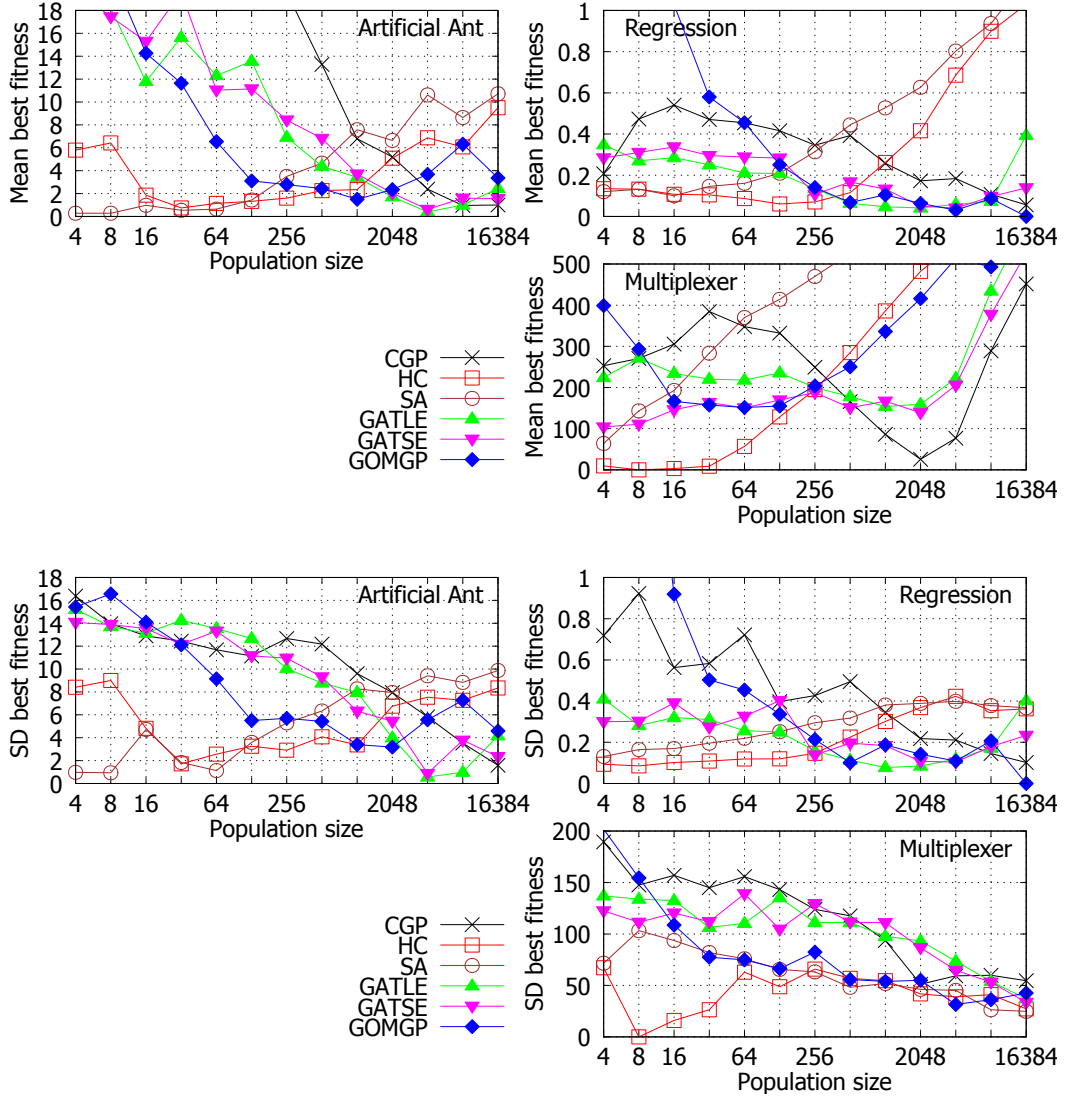


Figure 5.8: Mean best fitness (top) and standard deviation of best fitness (bottom) across various population sizes for best parameter combinations for all heuristics, chosen per experiment.

population sizes, most probably due to premature convergence. After this stops occurring, the variance has lower values than CGP, GATLE and GATSE in most of the cases.

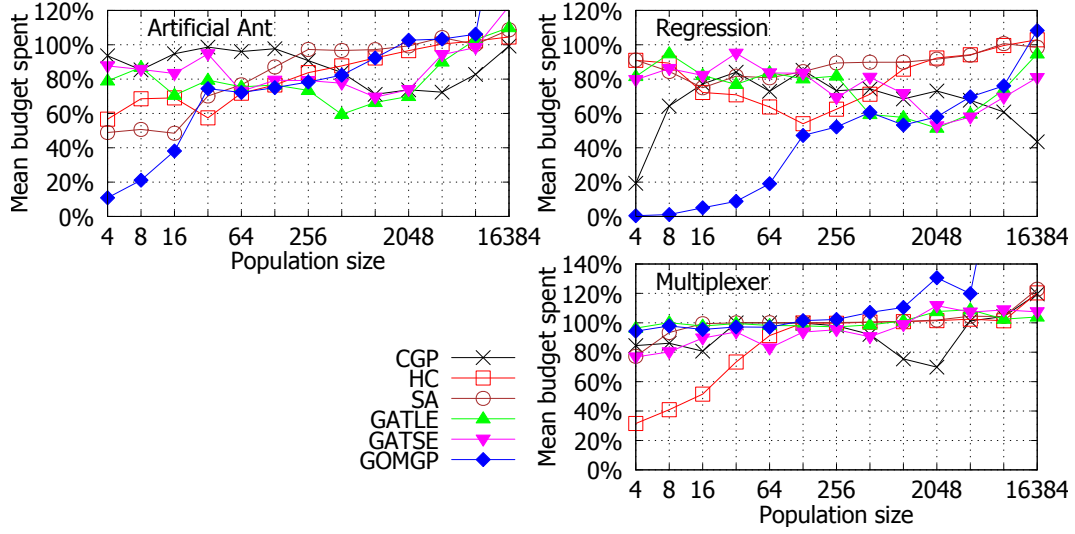


Figure 5.9: Mean computational budget (out of the 50.000 program executions budget) consumed per experiment across various population sizes for best parameter combinations for all heuristics, chosen per experiment.

Mean Computational Budget Consumed per Experiment

The mean computational budget consumed per run is plotted in Fig. 5.9. This metric tells us, how *hard* it is for a heuristic to find solutions. This is not the case for GOMGP, which can stop a run even if its allocated computational budget was not consumed (due to convergence). Also because GOMGP’s generations have a low granularity², and because a run is stopped only at the end of a generation, it can happen that GOMGP exceeds the allocated computational budget for large population sizes. We also observe that the Mutation operator alone, from HC and SA in Symbolic Regression for small population sizes, consumes almost all the computational budget, yet it delivers very good performance. This is not the case with the Mixed operations, which can deliver very good performance with using half of the computational budget, shown by SA in Artificial Ant and by HC in the 11-bit Multiplexer.

A very odd result is the performance of CGP, in the Symbolic Regression experiments for population sizes of 4 and 8, which is able to find a lot of solutions and while consuming a small portion of the computational budget. However, in the experiments when a solution is not found, the best fitness is large. This is shown by the standard deviation plot.

²A generation in GOMGP usually implies more budget use due to the multiple improvement attempts

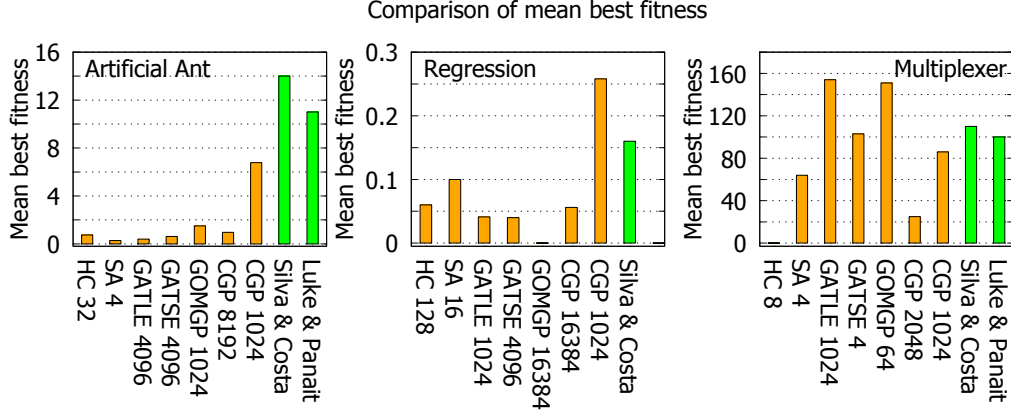


Figure 5.10: Comparison of mean best fitness between best parameter combinations of each heuristics, chosen per experiment (orange), and heuristics from the literature (green).

5.2.8 Comparison with State-of-the-Art Heuristics

In this section we compare the performance of the best parameter combinations for each of our heuristics with state-of-the-art heuristics, i.e., from Silva & Costa [88] and Luke & Panait [60]. In both papers, the computational budget per experiment is set to 50.000, the population size is set to 1000 and the Tournament size is set to 7. Furthermore, the authors use Crossover with a 90% probability and Reproduction³ with a 10% probability and no Mutation. The reason for not using mutation is not stated. The authors use the same set of experiments and the same fitness function as we use. We have selected the heuristics with the lowest mean best fitness. The heuristic from Silva & Costa uses a program size limitation mechanism similar to the one we use. The heuristic from Luke & Panait uses the Tarpeian method. The values are obtained through visual interpretation from the graphs. In case of the Symbolic Regression experiment from Luke & Panait, we could not read the best value with enough precision, due to the inadequate graph scale.

The comparison is shown in Fig. 5.10. On the X-axis, we show the name of the selected heuristic and the population size. The parameters for each of our heuristics are the ones listed in Table 5.3 (best parameter combinations). The CGP heuristic with a population size of 1024 is the closest, in terms of parameters, to the heuristics from the literature so it is also present in our graph. We observe that in two of the experiments CGP performs better than the heuristics selected from the literature. The differences may arise from the presence of Mutation and also from the specific operation of the Crossover operator. When it comes to the best heuristic for each specific

³Copying of the individual to the next generation unchanged.

problem, our heuristics are vastly superior. In the case of “GOMGP 16384” and “HC 8”, solutions were found in all the experiments. We do note, however, that we had performed a much more thorough parameter tuning (focused on performance) than in the related papers, which may explain the large performance differences.

5.3 Discussion

5.3.1 Parameters Common to All Heuristics

In the experiments performed in this chapter we observed that our main heuristic performance indicator (M1: mean best fitness) strongly depends on **population size** and the **genetic operations** used (defined in Section 3.2.3): Mutation-only, Koza and Mixed. In case of Mutation-only (used in HC and SA heuristics), we observe that it performs the best for small population sizes. This is because the heuristic has more chances (i.e., generations) to succeed improving each and every individual (than with larger population sizes). With more generations, there is a higher chance of discovering the series of mutations that lead to a solution.

At the other extreme, Koza-style operations deliver best performance for large population sizes. This is due to the Crossover operator and its ability to harness the large population diversity that is achieved given a large population. The Mixed operations combine Mutation-only and the Koza operations, by applying Mutation after every Crossover. The performance is better than Mutation-only at large population sizes and better than Koza operations for small population sizes. Furthermore, for some problems (“Artificial Ant”, “11-bit Multiplexer”) it is better than Mutation-only even for small population sizes. This makes it, for some problems, an *all-around* good genetic operation.

In the HC, SA and GOMGP heuristics, **Tournament selection** is used by the Crossover operator to select the second parent. When the population is small, it is better to have a small tournament size. Otherwise, there is the risk of selecting a small set of programs, out of the whole population, thus reducing diversity. In general, diversity is a must-have ingredient when using Crossover. Intuitively, the tournament size should grow with the population size, such that a selection of good individuals is performed effectively.

5.3.2 Parameters in GATLE and GATSE

In Section 5.2.2 we test the influence of the **generation gap** (defined in Section 3.4.1) on the performance of GATLE and GATSE. The influence observed is small, but as a general rule the higher the generation gap, the better the search performance. This is explained by the fact that a small generation gap can interrupt the improving of programs in very successful

generations, since only a small number of new programs are accepted per generation.

With respect to GATSE, we have tested in Section 5.2.3 two **acceptance probabilities** (for children worse than their parents): P_{gatse} and P_{sa} . The difference in performance between the two is minimal and with no clear winner. However, since GATSE is, in general, better than GATLE, we deduce that the most important aspect of accepting worse children is maintaining population diversity. Both P_{gatse} and P_{sa} succeed in doing that.

5.3.3 Advantages of GATLE and GATSE

We have seen in Fig. 5.7 (number of solutions found) and also in Table 5.3 (best parameter combinations) that GATLE and GATSE manage to find solutions once every three MetaCompiler runs for the “Artificial Ant” problem and once every two MetaCompiler runs for the “Symbolic Regression” problem, almost independent of the population size. The robustness of the heuristics to the variation of the population size and also to the generation gap and acceptance probability (in case of GATSE) constitute an advantage, because setup is easier.

5.3.4 Tree Linearization and Node Selection Strategies in GOMGP

In GOMGP, in each generation, every program has to be improved once, and there are several attempts to do this (in our experiments we used 10 attempts). Each attempt is focused on specific tree-nodes in the program’s linearized tree. The basic idea behind “size-adapted” **node selection strategy** in GOMGP is to select nodes spread across the entire tree. We expected that having the attempts start close to the program’s tree root and continue progressively to the leaf level represents a very good strategy. And conversely, if this ordering is not followed the performance will deteriorate. However, the fact that DFS, BFS and STS **tree linearization strategies**, perform very similar (as it is shown in Section 5.2.4), suggests that our expectation is not met. The reason is that, after a program is improved, by design GOMGP stops trying to further improve that program in the current generation. This creates a bias to improvement attempts that are focused on parts of the tree close to the root. This explains the good performance shown in the “Symbolic Regression” experiment, for which solutions have small trees.

5.3.5 GOMGP Halting Properties

We have seen in Section 5.2.5 that the “Leq” improvement condition yields the best search performance of GOMGP. However, this comes at the price

of potentially losing its halting properties. In order to study this effect we define the following metrics:

- M5 = Mean computational budget spent in halted runs (percentage of 50.000). Note that such a run ends not because a solution was found, nor because the computational budget was consumed
- M6 = Number of halted runs (percentage out of 50)

In Fig. 5.11 we plot the values for metrics M5 and M6. The statistics are gathered from the experiments performed in Section 5.2.5. As expected, the “Leq” condition greatly prevents halting compared to using the “Lt” condition. This is observed especially in the “11-bit Multiplexer” experiment. The reason is that it is easy to create a program having the same fitness as its parent, due to changes performed in the inactive code of a program. We tried to mitigate this effect by using a Tabu list, following the hypothesis that the genetic process produces the same programs over and over again. However, that is not the case, since it is clearly shown that the Tabu list is not effective in halting GOMGP.

On the other hand, the “Lt” improvement condition halts GOMGP much too fast for population sizes below 128. It, essentially, does not allow enough generations for solutions to be developed. As a consequence, GOMGP with “Lt” is able to find solutions only for high population sizes.

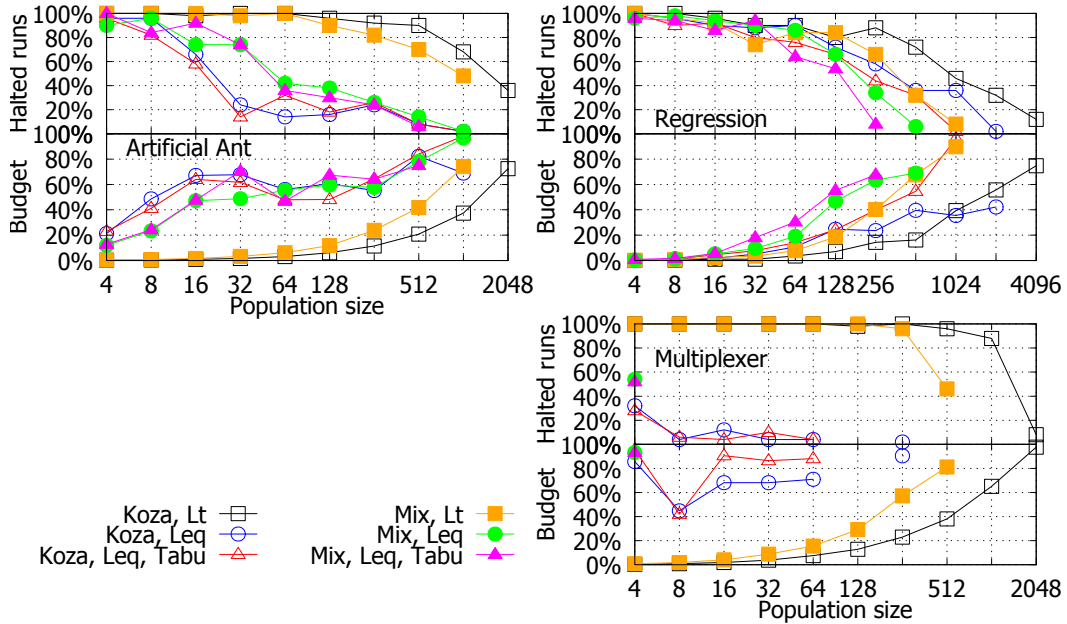


Figure 5.11: GOMGP: Halting statistics of GOMGP. Values plotted for parameter combinations that converge *at least* once.

Fig. 5.11 also shows us that the Mixed genetic operations delay the halting more than the Koza operations. This is because the improvement condition is satisfied for often child – parent pairs. This means that the Mixed operations find better programs (with respect to the improvement condition) easier.

5.3.6 Advantages of GOMGP

Our analysis has shown a potential flaw in the design of GOMGP, which makes it biased toward evolving solutions with small trees. However, this bias makes GOMGP especially good for problems such as “Symbolic Regression”, which have short solutions. Furthermore, the solutions are discovered using (on average) half the available computational budget (see Fig. 5.9).

5.4 Choosing the Best-Suited Heuristic

From what we have seen in Section 5.3 all heuristics have a population size interval in which they showcase very good performance. However, the scenarios for which we want to use these heuristics feature a long execution time for a given program [92]. Assuming that the simulation of one program (using the NetLogo library) takes about 1 second on an eight-core machine. Assuming there are 7 simulations per program. That means for a population size of 2048 (CGP good performance) one generation finishes in four hours. Conversely, it is more practical to have short-duration generations, since we would like to obtain some results as early as possible. For this reason heuristics that only perform well for high population sizes are unsuited.

From another point of view, most of the problems we tackle resemble a hybrid between the “Artificial Ant” and the “Symbolic Regression” experiments. A mathematical expression has to be discovered, sometimes with the presence of conditional statements. This statement is based on the experience with the problems in Chapter 6. Therefore, it makes sense to choose the heuristics that perform best for these two benchmarking problems and with small population sizes. Having these two criteria in place, from our benchmarking results, the best suited heuristic is **Simulated Annealing** with a population size of **8**. Furthermore, the **Mixed genetic operations** should be chosen since they showcase a better performance when searching for both short and long solutions. An alternative is represented by the **Hill Climber** heuristic with a population size of **32**. In case we search for programs composed of multiple nested conditional statements (such as the “11-bit Multiplexer”), the **Hill Climber** heuristic with a population size of **8** represents the best option.

Chapter 6

Algorithm Discovery Case Studies

In this chapter we document our experiments for discovering and improving algorithms for large-scale networks. In the first part, we showcase how to develop new algorithms for the maximal independent set formation problem and for the problem of event localization using graph spectral theory. In the second part of the chapter, we improve two recent algorithms: *FailDetect* [80] and *ChurnDetect* [79]. For every case study we list the building blocks that we made use of and present their specific properties with respect to the influence they have on agent’s behaviour. The taxonomy used for classifying the building blocks is described in Table 6.1.

Building block type	Description
1-hop	Collects information from 1-hop neighbors
local	Uses only local information (from itself)
condition	Takes a decision based on a condition
constants	A constant such as a float or an integer
mathematical	Contains a mathematical expression
historical	Uses information from the past
filter	Keep only elements that respect a condition

Table 6.1: Building block types.

6.1 Maximal Independent Set Formation

In this section we describe the “maximal independent set formation” problem, the experimental setup and the building blocks that are used to solve it. Then, we present the fitness function and finally we list and evaluate the found solutions.

6.1.1 Problem Description

The problem is to create a maximal¹ set of nodes that are independent of each other. A dependency between two nodes can be caused by a shared resource (e.g., access to a radio channel, a sensor, etc.). The dependencies between nodes are modelled with links (arrows). We use the following convention: blue nodes are part of the independent set and green nodes are not part of the set, see Fig. 6.1. The set of dependencies between nodes changes over time.

We use the following rules to test if the set of blue nodes represents a maximal independent set:

1. Every green node should have at least one blue neighbor
2. Every blue node should have only green neighbors
3. Every node without any neighbors should be blue

The MetaCompiler has to discover an algorithm (executed by every node) that uses only information from 1-hop neighbors, such that all of the above rules are respected.

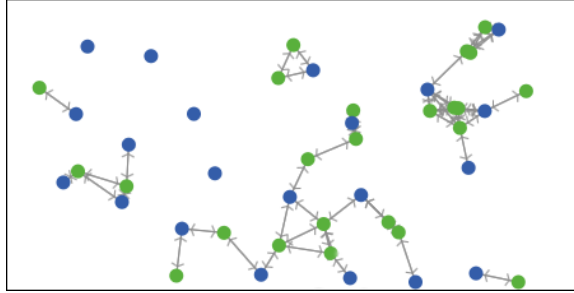


Figure 6.1: An example of a maximal independent set.

6.1.2 Experimental Setup

The simulation features a 2D world of size 5000×5000 space units (s.u.) with wrap-around world boundaries (toroidal world). The world contains 50 mobile agents, initially placed at random, with a transmission range of

¹A *maximal* set is a set to which no other node can be added such that the condition still holds. The *maximum* set is the largest maximal set for which the condition holds.

100 s.u. Every agent represents one node in the graph. When the distance between two agents is smaller than the transmission range, they form a link (thus marking a dependency). The average node degree of the network graph formed is $d \simeq 7$. The mobility model of the agents is random walk with a speed of $m = 5$ s.u./tick. With it, we simulate the dependency changes. The simulation is executed for 100 ticks.

With this case study we would like to test the capabilities of the Meta-Compiler to figure out solutions with different levels of problem knowledge incorporated in the building blocks. For this, we use three sets of building blocks (see Table. 6.2) corresponding to three difficulty levels. At the lowest difficulty level we use more specific building blocks (`CountWinnerNbr()`, `CountInsideNbr()`), created with some prior knowledge of the solution. At the medium level we replace the `CountInsideNbr()` block with a more generic building block (`CountNbrFilter("b")`). At the highest difficulty level we use only the generic building block. The higher difficulty is caused by the increase of the search space created by the use of a generic block and by the lack of problem knowledge incorporated in the set of building blocks.

Block name	Block functionality	Block type
<code>CountNbrFilter("b")</code>	Takes as parameter a "boolean as string" and returns the number of neighbors for which the condition is true	1-hop, filter
<code>CountInsideNbr()</code>	Returns the number of neighbors that are inside the set (neighbors having <code>inside</code> ≥ 1)	1-hop, filter
<code>CountWinnerNbr()</code>	Returns the number of neighbors that: (i) do not have any neighbors in the set and (ii) have a higher ID than the calling agent or are in the set	1-hop, filter
Read/write variable	<code>inside</code> : initialized at the beginning of the simulation to 0. Every agent has an unique numeric ID.	local
<code>ID()</code>	When it is part of the argument given to <code>CountNbrFilter</code> , it returns the ID of the neighbor. Otherwise, returns the ID of the calling agent.	local
<code>myID()</code>	Always returns the ID of the calling agent.	local
<code>storeResult(n)</code>	Store n in a hidden variable	local
<code>getResult()</code>	Return the stored result	local
If and If-Else	Branching statements	condition
Constants	0 ... 99; 0.01 ... 1.00	constants

Table 6.2: Building blocks and variables used in the "maximal independent set formation" case study.

Listing 6.1: Function counting how many nodes do not respect rules 1 – 3.

```
1 to-report tickFitness
2   report count agents with [inside < 1 and
3     (count link-neighbors with [inside >= 1] = 0)] +
4   count agents with [inside >= 1 and
5     (count link-neighbors with [inside >= 1] > 0)] +
6   count agents with [inside < 1 and (count link-neighbors = 0)]
7 end
```

Listing 6.2: Maximal independent set formation – Solution A.

```
1 ask agents [
2   set inside 0.0
3   if countNbrFilter ( "inside > 2" ) < 0.19
4     [ set inside 47 ]
5 ]
```

6.1.3 Fitness Function

The fitness of a program is defined as following: the number of agents that do not respect rules 1., 2. and 3. described in Section. 6.1.1. The NetLogo code for computing this number is shown in Listing 6.1. It is based on a local variable named `inside`.

Programs also have to discover the rule that an agent with `inside < 1` is not in the set while one having `inside >= 1` is in the set. At the end of every tick, the value returned by `tickFitness` is stored in a list. We compute the fitness as the mean of the last 50 values stored (i.e., fitness between ticks 51 and 100). Each program is executed 7 times and the average of the set of results is selected as the program’s fitness.

6.1.4 Found Solutions

We have observed that it is fairly easy for the MetaCompiler to discover solutions for this problem, i.e., almost half of the runs succeed finding solutions for the highest difficulty setup. One concise solution we found is shown in Listing 6.2.

The code works as follows: at the beginning of every tick, every agent assumes it is not in the set, and then if it does not have any neighbors that are in the set, it joins the set. This solution works because the code within the `ask agents` primitive is executed as a critical section (no two agents execute the code at the same time), and the set is stable because the agents reset their `inside` value only when they start executing their code block. In the first tick, the arbitrage is performed simply by the `ask` primitive,

Listing 6.3: Maximal independent set formation – Solution B.

```
1 ask-concurrent agents [
2     ifelse (countNeighborsFilter ( "inside = 1" ) != 0 )
3     [ set inside 0 ]
4     [ set inside 1 ]
5 ]
```

Listing 6.4: Maximal independent set formation – 2 rounds skeleton.

```
1 ask-concurrent agents [ storeResult <take decision> ]
2 ask-concurrent agents [ set inside getResult ]
```

which picks one agent at a time to execute its code. If we replace **ask** with **ask-concurrent** the fitness is still 0, but the maximal independent set is not stable anymore, i.e., the components of the set change even if the link graph did not change.

We modify the **tickFitness** function to also include the number of agents that enter or exit the set without any change in their set of neighbors (these agents are called “bouncers”). We restart the search with this new function and obtain the solution in Listing 6.3 (code is minimized manually).

Again, this solution finds a way to exploit the fact that in NetLogo statements are executed in an “atomic” manner. These types of solutions act as powerful “attractors” in the search space. To prevent this type of solutions, we split one tick into two rounds: (i) all agents take the decision to be or not to be in the set; (ii) all agents make their decision public. We implement round (i) with a procedure named **storeResult** that gets as parameter the value of **inside**. We make **inside** read-only. In round (ii), **inside** gets the stored result. Only after round (ii) **tickFitness** is called. The skeleton of the programs is shown in Listing 6.4. In this way agents find out about their neighbors’ decisions only after all decisions (to be or not to be in the set) have been taken.

This approach is unsuccessful, because it does not provide a way of selecting a single agent to join the set out of multiple mutually neighboring agents that can join the set. In other words, with the current set of building blocks, all agents have equal rights to join the set. One way of comparing two agents is to use their IDs (see **ID()** and **myID()** in Table 6.2). Using these two new building blocks, we obtain two solutions (C and D) with very small fitness ($\simeq 0.008$). The code for solution C is shown in Listing 6.5 (code is minimized manually). We do not list solution D, due to space constraints.

Solutions C and D work on the assumption that the agents are synchronized. Every tick has two parts: taking the decision, and making the decision

Listing 6.5: Maximal independent set formation – Solution C.

```

1 ask-concurrent agents [
2   ifelse inside = 0
3   [ ifelse countNbrFilter ("inside = 1") = 0
4     [ storeResult 1 ]
5     [ if countNbrFilter ("inside = 1 and ID > myID") = 0
6       [ storeResult 1 ]
7     ]
8   ]
9   [ if countNbrFilter ("inside = 1 and ID > myID") > 0
10    [ storeResult 0 ]
11  ]
12 ]
13 ask-concurrent agents [ set inside getResult ]

```

public. The solutions manage to find a decision tree, based on the IDs and their neighbors (one tick into the past) decisions. However, these solutions do not have a perfect fitness due to the mobility and the convergence time of more than one tick. We analyse the effect of mobility on solutions C and D in Fig. 6.2. We executed 100 runs for every solution and every mobility speed.

The results show clearly that the fitness increases with the mobility speed. Nevertheless, the median values are in the interval $[0.05, 0.4]$. To give a better intuition of the fitness values, a fitness of 1 means that during the last 50 ticks of the simulation, in every tick one agent out of the total of 50 did not obey the rules in Section 6.1.1. Therefore, a fitness between 0.05 and 0.4 shows good accuracy of the algorithm.

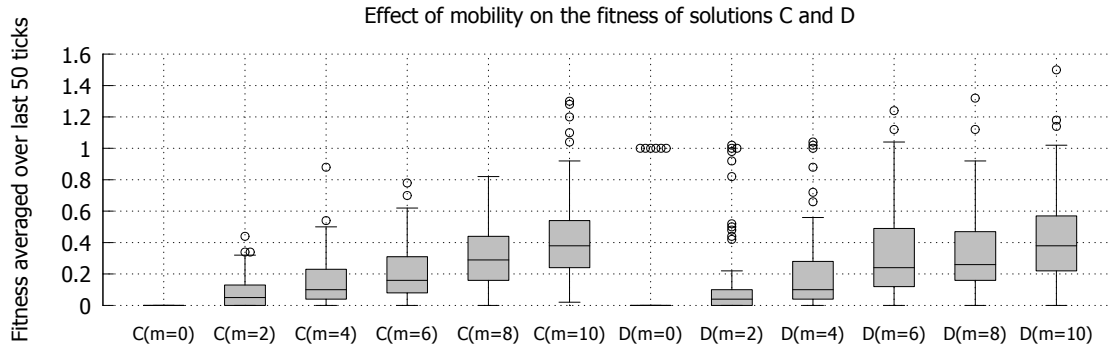


Figure 6.2: Fitness of solutions C and D for various mobility speeds.

6.1.5 Discussion

We have seen in this case study that the MetaCompiler can discover algorithms for the maximal independent set formation problem. Some of the discovered solutions are too simple, i.e., are viable within a strong set of assumptions, the strongest one being that agents take turns in executing their code. The simple solutions act as attractors since they have a fitness of 0.

In order to prevent this type of solutions we had to split every tick into two rounds. In this way, agents find out simultaneously about the decisions (of being or not being in the set) of their neighbors. Using this method, two neighboring agents can compare their IDs and decide which one of them joins the set. The solutions found with this method are not perfect as the convergence takes more than one tick. Nevertheless, they show very good accuracy for small mobility speeds.

As an important note, the requirement that at every tick, blue nodes should form a maximal independent set (i.e., have a convergence time of 1 tick) can be considered overambitious. In real-life networks, communication and computation happens orders of magnitude faster than the rate of change of the network graph. We retested solutions C and D with movement being performed every 10 ticks and the fitness being collected only in ticks before movement is performed. Indeed, with this modification, the fitness is always perfect and not depending on the mobility speed. In most of the cases, it takes only 2 to 4 ticks for the algorithm to converge.

The first set of solutions (A and B) reveals a limitation of NetLogo: it cannot easily simulate race conditions and delays in communication (which usually makes problems much more complicated). Furthermore, it is hard to identify the simplifying assumptions introduced by NetLogo, and whether the discovered algorithms use them or not.

6.2 Event Localization Using Graph Spectral Theory

In this case study we use the MetaCompiler to discover filters described in the spectral domain. First, we present the theory behind our approach, then we describe the experimental setup, together with the set of building blocks we use, and finally we analyse the discovered solutions.

6.2.1 Problem Description

There is an inherent interest in the design of decentralized filters for data defined on networks / graphs. For example, given a Wireless Sensor Network (WSN) deployment, with sensors that can measure temperature, the challenge is to use such a WSN to localize events (with clearly defined borders), such as fires, in a decentralized manner. Every node should know whether it is in the event area, outside of it or on the boundary. This is made difficult by noise in the sensed values. High frequency noise can be caused by imperfect sensors; medium frequency noise by sun – shade; and low frequency noise by day – night temperature shifts. In order to overcome this and identify signal trends, a decentralized filter has to be used.

Preliminaries

A typical event region detector is the combinatorial Laplacian [55]. Given a real-valued signal x (e.g., the sensed temperature), every node u in the network computes the Laplacian as

$$(Lx)(u) = \sum_v^{N_u} (x(v) - x(u)), \quad (6.1)$$

where N_u represents the set of nodes adjacent to node u . Each node u can decide whether it is on the boundary of an event area, by comparing the sign of its (Lx) value with the sign of (Lx) of its neighbors. Different signs mean the node is on the boundary. The Laplacian event detector is very sensitive to noise and cannot decide if flat network regions (having nodes with identical x values) are part of an event or not. This is usually overcome by using diffusion (of the sensed value). A typical diffusion operator is the Gaussian, defined by

$$(Gx)(u) = \frac{x(u) + \sum_v^{N_u} x(v)}{|N_u + 1|}. \quad (6.2)$$

By applying the Gaussian several times (e.g., 10) before the Laplacian, sudden signal transitions are smoothed and the flat regions become curved. We call an algorithm, such as “Laplacian of Gaussian”, a filter.

Spectral Framework

Similarly to signal processing, a filter can be described by how much it amplifies / attenuates different signals. For example, the response of a filter to an impulse signal reveals the size of the largest detectable event regions. However, the impulse response depends on the topology of the geometric graph (number of nodes, average node degree), the filter is applied on. One way to decouple the filter from the topology is to describe it as an operator in the spectral domain spanned by the eigenfunctions of the Laplacian [55]. This is similar to a Fourier-like decomposition, extended, however, to irregular graph topologies / spaces.

In order to test the response of such filters, a “spectral framework” for event detection is developed using the methodology described by Loukas et al. in [55]. The main idea is that given a linear laplacian filter F , one can estimate F ’s response $r(\lambda_k)$ by giving as input to the filter the eigenfunctions ψ_k of the Laplacian. Given a signal $x : N \rightarrow \mathbb{R}$, where N is the set of nodes, the output of a linear graph filter F can be written as

$$Fx = \sum_{k=1}^n r(\lambda_k) \langle x, \psi_k^\top \rangle \psi_k. \quad (6.3)$$

Then, by giving as input to the filter F the eigenfunctions ψ_m of the Laplacian and due to orthogonality of base $\langle \psi_m, \psi_k \rangle = 0$, we obtain

$$F\psi_m = r(\lambda_m)\psi_m. \quad (6.4)$$

The process is shown in Fig. 6.3.

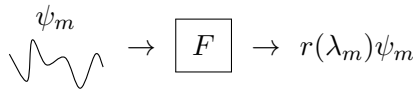


Figure 6.3: A method for obtaining the response of a filter F .

When F operates on input ψ_m , the magnitude of its output is its response. Since $\|\psi_m\| = \langle \psi_m, \psi_m \rangle = 1$ we obtain the response of the filter as

$$\|F\psi_m\| = |r(\lambda_m)| \|\psi_m\| = |r(\lambda_m)|. \quad (6.5)$$

We show in Fig. 6.4 (top row), the responses of several filters developed by Loukas et al. in [55], which are made out of the (Gx) and (Lx) operators. In this case study we want to discover new filters with different responses than the ones in Fig. 6.4. In the next sections, we describe the experimental setup and the set of discovered filters.

Block name	Block functionality	Block type
Laplacian(), L()	Applied on node u in iteration $t + 1$, returns $\sum_v^{N_u} (x_v^t - x_u^t)$, where N_u represents the set of neighbors of node u	1-hop, math
Gaussian(τ), G(τ)	Applied on node u in iteration $t + 1$, returns $(x_u^{\lfloor t-\tau \rfloor} + \sum_v^{N_u} (x_v^{\lfloor t-\tau \rfloor})) / (N_u + 1)$. If $\tau \leq 0$, returns Gaussian(0). If $\tau \geq t$, returns Gaussian(t)	1-hop, math, historical
GaussianCurrent(), GC()	Returns Gaussian(0)	1-hop, math
NeighborsAvg(), NA()	Applied on node u in iteration $t + 1$, returns $\sum_v^{N_u} (x_v^t) / N_u $	1-hop, math
CountNeighbors()	Applied on node u , returns $ N_u $	1-hop
WeightSum(c, d, α)	Returns $\alpha c + (1 - \alpha)d$	math
Plus(c, d)	Returns $c + d$	math
Minus(c, d)	Returns $c - d$	math
Division(c, d)	Returns c/d . If $d = 0$, returns 0	math
Mul(c, d)	Returns $c \times d$	math
Sqrt(c)	Returns $\sqrt{ c }$	math
Constants	0 ... 99; 0.01 ... 1.00	constants

Table 6.3: Building blocks used for the “Event localization using graph spectral theory” case study. c , d and α have a numerical data type.

6.2.2 Experimental Setup

The simulation contains n static nodes (i.e., nodes do not change their neighbors). Nodes and the links between them define the network graph also called a geometric graph. Let $\mathcal{L}_{n \times n}$ be the normalized Laplacian matrix of the graph defined as

$$\mathcal{L}_{i,j} = \begin{cases} 1 & , \text{ if } i = j \text{ and } d(u_i) \neq 0 \\ -\frac{1}{\sqrt{d(u_i)d(u_j)}} & , \text{ if } u_i \text{ is adjacent to } u_j \\ 0 & , \text{ otherwise} \end{cases} \quad (6.6)$$

where $d(u)$ is the node degree of node u . Then, let λ_k be the eigenvalues and ψ_k be the eigenvectors (with size n) from the eigendecomposition of $\mathcal{L}$, with $k = 1 \dots n$. The matrix operations are performed by a custom NetLogo extension, which we developed in Java.

The spectral response r of the distributed filter is obtained as follows. The simulation lasts for n rounds. We iterate over the ψ_k eigenvectors, therefore

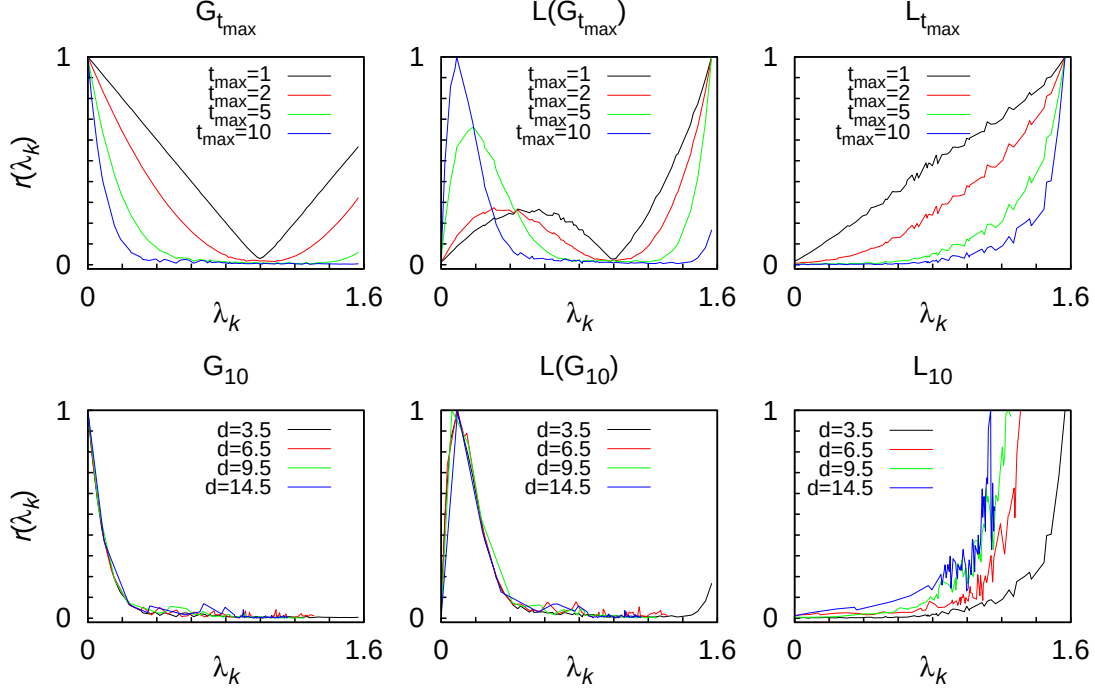


Figure 6.4: Top: spectral responses for different number of maximum iterations (t_{max}) on graphs with average node degree $d = 3.5$. Bottom: spectral responses for graphs with various values of d , for $t_{max} = 10$.

the number of rounds is equal to the number of eigenvectors and the number of nodes. At the beginning of round k , we iterate over the values of ψ_k and give the value $\psi_{k,i}$ to node u_i as the input value x_i^0 (e.g., the sensed value at $t = 0$). In each round, every agent repeats a block of code (the filter) t_{max} times (in our experiments $t_{max} = 10$). One repetition is called an iteration of the filter. The output of the filter on node i at iteration t is stored in x_i^t . After the last iteration, the amplitude of the response of the filter applied to ψ_k is computed as $r(\lambda_k) = \sqrt{\sum_{i=1}^n (x_i^{t_{max}})^2}$. The filter is composed of building blocks described in Table 6.3, which can be nested. An empty filter will output x_i^0 . After n rounds, the response of the filter is normalized to the $[0, 1]$ interval. We try to minimize the number of non-linear building blocks, therefore we do not use max, min and conditional statements.

In Fig. 6.4 (top row) we plot, the spectral response of three simple filters made out of two basic building blocks. “ $G_{t_{max}}$ ” is made out of one GaussianCurrent() repeated for t_{max} iterations. In “ $L(G_{t_{max}})$ ” a Laplacian() is applied after the t_{max} iterations of GaussianCurrent(). “ $L_{t_{max}}$ ” is composed of one Laplacian() repeated t_{max} times. The response of these filters does not depend on the average node degree, as it is shown in Fig. 6.4 (bottom).

In each simulation, we apply the filter to 5 geometric graphs: G_1, G_2, G_3, G_4, G_5 with the average node degrees of: $d_{G_1}=3.5, d_{G_2}=6.5, d_{G_3}=9.5, d_{G_4}=14.5, d_{G_5}=19$. The simulation features 64 nodes, and the filter is applied with $t_{max} = 10$ iterations. We use only 64 nodes for practical purposes, because the number of filter applications ($5 \times n^2 \times t_{max}$) is in the order $O(n^2)$.

6.2.3 Fitness Function

The goal of this case study is to develop other filters that have a similar response for geometric graphs with different average node degrees, like the ones in Fig. 6.4 (bottom row). To identify such filters, we choose the fitness value as the area between the highest response curve and the lowest response curve. In order to compare the filter's responses we first need to normalize them. This is necessary because the λ_k values come from a discrete interval with different interval limits depending on the graph's average node degree, and are non-equidistant. The higher the node degree, the smaller λ_n (the largest eigenvalue) is. In order to calculate the fitness we take the following steps:

1. we truncate the responses r_{G_p} to the interval $[0, \lambda_n^{min}]$, where r_{G_p} is the response of the filter applied on $G_p, p \in \{1...5\}$ and $\lambda_n^{min} = \min_p(\lambda_n \text{ of } G_p)$,
2. we renormalize the responses to the interval $[0, 1]$, and
3. we linearly interpolate the responses such that, for each graph G the response is given by a continuous function

$$r_{G_p}^{tr}(y) : [0, \lambda_n^{min}] \rightarrow [0, 1]. \quad (6.7)$$

Since it is also interesting to study the filter's response with respect to the order of the eigenvalue, we interpolate a second function

$$r_{G_p}^{ord}(z) : [0, n] \rightarrow [0, 1]. \quad (6.8)$$

Because the number of λ_k values is constant, truncation and renormalization are not needed. Basically, r^{ord} considers the λ_k values as equidistant. The two areas with respect to r^{tr} and r^{ord} are given by f_{tr} and f_{ord} :

$$f_{tr} = \sum_{y \in \mathcal{Y}} \left(\max_{p=1}^5 r_{G_p}^{tr}(y) - \min_{p=1}^5 r_{G_p}^{tr}(y) \right), \text{ with } \mathcal{Y} = \{0, 0.001, \dots, \lambda_n^{min}\} \quad (6.9)$$

$$f_{ord} = \sum_{z \in \mathcal{Z}} \left(\max_{p=1}^5 r_{G_p}^{ord}(z) - \min_{p=1}^5 r_{G_p}^{ord}(z) \right), \text{ with } \mathcal{Z} = \{0, 0.001, \dots, n\} \quad (6.10)$$

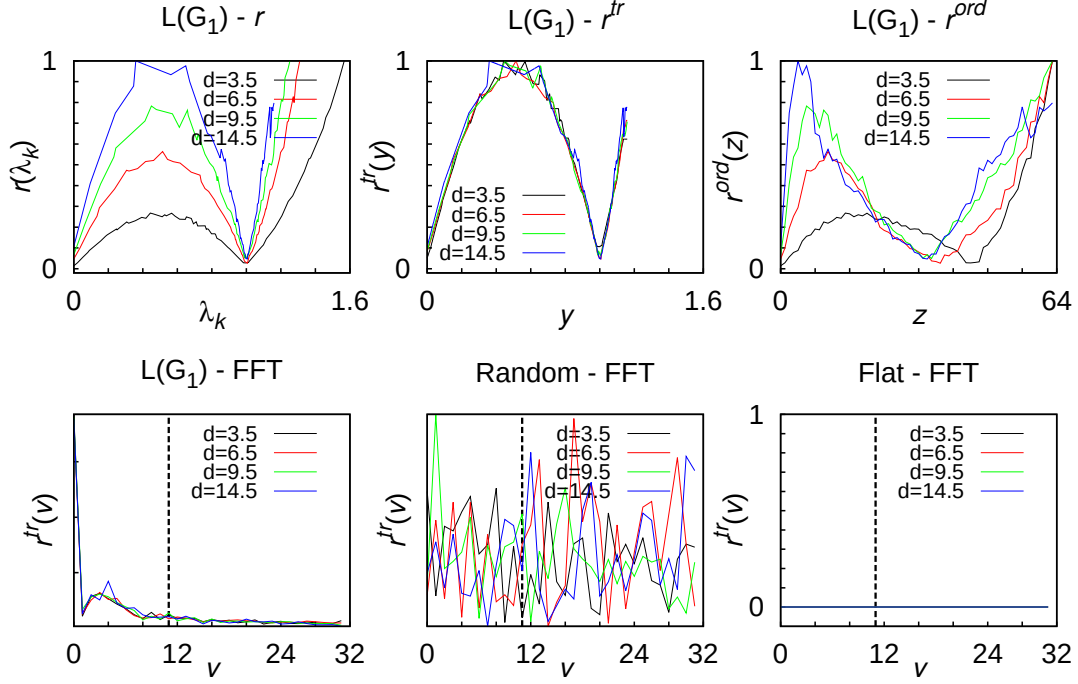


Figure 6.5: Top: spectral responses for graphs G_1 , G_2 , G_3 and G_4 with one iteration. Bottom: FFT for various filters. The vertical dashed line separates the first $1/3$ components.

Since we are searching for filters that have similar responses with respect to the r^{tr} or the r^{ord} functions, we compute the fitness value of a program as $f = \min(f_{tr}, f_{ord})$. A visual example of the effect of representing the responses r with $r^{tr}(y)$ and $r^{ord}(z)$ is shown in the top part of Fig. 6.5. We left out the response on G_5 because it was very similar with the one on G_4 .

We can immediately see that this fitness function represents an attractor for trivial solutions (e.g., a filter that always outputs 1). In order to punish flat and random spectral responses we compute the FFT of $r^{tr}(y)$ for 64 equidistant values of $y \in [0, \lambda_n^{min}]$. Then we require that (a) the sum of the first $1/3$ FFT components to be greater than the sum of the rest of the FFT components for every graph G_p . If not, a very high fitness value is assigned to the program. In the bottom part of Fig. 6.5 we plot the FFT components for three filters. The latter two show how we can recognize random and flat responses given condition (a).

6.2.4 Found Solutions

We discovered two types of filters. The first type has a very small area computed with f_{ord} value. The second type has a very small f_{tr} area.

Filters with very small area values computed with f_{ord}

In a first set of experiments we managed to find three filters. These were found using the $f = \min(f_{tr}, f_{ord})$ fitness function. We minimize the code of the first and second filter by hand, and we obtain the filters in Eq. 6.11 and Eq. 6.12.

Filter F1:

$$x^t = \text{NA}() \times \left(0.66 + \sqrt{|(90.59 - (\text{GC}() + 0.39) \times \text{NA}()) \times 0.1|} \right) \quad (6.11)$$

Filter F2:

$$\begin{aligned} x^t &= \text{G}((1 - \text{L}()) \times x^{t-1} - \text{L}() \times x^{t-1}) \\ x^t &= 30 \times (\sqrt{|x^t|} + 73) - x^t \times (72 + \sqrt{|x^t|}) \end{aligned} \quad (6.12)$$

The code for Filter F3 could not be made concise enough to fit on the page. Note that a filter is applied by every node for a number of $t_{max} = 10$ iterations. Filters F1, F2 and F3 allow only the 1st, 2nd and 3rd eigenvalue respectively to pass, with 2 to 5 orders of magnitude compared to the other eigenvalues. The responses are plotted in Fig. 6.6, top left. This is a very interesting behaviour, because a sharp response can be used for estimating the spectrum of a graph.

Upon closer inspection of the x^t values, we observe that the unnormalized $r(\lambda_k)$ are very big, in the order of 10^{50} . However, in the cases of $r(\lambda_1)$ of F1, $r(\lambda_2)$ of F2 and $r(\lambda_3)$ of F3, the r values are 2 to 5 orders of magnitude larger than the other $r(\lambda_k)$ values. Therefore, after the normalization step, it is just an illusion that only the respective $r(\lambda_s)$ are non-zero (with $s \in 1, 2, 3$). Nevertheless, it is interesting to study why only one eigenvalue is promoted even for graphs with different node degrees and cardinality. Our first intuition is that the combination of building blocks, being executed for specific values of ψ_s , make a more powerful increase of the x values of some nodes. This, in turn, creates very large $r(\lambda_s)$ values compared to the other $r(\lambda_k)$ values.

One way of avoiding this powerful increase of the x values of some nodes is to test for the convergence of the filter. Our approach is to minimize the difference between the filter's response at iteration t_{max} and the response at a previous iteration i.e., we add $\left| \sqrt{\sum_{i=1}^n (x_i^{t_{max}-1})^2} - \sqrt{\sum_{i=1}^n (x_i^{t_{max}})^2} \right|$ to the fitness value f .

Unfortunately, this does not prevent the same kind of solutions to be discovered. This is because the MetaCompiler finds a way to come up with very small x values for which the difference between the responses of the last two iterations is small, in the order of 10^{-20} . Now, all of the $r(\lambda_k)$ values will be very small, except for λ_s for which $r(\lambda_s)$ is a few orders of magnitude larger than the $r(\lambda_k)$ values. After normalization, we obtain similar results with the ones in the top left of Fig. 6.6.

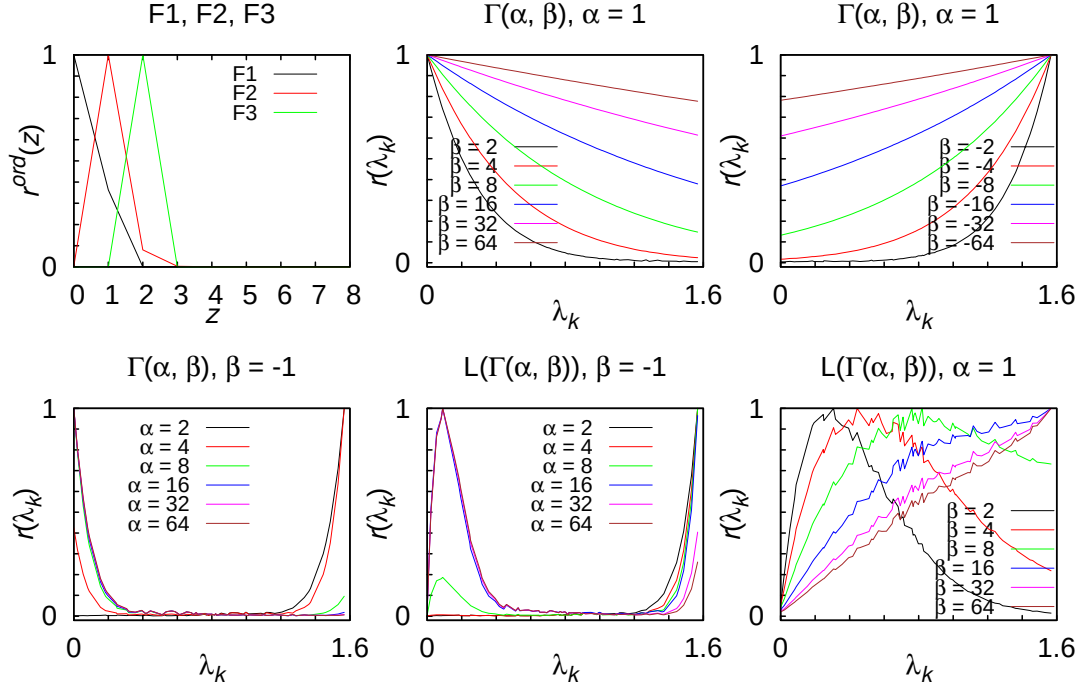


Figure 6.6: Spectral responses for various discovered filters. The filters are applied for $t_{max} = 10$ iterations.

Filters with very small area values computed with f_{tr}

The previous type of solutions represent strong “attractors” in the search space because their f_{ord} values are very small. Because of this, we make our fitness function $f = f_{tr}$ and do not consider the value of f_{ord} . With this fitness function we discover a second class of solutions, which are given in Eq. 6.13 and Eq. 6.14 (code manually simplified).

Filter F4:

$$x^t = \text{GC}() - 31x^{t-1} \quad (6.13)$$

Filter F5:

$$x^t = \text{GC}() + 24x^{t-1} \quad (6.14)$$

The responses are very interesting and, again, unlike any known response. We generalize filters F4 and F5, and obtain the following filters:

$$\begin{aligned} \Gamma(\alpha, \beta) : x^t &= \alpha \text{GC}() + \beta x^{t-1} \\ \text{L}(\Gamma(\alpha, \beta)) : &\text{Apply a Laplacian}() \text{ after 10 iterations of } \Gamma(\alpha, \beta) \end{aligned} \quad (6.15)$$

where $\alpha, \beta \in \mathbb{R}$. The responses of these filters for different values of α and β are shown in Fig. 6.6. We observe that for the $\Gamma(\alpha, \beta)$ with $\alpha = 1$, by changing the sign of β the response gets mirrored on the horizontal axis. This

is a powerful effect obtained by altering only one parameter. Furthermore, the response of $L(\Gamma(\alpha, \beta))$ with $\alpha = 1, \beta > 0$ is very similar to a mirrored version on diagonal of the response of “ $L_{t_{max}}$ ” filter from Fig. 6.4.

Some (α, β) combinations are not plotted. That is because their response is similar to the ones in Figure. 6.4:

- In the case of $\Gamma(\alpha, \beta)$ with $\alpha > 0, \beta = 1$, the response looks like the “ $G_{t_{max}}$ ” filter – α increase is the same as t_{max} increase.
- In the case of $L(\Gamma(\alpha, \beta))$ with $\alpha = 1, \beta < 0$, the response looks like the “ $L_{t_{max}}$ ” filter – β decrease is the same as t_{max} decrease.
- In the case of $L(\Gamma(\alpha, \beta))$ with $\alpha > 0, \beta = 1$, the response looks like the “ $L(G_{10})$ ” filter, with very little differences for α increase.

6.2.5 Discussion

With the help of the MetaCompiler we managed to come up with new and interesting spatial filters. We were able to generalize the discovered solutions and obtain filters with parameterizable responses. The results obtained and the set of observed behaviours give us inspiration for trying new linear combinations of basic building blocks such as the Laplacian and the Gaussian.

The experiments also tested the NetLogo experimental setup extensively. Some of the discovered solutions manage to exploit the very high range of double-precision floating point numbers and, thus, appear as filters with very sharp responses. One possible way to prevent this is to monitor the x values and penalize programs that have either too small, or too big values.

The *graphical* approach to computing the fitness function proved to be an efficient method for aggregating a complex set of responses into a single value. Furthermore, the usage of the FFT decomposition to avoid trivial solutions proved to be efficient and robust. Another possible way of identifying good filters is to analyse the first and second derivative of their response. A more general approach can be the use of image processing algorithms that detect the presence of user-specified features (e.g., a step) in the filter’s response.

For future work, we believe that the new set of filters with parameterizable responses opens the door to a new class of *adaptive filters*, which can react to their neighbors’ state by changing their filter’s parameters. It is certainly interesting, and may prove worthwhile, to explore this class of filters. Furthermore, one new building block that should be implemented is an extended $\text{WeightSum}(c, d, \alpha)$ that adds more than two terms, each with a custom weight (the sum of the weights equals 1). By following these two ideas (i.e., adaptive parameters, complex linear combinations), we want to focus our search on discovering “low-pass”, “high-pass” and “band-pass” filters with very steep slopes.

6.3 FailDetect Algorithm

In this section we describe our experiments for improving the *FailDetect* [80] algorithm. We start by briefly describing the original algorithm, then we present our experimental setup together with the used building blocks. Finally, we evaluate some of the best programs found.

6.3.1 Problem and Algorithm Description

The FailDetect algorithm is a fully distributed scheme for online estimation of the average packet loss in a network. This is accomplished without the nodes advertising their packet transmission success rate. The algorithm is based on gossip-like communication and makes the same assumptions as the ones used by the Push-Sum Algorithm [40]: “i) communication takes place at discrete time intervals; ii) nodes do not need to have globally unique IDs (although at the lowest communication layer we need to be able to distinguish between neighbors); iii) the network does not become partitioned with time.”

The general idea of the algorithm is that at each time unit, random subsets of nodes reset their state to a predefined value (the *DiffusionReset* algorithm [80]). The gossip-like communication creates an average aggregate value known by all nodes and node failures create deviations of this value. Therefore, the difference between the aggregate value and the expected value form an estimator for the failure rate within the network. For a full description please refer to the original paper [80].

Our attempt to improve this algorithm is based on the observation that the relative estimation error is non-zero.

6.3.2 Experimental Setup

We have implemented the algorithm using NetLogo (the original algorithm is referred to as “goal” from now on). The skeleton of the NetLogo experiment is shown in Listing 6.6. The estimate is calculated with the code in Listing 6.7, where `roundLength` and `avgWindow` are simulation parameters given at the start of the simulation. In our experiments `roundLength` = 100, `avgWindow` = 250 and `simulationLength` = 1000 ticks. The core of the FailDetect algorithm is the estimate calculation code, therefore this is where we focus our attempts of improvement. Using the MetaCompiler, we want to come up with a different way of calculating the fail estimate. We use the building blocks described in Table 6.4.

The simulation features 200 agents, placed at random in a 2D world of size 1000×1000 space units (s.u.) with wrapped around margins. The agents perform random walk with a speed of $m = 1$ s.u./tick. The transmission range of an agent is set to 10 s.u. The average node degree of the network

Listing 6.6: FailDetect NetLogo program skeleton

```

1 initialize failRate = random {0.01, 0.02, ..., 0.1}
2 every agent initializes clock = random [0, roundLength]
3
4 repeat simulationLength times:
5     move all agents
6     update neighbor knowledge
7     agents update estimate
8     agents with (clock = roundLength) reset the aggregate
9     unicast push gossiping with failRate chance of failing
10    increment clocks
11    save fitness

```

Listing 6.7: Code used by agents to update their fail rate estimate

```

1 set estimate ( - 1 / ( roundLength - 1 ) *
2   ( 1 - 1 / ( getAvgAggregate avgWindow ) ) )

```

graph formed is $d \simeq 20$. The simulation is executed for 1000 ticks.

6.3.3 Fitness Function

At the end of every tick the relative error of the estimate is saved into a list. The relative error of the estimate is given by η in Eq. 6.16, where N is the

Block name	Block functionality	Block type
CountNeighbors()	Returns the number of neighbors	1-hop
GetMaxEstimateNbr()	Returns the largest estimate of the 1-hop nbrs.	1-hop, filter
GetMinEstimateNbr()	Returns the smallest estimate of the 1-hop nbrs.	1-hop, filter
GetAvgEstimateNbr()	Returns the average estimate of the 1-hop nbrs.	1-hop, math
GetAvgAggregate(n)	Takes as parameter a number n . Returns the value of the local aggregate averaged over the last n ticks. The aggregate is a state variable at every node, correlated with gossiping.	local, math, historical
Plus, Minus, Div, Mul	Mathematical operations	math
If and If-Else	Branching statements	condition
Constants	0 ... 99; 0.01 ... 1.00	constants

Table 6.4: Building blocks used in the “Fail Estimate” case study.

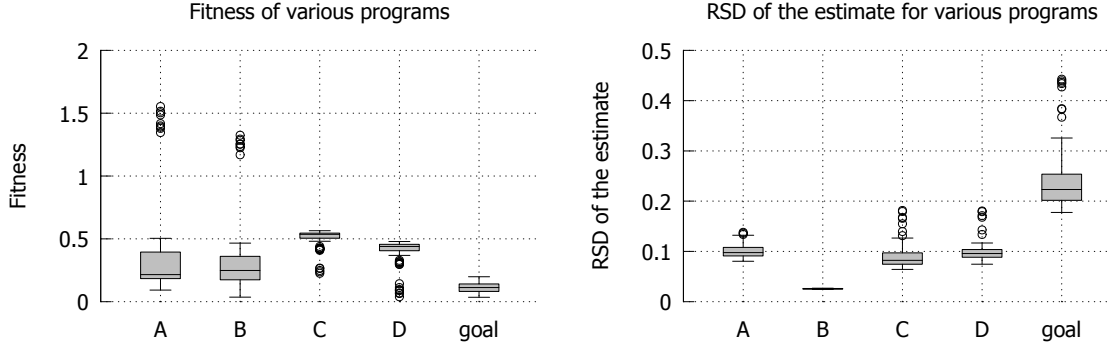


Figure 6.7: The fitness (left) and relative standard deviation of the estimate averaged over the last 250 ticks (right), for various programs and the “goal”.

set of agents.

$$\eta = \left| \left(\frac{\sum_i^N \text{estimate}(i)}{|N|} - \text{failRate} \right) / \text{failRate} \right| \quad (6.16)$$

Note that we also save, in a separate list, the relative standard deviation (RSD) of the estimate (for solution benchmarking in the next section), which is given by RSD in Eq. 6.17. σ is the standard deviation of the estimate of every node.

$$RSD = \left| \frac{\sigma}{\text{failRate}} \right| \quad (6.17)$$

The RSD is not used in the fitness value computation. The fitness is obtained by averaging η over the last 250 ticks of the simulation. Trivial solutions are avoided by initializing `failRate` at the beginning of the experiment with a random value between 0.01 and 0.1. Every program is simulated for 7 times and the maximum fitness is taken as the program’s fitness. Note that a trivial solution (e.g. `set estimate 0.08`) will have $RSD = 0$.

6.3.4 Found Programs

From what we have observed, no program manages to replicate the accuracy of the original algorithm. We have made a selection of programs having small fitness values. Due to space considerations we do not list the code of the programs found. The fitness and standard deviation of the estimate averaged over the last 250 ticks of the programs selected are shown in Fig. 6.7. We executed every program 100 times.

The programs we selected manage to have a good estimation accuracy, which is comparable with the one of the original algorithm. One of our

concern was that not all the nodes calculate the same estimate, and that the estimate somewhat gets close to the real failure rate due to averaging. The estimation consistency at node level (RSD) of program B is 10 times better than that of the original estimator. Suppose we did not know the original algorithm, program B could be a good enough replacement, since it has a median fitness of 0.25 and a 10 times smaller RSD than the original algorithm.

6.3.5 Discussion

We note one important insight related to avoiding trivial solutions. The method we used is to run a program 7 times and for each execution have a random failure rate chosen from the set $\{0.01, 0.02, \dots, 0.1\}$. The program's fitness is the maximum fitness obtained over all the runs. This approach is only partially effective, because over time, trivial programs can still obtain a good fitness. Take a program with a lot of code, having the last statement: `set estimate 0.08`. It can obtain a fitness of at most 0.33, if the random failure rate generator sets the failure rate to values from the set $\{0.06, 0.07, \dots, 0.1\}$, in all 7 executions. The chances of this not happening are 99.22% for one evaluation² (assuming a non-biased random number generator). When a trivial program gets a good fitness, it will be selected more often by the genetic operators to reproduce. Due to changes that occur in its first part of the code (result of which gets overwritten by the last statement), essentially the program that estimates the fail rate with 0.08 gets re-evaluated. This process of changing inactive code and re-evaluation can happen multiple times, therefore the chances of the program obtaining a very good fitness score grow over time. For example, for 100 evaluations the probability of it getting a fitness of 0.33 is:

$$p = 1 - 0.9922^{100} = 54.3\%$$

One way of avoiding this will be addressed in Section 8.2. Another method for avoiding this type of trivial solutions is to assign large fitness values to programs having a small *RSD* value.

We have seen that using the MetaCompiler, we can discover good enough programs that estimate the failure rate in a network. The median relative error of the estimation is two times larger than that of the original algorithm. However, the estimation consistency at a node level is 10 times better than that of the original algorithm (see program B, from Fig. 6.7). This makes the discovered programs useful for classes of applications where estimation consistency at node level is more important than estimation accuracy. One example can be a media access control (MAC) protocol that requires precise consensus between the nodes for a consistent retransmission strategy.

²One evaluation is represented by 7 executions of the program

6.4 ChurnDetect Algorithm

In this section we present our experiments of improving the *ChurnDetect* [79] algorithm. We give a very short description of the original algorithm. Then we describe the experimental setup together with the building blocks used and finally we evaluate some of the best programs found.

6.4.1 Problem and Algorithm Description

The ChurnDetect algorithm is a fully distributed scheme for online estimation of the average churn in a network. With churn we understand, the number of nodes that enter or leave the network over time. This is accomplished without the nodes advertising their departure from the network and without the nodes detecting that a node left the network. The algorithm is closely related to the FailDetect algorithm presented in the last case study.

ChurnDetect is also based on *DiffusionReset* [80]. The general idea is that new nodes enter with a random clock value and an aggregate of 0. At each reset the aggregate is set to 1. This means that every time a node enters the network, until its first reset, it creates an aggregate *loss* around it. The difference between the real aggregate value and the expected value (without any new nodes consuming aggregate) form an estimator for the churn rate. For a full description please refer to the original paper [79].

Our attempt to improve this algorithm is based on the observation that the relative churn estimation error is non-zero.

6.4.2 Experimental Setup

We have implemented the algorithm using NetLogo (the original algorithm is marked as “goal” from now on). The skeleton of the experiment is shown in Listing 6.8. The estimate is calculated with the code in Listing 6.9, where `roundLength` and `avgWindow` are parameters given as input. In our experiments `roundLength` = 50, `avgWindow` = 20 and `simulationLength` = 300 ticks. The churn is modelled at line 5 in Listing 6.8. Every agent leaves the network with chance `churnRate` / 100. For every agent that leaves, a new agent is spawned at a random position. The number of agents remains constant over time. The core of the ChurnDetect algorithm is the estimator from Listing 6.9, therefore this is where we focus our attempts of improve-

Block name	Block functionality	Bloc type
<code>sqrtN(n, ord)</code>	<code>if (ord mod 2 ≠ 0) or (ord mod 2 = 0 and n > 0)</code> <code>return $\sqrt[ord]{n}$ else return 10¹⁰</code>	math

Table 6.5: Building blocks added to the “Churn Estimate” case study.

Listing 6.8: ChurnDetect NetLogo program skeleton

```

1 initialize churnRate = random {0.1, 0.2, ..., 1.0}
2 every agent initializes clock = random [0, roundLength]
3
4 repeat simulationLength times:
5     create churn
6     move all agents
7     update neighbor knowledge
8     agents update estimate
9     agents with (clock = roundLength) reset the aggregate
10    unicast push-pull gossiping
11    increment clocks
12    save fitness

```

Listing 6.9: Code used by agents to update their churn rate estimate

```

1 set estimate
2 (1 - (sqrtN (getAvgAggregate avgWindow) roundLength)) * 100

```

ment. Using the MetaCompiler, we want to come up with a different code for calculating the estimate. We use the building blocks described in Table 6.4. In addition, we use the CountNbrFilter(“b”) block from Table 6.2 and a new mathematical block described in Table 6.5. In Appendix B we show the section of the MetaCompiler configuration file containing the definition of the building block set.

The simulation features 111 agents, placed at random in a 2D world of size 1000×1000 space units (s.u.) with wrapped around margins. The agents perform random walk with a speed of $m = 1$ s.u./tick. The transmission range of an agent is set to 20 s.u. The average node degree of the random geometric graph is $d \simeq 14$. The simulation runs for 300 ticks. Considering the maximum value of 1 for the `churnRate`, it can happen that on average one agent exits / enters the network per tick³. This can be considered a high value, therefore our experimental setup is ambitious because we are searching for an algorithm that can estimate very high churn rates.

6.4.3 Fitness Function

We calculate the relative error η of the churn estimate with Eq. 6.18, where N is defined as the set of old agents, i.e., the agents that reseted their aggregate at least once. The idea is that an agent that joins the network has to play the role of a new agent for at most 50 ticks. This is in order to

³One tick can be viewed as a communication round.

create the aggregate *loss* around it such that the old agents get a reliable churn rate estimate. Before the first aggregate reset, new agents cannot compute a correct estimate, because the agent enters the network with an aggregate of 0. The fitness is η averaged over the last 100 ticks. We also compute the average *RSD* in the same manner (see Eq. 6.19, where σ is the standard deviation of the estimate of every node).

$$\eta = \left| \left(\frac{\sum_i^N estimate(i)}{|N|} - churnRate \right) / churnRate \right| \quad (6.18)$$

$$RSD = \left| \frac{\sigma}{churnRate} \right| \quad (6.19)$$

A program is executed 7 times and we take the maximum fitness as the program’s fitness. In this case study we avoid the type of trivial solutions that set the estimate to a static value. The process of this type of trivial solutions getting a good fitness despite the random initial conditions was described in Section 6.3.5. We avoid them by assigning large fitness values to programs having a small *RSD* value.

6.4.4 Found Programs

We have selected 4 representative programs with good fitness scores and compared them with the original algorithm. While performing the comparison, we executed each program 100 times. The results are shown in Fig 6.8. One interesting program is program A, which is almost as good as the original. It, calculates the churn estimate using the code in Listing 6.10 (code simplified manually), where $\hat{\cdot}$ stands for power of. This code manages to approximate the behaviour of the original code (based on `sqrtn`). Similarly,

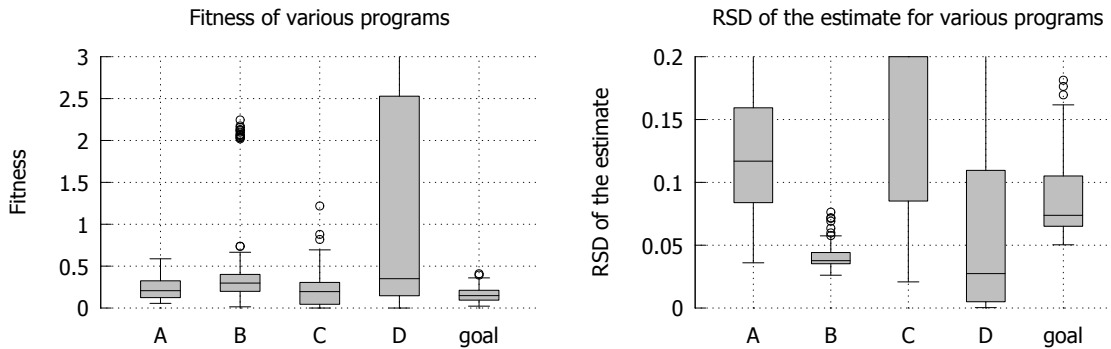


Figure 6.8: The fitness (left) and relative standard deviation of the estimate averaged over the last 100 ticks (right), for various programs and the “goal”.

Listing 6.10: Churn rate estimator in program A

```
1 set estimate 0.1 / ((getAvgAggregate avgWindow) ^ 6)
```

Listing 6.11: Churn rate estimator in program B

```
1 set estimate 0.28 / ((getAvgAggregate avgWindow) ^ 2)
```

program B estimates the churn with the code in Listing 6.11 (code simplified manually).

Program C is a trivial solution in which only the initial agents set an estimate. Due to a complicated if condition, new agents never set an estimate. By exploiting the way N is defined in Section 6.4.3, the fitness can be very small. Furthermore, in some program executions (out of the 100), by the end of the 300 ticks, no initial agent was left, causing a “division by 0” error. Nevertheless, this type of solutions, can improve their fitness over time, in a manner similar to the one discussed in Section 6.3.5. However, this type of solutions feature a large RSD.

Program D is based on a very long series of nested conditional statements. We did not list programs C and D due to space constraints.

The performance of programs A and B is comparable with the original algorithm. Program A has the best fitness out of the discovered programs but with a somewhat bad estimation consistency (RSD). Program B has an estimation consistency two times better than that of the original algorithm but with a slightly worse relative estimation error.

6.4.5 Discussion

It is important to note that the evaluated fitness of the MetaCompiler (chosen as the max out of 7 executions) for programs A-D was between 0.17 and 0.31. Yet the programs behave very differently. Furthermore, some of the programs from the top 20, were trivial solutions. This makes us wonder how well the quality of a program is represented by the fitness function used.

In this last case study, we managed to find two programs that have a performance comparable to that of the original algorithm. Program B has a much better estimation consistency at node level, which may prove essential for some applications. Moreover, the estimation computation pattern that appears in programs A and B gives us insights for future improvements of the *ChurnDetect* algorithm. As we observed, this pattern can lead to lower RSD values.

Chapter 7

Discussion

In this chapter we present and discuss the various insights gathered while performing the experiments described in Chapter 6. These insights are beneficial for the design of future MetaCompiler experiments aimed at discovering new or for improving existing algorithms for large-scale networks.

First, we describe the role of prior problem knowledge in the experimental setup definition. Then, we propose several fitness function designs for avoiding the discovery of trivial solutions. Afterwards, we state a positive effect of using generic building blocks. We also state a negative effect when used for the discovery of algorithms that proceed in rounds. We propose a design pattern to overcome this negative effect. Finally, we define a methodology for picking the right algorithmic building blocks.

7.1 Using Problem Knowledge

Even though it was stated by S. van Berkel (in [92]) that the MetaCompiler was designed to be used by non-expert distributed algorithms developers, we believe that the expert developers can make the most out of its use.

A MetaCompiler experiment has more chances of discovering useful programs, if the search is focused on one part of the program at a time, for example: a decision – Section 6.1; a filter – Section 6.2; an estimator – sections 6.3 and 6.4. This type of *focused* experiments, require a robust experimental setup containing a good fitness function, but also, initialization, global and local static procedures (program skeleton is described in Section 4.2.2). Problem knowledge is needed to, on the one hand, develop the experimental setup, and on the other hand, to validate the solutions found as real and non-trivial. Only an expert has this required level of problem knowledge. The effort put in using the MetaCompiler is well paid off. We have seen in Chapter 6 that the MetaCompiler can discover new algorithms, as well as provide new insights and ideas for future development of solutions for the problem at hand.

7.2 Fitness Function Design – Avoiding Trivial Solutions

At a first level of design, the fitness function evaluates the behavior of a program with respect to the desired outcome. However, this approach opens the door to trivial solutions. A trivial solution is a program that obtains the desired behavior with code that has a good fitness score given a trivial algorithm (i.e., an algorithm that can never work in reality). Therefore, at a second level of design, the fitness function, must detect trivial solutions and assign them large fitness scores.

One attempt of ensuring bad fitness scores for trivial solutions is to have an experimental setup with **random initial conditions**, such as the setups in the “FailDetect” and “ChurnDetect” case studies. This will also ensure that the solutions are feasible for more than a single set of initial conditions. Unfortunately, this approach is partly successful, because during the search, as generations are created, trivial solutions tend to improve their fitness (see Section 6.3.5). This can be overcome with a more direct approach: identify what defines a trivial solution in terms of behavior. We distinguish two cases:

1. **Unwanted behaviour represented by some small *value*.**

For example the standard deviation of the estimate in the “FailDetect” and “ChurnDetect” case studies is very small for trivial solutions. Banning such solutions is done by **assigning large fitness values to programs having that *value* under a certain threshold**.

2. **Unwanted behaviour represented by some large *value*.**

For example checking for convergence in the “Graph Spectral Theory” case study in Section 6.2 using the absolute difference between the last and the penultimate filter output. Such a value should be small, so in order to force this, we can directly **add the *value* to the fitness value** of the program.

For problems that feature some sort of random agent behaviour (e.g., random movement, random fail/churn rate), some level of **variation of some computed value is a hint for non-trivial solutions**. Also, there are problems that do not feature random behaviours, yet we require a non-constant result, such as the “Graph Spectral Theory” case study. In it, both too much (the result is random) and too little variation (the result is constant) are undesired. Therefore, we used a more generalized approach to measuring variation: **compute the FFT of the filter’s output over time**. This allowed us to reject both the static solutions and as an added bonus, the random solutions, and at the same time, keep the solutions that feature a relevant output pattern.

Finally, there are cases when we are not searching for a specific behavior, but instead, for a consistent behavior across multiple sets of initial conditions (such as in the “Graph Spectral Theory” case study). In these cases, a good

approach is to create a set of functions that compare the behaviours, by computing the areas between the function graphs.

7.3 Effect of the Generic Building Blocks

In the “Maximal Independent Set” case study, we successfully used the more generic building blocks (introduced in sections 4.1.1 and 4.2.3) for designing a new algorithm. Therefore, we were able to create an experimental setup with very little problem knowledge. The only design effort involved was to write the fitness function. However, this type of building blocks should be used with care because it can unnecessarily enlarge the search space. This, in turn, can prevent the finding of solutions. Furthermore, it can open the door to unforeseen trivial solutions.

7.3.1 Hiding the Intermediary States of an Algorithm

In the first two case studies, presented in sections 6.1 and 6.2, we used building blocks that access information from neighbors, e.g., CountNbrFilter, Laplacian, Gaussian, etc. Consider the set of neighbors of an agent executing such a building block. Due to NetLogo’s concurrency model (agents executing one at a time), at the moment the building block is executed by one agent, some of its neighbors have already executed this iterations’s code while others are still to execute it. We call this an intermediary state of an algorithm. Having access to the intermediary state can interfere with the good functioning of algorithms, such as the graph spectral theory based filters, that are designed to proceed in rounds. Furthermore, it can create trivial solutions, as we have seen in the “Maximal Independent Set” case study.

In order to prevent this, we can apply the following design pattern. We split every tick in two parts: a *compute result* part and *make result public* part. In the first part an agent can use only the public information of a neighbor. In the second part, the newly computed information is made public by every agent. Using this pattern, the two actions are completely separated: all agents *compute*, then all agents *make information public*. A different way of achieving this is by marking variables as private (i.e., hide them from neighbors). However, this feature is not currently supported and is left for future work.

7.4 Choosing the Right Set of Building Blocks

When creating the experimental setup, the algorithm designer has to make sure that the search is *focused* on a small part of the problem. For example in the “FailDetect” and “ChurnDetect” case studies, it would have been too

Phenomenon properties	Building block type
Temporal	Local, historical, filter
Spatial	K-hop, filter
Execution flow altering	Condition
Generic	Mathematical, constants
Problem specific	Any

Table 7.1: Mapping of phenomenon properties to building block types.

ambitious to have the MetaCompiler also discover when to do gossiping and how to reset the agent’s clock. Therefore, we believe that every experimental setup should have a set of static building blocks (see Section 4.2.2) that implement the parts of the algorithm that we do not want to touch and focus the search on a part the developer thinks can still be improved (we call this the *discovered* part).

The *discovered part* can be associated with one or multiple phenomenon properties. We identify the following properties:

- **Temporal:** uses information from earlier moments in time, not just the latest information.
- **Spatial:** uses information from other neighbouring agents, not just from the agent running the code.
- **Execution flow altering:** can alter the execution flow of the algorithm, i.e., via branching statements or loops.
- **Generic:** can use or represent any type of information, e.g., a multiplication, a constant.
- **Problem specific:** behaviour is specific to the usage scenario, e.g., move the artificial ant forward.

Coming from such properties we can create a mapping from a phenomenon property to the corresponding BB types, see Table 7.1. The BB types are described in Table 6.1. This mapping essentially tells us what set of BB types to choose for the problem at hand. In order to come up with a set of BBs, we can start from the already defined building blocks in tables 6.2, 6.3, 6.4, 6.5 from Chapter 6. Note that some of these building blocks are, to some extent, problem specific, because they use variables with clearly defined roles in the algorithm. For example the value of `inside` (from “Maximal Independent Set”) is used in the fitness function to identify if an agent is in the MIS or not. This role is consistent across all of the “Maximal Independent Set” building blocks (e.g., `CountInsideNbr()`) that use `inside`. Nevertheless, building

Abstraction level	Building block behaviour	Search space
1. Specific	Behaviour is specific and cannot be altered, e.g., constants, functions and procedures without parameters	Small
2. Parameterized	Behaviour can be specified by passing values as arguments, e.g., mathematical operations	Medium
3. Symbolic	Behaviour can be specified by passing symbols (i.e., lines of code) as arguments, e.g., 1-hop filtering BBs. Conditional statements are also part of this abstraction level.	Large

Table 7.2: Building block abstraction levels.

Problem	Properties
Artificial Ant	Spatial, execution flow, problem specific
Symbolic Regression	Generic
11-bit Multiplexer	Execution flow, problem specific
Maximal Independent Set	Spatial, execution flow, generic
Graph Spectral Theory	Spatial, temporal, generic
FailDetect	All
ChurnDetect	All

Table 7.3: Example of phenomenon properties of the problems presented in this thesis.

blocks that use such variables can be easily adapted to use variables from the algorithm at hand in a consistent manner with the other building blocks used.

After choosing the set of BBs, the developer has to identify the level of abstraction at which each BB is defined. We identify in Table 7.2, three levels of abstraction. A low level requires more problem knowledge and inherently limits the search space to a small set of possible behaviours. Higher levels of abstraction create more complex behaviours because the search space is enlarged. Even though it can be attractive to use many high-level BBs, these can create such a big search space that good solutions are never found within a given computational budget.

7.4.1 Examples of Choosing Building Blocks

In Table 7.3 we show a mapping between the various problems presented in this thesis and phenomenon properties associated with them. For each algorithm search we defined the BB set starting from the phenomenon properties. For example in the “Maximal Independent Set” case study, we defined the problem using a geometric graph, therefore it was natural to include spatial BBs. The agents had to decide whether to be or not to be in the set, therefore we added execution flow altering BBs (branching statements). Finally, we added generic BBs (constants) that were used as a set of values that can be assigned to `inside`. With `inside` we modelled set membership (if (`inside` *geq* 1) agent is in the set, otherwise is not in the set)

For another example, in the “Graph Spectral Theory” case study, we again use spatial and generic BBs. We had to use spatial ones to implement complex linear operators such as the Laplacian and the Gaussian. These operators represent basic components of filters applied on geometric graphs. The generic BBs were needed in order to implement basic mathematical operations. In the filters introduced by this case study, it is also common to apply operators on information from the past. Therefore, following this temporal property, we also added a Gaussian operator that uses past values to calculate its result (the neighbourhood average of the sensed value at a specific moment in time).

Coming to the abstraction level of a BB, we discovered that an iterative design process works the best. In case of the “Maximal Independent Set” case study, we started with specific BBs, and then challenged the MetaCompiler to find solutions with more abstract ones (simulating a lack of problem knowledge). For the “Graph Spectral Theory” case study, we started with more abstract BBs, and then gradually removed the more abstract ones, as we discovered that the solutions were featuring a non-linear behaviour. Therefore, we removed the BBs with non-linear behaviour, e.g., conditional statements and 1-hop filtering BBs. In the “FailDetect” and “ChurnDetect” case studies, we optimized the BB set to achieve a small as possible search space. For this reason we left out symbolic BBs such as `CountNbrFilter()` from Table. 6.2.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

The main goal of this thesis was to overcome the challenge of inventing distributed algorithms for large-scale networks. Our approach was to use Genetic Programming for realizing a *global-to-local compiler*. We split this goal into four objectives. The first was to extend the agent behaviour representation to be able to discover more complex algorithms. The second was to improve the performance of the MetaCompiler’s genetic heuristics. The last two were to prove the viability of our approach by discovering new and / or improve existing distributed algorithms for large-scale networks.

To extend the agent behaviour representation, we added more generic algorithmic building blocks that can receive parameters as symbols (strings). They enable agents to process information from their neighbors in more complex ways. Their use enables the discovery of more complex algorithms. It also helps the discovery of algorithms with less problem knowledge.

We implemented and tested six genetic heuristics, four inspired from state-of-the-art related work and two developed by us. After a thorough benchmarking using three well known problems (‘Artificial Ant on the Santa Fe Trail’, ‘Symbolic Regression’ and ‘Synthesis of an 11-bit Multiplexer’), we identified the best choice for our algorithms is an heuristic that uses steady-state Genetic Programming with a child acceptance condition inspired by Simulated Annealing. Our heuristics vastly outperform the state-of-the-art for the given benchmark problems.

With the MetaCompiler, we discovered new distributed algorithms for the ‘Maximal Independent Set Formation’ problem. Furthermore, we discovered novel distributed filters for the ‘Event Localization Using Graph Spectral Theory’ problem. The solutions developed open the door for future development of other custom and more advanced filters.

We also improved one of the characteristics of the ‘FailDetect’ and ‘ChurnDetect’ algorithms. Our improvement consists of lowering the vari-

ance of the estimate information available at each node. This comes at the cost of slightly worse accuracy of the estimation.

The experience with the four problems mentioned above enabled us to draw general guidelines for advanced fitness functions and building block designs.

In conclusion, we extended the agent behaviour representation language to be able to search for more complex programs and with less problem knowledge. We have improved the genetic heuristics of the MetaCompiler and we have also identified the optimal parameter combination for the heuristics. Finally, we have shown that the MetaCompiler can be successfully used as a tool for discovering new distributed algorithms for large-scale networks, as well as for improving existing ones.

8.2 Future Work

We identify several directions for future work.

The genetic heuristics can be extended to optimize more than the fitness score. We propose two more extensions to the optimization process: the contribution of a program to the diversity of the population and the program size parsimony. These can be used to differentiate between programs with equal fitness, or used in a Pareto multi-objective optimization technique.

The three problems we used as benchmarks only partly resemble algorithms for large-scale networks. We propose the creation of a new benchmark suite that contains problems closer to our domain.

Currently, the selection of the second parent in the HC, SA and GOMGP heuristics is done in a top-down manner: select one program, find a compatible subtree. It would be interesting to also test a bottom-up approach: identify all compatible subtrees from the population, select one. Furthermore, the GOMGP heuristic could benefit from a configurable halting mechanism.

Coming to the language used by the MetaCompiler to program agents, we can extend it by allowing user-defined types (e.g., by using Strongly-Typed GP [68] program representation). With this, we have more control on what variables can be passed on to functions as arguments. This will enable us to ensure correct variable usage and also variable privacy.

Finally, we believe that tracking the program's execution will open the door to several new useful features:

- building block identification at a functional level
- automatic code minimization of the best solution
- recognition of functionally identical programs

Bibliography

- [1] Y. Afek and E. Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 186–195. ACM, 1985.
- [2] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.
- [3] S. Baluja. Population-based incremental learning. a method for integrating genetic search based function optimization and competitive learning. Technical report, DTIC Document, 1994.
- [4] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. *arXiv preprint arXiv:1202.5509*, 2012.
- [5] S. Bleuler, M. Brack, L. Thiele, and E. Zitzler. Multiobjective genetic programming: Reducing bloat using spea2. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 1, pages 536–543. IEEE, 2001.
- [6] P. A. Bosman and E. D. De Jong. Learning probabilistic tree grammars for genetic programming. In *Parallel Problem Solving from Nature-PPSN VIII*, pages 192–201. Springer, 2004.
- [7] P. A. Bosman and D. Thierens. More concise and robust linkage learning by filtering and combining linkage hierarchies. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 359–366. ACM, 2013.
- [8] M. F. Brameier and W. Banzhaf. *Linear genetic programming*. Springer, 2007.
- [9] Y. Brun and all. Engineering self-adaptive systems through feedback loops. *Software Engineering for Self-Adaptive Systems*, pages 48–70, 2009.
- [10] E. Burke, S. Gustafson, G. Kendall, and N. Krasnogor. Advanced population diversity measures in genetic programming. In *Parallel Problem Solving from NaturePPSN VII*, pages 341–350. Springer, 2002.
- [11] E. K. Burke, S. Gustafson, and G. Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *Evolutionary Computation, IEEE Transactions on*, 8(1):47–62, 2004.
- [12] R. Burke, S. Gustafson, and G. Kendall. A survey and analysis of diversity measures in genetic programming. In *GECCO*, volume 2, pages 716–723, 2002.

- [13] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.
- [14] O. Cordon, F. Moya, and C. Zarco. A new evolutionary algorithm combining simulated annealing and genetic programming for relevance feedback in fuzzy information retrieval systems. *Soft Computing*, 6(5):308–319, 2002.
- [15] E. de Jong, R. Watson, and J. Pollack. Reducing bloat and promoting diversity using multi-objective methods. 2001.
- [16] E. D. De Jong and J. B. Pollack. Multi-objective methods for tree size control. *Genetic Programming and Evolvable Machines*, 4(3):211–233, 2003.
- [17] K. A. De Jong. Analysis of the behavior of a class of genetic adaptive systems. 1975.
- [18] K. A. De Jong and J. Sarma. Generation gaps revisited. In *FOGA*, pages 19–28. Citeseer, 1992.
- [19] A. Ekárt and S. Z. Nemeth. Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 2(1):61–73, 2001.
- [20] A. Ekárt and S. Z. Németh. Maintaining the diversity of genetic programs. In *Genetic Programming*, pages 162–171. Springer, 2002.
- [21] D. Epema. *Distributed Algorithms 6th edition - Lecture Notes*. Delft University of Technology, March 2011.
- [22] C. Ferreira. Gene expression programming: a new adaptive algorithm for solving problems. *arXiv preprint cs/0102027*, 2001.
- [23] G. Folino, C. Pizzuti, and G. Spezzano. Genetic programming and simulated annealing: a hybrid method to evolve decision trees. In *Genetic Programming*, pages 294–303. Springer, 2000.
- [24] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM, 2009.
- [25] A. Ghosh, S. Tsutsui, and H. Tanaka. Function optimization in nonstationary environment using steady state genetic algorithms with aging of individuals. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 666–671. IEEE, 1998.
- [26] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.
- [27] F. Glover. Tabu searchpart i. *ORSA Journal on computing*, 1(3):190–206, 1989.
- [28] F. Glover. Tabu searchpart ii. *ORSA Journal on computing*, 2(1):4–32, 1990.
- [29] D. E. Goldberg. The design of innovation: Lessons from genetic algorithms, lessons for the real world. *Urbana*, 51:61801, 1998.

- [30] V. Granville, M. Krivánek, and J.-P. Rasson. Simulated annealing: A proof of convergence. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(6):652–656, 1994.
- [31] G. R. Harik, F. G. Lobo, and D. E. Goldberg. The compact genetic algorithm. *Evolutionary Computation, IEEE Transactions on*, 3(4):287–297, 1999.
- [32] 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.
- [33] R. Harper and A. Blair. A structure preserving crossover in grammatical evolution. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 3, pages 2537–2544. IEEE, 2005.
- [34] J. H. Holland. Adaptation in artificial and natural systems. *Ann Arbor: The University of Michigan Press*, 1975.
- [35] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control and artificial intelligence*. MIT press, 1992.
- [36] A. Horner and D. Goldberg. Genetic algorithms and computer-assisted music composition. *Urbana*, 51(61801):14, 1991.
- [37] S.-T. Huang. Detecting termination of distributed computations by external agents. In *Distributed Computing Systems, 1989., 9th International Conference on*, pages 79–84. IEEE, 1989.
- [38] L. Ingber and B. Rosen. Genetic algorithms and very fast simulated reannealing: A comparison. *Mathematical and Computer Modelling*, 16(11):87–100, 1992.
- [39] B. Johanson and R. Poli. *GP-music: An interactive genetic programming system for music generation with automated fitness raters*. Citeseer, 1998.
- [40] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 482–491. IEEE, 2003.
- [41] S. Kirkpatrick, D. G. Jr., and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.
- [42] J. R. Koza. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Stanford University, Department of Computer Science, 1990.
- [43] J. R. Koza. Genetic programming: On the programming of computers by means of natural selection. 1992.
- [44] J. R. Koza. *Genetic programming II: automatic discovery of reusable programs*. MIT press, 1994.
- [45] J. R. Koza, V. A. Fairfax, F. H. Bennett, D. Andre, and M. A. Keane. Genetic Programming III: Darwinian Invention and Problem Solving. 2000.

- [46] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, G. Lanza, and J. Yu. *Genetic programming IV: Routine human-competitive machine intelligence*, volume 5. Springer Science+ Business Media, 2007.
- [47] J. Lahtinen, P. Myllymäki, T. Silander, and H. Tirri. Empirical comparison of stochastic algorithms. 1996.
- [48] W. B. Langdon. Fitness causes bloat: Simulated annealing, hill climbing and populations. 1997.
- [49] W. B. Langdon and R. Poli. *Fitness causes bloat*. Springer, 1998.
- [50] W. B. Langdon and R. Poli. Fitness causes bloat: Mutation. In *Genetic Programming*, pages 37–48. Springer, 1998.
- [51] W. B. Langdon and R. Poli. *Genetic programming bloat with dynamic fitness*. Springer, 1998.
- [52] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.
- [53] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.
- [54] D. Lehmann and M. O. Rabin. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 133–138. ACM, 1981.
- [55] A. Loukas, M. Zuniga, I. Protonotarios, and J. Gao. How to identify global trends from local decisions? spatial event detection on mobile networks. In *Proceedings of the 33rd IEEE International Conference on Computer Communications (INFOCOM14)*, April 2014 (to appear).
- [56] M. Lozano, F. Herrera, and J. R. Cano. Replacement strategies to preserve useful diversity in steady-state genetic algorithms. *Information Sciences*, 178(23):4421–4433, 2008.
- [57] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.
- [58] S. Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at <http://cs.gmu.edu/~sean/book/metaheuristics/>.
- [59] S. Luke and L. Panait. Fighting bloat with nonparametric parsimony pressure. In *Parallel Problem Solving from NaturePPSN VII*, pages 411–421. Springer, 2002.
- [60] S. Luke and L. Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, 2006.
- [61] S. Luke, L. Panait, et al. Lexicographic parsimony pressure. In *GECCO*, volume 2, pages 829–836, 2002.
- [62] N. A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

- [63] T. W. Manikas and J. T. Cain. Genetic algorithms vs. simulated annealing: A comparison of approaches for solving the circuit partitioning problem. 1996.
- [64] F. Mattern. Global quiescence detection based on credit distribution and recovery. *Information Processing Letters*, 30(4):195–200, 1989.
- [65] N. F. McPhee and J. D. Miller. Accurate replication in genetic programming. In *ICGA*, pages 303–309. Citeseer, 1995.
- [66] J. F. Miller and P. Thomson. Cartesian genetic programming. In *Genetic Programming*, pages 121–132. Springer, 2000.
- [67] J. Mogul. Emergent (mis) behavior vs. complex software systems. In *ACM SIGOPS Operating Systems Review*, 2006.
- [68] D. J. Montana. Strongly typed genetic programming. *Evolutionary computation*, 3(2):199–230, 1995.
- [69] M. Oltean, C. Groșan, L. Dioșan, and C. Mihăilă. Genetic programming with linear representation: a survey. *International Journal on Artificial Intelligence Tools*, 18(02):197–238, 2009.
- [70] M. O’Neill and C. Ryan. Evolving multi-line compilable c programs. In *Genetic Programming*, pages 83–92. Springer, 1999.
- [71] M. O’Neill and C. Ryan. Under the hood of grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1143–1148, 1999.
- [72] M. O’Neill and C. Ryan. Grammatical evolution. *Evolutionary Computation, IEEE Transactions on*, 5(4):349–358, 2001.
- [73] M. O’Neill, C. Ryan, M. Keijzer, and M. Cattolico. Crossover in grammatical evolution. *Genetic programming and evolvable machines*, 4(1):67–93, 2003.
- [74] F. Otero, T. Castle, and C. Johnson. Epochx: genetic programming in java with statistics and event monitoring. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, GECCO Companion ’12, pages 93–100, New York, NY, USA, 2012. ACM.
- [75] R. Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In *Genetic Programming*, pages 204–217. Springer, 2003.
- [76] R. Poli and N. F. McPhee. Covariant parsimony pressure in genetic programming. Technical report, Citeseer, 2008.
- [77] R. Poli and N. F. McPhee. Parsimony pressure made easy. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1267–1274. ACM, 2008.
- [78] A. Pruteanu, S. Dulman, and K. Langendoen. Ash: Tackling node mobility in large-scale networks. In *SASO, 2010*, pages 144–153, 2010.
- [79] A. Pruteanu, V. Iyer, and S. Dulman. Churndetect: a gossip-based churn estimator for large-scale dynamic networks. In *Euro-Par 2011 Parallel Processing*, pages 289–301. Springer, 2011.

- [80] A. Pruteanu, V. Iyer, and S. Dulman. Faildetect: Gossip-based failure estimator for large-scale dynamic networks. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–6. IEEE, 2011.
- [81] A. Rogers and A. Prügel-Bennett. Modelling the dynamics of a steady-state genetic algorithm. *Foundations of genetic algorithms*, 5:57–68, 1999.
- [82] F. Rothlauf and M. Oetzel. *On the locality of grammatical evolution*. Springer, 2006.
- [83] C. Ryan, J. Collins, and M. O. Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic Programming*, pages 83–96. Springer, 1998.
- [84] C. Ryan and M. O'Neill. Grammatical evolution: A steady state approach. *Late Breaking Papers, Genetic Programming*, 1998:180–185, 1998.
- [85] R. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.
- [86] L. Sánchez, I. Couso, and J. A. Corrales. Combining gp operators with sa search to evolve fuzzy rule based classifiers. *Information Sciences*, 136(1):175–191, 2001.
- [87] K. Sastry and D. E. Goldberg. Probabilistic model building and competent genetic programming. In *Genetic Programming Theory and Practice*, pages 205–220. Springer, 2003.
- [88] S. Silva and E. Costa. Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genetic Programming and Evolvable Machines*, 10(2):141–179, 2009.
- [89] T. Soule and J. A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 781–786. IEEE, 1998.
- [90] G. Syswerda. A study of reproduction in generational and steady-state genetic algorithms. *Foundation of Genetic Algorithms*, pages 94–101, 1991.
- [91] E.-G. Talbi. A taxonomy of hybrid metaheuristics. *Journal of heuristics*, 8(5):541–564, 2002.
- [92] S. van Berkel. Automatic discovery of distributed algorithms for large-scale systems, August 2012.
- [93] S. van Berkel, D. Turi, A. Pruteanu, and S. Dulman. Automatic discovery of algorithms for multi-agent systems. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, pages 337–344. ACM, 2012.
- [94] G. Werner-Allen, G. Tewari, A. Patel, M. Welsh, and R. Nagpal. Firefly-inspired sensor network synchronicity with realistic radio effects. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 142–153. ACM, 2005.

- [95] P. A. Whigham et al. Grammatically-based genetic programming. In *Proceedings of the workshop on genetic programming: from theory to real-world applications*, volume 16, pages 33–41, 1995.
- [96] K. Winstein and H. Balakrishnan. Tcp ex machina: Computer-generated congestion control.
- [97] K. P. Wong and Y. W. Wong. Genetic and genetic/simulated-annealing approaches to economic dispatch. In *Generation, Transmission and Distribution, IEEE Proceedings-*, volume 141, pages 507–513. IET, 1994.

Appendices

Appendix A

Pseudocode

In this Appendix the Pseudocode of the various search heuristics is listed.

1. Classic Genetic Programming

```

1:  $\mathcal{N} \leftarrow \text{GETINITIALPOPULATION}(); \mathcal{F} \leftarrow []$ 
2:  $\text{EVALUATEANDSTOREFITNESS}(\mathcal{N}, \mathcal{F})$ 
3:  $G_c \leftarrow 1$ 
4: while  $G_c < G_{max}$  and  $\neg \text{FITNESS0FOUND}(\mathcal{N})$  do
5:    $\mathcal{N}_{sorted} \leftarrow \text{SORTASCENDINGBYFITNESS}(\mathcal{N}, \mathcal{F})$ 
6:    $\mathcal{N}_{new} \leftarrow \text{TAKEFIRST}(\mathcal{N}_{sorted}, \text{EliteCount})$  ▷ Perform Ellitism
7:    $\text{GENERATENEWPOPULATION}(\mathcal{N}, \mathcal{N}_{new})$ 
8:    $\text{EVALUATEANDSTOREFITNESS}(\mathcal{N}_{new}, \mathcal{F})$ 
9:    $\mathcal{N} \leftarrow \mathcal{N}_{new}; G_c \leftarrow G_c + 1$ 
10:  $x^{best} \leftarrow \arg \min_{x \in \mathcal{N}} \{\mathcal{F}(x)\}$ 
11: return  $(x^{best}, \mathcal{F}(x^{best}))$ 
12:
13: procedure  $\text{GENERATENEWPOPULATION}(\mathcal{N}, \mathcal{N}_{new})$ 
14:   while  $|\mathcal{N}_{new}| < \text{PopulationSize}$  do
15:      $(x^1, x^2) \leftarrow (\text{TOURNAMENT}(7, \mathcal{N}), \text{TOURNAMENT}(7, \mathcal{N}))$ 
16:     if  $\text{RANDOM}(0.0, 1.0) < \text{CrossoverProbability}$  then
17:        $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + \text{CROSSOVER}(x^1, x^2)$ 
18:     else
19:        $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + \text{MUTATE}(x^1)$ 
20: end procedure
21:
22: procedure  $\text{EVALUATEANDSTOREFITNESS}(\mathcal{C}, \mathcal{F})$ 
23:   for all  $x \in \mathcal{C}$  do
24:      $\mathcal{R} \leftarrow []$ 
25:     for  $i \leftarrow 1$  to  $\text{SimulationRepeat}$  do
26:        $\mathcal{R} \leftarrow \mathcal{R} + \text{EVALUATE}(x)$ 
27:      $\mathcal{F} \leftarrow \mathcal{F} + (x, \text{AGGREGATEFITNESS}(\mathcal{R}))$ 
28: end procedure
29:
30:  $\text{AGGREGATEFITNESS}(\mathcal{R}) \in \{\text{MAX}(\mathcal{R}), \text{MIN}(\mathcal{R}), \text{MEAN}(\mathcal{R}), \text{MEDIAN}(\mathcal{R})\}$ 

```

2. General Structure of the Conditional Whole-Population Heuristics

```

1:  $\mathcal{N} \leftarrow \text{GETINITIALPOPULATION}(); \mathcal{F} \leftarrow []$ 
2:  $\text{EVALUATEANDSTOREFITNESS}(\mathcal{N}, \mathcal{F})$ 
3:  $G_c \leftarrow 1$ 
4: while  $G_c < G_{max}$  and  $\neg \text{FITNESS0FOUND}(\mathcal{N})$  do
5:    $x^{best} \leftarrow \arg \min_{x \in \mathcal{N}} \{\mathcal{F}(x)\}$ 
6:    $\mathcal{S} \leftarrow \mathcal{C} \leftarrow \mathcal{N}_{new} \leftarrow []$ 
7:    $\text{GENERATENEWPOPULATION}(\mathcal{N}, \mathcal{C}, \mathcal{S})$ 
8:    $\text{EVALUATEANDSTOREFITNESS}(\mathcal{C}, \mathcal{F})$ 
9:    $\text{SELECTNEXTPOPULATION}()$ 
10:   $\mathcal{N} \leftarrow \mathcal{N}_{new}; G_c \leftarrow G_c + 1$ 
11:  $x^{best} \leftarrow \arg \min_{x \in \mathcal{N}} \{\mathcal{F}(x)\}$ 
12: return  $(x^{best}, \mathcal{F}(x^{best}))$ 
13:
14: procedure  $\text{GENERATENEWPOPULATION}(\mathcal{N}, \mathcal{C}, \mathcal{S})$ 
15:   for all  $x \in \mathcal{N}$  do
16:      $x_c \leftarrow \text{GETCHILD}(\mathcal{N}, x)$ 
17:      $\mathcal{C} \leftarrow \mathcal{C} + x_c; \mathcal{S} \leftarrow \mathcal{S} + (x_c, x)$ 
18: end procedure
19:
20: function  $\text{GETCHILD}(\mathcal{N}, x)$ 
21:   if MutationAlone then
22:      $x_c \leftarrow \text{MUTATE}(x)$ 
23:   else if KozaOperator then
24:     if  $\text{RANDOM}(0.0, 1.0) < \text{CrossoverProbability}$  then
25:        $(x_c, x_t) \leftarrow \text{CROSSOVER}(x, \text{TOURNAMENT}(7, \mathcal{N}))$ 
26:     else
27:        $x_c \leftarrow \text{MUTATE}(x)$ 
28:   else if MixedOperator then
29:      $(x_{t1}, x_{t2}) \leftarrow \text{CROSSOVER}(x, \text{TOURNAMENT}(7, \mathcal{N}))$ 
30:      $x_c \leftarrow \text{MUTATE}(x_{t1})$ 
31: return  $x_c$ 
32: end function

```

3. Hill Climber Heuristic

```

1: procedure  $\text{SELECTNEXTPOPULATION}(\mathcal{S}, \mathcal{F}, \mathcal{N}_{new})$ 
2:   for all  $(x, x_p) \in \mathcal{C}$  do
3:     if  $\mathcal{F}(x) \leq \mathcal{F}(x_p)$  then
4:        $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + x$ 
5:     else
6:        $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + x_p$ 
7: end procedure

```

4. Simulated Annealing Heuristic

```
1: procedure SELECTNEXTPOPULATION( $\mathcal{S}, \mathcal{F}, \mathcal{N}_{new}, x^{best}$ )
2:   for all  $(x, x_p) \in \mathcal{S}$  do
3:     if  $\mathcal{F}(x) \leq \mathcal{F}(x_p)$  then
4:        $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + x$ 
5:     else if  $\text{RANDOM}(0.0, 1.0) < P_{sa}$  and  $x_p \neq x^{best}$  then
6:        $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + x$ 
7:     else
8:        $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + x_p$ 
9: end procedure
```

5. Tabu Search Heuristic

```
1: procedure SELECTNEXTPOPULATION( $\mathcal{S}, \mathcal{F}, \mathcal{N}_{new}$ )
2:   for all  $(x, x_p) \in \mathcal{S}$  do
3:     if  $\nexists x \in \mathcal{F}$  then
4:        $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + x$ 
5:     else
6:        $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + x_p$ 
7: end procedure
```

6. GOMGP

```
1:  $\mathcal{N} \leftarrow \text{GETINITIALPOPULATION}(); \mathcal{F} \leftarrow []$ 
2:  $\text{EVALUATEANDSTOREFITNESS}(\mathcal{N}, \mathcal{F})$ 
3:  $G_c \leftarrow 1$ 
4:  $t^{NIS} \leftarrow 0$ 
5:  $x^{best} \leftarrow \arg \min_{x \in \mathcal{N}} \{\mathcal{F}(x)\}$ 
6: while  $G_c < G_{max}$  and  $\neg \text{FITNESS0FOUND}(\mathcal{N})$  do
7:    $\mathcal{V} \leftarrow []$ 
8:   for all  $x \in \mathcal{N}$  do
9:      $\mathcal{V} \leftarrow \mathcal{V} + (x, 0)$  ▷ Initialize change counter to 0
10:   $\text{GENERATIONPHASE}(\mathcal{N}, \mathcal{V}, 1)$ 
11:  if  $t^{NIS} > 1 + \lfloor^{10} \log(|\mathcal{N}|) \rfloor$  then
12:     $\text{RESETALLVALUESTo0}(\mathcal{V})$  ▷ Reset change counter to 0
13:   $\text{GENERATIONPHASE}(\mathcal{N}, \mathcal{V}, 2)$ 
14:   $\text{REPLACEUNIMPROVED}(\mathcal{N}, \mathcal{V})$ 
15:   $\text{UPDATETNIS}(\mathcal{N}, t^{NIS}, x^{best})$ 
16:  if  $\text{and}_{x \in \mathcal{N}} \{x = x^{best}\}$  then
17:    break while ▷ Algorithm has converged
18:   $G_c \leftarrow G_c + 1$ 
19: return  $(x^{best}, \mathcal{F}(x^{best}))$ 
20:
21: procedure GENERATIONPHASE( $\mathcal{N}, \mathcal{V}, phase$ )
22:    $i \leftarrow 1$ 
23:   while  $i \leq \text{MaxAttempts}$  and  $\neg \text{FITNESS0FOUND}(\mathcal{N})$  do
24:      $\mathcal{S} \leftarrow \mathcal{C} \leftarrow []$ 
25:     for all  $x \in \mathcal{N}$  do
```

```

26:       $\mathcal{L} \leftarrow \text{GETNODELIST}(x)$ 
27:      if  $phase = 1$  then
28:           $x_p \leftarrow \text{TOURNAMENTOTHERPARENT}(\mathcal{N}, x)$ 
29:      else if  $\mathcal{V}(x) = 0$  then ▷ If  $x$  is unchanged
30:           $x_p \leftarrow x^{best}$ 
31:      else
32:          continue ▷  $x$  already improved, skipping it
33:       $pos \leftarrow \lceil i \times |\mathcal{L}| / \text{MAXATTEMPTS} \rceil - 1$ 
34:       $x_c \leftarrow \text{GETCHILDATNODE}(x, x_p, \mathcal{L}[pos])$ 
35:       $\mathcal{C} \leftarrow \mathcal{C} + x_c$ ;  $\mathcal{S} \leftarrow \mathcal{S} + (x_c, x)$ 
36:       $\text{EVALUATEANDSTOREFITNESS}(\mathcal{C}, \mathcal{F})$ 
37:       $\text{ADDTONEXTPOPULATION}(\mathcal{S}, \mathcal{N}, \mathcal{V})$ 
38: end procedure
39:
40: function  $\text{GETCHILDATNODE}(x, x_p, node)$ 
41:     if MutationAlone then
42:          $x_c \leftarrow \text{MUTATEATNODE}(x, node)$ 
43:     else if KozaOperator then
44:         if  $\text{RANDOM}(0.0, 1.0) < \text{CrossoverProbability}$  then
45:              $(x_c, x_t) \leftarrow \text{CROSSOVERATNODE}(x, x_p, node)$ 
46:         else
47:              $x_c \leftarrow \text{MUTATEATNODE}(x, node)$ 
48:     else if MixedOperator then
49:          $(x_{t1}, x_{t2}) \leftarrow \text{CROSSOVERATNODE}(x, x_p, node)$ 
50:          $x_c \leftarrow \text{MUTATEATNODE}(x_{t1}, node)$ 
51: return  $x_c$ 
52: end function
53:
54: procedure  $\text{ADDTONEXTPOPULATION}(\mathcal{S}, \mathcal{N}, \mathcal{V})$ 
55:      $\mathcal{N} \leftarrow []$ 
56:     for all  $(x, x_p) \in \mathcal{S}$  do
57:         if  $\mathcal{F}(x) \leq \mathcal{F}(x_p)$  then
58:              $\mathcal{N} \leftarrow \mathcal{N} + x$ 
59:              $c \leftarrow \mathcal{V}(x_p)$ 
60:              $\mathcal{V} \leftarrow \mathcal{V} - (x_p, c)$ ;  $\mathcal{V} \leftarrow \mathcal{V} + (x, c + 1)$ 
61:         else
62:              $\mathcal{N} \leftarrow \mathcal{N} + x_p$ 
63: end procedure
64:
65: procedure  $\text{REPLACEUNIMPROVED}(\mathcal{N}, \mathcal{V})$ 
66:      $\mathcal{N}_{new} \leftarrow []$ 
67:     for all  $x \in \mathcal{N}$  do
68:         if  $\mathcal{V}(x) = 0$  then
69:              $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + x^{best}$ 
70:         else
71:              $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + x$ 
72:      $\mathcal{N} \leftarrow \mathcal{N}_{new}$ 
73: end procedure

```

```

74:
75: procedure UPDATETNIS( $\mathcal{N}, t^{NIS}, x^{best}$ )
76:    $x_{old}^{best} \leftarrow x^{best}$ 
77:    $x^{best} \leftarrow \arg \min_{x \in \mathcal{N}} \{\mathcal{F}(x)\}$ 
78:   if  $\mathcal{F}(x^{best}) < \mathcal{F}(x_{old}^{best})$  then
79:      $t^{NIS} \leftarrow 0$ 
80:   else
81:      $t^{NIS} \leftarrow t^{NIS} + 1$ 
82: end procedure

```

7. GATLE

```

1:  $\mathcal{N} \leftarrow \text{GETINITIALPOPULATION}(); \mathcal{F} \leftarrow []$ 
2:  $P_{max} \leftarrow |\mathcal{N}|$ 
3:  $P_{gap} \leftarrow \lfloor P_{max} \times \delta \rfloor$ 
4: if  $P_{gap} < 1$  then
5:    $P_{gap} \leftarrow 1$ 
6:  $P_{min} \leftarrow P_{max} - P_{gap}$ 
7:  $\text{EVALUATEANDSTOREFITNESS}(\mathcal{N}, \mathcal{F})$ 
8:  $G_c \leftarrow 1$ 
9: while  $G_c < G_{max}$  and  $\neg \text{FITNESS0FOUND}(\mathcal{N})$  do
10:    $\mathcal{C} \leftarrow \mathcal{N}_{new} \leftarrow []$ 
11:    $\text{PERFORMELLITISM}(\mathcal{N}, \mathcal{N}_{new})$ 
12:   for  $attempts \leftarrow 1$  to  $\text{MaxAttempts}$  do
13:     if  $|\mathcal{N}_{new}| < P_{max}$  and  $\neg \text{FITNESS0FOUND}(\mathcal{N})$  then
14:        $\text{GENERATENEWCANDIDATES}(\mathcal{N}, \mathcal{C})$ 
15:        $\text{EVALUATEANDSTOREFITNESS}(\mathcal{C}, \mathcal{F})$ 
16:        $\text{ADDTONEXTPOPULATION}(\mathcal{C}, \mathcal{N}_{new})$ 
17:    $\mathcal{N} \leftarrow \mathcal{N}_{new}; G_c \leftarrow G_c + 1$ 
18:  $x^{best} \leftarrow \arg \min_{x \in \mathcal{N}} \{\mathcal{F}(x)\}$ 
19: return  $(x^{best}, \mathcal{F}(x^{best}))$ 
20:
21: procedure PERFORMELLITISM( $\mathcal{N}, \mathcal{N}_{new}$ )
22:    $\mathcal{N}_{sorted} \leftarrow \text{SORTASCENDINGBYFITNESS}(\mathcal{N}, \mathcal{F})$ 
23:   if  $|\mathcal{N}| \leq P_{min}$  then
24:      $\mathcal{N}_{new} \leftarrow \mathcal{N}$ 
25:   else
26:      $\mathcal{N}_{new} \leftarrow \text{TAKEFIRST}(P_{min}, \mathcal{N}_{sorted})$ 
27:      $f_{goal} \leftarrow \mathcal{F}(\text{LAST}(\mathcal{N}_{new}))$ 
28:      $f_{worst} \leftarrow \mathcal{F}(\text{LAST}(\mathcal{N}_{sorted}))$ 
29: end procedure
30:
31: procedure GENERATENEWCANDIDATES( $\mathcal{N}, \mathcal{C}$ )
32:    $\mathcal{C} \leftarrow []$ 
33:   for  $i \leftarrow 1$  to  $2 \times P_{gap}$  do
34:      $\mathcal{C} \leftarrow \mathcal{C} + \text{GETCHILD}(\mathcal{N})$ 
35: end procedure
36:
37:

```

```

38: function GETCHILD( $\mathcal{N}$ )
39:   if MutationAlone then
40:     return MUTATE(TOURNAMENT(7,  $\mathcal{N}$ ))
41:   else if KozaOperator then
42:     if RANDOM(0.0, 1.0) < CrossoverProbability then
43:       return CROSSOVER(TOURNAMENT(7,  $\mathcal{N}$ ), TOURNAMENT(7,  $\mathcal{N}$ ))
44:     else
45:       return MUTATE(TOURNAMENT(7,  $\mathcal{N}$ ))
46:   else if MixedOperator then
47:      $(x_{t1}, x_{t2}) \leftarrow$  CROSSOVER(TOURNAMENT(7,  $\mathcal{N}$ ), TOURNAMENT(7,  $\mathcal{N}$ ))
48:     return (MUTATE( $x_{t1}$ ), MUTATE( $x_{t2}$ ))
49: end function
50:
51: procedure ADDTONEXTPOPULATION( $\mathcal{C}, \mathcal{N}_{new}$ )
52:    $\mathcal{C}_{sorted} \leftarrow$  SORTASCENDINGBYFITNESS( $\mathcal{C}, \mathcal{F}$ )
53:   for all  $x \in \mathcal{C}_{sorted}$  do
54:     if  $\mathcal{F}(x) < f_{goal}$  and  $|\mathcal{N}_{new}| < P_{max}$  then
55:        $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + x$ 
56: end procedure

```

8. GATSE

```

1: procedure ADDTONEXTPOPULATION( $\mathcal{C}, \mathcal{N}_{new}$ )
2:    $\mathcal{C}_{sorted} \leftarrow$  SORTASCENDINGBYFITNESS( $\mathcal{C}, \mathcal{F}$ )
3:   for all  $x \in \mathcal{C}_{sorted}$  do
4:     if  $|\mathcal{N}_{new}| < P_{max}$  then
5:       if  $\mathcal{F}(x) < f_{goal}$  then
6:          $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + x$ 
7:       else if RANDOM(0.0, 1.0) <  $P_{gatse}$  then
8:          $\mathcal{N}_{new} \leftarrow \mathcal{N}_{new} + x$ 
9: end procedure

```

Appendix B

Sample Configuration File

Listing B.1: Excerpt of a building block setup for the “ChurnDetect” configuration file.

```
1 simulation_configuration:
2 <...>
3
4 - libraries: "churn-estimate.nls"
5
6 global_fields:
7   - hidden fitness {}
8   - hidden fitnessNow 0
9   - hidden stdDevList {}
10  - hidden stdDevNow 0
11  - hidden avgNetSize 111
12  - hidden churnRate // set by createRandChurnRate()
13  - hidden transRange 20
14  - hidden nodeSpeed 0.1
15
16  - roundLength 50
17  - avgWindow 20
18
19 agent_fields:
20   - hidden clock // set by initClock()
21   - hidden aggregate 0.0
22   - hidden aggregatePrev {}
23   - hidden estimateSet false
24   - hidden resetValue 1.0
25   - estimate 0.0
26
27 initialization_primitives:
28   - initClock ()
29   - createRandChurnRate ()
30   - updateTransRange (15 100)
31
```

```

32 agent_action_primitives:
33
34 agent_returning_primitives:
35     - number countNbrFilter (CONDITIONAL_AS_STRING)
36     - number countNeighbors ()
37     - number getMaxEstimateNbr ()
38     - number getMinEstimateNbr ()
39     - number getAvgEstimateNbr ()
40     - number getAvgAggregate (NUMBER)
41     - number plus (NUMBER NUMBER)
42     - number minus (NUMBER NUMBER)
43     - number division (NUMBER NUMBER)
44     - number mul (NUMBER NUMBER)
45     - number sqrtN (NUMBER NUMBER)
46
47 agent_static_primitives:
48
49 global_static_primitives:
50     - begin updateNodeCount (avgNetSize churn)
51     - begin moveRandomWalk (nodeSpeed)
52     - begin updateNetworkGraph (transRange)
53     - end resetAggregate ()
54     - end gossipPushPull ()
55     - end incrementAgentsClocks (roundLength)
56     - end updateFitness ()
57
58 fitness_function: "getFitness 100"

```
