Delft University of Technology
Master's Thesis in Computer Science

Automatic Discovery of Distributed Algorithms for Large-Scale Systems

Sjors van Berkel

Universiteitsfonds Delft

embedded software

TU Delft Delft University of Technology

Automatic Discovery of Distributed Algorithms for Large-Scale Systems

Master's Thesis in Computer Science

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

Sjors van Berkel
s.e.f.vanberkel@student.tudelft.nl

8th August 2012

Author
Sjors van Berkel (s.e.f.vanberkel@student.tudelft.nl)

Title
Automatic Discovery of Distributed Algorithms for Large-Scale Systems

MSc presentation
22nd August 2012

Graduation Committee

Prof. dr. K.G. Langendoen (chair)	Delft University of Technology
Dr. S.O. Dulman	Delft University of Technology
Dr. H.G. Gross	Delft University of Technology
Dr. P.A.N. Bosman	Centrum Wiskunde & Informatica
A.S. Pruteanu, MSc.	Delft University of Technology

Abstract

In recent years, large-scale systems have become mainstream at a very high pace. Typical examples of large-scale systems are MANETs, Wireless Sensor Networks, Pervasive Computing, Swarm Robotics, etc. These systems distinguish themselves by the large number of devices they embody, and emergent behaviors they exhibit: Behavior that is globally perceivable, but that is made up of only local interactions of the system elements.

Because of the vast amount of devices that make up a large-scale system, it is infeasible to exhibit centralized control. As an alternative, we need to leverage distributed algorithms to create and control emergent behaviors for the global goal we want the system to exhibit.

Since there is no linear mapping from local interactions to global behavior, we present a global-to-local compiler to automatically generate these distributed algorithms for large-scale systems. By using Genetic Programming to combine already known building blocks from other distributed algorithms, we provide a high-level, goal-driven framework for algorithm designers to design distributed algorithms.

Evaluation shows that the framework we present is indeed a valuable tool for designing distributed algorithms for large-scale systems. Improving the development speed, allowing the designer to be agnostic to the underlying details, but nevertheless providing a flexible interface, to acquire the algorithm desired.

In dedication to my father,
who has inspired me for many years to come

Preface

This thesis is the result of the work I did from September 2011 to August 2012 at the Embedded Software Group of the Delft University of Technology. I was introduced to the topic of Wireless Sensor Networks in an elective course in 2009, where students were supposed to read and present works on the topic. It got me excited, and not only did I decide to follow another course on the subject, I also stayed in touch with the people at the Embedded Software Group. When I was ready to pick a topic for my thesis, and I decided I preferred to do *something crazy* rather than an ordinary Computer Science topic, I knew just on which door to knock...

First of all, I would like to thank both my supervisors, Andrei Pruteanu and Stefan Dulman, for their support, guidance, and insights throughout the whole process. I would also like to thank the other members of our Snowdrop team, who were working on their theses at the same time, it was a good year! I would like to thank UniversiteitsFonds, ACM/GECCO, and the Embedded Software department for funding my trip to the GECCO'12 conference in Philadelphia. And of course I would like to thank my friends and family, your support and love are simply amazing at all times, thanks!

Sjors van Berkel

Delft, The Netherlands
August 9, 2012

Contents

Chapter 1

Introduction

The current technological advances have lead to the creation of engineered computing systems characterized by high complexity in terms of software [32], system engineering [14] and expected quality of service [13]. Most of these systems are made out of a large number of structural elements that have complex dependencies and emergent properties. Some of them exhibit a tight-coupling between their structural elements, that are split across a large number of layers [10], while others are less hierarchical while being topologically distributed over large physical spaces [60].

The scale of the computing systems, measured in terms of the number of constituent elements, the complexity of the software stack required to control them, the tolerance to failures, etc., require novel programming paradigms. [3] Besides that, a well-known property of large-scale systems is that above a certain size, global properties occur somewhat unexpectedly out of simple, local interactions.

In most cases *emergent behavior* has mainly negative effects [33]. In some other cases, there are positive properties that can be put to good use, like synchronization algorithms [52], clustering schemes [43], distributed feedback mechanisms [9] etc. Looking at these properties from a constructive point of view, there is a high interest in programming in terms of emergent behavior, using generative local behaviors (both computation-wise as well as communication-wise), since such programs scale-up very well for large systems [2].

Since large-scale systems are vast networks of devices, controlling all of them centrally is simply infeasible. The classic way to write a program for such a system is to manually define local interactions. Current distributed algorithms that produce global (aggregate) behaviors like synchronization, clustering, and leader election, were created by algorithm designers in a bottom-up manner by testing various local behaviors that produce interesting global patterns. Many

algorithm designers however, are interested in aggregate behaviors and cannot easily decompose them into local interaction.

We propose a so-called "Global-to-local compiler" [59] to automatically generate distributed algorithms by combining local (inter)actions. These local interactions can be a pair-wise exchange of state information in one-hop neighborhoods using gossiping, or an update of the local state based on neighboring information. Since there is no linear mapping between the two (global and local) levels, this has proven to be a very complex problem [38]. Previous attempts to create a global-to-local compiler, like BehaviorSearch[48] or ABM[31], have not succeeded in offering a high-level, and flexible interface to the user, and therefore they have not succeeded in becoming popular tools. Our approach to solve the problem at hand is to use *Genetic Programming* in order to discover algorithms that fulfill the desired global behavior. A large number of combinations of local behaviors is evaluated in order to determine how well they map onto this behavior.

What differentiates this project from other projects?

- It focuses on the world of **large-scale** systems. Since these systems have shown to behave differently from other networked systems, it is important to incorporate these differences in the design phase of algorithms. Differences include emergence properties, scalability issues, restricted communication, and geographical relevance for some problems.
- The project takes a **holistic**, top-down approach to designing complex global behaviors; It tries to come up with a solution for the full problem, so the user does not have to break up the problem into subproblems.
- The algorithm designer is **agnostic** on how the building blocks he provides are put together to generate the algorithm.
- It uses a generic agent-based system simulator that is very **flexible**; It's not bound to a certain simulation scenario or network topology.
- We test the project with the discovery of already known large-scale distributed algorithms, to show that our approach is well equipped to **speed** up the discovery of these kinds of algorithms.

To achieve our goals, we make sure that our global-to-local compiler tailors programs to be executed on the target systems(i.e. not use functionalities the systems are not capable of). Envisioned platforms are embedded networked platforms like Wireless Sensor Networks, MANETs, swarm robotics, sensors, the internet of things, and pervasive computing. Offering algorithm designers a way to think in global behaviors for these kinds of systems will speed up the implementation tremendously. This opens up the possibility of easier high-level programming of such systems.

1.1 Problem Statement

The main goal for the thesis is to start from a description of the global behavior of the large-scale system, and automatically discover local rules that compose it. The thesis answers the following research questions:

1. *What Genetic Programming techniques can we use to invent algorithms? What are the drawbacks? Can we solve these?* We take a look at the current state of Genetic Programming techniques, consider their advantages and disadvantages, and pick a technique to tailor to our needs. Then we build the system and analyze its performance. Based on these findings we come up with a number of enhancements in order to find better solutions, including pattern detection and protection of good algorithm structures, and syntactic-aware genetic operators.

2. *How can we build a Global-to-Local compiler for Large-Scale Systems using Genetic Programming?* We investigate in already known large-scale distributed algorithms, and define a number of rules for what this class of algorithms can and cannot do, and how we can use that knowledge in Genetic Programming.

1.2 Organization

This thesis is organized as follows. In Chapter 2, we look at some background information like the multiple subjects involved, related projects, and auxiliary thoughts on this project. Next, in Chapter 3, we look at Genetic Programming as a technique for discovering programs. Then, in Chapter 4, we look at the enhancements we added to the Genetic Programming process in order to increase the effectiveness of the algorithm discovery process. Subsequently, in Chapter 5, we look at large-scale systems to define what they are, and how that influences the discovery of algorithms. In Chapter 6, we look at the implementation of the MetaCompiler, our framework for algorithm discovery for large-scale systems. In Chapter 7, we look at the experimental results, and see if our enhancements have had any effect. Chapter 8 contains a discussion on the previous chapters. In Chapter 9, we conclude the thesis and look at possibilities for future improvement.

Chapter 2

Background

This chapter serves to give background information information about the MetaCompiler, which is the name for our global-to-local compiler, and subjects related (see Figure 2.1).

2.1 History

The MetaCompiler is part of a bigger project called the Snowdrop project (`http://code.google.com/p/snowdrop/`). The project was started by Stefan Dulman and Andrei Pruteanu in order to construct a set of well connected MSc projects in the academic year of 2011-2012. Its main goal is to ease the development of large-scale networks, improving both the ease of software development for such systems, as well providing a hardware platform to run the software on. Typical large-scale systems we aim for are embedded networked systems, like robotics, sensors, the internet of things, or pervasive computing.

Steffan Karger, a MSc student in Electrical Engineering, has been working on software for a hardware platform that enables programs written in the MIT developed Proto language, to be able to run on a hardware platform. Agostino di Figlia has been working on an interface for designers to program distributed systems, and translate their programs to eLua, a version of Lua that runs on embedded devices. Another outcome of his project is a so-called state chart compiler that allows a designer to think in program states for each system rather than in low-level programming. Both projects aim to ease the process of letting the developer develop programs in a bottom-up approach.

The Snowdrop project acknowledges that the projects mentioned above are necessary programming methodologies for large-scale systems, but it also reco-

Figure 2.1: Subjects related to the MetaCompiler

Figure 2.2: protoSPACE 3.0

gnizes that there is still a big gap between the knowledge of non-IT users, the usual designers of systems (architects), and the necessary software development skills. One part of the gap is the obvious lack of programming skills, which we would like to solve by offering a more visual approach and other simple interfaces that designers can more easily understand and leverage. The other part of the gap resides in the observation that people, when thinking of systems, like to think of the big picture rather than the cogs that make it run. This contradicts the fact that when we want to write a program for such a network of agents, we have to make the leap to programming this local behavior. The Snowdrop project tries to close this gap with a framework called the MetaCompiler. The MetaCompiler tries to come up with a program consisting of local actions and rules for all separate elements of the network, that achieve the global goal set by the designer of a system.

The MetaCompiler uses building blocks that are either provided by the user or are readily defined in default libraries. Genetic Programming is used to combine these building blocks, and assess how good the programs are. Because of the vast amount of combinations of building blocks, and the fact that every generated program has to be thoroughly tested, this takes up a large amount of computing time.

In order to reduce this computing time, another subproject project aims to speed up the simulations. Dániel Turi, another MSc student in our group, has worked on developing a way to run NetLogo programs on the GPU architecture CUDA by Nvidia, rather than the default way of running simulations on the CPU. Because a GPU has many more separate cores that can run programs in parallel than a CPU, this model should be a more natural fit to all the agents running their programs in parallel, each maintaining their own state.

Figure 2.3: protoDECK

One of the envisioned user-communities for such systems are designers of interactive buildings. First and foremost is alive and keeps on growing. Over the last couple of years, there is an interest from architects and designers to embed smart components into buildings, but the knowledge about technology has been a bottleneck for a lot of projects, so there is the incentive to help bridging this gap. Secondly, there is a lot of feedback in the creativity-process. In our experience, designers are always pushing or asking for new features once you expose them to the current features, so this gives a really natural way of seeing whether the platform we are building is useful, and where it lacks functionality.

As a test case for the Snowdrop project, we have been working together with the faculty of Architecture from the TU Delft. They have instated a lecture room that serves to experiment with interactive environments. ProtoSPACE 3.0 (Figure 2.2), as the lecture room is called, contains beamers, a complex sound system, and several interactive objects. All of which can be used to immerse the user into a interactive experience. Serving as a pilot, the Snowdrop project got involved[20] with the distributed embedded systems part of the protoSPACE that is named protoDECK (Figure 2.3). ProtoDECK is an interactive floor system, consisting of 189 tiles, each embedded with a simple processing unit, a pressure sensor for input purposes, and a light for feedback purposes.

2.2 Related work

Since this project combines a few fields of Computer Science, there related work is also spread over a number of fields. Here we will look at a brief overview of the work related to this thesis. To start, we would like to mention that this thesis is in many ways an extension on the previously published paper[51] for the GECCO'12 EcoMASS workshop.

2.2.1 Genetic Programming and Grammatical Evolution

In this project we use a technique called Genetic Programming[22, 23, 24] as introduced by Koza in 1990. We have considered different versions of GP, default tree GP by Koza, Strongly-Typed Tree GP[34] by Montana, and Context-Free Grammar GP by Whigham[53]. Eventually we chose Grammatical Evolution[6] by Ryan, Collins and O'Neill. Later on, when we wanted to improve the results, we have also looked at Tree-adjunct grammatical evolution[36] by Murphey et al., πGrammatical Evolution[39] by O'Neill et al. and many other alternatives. For genetic operators, we looked through much research, but 2 important papers that provided us with the LHS Replacement Crossover[17, 18]. The distributed dispatcher is probably mostly inspired by botnet articles in 2600 magazine[54]. The software we used to build the server and clients IRC bots is [37].

In searching for a good software framework to help us with the Genetic Programming implementation, we have considered a number of alternatives. Some alternatives were ECJ[26], JGAP[29], Watchmaker[12]. Eventually we chose EpochX[41]. All of these are Java frameworks implementing at least Genetic Programming, for more information please refer to Section 6.2.

2.2.2 Pattern Detection

For implementing a pattern detection mechanism on the programs our framework generated, we took inspiration from existing mechanisms. Most popularly, it has been covered in Automatically Defined Functions[24] by Koza. Other approaches are module acquisition (MA)[1] by Angeline, adaptive representation (AR)[46] by Rosca and Ballard, and adaptive representation through learning(ARL)[4], also by Rosca and Ballard. Also we looked at tools like *diff*[28] to determine the differences and similarities in the training set.

2.2.3 Distributed Algorithms

For simulating distributed algorithms we use an agent-based simulator called NetLogo[49]. Another simulator used in the Snowdrop project is Proto[7].

For inspiration on what distributed algorithms look like, we have used a number of already existing examples. The Firefly algorithm[52] by Werner-Allen et al., aims to synchronize distributed systems much in the same way as real fireflies synchronize the illumination of their backsides. Ant-foraging[42] by Panait and Luke, simulates the gathering of food and spreading pheromones to find food by ants. Leader election[15] tries to elect one leader in an otherwise uniform ring-shaped network. Failure detection[45] by Pruteanu et al., tries to detect the percentage of communication packages that get dropped during transmission between nodes. Churn detection[44] by Pruteanu et al., tries to measure the amount of churn in a network.

2.2.4 Global to local compilation

There have been a number of attempts in the last couple of years to build a global-to-local compiler. An interesting approach called ABM[31] was done by Miner. This system tries to predict the parameters a system must have to fulfill some behavior provided by the user. It does this by doing simulations for a number of combinations of parameter settings, and tries to interpolate the behavior happening for the other parameter settings. The simulations are referred to as forward mapping, or bottom-up approach, and the interpolations are called reverse mapping, or top-down approach. The difference with our system is that it is parameter based, and does not allow freedom in the program structure of the agents. BehaviorSpace, a plugin for NetLogo[50], also uses this bottom-up approach and allows users to record system properties while running the provided model many times for different parameter settings. BehaviorSearch[48], also a NetLogo plugin, allows the user to provide a goal for the system properties, much like a fitness function in Genetic Algorithms or Genetic Programming. The difference with our approach is that the models do not change and hence there is again only parameter variation possible to get the desired result. A system proposed by Luke et al.[27] proposes also a bottom-up approach, but it is different in the sense that it measures the parameter space more concisely where the programs are rapidly changing behavior, in order to better see the influence of different parameters. Notice that all of these approaches are changing the parameters, whereas we are changing the program structure to achieve the desired behavior of a system.

term	explanation
context-free grammar	(Language Design) Defines the valid structure of a language. A computer program that complies with the grammar is a syntactically correct program.
terminal	(Language Design) One of the two structural elements of a context free grammar. A terminal is a final and concrete part of a program.
non-terminal	(Language Design) The other of the two structural elements of a context free grammar. A non-terminal is a structural element in a program, that has to be filled with one or more terminals to be concrete.
semantics	(Program Interpretation) High level interpretation of meaningfulness of a program. What the program actually does and why it works.
syntax	(Program Interpretation) Refers to what we can express in a language, how we write down a program.
encoding	(Genetic Programming) Refers to how a program is encoded in Genetic Programming. What a program looks like before it is converted into a syntactical representation.
phenotype	(Genetic Programming) Refers to the program in its syntactical form (see syntax), the outcome version of an individual that is tested for fitness.
genotype	(Genetic Programming) Refers to the program in its encoded state (see encoding), this is the version of an individual that lives in the Genetic Programming process, and goes through mutation and crossover processes to evolve.
generative grammar	(Genetic Programming) A grammar that is used to transfer an encoded program (see encoding) into a syntactical representation (see syntax) of that program.
parsing grammar	(Language design) A grammar that is used to convert a (generated) program into a representation meaningful for the computer (parse-tree). Mostly used to interpret programs or translate them into runnable code.
GP library	(Genetic Programming) A software library that contains a full implementation of the Genetic Programming process. We have built our framework upon this library (EpochX).

Lexicon for Chapter 3

Chapter 3

Genetic Programming

Genetic Programming is a technique that comes from the family of Evolutionary Computation. In Section 3.1, we look briefly at this family. In Section 3.2, Genetic Programming is fully described, and we explain why we think this approach is suitable for achieving distributed algorithm discovery. The Genetic Programming flavor we use for the MetaCompiler is called Grammatical Evolution, which is described in Section 3.3. Of course there are also some challenges in using Genetic Programming, which are described in Section 3.4.

3.1 Evolutionary Computation and Evolutionary Algorithms

Evolutionary Computation (EC) is a field of Computer Science that takes inspiration from evolutionary strategies as posed by Charles Darwin in *On the Origin of Species: By Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*[11] [19]. These strategies are used to search for solutions to problems that are often hard to solve but possible to approximate. One requirement of EC is that individuals (a possible solution generated by EC) can be assessed on how good they solve the problem at hand. This property is the so-called fitness of an individual.

Evolutionary algorithms (EA) are a prominent subfield of the of Evolutionary Computation. Algorithms in EA use the mechanisms of reproduction, crossover, mutation and selection to achieve evolution of individuals over multiple generations. See Figure 3.1 for a typical EA lifecycle.

The biggest difference in variations in EA is usually what an individual encodes, and hence what is evolved over the generations. This can range from pa-

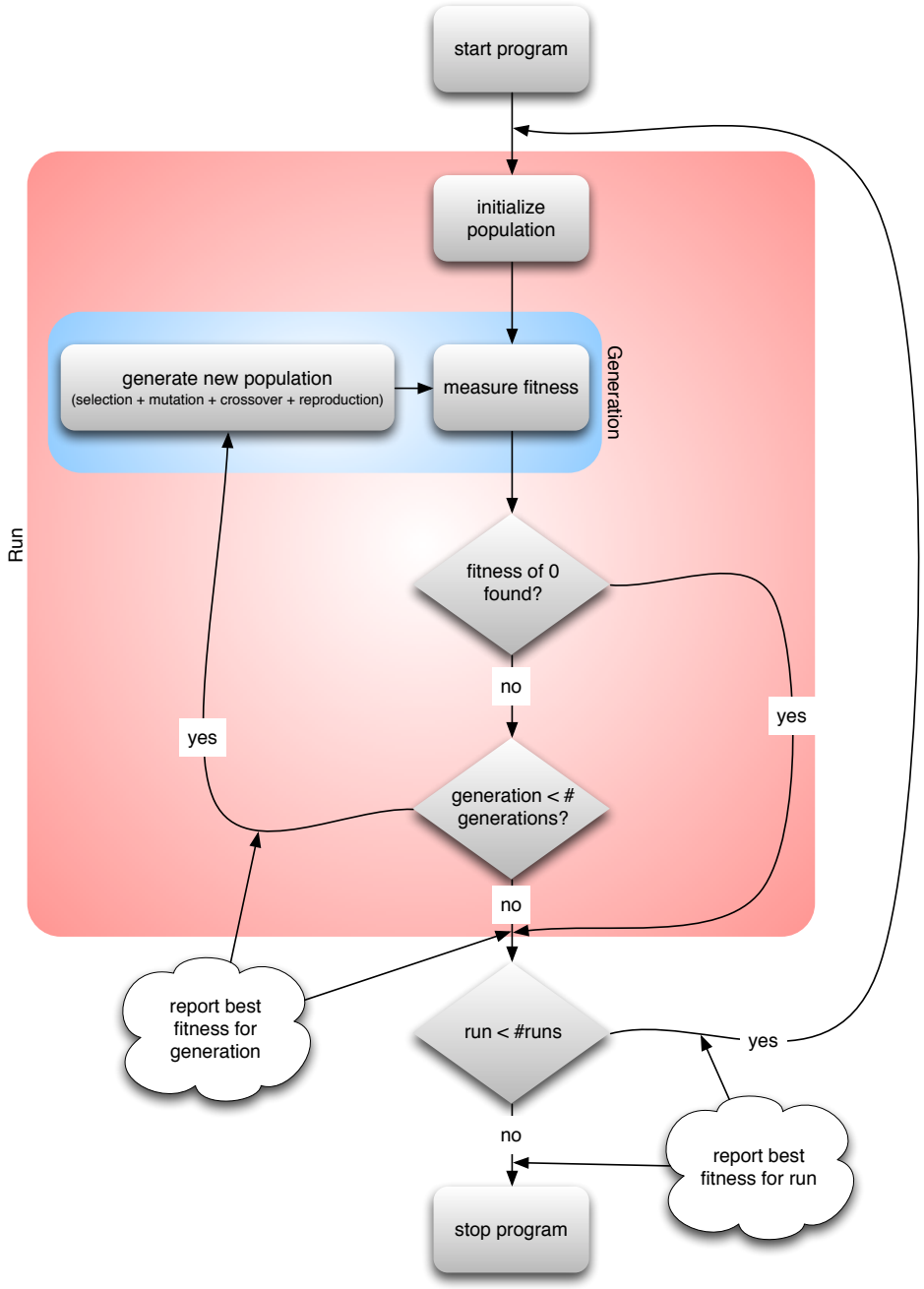

Figure 3.1: A typical Evolutionary Algorithms / Genetic Programming lifecycle

rameters used for the solution to a problem in Genetic Algorithms, to the encoding of a finite state machine in Evolutionary Programming, to a whole computer program in Genetic Programming.

3.2 Genetic Programming

The method we use for algorithm discovery is called *Genetic Programming* (GP)[22][23][24]. In GP, computer programs are evolved by mutation and crossover in order to find a program that fulfills a user defined task. We have chosen this method from the family of EC and EA, because it allows a very big freedom in solution representation. Because we have little or nu clue what a typical solution may look like for any problem specified by the user, and every problem may need a very differently structured solution, we need this freedom to support the domain of problems we want to cover with our framework.

In Table 3.1, the terms used in GP are briefly explained for use further in the thesis.

3.2.1 Initialization of a population

In the beginning of the Genetic Programming lifecycle (Figure 3.1), the GP library has to come up with an initial population of individuals. In the other parts of the lifecycle, this happens by evolving the previous population, but since this is the beginning, there is no previous population yet. This problem is solved by a process called initialization. Initialization comes up with a set of individuals to serve as the current population. The way in which these programs are generated is very dependent on the specific type of Genetic Programming at hand, but there are a few requirements for a good initialization process.

The first requirement is that the programs it generates should not be useless. For example, if the process would generate programs that are syntactically incorrect, these programs are useless, and the Genetic Programming process will suffer from this lack of diversity (it can only use the good programs), in later generations.

The second requirement is that the initialization process should generate a diverse set of programs. The diversity of programs in future generations, and consequently the fitness of the final solution, is partly dependent on the diversity of the first generation.

Since computer programs are often expressible in trees, a common way to satisfy both requirements is to impose requirements on the generated trees. By generating trees of which all child and parent nodes are compatible with each other, the syntactical correctness requirement is often already granted. To fulfill the diversity requirement, a common strategy is to impose depth requirements on the generated trees. Two strategies are full, which generates a so-called “full bushy” tree, of which all leaf-nodes (the ultimate child nodes) are at the depth

term	**explanation**
fitness function	A function returning a numerical value that describes how well a individual fulfills its goal. In our case, the lower the value, the better the goal is fulfilled (implementation dependent), with the value 0 denoting that a perfect solution was found.
individual	A single computer program
population	The set of individuals during one generation
generation	One of the iteration units in a Genetic Programming run, each having their own population
run	A single unit containing the whole Genetic Programming life-cycle, the run ends when the number of generations reaches its maximum, or whenever the perfect solution is found
crossover	A method to create 2 new individuals from 2 "parent" individuals by interchanging parts between their individuals. Crossover serves to combine good properties of individualss (Figure 3.4). Crossover is usually applied in 90% of the productions of a new individual.
mutation	A method to randomly change a part of an individual, this can range from changing the entire individual to changing one of its very specific aspects. Mutation serves in two ways, to refine existing individuals (Figure 3.2), and to find completely or partly novel individuals (Figure 3.3). Mutation is usually applied in 10% of the productions of a new individual.
reproduction	Reproduction makes a copy of the individual without permutations, and transfers it to the next generation. Reproduction is usually ignored as an option in GP.
selection	The process of picking out candidates for mutation/crossover/reproduction. Common ways to do selection are: Tournament selection, where for each selection, a number of randomly picked individuals compete and the one with the best fitness wins. Roulette wheel selection, where the chance of being picked is related to the fitness of an individual, increasing the odds of good individuals.

Table 3.1: Terms generally used in Genetic Programming

specified, and grow, which only requires minimally one node to be at the specified depth. These two strategies are combined in a popular initialization method called the ramped half-and-half initializer[47]. This initializer tries to generate an equal amount of programs for each possible tree depth between the minimum and maximum depth, with as many programs derived by the full as the grow method for each depth. This method gives a relatively diverse initial population, and we have chosen it to be the default initialization method for our framework. In Section 4.1, we talk more about the influence of initialization.

3.2.2 Genetic operators: Crossover, Mutation, Reproduction, and Selection

For the GP process to be able to evolve from one population to a new one, likely a better one, there are some mechanisms in place. Crossover, mutation, and reproduction all use a mechanism called selection in order to pick good individuals from the current population. Selection enforces the *Survival of the fittest* principle, by decreasing the likelihood that descendants of weak individuals will end up in the next population. Selection is most commonly done through Tournament Selection, in which a few individuals are randomly selected, and the one with the best fitness wins, or Roulette Wheel Selection, where individuals are selected from the whole population, but their chance of selection is dependent on their fitness.

The first, and most simple way to create a new individual for the new population is reproduction, which copies an individual, picked by selection, from one generation to the next generation. Reproduction is usually not used in the GP process.

The seconds mechanism is mutation, which transforms parts of programs randomly to achieve two goals. One is coming up with novel (sub)programs (Figure 3.3), and the other one is to refine small parts of programs (Figure 3.2). Mutation is done by selecting an individual from the current population through selection, and modifying its genetic structure such that a new individual is derived. The similarity of the genetic structure of the new and old individual can be very high or very low, depending on which part was mutated.

The third mechanism is crossover (Figure 3.4), which should combine the genetic structure of 2 parents acquired through selection in order to generate 2 new children for the new population. This serves to combine the good properties of 2 individuals in order to create individuals that contain the good properties of both, and thus have a better fitness.

3.3 Grammatical Evolution

One thing we can say about the definition of GP is that it is general. For example, it leaves it entirely up to the programmer/scientist how the encoding of programs

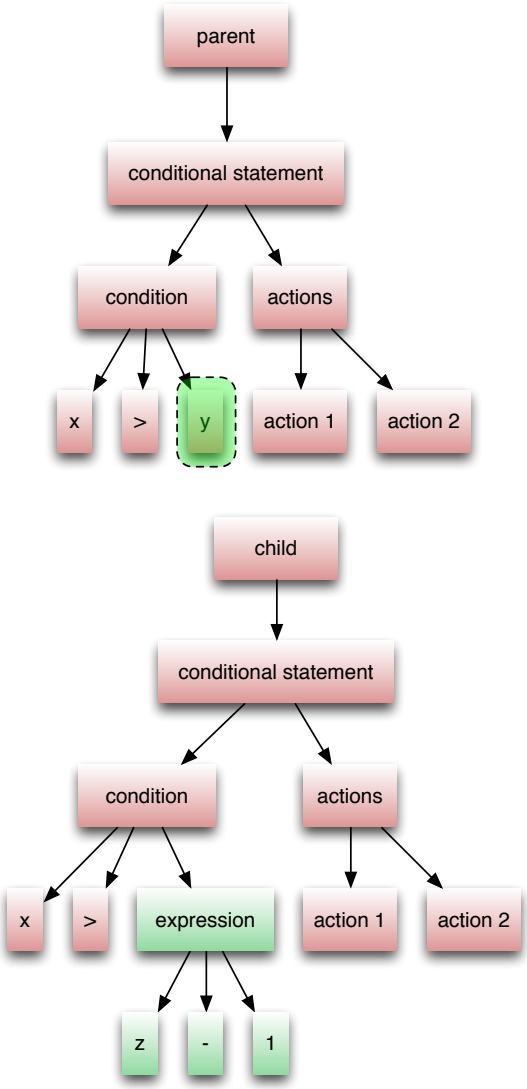

Figure 3.2: Small mutation of a program

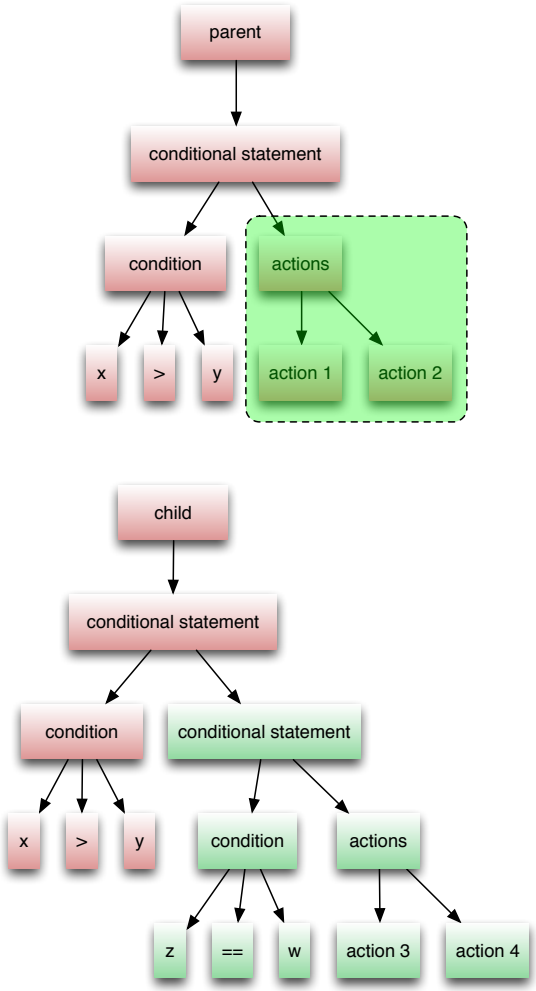

Figure 3.3: Mutation on a subtree of a program

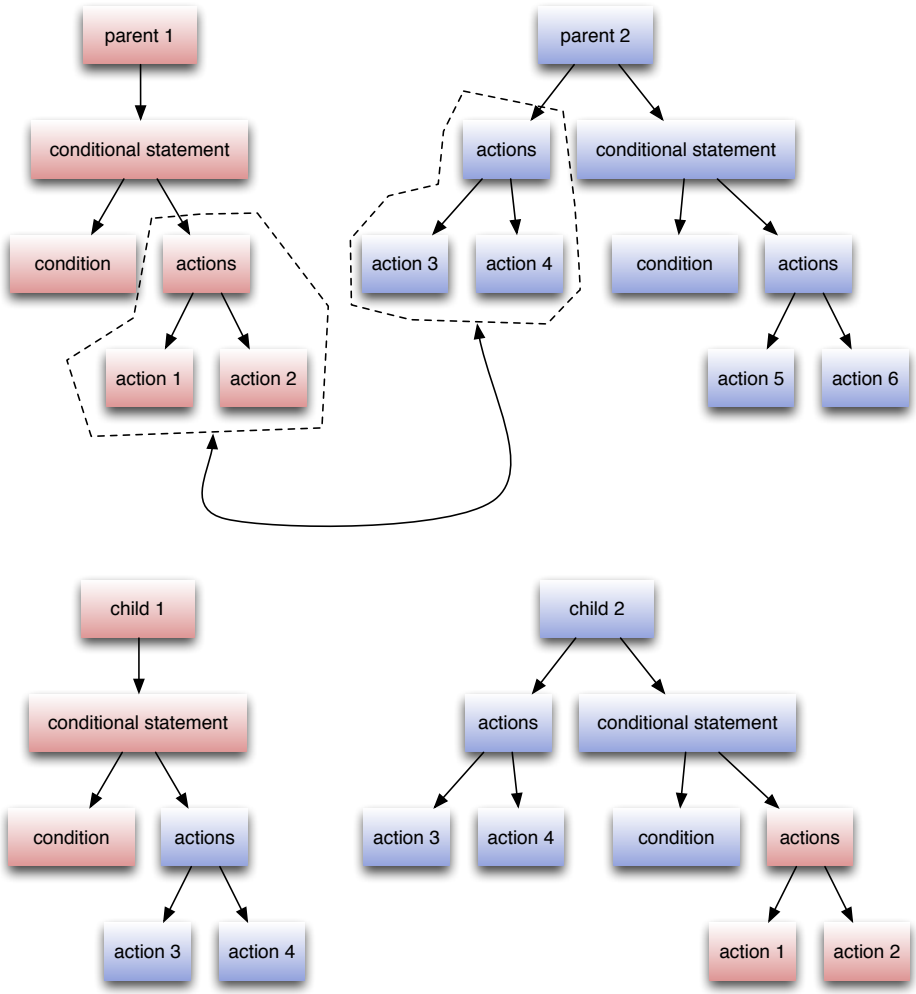

Figure 3.4: Crossover

should be. Related to that, typical operations like crossover and mutation are also left up to the programmer/scientist to figure out.

There are many encodings possible for Genetic Programming. The GP library we use comes with 3 representations by default:

- Strongly-Typed Tree GP[34] expands on the default program-tree implementation proposed by Koza in the original Genetic Programming publication[22]. The original tree was only allowed to use one data-type for all nodes in the tree, to ensure having compatible input and output types for all nodes in the program-tree. This property is called closure, which states that any non-terminal should be able to handle as an argument any data type and value returned from a terminal or non-terminal[34]. In practice, this closure property was often satisfied by letting a program gracefully fail whenever types of parents and their children were not compatible, returning a bad fitness value. However, this is bad for the search process because a lot of programs will already fail on a syntactical level, before they are evaluated semantically. STGP fixes this by adding types to all nodes, and specifying the types their children should have. In this way, compatibility of tree-nodes can be checked whenever the tree is modified, enabling trees that are always syntactically valid before evaluating them, and thus judging them purely on their semantic qualities. The GP framework lets the developer encode every different node type and its functionality in a new Java class, and forces the developer to specify the types of the child nodes.

- Context-Free Grammar GP is a technique published by Whigham[53]. CFG-GP uses a context-free grammar to explore the program space. It uses the grammar to generate parse trees and later on guide their adaptation. CFG-GP evolves the grammar during a run to steer the discovery of new parse trees. This effectively also takes care of the closure property, because grammars are syntactically valid by nature because they define the syntax. The GP framework allows the grammar to be provided as a Java string. Functionality is not included in this string, and thus there is usually an external parser needed to evaluate the programs.

- Grammatical Evolution, a widely used[25] encoding method in GP, introduced by O'Neill and Ryan in [6], is similar to CFG-GP, except for 2 aspects. GE does not create or evolve the parse-tree directly, but rather works on a sequence of integers that encode the parse-tree. To read more about this separation, read Section 4.2.4. The other difference between GE and CFG-GP is that GE does not evolve the grammar during the GP process.

Next to these three already implemented encodings, there is a multitude of alternatives. These alternatives are usually developed by identifying a problem with an existing encoding and improving upon that. A few examples are TAGE[36], which replaces Context-free grammars in Grammatical Evolution by

the more advanced Tree-adjunct grammars, because it recognizes that type of grammar could help guide the search process. Christiansen Grammar Evolution (CGE)[40] replaces the context-free grammar, by a so-called Christiansen Grammar, which adds semantic to syntactic rules in the grammar. The authors of Linear Genetic Programming (LGP)[8] recognize that it is not always desired to have a tree-structured program like the encodings above. Therefore they came up with a encoding that creates linear programs, much like a sequence of processor instructions, to be able to generate code that runs directly on a processor.

3.3.1 Why Grammatical Evolution?

For a few reasons we have picked GE to be the encoding of individuals in our framework. Looking through literature, we have not found a widely spread technique that has been picked up by the community as the definitive replacement for GE. As mentioned above, the alternative encodings usually deal with a problem in one of the existing representations, and improve upon that problem. However, for us it was not yet clear which problems were going to occur, so we decided GE would be a good option for the first version of our framework. Another reason that we picked GE is that it is easy. The way in which programs are encoded into GP, and translated into programs (see Section 3.3.2) is straightforward, and the result is readable source code.

Another advantage of GE is that, through adjusting the grammar, we can adjust the likelihood a programming structure ending up in a program. We have used this technique a couple of times to balance the chances of discovery of program structures. There are two ways in which it is possible is to adjust the likelihood of a terminal or non-terminal to be picked. One way is to try to eliminate the number of non-terminals it takes to get to the (non-)terminal you want to promote, however if that is done for too many (non-)terminals the grammar will eventually get a very flat structure, in which it the odds of (non-)terminals being picked are again equalized. Another way to promote a (non-)terminal is to use redundancy in the grammar. By referring to the same (non-)terminal in different parts of the grammar, the odds of it being used in a program is increased. This is an advantage of separating the generative grammar and the parser grammar, since in a parsing grammar, every (non-)terminal is usually only defined once to prevent code duplication and parsing ambiguity.

During the development process, we found the readability of the grammar a nice feature to overview the way in which the GP library builds up programs. A couple of times it has helped us to debug the generative process, and in combination with adjusting the grammar as described above, to cut corners for discovering certain language constructs. We think this kind of insight lacks in a purely programming oriented approach like Strongly-Typed Tree GP.

3.3.2 Encoding in GE

In this technique, programs are encoded through a given BNF grammar, and a sequence of integers denoting choices in the grammar, starting from the start rule.

If we have the following grammar:

```
<program>  := <expr>
<expr>     := <a> | <b>
<a>        := c | d
<b>        := e | <expr>
```

And a sequence of integers:

```
15 67 54 30
```

Then this will encode a program as follows:

```
<program> 15 % 1 == 0 == <expr>
<expr> 67 % 2 == 1 == <b>
<b> 54 % 2 == 0 == e
```

So the combination of grammar and integer sequence effectively encodes the program `f`, leaving the value `30` unused.

3.3.3 Genetic Operators in GE

The example above plainly describes how a program is encoded in GE, but says nothing about how we can evolve programs using that notation. To actually evolve programs, we have to look at what the MetaCompiler can do to change programs (mutation), and interchange parts between programs (crossover), doing that in GE is an actual challenge. We looked in the literature for more information.

For mutation, the default operators for GE work on the sequences of integers that, in combination with a grammar, encode a certain program. In the first default mutation operator, single point mutation, a random position in the current integer sequence is picked, and the value on that position is exchanged with a random other value. Note that it might be possible that this value in the integer sequence is not part of the program generation process, because not all integers are always used (as can be seen in the example above). In the other default mutation operator for GE, point mutation, each of the codons can be changed to a random other value with probability `pointProbability`.

For both of those methods, it can easily be seen that even though the consequences for the integer sequence are small to moderate (if `pointProbability` is small), the ramifications for program creation based on grammars are a lot bigger. This is especially true because a change in one value can carry through the rest

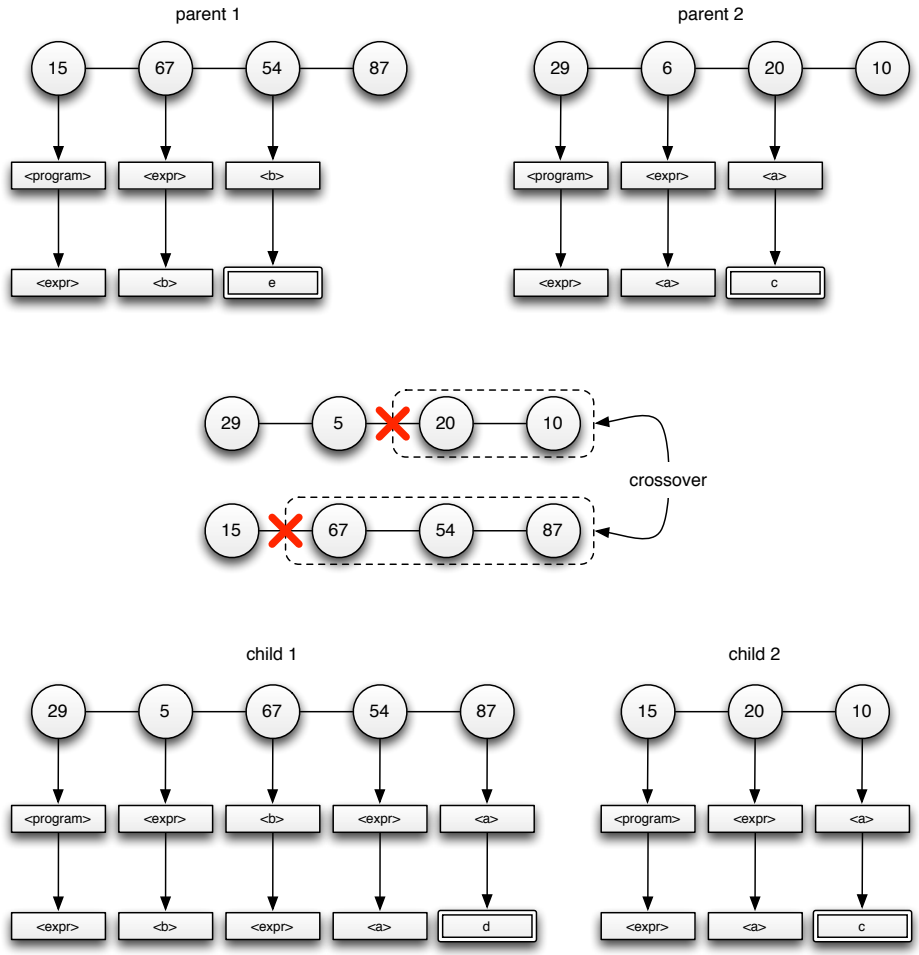

Figure 3.5: Standard one-point crossover in Grammatical Evolution

of the program creation. In other words, the integer values represent choices between different alternatives in one part of the program, but the integers right of the changed value are also influenced by the change, leading to a chain of various "different choices".

For the default crossover operator in GE, the process is a bit similar. The default operator is the one-point crossover (see Figure 3.5), in which the integer sequence for both programs is divided into two parts, a left and a right part. For both programs, this division happens at random points. Two new programs are created from the combinations of the left part of one program, with the right part of the other program. The other method, called fixed point crossover, is very similar to one point crossover. The only difference is that the crossover point (i.e. cutting point in the integer sequence) is the same for both programs.

Much in the same way as in mutation, these two operators change the pro-

grams by adjusting the underlying integers that make it up. Mutation completely changes a (part of a) program, because mutation should take care of radical change on one hand, and refinements on the other hand. However, for crossover, which should interchange useful parts between programs, the default operators implemented by the GP library do not make much sense.

3.4 Challenges in Genetic Programming

In this section, we review some of the downsides and dangers of using Genetic Programming. Some of these problems may not be specific to Genetic Programming, but it is important to mention them here, as they are influencing the outcome of our work.

3.4.1 Premature convergence

One of the main dangers of using GP, and other searching techniques is the danger of premature convergence. In other words, having the search heuristic converging to a solution that is never going to work in the first place. Because candidate solutions to a problem are weighed trough a fitness function, their representation for the search heuristic is in the end, a number. If this number is good enough, the search heuristic will consider this a good solution. Even worse, it is a relative evaluation. If all other solutions are entirely bad, the chance that the program with a relatively good fitness will be used in the crossover and mutation phase of a generation is very large. This problem is really fundamental to heuristic search methods, because having a solution that is "half-good" does not imply that we are on a path to a good solution.

What we can do about this is first to manually adjust the fitness function to "punish" bad behavior of programs. Secondly, it is important to maintain diversity in the population. This can be done through adjusting of the fitness function not to be too high (bad) for bad solutions, and not to be too low (good) for moderately good solutions. As long as good programs have lower (better) fitness values than bad programs, good programs will always win. In the meanwhile, other solutions stay in the population to maintain diversity.

Another way would be to create a multi-objective fitness function. It avoids expressing the fitness as a single number, but rather value aspects of the fitness separately. Doing this creates the possibility of introducing hierarchy between separate parts of the fitness function and thus control the way separate parts of the wanted functionality are discovered. It must be noted that this sort of hierarchy can be introduced through means of adjusting the fitness function. The big difference with embedding it into the framework is that you could use multi-objective fitness evaluation to maintain variety in the population by making sure that programs that are fulfilling different important properties do not suddenly disappear in the next generation. Another strategy that it enables is to deliberately crossover programs fulfilling different requirements, in the hope that the

combination will have indeed the best of both worlds. These are typically things we cannot do with "simple" fitness functions.

3.4.2 Search traps

While experimenting, we ran into a couple of problems that are rooted in the way Genetic Programming works. In short, the Genetic Programming process is free to do whatever it wants, as long as it plays by the rules indicated by a grammar in our case. While this may sound a bit childish, even though the GP process always plays by the rules, it can (and will!) still generate unwanted combinations.

One case we ran into was that very often, the MetaCompiler would generate programs in which a lot of variables would be assigned all sorts of convenient but trivial values. Even though the MetaCompiler played by the rules, since variable assignments were allowed at all times according to the grammar, the programs it created look erratic, and were very simple. Very often, every agent would have the same value for specific variables.

However, sometimes having all agents assign a value to a variable would give unwanted results. One very simple example to demonstrate this problem was when we tried to invent a synchronization algorithm using the building blocks of the Firefly synchronization algorithm (see Section 5.3 for more information). What the MetaCompiler came up with effectively granted our wishes to have every agent in the system be synchronized: reset the clock of every agent to 0 at every round.

To prevent this kind of behavior we introduced "conditional assignments" in our grammar, which is basically an if-statement and an assignment combined. If the condition of the if-statement can be fulfilled, the assignment will be made. What we hoped to achieve with this measure was that assignments would only fire on certain conditions, and that it would bring some diversity among the agents in terms of variable values. Unfortunately, the condition that would be generated for the conditional assignment would often be very trivial, and the MetaCompiler seemed to circumvent the barrier we introduced. For example, as a condition it would use `1 == 1` or `1 < 2`, or anything similar, and thus situations like the one described above were still to easy to get into. Effectively, adding conditional assignments did not help solving the problem, but it was a nice shortcut to having and if-statement and assignment combined, and hence we did decide to keep it. We also reintroduced (normal) assignments, for the general purpose assignments, or for use within existing if-statements. In the process we also figured out that sometimes, it is nice to be able to read a certain variable, but not to overwrite it, and hence we introduced the "volatile" keyword in the configuration file to indicate a variable may be used to store values in.

3.4.3 Use of a dynamic runtime

One case we regularly ran into while testing the MetaCompiler was that our examples would be too static. Because each generated program is evaluated a number of times, and the outcomes are (by default) evaluated by taking the mean of all outcomes, we want to leverage this in order to see if the program does well under a number of circumstances. Even more important, sometimes it is mandatory to use dynamic elements in the runtime of programs. For example, in examples where we wanted the MetaCompiler to come up with an estimate for a network property (e.g. failure rate detection, churn detection, or network size estimation) based on the local variables, it would often happen that the MetaCompiler would just estimate a static value. Obviously, if the values that should be estimated are the same in every experiment, it is very easy for the MetaCompiler to find a program that returns a good value. Through every generation it can adapt the value to return, and gradually move it to the right answer.

A (partial) solution to this problem is to vary the value to estimate at every simulation. In this way, every static value will be wrong a number of times for every program, and this should discredit picking the static value. It should be noted that having an outcome with a static value does not help us in any way because we aim to generate algorithms that are usable in real-life scenario's, where system properties are dynamic Unfortunately, dynamic runtime properties are only guiding us in the right direction of finding a good algorithm, but there are still other problems. What we regularly found after the introduction of dynamic variables at runtime, was that the MetaCompiler would aim for a value that was as close to all possible values as possible: the mean. Of course, this is also a dependent on the fitness function; If estimates that are off in a simulation are strongly discouraged by the fitness function, this could change the situation, but in our experience picking a static value was still easier than having a very complicated estimation function that has to be gradually discovered over generations of the MetaCompiler, and thus the static value would often be dominant.

Of course there are other ways to introduce dynamicity into an environment, for example by changing the way nodes are distributed in the simulation world, changing the size of this world, or changing the population size. All of these options are possible in the MetaCompiler, and encouraged if it gives a more stable algorithmic outcome. In addition, it should also be noted that adding dynamicity does not come at any cost except for a slightly more complicated configuration file.

Another interesting approach to capture and discredit the cases that use static values could be to apply deep analysis (i.e. semantic evaluation) on the generated programs. There are a few things we can conclude after having giving it some thought. First, the deep analysis of generated programs would add a whole new layer to the MetaCompiler, and in a way abandon the Genetic Programming approach, where each program is evaluated against the same fitness function. Second, static values can come in many forms, they might be a numerical value,

a variable that never changes, or a numerical function that for some reason always evaluates to the same result, and hence it can be hard to detect. In addition, for some programs, using static values can be essential. For instance, a program that tries to reach consensus in a network might demand that a node looks at least at 3 neighboring nodes before setting a variable. In that case, the value of 3 is essential for the algorithm to work, it should not be able to be 1 or 2, and hence it should be a static value. In conclusion, we would like to say that static analysis of generated programs is interesting, but at the same time difficult, and not applicable everywhere.

term	explanation
(Program) node	(Language design & Genetic Programming) When expressing a computer program as a tree, each node of a tree is contains a code construct. This construct is in turn part of the construct of the parent node, and maybe it contains constructs of its own, as childs in the tree
EpochX	(Genetic Programming) The library we have built the MetaCompiler upon, that implements the Genetic Programming process, also sometimes referred to as "GP library"
context-free grammar	(Language Design) Defines the valid structure of a language. A computer program that complies with the grammar is a syntactically correct program.
terminal	(Language Design) One of the two structural elements of a context free grammar. A terminal is a final and concrete part of a program.
non-terminal	(Language Design) The other of the two structural elements of a context free grammar. A non-terminal is a structural element in a program, that has to be filled with one or more terminals to be concrete.
semantics	(Program Interpretation) High level interpretation of meaningfulness of a program. What the program actually does and why it works.

Lexicon for Chapter 4

Chapter 4

Genetic Programming Enhancements

After we implemented the MetaCompiler using the default Genetic Programming approaches, we found that there were some issues (some of which are described in Section 3.4). We have implemented a few countermeasures to try to overcome these challenges, which we will describe in this chapter. For each countermeasure, we first introduce the actual problem, followed by our intuition on why this happens, followed by the countermeasure we came up with.

4.1 Initial generations

One of the first problems we encountered when the MetaCompiler was up and running, was that the solutions it provided were often very simple and short. In the genetic programming process, all the programs that are generated are derived from the programs of the first generation, either through mutation, or crossover, but most often, both. Since the programs in the first generation were already very simple, the odds of finding advanced programs were not very high.

If we think about this some more, we find that this is also due to the nature of the selection process in Genetic Programming, not only the evolution.

Especially crossover between two simple programs often yields other simple programs. Since it uses genetic material from two simple programs, the new program is also likely to be simple. Because the newly created programs are probably not distinguishing themselves functionally, or are even programs that were discovered before, the pool of new and potent programs quickly runs dry. In turn, this will propagate to the next generation and the one thereafter. Obviously, this will have a gigantic impact on the discovery of good solutions for a given problem, and thus this must be solved.

In the process of finding out why this happens, we looked at papers that deal with the initialization problems in Grammatical Evolution. Important papers on this subject are [16], and [35].

What we found was that the initialization methods were not much more advanced than ours, and that there was likely another issue causing the problem of generating programs that are too simple. However, we had to solve the problem in order to continue working on the MetaCompiler. Since implementing another initialization method would not solve it and take up valuable time, we decided to solve it in another way. What we did was to change the fitness function in the first N generations (we called "expansion generations") not to look at the fitness of the solution but at the actual size of the program. By doing that, we were able to create a more diverse base of programs before starting the search for programs that fulfill the actual fitness function as provided by the user.

There was a small problem with this approach that we did not foresee directly and could cause a very strange outcome of the Genetic Programming process. Since we dynamically change the fitness evaluation without consent of the EpochX framework, the values it gathers for the fitness are undistinguishable for the framework, and hence, if the "expansion fitness" gives lower values than the user provided fitness, big programs would win instead of good programs. What we did to overcome this was to define a fitness function that returns values ranging from 0 to 1, while the "expansion" fitness was allowed to use the range from 1 to infinity. However, this imposes a very strict limitation on the designer since the upper bound of values is not always known, and thus it is hard to scale to a 0–1 interval. Another problem with this is that normally this would cause the framework to return a value of `INFINITY` for faulty programs, while now we should put all programs into a 0–1 interval, should that include the faulty programs? The last problem we ran into a couple of times while testing was that, because the restriction for fitness values was not made explicit, and was conditional (i.e. without the expansion generations, there was no need for the restriction), we made the mistake of using arbitrary fitness values, which got then entangled with the "expansion fitness" values, giving random big programs as a result for the genetic programming process.

Unfortunately we only found out later that the initialization problem was mostly happening due to a bug in the code (mostly, because we find the existing initialization methods are still very simple). The problem was that we were setting up the maximum initialization tree depth, that determines how deep programs may get in the initialization process (i.e. how many non-terminals may happen before the grammar finds a terminal value). However, we forgot to subsequently adapt the maximum depth for programs, so the initialization process was actually able to generate programs that were rejected by (the parser of) the GP framework, because that decided the program was deeper than allowed, and thus the programs would end up being `null` values in the framework. In order to solve this, we automatically set the maximum tree depth to the maximum initialization depth, but the user may override this in the configuration file if desirable.

With hindsight, because the fitness function of the "expansion generations" becomes entangled with the user defined function, using them is discouraged, but because it is disabled by default, we decided to let it up to the designer whether he or she wants to use it or not.

4.2 Tree-based operators

Due to the problems of the default operators for Grammatical Evolution (see Section 3.3.3) we have decided to improve them in order to smoothen the process of finding new algorithms. As mentioned, the genetic operators for Grammatical Evolution operate on the integer sequence that in turn encodes a program. Because the operators do not consider the syntactic structure of the programs that they encode, there is a big difference between the syntactic program that is encoded before and after the operation is done.

We wanted to improve on that such that the transitions that happened in programs would become more incremental. To shortly repeat their purpose as stated in Chapter 3, the main goal of crossover is to exchange useful parts between two programs, and the main purposes of mutation are to refine bits of programs, and come up with total new structures or even whole programs.

In the process of finding out what would be the best approach, we searched for papers that also had this problem and maybe even solved it. We did not find many. This is surprising since Grammatical Evolution is such a widely used method for Genetic Programming, and the problems with the operators, especially the crossover operator, are so obvious.

During the GECCO 2012 conference we asked Una-May O'Reilly from MIT, a prominent researcher in the Genetic Programming, her opinion on incremental changes by mutation and crossover. We mentioned that crossover is designed to mix the behaviors of 2 parent programs, but often the child programs are functionally very different from either parent program. She asserted this statement, and noted that in practice, crossover is sometimes abandoned, leaving all evolutionary tasks in the GP process to the mutation operator. Even though this is almost an approach opposite to the pattern detection and protection we discuss in Section 4.3, because mutation will happen very often in every structure, it would be an interesting approach to try.

4.2.1 Attempt 1: "Random tree node"

Sometimes it is very hard to find research papers meaningful to a problem you run into. In this case, because we did not find any paper that would solve our problem, we made an attempt to solve the problem at hand ourselves. To do so, we have defined a few requirements we wanted the operator to comply with:

- There should not be a disruption in the "right side" (or ripple) of the program when an intermediary block is replaced. This is a common artifact of

operators for GE, but the effect is not good for the search.

- The programs that are generated should be able to reach every part of the grammar, because we want the framework to be able to recombine all building blocks, as long as it is syntactically valid
- Preferably, it should be more or less compatible with the current GE implementation (the more compatible, the less code should be rewritten, which would be nice)
- Crossing over a part of the program should yield straightforward results (i.e. the part that is crossed over should yield the exact same code that it did in the old program). This seems obvious, but it was not true for the old GE operators.
- There should be possibility to slightly adjust bits of programs in order to gradually move towards a better solution, in order to be able to evolve a given program towards a, possibly local, optimum

In one case we seemed to have found a viable candidate for the structure that fulfilled a few of these points. It was based on the following observation: the grammar contains the non-terminal called `programRule`, which is basically the beginning for every standalone statement in the program. The best Computer Science term to describe it would probably be an expression. Our idea was that since this `programRule` is the most basic building block in the grammar, you could write each program with random seeds. These random seeds would in turn stand for a sequence of choices that are made in the grammar, starting from the `programRule`. Whenever a `programRule` would recursively call another `programRule`, a new random seed would be used to build up the child `programRule`.

What this representation does well is that it encodes pieces of the program that are sensible units, like one expression, or a block of expressions. These units could be used for crossover in programs, in a way to protect the unit, as well as the information in the other units situated around it. Also, mutation could be done by just changing the integer that is the seed of the random generator, thus creating a whole new `programRule`.

Unfortunately, there are also some disadvantages of the representation that eventually kept us from implementing it. One disadvantage is that the granularity of the units created by the random number generator might be too big. For instance, an if-statement (excluding the recursive call to one or multiple programRules), would be encoded with one random seed. If one random seed represents grammar choices that form `if(x<23)`, and a good solution would need `if(x<25)`, then you would expect this problem to be solved by mutation, which should be able to only change the compare value from 23 to 25. If the mutation operator would only have to replace the number 23 by 25 by guessing the right random number, this could be likely to happen in a number of generations (if the

program with the value 23 already scored a decent fitness), and a good solution would be found. However, because the mutation operator would have to come up with a random seed that also generates the choices for `if`, `x`, `<`, and of course `25`, the odds of getting the good solution reduces quickly.

One other slightly more theoretical disadvantage was that there is no guarantee that certain parts of the grammar can be traversed, because this would imply that the range of all possible seeds covers all choices in the grammar, something we do not know theoretically, but it is likely to be true because of the large number of possibilities.

Even though the separate `programRules` are protected against unwanted destruction in this scheme, the hierarchy of the program is not. Since every recursive call to a programRule is fulfilled by the next seed in the sequence of random seeds that represents a program, the hierarchy of the program is not maintained whenever an programRule gets replaced by another one with a different number of recursive calls to programRule. In the worst case, a programRule that is very important for a program, and is at the top-level of the program (i.e. not conditional but always called), could end up in a very niche or even impossible location, making the important programRule effectively non-existent. The other way around, a very destructive program rule, or something that should happen rarely might suddenly become very prominent in the program. Even though these kinds of radical changes are allowed in Genetic Programming, but we think it deserves mentioning.

Because of the reasons provided above, we are very skeptical about the method that seemed to be nice at first. Therefore, we decided not to implement the solution, and to try to think of another better suited approach, and also look for solutions in literature again.

4.2.2 LHS Replacement Crossover Operator

Where we were not able to find a good representation in the first attempt, we did succeed this time and found [17]. This paper describes the Left Hand Side Replacement Crossover operator or LHS crossover, that implements a tree type crossover similar to crossover types found in program representations other than Grammatical Evolution, namely Strongly Typed Genetic Programming. Furthermore, the paper shows very positive results for the LHS crossover operator after having compared it to several other types of operators for GE crossover, so that was a good reason to choose for the implementation LHS crossover.

The LHS crossover operator operates on the tree representation of a program rather than on the integer sequence encoding it. It starts out by building up the tree representation for the combination of a grammar and an integer sequence. Once it has made this representation for the two crossover candidate programs, it selects a random node in the first program.

It is important to remember that every node in the tree stands for a non-terminal rule and number to pick a production from that non-terminal rule. In

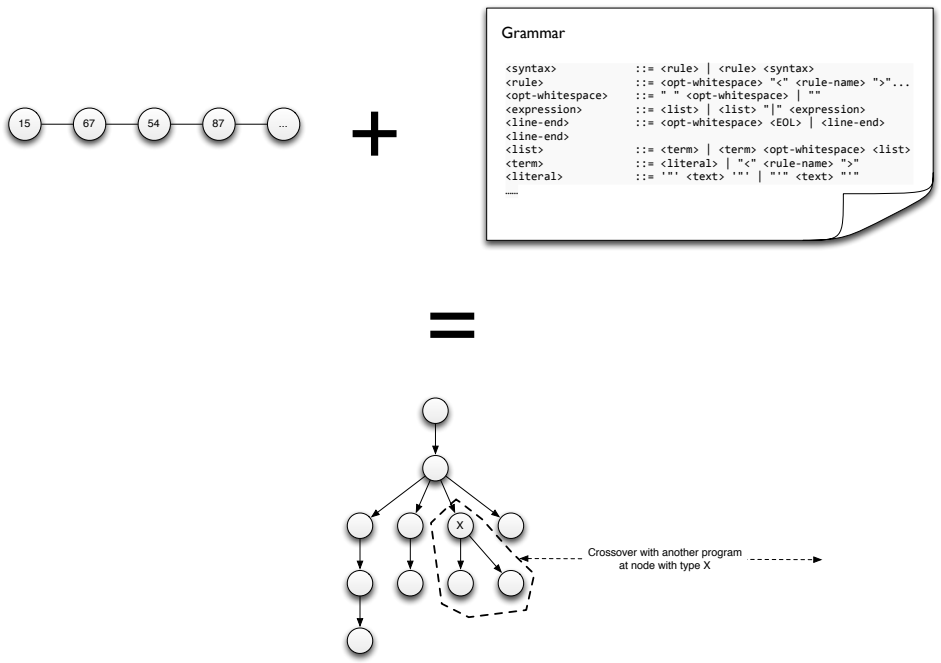

Figure 4.1: LHS Replacement Crossover

order words, the tree represents a program, and every node in that tree is a choice made in the grammar in order to achieve that program. For implementation specifics, please see Section 6.5. The non-terminal decides the syntactical type of the node, so each node we would swap it with, having the same non-terminal, is of syntactically the same type, and thus the program remains syntactically correct after having swapped the 2 nodes between the programs. This gives us the same benefits as Strongly Typed Genetic Programming[34], but without any extra administration because it is taken care of by the grammar. It is quite logical, and argued in [34], having types tightly restricted has a positive effect on the search space, because only syntactically correct programs exist in the population, keeping it "clean" and diverse. On the other hand no effort is required in order to evaluate programs that will never be able to run in the first place.

When doing crossover with the usual two parent programs, after having selected the node of a particular type in program one, we want to find a node of a matching type in program 2. We do this by putting all of the nodes of the program tree of program 2 into a list that we randomly shuffle with the Fisher-Yates algorithm[56]. Next, we go through all elements of the list to see if there is a node of the same type as the selected node in program 1. As soon as we find a similar node, we proceed with crossing over their subtrees between the program (Figure 4.1).

After having done the crossover, we have 2 new program trees. By traversing

the tree in a depth-first manner, we can rebuild the Grammatical Evolution integer sequence, such that the new crossover method is compatible with the rest of the framework.

If we look at the requirements for a good crossover operator as we stated in 4.2.1, we see that this operator scores well for all criteria:

- Ripple effect: as the LHS crossover operator preserves the structure of the program and only replaces the subtree of the selected nodes, and goes back to the integer sequence representation of programs, there is no
- The encoding of a program should be able to reach all parts of the grammar: since this crossover only goes to program structures already known in the population, so this task is left to the mutation operator to deal with, see Section 4.2.3 for information on the LHS-like mutation operator that handles this.
- Compatibility with GE: since the LHS crossover operator only uses an intermediary representation of the program during the process, and reverts back to the default GE representation before finishing, it is fully compatible with GE. It does assume and check that depth first traversal of the grammar is being used by GE, other traversal methods will require extra implementation.
- Results should be straightforward: Because the operator transfers nodes with subtrees, the programs will be the same as the original, except for the exchanged parts, very straightforward indeed.
- Possibility to slightly adjust the program: The size of the subtree depends on the node selected in the program, but it can change very fine-grained pieces of the program if it picks a node with a very small subtree, making it possible.

4.2.3 LHS-like mutation operator

Because we were not yet satisfied with the default mutation operator, we have also looked for a suitable candidate to replace it. After having implemented the LHS crossover operator, we were still looking for this replacement, and we decided to make a mutation operator that was inspired by LHS crossover. To reiterate what the mutation operator should do (Table 3.1): it should alter existing programs, and find novel program structures. Since the default operator usually breaks the structure of the program because it radically changes the path that is chosen when traversing the grammar, it probably changes the whole program from the point of mutation.

After having seen the effect of LHS crossover, we envisioned that a good way to do mutation could be to start from a random program node, dispose its children, and randomly pick grammar productions from there in order to make new

child nodes. In this way we are both able to refine programs, when mutations take place on nodes with few or no children, or to come up with new structures, when doing mutation on a node with a bigger number of children. After the mutation has taken place, we can again reconstruct the integer sequence for GE by traversing the program tree, to make it compatible with the rest of the framework. Because this approach was more likely to preserve the program structure and improve on the programs than the default mutation operators, we implemented it to work together with the LHS crossover operator.

4.2.4 Genotype-Phenotype Mapping

One aspect of Genetic Programming we have not touched up until now is the notion of separating the genotype and phenotype representation. In our case, the genotype is the integer sequence that encodes each program. The phenotype is the actual outcome program. If we look at the way programs are encoded in integer sequences, we see that there is room for integers that are not used to construct the program, because all non-terminals in the grammar have already been fulfilled (see Section 3.3). Some papers, including [5], [21], and [30] argue strongly that it is important to include some redundancy in the genotype-phenotype mapping. The redundancy in this case is the part of the genotype that does not get used to create the phenotype, but ís altered by the genetic programming process. By having this, advanced structures are able to evolve before ending up in the phenotype. This is important because undeveloped structures may exhibit a very bad fitness until they are fully developed, effectively meaning they will be erased from the population if they were directly visible in the phenotype.

With the LHS-operators, we employ a direct mapping from the genotype to phenotype, and thus we miss this redundancy in order to develop hidden structures. In other words, the LHS crossover and mutation operators are both unable to operate on the part of the integer sequence that does not belong to the program tree, and they are not able to change the program in such a way that it suddenly makes use of previously unused parts of the genotype. We do however keep track of the remaining integers that are not used in building up the program, and we append them back to the program whenever we revert back from the tree to the integer sequence representation. In this way we enable future extensions to make use of this redundancy. One simple solution would be to mix crossover operators at runtime, such that both LHS operators, as well as the simple operators for GE are used, but this would lead again to some regression in maintaining the structure of programs, something we wanted to use the LHS operators for in the first place.

4.3 Pattern Detection and Protection

Even though we have implemented crossover and mutation operators that are nowhere near as destructive as the default GE operators, there is still a high

chance that important structures of the program are altered. We would like to detect patterns that exist across the best programs and encapsulate or at least strengthen them such that they are unlikely to be changed, and to increase the chance of other parts of the program being adapted. We were not the first one to think of this strategy. Already in the original Genetic Programming paper[22], *encapsulation* was mentioned as an additional genetic operator, to capture some subtree of a program one unbreakable element. Most popularly, it has been covered in Automatically Defined Functions[24]. Other approaches are module acquisition (MA)[1], adaptive representation (AR)[46], and adaptive representation through learning(ARL)[4].

Because of our particular program structure and grammar based approach, we have implemented a similar approach suited to our software that exhibits the same kind of behavior as the works references above. We found out that it is very hard to quantify the similarity of 2 programs, or sub-structures in a program. Checking for equal structures can easily be done by trying to match as much of a structure as possible while traversing the program tree in two programs. However, this gets more difficult whenever parts of the overall structure are slightly different. A greedy matching algorithm would stop finding similarities on the first encounter of a difference, and thus the search for structural elements that are important for good programs would already stop at the first inconsistency between two programs. We have also considered using a "diff" algorithm (software that is able to compare 2 files and output the differences) to capture the similarities and differences between programs, but since this would only consider the source code that was output, it would be unable to consider the underlying structure of non-terminals in the grammar.

Since we were unable to find suitable solutions in literature we came up with our own approach (Figure 4.2) to detect and protect patterns in the programs. Based on the observation that approaches mentioned above are top-down approaches, we decided to look at a bottom-up approach to solve this problem. First we take the N best programs of a generation as a training set, a configurable number to determine the strength of the pattern detection algorithm. Then for each program in the population, including the best programs, the algorithm simply looks at every program node in the program tree, and its parent. If that parent-node relationship also exists in a program from the training set the connection is strengthened. If it exists multiple times in best programs, the connection is strengthened just as many times.

While every step is very granular, we think it works because every step adds very little protection, but it works on a large scale. The last step of the algorithm is to normalize the protection to a 0-1 interval, in which 0 is unprotected, and 1 is the strongest protection a connection has, being unmodifiable. When the mutation or crossover operator try to adjust the program node, a random number between 0 and 1 is picked. If that random number is bigger than the strength of the connection, the operator may succeed, if it is smaller the operator tries to find a new program node to apply mutation or crossover. In this way, it protects

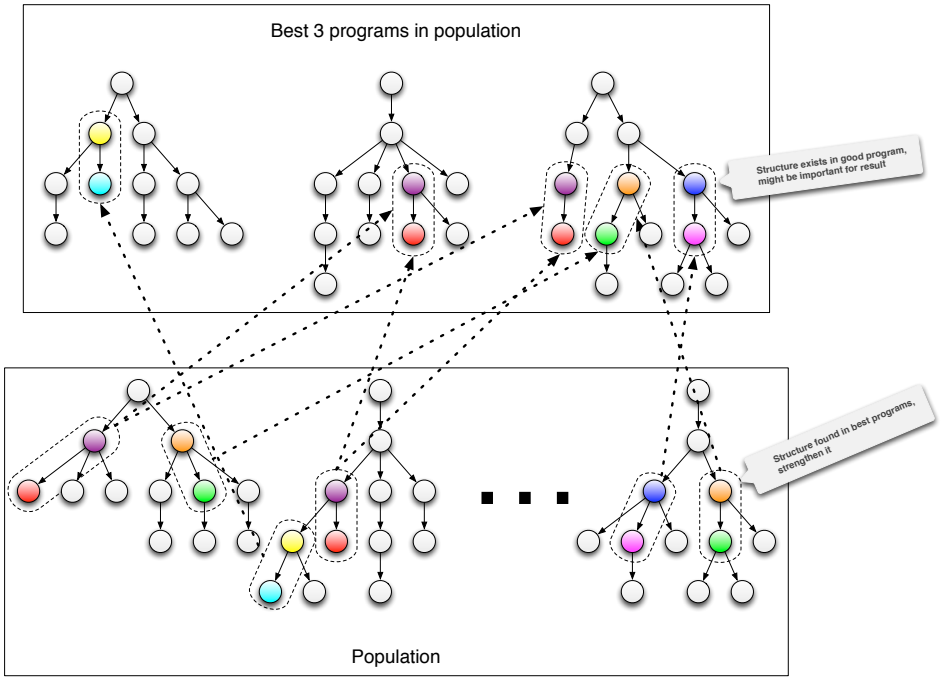

Figure 4.2: Pattern detection

from breaking important structures, and instead tries to change other parts of the program.

A possible downside of using this pattern protection mechanism, is that it can protect patterns that have not yet been fully developed. In that case, it will become almost impossible for the mutation and crossover operators to modify that pattern to become better, and the GP process gets stuck at a suboptimal point. We test the influence of pattern detection and protection in Chapter 7.

4.4 Distributed dispatcher

The number of experiments we need to run to get a result from the MetaCompiler are roughly runs × generations × population size × simulations per program, excluding the caching of results for already known programs. Over the course of the project, we have been running experiments to assess the functional performance of the MetaCompiler. For most of the experiments, we could use a to see if it performed as expected, but for more serious experiments we needed more time and computing power.

Some differences are that to get stable measurements we will want to do our experiments multiple times (runs). To know for sure that a program is valid, we will want to have a good number of simulations per program. And, to invent useful algorithms, we will usually want to increase the population size.

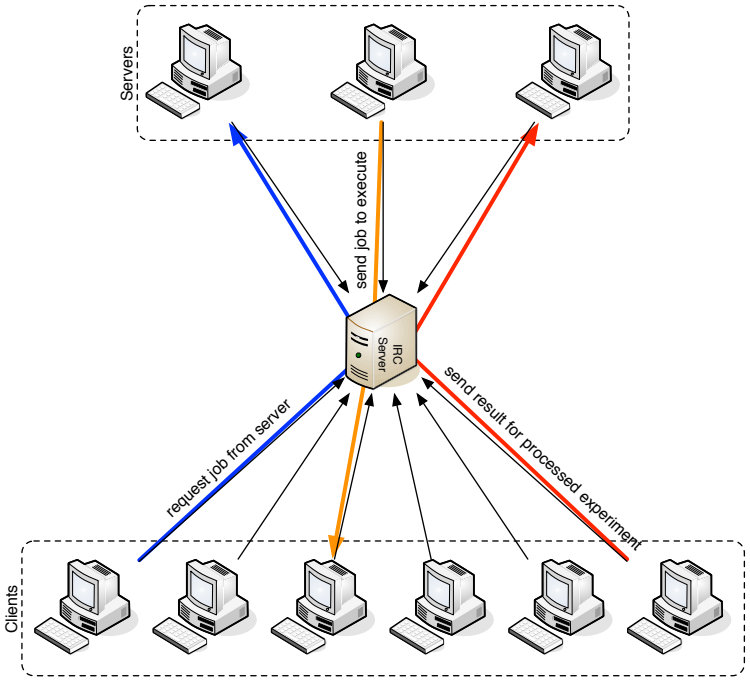

Figure 4.3: Distributed dispatcher

The default way to run the MetaCompiler is to use a single machine, possibly with multiple cores, which are fully used. We found it is often difficult to find a machine on which there is both time you can use, and there is sufficient computing power. Also, if a computer is powerful, often someone else is in control of it, and we have experienced multiple times that our task was suddenly killed, with unknown reason. Another option was to use my own laptop, which is quite powerful, but this was not really practical with as well.

To solve these problems, we have built an extension to the MetaCompiler to distribute tasks over multiple computers (Figure 4.3). The system is designed to have two types of computers: servers, that delegate tasks, and clients, that receive and process tasks, and return the results to servers afterwards. Also, it is designed to be multi-server and multi-client, such that we can run multiple experiments in parallel on the same base of clients. To get a good trade-off between communication and distribution of experiments, we chose to send each separate program, but let it run on one client as many times as needed to get a stable measurement (the `simulations_per_program` setting).

The system is also resilient to failures on the client-side. Each client is allowed to fail at any time, such that it cannot return the result of an experiment anymore. In this case, the server will delegate the experiment to another client, and wait for the result to come back from either one of them. Also, servers are allowed to lose their internet connection or go in standby mode, they will reconnect afterwards

and continue delegating experiments. The only thing we really need to take care of is that the server-process does not get killed, or we will have to start it again.

One important note that has to be made with regards to the usability of the distributed dispatcher, is that there is some overhead time between the starting of a server and the actual execution of jobs, and returning results by clients.

See Section 6.6 for details on the implementation of the distributed dispatcher.

term	explanation
fitness	(Genetic Programming) A score for how well a program fulfills a given task. Usually expressed in a single number.
parsing	(Language Design) The process of processing a string of text (a program generated by our framework in this case) and converting it into a more meaningful representation for a computer.
parse tree	(Language Design) The result of parsing, in which the program is represented as a tree structure.
node	(Large-Scale Systems)A separate, autonomous element of a Large-Scale System
round	(Large-Scale Systems) The programs that run on nodes of Large-Scale Systems run for an indefinite amount of time, each time repeating the same code. Each repetition of executing the code is called a round.
NetLogo	(Large-Scale Systems) A simulator for multi-agent systems. The MetaCompiler uses NetLogo to run the code our framework comes up with, and evaluate its fitness.
GP library	(Genetic Programming) A software library that contains a full implementation of the Genetic Programming process. We have built our framework upon this library. (EpochX)
MetaCompiler	The global-to-local compiler we present

Lexicon for Chapter 5

Chapter 5

Distributed Algorithms for Large-Scale Systems

5.1 Defining Large-Scale Systems

In order for us to build a framework that tries to come up with distributed algorithms for large-scale systems, it is important to specify what we mean by these terms. Typical large-scale systems we aim for are embedded networked systems, such as Wireless Sensor Networks, MANETs, swarm robotics, sensors, the internet of things, or pervasive computing. These kinds of systems have some important properties. The systems are composed of a vast number of autonomous elements. Since maintaining a connection to every component of the system from a single point is infeasible, there is a lack of central control of the system. Usually system components are geographically restricted in communication because of radio signal strength and interference, and thus for every element there is limited capability to communicate with other elements of the system. Because of the distributed nature of the systems, components lack a full view of the system and consequently they are only able to act upon local information. Also, embedded systems are usually simple devices, both on a hardware, as a software level, such that they can only do relatively simple tasks.

5.2 Structure of generated programs

In order to let the MetaCompiler generate programs that are useful for large-scale systems, we have to provide it with guidelines about what kinds of programs it can come up with. We do this by providing the GP framework with a so-called language definition it can use to generate programs with. Because we have influence on what programs the GP framework is allowed to generate, this

is an important place to target the GP framework towards our problem domain of large-scale systems by creating a simple custom language. In designing the language we have made a couple of important decisions.

One decision that influences the flow of programs enormously is whether or not loops or jumps are allowed, specifically while- and for-loops. Because nodes in our definition are supposed to "infinitely" rerun the same code (a round), the programs already have a loop-like structure. We would like our simulations to end in a finite number of rounds, if a round takes forever to execute, this will cause the simulation to run forever. In other words, nodes can behave repetitively over multiple round, but at least rounds will progress. Hence, we decided not to include loop or jump structures in our language. In Turing machine terms, this influences the graph of state transitions: since there is no way to jump back to a prior state, and the number of states is finite[58], the graph of state transitions is a tree, and the program will eventually finish as long as the program progresses from one state to the next. It should be noted though that since we allow runtime libraries, loops or jumps can still be used, and thus still be an issue in our programs. At least it will not happen because the GP combinatorial nature created such a program structure.

Another important decision in designing the language was the strict separation of global information and actions from node (individual) information and actions. We wanted to prevent the GP library to come up with programs that use either global information or global actions to achieve the goal set by the designer. As mentioned above, individuals in a system (nodes) lack a full system view, and therefore global information is non-existent for a node. Obviously global actions are also non-existent in such a system, because everything is the result of the composite of node actions. We have however maintained the ability for the designer to use global information and actions to be called at the beginning or end of every round, for administrative purposes, because we noticed that without them, measuring the fitness of a program was often very hard. Next to these "static" calls to global functions, we also allow static calls to node functions. This is very helpful when we want force a node execute some code every round, for example to force movement, or to simulate decreasing battery life by decreasing a counter every round.

Once we have this language, we can give it to the GP library to generate programs we can translate into code running on our simulator, in order to assess the fitness of the generated programs. The lifecycle of a simulation is depicted in Figure 5.1.

5.3 Example Algorithms

While assessing the performance of the MetaCompiler, we were confronted with the question: what do we want to test our software with? Clearly, the software allows designers to input a specific aggregate behavior and get an algorithm in

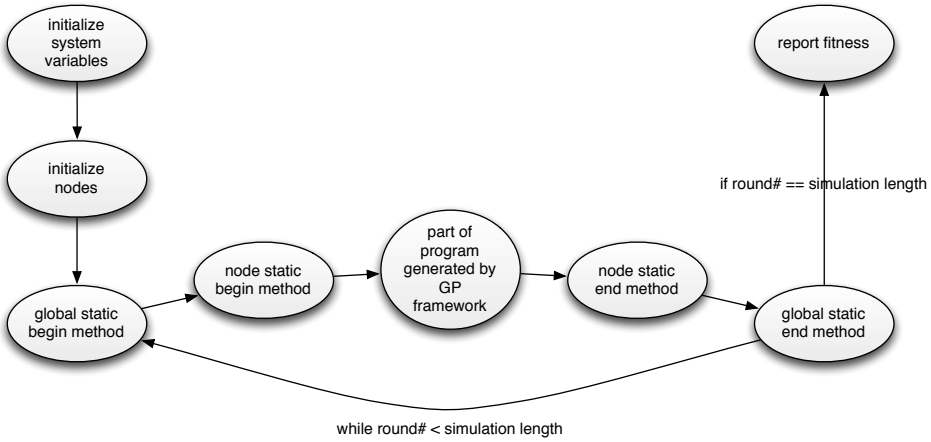

Figure 5.1: The lifecycle of a simulator run

return, but we needed examples. We decided to start with algorithms we already know, and try to re-discover them. Since we already know these algorithms, we have a good idea of how to convert them into configuration files and libraries for our software. This conversion mainly consists of two steps:

- Breaking the algorithm down into logical building blocks that can be re-combined to reform the initial structure;
- Thinking of a fitness function that describes what the algorithm actually tries to do.

Additionally, programming these examples gave us useful feedback on additional features we needed in the framework in order to use it for other examples. Especially the first examples we wrote gave a lot of inspiration for must-have or nice-to-have features. Since new changes were not necessary, we are confident that our software supports the majority of scenarios it was designed for.

The following sections mention example algorithms that we have tested. Each section explains what the algorithm is supposed to do, and what building blocks we used to compose it.

Firefly

The Firefly algorithm[52], aims to synchronize distributed systems much in the same way as real fireflies synchronize the illumination of their backsides. Each node has an *on* and *off* state, depending on the value of its wrap-around clock. Nodes are in the *on*-state in the first N ticks of its period, whereas the *off*-state takes up the rest of the time until the timer expires. At any point in time, each node monitors how many of its neighbors are in the *on*-state. If this number reaches a certain threshold, and the node is not in the *on*-state itself, it will reset

its clock to be in *on*-state. It is important that every node has the same value in order for them to transition to 0 at the same time, maintaining synchronization.

Once the threshold is reached somewhere in the network, meaning that there were at least a few nodes in the *on*-state, the cluster of synchronized nodes becomes larger, spreading to the unsynchronized parts of the network. There may however be conflicts when reaching other separately synchronized parts of the network, making it hard for the whole network to be synchronized.

Ant foraging

In order to test our framework we have also implemented the ant-foraging problem[42] to see if the framework was able to discover the solution. Ant-foraging tries to simulate the behavior that ants exhibit in nature, spreading pheromone trails for other ants, in order to find the shortest path to a non-depleted food source. In this way, the gathering of food by ants is shorter, and thus the time required to gather all the food located in the field is decreased.

Leader election

Leader election[15] tries to elect one leader in an otherwise uniform ring-shaped network. It works by communicating the ID's of nodes, and picking the node with the largest ID to be a leader.

Failure detection

Failure detection[45] tries to detect the percentage of communication packages that get dropped during transmission between nodes.

Churn estimation

Churn is a term that describes the flow of incoming and outgoing nodes in a network. It is an important property for dynamic networks such as peer-to-peer networks and large-scale systems, because it shows that nodes, and the information they keep, do not remain in a network forever, and fresh nodes will also enter the network. Churn detection[44] tries to measure the amount of churn in a network.

5.4 Notes on realism of the simulations

Synchronicity

The first note that should be made on the realism of simulations is a simulator specific one. In a real-world case, nodes will very likely be in different parts of the round than their neighbors due to differences in startup time, clock drifts and execution paths (see next item). In other words, the nodes are all unsynchronized.

Thinking of them as being synchronized is an acceptable idealization for some application domains.

However, in NetLogo, the `ask` command, which is used to control execution for a set of nodes, ís synchronized; It goes from over all nodes in predefined order (ordered by ID), and executes the code there sequentially. Calling this synchronized is debatable, but at least at any point in time, most of the nodes are in idle state, between the end of the last round, and the beginning of the next. Only one node is really executing its code. Of course, in reality, all nodes work concurrently, and thus this simplified execution model's realism is questionable.

NetLogo also offers an alternative to the `ask` statement: the `ask-concurrent` statement. In somewhat similar fashion, code is executed sequentially from, going from node to node, but the turn will go over whenever an node changes the state of the system (for example: changing a global variable). Because the nodes in our framework are not allowed to change global variables, this will not happen that often. Another way in which a node can change the state of the global system is by moving, which is possible in our framework, but overall using the `ask-concurrent` statement is not likely to give very different results than the `ask` statement.

Execution time depends on execution path

Even if we assume that systems are using the same hardware, and that they were at one point synchronized, there is another reason why the assumption of synchronicity is not realistic. Generated programs often contain conditional logic to distinguish one node from another, execution paths among nodes are different, and thus execution times also differ. If we assume a round-based paradigm in which all nodes wait for the others to enter the next round, this is valid. Otherwise, this would lead to some nodes being slower than others, in the sense that they run less iterations than others, only adding to the asynchronous nature of the program execution. Hence, NetLogo executing the code for every system before moving on the the next round is an assumption that can be true for some systems, but may not be applicable to others.

Round length

In the programs that are generated by the MetaCompiler, program code for each system is supposed to be executed repeatedly for a large number of rounds. For some programs it is important that the fitness quickly converges, or in other words, that the aggregate behavior of all programs quickly exhibits the desired behavior. Another requirement of the MetaCompiler is that the simulations take only a certain amount of rounds to run. For these requirements to be fulfilled, we have introduced the property of simulation length, which indicates how many cycles a program should run before returning the fitness value.

Because the grammar uses the depth of the parse tree as a maximum length

indicator for a program, the program is free to grow within the boundaries of this limitation. For some programs, some actions, especially "expensive actions" might be implicitly related to the simulation length.

For instance, we came across a problem when letting the framework invent an ant trail and ant foraging algorithm. In both examples there are a number of mobile nodes that need gather food, that was spread over the virtual field at the start of the program. A designer setting a simulation length of 200 rounds for this program might expect that these nodes are not able to move more than 200 times because there are no more than 200 rounds. This is however not true, and if moving is an action related to the success of the algorithm, the chances nodes will move for over 200 times are very big.

The fitness of these generated programs was related to the number of pieces of food the nodes had gathered when the number of rounds had reached the simulation length and the simulation stopped. In other words, the better the mobile nodes were able to gather the food, the better the score of the generated program was. In turn, the simulation length was related to the initial number of pieces of food located on the field, in order to stimulate the creation of an efficient in terms of food collection speed. However, what the MetaCompiler came up with was an algorithm that would let nodes move a couple of times, every round. The effect over multiple cycles was that each node looked much like a Tasmanian Devil cartoon, running around randomly, accidentally gathering up all the food. For the MetaCompiler, this meant only one thing: mission accomplished, but of course this can not be the case because this behavior is unwanted.

To avoid this behavior, we extended the code to register the number of moves made by the nodes. This was incorporated into the fitness function, such that for programs in which the number of moves was much higher than the number of collected food pieces, we could give a very bad (high) fitness score in order to effectively disqualify this program from the selection process.

Chapter 6

Implementation

This chapter describes the implementation of the MetaCompiler. It describes the actual design choices and the motivation behind them. In Section 6.1, we look at the full MetaCompiler, what it contains and how it is structured. In the Section 6.2, we look at EpochX, a genetic programming framework, the main building block of the MetaCompiler, and why we favored this framework over its competitors. NetLogo is an agent based simulator, very popular in the academic world. In Section 6.3, we look at how it is loosely integrated into our framework. For a multitude of reasons, we have chosen to build a small language targeted towards Agent Based Systems. In Section 6.4 we describe it and provide our motivation, as well as usage experiences. One of the improvements we have implemented, as mentioned in Section 4.2, are tree-based operators for Grammatical Evolution. In Section 6.5, we will look at their implementation details. In Section 6.6 we will take a look at the implementation details of the Distributed Dispatcher.

6.1 MetaCompiler System Architecture

The software toolchain we have built is fairly complex in that it expands the used EpochX (Section 6.2) library in a number of ways. To describe the system architecture we first enumerate what functionality we have built, followed by how this is integrated into the system, to look at the separate parts of the system in the following sections.

We have implemented the following functionality:

- A BNF grammar containing our agent specific language to use as an input grammar for Grammatical Evolution

- A parser grammar to parse the generated programs to a generic AST
- A tree parser grammar to translate the AST to the target language (we have built one for NetLogo, but there can be other ones for other target languages)
- A Grammatical Evolution to Tree transformer that builds up a tree out of the EpochX Grammatical Evolution Integer sequence that encodes a program
- An implementation of LHS crossover operator for the tree representation mentioned above
- A pattern detection and protection mechanism, that works on the tree representation
- An implementation of an LHS like mutation operator, that works on the tree representation
- An implementation of random search, also using the tree implementation
- A dispatcher interface, to offload simulation of programs to different computational units, of which we have written a CPU implementation, and a distributed dispatcher, that dispatches tasks to a pool of other machines. A GPU implementation is in the making by another student.
- The full MetaCompiler, which integrates all components into a single package, and uses a configuration file as input

6.2 EpochX

Because the main target of our project is to use Genetic Programming to discover algorithms, it makes sense to look at what other people have done in this area, and see whether we can make use of tools that are already available. Since Genetic Programming has been around for a while, there have been many projects that have programmed it for a specific language or scenario. Some prominent frameworks in the field are ECJ[26], JGAP[29], Watchmaker[12], and EpochX[41], but there are many others (See this page[1] for an overview).

ECJ

ECJ is possibly the most well-known Evolutionary Computation framework, written in Java. It supports a lot of different variations of EC, making it interesting

[1] Overview of GP frameworks: `http://www.tc33.org/genetic-programming/genetic-programming-software-comparison/`

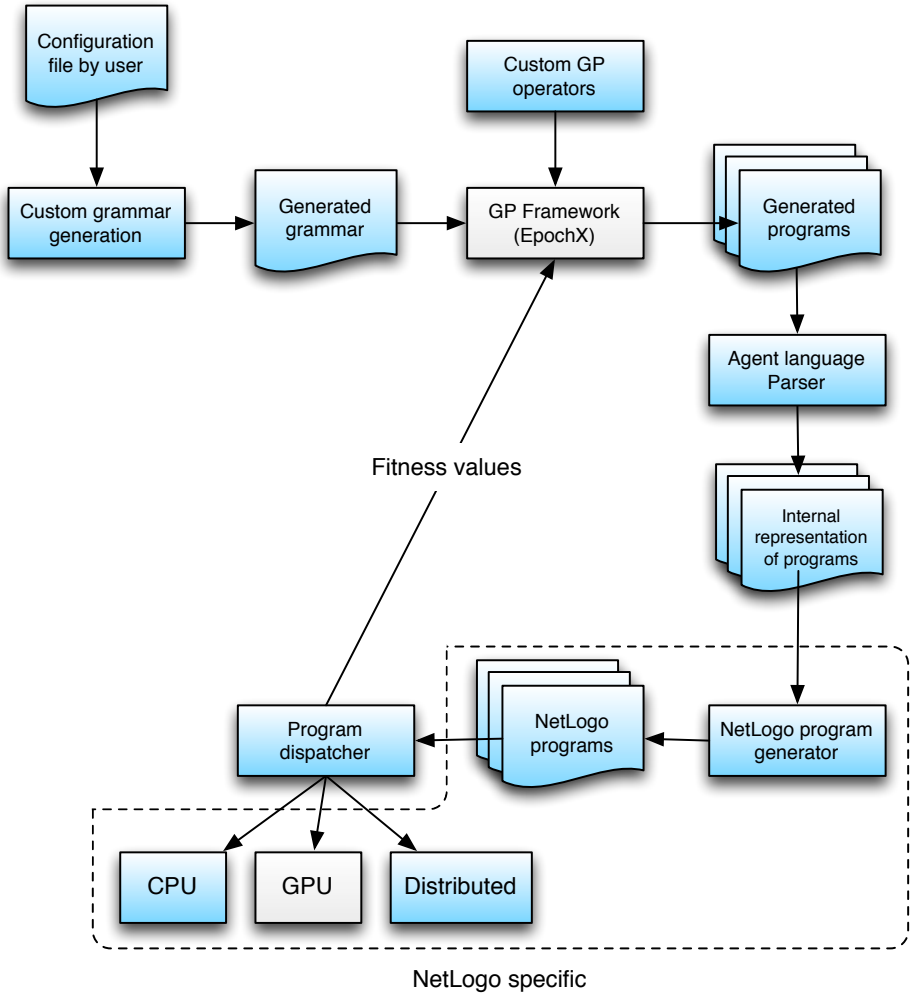

Figure 6.1: The MetaCompiler. The blue portions denote my contributions.

when you want to try a lot of different techniques. We were however mostly interested in Genetic Programming, and therefore we preferred a framework with a little more focus. Also, having looked at the code, the Genetic Programming process was not that comprehensible because of the size of the framework. ECJ offers a permissive Academic Free License, which is nice because it even allows our software to be made proprietary, if that would be a future direction of our project.

JGAP

JGAP mixes Genetic Programming and Genetic Algorithms into one software package written in Java. Unfortunately it offers only one variation of GP: Tree GP, which requires that all code to be used in generating programs is written in Java. Furthermore it is highly integrated with the graphical user interface, the code for GP is often mixed with code for the GUI, lacking a neat distinction. The code is very clean and simple, so it can be a good inspiration for writing a new framework, but it was not what we were looking for because it lacked good extension possibilities. Also, because it uses a Lesser GPL license, source code modifications are supposed to be communicated back to the original author.

Watchmaker

Watchmaker is an Evolutionary Computation framework written in Java. Much like ECJ, it supports a lot of different Evolutionary Computation approaches. Downsides of this framework are that due to the freedom in EC approach there is a lot of code to implement by the user before he can use GP. Also, the last updates of the code happened 2 years ago. Watchmaker has a permissive Apache license, which would allow our project to go in all directions.

EpochX

The framework we finally picked is called EpochX. It is not the most well known framework in the field but it suited our project very well. The framework provides a number of different GP implementations, which allowed us to play with different approaches of code generation and pick the solution we preferred (GE, see Chapter 3.3.1). Also EpochX is built only for Genetic Programming, leading to a lean set of classes only supporting what we needed. The code is really clean and built around the logical steps of the GP process, making it very intuitive to look through.

The best feature of the framework is that it is built to extend in 3 major ways. The code is full of places the system designer can add hooks at important moments of the GP process, to intervene with it, for example by changing the population just before the selection process. It also offers adding monitor functions, mostly at the same parts as the hooks, for example to get the result of the generation that just finished. There is a rich structure of extensible and implementable

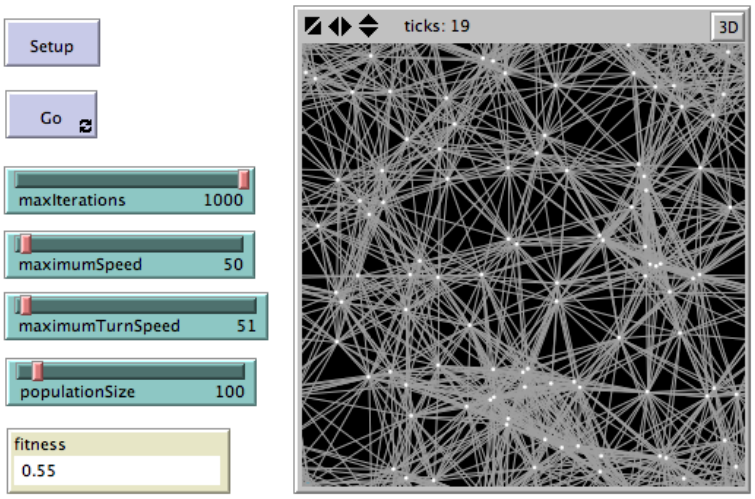

Figure 6.2: Experiment running on NetLogo

Java classes, but there is always a default behavior, such that the system designer can always get the behavior he wants, but usually does not have to.

In these ways the framework is highly incorporable in other frameworks without ever having to touch the source of EpochX itself (unless you find a bug in EpochX). We also liked the fact that everything happens in runtime and without Java reflection, so there are no strange config files we have to include or error-prone class loading errors.

Unfortunately the support was not as good as we had thought, there is a forum but it takes a long time to get an answer. Also, we have uncovered 1 important bug in the framework, and it is acknowledged by the author of EpochX, but still there has been no release with the bug fixed. There we did need to adjust the source code ourselves.

It has a Lesser GPL license, but because there is usually no need to adjust the source code yourself this is in practice no different from permissive Apache or BSD licenses.

6.3 NetLogo

NetLogo[50] is a programming and simulation environment for multi-agent systems. It it especially targeted towards education and according to its website it is used by tens of thousands of students, teachers, and researchers worldwide. Because NetLogo is targeted towards eduction, it is very straightforward to write code and model systems in it, and understand code from other authors. Because of these properties, the Embedded software department has been using NetLogo for the last couple of years to test distributed algorithms.

For the Genetic Programming process to work, there has to be an element that

is able to compute the fitness of a generated program. The MetaCompiler translates a generated program to NetLogo, runs the code, and evaluate its fitness. A nice advantage of NetLogo is that it offers a graphical user interface (Figure 6.2) for each of the generated programs, helping to envision and debug what a generated program does.

The MetaCompiler is designed to be agnostic towards the output language and the corresponding simulator. As noted in Section 5.4, the NetLogo simulator is in some cases not a realistic simulator. Having another simulator that takes care of these concerns delivers more reliable and realistic fitness values, and possibly algorithms. Also, it should be noted that NetLogo is not optimized for speed, so running a lot of generated programs could probably be much faster in another simulator. However, since we have used NetLogo from the beginning of the project, we have developed a number of tools that are NetLogo specific (see Figure 6.1), making the its use very convenient. Redeveloping these tools for another language and simulator will take some development effort.

6.4 A custom agent based language

In Chapter 5, we have set a number of properties that define the Multi-Agent Systems models we make use of. It is important for Grammatical Evolution to produce a program in a certain target language, but we want to design our system to be able to generate programs for various platforms. In order to achieve this, we decided to design a custom internal language to tightly fit our needs. This section will only cover the implementation of the language. To read more about the motives behind the custom language, please refer to Chapter 5.

The language implementation consists of 3 separate parts:

1. A BNF grammar to be used for program generation in EpochX
2. An LL(*) grammar to parse the programs generated by EpochX to an Abstract Syntax Tree
3. A tree parser grammar that converts the generated AST to an output program

Looking at this from a language design perspective, one might think you could either implement parts 1, 2, and 3 into a single part, or more elegantly (and more common in language design) combine parts 1, and 2, and leave 3 as a separate module. However, there are a few reasons we have picked this design.

First, a very pragmatical issue, EpochX only understands BNF grammars. BNF stands for Backus-Naur Form[55], that is a very commonly used method to encode grammars. There is one main advantage of BNF to other representations, and that is that it is simplicity. This makes the grammars expressed in BNF very simple to understand for humans and straightforward to parse by computers. Unfortunately, BNF grammars are not the most natural way to describe context-free

languages, because it does not support optional statements, and repeating statements. Two reasons for it being used in EpochX are possibly the fact that the parsing is easy, and that it is expressive enough for simple examples.

For parsing the programs output by EpochX, we wanted to use an industry standard parser generator, preferably in Java. One very popular parser-generator for Java is ANTLR. Additionally, we had previous experience with this software, it seemed convenient to use it for our project. ANTLR uses a grammar system called EBNF, which extends BNF with regular expression like constructs that allow users to use optional and repeating statements to more elegantly design a language.

Because EBNF is an extension of BNF, the developer is ultimately allowed to use just BNF, but it is unlikely to do so. Additionally, all statements that can be expressed in EBNF can be rewritten to BNF. These two facts were nice to know in advance when developing a language with 2 separate tools.

An unforeseen positive consequence of separating the generative grammar and the parsing grammar was that we were able to play around with the generative grammar to change the odds of a production being picked, for example by using redundant productions, or reducing and increasing the number of non-terminals that need to get picked before we get to a terminal. Because the parsing grammar is separate, that grammar still contains the clean structural description of all possible agent programs, while the generative grammar is tweaked to optimize the performance of the Genetic Programming process.

After the program is generated by EpochX, it is parsed by the parser grammar, giving us an abstract syntax tree. This AST is then parsed by the tree grammar in order to generate the output program. Because we want to run the generated code in NetLogo for simulation, we have made a tree grammar to generate NetLogo, but because the abstract syntax tree is very simplistic, adding a new tree grammar for another output language is an easy thing to do.

6.5 Tree based operators for Grammatical Evolution

To implement the LHS operator and adjoining mutation operator as described in 4.3, we have converted the default Grammatical Evolution integer sequence representation into a tree representation. To reiterate, the default encoding of a program consists of an sequence of integers that is used to pick productions in a grammar until either the integer sequence runs out, or there are no non-terminals in the grammar to fulfill. There are two methods to do the traversal, breadth first and depth first, the latter one being the default mode for EpochX.

Since we want the operators to seamlessly integrate with the EpochX Grammatical Evolution implementation we decided to make it resemble the depth first search traversal of the integer sequence, such that all operations in EpochX agree on the cohesion between the encoding. Unfortunately, we were unable to use the depth first implementation of EpochX, so we had to meticulously resemble its

implementation to match its behavior, which might give some problems in future upgrades of the EpochX framework. Fortunately, we were able to fully use the implementation of the grammar interpretation provided by EpochX.

EpochX distinguishes between 4 types of relevant Java classes in the interpretation of the grammar:

- `GrammarProduction` a class that encodes the left hand side of a grammar rule, possibly containing multiple `GrammarNodes`.
- `GrammarRule` This class encodes a non-terminal rule at the right-hand side of the grammar, depth first traversal will first expand this rule before continuing to the next part of the current grammar production
- `GrammarNode` A superclass of `GrammarRule`, and `GrammarLiteral`
- `GrammarLiteral` an actual terminal symbol, a "dead end" for depth first traversal

To encode a tree that is based on the depth first traversal and the grammar, we have added two Java classes:

- `ProgramNode` encodes a single node in the the `ProgramTree`. Every instance has a reference to a `parent` node, which in the case of the root is `null`. Every `ProgramNode` is instantiated with a `GrammarRule`, and an integer called the `codon value`. This value determines the choice made in the `GrammarRule`, giving the instance a `GrammarProduction`. Also, every node has one or more children, if the GrammarProduction contains a non-terminal that needs to be expanded, otherwise the list of children is empty.
- `ProgramTree` is used to encode the tree as a full object to easily couple it to the corresponding program in EpochX. It has a pointer to the `ProgramNode` instance that is the root of the tree, and it has extra methods to retrieve specific nodes, random nodes, or all nodes from the tree, which is useful in the crossover and mutation operators.

Now that we have a way to transform the typical GE encoding, a sequence of integers, into a tree representation, we can do transformations on this tree to do mutation and crossover. For the LHS crossover operator (Section 4.2.2), and the corresponding mutation (Section 4.2.3), we were now able to copy `ProgramTrees` and replace `ProgramNodes` in order to get new programs. After this transformation the `ProgramTree` could return a sequence of integers encoding to feed the generated program back to the GP library again.

For the pattern detection on `ProgramTrees`, we simply request all of the nodes of the best programs (the training set), and compare the nodes of all `ProgramTrees` of the current population to these nodes, and if a similar parent-child pair is found the connection between that pair is strengthened. While this comparison of all

nodes is a bit brute-force, and could be optimized, it did not lead to performance or memory problems.

While searching for similar parent-child pairs, the weight of the most protected connection is registered, such that we can map back all connection weights to a 0-1 interval. If pattern detection is used, this connection weight denotes the odds of a connection being able to be changed, with 0 being always able to be changed, and 1 being unable to be changed. We do this by generating a random number. If that number is bigger than the connection weight, the transformation is allowed to happen.

6.6 Distributed Dispatcher

Inspired by so-called bot nets used by hackers to coordinate big computer networks, we decided to make a tool to delegate our experiments to other computers to parallelize the most computation and time intensive part of the MetaCompiler. Because bot nets often use the IRC chat protocol to delegate tasks, we decided to also take a look at using this. IRC has a number of nice features that make it suitable for controlling a network of computers:

- Channels, where all the bots are gathered, so they are easily addressible
- Broadcasting messages over the channel, in order to address all the bots on the channel, for example to signal there are jobs available, or that a new generation has started
- Private messaging, to address a specific bot with a specific task, for example to send a specific experiment, or do a handshake between server and client
- File transfer possibility, to transfer executables, unfortunately we were unable to use this due to firewall issues, so we have built a small fileserver to handle this

To start, we implemented a layer on top of Java IRC bot implementation PircBot[37], in order to transfer object instead of messages. A disadvantage of IRC is the fixed length of messages, which is about 500 bytes. Since serialized objects in Java are usually much bigger than that, sending objects takes a few IRC messages, and overloading the IRC server is a risk that has to be avoided. We then implemented a simple handshake protocol with a couple of encrypted challenges such that only clients with the right key can contact servers with the right key and vice versa. We used an asymmetric key protocol to assure that experiments (from servers), are sure to come from a verified source, and do not contain malicious code. The other way around, the server knows that results for these experiments come from a client that has the right key, but since clients run on external computers, we cannot fully guarantee that the key has not been copied to another machine with a fake client returning bogus results.

Since clients are usually busy executing jobs, we found it makes more sense for the clients to request jobs than for the servers to push jobs, since the servers have no knowledge of how busy clients are. Once the handshake process is done, a client will request a job from the server and will try to execute another job while waiting for the response from the server. In this way, the request process is pipelined with the execution of jobs, and the result in practice is a virtually constant execution of jobs by clients. Also, since jobs for multiple servers are sequentially executed, a request for a job to a certain server even has more time to be responded to, preventing an overflow of messages on the IRC server. Figure 6.3 show the program flow for clients, and Figure 6.4 shows the program flow for servers.

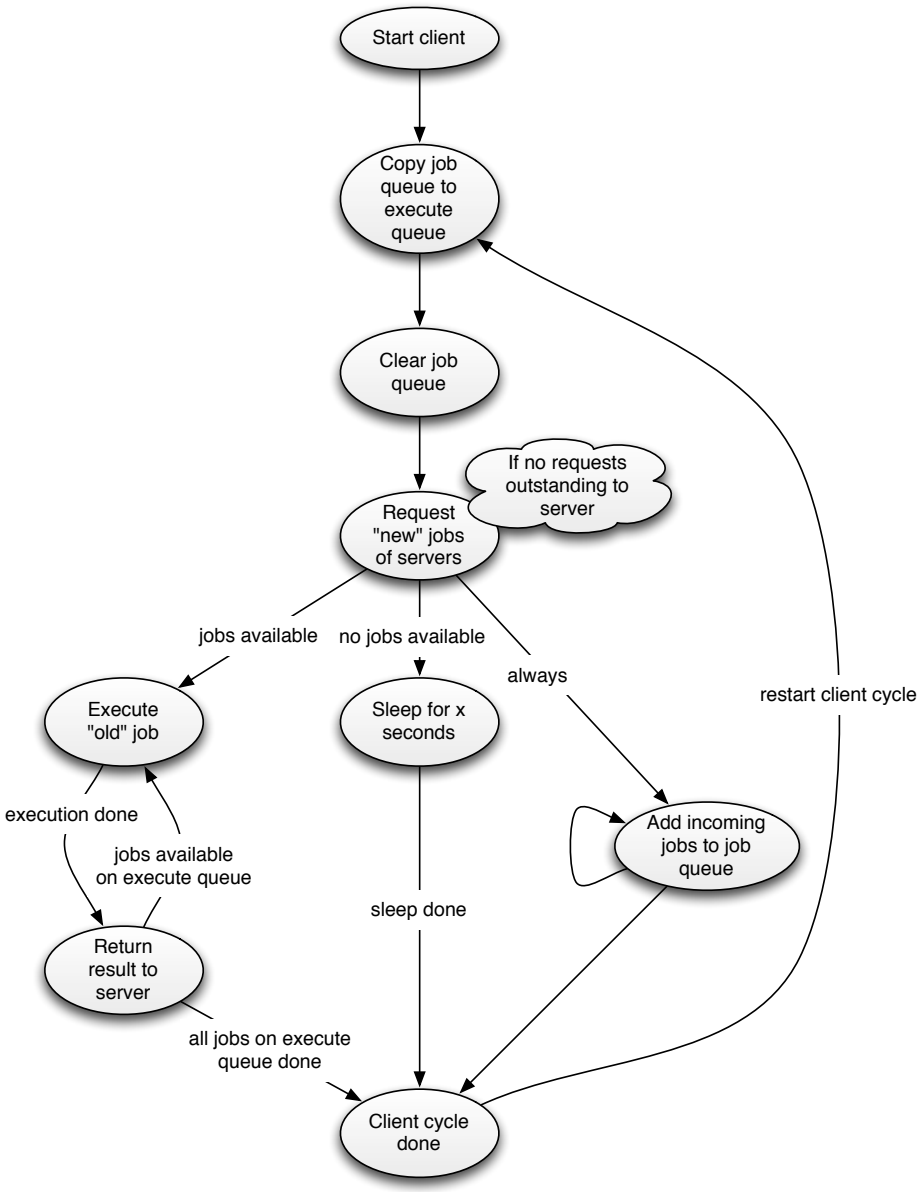

Figure 6.3: Distributed dispatcher: Client side

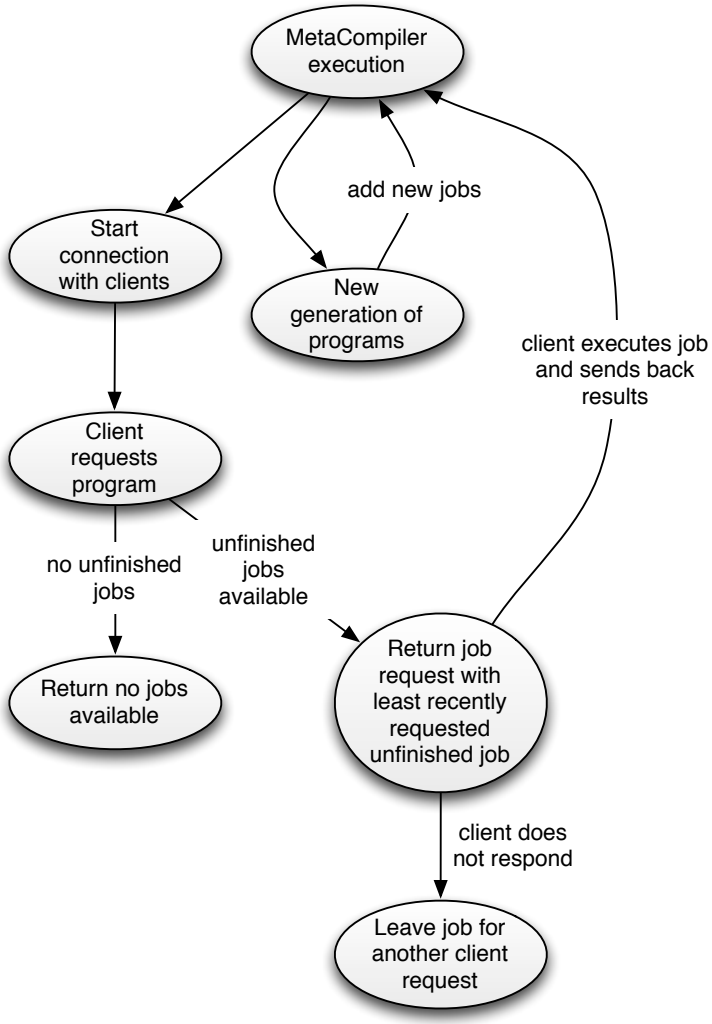

Figure 6.4: Distributed dispatcher: Server side

Chapter 7

Experimental Results

This chapter shows the experiments we have done to assess the performance of the MetaCompiler. Each experiment is introduced with the necessary background information, and some intuitions about the results. In the results for each experiment we highlight the differences and similarities of the results, as well as highlighting other notable properties. Each of the experiments is done 8 times to get a stable measurement value, unless otherwise mentioned. Because testing the combination of all possible settings is impossible, and the influence of individual settings would be hard to see, most experiments refrain to adjusting a single setting and looking at its influence.

7.1 Influence of settings on fitness

7.1.1 Influence of Genetic Operators

In the following experiment, we try to discover the failure rate estimation algorithm for all of the different types of genetic operators we have: random search, old operators (the default Grammatical Evolution operators), tree operators, and tree operators with pattern detection with the 10 best programs as a training set (see Section 4.3). Our expectations are that the random search operator will perform the worst, followed by the simple operators. It must be noted that we do not know this for a fact. Random search starts each generation with a clean sheet of programs, giving it the possibility to land upon a good program many times, while the simple operators (while not very capable) can refine good solutions to come up with better solutions. Obviously, both have some advantage over the other, so it is hard to say which one will be better in the end.

population size	150
number of agents per program	50
simulations per program	8
fitness evaluation	median
number of runs per experiment	8

Table 7.1: experiment settings for the discovery of a failure rate algorithm

population size	50
number of agents per program	20
simulations per program	8
fitness evaluation	median
number of runs per experiment	8

Table 7.2: experiment settings for the discovery of an ant foraging algorithm

For both tree operators, we expect the ones with pattern detection to outperform the operator without pattern detection. We expect this because the pattern detection should maintain viable program structures, and focus the operators to change other parts of the program in order to enhance the working of the good programs. However, the downside of this pattern protection is that good patterns that should be adjusted to become better, are protected too well, and thus it will not evolve as it should.

Experimental Results

This experiment was done with the settings in Table 7.1. After having run the experiments described above (Figure 7.1) we can see a few interesting things. The first one is that the old operators almost outperform the best genetic operators: the tree operator without pattern detection. We find this somewhat surprising because when we analyzed the old operators in Section 3.3.3, we expected it to break the programs so often, that it would not be able to much better than random search. One possible reason that the old operators have a more loosely coupled genotype to phenotype mapping, as explained in Section 4.2.4. With this advantage it might be able to develop the structures needed to solve the problems.

Another interesting finding is that the tree operators with pattern detection are outperformed by tree operators without pattern detection. This could be the effect of the fact that it uses only the 10 best programs of the population as an example for pattern detection. We test this more elaborately in Section 7.1.2.

A positive and expected finding is that the developed genetic operators easily outperform random search. Random search was implemented in the same way as the LHS-like mutation operator works (Section 4.2.3). It starts every time from the root of a new program. In this way it creates random but syntactically correct programs.

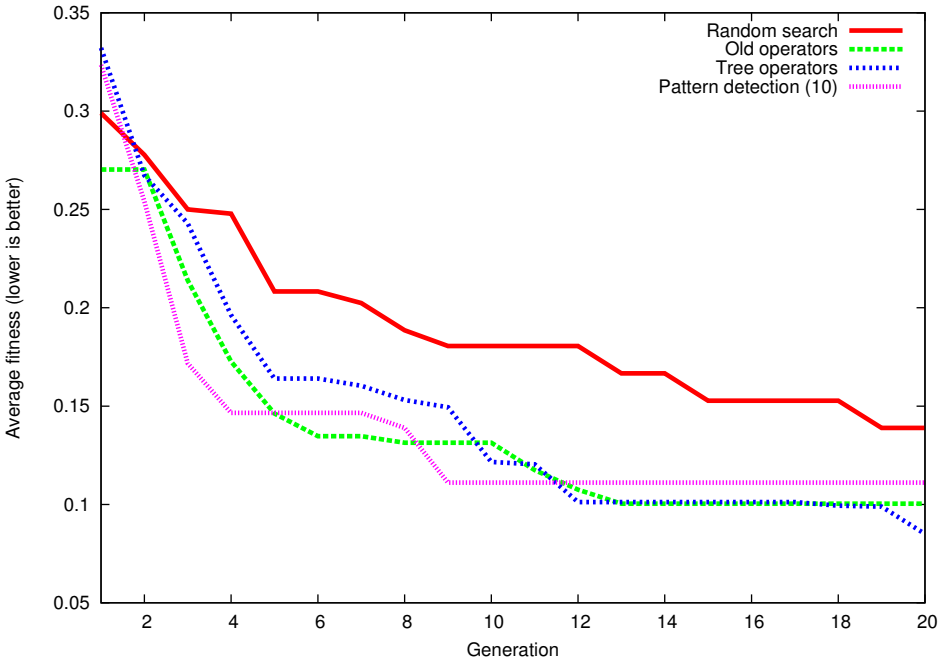

Figure 7.1: Average fitnesses for different genetic operators

An important note on this experiments is the population size. Even though 8 runs for each of the experiments show clear distinctions between the different genetic operators, the results might be different for larger population sizes, when the number of diverse programs is bigger. The recombination and refinement of program parts might become more important and the difference between the old operators and tree operators could change. The same accounts for the different types of programs our framework can handle. In this experiment we have only tested the discovery of the failure detection algorithm, but other algorithms could perform differently with the genetic operators, changing the differences between the different types of genetic operators.

7.1.2 Influence of training set size on pattern detection

To assess what the effect of pattern detection in the tree operator is, we have conducted a test (Table 7.1) that runs the same program with different training set sizes. The level of pattern detection is determined by the number of programs that serve as a training set for what patterns are good. The more programs that are part of the training set, the more gradual the eventual edge weights will be.

Looking at the relationship between the experiment with no pattern detection and the other experiments, it is hard to say on beforehand to which what is mostly equal to no pattern protection. On one end, we have the pattern detection with only the best program in the training set (Pattern Detection 1). If a pattern

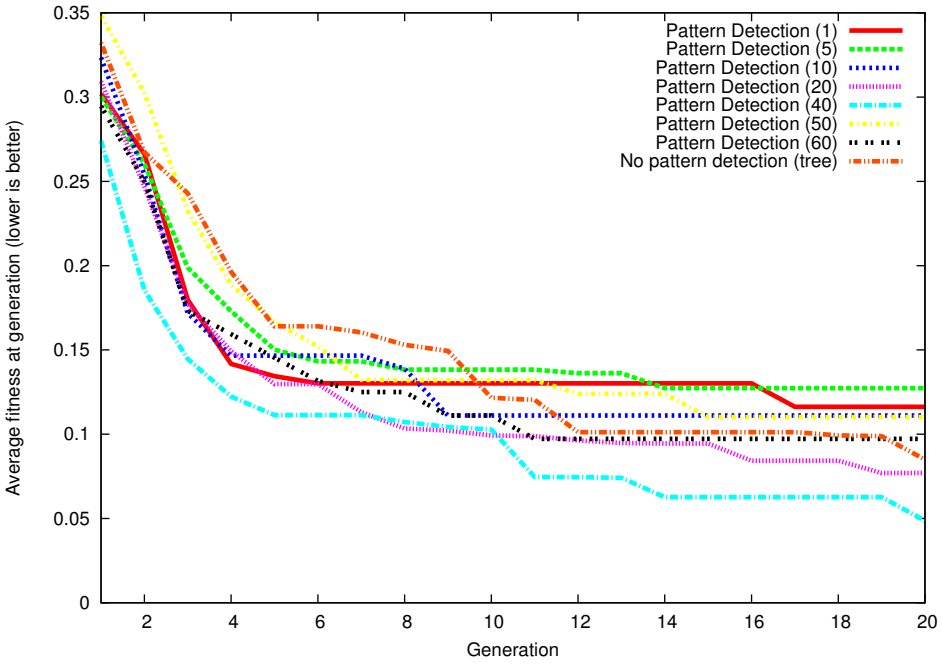

Figure 7.2: Average fitnesses for different training set sizes for pattern detection

exists in the best program, it will be protected against the crossover and mutation processes. If however the pattern did not exist in that particular program, there is no protection against crossover or mutation. On the other end, if all programs in the population would serve as a training set, the patterns that are be protected are the patterns that are most common throughout the whole population. The question is how big a training set should be for achieving a helpful training set size.

Experimental Results

This experiment was done with the settings in Table 7.1. The results of the experiment described above are displayed in Figure 7.2. We can see that only training set sizes 20 and 40 of pattern detection are able to outperform the tree operators without pattern detection. This indicates that a training set size close to 20 or 40 is an optimal setting of pattern detection for this particular experiment.

Also important, there is a trend visible that shows that the tree operators without pattern detection are closest to the tree operators with pattern detection that use a large training set. If the training set is large, there are a lot of different patterns detected in the population of programs. This decreases the strength of detected patterns because the strength is related to the dominance of the pattern in the best programs. If there are many different patterns, the relative occurrence of particular patterns decreases.

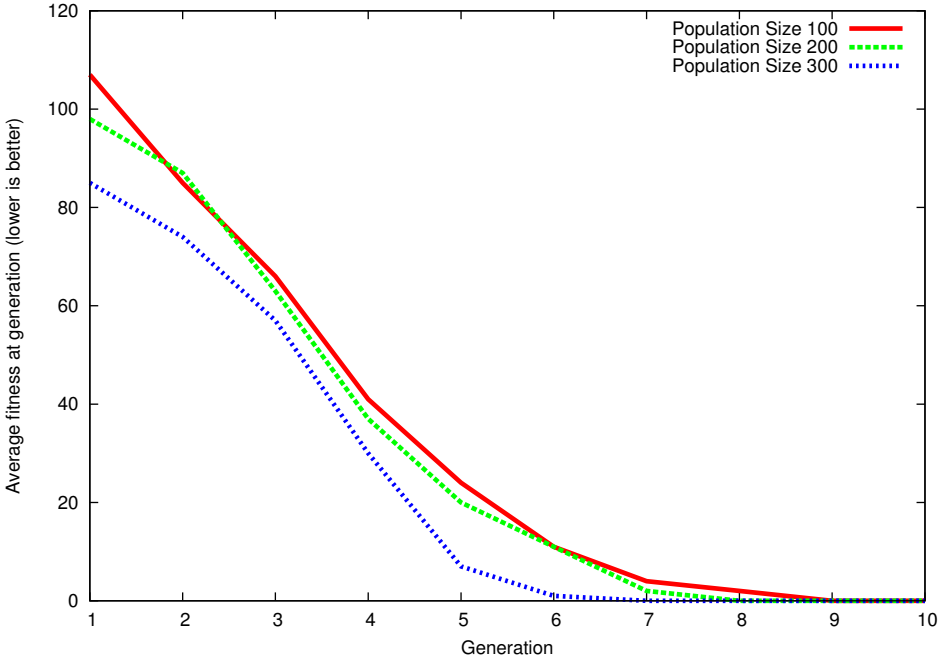

Figure 7.3: Average fitnesses for different population sizes

This experiment confirms the thought that pattern detection and protection can obstruct the search process rather than help it, but is helpful with the right training set size. It should be noted that again, results may differ for different algorithms. If the structure of a solution is important, protecting patterns can become a more important feature, more protection can give a better result than unprotected tree operators. Also, as noted in the previous section, the population size is of importance, because a bigger population may result in more rich structures, on which pattern detection can be of importance.

7.1.3 Influence of Population Size

In this experiment we investigate the influence of the population size on the average fitness through multiple generations. We expect that a bigger population size leads to a faster convergence, because the number of evaluated programs is bigger, the variety in programs is bigger, and there is more space to crossover and mutate programs for the next generation.

Experimental Results

This experiment was done with the settings in Table 7.2. Figure 7.3 shows that the bigger the population size is, the faster the convergence to a good fitness happens. Also notable is that bigger population sizes start with a better fitness.

Because the initialization process generates more programs for bigger population sizes, the odds of finding a good program are already higher when the population size is bigger. Since this "head start" possibly gives an advantage to the rest of the generations, it is hard to conclude that bigger population sizes also have benefits in generations following the initialization, but in general, we can conclude that bigger population sizes are beneficial for the discovery process. In hard problems, bigger population sizes can even make the difference between being able or not being able to discover a solution.

7.1.4 Influence of Minimum Initialization Tree Depth

The initialization process in Genetic Programming should create a set of programs from scratch. This only happens in the first generation. The process has two important parameters, minimum depth and maximum depth of generated programs. The depth is in this case the number of steps that have to be taken to get to a terminal (see Section 3.3.2 for more information). The default initialization process used by Grammatical Evolution is called a Ramped-Half-And-Half Initializer. This initializer is designed to provide an equal amount of programs between the minimum and maximum initialization depth. Since the initialization process is important for the following generations, this can influence the programs discovered throughout the whole run.

In the following experiment, we investigate the influence of the minimum initialization depth of programs on the fitness progress. We expect that if the minimum depth is set too low, the initialization process will deliver programs that are too simple, and the GP process will have problems finding a well suited solution. On the other hand, if the minimum depth is set too high, a lot of programs of which the depth is too low, will be disposed, and the population will lack variation. Consequently the GP process will again have problems finding a suited solution.

Experimental Results

This experiment was done with the settings in Table 7.2. In Figure 7.4, we see the positive effects of increasing the minimum tree depth, because the higher the minimum depth, the faster the GP process converges to finding solutions. We are however still convinced that setting a minimum depth too high is bad for the initial population and the generations that come after it, but apparently a value of 25 did not yet reach that threshold.

7.1.5 Influence of Maximum Initialization Tree Depth

We have also tested the influence of the maximum tree depth setting. Since the initialization process tries to generate an equal amount of programs for depth values between the minimum and maximum initialization depth, having a value

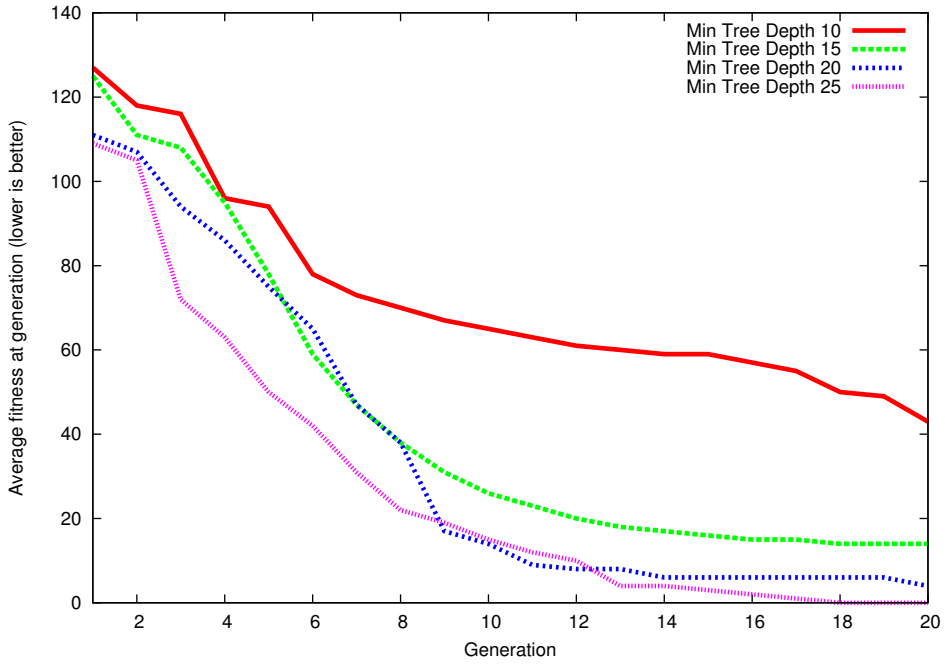

Figure 7.4: Average complexity for different minimum initialization tree depths

too close to the minimum depth will create a population with very similar program depths and structure. This is bad for the variety in the population, and therefore we think that it will result in difficulties for finding a good solution. Setting the value too high will try to force the creation of very "deep" programs. If the initialization process fails to find these deep programs, the first population will contain only a few (or no!) individuals. This will result in a lack of variety in the population and consequently in problems of finding a good solution.

Experimental Results

This experiment was done with the settings in Table 7.2. Figure 7.5 shows results that are different than our intuitions about this experiment. The experimental results for maximum tree depth 40 could show that the initialization succeeded in finding varied populations (the experiments were repeated 8 times), and hence finding good solutions. However, the results for maximum tree depth 60 and higher do not support this statement. The idea that picking a maximum tree depth too small is bad for the population is also not supported by the experiments. Hence there is nothing we can conclude on the influence of tree depths on the fitness, and we should run more experiments to be able to do this.

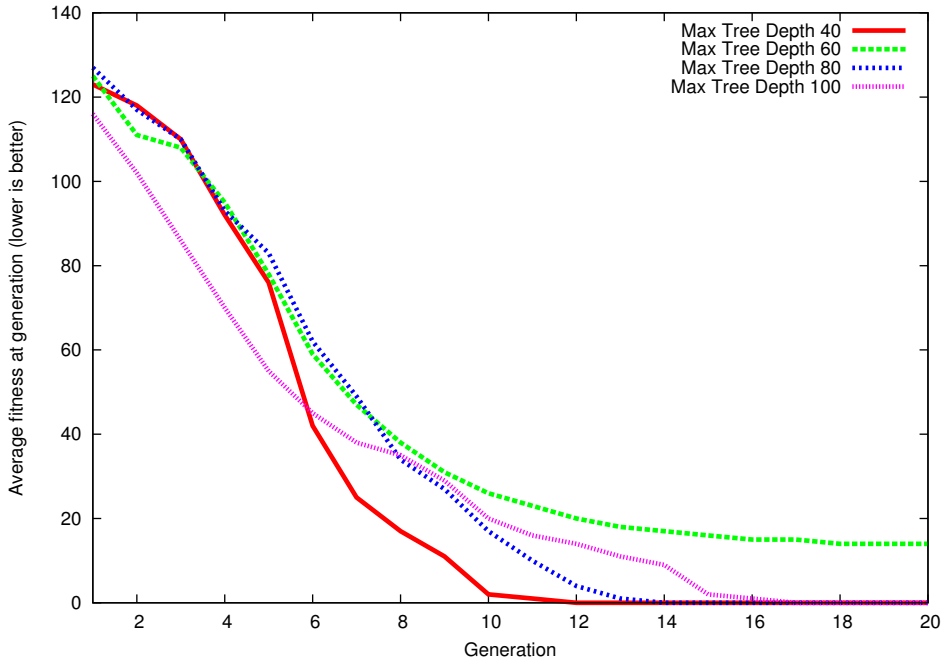

Figure 7.5: Average complexity for different maximum initialization tree depths

7.2 Granularity of building blocks

In this experiment we test the influence of the granularities of building blocks, we have done a test in which we take a set of building blocks (level 1), and test how well they perform in finding the given goal. Then we cut the building blocks into more granular building blocks (level 2) and test how well these perform in finding the goal. Cutting up these building blocks is done by taking a predefined value out of a building block and making it a parameter, such that the GP process has to try to find the right parameter to succeed.

We expect that the difficulty of converging to a good fitness is a lot higher with the level 2 blocks,

Experimental Results

The algorithm we did this experiment with was churn detection with a population size of 200. Unfortunately we were only able to do 4 runs for the level 1 building blocks, and 3 runs for the level 2 building blocks, so the results are not considered stable. However, Figure 7.6 clearly shows that level 2 has more difficulty finding the right goal. In the last generations, we can see that the experiment for level 2 building blocks improves the result, but it is having a very hard time doing this. We will need to work with larger population sizes to get results similar to the results for level 1 building blocks.

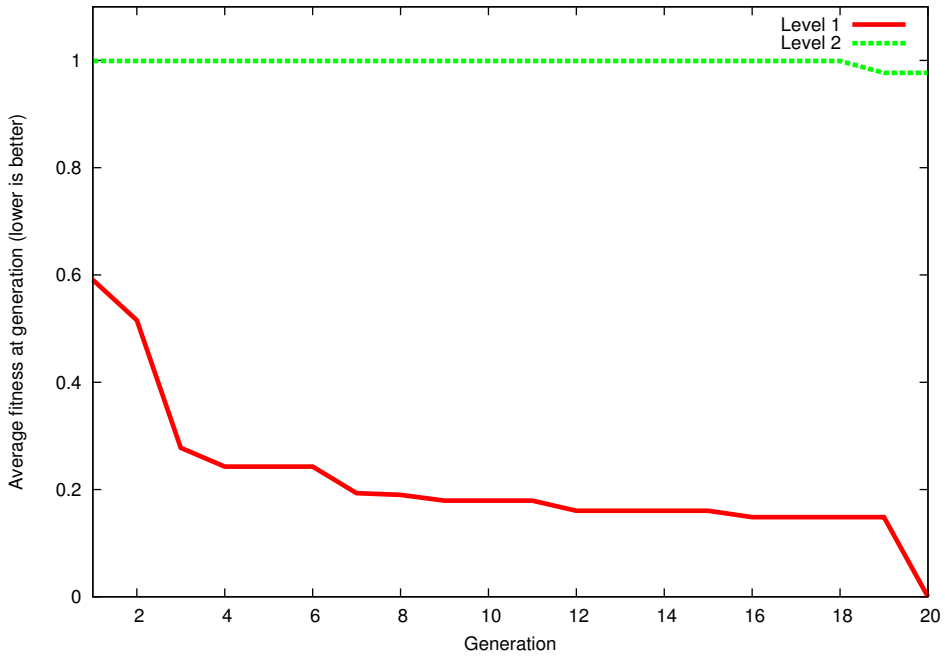

Figure 7.6: Average fitness for different granularities

7.3 Ability to discover new programs

An important property of genetic operators is how well they are able to explore the search space. To assess this property, we have measured how many programs are on average evaluated for fitness in each generation for each type of genetic operator. Since the MetaCompiler caches results for already evaluated programs in one Genetic Programming run, finding a new program in generation N means that it was not found in the $N - 1$ generations before it.

7.3.1 Different genetic operators

For this experiment, we expect random search to find the largest amount of programs on average, because it does not try to modify "legacy" programs that could restrict its search for programs. We expect the old operators to find about the same number of programs as the tree operators, because they both have "legacy" programs that they modify without a bias towards grammar rules. Lastly, we expect the genetic operators with pattern detection to be somewhat restricted in their search process, and thus to find fewer programs because they are biased to modifying only parts of a program, more often resulting in duplicate programs than more unbiased genetic operators.

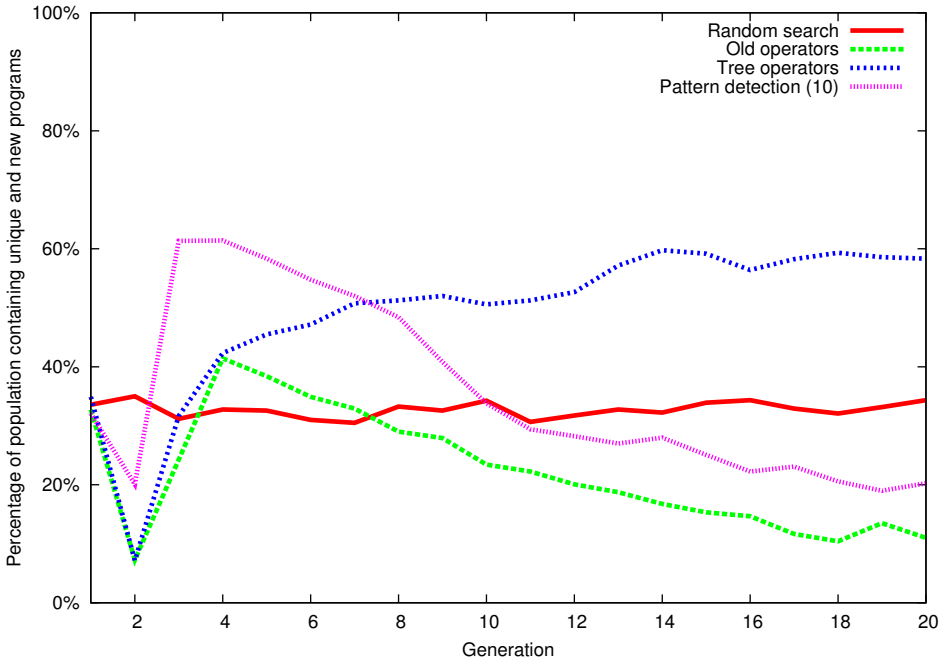

Figure 7.7: New programs discovered in each generation for different genetic operators

Experimental Results

This experiment was done with the settings in Table 7.1. Figure 7.7 shows the results for the different types of genetic operators. The first thing to notice is that except for Random Search, all the other operators seem to find a reduced amount of new programs in the second generation. We think this is due to the initialization phase, that fails to generate a lot of programs on itself (see Section 4.1), and probably mutation and crossover take some time to come up with totally new programs afterwards.

We notice that Random Search has overall the most steady number of new programs, however it is unexpectedly outperformed by the tree operator in the number of new programs that were found. Even though the tree operator only shows a very small victory in Figure 7.1, we find it interesting that the tree operator is able to come up with a lot of programs every generation, because it is good for the diversity of the population, and the exploration of the search space. This is different from what we see from the old operators, that are less successful in coming up with new programs, from which we can conclude that it is bounded too much by the "legacy" programs it has to evolve from.

As expected, the genetic operators with pattern detection are more restricted in exploring the search space.

One pragmatical note we want to make is that since the tree operators come

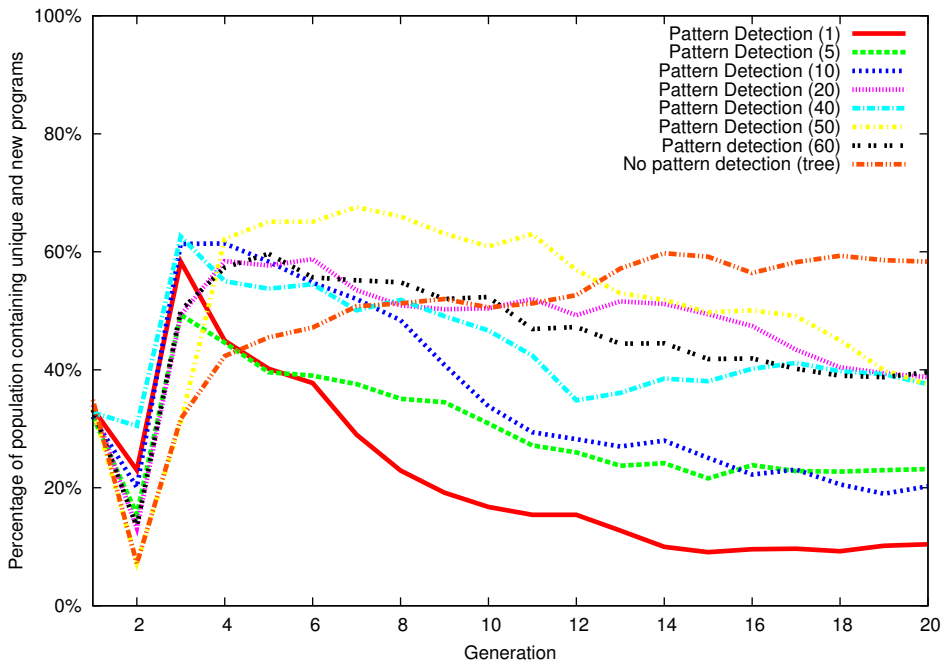

Figure 7.8: New programs discovered in each generation for different training set sizes

up with more programs, runs with the tree operators usually take up a lot more time because many more unique programs have to be evaluated.

7.3.2 Influence of training set size on discovering new programs

We have tested for seven different training set sizes for pattern detection. We expect that the number of discovered programs will become less whenever pattern protection is stronger, since the GP process is not allowed to change every part of the program, restricting the search space.

Experimental Results

This experiment was done with the settings in Table 7.1. In Figure 7.8, we see the results for different training set sizes. We see that having pattern detection based on less programs, making the detected patterns stronger, restricts the searching process. Even though this is effect is not perfectly clear, a correlation is visible.

Another important difference that is apparent from the graph, is that the genetic operators without pattern detection are the only operators that show an upward trend towards discovering new programs. For the other operators, there seems to be a correlation between the strength of the pattern detection and the downward trend in finding new programs.

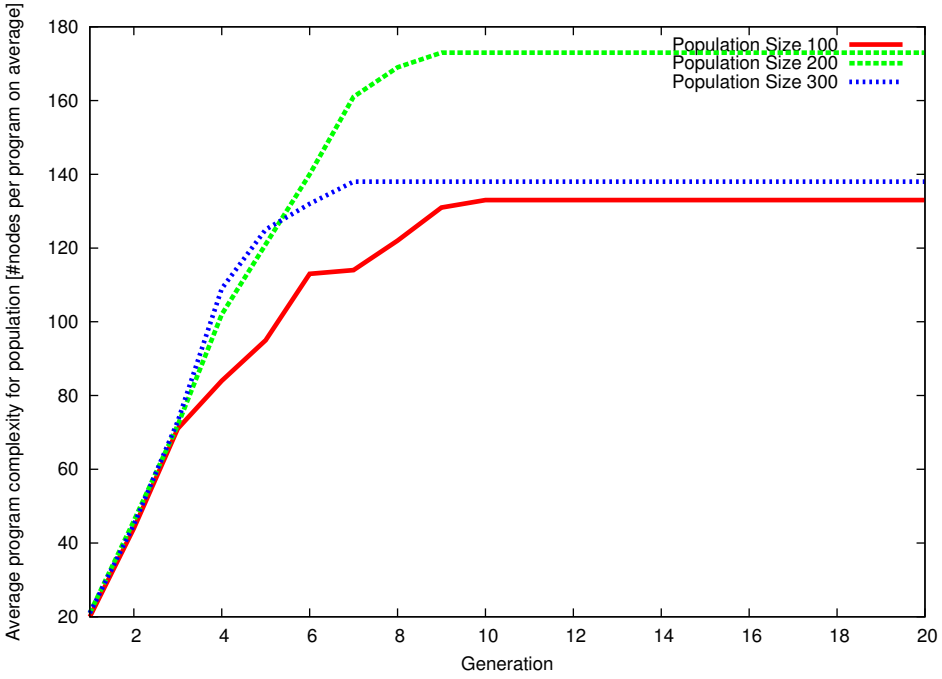

Figure 7.9: Average program complexity for different population sizes

7.4 Program Complexity

One aspect of programs in the Genetic Programming process that usually changes throughout a run is the program complexity. The MetaCompiler measures program complexity by counting the number of nodes that are necessary to encode it into a tree (see Section 6.5). In the following sections, we assess the influence of different settings on the program complexity, and how it evolves over time.

7.4.1 Influence of population size

Since the initialization process does not create very complicated programs, the crossover operator creates more complex programs, unless the very simple programs have an especially good fitness. If the solution the MetaCompiler is searching for is a complex one, having a bigger population size helps increasing average complexity of programs in a population faster. It is unknown whether the population size influences the complexity if the solution does not require a very complex solution, possibly it will converge to the required complexity faster.

Experimental Results

This experiment was done with the settings in Table 7.2. In Figure 7.9, we see the influence of the population size on the program complexity for the Ant Foraging problem (Table 7.2). There is no visible relationship between the population size

and the complexity of the programs. Instead, each of different experiments for different population sizes stabilizes at another program complexity, not related to the population size (100-300-200 in order of increasing complexity).

7.4.2 Influence of initialization tree depth on complexity

In this experiment we vary the minimum initialization tree depth to see how that influences program discovery. Our expectation is that using a minimum depth that is too low will deliver too many simple programs, and hence the discovery process in the following generations will have problems discovering more complex programs. On the other hand, when having a minimum depth that is too large, the variety of programs will suffer, because the initialization process only accepts programs with a depth that may be difficult to reach in the grammar.

Next, we vary the maximum initialization depth, that provides an upper bound for the depth of programs provided by the initialization process. As mentioned above (Section 7.1.4), the initialization process tries to provide an equal amount of programs for all depths between the minimum and maximum initialization tree depth. Setting the maximum depth too close to the minimum depth will result in very similar programs, resulting in a population that lacks diversity and consequently exploratory power. Hence it is recommended to leave enough space between the minimum and maximum depth. Setting the maximum depth too large however might give problems for the initializer to come up with programs that are in the larger spectrum of tree depths. When this fails, the default implementation of the GP library is to return no program for that depth, resulting in a population that is only partially filled with programs, because it did not add programs for depths that were too large. This results in a population that lacks diversity, so it is important to pick the right maximum depth.

Experimental Results

This experiment was done with the settings in Table 7.2. In Figure 7.10 we see a clear relation between the minimum tree depth and the average program complexity. Hence, if it is expected that a fitness function set by the user can only be acquired by complex behavior, it is important to pay attention to the minimum initialization depth. It must be noted that the evolution of program complexity throughout the generations is also very dependent on the fitness function The influence of the minimum initialization depth on the program complexity is unmistakable.

In Figure 7.11 we see the influence of the maximum tree depth setting. Even though it takes a number of generations to display, there is a clear inverse relationship visible between the maximum depth and program complexity. A possible explanation for this is that the initialization process had difficulties finding programs with depth > 40, or maybe already for depth $= 40$. Hence, the higher the value of maximum depth, the less programs were discovered by the initializa-

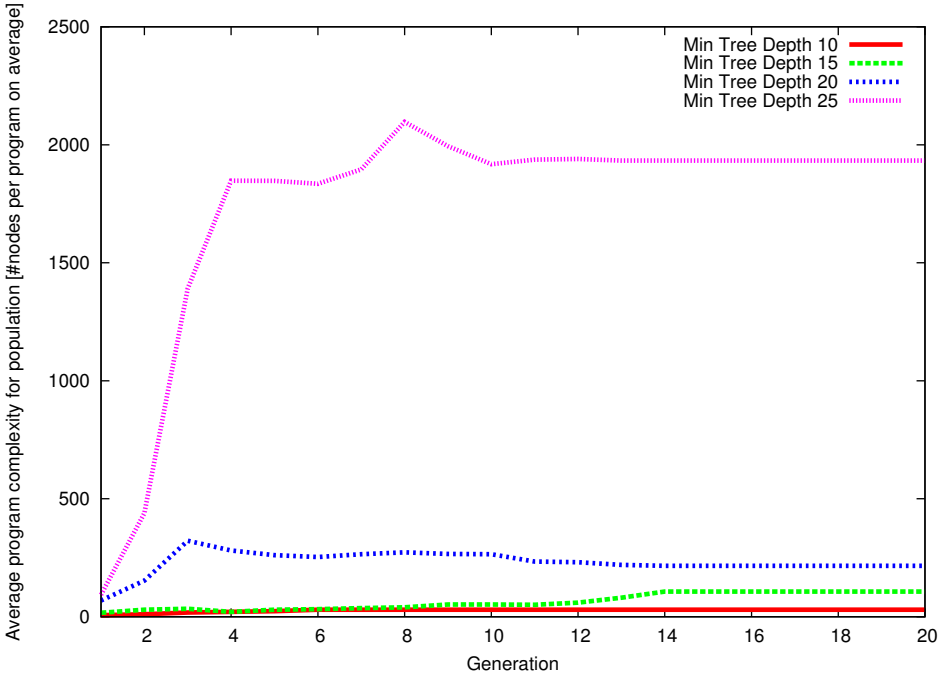

Figure 7.10: Average program complexity for different minimum initialization tree depths

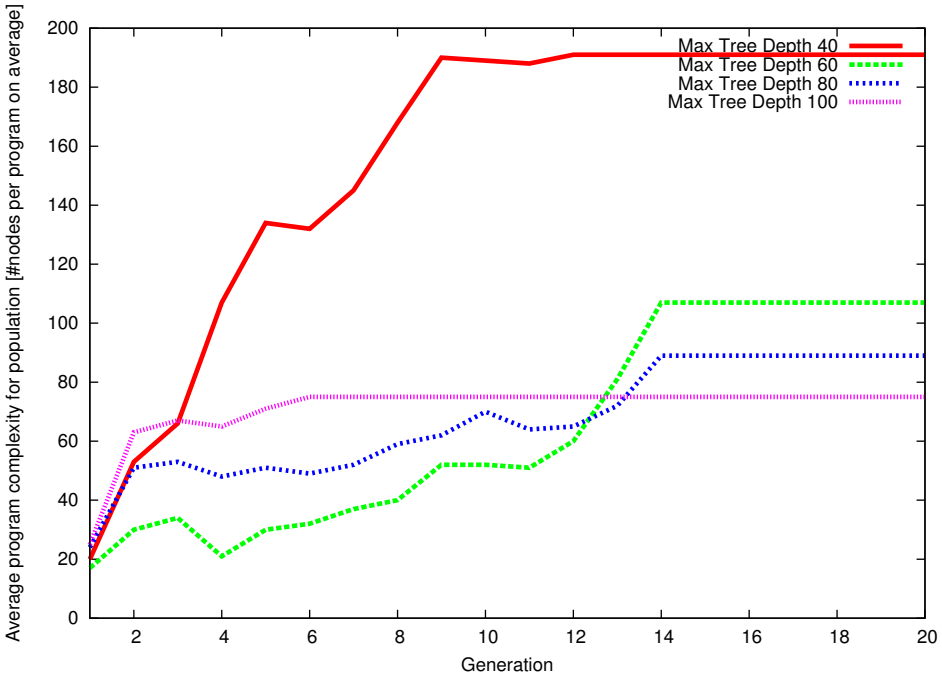

Figure 7.11: Average program complexity for different minimum initialization tree depths

tion process, which has consequences for the discovery of programs in following generations.

7.5 Execution Time

An important aspect of every Genetic Programming process is the execution time of an experiment. Since every experiment has multiple runs $\times$ multiple generations $\times$ multiple programs to evaluate $\times$ number of evaluations per program, the execution time of an experiment quickly grows. Clearly, the time of evaluating a program is the most dominant here, so it is interesting to look at the cost of evaluating one single program.

7.5.1 CPU Dispatcher

The CPU dispatcher is the most straightforward method to evaluate programs with the MetaCompiler. For each separate program, it uses a so-called NetLogo headless workspace (i.e. a version of NetLogo without the overhead of a graphical user interface), to run the code. Since it is impossible to run multiple programs in one headless workspace, a new one has to be instantiated for every program execution, even if one program is executed multiple times in order to get a stable measurement. We suspect there is a lot of overhead in this instantiation. Also, the static methods that are imposed by the user are part of the overhead execution time, because they are called regardless what the Genetic Programming process has generated. See Section 5.2 for the structure of a program generated by the MetaCompiler.

The following data was extracted from the experiment investigating the influence of initialization tree depths in Ant Foraging (Section 7.4.2 and Table 7.2). We compare the execution times of programs of different complexities in order to see the impact of program complexity and workspace instantiation overhead on the execution time. To get a fair measurement, the execution times of this experiment were measured on the same machine (2GHz Intel Core i7), that was not running other heavy tasks at the moment of execution. The values were calculated by taking the total execution time of one generation divided by the number of program instances ran in that generation, to spread the overhead of the Genetic Programming process over multiple instances to make it insignificant. Because both the execution time and the complexity are not reproducible, the values are the values were not measured multiple times as in other experiments.

Experimental Results

In Figure 7.12 we see the overhead imposed by instantiating the headless NetLogo workspace. We see that the minimal cost to run a program with low complexity is about 350ms. If we consider an experiment with 8 runs $\times$ 20 generations $\times$ 200 programs to evaluate per generation on average $\times$ 8 evaluations per program $\times$

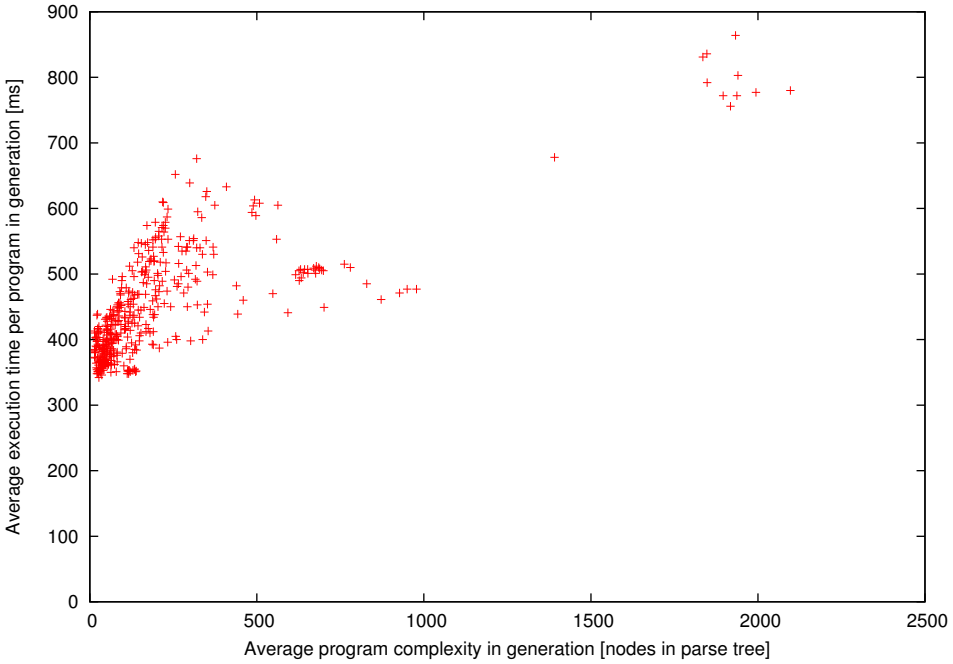

Figure 7.12: Relation between program complexity and execution time

350ms workspace instantiation, this overhead execution time already amounts to more than 180 minutes for that experiment.

The relationship between execution time and program complexity is also clearly visible. For the most complex programs the execution time doubles. Since other experiments in this chapter have already shown the complexity of programs can be influenced by other settings, the execution time is also indirectly influenced by other settings.

It must be noted that the discovery for Ant Foraging is a relatively lightweight program to execute, it has no time consuming static methods, and the methods that can be used by the MetaCompiler to generate a program are also not heavily time consuming. Other examples like Failure Detection and Churn Detection take considerably more time to run, especially due to time consuming static methods. In these cases, the relative amount of overhead will be even bigger than for Ant Foraging, having a vast influence of the execution time of an experiment.

7.5.2 Distributed Dispatcher

In this experiment we assess the runtime of the distributed dispatcher as opposed to the cpu dispatcher. The decreased runtime is only one of the advantages of the advantages of the distributed dispatcher, but it would be nice to see it at work. Because at the time of the experiment there were only 9 clients connected, and there is overhead in the communication process of servers and clients, we do

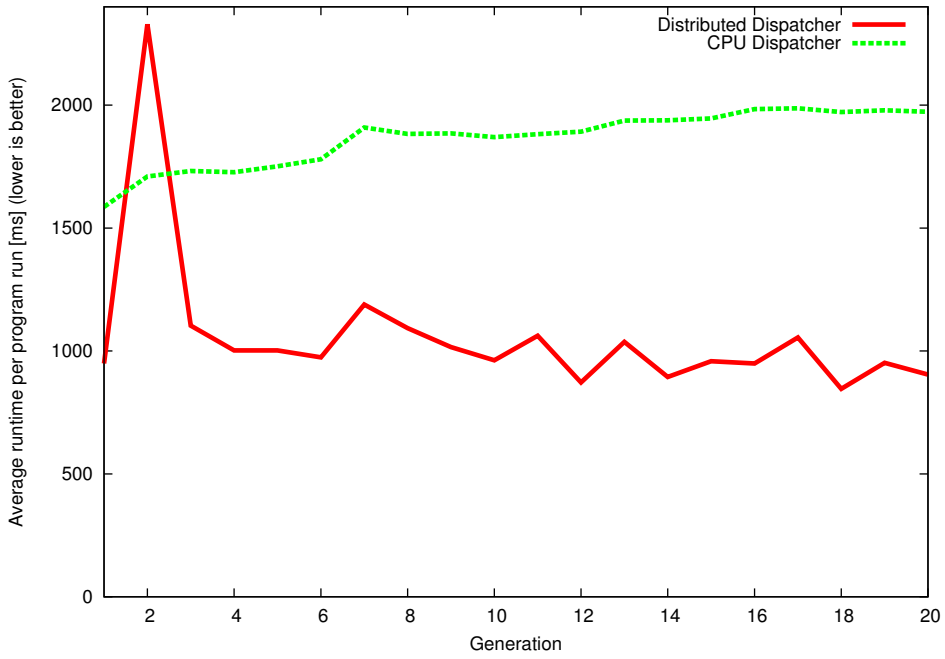

Figure 7.13: Average program complexity for different population sizes

expect a decrease in runtime, but not an enormous difference.

Experimental Results

This experiment was done with the settings in Table 7.1, with 4 runs for the distributed dispatcher, and 1 run for the cpu dispatcher. It must be noted that the cpu dispatcher is much more stable in terms of runtimes (small overhead of task delegation etc.) so doing 1 run suffices. At the time of running the distributed dispatcher, there were 9 active clients connected to the network and no other servers, so the clients were only running this experiment.

The experiment with the cpu dispatcher was conducted on a 16 core using its full capabilities. For the distributed dispatcher, the clients only use $N/2+1$ cores in order to try to keep the machine usable for the user running it.

In Figure 7.13 we can see that the distributed dispatcher has better runtimes than the CPU dispatcher. What is also interesting is that the distributed dispatcher shows a spike at the second generation. This is due to the fact that the second generation usually has a small number of programs, and the distributed dispatcher has some overhead in distributing the programs and waiting for results. The spike occurs because this overhead time is divided over running only a few programs in that generation as opposed to many in the other generations. We think that overall this is a very nice and promising result, especially because the communication process can be optimized much further, making it more efficient.

Chapter 8

Discussion

This chapter gives an overview of the points open to discussion. As these points are highly related to the major topics of the thesis, they are organized per topic.

8.1 Distributed Algorithms for Large Scale-Systems

8.1.1 Simulation realism

As already noted in Section 5.3, there is a lack of realism in NetLogo simulations. By having a deterministic way of traversing over all the nodes and executing their code sequentially, we miss the aspect of chaos that is inherently important for large-scale systems. This is not to say that NetLogo does not produce random behaviors. The factors that are currently random in our simulations are especially the distribution of nodes in a virtual field, and the initialization of state variables (either global or local) that are set to be random by the user. We already do multiple experiments per program we execute because of these random factors, but if we were to replace NetLogo by a more realistic simulator, we would probably have to do more experiments per program to get a stable measurement. Nevertheless, the framework is designed to be agnostic to the kind of simulator, making this a nice possibility to extend the MetaCompiler.

8.1.2 A unified language for Large-Scale Systems

One yet unaddressed point of discussion is the lack of a unified way to express goals (i.e. the fitness function), and building blocks for the MetaCompiler. During the work we have done on the MetaCompiler, we have often used existing distributed algorithms like the ones mentioned in Section 5.3 to test the system.

Even though we have worked to define the field of computing for which we want to generate algorithms, we were never able to achieve a generic language to express the building blocks or fitness functions.

Why is this important? The envisioned goal for the MetaCompiler is to provide a very high level interface for algorithm designers to define a global goal, and let the framework generate a local program to achieve this behavior. However, right now implicit knowledge of the algorithmic building blocks is needed to define the fitness function. For example, what variable is adjusted by a building block, such that we know what variable to ask for in the fitness function. Also, the connectivity between building blocks suffers from the lack of a common language. If we build a library from all the building blocks defined in the examples, we could make a fitness function that describes behaviors of all of the examples by joining the example specific fitness functions together in one function.

However, it would already be a lot harder to define something that is not yet covered in the examples but possible with the building blocks. The algorithm designer would have to look up the variables used by the examples. This already defeats the global-to-local purpose because the designer has to use his own knowledge of the local algorithms. For somebody who has no knowledge of the building blocks, also a targeted user of the MetaCompiler, it would be impossible at this moment to provide the framework with a global goal.

Even though we have not been able to solve this problem, we think it should be possible to design a library of building blocks that are more coherent than the current examples. For example, a library of building blocks that is focused on the subject of movement and positioning could provide a lot of building blocks regarding movement actions and information about the position of a node relative to other nodes. This would already be helpful in solving geographical clustering problems, and the algorithm designer only has to have knowledge of a few keywords used by this library.

In short, having a totally coherent way of defining global goals and consistency between building blocks may be difficult, but it should be possible to do better than the current state of the examples we use.

8.2 Genetic Programming

8.2.1 The problems of defining a good fitness function

One of our most profound challenges in using the framework we created, has been to design fitness functions that guide the search process well. Whenever we managed to design a fitness function that worked, it became more apparent that there is nothing magical about a good fitness function. It is mostly trading in the task of designing local behavior for designing and refining the global behavior, because there are often aspects in the generated result the designer did not foresee in the first iteration. This is however not in contradiction to the goals of the

MetaCompiler, and after all there is no such thing as a free lunch[57], but it takes some time getting used to.

The difficulty of defining a fitness function has multiple reasons, described in the following sections.

It's an iterative process

As mentioned above, the MetaCompiler will often come up with a solution that contains aspects you would not have thought of in the first iteration, in other words a program with a good fitness is not good per definition. Designers can refine the fitness function and try it again. The problem here is that the Genetic Programming process at this moment takes too long to run, making iterating a bit of a cumbersome process. A positive aspect of these iterations is that they are very insightful for the designer, often the results are logical consequences of the provided global goal. If experiments could be done in less time, we think these iterations on the fitness function are no problem and maybe even good for the designer.

A good fitness function is often multi-objective

As described above, a good fitness function is often a collection of wanted and unwanted behavior captured as a single goal. The fitness function does however return a single value for all of the subgoals combined. This creates a problem because if in program 1 only subgoal A scores well, and in program 2 only subgoal O scores well, we are effectively comparing A and O when we express everything in one value. It is the problem for the designer to decide how the different subgoals are weighed against each other, while he probably wants to achieve all goals instead of prioritizing them. To solve this problem, we should use a technique called Multi-Objective Optimization, which recognizes there are different subgoals, and tries to optimize all of them instead, maintaining a diverse population, which is good for program discovery.

Some things are not a gradient

Genetic Programming (and other EC strategies) work best if they can gradually work towards a solution. In this way there is direct feedback to the GP process whether the mutation or crossover it just did worked or not, and it can gradually move to a better solution. Often times however a goal is just boolean, it is achieved or not. In these cases it is hard for Genetic Programming to have a clue whether it is close to finding the good solution and it will possibly never find it. The designer is left with the task to try to define a gradual value to solutions that are not good but close to achieving the goal. The problem here is that even though this could sometimes be possible, this challenge may lead the designer to think about the local behavior rather than the global behavior, which is of course

not the intention of the MetaCompiler, where designers are ought to think about the global behavior.

8.3 Genetic Programming Enhancements

8.3.1 Initialization

Our initialization tactic of optimizing for size before optimizing for functionality (Section 4.1) had some issues. We discourage its use but recognize that there are still problems with the initialization process, as can be seen in Figure 7.7 and Figure 7.8. Resolving these issues would be good for the diversity of the population and hence the overall results generated by the MetaCompiler.

8.3.2 Tree-based operators

We strongly believe that our tree based operators are better at fulfilling the functions of crossover and mutation. The experiments have however not convincingly shown that this is true or that it results in better fitnesses. The only reason we know for this difference is the lack of an advanced genotype-phenotype mapping (Section 4.2.4), so we should add this to the genetic operators and test the performance again to see if they match our expectations better, or that there are other problems.

8.3.3 Quality of pattern detection

The experimental chapter has shown that the way in which the MetaCompiler does pattern detection and protection is in some cases helpful for the process of achieving a better fitness, but it is very dependent on the size of the training set. If the size of the training set is not at its optimal setting, the experiments have shown that using pattern detection causes more damage than it does good.

It is nice that our simple implementation of pattern detection has shown to have worked, but we would need to do more testing to obtain guidelines for how to use it. We would have preferred to use a scientifically proven, and thoroughly tested method if it would be applicable in our framework, but we were unable to find it in literature. Also, the fact that our pattern protection approach only uses the current population as a knowledge base is something that could probably be improved upon by building up a knowledge base of well performing patterns in other problems, with a side note that well performing patterns may be very problem specific, in which case that would not work.

8.4 Implementation

8.4.1 A year of iterative implementation

The first and only take on the MetaCompiler has been under development for almost a year. The good thing is that the initial design is still nearly the same. Because we have built the MetaCompiler mostly upon EpochX, and EpochX is built for extension, we were able to extend especially the GP part without having to write dirty code. Also almost all of the statistics we gather were already embedded into EpochX, and the statistics that were specific to the MetaCompiler were easily added to EpochX.

What the code does need is a descent cleanup. Especially the first phase of the project, where (because the design was already known) many parts were built in parallel, resulted in a lot of code put together in only a few classes. In the following phases, a lot of tweaks were incrementally added to the code following our experiences with early versions of the MetaCompiler. It would be good for the future if the functionality was split up in well-defined modules, now that the tweaking phase is over. Because of the very specific nature of the MetaCompiler, we do not believe that these modules will ever be off-the-shelf modules for other projects, but it will merely help in maintaining the code. The parts that were added to the MetaCompiler later, like the new operators, pattern detection, and the distributed dispatcher, have been built up more modularly, so they need no extra attention.

8.5 Experimental Results

8.5.1 Search space

Since the MetaCompiler takes an holistic approach to algorithm design, it will have to search through a combination of many building blocks to find the right solution. In our examples, we used sets of building blocks that were relevant to discovering the algorithm the example was targeted at. The experiment in Section 7.2 shows that finding a combination of the right building blocks can be difficult. We expect that we will have to work with very large population sizes in order to generate real world solutions, in which there are many building blocks available for the MetaCompiler to use. Fortunately we have already found a way to fully parallelize the evaluation phase of a population with the distributed dispatcher (see Section 8.5.2), so with enough computers to back up this process, having large populations is feasible.

8.5.2 Execution time

In the experiments, we have shown that the current execution times of the MetaCompiler are still too high, and increase drastically with problem complexity, population size, the number of generations, and the number of runs. Because

every generation is dependent on the previous generation, different generations are not parallelizable, but everything within one generation is, and separate runs of the GP process are as well. The distributed dispatcher already parallelizes program executions within one generation, but at this point the overhead of communication is still too high to make a profound difference. Since there is still a lot of room for optimization in the distribution process, this approach remains promising. Also, execution of experiments on the GPU is in the making. We expect that reducing the execution time of the MetaCompiler will change the user experience, and is very important for making it an interesting and accessible tool for designers.

8.5.3 Abundance of Parameters

The MetaCompiler offers a number of settings that can be adjusted by the user, and the experiments have clearly shown is that each of these settings is of very big importance to the search process. The experiments were deliberately done on each parameter separately, to show their influence, but when using the MetaCompiler, we will have to set all parameters at once. Foreseeing the influence of that particular combination of settings is very hard, especially for a novice user like a designer, which is in the end the envisioned user for the MetaCompiler. In addition, the settings are also very dependent on the problem at hand, complicating the parameter issue even more.

We think that adding automatic parameter tuning would help the user because the MetaCompiler would be able to automatically adapt the settings in order to achieve a good solution without user intervention. The problem with automatic parameter tuning is however that it will increase the runtime of the MetaCompiler because multiple parameter configurations are tested to see which one works best. Automatic parameter tuning remains interesting, but will only be really useful if the runtimes of the MetaCompiler are drastically decreased.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

The main goal of this thesis was to develop a global-to-local compiler for generating distributed algorithms for large-scale systems. We have done so by integrating Genetic Programming and large-scale system specific algorithm design.

For Genetic Programming we have used the Grammatical Evolution approach because of its flexibility and adaptivity in creating arbitrary target languages. To target our global-to-local compiler to large-scale systems, we have written a custom language that can be used to describe typical algorithms designed for these kinds of systems. In combination with Grammatical Evolution, we are able to explore the search space of these algorithms, creating solutions that are specifically applicable to large-scale systems.

To increase the usability of the glocal-to-local compiler, we have enhanced the Genetic Programming process in three major ways: The first enhancement is the use of new genetic operators that are used to evolve populations through the Genetic Programming process. Since the new genetic operators are a better match to the structure of the generated programs than the old genetic operators, they are able to find new programs more intelligently.

The second enhancement is a pattern detection algorithm that works on the generated programs. It serves to protect valuable structures in good programs, and to diverge the search to enhance other structures. The experiments have shown this can indeed enhance the search process, in order to find better programs more quickly.

The third enhancement is a way to offload the computing power Genetic Programming needs to evaluate programs to other computers. This helps to decrease

the computing time immensely, and makes the process resilient to individual machine failures, two very important properties when using Genetic Programming.

In conclusion, we have created a full global-to-local compiler generating distributed algorithms for large-scale systems, and we have optimized it to be usable by algorithm designers. We have achieved a holistic approach, allowing the algorithm designer to be agnostic to the algorithm specifics, increasing the speed of algorithm design, but never at the cost of flexibility.

9.2 Future Work

The work we have done allows for future work in a couple of directions.

To better establish that the global-to-local is fit for real-life applications there needs to be more testing.

The first scenario to test is trying to invent new algorithms with the global-to-local compiler. We have tested a number of existing examples to test our framework, but we have not yet had an idea for a new algorithm we wanted to discover. Trying to do this may give more insight on how to make well connecting building blocks, and the computing power that is needed to develop advanced algorithms with these building blocks.

Another interesting test is to see how well discovered algorithms run on real-life hardware. Since the Snowdrop project has already brought forth a hardware platform to run large-scale applications, this is perfectly possible. To fully automate this process, the next step can be to automatically translate generated programs into programs for the target platform, something the global-to-local compiler is designed to be able to do.

Another part that needs more testing is the pattern detection mechanism we have built. The experiments have shown that pattern detection can be of advantage, but only in some cases. To be really useful, we need to know when, and on what kinds of algorithms the pattern detection works well in order to benefit from it.

This also extends to other parameters for the global-to-local compiler, and thus automatic parameter tuning is a good feature to build into it.

Since computing time is always valuable time, every attempt for decreasing this time helps to make the global-to-local compiler a more usable and hopefully popular tool.

Bibliography

[1] P.J. Angeline. Genetic programming and emergent intelligence. *Advances in genetic programming*, 1:75–98, 1994.

[2] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[3] J. Bachrach, J. Beal, and T. Fujiwara. Continuous space-time semantics allow adaptive program execution. In *SASO'07. First International Conference on*, pages 315–319. IEEE, 2007.

[4] D.H. Ballard. Discovery of Subroutines in Genetic Programming. 1996.

[5] W. Banzhaf. Genotype-phenotype-mapping and neutral variation - a case study in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard MÃďnner, editors, *Parallel Problem Solving from Nature - PPSN III*, volume 866 of *Lecture Notes in Computer Science*, pages 322–332. Springer Berlin / Heidelberg, 1994.

[6] W. Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C Fogarty, editors. *Grammatical evolution: Evolving programs for an arbitrary language*, volume 1391 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin/Heidelberg, 1998.

[7] J. Beal and J. Bachrach. Mit proto, 2011.

[8] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *Evolutionary Computation, IEEE Transactions on*, 5(1):17–26, 2001.

[9] Y. Brun and all. Engineering self-adaptive systems through feedback loops. *Software Engineering for Self-Adaptive Systems*, pages 48–70, 2009.

[10] M. Conti, G. Maselli, G. Turi, and S. Giordano. Cross-layering in mobile ad hoc network design. *Computer*, 2004.

[11] C. Darwin, J.W. Burrow, et al. *The origin of species by means of natural selection: or, the preservation of favored races in the struggle for life*. AL Burt, 2009.

[12] D.W. Dyer. Watchmaker framework for evolutionary computation. *URL: http://watchmaker.uncommons.org*, 2006.

[13] M. Fiedler, T. Hossfeld, and P. Tran-Gia. A generic quantitative relationship between quality of experience and quality of service. *Network, IEEE*, 2010.

[14] P. Fortescue, G. Swinerd, and J. Stark. *Spacecraft systems engineering*. Wiley, 2011.

[15] G.N. Frederickson and N.A. Lynch. Electing a leader in a synchronous ring. *Journal of the ACM (JACM)*, 34(1):98–115, 1987.

[16] M. García-Arnau, D Manrique, J Rios, and A. Rodríguez-Patón. Initialization method for grammar-guided genetic programming. *Knowledge-Based Systems*, 20(2):127–133, 2007.

[17] R. Harper and A. Blair. A structure preserving crossover in grammatical evolution. *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, 3:2537–2544 Vol. 3, 2005.

[18] R. Harper and A. Blair. A self-selecting crossover operator. *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 1420–1427, 2006.

[19] Eggermont J. Genetic programming. In *IPA Herfstdagen 2007*, 2007.

[20] S. Karger, A. Di Figlia, M. Bos, A. Pruteanu, and S. Dulman. Spatial computing for non-it specialists. *Spatial Computing Workshop (SCW2012), AAMAS2012*, 2012.

[21] R.E. Keller and W. Banzhaf. Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In *Proceedings of the First Annual Conference on Genetic Programming*, GECCO '96, pages 116–122, Cambridge, MA, USA, 1996. MIT Press.

[22] J.R. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems, 1990.

[23] J.R. Koza. *Koza: Genetic Programming: On the Programming of... - Google Scholar*. See http://miriad. Iip6. fr/microbes Modeling Adaptive . . . , 1992.

[24] J.R. Koza. *Genetic programming II: automatic discovery of reusable programs*. 1994.

[25] W.B. Langdon, R. Poli, N.F. McPhee, and J.R. Koza. *Studies in Computational Intelligence*, volume 115 of *JanuszKacprzykStudies in Computational Intelligence1860-949X*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[26] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, J. Bassett, R. Hubley, and A. Chircop. Ecj: A java-based evolutionary computation research system, 2007.

[27] S. Luke and D. Sharma. Finding Interesting Things: Population-based Adaptive Parameter Sweeping. 2007.

[28] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd., 2003.

[29] K. Meffert, N. Rotstan, C. Knowles, and U. Sangiorgi. Jgap-java genetic algorithms and genetic programming package. *URL: http://jgap. sf. net*, 2009.

[30] J.F. Miller and S.L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *Evolutionary Computation, IEEE Transactions on*, 10(2):167 – 174, april 2006.

[31] D.P. Miner. A framework for predicting and controlling system-level properties of agent-based models. 2011.

[32] D. Mishra and A. Mishra. Complex software project development: agile methods adoption. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[33] J.C. Mogul. Emergent (mis) behavior vs. complex software systems. In *ACM SIGOPS Operating Systems Review*, 2006.

[34] D.J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199–230, June 1995.

[35] E. Murphy, E. Hemberg, M. Nicolau, M. O'Neill, and A. Brabazon. *Grammar Bias and InitialisationâĂĺin Grammar Based Genetic Programming* , volume 7244 of *DavidHutchisonTakeoKanadeJosefKittlerJon M.KleinbergFriedemannMatternJohn C.MitchellMoniNaorOscarNierstraszC.Pandu RanganBernhardSteffenMadhuSudanDemetriTerzopoulosDougTygarMoshe Y.VardiGerhardWeikumLecture Notes in Computer Science0302-97431611-3349*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[36] E. Murphy, M. O'Neill, E. Galván-López, and A. Brabazon. [TAGE] Tree-Adjunct Grammatical Evolution. In *2010 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE.

[37] P. Mutton. Pircbot java irc api: Have fun with java. *Java Developer's Journal*, 8(12):26–32, 2003.

[38] R. Nagpal and M. Mamei. Engineering amorphous computing systems. *Methodologies and Software Engineering for Agent Systems*, pages 303–320, 2004.

[39] M. O'Neill, A. Brabazon, and M. Nicolau. [πGE] πGrammatical Evolution. *Genetic and ...*, 2004.

[40] A. Ortega, M. de la Cruz, and M. Alfonseca. Christiansen Grammar Evolution: Grammatical Evolution With Semantics. *Evolutionary Computation, IEEE Transactions on*, 11(1):77–90, 2007.

[41] 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*, pages 93–100. ACM, 2012.

[42] L. Panait and S. Luke. A pheromone-based utility model for collaborative foraging. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 36–43. IEEE Computer Society, 2004.

[43] A. Pruteanu, S. Dulman, and K. Langendoen. Ash: Tackling node mobility in large-scale networks. In *SASO,2010*, pages 144–153, 2010.

[44] A. Pruteanu, V. Iyer, and S. Dulman. Churndetect: a gossip-based churn estimator for large-scale dynamic networks. *Euro-Par 2011 Parallel Processing*, pages 289–301, 2011.

[45] A. Pruteanu, V. Iyer, and S. Dulman. Faildetect: Gossip-based failure estimator for large-scale dynamic networks. In *ICCCN,2011*, pages 1–6. IEEE, 2011.

[46] J.P. Rosca and D.H. Ballard. Genetic programming with adaptive representations. 1994.

[47] C. Ryan and RMA Azad. Sensible initialisation in grammatical evolution. In *GECCO*, pages 142–145, 2003.

[48] F. Stonedahl and U. Wilensky. Behaviorsearch [computer software]. *Center for Connected Learning and Computer Based Modeling, Northwestern University, Evanston, IL. Available online: http://www. behaviorsearch. org*, 2010.

[49] S. Tisue. NetLogo: A simple environment for modeling complexity. *International Conference on ...*, 2004.

[50] S. Tisue and U. Wilensky. Netlogo: A simple environment for modeling complexity. In *International Conference on Complex Systems*, pages 16–21, 2004.

[51] 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*, GECCO Companion '12, pages 337–344, New York, NY, USA, 2012. ACM.

[52] Werner-Allen and all. Firefly-inspired sensor network synchronicity with realistic radio effects. In *Sensys*, 2005.

[53] P.A. Whigham. Grammatically-based genetic programming. In *Proceedings of the Workshop on Genetic Programming ...*, 1995.

[54] Wikipedia. 2600: The hacker quarterly — wikipedia, the free encyclopedia, 2012. [Online; accessed 8-August-2012].

[55] Wikipedia. Backus-naur form — wikipedia, the free encyclopedia, 2012. [Online; accessed 13-June-2012].

[56] Wikipedia. FisherâĂŞyates shuffle — wikipedia, the free encyclopedia, 2012. [Online; accessed 4-July-2012].
[57] Wikipedia. There ain't no such thing as a free lunch — wikipedia, the free encyclopedia, 2012. [Online; accessed 5-August-2012].
[58] Wikipedia. Turing machine — wikipedia, the free encyclopedia, 2012. [Online; accessed 7-August-2012].
[59] D. Yamins and R. Nagpal. Automated global-to-local programming in 1-d spatial multi-agent systems. pages 615–622, 2008.
[60] F. Zambonelli and M. Mamei. Spatial computing: An emerging paradigm for autonomic computing and communication. *Autonomic Communication*, 2005.