IT 11 014
Examensarbete 30 hp
Mars 2011

# Automatic Modularization in Genetic Programming

Martin Norin

Institutionen för informationsteknologi
*Department of Information Technology*

UPPSALA
UNIVERSITET

**Teknisk- naturvetenskaplig fakultet**
**UTH-enheten**

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

# Abstract

## Automatic Modularization in Genetic Programming

*Martin Norin*

This master's thesis is an investigation into automatically created functions when applying Genetic Programming to a problem. The thesis begins with a description of Genetic Programming and the different parts that is required to apply Genetic Programming to a problem. This is followed by descriptions of different methods for automatically creating functions when using Genetic Programming as a problem solving method. A new method for automatically creating functions is proposed with the name Structural Module Creation. Structural Module Creation creates new functions based on the most frequent subtrees in the population of individuals. Experiments are conducted on the even-k-parity problem to compare Structural Module Creation to Module Acquisition (another modularization method) and Genetic Programming without a modularization method. The result of the different experiments showed no improvement when using Structural Module Creation compared to Module Acquisition and Genetic Programming without a modularization method. The conclusion that can be drawn is that Structural Module Creation is not applicable to the even-k-parity problem. Appendix A contains graphs depicting the different experiments. Appendix B contains a description of the implementation of the system.

Handledare: Jim Wilenius
Ämnesgranskare: Olle Gällmo
Examinator: Anders Jansson
IT 11 014
Tryckt av: Reprocentralen ITC

# Sammanfattning

Detta examensarbete undersöker hur man automatiskt kan skapa funktioner vid applicering av Genetisk Programmering på ett problem. Rapporten börjar med att beskriva Genetisk Programmering samt de olika delarna som krävs för att kunna applicera Genetisk Programmering på ett problem. Detta följs av beskrivningar av olika metoder för att automatiskt skapa funktioner när man använder Genetisk Programmering som problemlösningsmetod. En ny metod för att automatiskt skapa funktioner föreslås med namnet Structural Module Creation. Structural Module Creation skapar nya funktioner baserat på de mest frekventa delträden i populationen. Experiment körs på even-k-parity problemet för att jämföra Structural Module Creation med Module Acquisition (en annan modulariseringsmetod) och Genetisk Programmering utan någon modulariseringsmetod. Resultaten av de olika experimenten visade ingen förbättring när Structural Module Creation användes jämfört med Module Acquisition och vanlig Genetisk Programmering. Slutsatsen är att Structural Module Creation inte är användbart för even-k-parity problemet som användes för experimenten. Appendix A innehåller grafer för de olika experimenten. Appendix B innehåller en beskrivning av implementationen av systemet.

# Acknowledgements

I would like to thank my thesis advisor Jim Wilenius and my thesis reviewer Olle Gällmo for their help, support and patience in writing this thesis.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

There is something exciting about the notion of a computer solving problems on its own or rewriting its own code to adapt to changing conditions. Genetic programming is a step in that direction.

Genetic programming can be seen as semi-automatic programming, a way of removing the manual writing of programs in areas where either the solution is unknown, for example finding a mathematical function that corresponds to given input-output pairs, optimization, or where the complexity of the problem makes it hard to find a good solution by hand. What exactly constitutes Genetic Programming varies but this thesis is based on the work of John Koza [Koza, 1992], [Koza, 1994b], [Koza et al., 1999] and [Koza et al.].

In the case of solving complex problems it is sometimes not desirable to write one big program but to divide the problem into smaller subproblems. Influenced by human programmers it is natural to introduce functions to increase the expressiveness of the genetic programming system, to structure the code in a more efficient way and enabling code reuse and a logical division of the program structure. This can sometimes be essential in order to even be able to solve the problem, because it is not always possible to evolve a solution in a reasonable amount of time with only basic computer instructions. Furthermore, a not unimportant part of the divide-and-conquer paradigm is that a program comprised of small parts is easier to read and understand for a human than one long string of instructions.

The role of the newly introduced functions, besides from code reuse and aforementioned reasons, is to protect parts of the program that is deemed good *building blocks* from being destroyed in the evolutionary process. Introducing new functions can either be done by the programmer before starting the genetic programming process, or it can be done automatically during the evolution of the program. The latter makes for an increased automation of the Genetic Programming process. By automating the process of introducing new functions it is possible to remove any direct bias from the programmer regarding the structure and content of the module[1]. It also enables the genetic programming system to find functions that the programmer were unaware of. In essence, it gives great freedom of what the solution will look like. However, this freedom comes with a price. If the evolutionary process fails to evolve the necessary building blocks that could be provided by the programmer because of prior knowledge, there is a risk that a solution will not be found in time, or at all. It remains to be seen if providing the genetic programming system with only basic instructions is enough to find a solution given the right modularization method.

[1] Indirect bias exists in the shape of the fitness function.

## 1.2 Thesis overview

Chapter 2 contains a brief explanation of genetic programming. It describes how genetic programming is inspired by evolutionary theory of Charles Darwin and how it translates to programming. This chapter also explains the necessary elements that the programmer has to provide the system with before genetic programming can be applied to the given problem. Chapter 3 explains the theory of building blocks and different methods of automatic modularization. That is, how to automatically create new functions from existing computer instructions and how to use them in the evolutionary process. The modularization methods that are described are *Encapsulation*, *Automatically Defined Functions*, *Automatically Defined Functions with Architecture-Altering Operations*, *Automatically Defined Macros*, *Adaptive Representation*, *Adaptive Representation through Learning*, and *Module Acquisition*. Chapter 4 contains my proposed modularization method based on Module Acquisition, called *Structural Module Creation*, that creates new modules based on the frequency of subtrees with the same structure. Chapter 5 contains the experiments. The even-k-parity problem (k = 3,4,5) is used as the benchmark problem to compare standard genetic programming with Module Acquisition, and Structural Module Creation. Chapter 6 contains the conclusions of the thesis. Chapter 7 contains further work. Appendix A contains graphs of experiments showing the average fitness of the best, worst, and average individual over 50 iterations. Appendix B explains the implementation of the genetic programming system, including the arguments that can be given to the system to control the evolutionary process.

# Chapter 2

# Genetic Programming

## 2.1 Introduction

Genetic programming (GP) is a domain-independent problem solving method influenced by Darwin's theory of survival of the fittest. Genetic programming belongs to the family of problem solving methods called Evolutionary Computation, including Genetic Algorithms, Evolution Strategies, and Evolutionary Programming [Jong, 2007].

In genetic programming, a *population* of different computer programs, referred to as *individuals*, is used to breed a solution to a given problem. The breeding process includes operations on programs where the most common are *crossover* (a form of sexual recombination of individuals), *mutation*, and *reproduction* (not in the sense of human reproduction, but rather a copying process). Each individual is evaluated on the specific problem and given a numerical value called the *fitness*, representing how well the individual solves the problem. The idea is that programs that have a high fitness will combine to make more fit offspring that are closer to the solution.

Seen from another point of view, genetic programming performs a parallel search through the search space of possible programs[1]. Each unique individual in the population represents a point in the search space. Depending on the size of the population and the distribution of the individuals in the search space, a large part of the search space can be covered, thus minimizing the danger of being trapped in local minima.

Genetic Programming is an extension of Genetic Algorithms (GA) [Holland, 1975]. Genetic Algorithms uses binary strings of a fixed length to encode the problem, whereas Genetic Programming uses computer programs of varying lengths to represent the problems. One difference between Genetic Algorithms and Koza's version of Genetic Programming is that the mutation operation is rarely used in Koza's version of Genetic Programming, see section 2.7.2.

The process of genetic programming can be summarized in three steps:

1. Generate a random initial population
2. Repeat the following until the termination criteria has been fulfilled:
   (a) Evaluate each individual in the population and assign it a fitness value
   (b) Create a new generation of individuals using genetic operators. The new generation replaces the old generation in the population.
3. The best individual in the last generation is the solution, or approximate solution to the problem

[1] The search space is dependent on the function set and the terminal set.

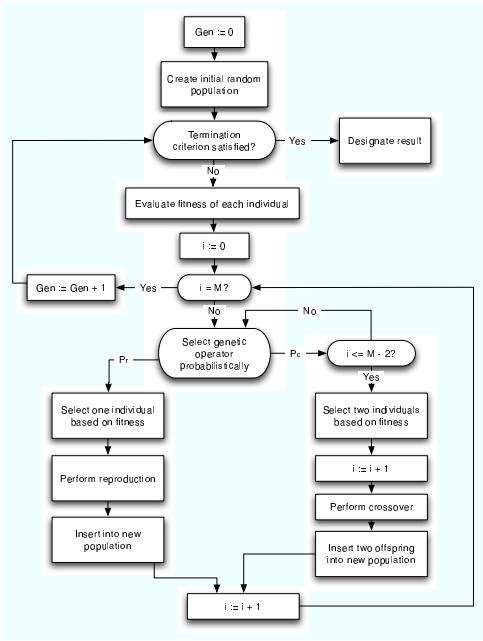



Figure 2.1: A flowchart of the genetic programming process.

A flowchart can be seen in figure 2.1.

Before genetic programming can be applied, there are some preparatory steps that need to be taken. [Koza, 1994b] lists five major preparatory steps that specifices

1. the set of terminals (section 2.3)
2. the set of functions (section 2.4)
3. the fitness measure (section 2.5)
4. the parameters for controlling the execution of the Genetic Programming system (section 2.8), including the selection method (section 2.6) and the probabilities for the genetic operators (section 2.7)
5. the method for designating a result and the termination criterion (section 2.9).

Each of these steps will be covered in their corresponding sections. Due to the probabilistic and random nature of genetic programming, it is sometimes necessary to repeat the Genetic Programming process in order to find a solution.

## 2.2 Program representation

Computer programs can be represented in different ways. [Banzhaf et al., 1998] lists three main styles of program representation: *linear* representation, *tree* representation, and *graph* representation. In linear representation the operations are executed in sequential order. [Nordin, 1994] used machine code to increase the efficiency of program evaluation. In tree representation, the execution of the program can be in postfix order, where each child node is evaluated before the parent node; or in prefix order, where the parent node is evaluated first, followed by evaluation of the child nodes. The graph representation consists of nodes and directed edges between nodes. Graph representation is well suited for programs containing loops and recursion [Banzhaf et al., 1998]. In this paper, the focus is on tree representation with postfix execution order. Also, a

limitation of the program size has been set for practical reasons. The depth of the tree and the number of nodes are two different size measurements that are used.

## 2.3 Terminal set

The terminal set is how the genetic programming system takes input from the environment. The terminals can, for example, represent sensors on a robot, or variables in a mathematical function. In the even-k-parity problem [Koza, 1994b], the terminal set corresponds to the inputs of the circuit, named D0, D1, D2, ..., D(k-1) with TRUE and FALSE as possible values. Constants and zero-argument functions can also be included in the terminal set [Banzhaf et al., 1998].

## 2.4 Function set

The function set contains the instructions that are to be carried out by the genetic programming system. They are the basic building blocks that are used to solve the problem. The function set is problem specific and defines the search space. According to [Koza et al., 1999], operators usually used in the function set are addition, subtraction, multiplication, division, and a conditional branch operation, for example the if-then operation. This specific function set is more suited for mathematical problems, while another function set comprised of the logical operators AND, OR and NOT are more suited for example in problems like the even-k-parity problem [Koza, 1994b].

The function set must contain operators such that the solution (or solutions) is included in the set of possible programs, otherwise the solution will, obviously, never be found. This property is called the *sufficiency property*. Making sure it holds often requires a deep understanding of the problem [Koza, 1994b]. Another difficulty is that the function set cannot be to large. By increasing the function set, the search space also grows which in turn makes the search harder [Banzhaf et al., 1998]. A property that is desirable for the functions in the function set to have is the *closure property*. It means that the entire terminal set as well as every possible output from every function in the function set is valid input to every function in the function set. For example, a function set comprised of the logical operators AND, OR, and NOT satisfies the closure property if the terminal set only contains the terminals TRUE or FALSE and the logical operators only produce TRUE or FALSE. However, if the terminal set included a terminal that isn't a valid boolean value then the function set would not satisfy the closure property. Ordinary division is an operator that does not satisfy the closure property since division by zero is not allowed[2]. This can be remedied by introducing *protected division*, an augmented division operator. It works as the regular division operation but if the denominator is zero, the result of the operation is 1.

## 2.5 Fitness function

The evolutionary process needs some kind of measure of how well each individual solves the given problem. This measure, called the *fitness measure*, is used to guide the evolutionary process in directions that look promising by selecting with high probability individuals with high fitness. The fitness measure is calculated by the *fitness function* which is problem specific. The fitness function is the core of Genetic Programming. It is applied to every individual in every generation so optimizing the fitness function with respect to computational complexity should be a priority to improve performance.

There are different types of fitness measures.The basic fitness measure that is used is called *raw fitness*. It is a direct measure of the fitness function, for example the number of correct classifications in a classification problem. *Standardized fitness* is a measure independent of the problem. Lower values of standardized fitness are better than higher values and the best fitness is always 0. This reformulates the problem to a minimization of standardized fitness instead of

[2]Division satisfies the closure property if no function produces 0 and no terminal can have the value 0

maximizing fitness. Standardized fitness is calculated by subtracting the raw fitness from the maximum value of fitness for that given problem. Standardized fitness is used for example in classification problems. A standardized fitness of 0 means that all fitness cases have been correctly classified. My research has not uncovered any information that standardized fitness can be used in areas where a maximum value of fitness is not available so I assume that standardized fitness cannot be used on problems where a maximum fitness value doesn't exist, or is unknown.

Another fitness measure is the *normalized fitness* which assigns each individual a value between 0.0 and 1.0, where the higher values are better, calculated by dividing the raw fitness of the individual with the combined raw fitness of the entire population. Normalized fitness is used when the selection method is fitness-proportional but it is not relevant, and therefore not used, when the selection method is tournament selection (section 2.6) [Koza, 1992].

Fitness can be measured in different ways, either the cumulative error from a set of *fitness cases*, which are representative of the search space, or by some numerical value that comes from the problem specification, for example the number of squares mowed in the lawnmower problem [Koza, 1994b], or the number of correct classifications in a classification problem. The fitness cases consist of inputs to the problem with their corresponding desired outputs. They can either be chosen by the programmer, or can be sampled from the search space at random [Koza, 1994a]. In the case of the even-k-parity problem, which is the benchmark problem used in the experiments, the fitness cases are all permutations of $k$ arguments.

Sometimes finding a solution is not enough, other criteria has to be satisfied as well. For example, a solution that is compact is often preferable to a larger, but equally correct, solution. In those cases, a *multi-objective fitness measure* is used. When size is of importance, the fitness is a combination of the fitness value from solving the problem and a size measurement, for example, the number of nodes or the depth of the solution in tree representation.

Since the fitness function is applied to every individual in every generation it is clear that the complexity of the fitness function affects the overall complexity of Genetic Programming. The complexity of the fitness function is problem specific and it is therefore not possible to give a general statement on the complexity. Looking at the problem that is used for testing in this thesis, the even-k-parity problem, the complexity of the fitness function is linear with respect to the number of nodes for each individual.

## 2.6 Selection

If the fitness function is the guiding force in Genetic Programming, then *selection* is the actual road taken by the Genetic Programming system. [Banzhaf et al., 1998] say that the most important decisions to be made in the application of genetic programming is the one of choosing the selection method. One reason for its importance is that selection is responsible for the speed of evolution. The speed of evolution is the number of generations it takes to find a solution.

The choice of selection method also affects the overall number of computational steps that is required to find a solution.

The following are some of the selection methods that can be used:

- **Fitness-proportional selection**, or **roulette-wheel selection**, assigns each individual a probability of being selected based on the individuals fitness. The probability is calculated by dividing the fitness of the individual with the combined fitness of the entire population.
- **Ranking selection** works by sorting the individuals based on their fitness into a hierarchy similar to a ladder, giving each individual a rank based on its position in the ladder. Each rank has a probability of being selected. The effect of this form of selection is that individuals with extremely high fitness will be selected less frequently, thus giving less fit individuals a better chance of being selected, leveling the playing field a little[3]. Another effect of rank

[3]The current highly fit individuals can be suboptimal solutions, so it can be advantageous to limit their dominance.

selection is that if many highly fit individuals are grouped together in the search space, they are more likely to be selected [Koza, 1992].

- In **tournament selection** [Banzhaf et al., 1998] a specified number of individuals (two or more) are selected uniformly from the population to be in the tournament. The individual with the highest fitness in the tournament is selected for the genetic operation. By changing the size of the tournament, a selection pressure can be added. A large tournament size decreases the chance of selecting individuals with lower fitness, while a smaller tournament size increases the chance for individuals with low fitness to be selected. Also, it doesn't require a centralized fitness comparison between all individuals in the population. That means that it removes the sorting of all individuals according to fitness, which saves time, especially with large populations.

The speed of evolution is affected by the choice of selection method. An example of this can be seen in [Koza, 1994b] where fitness-proportional selection is compared to tournament selection.

As stated above there is a difference in the number of computational steps per generation for the different selection methods. This becomes clear when comparing fitness-proportional selection to tournament selection. The fitness-proportional selection must make two passes over the population to assign each individual its probability of being chosen by the genetic operators. It also includes the sorting of all individuals in the population according to fitness in order to select individuals probabilistically according to their fitness. Contrast this with tournament selection: only one pass is necessary in order to assign fitness to the individuals in the population. There is also no sorting involved, only a linear search for the maximum fitness value.

Most systems allow that an individual is selected more than one time, called *reselection*. This helps highly fit individuals to have a larger impact on the population because they are more likely to be chosen more often by the genetic operators than less fit individuals.

## 2.7 Genetic Operators

In Genetic Programming, the operations performing the search are the *genetic operators*. They operate on the individuals to produce new and hopefully more fit individuals. The three most common genetic operators are *reproduction* (copying one individual from one generation to the next), *crossover* (a form of sexual recombination of individuals), and *mutation*. Other genetic operators exist but are not used as frequently as reproduction, crossover and mutation, usually because other operators are variations of either reproduction, crossover or mutation.

### 2.7.1 Crossover

The crossover operator tries to combine fit individuals to hopefully produce more fit children in an effort to take steps closer to the solution. Of the three most common genetic operators, it is the operator that is responsible for creating most of the new individuals in the population. It works as follows: two parents are chosen based on their fitness using a selection method (see section 2.6) and then cloned to produce two children. For each child, a *crossover point*, that is, a node in the tree, is then chosen. Function nodes have a much higher probability of being chosen as the crossover point compared to terminal nodes, see section 2.8. Each child then switches subtrees at their crossover points. Finally, the children are inserted into the next generation of the population. The procedure is shown in figure 2.2. If the size of one or both of the children exceeds the maximum allowed size after the operation, the child (or children) with a size exceeding the maximum will be inserted unchanged into the next generation.

The crossover operation is a very destructive operator. [Nordin et al., 1995] measured the difference in fitness between the parents and the children in the crossover operation and found that the fitness of the children were lower than the fitness of the parent in a majority of the crossover events. The reason that crossover contributes positively to the evolutionary forces of genetic programming despite its destructive powers is probably due to the fact that the small

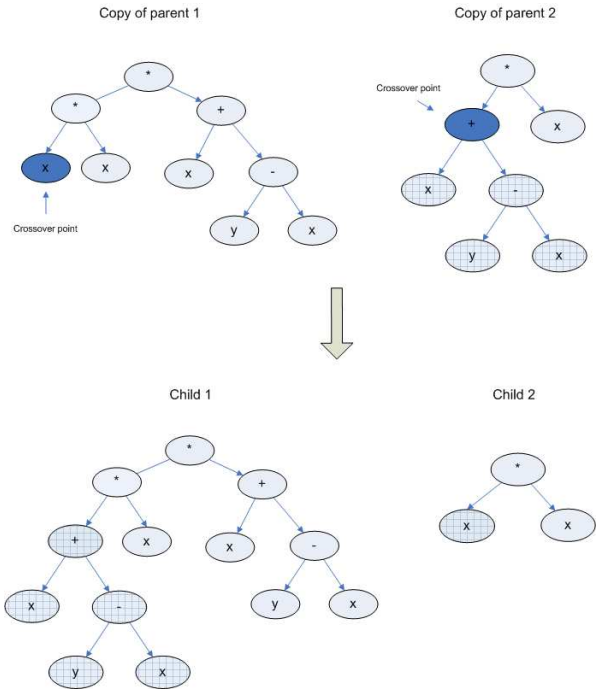



Figure 2.2: The crossover operation.

number of crossover events that produce children with higher fitness than their parents is enough to drive the evolutionary process forward. There should however be room for improvement on the crossover operator to reduce its destructive effects and thereby possibly make the evolutionary process more efficient. [Banzhaf et al., 1998] mentions *brood recombination* that creates more than two offspring by repeating the crossover operation on two selected individuals. The resulting offspring are then compared to each other and only the two fittest offspring are inserted into the new population.

### 2.7.2 Mutation

The mutation operator replaces random subtrees of individuals with new subtrees to emulate real world mutation of genes. The process is as follows: after an individual has been selected from the population based on its fitness by some selection method (see section 2.6), a mutation point is randomly selected. The chosen subtree is then removed and replaced by a new subtree. [Banzhaf et al., 1998] lists different ways to create the new subtree. Among others are to randomly generate a subtree from the function set and terminal set, or to pick a random terminal from the terminal set and insert it as the new subtree.

On a problem solving level, the mutation operator is included to prevent that the search gets stuck in local optima, but it is used sparingly according to [Koza et al., 1999]. He claims that the mutation operator is a special case of the crossover operator since the only difference between the two operators on the individual level is where the new subtree comes from. A randomly created subtree can be seen as belonging to an individual from the initial generation. A randomly chosen subtree in the crossover operation comes from a fitness-selected individual in generation $i$. He advocates the use of a randomly chosen subtree from a fitness-selected individual in generation $i$ over the use of a randomly chosen subtree from a randomly selected individual from the initial generation. On this basis the mutation operator is not used often. It should be noted that this reasoning is applied to Koza's version of Genetic Programming.

| Parameter | Default value |
|---|---|
| Population size (M) | 500 |
| Maximum generations (G) | 51 |

Figure 2.3: Major parameters.

| Parameter | Default value |
|---|---|
| Crossover probability | 0.9 |
| Reproduction probability | 0.1 |
| Mutation probability | 0 |
| Internal node probability | 0.9 |
| External node probability | 0.1 |
| Maximum depth | 17 |
| Maximum inital depth | 6 |

Figure 2.4: Minor parameters.

However, there are other forms of mutation that can be performed on individuals. [Engelbrecht, 2007] lists for example

- **function node mutation** where a function node is randomly selected and the function is replaced with a randomly selected function from the function set with the same arity.
- **terminal node mutation** where a terminal node is randomly selected and the terminal is replaced with a randomly selected terminal from the terminal set.
- **swap mutation** where a function node is randomly selected and its arguments are swapped.

### 2.7.3 Reproduction

The reproduction operator ensures that highly fit individuals are retained from generation to generation. This is in accordance with Darwin's survival of the fittest. If the highly fit individuals are not retained from generation to generation it can slow down the evolutionary process by not taking advantage of the progress that has been made so far. The reproduction operator starts with the selection of an individual based on its fitness by some selection method (see section 2.6). Then the selected individual is copied and inserted into the next generation.

## 2.8 Preparatory steps

Before genetic programming can be applied to a problem the parameters of the system must be set. [Koza, 1992] names three categories of parameters: major parameters; minor parameters; and qualitative parameters that determine the execution of the Genetic Programming system. They can be seen in table 2.3, table 2.4, and table 2.5 respectively. The two major parameters control the size of the population ($M$), and the maximum number of generations ($G$). Default value is 50 generations plus the initial randomly generated population. The minor parameters include the probabilities of genetic operators and node selection, and size constraints of crossover[4]. The last six qualitative parameters specifies how a certain application of the genetic programming system should be executed. The first specifies which generative method to be used, default is ramped half-and-half described below; the second and third parameters specify the method of how individuals are selected from the population for reproduction and crossover operations; the fourth indicates if an additional fitness measure is used; the fifth indicates if *over-population* is used. Over-population

[4]The number of nodes can also be a size constraint due to the branching factor of functions with many arguments.

| Parameter | Default value |
|---|---|
| Generative method | Ramped half-and-half |
| Selection method, first parent | Tournament selection |
| Selection method, second parent | Tournament selection |
| Additional fitness measure | No |
| Over-population | No |
| Elitist strategy | No |

Figure 2.5: Qualitative parameters.

involves creating more individuals than the population size. This new population of individuals is then reduced to the maximum population size by removing individuals with lower fitness from the population; the sixth parameter indicates if the *elitist* strategy is used. The individual with highest fitness of each generation is always copied into the next generation if the elitist strategy is used. The default values of all the parameters should not be attributed any importance according to [Koza et al.]. According to them, this is to reflect that the strength of genetic programming does not come from choosing good parameters when applying genetic programming to different problems but from the evolutionary process itself. Of course, if the values differ considerably from the default values it can be expected that the result of applying Genetic Programming to a problem will be affected in either positive or negative ways. An example of this can be seen in section 5.4 where the crossover operation is applied to 50 % of the individuals in the population, resulting in a clear degradation of the number of iterations before the termination criteria has been met compared to the reference experiments that applied the crossover operation to 90 % of the individuals in the population.

After the parameters have been set, the population must be initialized. The standard approach is to create the initial population by generating random programs using the *ramped half-and-half method*. This means that the population is logically divided into equally large parts where each part contains programs of a certain maximum depth. These parts contain equally many programs generated using the *full method* as programs generated using the *grow method*. The full method generates perfect trees of a specified depth[5]. Initially, a function is selected from the function set and is designated to be the root of the tree. Then, for each argument of the chosen function, a function is randomly chosen from the function set and added to the program. This is repeated until the specified depth is reached. Finally, only terminals are chosen as arguments to the last layer of functions. The grow method generates programs by randomly selecting from both the function set and the terminal set (after a function has been chosen to be the root) until the specified depth is reached, then only terminals are picked. If a function is picked, then another random selection from the terminal set and the function set is made for each argument of the function. Logically, this approach to creating the initial population leads to a greater structural diversity among the individuals in the initial population compared to just randomly creating individuals for the initial population. I base this on the fact that the probability of randomly generating a perfect tree is smaller than the probability of randomly generating a non-perfect tree. This fact becomes even more evident when the depth of the perfect tree increases. A greater structural diversity of the initial population is most likely beneficial for the evolutionary process because the structure of the solution is usually unknown beforehand. By supplying the Genetic Programming system with as much material as possible a solution is more likely to be found.

[5] A perfect tree has all terminal nodes on the same level and all function nodes in the tree have the same number of child nodes.

## 2.9 Termination criterion

To ensure the termination of the algorithm, the user has to specify a *termination criterion*. Several criteria can be used, for example, elapsed time since the execution of the Genetic Programming system started, or number of executed operations. [Koza, 1992] suggests two critera for termination: a maximum number of generations, and a success condition. If either of these two conditions are met, the execution of the Genetic Programming system is terminated. The maximum number of generations is one of the two major parameters specified before the system is started (see section 2.8). The success condition is problem specific and often involves finding a 100%-correct solution to the problem. For problems where one doesn't expect an exact solution, a less exact criterion for success can be used. An example of the latter case can be to create a mathematical model of a noisy environment [Koza, 1992]. There are however problems where an exact solution is unknown. One possible termination criteria can be that no measurable fitness improvements have been made over a specified number of generations.

## 2.10 Applications of Genetic Programming

Genetic programming is a domain-independent problem solving method which has been shown by its successful applications in many different areas. [Xie et al., 2006] examined how Genetic Programming can be used to detect rhythmic stress in spoken English. Rhythmic stress differs from lexical stress by looking at more than single isolated words. Detecting stress is an important aspect of creating a system for automated language learning in order to give feedback to the student and improving the student's learning. They measured prosodic features (the relative duration of syllables, the amplitude of syllables, the pitch of syllables), as well as the quality of the vowels in the syllables. They showed that by using Genetic Programming they could construct a more effective automatic stress detector than decision trees created using the C4.5 algorithm and support vector machines. According to them Genetic Programming showed to have stronger feature selection ability, meaning that Genetic Programming is more suited for finding the duration, amplitude, and pitch of syllables than the other two methods.

The design of navigational controllers for unmanned aerial vehicles was successfully evolved in [Barlow, 2004]. The task was to let an unmanned aerial vehicle locate a radar source, navigate to the radar source using only on-board sensors, and then circle around the radar source. He used multi-objective genetic programming with four different fitness functions derived from flight simulations to solve the task. The first fitness function, using normalized distance to the radar source, was used to fly to the radar source as fast as possible. The second fitness function was used to enable the unmanned aerial vehicle to circle around the radar source. The third and fourth fitness functions were used to make the vehicle fly as efficient and safe as possible. The multi-objective genetic programming algorithm that he used is similar to NSGA-II [Deb et al., 2000]. The experiments were successful on different sets of radar sources: stationary and continuously emitting, stationary and intermittently emitting, and mobile and continuously emitting.

[Azaria and Sipper, 2005] used Genetic Programming for evolving game playing strategies for Backgammon. They divided each individual into two distinct programs. One program is called during the *contact* stage of the game. The contact stage is when players can hit each other. The other program is called during the *race* stage of the game. The race stage is the part of the game where no contact exists between the players. The individuals are evaluated by playing against each other in a tournament form. As benchmark they used *Pubeval*, a widely used program for measuring the ability of Backgammon playing programs. The result was a program that won 56.8% of the games played against Pubeval, making it a very strong player in human terms.

Finally, [genetic-programming.com] lists 36 human-competitive results produced by genetic programming. Most notably are two instances where genetic programming has created a patentable new invention. The 36 instances come from "the fields of computational molecular biology, cellular automata, sorting networks, and the synthesis of the design of both the topology and component sizing for complex structures, such as analog electrical circuits, controllers, and antenna." The site

claims that "genetic programming can be used as an automated invention machine to create new and useful patentable inventions."

# Chapter 3

# Modularization

## 3.1 Introduction

There are two main problems in standard Genetic Programming according to [Banzhaf et al., 1998], namely scalability and efficiency. One approach to solving these issues is to define and use modules inside programs. There are several different approaches on how modules should look and how they should be used. Some techniques have modules that are local to an individual (Automatically Defined Functions in section 3.4), while other approaches have global modules (Module Acquisition in section 3.3 and Adaptive Representation section in 3.7). Some techniques evolve the modules after they have been defined (section 3.4), and other techniques see a module as a static entity that never changes (sections 3.3 and 3.7). Some approaches can be seen as introducing new genetic operators, for example Module Acquisition in section 3.3. Others are applied after the genetic operators (section 3.7). All techniques have one thing in common however, and that is they try to minimize the destructive effects of the genetic operators, specifically crossover, which in 75% of the cases has a 50% loss of fitness from parent to child [Banzhaf et al., 1998].

The idea of using modules to help with scalability of problems comes in part from the schema theorem of [Holland, 1975]. The schema theorem is briefly explained because it is outside the scope of the thesis to go into the details of the theoretical foundation. The schema theorem which applies to Genetic Algorithms with fixed length strings states that certain problem specific strings contains substrings that are beneficial for the evolutionary process and will multiply exponentially in the population. These strings are good building blocks which are combined to form other strings that constitutes new good building blocks. This process is repeated until hopefully a solution has been found. Individuals with high fitness are more likely to include good building blocks and therefore spread these good building blocks throughout the population. [Koza, 1992] claims that the schema theorem can be applied to variable length Genetic Programming as well, making it a good foundation for using modules in Genetic Programming.

As a final introductory note, it is worth mentioning that no new modularization methods have surfaced after 1998 and no explanations why this is the case has been found. One possible explanation is that better ways of dealing with scalability has been discovered and therefore made modularization methods obsolete. Another explanation is that modularization is not an effective way of dealing with the issue of scalability in Genetic Programming.

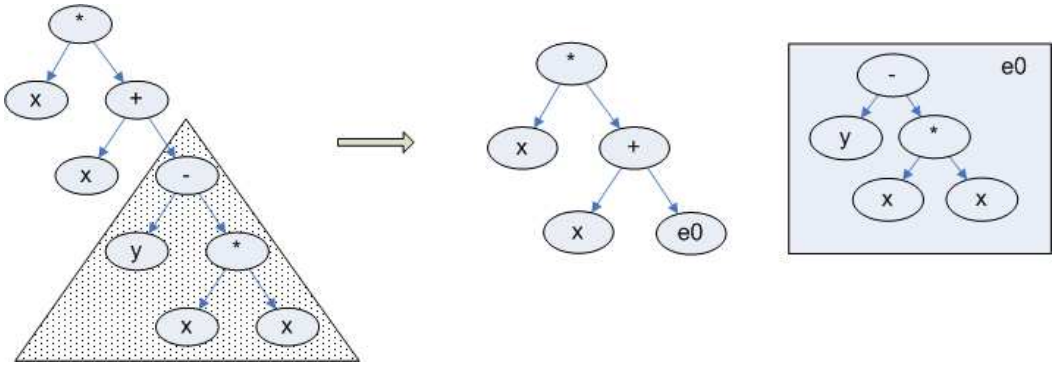



Figure 3.1: The encapsulation operation. x and y are arguments, e0 is a module created by the encapsulation operation.

## 3.2 Encapsulation

[Koza, 1992] suggests a basic modularization idea. An individual in the population is chosen by some selection method based on its fitness and then copied. Then a non-terminal subtree from the copy is picked to be a module. The subtree is replaced by a terminal placeholder and then saved so that other individuals can use the placeholder and refer to the subtree, see figure 3.1. This means that the entire subtree is chosen and no arguments can be given to the encapsulated function. Once the subtree has been removed it can not evolve further. However, the original individual that is selected for the encapsulation operation is unaltered and can be involved in other genetic operators.

Encapsulation is a technique that have global and static modules and can be applied to any tree or linearly structured program.

## 3.3 Module Acquisition

[Angeline and Pollack, 1992] proposed the idea of Module Acquisition, an idea similar to encapsulation. First, an individual is selected from the population based on its fitness. Then, a subtree is randomly chosen to be the basis of the new module. The new module is created by including all nodes up to a certain specified *trim depth* starting from the root of the subtree. The remaining nodes of the subtree are cut off and considered to be arguments to the module. The part of the subtree that constitutes the module is now removed and replaced by a call to the new module. The body of the module is stored outside the tree to be invoked and evaluated when the individual is evaluated. This operation, called *compression*, is illustrated in figure 3.2. To prevent that this process will lead to local minima an inverse function to the compress function, called *expansion*, is introduced. Both compression and expansion are genetic operators that are chosen with a given probability, thus decreasing the impact of other genetic operators.

As with encapsulation, Module Acquisition use static modules that can be used in a global environment.

## 3.4 Automatically Defined Functions

Another approach, also by [Koza, 1994b], is called Automatically Defined Functions (ADFs). This modularization method defines a number of functions local to an individual that evolve alongside the individual. Since an ADF is local to an individual it cannot be called by other individuals thereby having no impact on the rest of the population. The ADFs are included in the function set of the individuals with each ADF having its own function set and terminal set.

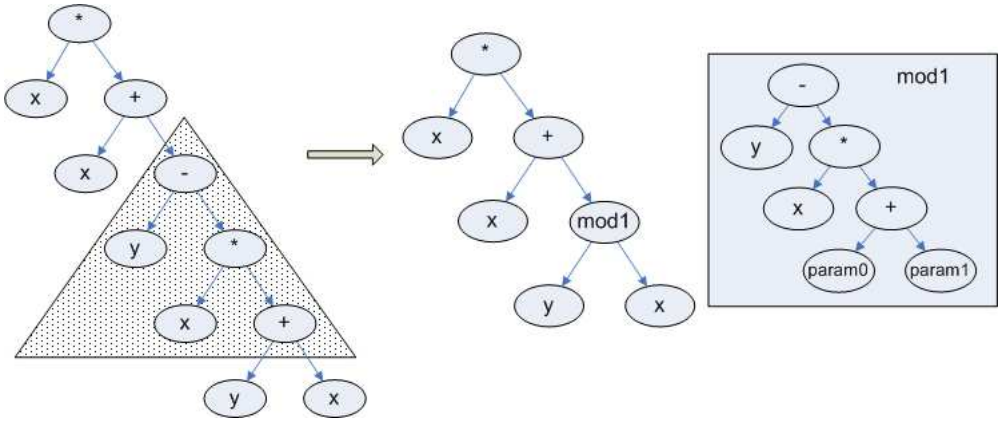



Figure 3.2: The compression operator in Module Acquisition. X and y are arguments, mod1 is a module created by the compression operator, param0 and param1 are parameters to the module.

Programs that use ADFs have a very strict syntactic structure that require a special form of crossover that only operates on certain branches in the tree. The structure of the tree used with ADFs can be seen in figure 3.3. The left branch of the tree, called the *function-defining branch*, contains the functions that can be invoked. The leftmost child of the function node is the name of the function, the middle child contains a list of the the arguments of the function, and the rightmost child is the body of the function. The right branch, called the *result-producing branch*, contains the program that computes the solution. Only the body of the ADF and the result-producing branch, the nodes below the line in figure 3.3 (the variant parts), are active in the evolution of the program.

Because of the strict syntactic structure of programs that use ADFs it is necessary to modify the crossover operation to work on individuals with ADFs. To preserve the function and terminals sets of each branch of the individual during the crossover operation it is constrained to only work on branches with the same function and terminal sets. Furthermore, only ADFs with the same name can exchange subtrees. This means that if a node in, for example, adf0 is chosen for the crossover operation in the first parent only the nodes in adf0 of the second parent are eligible as possible crossover nodes. The crossover operation on individuals with ADFs has one side effect not found in regular crossover. First, let me restate that all references to ADFs are local to the individual. All nodes referencing ADFs that are part of the subtree in the crossover operation will after the crossover operation point to a ADF in the new individual. This has the effect of not only changing the subtree in the crossover operation but it also adds references to local ADFs, if they are present in the subtree. This can lead to significant changes in the individual from one generation to the next. An example of the process can be found in figure 3.4. The crossover nodes are 5 and 15 from parent0 and parent1 respectively. After the crossover the adf0 in child1 no longer points to node 20 (the copy of node 0) but points to node 29 in child1 instead. This has resulted in child1 having two references to adf0 compared to only one before the crossover operation.

In addition to the restriction on the possible crossover nodes in crossover operation, another restriction on the genetic operators comes from the fact that the function and terminal sets for the two different branches can be different. To avoid, for example, infinite loops, only parts from the same type of branch can be combined, i.e. only subtrees from the result-producing branch can be used in crossover with another result-producing branch from another individual, and similarly only function-defining branches from different individuals can be combined. This is achieved through *branch typing*, where each ADF, as well as the main program, is assigned a unique type.

When ADFs are used, apart from the parameters that need to be specified in basic Genetic Programming, one need to decide the number of ADFs and how many arguments each ADF will have, as well as the function and terminal sets for each ADF. When defining the function set for

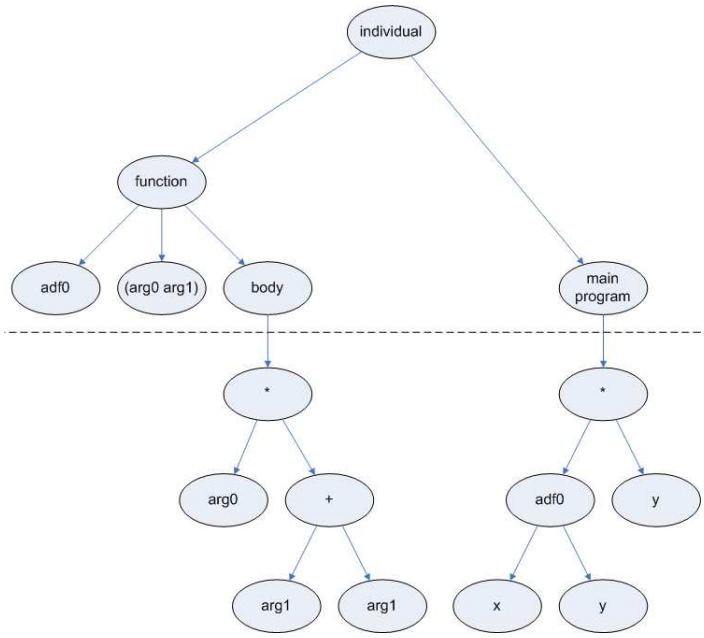



Figure 3.3: The structure of an individual with one Automatically Defined Function. This structure is mostly applicable to Lisp. The dotted line separates the invariant part (top half) from the variant part (lower half)

an ADF other ADFs can be included to form a hierarchy of ADF invocations. This can result in infinite loops if, for example, adf0's function set includes adf1 and adf1's function set includes adf0.

Compared to Module Acquisition (section 3.3) ADFs performed better on the even-4-parity problem [Kinnear, Jr., 1994]. He attributes this to the structural regularity of the ADFs, which is introduced when declaring the function sets and terminal sets, as well as the large number of function calls to the ADFs. When using ADFs there is always an implicit problem specific bias on the part of the designer when defining the function sets and terminal sets for the ADFs. This bias, however difficult it is to measure, should also be a factor when comparing different modularization methods.

## 3.5 Automatically Defined Functions with architecture-altering operations

One drawback with ADFs is that the number of ADFs and the number of arguments for each ADF has to be specified beforehand. This can be difficult for certain problems where it is hard to determine in advance how many functions the problem will need to be solved. [Koza, 1995] proposed six architecture-altering operations to help evolve the architecture of the solution in terms of number of ADFs and their arguments at the same time as the solution is evolved. In each of these operations the individual is chosen based on fitness.

- *Branch duplication* picks one ADF at random from a selected individual and makes a copy of that ADF. For each invocation of the selected ADF in other ADFs, if present, and in the main program, a random choice is made whether to replace the invocation of the original ADF with an invocation of the duplicated ADF. Since the second ADF is a copy of the original the result of the main program does not change.

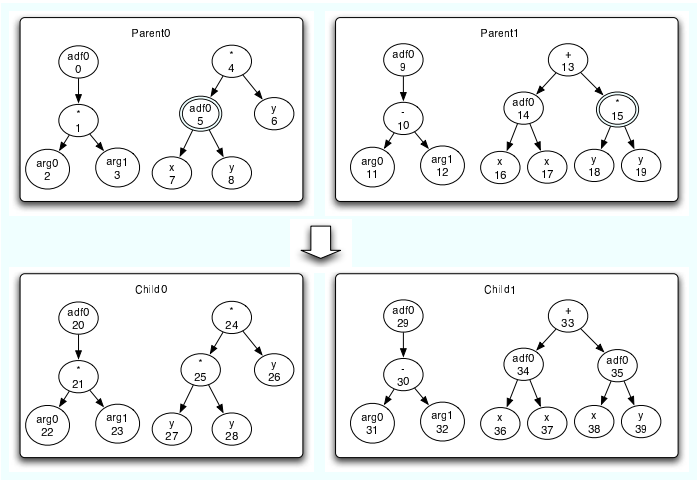



Figure 3.4: The crossover operation on individuals with Automatically Defined Functions. The crossover points are marked with a circle. Only the variant parts are shown.

This operation is used to increase the number of ADFs in selected individuals. The result of the new program with the added function is identical to the old program until either the original ADF or the newly created function is involved in the crossover operation. The crossover operation then changes the function and thereby the result of the main program to hopefully being a step closer to the solution.

- *Branch deletion* removes one ADF from a selected individual. One ADF is picked for removal[1]. After the picked ADF has been removed, each subtree $s_{old}$ starting at an invocation of the picked ADF in other ADFs and the main program is replaced with another subtree $s_{new}$. There are three methods for determining which subtree to insert:
  - use $s_{old}$ but replace the invocation of the removed ADF with an invocation of an existing ADF (*branch deletion by consolidation*),
  - randomly generate a new subtree (*branch deletion with random regeneration*), or
  - use the body of the removed ADF (*branch deletion with macro expansion*). This is similar to the expansion operator in Module Acquisition.

  The branch deletion operation is used to reduce the number of ADFs, thereby making it the opposite of the branch duplication operation and the branch creation operation.

- *Branch creation* creates a new ADF for an individual. The body of the new ADF is created by removing a part of a subtree or an entire subtree from an existing ADF or the main program of the individual. The removed subtree will be the body of the new ADF and a call to the new ADF is inserted in lieu of the removed subtree. The arguments to the new ADF will be all subtrees below the root of the chosen subtree that were not included as the body of the ADF.

  The branch creation operation together with the branch duplication and branch deletion operations are included to change the number of ADFs in the selected individuals.

- *Argument duplication* duplicates an argument in a chosen ADF by adding a new argument node to the chosen ADF with the same argument as the argument being picked for duplication. For each occurence of the original argument in the body of the picked ADF, a choice is made whether or not to replace it with an occurence of the new argument. For each

[1] Koza doesn't state how the ADF should be picked but it is not unreasonable to assume it is picked at random.

invocation of the selected ADF, the subtree corresponding to the original argument is copied and inserted as the argument to the newly created argument.

- *Argument deletion* removes an argument from a picked ADF. For each invocation of the picked ADF, the subtree corresponding to the argument-to-be-deleted is removed. Each occurence of the argument-to-be-deleted in the picked ADF is replaced with another subtree. As with branch deletion, there are three methods for choosing a suitable subtree:
  - use another argument to the picked ADF (*argument deletion by consolidation*),
  - randomly generate a new subtree (*argument deletion with random regeneration*),
  - use *argument deletion by macro expansion*. This involves replacing each invocation of the ADF in other ADFs and the main program with a copy of the ADF. The argument-to-be-deleted is replaced by the subtree corresponding to the argument in each copy.
- *Argument creation* adds an argument to a chosen ADF and replaces a picked subtree in the ADF with a uniquely named argument and for each invocation of the selected ADF the picked subtree is added as an argument.

When crossover takes place in an architecturally diverse population care must be taken to ensure that the crossover operation produces valid offspring. Valid offspring means that the terminal set and function set are not extended after a crossover operation. Therefore *point typing* is needed. Point typing assigns a type to each node in the main program and to each node in each ADF. The type is based on the function and terminal sets of the subtree of the node, as well as the function set and terminal set of the branch where the node is located. Point typing has three general principles: First, the function sets and terminal sets of the branches involved in crossover must be the same. Second, the number of arguments of each function must be the same in both subtrees. Third, all other problem-specific syntactic rules of construction must be satisfied.

Because of the syntactic differences in the population, the crossover operation has to be modified. As before, two individuals are chosen to take part in the crossover operation. Now, one individual is designated the contributing parent while the other individual is designated the receiving parent. The crossover point in the receiving parent must be chosen so that it has the same type as the crossover point from the contributing parent. This new crossover operation produces only one child with the architecture of the receiving parent but with a subtree from the contributing parent.

Comparing ADFs with architecture-altering operations with ADFs and standard Genetic Programming [Koza et al., 1999] showed that the ADFs with architecture-altering operations were computationally more intensive than ADFs but computationally more efficient compared to standard Genetic Programming. Looking at the time consumption, the standard Genetic Programming takes longest time to find a solution, ADF takes the least amount of time to complete, while ADFs with architecture-altering operations falls between the other two. However, the time measurement probably does not include setup time for the different approaches, which also should be a factor. The benefit of ADFs with architecture-altering operations was in the structural complexity, that is, in the number of nodes in the solution, including nodes in ADFs. It outperformed both standard Genetic Programming and ADFs.

## 3.6 Automatically Defined Macros

A technique similar to ADFs is automatically defined macros (ADM) by [Spector, 1996]. Instead of using functions, as Koza did, Spector uses macros, especially substitution macros, when evolving the programs. A macro is a placeholder where the body of the placeholder replaces the call to the macro before the program is compiled and run. In Lisp, macros can be used to implement control structures, e.g. if-statements, that can not be produced by functions because arguments to functions are evaluated before the function is called.

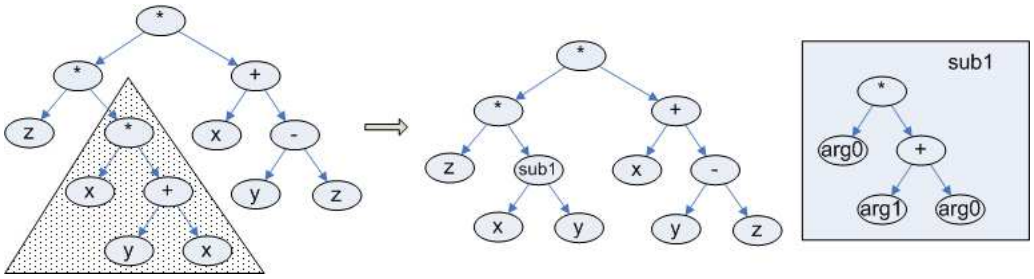



Figure 3.5: Block generalization in Adaptive representation

A comparison between ADMs and ADFs showed that in an experiment with a robot mopping the floor in a room with obstacles, the performance of ADMs were better than that of iterations. On the other hand, on the Lawn-Mower problem, the performance was roughly similar when ADMs were used compared to using ADFs. However, the number of individuals needed when ADMs were used were greater then when ADFs were used. According to Spector, ADMs are most useful in environments that include operators that work by side effect or are sensitive to their calling environments. Spector also says that ADMs and ADF can be used simultaneously to evolve both controll structures using ADMs and functions using ADF.

## 3.7 Adaptive representation

The idea of Adaptive Representation (AR) is due to [Rosca and Ballard, 1994a]. Their idea involves global modules that do not evolve after they have been created. They propose to introduce new functions into the function set by discovering what they call *candidate building blocks* in a bottom-up fashion. The discovery of candidate building blocks is done either by looking at the fitness of subtrees, using some form of fitness function, or by looking at the frequency of subtrees. They suggest three methods for evaluating the fitness of subtrees:

- use the program fitness function
- use a slightly modified version of the program fitness function. For example, the block fitness can be measured on a reduced set of fitness cases, dependent on the variables used in the block
- domain dependent block fitness functions based on supplementary domain knowledge

Subtrees with a high fitness are considered candidate building blocks, which in turn is the basis for choosing functions to include in the function set. A high frequency of a subtree does not imply that the subtree is a good candidate building block, but used together with the fitness of a subtree, the method could be useful for choosing candidate building blocks.

The function set is dynamically extended by a subset of the functions in the set of candidate building blocks. Each candidate building block is given a unique function name. These chosen candidate building blocks are generalized by replacing the terminals in the building blocks with terminals local to the function, see figure 3.5. Once the function set has been extended, individuals with high fitness will be retained while low-fitness individuals are discarded and replaced by new individuals, created from the new function set. This process marks the beginning of a new *epoch*.

## 3.8 Adaptive representation through learning

An extension to and modification of Adaptive Representation is proposed by [Rosca and Ballard, 1996], called Adaptive representation through learning (ARL). New subroutines are introduced by looking at the *differential fitness* and *block activation*. Differential fitness is the difference in fitness

between the least fit parent and the children. The search for blocks are done in children with high differential fitness, under the assumption that children with high differential fitness contain useful code that can be made into subroutines. Block activation is the number of times the root node of a subtree has been evaluated during repeated executions of the program, for example on the fitness cases. Subtrees with high block activation and small height are chosen to be candidate building blocks[2]. The candidate building blocks are then generalized by replacing a subset of the terminals in the building blocks with local variables. Finally, each candidate building block is given a unique name and placed in a global function set. The subroutines are created when the population diversity has decreased under a long time. This is measured by population *entropy*.

Each subroutine has a *utility value* that is used to keep the number of subroutines below a specified number by removing subroutines with low utility value when new subroutines are created. The utility value is the average fitness of all individuals invoking the subroutine accumulated over the past $W$ generations.

## 3.9 Summary

These seven modularization methods can be divided into two different categories, as seen in table 3.1. The first category includes modularization methods that evolve modules after they have been created. The modules in the first category are local to an individual thereby limiting their direct impact on the population as a whole. The second category includes methods that don't evolve the modules after creation but the modules can be used by all individuals in the population.

| | | Static | Evolving | Global | Local |
|---|---|---|---|---|---|
| Category 1 | ADF | | x | | x |
| | ADF aao | | x | | x |
| | ADM | | x | | x |
| Category 2 | Encap. | x | | x | |
| | MA | x | | x | |
| | AR | x | | x | |
| | ARL | x | | x | |

Table 3.1: Summary of modularization methods

The modularization methods in the first category are Automatically defined functions (ADF) which evolves a program alongside a given set of functions; Automatically defined functions with architecture-altering operations (ADF aao) evolves a program as well the number of functions, the number of arguments of each ADF, and the content of each function; Automatically defined macros evolves a program and a macro similar to ADF.

The methods in the second category are Encapsulation (Encap.) that creates a static, global function from an entire subtree; Module acquisition (MA) also makes a static, global function with a certain depth from a random subtree. Module Acquisition also reintroduces functions into the population with a expansion operator; Adaptive representation (AR) creates new functions based on the fitness and frequency of subtrees; Adaptive representation through learning (ARL) creates functions from frequently evaluated subtrees in children that have better fitness than their parents.

[2]The height ranges from 3 to 5.

# Chapter 4

# Structural Module Creation

## 4.1 Introduction

I propose a modularization method based on Module Acquisition, called *Structural Module Creation*. Structural Module Creation creates new modules based on the most frequently occurring subtree structures in the population instead of randomly selecting a subtree as basis for a module. I chose to extend Module Acquisition because of its appealing simplicity compared to the other modularization methods. After analyzing the subtree structures for the even-k-parity problem I found that some structures appeared more often than others in the individuals. This gave me an indication that looking at the most frequent subtree structures of individuals might be a possible way of choosing subtrees suitable for modules instead of choosing subtrees at random. There is no guarantee that a structure that is frequent among the individuals in the population is a good choice for a module but in any case replacing a frequent subtree structure for a module reduces the number of nodes in the individual, and increases the function set and thereby the expressiveness of the system. If a subtree structure is frequent it is likely that later generations will contain a solution with a frequently occurring subtree structure. However, by removing frequently occurring subtrees and replacing them with static functions there is a risk that the genetic variation is reduced. This can be remedied by not replacing all occurrences of the subtree, thus leaving the genetic material in the population for further evolution.

## 4.2 Method

During the evaluation of individuals, an encoding of a node and its children down to a certain depth is stored with the node. An example of an encoded subtree can be seen in figure 4.1. A frequency table is then built up containing each encoding that exists in a selected number of individuals from the population[1]. Each encoding, in order of descending frequency, is then used to create new modules. A certain percentage of the population is modified by replacing, with a specified probability, each node that has the corresponding encoding with the module in a bottom-up fashion. A node is replaced with a module only if the structure has not changed since the node was encoded during the evaluation process. This means that a node with the most frequent structure aab will not be modified if the first child has the same structure aab. The first child will be replaced with a module resulting in a new encoding of the parent node, amod0b, where mod0 is the name of the newly created module. This new structure is not represented in the frequency table for this generation and is therefore not eligible for module creation. There is no re-encoding of nodes because the encoding is only used at the beginning of each generation in the building of the subtree frequency table. If nodes would be re-encoded and the new encoding be used for creating new modules, then the frequency table would need to be rebuilt after each module creation to account for the changes in the subtree frequency.

[1] Either the entire population can be selected or a percentage of the population

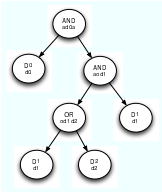



Figure 4.1: A depth-1 encoded subtree.

In Module Acquisition, the compression operator is chosen as the genetic operator alongside the crossover operator and the reproduction operator. In Structural Module Creation, the compression operator operates on the selected portion of the population before the other two genetic operators are involved, leaving a smaller portion of the population to the other two genetic population. The reason for this is to preserve the encoding of the nodes. If a crossover operation was performed on an individual then the nodes of that individual would need to be re-encoded and the subtree frequency table would need to be rebuilt.

The expansion operator in Module Acquisition is not included in Structural Module Creation. Since only the most frequently occurring subtrees are used for making modules, and not all subtrees with the same structure are replaced with modules[2], the genetic material comprising the bodies of the modules are still existing in the individuals in the population, removing the immediate need for the expansion operator.

When comparing Structural Module Creation to the modularization methods as in section 3.9 it falls into category 2. A summary of the modularization methods can be seen in table 3.1. Like Module Acquisition, Structural Module Creation creates static modules that are accessible to every individual in the population.

There are some similarities between Structural Module Creation and Adaptive Representation (section 3.7). For example, both look at the frequency of subtrees when trying to find a suitable subtree for modules, and both use global modules that extend the function set and thereby increases the expressiveness. But there are also some differences between the two methods. One big difference is that Adaptive Representation creates new individuals from the extended function set at the beginning of a new epoch while Structural Module Creation only modifies existing individuals and never creates new individuals after the initial population has been created. Another difference is that Adaptive Representation generalizes the arguments of the modules while Structural Module Creation leaves the arguments untouched.

Experiments comparing Structural Module Creation with Module Acquisition and standard Genetic Programming can be found in chapter 5.

## 4.3 Algorithmic complexity

The added computational complexity of using Structural Module Creation with Genetic Programming can calculated by looking at the following parts:

- node encoding
- frequency table sorting
- creating modules

The encoding of the nodes of the individuals in the population is $O(n)$ where $n$ is the total number of nodes of the individuals in the population. This can be done during the fitness evaluation phase.

[2]There is a parameter specifying the probability of replacing a subtree with a module.

The sorting of the frequency table can be done in $O(flogf)$ where $f$ is the number of encodings in the frequency table.

Creating modules is done by finding the individuals that contain the most frequent encodings. By storing the individuals that contain a certain encoding together with the frequency of the encoding the process of finding the individuals that contain a certain encoding is $O(1)$. Finding a node with a certain encoding in an individual entails traversing all nodes of the individual which takes $O(n)$ where $n$ is the number of nodes in the individual. This is done for all individuals that are chosen for module creation. This leads to a computational complexity of $O(m)$ where m is the total number of nodes in the individuals chosen for module creation.

The number of encodings is small because of the limited encoding depth (depth 1 is used) and the small function set (four logical operators). The number of encodings grow for each generation because of all the modules that are created each generation but are still probably smaller than the number of nodes in all the individuals that are chosen for module creation. Therefore the added computational complexity of using Structural Module Creation is $O(m)$ over the number of nodes $m$ in the individuals chosen for module creation in each generation.

# Chapter 5

# Experiments

## 5.1 Measurements

There are several things to measure in an execution of the Genetic Programming system. In standard Genetic Programming, the best, worst, and average fitness of each generation is of interest in order to compare different generations.

From a computational perspective, the number of individuals required to find a solution is of interest, as well as the number of generations. [Koza, 1992] introduced the measure of *cumulative probability of success* $P(M, i)$, the probability that an execution[1] of the Genetic Programming system with population size $M$ yields a solution by generation $i$. For each generation $i$ this is simply the total number of executions of the Genetic Programming system that succeeded on or before the $i$th generation, divided by the total number of executions conducted. Based on $P(M, i)$, $I(M, i, z)$ can be calculated, see equation 5.1.

$$I(M, i, z) = M * (i + 1) * \left\lceil \frac{\log(1 - z)}{\log(1 - P(M, i))} \right\rceil \tag{5.1}$$

$I(M, i, z)$ is the number of individuals that must be processed in order to find a solution with probability $z$ by generation $i$ with a population size of M.

If the individuals contains modules, it can be interesting to know the frequency of calls to each module and how many modules are used in each solution. To get a feel for the overhead incurred by each modularization method I include a time measurement in each experiment. The time for each experiment to finish should be considered only as a guide to how big the overhead is for each method, not an absolute value of the incurred overhead.

The experiments were made on standard Genetic Programming, Genetic Programming using Module Acquisition, and Genetic Programming using Structural Module Creation. The plan was to include Genetic Programming using Automatically Defined Functions but because of problems during the implementation phase, Automatically Defined Functions were not included in the experiments.

Several executions of the Genetic Programming system are made on the same problem instance to get an average of the number of individuals and generations it takes to find a solution[2]. The experiments were conducted on a AMD Athlon X2 3800+ with dual core running 64-bit linux with 2 Gb of RAM.

[1] An execution of the Genetic Programming system is defined as the operations from the creation of the first generation until the termination criterion has been met.

[2] Other measurements can also be of interest, for example individual size, computation and memory consumption.

## 5.2 Even-k-parity problem

The Boolean even-k-parity problem of $k$ Boolean arguments returns TRUE if an even number of the $k$ arguments are TRUE, otherwise it returns FALSE. Parity functions are often used to check the accuracy of stored or transmitted data in computers because a change in the value of any one of its arguments toggles the values of the function [Koza, 1994b]. The answer can easily be calculated by just counting the number of TRUE arguments of the $k$ given arguments and checking if they are even, but by using Boolean functions this function can be implemented in hardware instead of software.

The even-k-parity problem was chosen as benchmark problem because it is a common benchmark problem for testing the different modularization methods.

The parameters used for the even-k-parity problems, ($k = 3, 4, 5$), are as follows: the population size is 1000, the maximum number of generations is 51 (the initial generation plus 50 generations), the crossover and reproduction probabilities are 0.9 and 0.1, respectively, for the standard Genetic Programming. For Module Acquisition, the crossover, reproduction, compression, and expansion probabilities are 0.8, 0.1, 0.05, and 0.05, respectively. The trim depth for modules in Module Acquisition lies between 2 and 4. The probability of choosing a function node in the crossover operation is 0.9, and the probability of choosing a terminal node is 0.1 for the same operation. When Structural Module Creation is used as the modularization method, the crossover and reproduction probabilities are the same as standard Genetic Programming but crossover and reproduction only create the remainder of the new population after the extended compression operator has done its work. The parameters specific to Structural Module Creation are: percent of population to modify is 50%, probability of module insertion is 1.0, and number of new modules per generation is 10. The subtree structures included in the frequency table is of depth 1 with at most four arguments. The reason that I chose only subtrees of depth 1 is to reduce the number of arguments to the module[3].

The fitness cases for the even-k-parity problem consists of all $2^k$ permutations of the $k$ arguments with their corresponding correct answer. The success condition for a solution is to correctly classify all fitness cases. The function set consists of the four Boolean functions NAND, NOR, AND, and OR. The terminal set consists of $k$ terminals, named D0, D1, ... , D[$k-1$]. Each instance of the problem has been executed 50 times to find an average of best, worst, and average fitness of each generation.

## 5.3 Reference experiments

Firstly, some reference experiments on standard Genetic Programming and Genetic Programming using Module Acquisition were conducted. The change of fitness over time that can be seen in figures 5.1 and 5.4 shows no improvement by using Module Acquisition over standard Genetic Programming[4]. The average number of generations to find a solution, and the time it took to complete the experiments can be seen in figures 5.2 and 5.3. This shows that using Module Acquisition does not improve the result of the standard Genetic Programming. The results are similar to [Kinnear, Jr., 1994] where he compared Module Acquisition to Automatically Defined Functions on the even-4-parity problem. Although he used a different performance measure, the probability that a given run will will evolve a complete solution to the problem, the results are comparable because they both show similar results to the baseline experiments which is standard Genetic Programming.

The difference in time consumed by each method to finish all 50 executions of the Genetic Programming system differs considerably. When using Module Acquisition the time consumed is almost twice that of standard Genetic Programming. This time difference can be explained by the

[3]Subtrees with depth 1 are subtrees with one root node and one or more child nodes. All child nodes have the root node as its parent.

[4]The graphs were created using [Gnuplot]

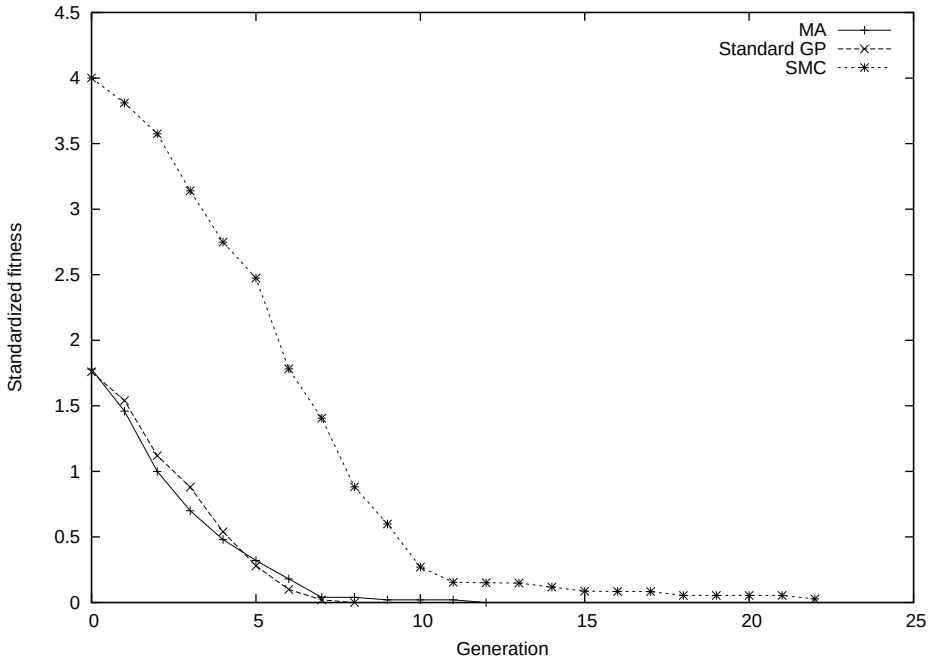



Figure 5.1: Comparing standard Genetic Programming with Module Acquisition and Structural Module Creation on the even-3-parity problem

| Method | Average generations | Time consumed |
|---|---|---|
| Standard | 5 | 9 m 17 s |
| MA | 5 | 16 m 38 s |

Figure 5.2: Even-3-parity problem over 50 iterations.

fact that when a compression or expansion operation takes place an overhead involving changing node references and creating copies of nodes is introduced.

Looking only at these results, it seems that randomly selecting subtrees as modules and letting them propagate through the generations by the virtue of the fitness of the individual containing the module does not seem to be sufficient to improve the effects of Genetic Programming on this problem. Furthermore, by taking a passive role in the distribution of these new modules, that is, by not creating new individuals with the newly created modules as functions, it seems that although a module can be a good building block it is left up to the fitness of the individual if the new module will have a larger impact on the effects of Genetic Programming. A module can be a good building block but if it is contained in an individual with low fitness it has a low probability of being carried on to the next generation.

## 5.4 First Structural Module Creation experiment

The first experiment on the even-3-parity problem with Structural Module Creation proved to be a clear degradation of performance compared to both standard Genetic Programming and Module Acquisition. On average it took 8 generations to find a solution with Structural Module Creation compared to only 5 generations for both standard Genetic Programming and Genetic Programming using Module Acquisition. The time consumed to find a solution was 27 minutes 23 seconds using Structural Module Creation, almost three times that of regular Genetic Programming. I ran

| Method | Average generations | Time consumed |
|---|---|---|
| Standard | 33 | 3 h 32 m |
| MA | 32 | 7 h 26 m |

Figure 5.3: Even-4-parity problem over 50 iterations.

the even-4-parity problem on the same set of parameters. The result was, as in the previous experiment, worse than both standard Genetic Programming and Genetic Programming using Module Acquisition. It took on average 42 generations to find a solution (33 and 32 for standard Genetic Programming and Genetic Programming using Module Acquisition, respectively). The time consumed was 8 hours 10 minutes. The added overhead compared to Module Acquisition can be accounted to the need of Structural Module Creation to create the frequency table on every generation.

An explanation for the large number of generations it took for Structural Module Creation to find a solution can be that too many individuals in the population (50%) was modified. The absence of an inverse to the compression operation in Structural Module Creation could play a role in the poor result. Because such a large portion of the population was used for the compression operator it meant that the population most likely became more homogenous for each generation and reduce the genetic diversity. With the remaining 50% of the population left for the crossover and reproduction operators there was just to little material for the two operations to work with. As mentioned before, the crossover operation is the driving force in the evolutionary process. In effect, by reducing the number of individuals to only half the normal population size it was no surprise to see that the result was poor compared to the reference experiments.

Since such a large percentage of the population was given to the Structural Module Creation compression operator one could expect that they should have had a larger positive impact on the result than they had, if they are to be an improvement to Module Acquisition. But the poor result gives an indication that Structural Module Creation's compression operator will not have a large positive effect on the results.

## 5.5 Fewer modified individuals

I then returned to the even-3-parity-problem and reduced the percent of population to modify to 10% to see if the result improved by letting a larger portion of the population be affected by the crossover and reproduction operators. The change gave an improvement compared to the previous Structural Module Creation experiment on the even-3-parity problem, both in the average number of generations until a solution has been found: 5 generations, the same as standard Genetic Programming and Module Acquisition; and the time consumed: 12 minutes 43 seconds, better than Module Acquisition but worse than standard Genetic Programming. On the even-4-parity problem with the same parameters, the result was not as good: the average number of generations was 39, and the time consumed was 10 hours 41 minutes.

The conclusions one can draw from this experiment is that looking at the entire population for subtree structures may not be a good idea when the problem is scaled up. By reducing the number of individuals that are involved in the counting of subtree frequencies to only the most fit individuals in the population it can be expected that the new modules will be more likely to be better candidates for building blocks and that they are more likely to spread to other individuals in coming generations.

## 5.6 Fewer inserted modules

Being curious if the number of inserted modules with the same encoding could have an impact on the result, I returned to the even-3-parity problem with an even smaller percent of population

to modify (5%) and the probability of module insertion set to 0.1. The result showed a slight improvement in the time consumed but that was to be expected since replacing a node with a module is time consuming and less module insertions were made. No effect on the number of generations were observed.

By reducing the number of inserted modules into individuals to only 10% the modified compression operator lost most of its impact on the population, decreasing the difference in behavior between Structural Module Creation and standard Genetic Programming.

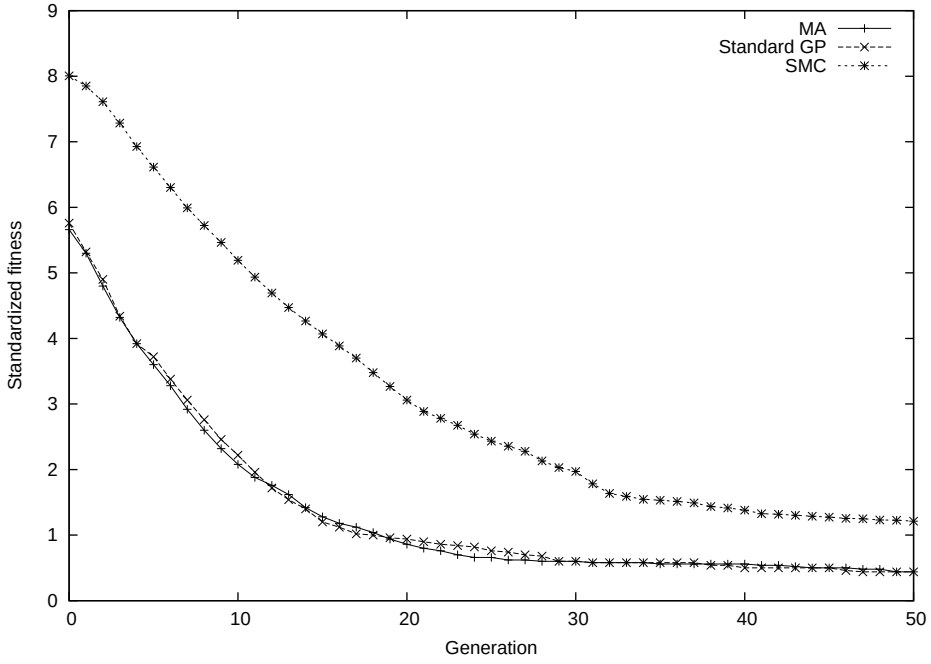



Figure 5.4: Comparing standard Genetic Programming with Module Acquisition and Structural Module Creation on the even-4-parity problem

## 5.7 Module placement

To find out if the placement of the module had any effect on the result, I tried both creating modules in a bottom-up fashion and creating modules with the root chosen from all subtrees with function nodes, using the same parameter values as before. It turned out that there were no difference to speak of between the two methods. There was a small difference in the time consumed but that was negligible.

The lack of difference between the two methods can indicate that although the even-parity-problem can be solved by creating modules in a bottom-up fashion [Rosca and Ballard, 1994b] there is no benefit to gain by using the bottom-up approach on Structural Module Creation.

## 5.8 Modified frequency

Since only looking at the frequency of the subtrees did not improve the performance, I increased the frequency of each subtree occurrence not by 1 but with a value proportional to the raw fitness of the individual holding the module.

Up to this point, the entire population was included in the frequency count. Now, only the percent of the population that was selected for modification was included. The individuals se-

| Method | Average size | Average number of modules |
|---|---|---|
| Standard GP | 118 | - |
| MA | 120 | 0.18 |
| SMC (50%) | 168 | 2.48 |
| SMC (10%) | 131 | 0.22 |

Figure 5.5: The average size and modules of Module Acquisition and Structural Module Creation on the even-3-parity problem.

lected for modification was selected using tournament selection, resulting in a high probability of modifying highly fit individuals, as proposed in 5.5.

To get an indication if this led to an improvement, only three iterations were made with different parameters. All following experiments had a 1.0 probability of module insertion, only the number of new modules per generation and the percent of population to modify was changed. The first experiment modified 5% of the population and created one new module per generation. These parameters together with the changed frequency value showed no improvement over previous experiments. To see if a change in parameter values had an effect on the result, the number of new modules per generation was increased to 2, but no improvements were observed. With only 1 new module per generation but by modifying 10% of the population the result showed also here no improvement.

Had there been more time available it may have been possible to test both frequency proportional to the raw fitness of the individual and counting subtree structural frequency separately to see if one of them had positive impacts on the evolutionary process. It is possible that using both at the same time cancelled out any benefits that one of them had.

## 5.9 Average number of modules

To get an indication of the impact of modules in the solutions, I compared how many modules an average solution had in the reference experiments and the first two experiment on Structural Module Creation, all executed 50 times each. The result can be seen in figures 5.5 and 5.6. On average, each solution contained less than one module in the even-3-parity problem when Module Acquisition and Structural Module Creation with 10% modified population was used. A larger number of modules were present in the solutions when Structural Module Creation with 50% modified population was used but that did not improve the overall performance. On the even-4-parity problem, solutions created with Module Acquisition contained, as before, less than one module while solutions created with Structural Module Creation contained more modules. Even though solutions created with Structural Module Creation contained more modules than their counterparts created with Module Acquisition, it did not lead to any improvements.

Structural Module Creation and Module Acquisition has no improving effect on the results, despite that more modules are introduced in the case where 50% of the population was involved in module creation. It is difficult to draw any definite conclusions about the nature of how modules should be selected based on these figures because only very few modules are present in the solutions. Had the newly created modules been used in conjunction with the creation of new individuals like Adaptive Representation (section 3.7) they would possibly have had a larger impact on future generations and then it would have been possible to see if Module Acquisition or Structural Module Creation have any benefits. It is also possible that neither Module Acquisition or Structural Module Creation are suited for problems like the even-k-parity problem.

| Method | Average size | Average number of modules |
|---|---|---|
| Standard GP | 321 | - |
| MA | 327 | 0.74 |
| SMC (50%) | 559 | 4.26 |
| SMC (10%) | 817 | 1.68 |

Figure 5.6: The average size and modules of Module Acquisition and Structural Module Creation on the even-4-parity problem.

## 5.10 Average size

One of the reasons to use a modularization method is to reuse code, thereby reducing the size of the solution programs. Therefore I measured the average size in nodes, including nodes in called modules, of the solutions to see if the two modularization methods had any effect on the size. As reference I used the standard Genetic Programming and Genetic Programming using Module Acquisition on the even-3-parity and even-4-parity problems. Figures 5.5 and 5.6 shows the result. Module Acquisition had almost the same average number of nodes per solution as standard Genetic Programming. This can maybe be accounted to the idea that the modules in Module Acquisition has little effect on the evolutionary process compared to standard Genetic Programming. The first experiment using Structural Module Creation (50% modified population) created solutions with larger size than both standard Genetic Programming and Genetic Programming using Module Acquisition, indicating that though the solutions contained on average more modules the modules were not very useful in reducing the number of nodes. When the percentage of modified individuals was decreased to 10% the average size was not far from standard Genetic Programming on the even-3-parity problem but was 2.5 times that of standard Genetic Programming on the even-4-parity problem.

It is hard to explain why Structural Module Creation exhibited so large average size of the solutions on the even-4-parity problem. Since very few modules were used in the solutions and they were of limited size (maximum three nodes with four arguments) it is hard to see that they could be the factor behind such large average sizes. This becomes even more clear when looking at Structural Module Creation that modified 10% of the population compared to modifying 50% of the population on the even-4-parity problem. It was expected that when the crossover operator had more material to work with it should result in solutions with average size closer to Module Acquisition and standard Genetic Programming but the average size increased dramatically although fewer modules were used.

## 5.11 Even-5-parity problem

The result of the even-5-parity problem with standard Genetic Programming and Module Acquisition can be seen in figure 5.7. On average, neither standard Genetic Programming or Genetic Programming using Module Acquisition found a solution in 50 generations. The time consumed for the two experiments were 10 hours 33 minutes and 21 hours 44 minutes respectively. Since Structural Module Creation had shown no improvement on the even-3-parity and even-4-parity problems, there was no point in running an experiment on the even-5-parity problem using Structural Module Creation when a solution was not likely to be found.

[Koza, 1992] ran experiments on the even-parity problem and succeeded in solving the even-5-parity problem using Standard Genetic Programming. He used a larger population, 8000 individuals, compared to the 1000 individuals in my experiments. Had I used a larger population it is possible that I also had been able to solve the even-5-parity problem. Furthermore, increasing the maximum number of generations could have led to a solution being found. As can be seen in figure 5.7, the standardized fitness was steadily approaching zero which would indicate that a larger value for the maximum number of generations could have led to a solution. However, a

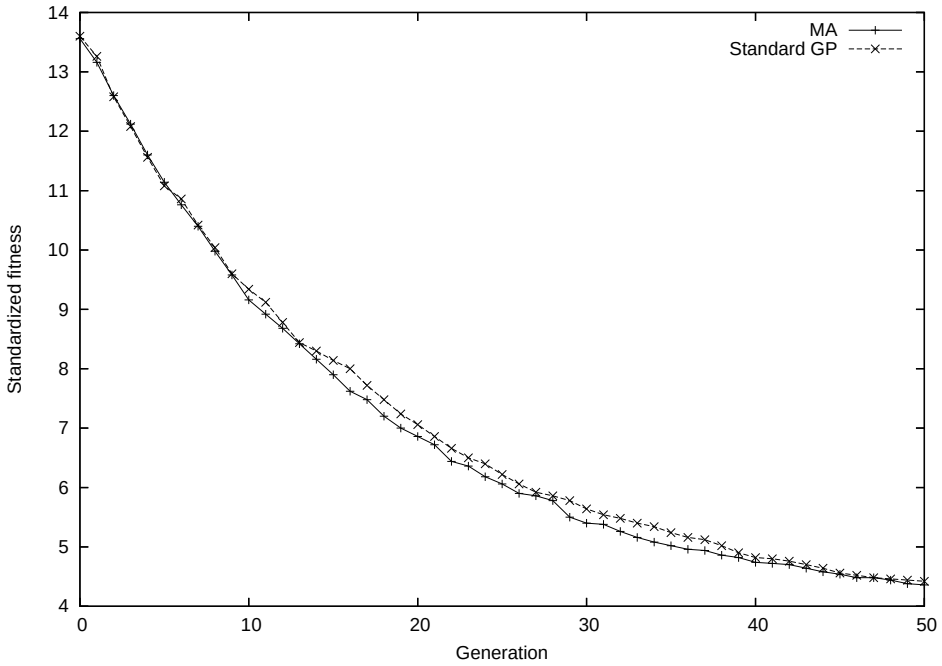



Figure 5.7: Comparing standard Genetic Programming and Module Acquisition on the even-5-parity problem

larger maximum number of generations is not a guarantee for finding a solution in this case. The standardized fitness might level out on a the value of one or higher, regardless of the maximum number of generations.

## 5.12 Conclusions

Looking at the even-3-parity problem, there is not much difference in either the average size nor the number of modules per solution for the standard Genetic Programming, Genetic Programming using Module Acquisition, and Genetic Programming using Structural Module Creation with 10% modified population. This gives and indication that both modularization methods have little positive impact over standard Genetic Programming, and in the case of Structural Module Creation with 50% modified population, a clear degradation is showed. On the even-4-parity problem, Module Acquisition followed standard Genetic Programming with respect to both size and number of generations until a solution was found. Structural Module Creation on the other hand did not improve either the size of the solutions nor the number of generations it took to find a solution.

The experiments showed no improvement when using Structural Module Creation over Module Acquisition except for a small time gain on the even-3-parity problem but compared to standard Genetic Programming using Structural Module Creation is a degradation in all respects, including time consumption, average size of solutions, and the number of generations until a solution has been found. Altering the increment values of the frequency table to did not help. The location of the module insertion(bottom-up or regular) appeared to have no effect on performance.

Seeing that both Module Acquisition and Structural Module Creation only propagate modules from generation to generation by the virtue of the fitness of individuals it is a clear possibility that useful modules created early are never used in later generations because they belong to individuals with too low fitness to survive. Had the modules been used to create new individuals

like Adaptive Representation (section 3.7) it is possible that they would have had a larger impact on future generations and then it would have been possible to see if choosing modules randomly like Module Acquisition or based on frequency of subtree structures like Structural Module Creation have any benefits.

The population size, 1000 individuals, was constant over all experiments. Had I tried to increase the population size for each subsequent increase in k, it is possible that a solution had been found for the even-5-parity problem, both using Standard Genetic Programming as well as Module Acquisition and Structural Module Creation. Also increasing the maximum number of generations could have led to a solution being found.

# Chapter 6

# Conclusions

This paper was an investigation into how modules can be created automatically with Genetic Programming. A description of Genetic Programming and how it has been applied to different areas was provided. Several modularization methods were presented and comparisons to other methods were referenced. A new modularization method was proposed, called Structural Module Creation, that creates new modules based on the frequency of subtree structures.

The results of experiments showed that Structural Module Creation introduced no improvement compared to Module Acquisition on which is was based. Experiments on the even-k-parity problem ($k = 3, 4, 5$) were conducted that measured average number of generations until a solution was found, the average number of modules in solutions, the average size of the solution, and time consumed until a solution was found.

One explanation that Structural Module Creation, and Module Acquisition, didn't perform well compared to standard Genetic Programming on the even-k-parity problem is that the newly created modules are only propagated through generations by the virtue of the fitness of the individual that contains invocations of the module. An increase in both population size and the maximum number of generations could have benefitted the problem solving process by allowing it more material and more generations to find a solutions.

During the research phase of this paper I discovered that no new research into modularization methods has been conducted since 1998. I have found no reason for this but it is possible that other methods have been discovered that solved the problems of efficiency and scalability that modularization methods try to solve. Another explanation is that modularization is not an effective way of dealing with the issue of scalability in Genetic Programming.

Initially the experiments would compare standard Genetic Programming with Module Acquisition, Structural Module Creation, and Genetic Programming using Automatically Defined Functions but due to problems during the implementation of Automatically Defined Functions I was forced to exclude them from the experiments.

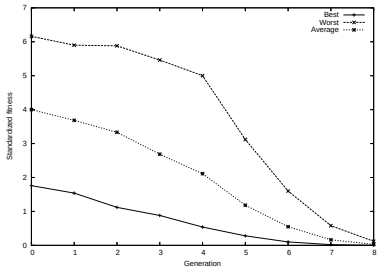
# Chapter 7

# Further work

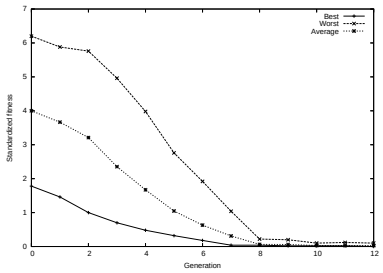
As mentioned in this paper, one of the possible reasons that Structural Module Creation performed so badly compared to standard Genetic Programming is that newly created modules have little impact on later generations because they are only propagated to later generations based on the virtue of the fitness of the individuals with invocations of the module. Therefore it would be interesting to see if Structural Module Creation has beneficial characteristics compared to standard Genetic Programming by letting modules have a larger impact from generation to generation. One way of doing this would be, like Adaptive Representation, to create new individuals each generation from the given function set, the terminal set and the created modules. Another way would be to randomly insert modules into individuals before they have been evaluated.

# Appendix A

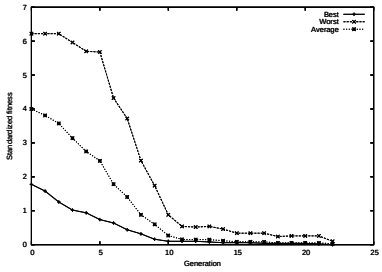
# Test runs



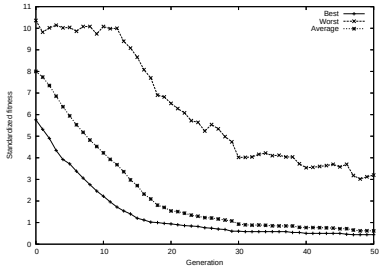

Even-3-parity problem with standard Genetic Programming
averaged over 50 iterations.



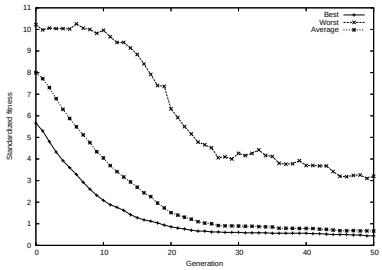

Even-3-parity problem with Module Acquisition
averaged over 50 iterations.



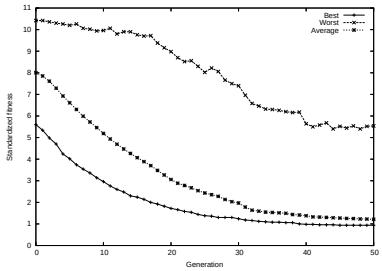

Even-3-parity problem with Structural Module Creation
averaged over 50 iterations with 50% of the population modified.



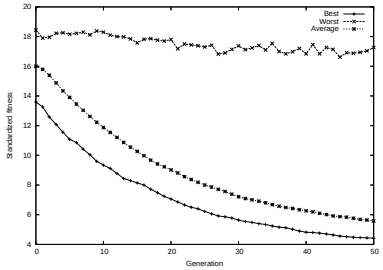

Even-4-parity problem with standard Genetic Programming
averaged over 50 iterations.



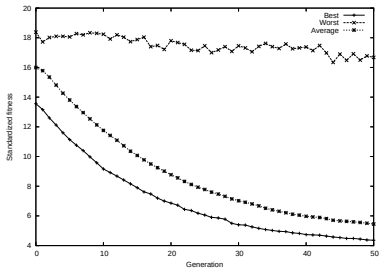

Even-4-parity problem with Module Acquisition
averaged over 50 iterations.




Even-4-parity problem with Structural Module Creation
averaged over 50 iterations with 50% of the population modified.




Even-5-parity problem with standard Genetic Programming
averaged over 50 iterations.




Even-5-parity problem with Module Acquisition
averaged over 50 iterations.

# Appendix B

# Implementation

The implementation was done in Java 5.0 using the Netbeans 5.0. The implementation of the genetic programming system can be seen to have three different parts: the Java classes, the input system, and the output and visualization system. The major Java classes can be divided into three parts:

- functions and terminals,
- tree representation of programs, and
- genetic programming concepts.

First, the functions and terminals part consists of the abstract class **Function** that is a virtual function[1] with the important method **evaluate()** that takes as arguments the inputs to the function. The return value is the result of the particular functions implemented by its subclasses. The subclasses to Function are the four functions **AND**, **OR**, **NAND**, and **NOR**. They each implement their corresponding binary boolean function resulting in a return value of either **TRUE** or **FALSE**. Both inputs and outputs from functions are stored as strings. The class **Terminal** is also a virtual terminal which holds, besides the name of the terminal, the value of the terminal. Both are stored as strings.

The tree representation consists of the class **Node** and its two subclasses **FunctionNode** and **TerminalNode**. The node contains references to parent nodes and child nodes, as well as various information used for visualizing the tree. A FunctionNode also contains a reference to a Function or one of its subclasses. A TerminalNode contains a reference to a Terminal.

The genetic programming concepts involves implementations of the population, individuals, programs, fitness, genetic operators and other similar classes. The three most important classes are **Individual**, **Fitness**, and **GeneticOperator**. The Individual class contains the program in tree representation, the fitness of the individual, and other information. The Fitness class contains the raw fitness and standardized fitness of the individual. There is also the possibility to extend this class to include normalized fitness, adjusted fitness, and parsimony fitness. Finally, the GeneticOperator class is an abstract class containing the probabilities for choosing function and terminal nodes in the genetic operator stage of the evolutionary process. Among the subclasses are Crossover, Mutation, Compress, and Expand. The reproduction operator is implicitly performed whenever an individual is copied and is therefore not included as a class.

The input system enables the system to set values of parameters that are stored in a file. The contents of the file (an ini-file) are just regular key-value pairs separated with the "=" character. If a key is not included in the file, the default value of the key is used instead. Some of these default values are not recommended. Currently, the following keys can be included in the file:

[1] The virtual functions exhibit the same behavior as their real world counterparts but they are implemented in software instead of hardware.

**maxProgramSize** the maximum depth of the individual. Currently, this has been converted into the maximum number of nodes by $2^{maxProgramSize}$. The default value is 17.

**maxInitialProgramSize** the maximum depth of the individual when it is randomly generated to be included in the initial population. Default value is 6.

**runs** the number of iterations the genetic programming system should make on the current problem. This is used to get an average over the measured quantities. The default value is 3.

**stream** saves the files in a gzipped file if the value is 'true'. Otherwise, if the value is 'false', the files are saved in dot-format and txt-format. The default value is 'true'.

**convert** converts the dot-files into jpg-files. Possible values are 'true' and 'false'. If the value is 'true' the **stream** key must be present and given the value 'false'. Default value is 'false'.

**even-k-parityproblemArguments** $k$ number of arguments to the even-k-parity problem. The default value is 0 for some reason.

**populationSize** the number of individuals in the population. The default and minimum value is 10.

**maxGenerations** the maximum number of generations to evolve before terminating the execution of the Genetic Programming system. Default value is 2.

**fitnessThreshold** the minimum value of the standardized fitness that should be considered a success of the execution of the Genetic Programming system. Default value is 0.1.

**hitsThreshold** the number of successfully classified fitness cases. The default value is 0.1.

**selectionMethod** the selection method that is used to select individuals for genetic operators. Currently, the only valid value, and also default value, is 'tournament'. This uses tournament selection.

**tournamentSize** the size of the tournament. The default value is 7.

**crossoverProbability** the probability that crossover is chosen as the genetic operator during the creation of the next generation. The default value is 0.9.

**reproductionProbability** the probability that reproduction is chosen as the genetic operator during the creation of the next generation. The default value is 0.1.

**mutationProbability** the probability that mutation is chosen as the genetic operator during the creation of the next generation. The default value is 0.0.

**modularizationMethod** the used modularization method. The possible values are 'none' (default), 'MA' (Module Acquisition), and 'SMA' (working name for Structural Module Creation was Structural Module Acquisition, hence the abbreviation SMA).

**compressionProbability** if Module Acquisition is used, this key determines the probability for compression to be selected as the genetic operator during the creation of the next generation. The default value is 0.0.

**expansionProbability** if Module Acquisition is used, this key determines the probability for expansion to be selected as the genetic operator during the creation of the next generation. The default value is 0.0.

**trimDepthMin** if Module Acquisition is used, this key determines the minimum of the trim depth. The default is 1.

**trimDepthMax** if Module Acquisition is used, this key determines the maximum of the trim depth. The default is 4.

**probabilityOfModuleInsertion** if Structural Module Creation is used as the modularization method, this key determines the probability that a module should replace a node in the tree. The default value is 1.0.

**numberOfNewModulesPerGeneration** if Structural Module Creation is used as the modularization method, this key determines the number of new modules that will be created per generation. The default value is 10.

**percentOfPopulationToModify** the percent of the population to modify with modules. The default value is 0.5.

**saveResultOfFitnessCase** saves the result of each fitness case in a new file. Used mainly for debugging. Possible values are 'true' and 'false'. The default value is 'false'.

Output from the system can be either in dot-files with the option of converting them to jpg-files, or it can be in the form of a gzipped file containing the all created individuals during the execution of the system. If MA or Structural Module Creation are used as modularization method, gzipped files of each body of the modules as well as the information about all modules during the execution of Genetic Programming system are created.

The visualization system uses dot, a part of [Graphviz], a program that converts specially formatted text to images. This program must be installed in order to graphically present individuals. All images are in the JPEG-format.

# Bibliography


Peter J. Angeline and Jordan B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 236–241, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum. URL `http://www.demo.cs.brandeis.edu/papers/glib92.pdf`.

Yaniv Azaria and Moshe Sipper. Using GP-gammon: Using genetic programming to evolve backgammon players. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 132–142, Lausanne, Switzerland, 30 March - 1 April 2005. Springer. ISBN 3-540-25436-6.

Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, January 1998. ISBN 1-55860-510-X.

Gregory J. Barlow. Design of autonomous navigation controllers for unmanned aerial vehicles using multi-objective genetic programming. Master's thesis, North Carolina State University, Raleigh, NC, USA, March 2004.

Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. 2000.

Andries P. Engelbrecht. *Computational Intelligence. An Introduction*. 2007.

genetic-programming.com. www.genetic-programming.com, 2007. URL `www.genetic-programming.com`.

Gnuplot. URL `http://www.gnuplot.info/`.

Graphviz. URL `http://www.graphviz.org/`.

John H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975. ISBN 0472084607. URL `http://www.amazon.fr/exec/obidos/ASIN/0472084607/citeulike04-21`.

Kenneth De Jong. Evolutionary computation: a unified approach. 2007.

Kenneth E. Kinnear, Jr. Alternatives in automatic function definition: A comparison of performance. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 6, pages 119–141. MIT Press, 1994.

John R. Koza. Introduction to genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 2, pages 21–42. MIT Press, Cambridge, MA, USA, 1994a.

John R. Koza. Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations. In John Robert McDonnell, Robert G. Reynolds, and David B. Fogel, editors, *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 695–717, San Diego, CA, USA, 1-3 March 1995. MIT Press. ISBN 0-262-13317-2.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994b. ISBN 0-262-11189-6.

John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV : Routine Human-Competitive Machine Intelligence (Genetic Programming)*. Springer. ISBN 1402074468.

John R. Koza, David Andre, Forrest H Bennett III, and Martin Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, April 1999. ISBN 1-55860-543-6. URL `http://www.genetic-programming.org/gpbook3toc.html`.

Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6–22, Tahoe City, California, USA, 9 1995. URL `citeseer.ist.psu.edu/nordin95explicitly.html`.

Mihai Oltean. Solving even-parity problems using traceless genetic programming, 2004.

J. P. Rosca and D. H. Ballard. Learning by adapting representations in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA*, Orlando, Florida, USA, 27-29 June 1994a. IEEE Press.

Justinian P. Rosca and Dana H. Ballard. Genetic programming with adaptive representations. Technical Report TR 489, Rochester, NY, USA, 1994b. URL `citeseer.ist.psu.edu/rosca94genetic.html`.

Justinian P. Rosca and Dana H. Ballard. Discovery of subroutines in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-01158-1. URL `ftp://ftp.cs.rochester.edu/pub/u/rosca/gp/96.aigp2.dsgp.ps.gz`.

Lee Spector. Simultaneous evolution of programs and their control structures. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 7, pages 137–154. MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-01158-1.

Huayang Xie, Mengjie Zhang, and Peter Andreae. Genetic programming for automatic stress detection in spoken english. In Franz Rothlauf, Jurgen Branke, Stefano Cagnoni, Ernesto Costa, Carlos Cotta, Rolf Drechsler, Evelyne Lutton, Penousal Machado, Jason H. Moore, Juan Romero, George D. Smith, Giovanni Squillero, and Hideyuki Takagi, editors, *Applications of Evolutionary Computing, EvoWorkshops2006: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoInteraction, EvoMUSART, EvoSTOC*, volume 3907 of *LNCS*, pages 460–471, Budapest, 10-12 April 2006. Springer Verlag. ISBN 3-540-33237-5.