

MASTER'S THESIS
UNRAVELLING THE GENETIC STRUCTURE
OF NP-COMPLETENESS

Tijn Witsenburg
Leiden Institute of Advanced Computer Science

October 26, 2006

This Master's Thesis was written at the Leiden Institute of Advanced Computer Science (LIACS), part of Universiteit Leiden, under the supervision of dr. W.A. Kusters and dr. M.T.M. Emmerich.

Acknowledgements

First and foremost I would like to gratefully thank my thesis advisor Walter Kusters who, throughout the entire extend of my research, was available for helping me on all sorts of problems. Also I would like to thank Michael Emmerich for his useful advice.

Furthermore, I would like to thank Remco Nabuurs for his help on Energy Bound Genetic Programming and Yves van Roon for reading it all and his suggestions on the text.

Last, but not least, I would like to thank my parents, Jacques and Tinie, for all their support during the many years I of my education.

Contents

1 Preface	3
1.1 Introduction	3
1.2 Objectives	3
1.3 Overview	4
2 Historical Background	5
2.1 Genetic Programming	5
2.1.1 Basic Concepts of Evolutionary Algorithms	5
2.1.2 Difference between GP and EA	7
2.2 NP-Complete Problems	8
2.2.1 Complexity	8
2.2.2 The Big Question: Is P Equal to NP?	9
3 Six Problems to Solve	13
3.1 General Principle	13
3.1.1 Greedy Algorithms	14
3.1.2 Training Sets	15
3.2 Maximum Clique Problem	15
3.2.1 Greedy Algorithm for MCP	16
3.2.2 Used Graphs for MCP	16
3.3 Graph Colouring Problem	17
3.3.1 Greedy Algorithm for GCP	18
3.3.2 Used Graphs for GCP	18
3.4 Hamilton Cycle Problem	19
3.4.1 Greedy Algorithm for HCP	19
3.4.2 Used Graphs for HCP	19
3.5 Minimum Spanning Tree	20
3.5.1 Greedy Algorithm for MST	21
3.5.2 Used Graphs for MST	21
3.6 Bipartite Graph Problem	22
3.6.1 Greedy Algorithm for BGP	22
3.6.2 Used Graphs for BGP	23
3.7 Connected Component Problem	23
3.7.1 Greedy Algorithm for CCP	24
3.7.2 Used Graphs for CCP	24

4	Setup of the Program	26
4.1	Working of an Individual	26
4.1.1	The Function Set	26
4.1.2	Processing a Graph	30
4.2	Energy Bound Genetic Programming	31
4.2.1	Explosive Intron Growth	31
4.2.2	Energy Bound GP, a Practical Solution	32
4.2.3	Principles of Energy Bound GP	32
4.2.4	A Flaw in the Theory	33
4.3	Working of the Genetic Program	35
4.3.1	Is there enough Energy?	35
4.3.2	Initialisation	35
4.3.3	Evaluation and Viability Check	36
4.3.4	Selection	37
4.3.5	Creation of Offspring	37
4.3.6	Making a Time Step	40
4.3.7	Termination	41
5	Fine-tuning the Program	43
5.1	General Settings	43
5.2	Tuning E_{fixed} and the Genetic Operators Rate	44
5.3	Tuning E_{total} and E_{birth}	46
5.4	The Chosen Settings	47
5.5	Testing the Results on a Series of Test Graphs	49
6	Results	51
6.1	Results for the Test Set	51
6.2	Influence of the Population	51
6.3	Influence of the Individuals	53
6.3.1	Fitness of the Individuals	55
6.3.2	Size of the Individuals	58
7	Conclusions	61
7.1	Main Conclusion	61
7.2	Something to Think About	61
7.3	Further Research	62

Chapter 1

Preface

1.1 Introduction

One of the most intriguing questions of contemporary computer science is the question “Is P equal to NP?”. There even is a million dollar price [1] for the first person to come up with the answer. Provided, of course, that he or she can prove its correctness.

Over time computer science has developed many different ways to solve many different types of problems. One of these is Genetic Programming (GP). GP is said to be able to solve all sorts of problems, or at least come to a reasonable answer within a reasonable time.

Now the question that rises is: “Why don’t we use Genetic Programming to look for the answer to the question whether P equals NP?” In theory that would be a good idea. Unfortunately, in practice it would turn out that Genetic Programming is not that powerful yet. On the other hand, we might be able to use Genetic Programming to provide us with more insights into the structural differences between NP-complete problems and problems that are known to have polynomial complexity.

1.2 Objectives

For this thesis a Genetic Program which can solve different problem types is designed. For every problem the GP will receive a graph as input and return a permutation of all the vertices. This permutation will be used to feed a greedy algorithm, which is custom made for each of the problems. The quality of an individual in the GP will be ascertained by the way it is able to arrange the vertices in such an order that the greedy algorithm can come to a good result. This Genetic Program will be used to solve six different problems. Three of them are known to be NP-complete. These are:

- Maximum Clique Problem
- Graph Colouring Problem
- Hamilton Cycle Problem

The three others are known to be solvable in polynomial time. These are:

- Minimum Spanning Tree
- Bipartite Graph Problem
- Connected Component Problem

Subsequently the results of the GP will be compared to see if there are differences between the two types of problems. The aspects that will be compared are:

- The quality of the found solutions
- The time in which it reaches the found solution
- The size of the found solutions

1.3 Overview

First a historical background for the things we will come across in this thesis is given. This is found in Chapter 2 where the first part is dedicated to Genetic Programming and the second to the question “Is P equal to NP?”.

Chapter 3 is all about the six problems the GP will try to solve. First is explained why this setup with greedy algorithms is chosen. This is followed by a description of the six problems. Besides a formalization of the problems there is also a description of the greedy algorithms used and all their aspects. This is followed by Chapter 4 in which is shown how the Genetic Program is built up and how it works. This is done on both the level of the genetic structure of an individual and on the level of the life cycle of a population.

In Chapter 5 the result of the first runs of the GP are presented. These runs are needed to find good parameters for the GP. Then in Chapter 6 are the results for the GP's on the different problems, which is followed by Chapter 7 where the main conclusion and recommendations for further research are presented.

Chapter 2

Historical Background

2.1 Genetic Programming

There are two ways to make a computer do what you want it to do. One way is to tell it exactly what it needs to do. The other is to let it learn by itself what the best way is to solve a problem. In that case you need to tell the computer exactly how to learn. The first attempts to let a computer learn by itself were done by Friedberg in 1958 [17]. From there it took another 40 years before we came to Genetic Programming (GP).

2.1.1 Basic Concepts of Evolutionary Algorithms

Darwin argued that

...if variations useful to any organic being do occur, assuredly individuals thus characterised will have the best chance of being preserved in the struggle for life; and from the strong principle of inheritance they will tend to produce offspring similarly characterised. This principle of preservation, I have called, for the sake of brevity, Natural Selection.

C. Darwin, 1859 [14]

From the concepts of the theory of evolution, combined with the field of research in biology that researches natural reproduction, different types of computer simulation of evolution have arisen. They are inspired by the theory of evolution, combined with biology, but they have never intended to give a good description of natural reproduction. They only use the concepts of evolution applied in an algorithm to search for optimal solutions. These solutions can be of any form, from as simple as a single integer to as complex as a complete algorithm.

The general term for all these techniques is Evolutionary Algorithms (EA). GP is one of them. Other important EA's are Evolutionary Programming developed by Fogel, Owens and Walsh [16], Evolutionary Strategies developed by Rechenberg [34] and Schwefel [36]¹ and Genetic Algorithms, developed by Holland [20]. And even though they all vary in the exact way how they work, they all work according to the basic concepts of EA.

¹Rechenberg and Schwefel did their initial research in 1965 but they both refined their work in resp. 1994 and 1995.

Banzhaf et al. stated [5]: “Evolutionary algorithms work by defining a goal in the form of a quality criterion and then use this goal to measure and compare solution candidates in a stepwise refinement of a set of data structures.” In this way, speaking in terms of the theory of evolution, a better solution, as defined by our quality criterion, will have a better chance of being preserved in the struggle for life. The quality of an individual is generally referred to as its ‘fitness’ and a higher fitness means that it has a greater chance of being selected. From these better solutions new offspring is created which has a good chance of being even better. The schematic form for such algorithms is drawn in Figure 2.1.

To create new offspring there are many different methods e.g.: copying or random generation. An algorithm is said to be an evolutionary algorithm when among its creation methods there are at least *mutation* or *crossover*. Therefore these two are called the *genetic operators*.

When a new individual is created using mutation, one individual is partly changed and the result is added to the population. The way mutation is implemented depends on how an individual is represented. Apart from that, normally, programmers tend to let the effects of mutation come along with the normal distribution. This means that small changes in the individual will have a much greater chance than large changes.

When a new individual is created using crossover, two individuals combine to produce new offspring. Again, here the exact implementation differs with the sort of individuals used. Generally, in all the parents one or more points are chosen and all the information between those points is exchanged.

There is a distinction between the way an individual looks and its genetic structure. For this Johannsen defined the terms genotype and phenotype [22]. The genotype of an individual is its genetic structure, also known as its DNA. The phenotype is the set of observable properties of an individual; the way an individual looks. Mutation and crossover are performed on the genotype which will result in the phenotype looking differently. On the other hand, the fitness of an individual is determined by the way the phenotype is able to face up to the problems it encounters, thus allowing the genotype to reproduce.

Using mutation and crossover, new individuals are created to replace the old. There are many ways to do this, each with their own, sometimes subtle, differences. The two most famous of these are ‘generational’ and ‘steady-state’. With the generational replacement method, first, a part of the generation is selected. This selection is used to create an entire new generation which will completely replace the old one. Here all individuals will reach the same age, namely ‘1’.

With the steady-state replacement method things run more smoothly. Very often it is combined with tournament selection where a certain number of individuals is selected for a tournament. The winner of the tournament (the individual with the best fitness) is allowed to reproduce offspring. This offspring will replace the loser of the tournament. In this way, many individuals can survive multiple tournament-rounds since they are either not chosen to compete in a tournament or they do not lose it. Here, after a while, we can see a variety of ages coming into being. For experiments and detailed references on generational versus steady-state GP see [24].

In either way generation after generation is created until the found solution is good enough. This “good enough” is to be decided by the programmer. It can be for instance when a solution reached a certain value or when the quality of the population does not grow anymore.

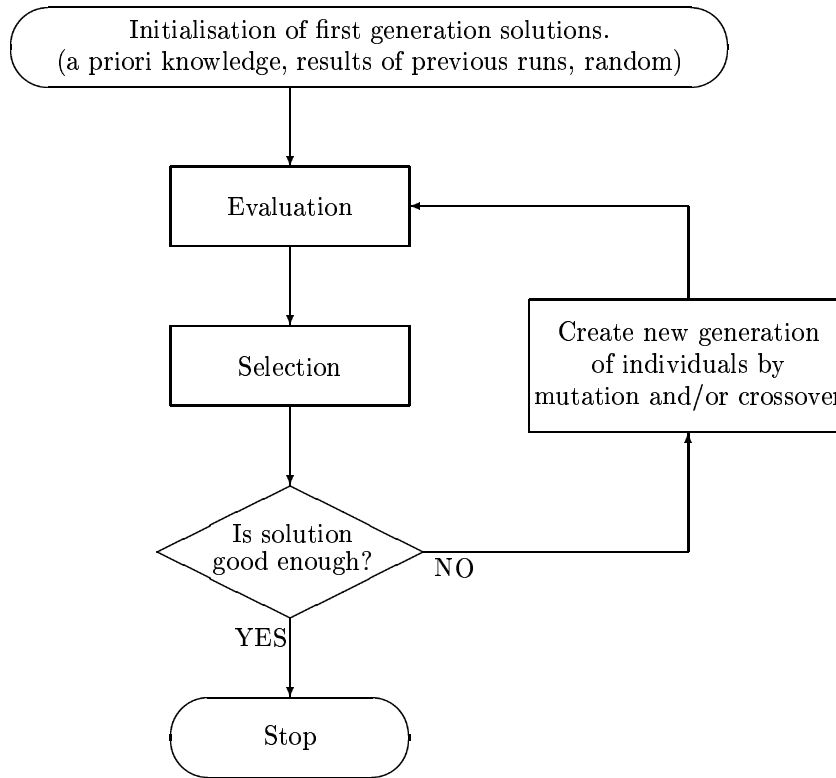


Figure 2.1: General working of an Evolutionary Algorithm

2.1.2 Difference between GP and EA

Genetic Programming is a type of Evolutionary Algorithm. In 1992 Koza wrote a treatise entitled “Genetic Programming. On the Programming of Computers by Means of Natural Selection” [26]. He was the first to recognize GP was something new and different and gave the new discipline its name. Koza achieved his results by evolving tree structures. Therefore even until this day almost all Genetic Programming is done using tree structures.

GP distinguishes itself from EA in several ways. The main difference is the fact that every individual in a genetic program is a program itself. A Genetic Program therefore is a program that uses the principles of Evolutionary Algorithms to search for a program that can solve a particular, general problem.

A second important difference is the fact that Genetic Programming does not really distinguish between the genotype and the phenotype. The program is stored in a tree structure and this structure is both used for mutation and crossover as well as for the execution of the program. The only way they are distinct is the fact that the exact tree structure of the code can be seen as the genotype and that the behaviour of the program originated from the code can be seen as the phenotype.

2.2 NP-Complete Problems

Problems can vary greatly in their level of difficulty. They go from arbitrary via easy and difficult to impossible. In addition there are problems of which it is unknown how difficult they are. Among them is an entire class of problems with the same, but yet unknown, difficulty. These are in the class of NP-complete problems. Let us take a closer look at them.

2.2.1 Complexity

Computer programming is all about solving problems using algorithms. The quality of an algorithm depends on a variety of factors. Two of them will be discussed here. First, it is important to know how well an algorithm performs, and second it is important to know how much resources it needs to come to a solution.

It is quite arbitrary to see why the quality of an algorithm depends on its performance. When the solution is most of the time completely wrong, the algorithm is pretty useless. When the algorithm always returns a fully correct and near optimal solution, this algorithm can be a very good and useful one. That is if the amount of resources needed is in such a range that we can actually use it. Consider an algorithm which is known to always correctly solve a certain problem. That sounds like a good algorithm. But what if it will take several centuries to come to that solution? Then you would be dead by the time it finishes and it would therefore be pretty useless. Maybe another version of that algorithm can speed up things by splitting the algorithm up and calculate in parallel. Unfortunately, if the original algorithm needed several centuries to come to completion, we still may need several thousands of computers to solve the problem fast enough. If it is The Algorithm to find the answer to the ultimate question of life, the universe and everything, you might consider making the extra costs. However, when it is just a problem of minor importance it is clear to all that this is not worth it.

Both the amount of time and computer resources an algorithm needs is referred to when speaking of the complexity of a problem. Of course these two are inter-linked as we have seen earlier. When looking at the amount of time an algorithm uses this is referred to as the time-complexity and with respect to the amount of computer resources needed, this is referred to as space-complexity.

When regarding a certain problem, the most efficient algorithm known to solve it determines its complexity. This of course is arbitrary. When you have a very simple problem, that is, with low complexity, writing a very inefficient algorithm will not make the problem more difficult. In contest, writing an improved and more efficient algorithm will reduce the complexity of a problem. For some problems it is proven that it is impossible to find a better algorithm. Their complexity has been fully determined. For other algorithms their complexity is not yet fully determined.

Now, complexity can be defined as the intrinsic minimum amount of resources, for instance, memory, time, messages, etc., needed to solve a problem or execute an algorithm [7]. This complexity is written as a function of the number of inputs. It says something about the ratio of the number of inputs to the amount of resources needed rather than giving an exact value for them. This of course because the latter differs from computer system to computer system.

To formalize this we need a special notation. First, let us define n as the number of inputs. When we regard a sorting algorithm, n will be the number of elements we need to sort. When regarding a graph problem, determining n is already more ambiguous. We could choose the number of vertices plus the number of edges ($n = |V| + |E|$) or we could look at the adjacency matrix which will result in $n = |V|^2$. Therefore, in case of doubt, you always need to clarify how you defined n .

Now, assume the function $R(n)$ gives the amount of resources needed to run a certain algorithm. Please keep in mind that $R(n)$ is not a fully determined function. We can then say that $R(n) = \Theta(g(n))$ where $g(n)$ is a fully defined function. For a given function $g(n)$ we denote by $\Theta(g(n))$ the set of functions:

$$\Theta(g(n)) = \{f(n) : \exists c_1 > 0, c_2 > 0, n_0 : \forall n > n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be “sandwiched” between $c_1 g(n)$ and $c_2 g(n)$ for sufficiently large n ([12]). When function $R(n)$ is part of the set of functions $\Theta(g(n))$ we say $R(n) = \Theta(g(n))$ instead of the syntactically more correct $R(n) \in \Theta(g(n))$. For example, painting a fence has complexity $\Theta(n)$. When the fence is twice as big, it will take you about twice the amount of time to paint it. On the other hand, looking something up in a dictionary has complexity $\Theta(\ln(n))$. When the dictionary is twice as big, it will only take you one action (checking the word in the middle to see in which half of the dictionary the word you are looking for is) to reduce the problem to the half.

Now the Θ -notation bounds the function from above and from below. When we only have an upper bound, we use the \mathcal{O} -notation which only holds the condition $0 \leq f(n) \leq cg(n)$. On the other hand, when we only have a lower bound we use the Ω -notation which only holds the condition $0 \leq cg(n) \leq f(n)$.

2.2.2 The Big Question: Is P Equal to NP?

Problems have many different aspects which differ from one problem to the next. There are differences in the types of input (lists, matrices, graphs etc.) as well as in the types of output (Boolean, integer, set of edges etc.). Problems that return a Boolean are known as decision problems. This is, for instance, “Is there a path from one vertex to another in a given graph?” The answer will be a simple ‘yes’ or ‘no’. Another type of problem is that of optimization, for instance, “How long is the shortest path from one vertex to another in a given graph?” The solution will be a number indicating the length of that path.

Most of the time, optimization problems are considered to be more interesting since they provide us with more complex information. On the other hand, decision problems can be more easily formalized. This makes it easier to use them in complicated theories. Fortunately, every optimization problem can be written as a decision problem. For instance, instead of questioning what the shortest path is, we can ask if there is a path with length $< \ell$, which is a decision problem. If it exists and there is no path with length $< \ell - 1$ we have found the length of the shortest path.

The theory of NP-completeness is meant for decision problems. This leads us to rewrite optimization problems as decision problems. Although this can be more time consuming than you sometimes wish for, it does not weaken the strength of the theory. When an optimization problem is known to be quickly solvable, its decision problem is that as well: just solve the optimization problem and compare its value with the bound of the decision problem. And vice versa, when a decision problem is hard, its related optimization problem is hard as well. Thus despite the fact that this theory is formalized for decision problems, it applies much more widely.

The formalization of NP-completeness starts at the level of Turing machines and their input. Here it will be explained more intuitively. We have seen that the complexity of a problem says something about whether it is possible to solve it efficiently or whether we might have to wait for a long time. Now, it is good to know when a problem is still considered to be solvable. Therefore computer scientists decided that any problem with polynomial complexity, that is with complexity $\Theta(n^c)$, for some constant c , is solvable within a proper timeframe. This looks a bit like it is chosen arbitrary, but that is not quite the case. Years of experience with computer problems have shown that this is a reasonable boundary to divide the problems. It can be discussed, since, for instance, a problem with complexity $\Theta(n^{1000})$ does not look that solvable. On the other hand, the most polynomial problems used in practise seem to have a fairly low exponent, this is not considered a big issue and it is a bit harsh to exclude all problems with polynomial complexity because of a single theoretical problem.

A set of problems with the same complexity, the same bounds $\Theta(f(n))$ for needed resources, is called a complexity class. The complexity class P is the set of concrete decision problems that are solvable in polynomial time. They can be solved easily, which, unfortunately, cannot be said for all problems. The Hamilton Cycle Problem is considered more difficult. A Hamilton cycle is a route that goes through all the vertices exactly once and ends at the same vertex it started. When trying to find a Hamilton cycle there is basically no faster solution than to just try and as long as you do not find one, you might end up having to try all possibilities. This of course can not be done in polynomial time.

Although it can take an enormous amount of time to find a Hamilton cycle it is fairly easy to verify if a given string of vertices represents such a cycle. This is simply a matter of checking whether it is a permutation of all vertices and whether every consecutive vertex is connected to the previous. This can be done in polynomial time in the length of the input. From this principle we can define a new complexity class, known as NP which is the set of problems for which a given solution can be verified in polynomial time. NP stands for non-deterministic polynomial time. This finds its origin at Turing machine level and, as mentioned before, this will not be discussed deeply. People who are interested in that are recommended to read the book by Hopcroft and Ullman [21].

For every problem that can be solved in polynomial time, the solutions can be verified in polynomial time. Therefore we can say that every problem in P is in NP or, to say it more formally, $P \subset NP$. The million dollar question [1] now is if every problem in NP is in P, making the two classes identical and ultimately defining the question as: "Is P equal to NP?"

Consider again the Hamilton cycle problem which is in NP. A way has not yet been found to solve it in polynomial time. This, of course, does not mean that it is impossible to solve it in polynomial time. A sharp lower bound on

its complexity has not yet been determined. Like the Hamilton cycle problem, there are more problems which are verifiable in polynomial time but of which it is not sure whether they can be solved in polynomial time. They all belong to a special complexity class called NP-complete. Since they are all verifiable in polynomial time we can say $\text{NP-complete} \subset \text{NP}$.

In order to be part of the NP-complete class, a problem needs to have three specific properties. The first we have already seen and that is the fact that it needs to be verifiable in polynomial time. The second is the fact that it needs to be a decision problem. Since we have already seen that we can rewrite a discrete optimization problem as a decision problem, this property still applies to a lot of problems. We only need to be very careful when defining our problem. Finally, the third property is the fact that the problem could be reduced to any other NP-complete problem in polynomial time. Since they are all reducible to each other, you only need to prove it is reducible to one other known NP-complete problem to prove a problem is NP-complete.

This theorem was introduced by Steven Cook in 1971 [11]. In his famous article he also proved the Boolean satisfiability problem (SAT) to be NP-complete. In doing so he made SAT the first known NP-complete problem. This was quite an effort of course, since he could not use the reducibility to another problem. It was Richard Karp who in 1972 proved that one could prove a problem to be NP-complete by reducing it to another. He used it to show that 21 diverse combinatorial and graph theoretical problems, as shown in Table 2.1, each infamous for their intractability, were all NP-complete [23]. Many problems have since been proven NP-complete using this reducibility. Today the list of NP-complete problems contains several hundred and more problems are still added. The class of NP-complete problems is known to be the most difficult of all problems in NP. Therefore they are most likely of all NP-problems to not be in P, although this has not yet been proven. There could still be a chance that they are all in P. When one problem is proven to be in P, then, due to their reducibility in polynomial time, they will all be in P. This is good news for those who want to prove that $P = \text{NP}$ since it will save them a lot of work. They only need to find one. On the other hand, the many years of fruitless search for a polynomial time solution for any of the NP-complete problems does not leave much room for hope.

If you want to go deeper into the matter of NP-completeness, it is recommended that you read the book by M.R. Garey and D.S. Johnson [18].

- SATISFIABILITY (Boolean satisfiability problem, proved by Cook)
 - CLIQUE
 - SET PACKING
 - VERTEX COVER
 - SET COVERING
 - FEEDBACK ARC SET
 - FEEDBACK NODE SET
 - DIRECTED HAMILTONIAN CIRCUIT
 - UNDIRECTED HAMILTONIAN CIRCUIT
 - 0-1 INTEGER PROGRAMMING
 - 3SAT
 - CHROMATIC NUMBER
 - CLIQUE COVER
 - EXACT COVER
 - 3-dimensional MATCHING
 - STEINER TREE
 - HITTING SET
 - KNAPSACK
 - JOB SEQUENCING
 - PARTITION
 - MAX-CUT

Table 2.1: Karp's 21 problems, many with their original names, with the nesting indicating the direction of the reductions used. For example, KNAPSACK was shown NP-complete by reducing EXACT COVER to KNAPSACK.

Chapter 3

Six Problems to Solve

3.1 General Principle

In this thesis a Genetic Program is made and tested on six different problems. Some of these problems are known to be NP-complete and some are known to be solvable in polynomial time. The question is whether we can see a difference in performance between the two kinds of problems.

The genetic program will evolve individuals which are all algorithms. The precise working of the program and its individuals is described in chapter 4. In general, every individual is an algorithm that gets a graph as input and returns a permutation of all the vertices as output. This permutation will serve as the input for a greedy algorithm which will use it on the graph to produce a solution for the given problem. This can be seen schematically in the data flow chart in Figure 3.1. The found solution determines the fitness value for the individual.

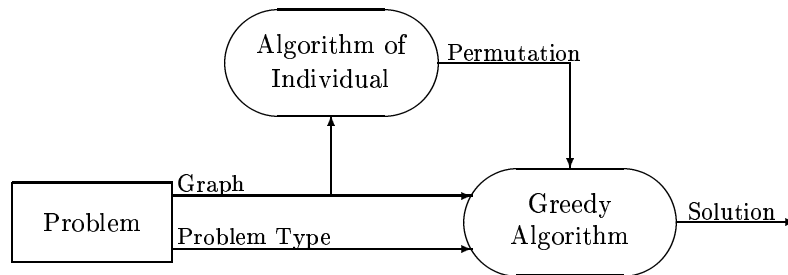


Figure 3.1: Data flow chart of how an individual combined with the greedy algorithm solves a problem.

3.1.1 Greedy Algorithms

While searching for good algorithms to solve the Maximum Weight Clique Problem (MWCP, see Section 3.2) researchers came to several interesting discoveries. These will not be discussed too deeply, but interested readers can read the proposed literature. There are many reformulations of MCP and MWCP as continuous global optimization problems (see [8]). In particular, a generalization of an earlier result by Motzkin and Strauss [31] shows that MWCP can be reformulated as the problem of minimizing a quadratic function over the unit simplex. Based on this generalization Massaro et al. wrote a paper in which they proposed a new Pivoting-Based Heuristic (PBH) [30]. The algorithm is essentially a variant of Lemke's classical algorithm [28] that incorporates an effective look-ahead pivot rule, and it proved to be among the most powerful MWCP heuristics available in literature.

In their paper in 2004 [29] Locatelli et al. showed theoretical results which allow an interesting combinatorial interpretation of the algorithm: PBH is essentially equivalent to a greedy combinatorial heuristic. In this paper they used a setup which was almost identical as the one I am using now. Although its effectiveness has only been shown in the case of MWCP, intuitively we can see that using this approach for all six problems is not all too farfetched.

The greedy algorithms used for the six problems are all written in the same form:

1. All vertices are ordered (v_1, v_2, \dots, v_N) by the permutation (with N the amount of vertices).
2. There is a number of sets, with a maximum of $N + 1$.
3. Put the first vertex of the permutation in the first set.
4. For every other vertex do:
5. evaluate the vertex;
6. put the vertex in one of the sets.

This form satisfies a number of requirements. Every vertex is considered once and will be placed into exactly one of the sets. When it is placed, it can never be replaced again. The exact way in which a vertex is evaluated depends, of course, on the problem it is for. Furthermore, during this evaluation the algorithm is only allowed to examine the relation between the vertex which is currently being evaluated and any of the vertices which are already evaluated. Examining this relation solely involves looking in which set the other vertex is, checking if there is an edge between them and if it is, appraising its weight.

All this ensures a simple, straight forward and thus greedy algorithm which is deterministic when the evaluation part is deterministic. Despite all these limitations, the greedy algorithm needs to be able to solve the problem. This means that for every possible graph, there must be at least one permutation which will result in the correct answer. For this the algorithm may use some additional variables which are only used to store information.

These algorithms are greedy, but their complexity is not always $\Theta(|V|)$ with $|V|$ the amount of vertices in the graph. This is because sometimes for every vertex we need to consider its relation with all previously considered vertices

which will result in an upper bound of $\mathcal{O}(|V|^2)$. This could have its effect on the complexity of the schema in Figure 3.1 but that is not a problem. This thesis is not about the complexity of the genetic program, but it will try to show if there is a difference in performance of the genetic program when trying to solve problems which may, or may not, have a different complexity.

3.1.2 Training Sets

Every individual needs to be tested in order to determine its fitness. To do so a certain number of graphs is needed. These graphs are called the *training set* and are determined to contain 100 graphs. They are created randomly for each run of the genetic program. A good training set should be challenging and leave a wide variety of solutions, ranging from poor to very well. This is to ensure that the program is able to make a good distinction between the performances of the individuals. Thus, taking “just” random graphs may not always give a good performance of the program.

In order to make good training sets it is therefore wise to create the random graphs of the training set in such a way that they have a fair chance of possessing certain “features”. These features should make it easier for the program to find a difference in performance for the individuals and they are, of course, different for each of the six problems and are described further on in this chapter.

3.2 Maximum Clique Problem

Considering a graph, we say that a clique is a subset of the vertices for which every two vertices in that subset are connected with an edge. When it is not possible to add a vertex to this subset and still have a clique, we say it is a local maximum. Of all the cliques in a graph, the cliques with the biggest amount of vertices are considered to be a maximum clique of that graph.

Finding the biggest clique in a graph is obviously an optimization problem. It can easily be rewritten as a decision problem. We can define the Maximum Clique Problem (MCP) as the following decision problem:

Considering a graph, is there a subset of at least ℓ vertices that are all connected to each other?

Now if we would ask ourselves this question with slowly increasing values for ℓ , we can then find the answer for the optimization version of this problem at the point where the answer to the question flips from ‘yes’ to ‘no’.

MCP is known to be NP-complete and even approximating it within a constant factor has recently been shown to be NP-complete. Strong evidence of this fact came in 1991, when Feige et al. [15] proved that if there is a polynomial-time algorithm that approximates the MCP within a factor of $2^{\log^{1-\epsilon} n}$ with ϵ a very small, positive number, then any NP problem can be solved in “quasi-polynomial” time. The result was further refined by Arora et al. [4] one year later. Specifically, they proved that there exists an $\epsilon > 0$ such that no polynomial-time algorithm can approximate the size of the maximum clique within a factor of n^ϵ , unless $P = NP$. More recent developments along these lines can be found in [6, 19].

MCP was one of Richard Karp's original 21 problems shown to be NP-complete (see Table 2.1 where it is called 'CLIQUE', [23]) and it was already mentioned in Cook's paper introducing the theory of NP-complete problems [11]. When the edges are weighted we can define the Maximum Weight Clique Problem (MWCP) as the problem of finding the clique with the biggest weight. This does not necessarily need to be the clique with the highest cardinality. MWCP has important applications in such fields as computer vision, pattern recognition and robotics. More about MCP can be found in [8].

3.2.1 Greedy Algorithm for MCP

The greedy algorithm for MCP goes like this:

1. Let v_1, v_2, \dots, v_N be the permutation of the vertices (with N the amount of vertices).
2. Let $S_1 = \{v_1\}$ and $S_2 = \emptyset$ be sets of vertices.
3. for i is 2 to N :
 4. if v_i is connected to all vertices in S_1 , add v_i to S_1 .
 5. else add v_i to S_2 .

When this algorithm is finished the found solution, the proposed maximum clique, is in S_1 . The good thing about this greedy algorithm is that it will always find at least a local maximum clique. For every vertex that is not in S_1 it holds that it has been shown not to be connected to at least one vertex in S_1 . Therefore it will be impossible to find a vertex not in S_1 which can be added to the clique and still form a clique.

This greedy algorithm is exactly the same as the one proposed by Locatelli et al. [29] in their paper.

3.2.2 Used Graphs for MCP

Table 3.1 shows the parameters used to create the training set for the genetic program while trying to solve MCP. The column labelled with '#' shows the index number for the graph. The $|V|$ -column shows the amount of vertices. The EDGE-column is used for determining which vertices are connected. The C1- and C2-columns show the chance that a vertex is in a clique which is put in the graph. There can be zero, one or two of these predefined cliques. This can be seen in the table where cliques with chance 0 do not exist. All sets are used three times to create a graph, except for the set labelled with ' \emptyset ' which is only used once.

When creating a graph, first if the graph has any predefined cliques, for each vertex it is randomly determined whether it is in such a clique. A vertex can be in both cliques if there are two. Then all vertices in the same clique are connected with a randomly chosen edge. Finally, for all two vertices that are not in the same clique there is a chance, defined by EDGE, that they are connected with an edge.

#	V	EDGE	C1	C2	#	V	EDGE	C1	C2
0	10	0.50	0	0	6	15	0.50	0	0
1	10	0.70	0	0	7	15	0.70	0	0
2	10	0.70	0.40	0	8	15	0.70	0.30	0
3	10	0.85	0.30	0	9	15	0.85	0.40	0
4	10	0.85	0.40	0	10	15	0.85	0.30	0.30
5	10	0.85	0.40	0.40	11	15	0.85	0.33	0.35
12	20	0.50	0	0	18	25	0.50	0	0
13	20	0.70	0	0	19	25	0.70	0	0
14	20	0.70	0.30	0	20	25	0.70	0.25	0
15	20	0.85	0.40	0	21	25	0.85	0.35	0
16	20	0.85	0.30	0.30	22	25	0.85	0.25	0.25
17	20	0.85	0.33	0.35	23	25	0.85	0.27	0.30
24	35	0.50	0.15	0	29	50	0.50	0.15	0
25	35	0.70	0.18	0	30	50	0.70	0.18	0
26	35	0.70	0.15	0.18	31	50	0.70	0.15	0.18
27	35	0.85	0.20	0	32	50	0.85	0.20	0
28	35	0.85	0.18	0.20	33	50	0.85	0.18	0.20

Table 3.1: Parameters for the creation of the training set for MCP

3.3 Graph Colouring Problem

Consider a topographical map with regions on it, for instance a map of the world. In order to make it easy to distinguish the different countries it would be good if the countries would be coloured. Of course we want adjacent countries to have a different colour. When we would give every country its own, unique colour it is of course easy to fill in the map. Unfortunately, this will result in the fact that some of the colours will look very much alike. When they are used for two connected countries they are still hard to distinguish.

In order to solve this, we want to use as few different colours as possible. This will enable us to choose colours that are very different from each other. The question is what the minimum amount of colours is that we will need to colour a map. This can be formalized to a graph-problem where the countries are the vertices and two vertices are connected when two countries are adjacent. Every vertex should get a colour. Then, the Graph Colouring Problem (GCP) is defined as followed:

Considering a graph, is it possible to give every vertex a colour, where two connected vertices are not allowed to have the same colour and we can use at most ℓ different colours?

This is also obviously a decision problem which can be turned into an optimization problem by repeatedly asking the question with ever smaller values for ℓ . GCP was introduced by Brooks in 1941 [10]. Later on, it is proven to be NP-complete.

#	V	EDGE	#	V	EDGE	#	V	EDGE	#	V	EDGE
0	25	0.40	3	25	0.50	6	30	0.40	9	30	0.50
1	25	0.40	4	25	0.60	7	30	0.40	10	30	0.60
2	25	0.50	5	25	0.60	8	30	0.50	11	30	0.60
12	35	0.40	15	35	0.50	18	40	0.40	21	40	0.50
13	35	0.40	16	35	0.60	19	40	0.40	22	40	0.60
14	35	0.50	17	35	0.60	20	40	0.50	23	40	0.60
24	45	0.40	27	45	0.55	29	50	0.40	32	50	0.55
25	45	0.45	28	45	0.60	30	50	0.45	33	50	0.60
26	45	0.50				31	50	0.50			

Table 3.2: Parameters for the creation of the training set for GCP

3.3.1 Greedy Algorithm for GCP

The greedy algorithm for GCP goes like this:

1. Let v_1, v_2, \dots, v_N be the permutation of the vertices (with N the amount of vertices).
2. Let $S_1 = \{v_1\}, S_2 = \emptyset, S_3 = \emptyset, \dots, S_N = \emptyset$ be sets of vertices.
3. for i is 2 to N :
4. for j is 1 to N :
5. if v_i is not connected to any vertex in S_j or $S_j = \emptyset$
6. put v_i in S_j and break.

When this algorithm is finished all vertices with the same colour are in the same set. The amount of nonempty sets is the amount of colours needed. This amount does not necessarily have to be the minimum amount of colours in which it is possible to colour this graph. The amount of colours needed depends on the quality of the presented permutation. In any case the graph will be coloured correctly.

3.3.2 Used Graphs for GCP

Table 3.2 shows the parameters used to create the training set for the genetic program while trying to solve GCP. The column labelled with ‘#’ shows the index number for the graph. The |V|-column shows the number of vertices. The EDGE-column is used for determining which vertices are connected. All sets are used three times to create a graph, except for the set labelled with ‘0’ which is only used once.

For this problem there was no special way to create the graphs required as long as they had the right size and density. When colouring small graphs using the above described greedy algorithm, it hardly ever occurs that the amount of colours needed is very high or very low. Therefore it is better to use bigger graphs. The same is true when we consider both high- and low-density graphs and therefore the chance that there is an edge should be in the range from 0.4 to 0.6.

3.4 Hamilton Cycle Problem

A path is a series of vertices where two successive vertices are connected by an edge. A cycle is a path where the first vertex is the same as the last vertex. A Hamilton cycle is a cycle which goes through all the vertices once. While searching for a Hamilton cycle the length of the edges is not important. Then we could define the Hamilton Cycle Problem (HCP) as followed:

Considering a graph, is it possible to find a cycle which goes through all the vertices exactly once?

A special situation is there where we use a directed graph. Both the directed and undirected Hamiltonian cycle problems were two of Karp's 21 NP-complete problems (see Table 2.1, [23]).

3.4.1 Greedy Algorithm for HCP

The greedy algorithm for HCP goes like this:

1. Let v_1, v_2, \dots, v_N be the permutation of the vertices (with N the amount of vertices).
2. Let $S_1 = \{v_1\}, S_2 = \emptyset, S_3 = \emptyset, \dots, S_N = \emptyset$ be sets of vertices.
3. Let $x = 0$ be a counter.
4. if v_1 is connected with an edge to v_N , set x to be 1.
5. for i is 2 to N :
6. if v_i is connected with an edge to a vertex in S_{i-1} , raise x with 1.
7. put v_i in S_i .

A series of vertices is a Hamilton cycle when the series is a permutation of all the vertices and all successive vertices are connected with an edge. Keep in mind that the last vertex also needs to be connected to the first. Since any individual always returns a permutation this does not need to be tested. The only thing we need to do is count the amount of successive vertices that are connected. At the end of the algorithm this amount can be found in x . When $x = N$ we have found a Hamilton cycle. If not, the difference between x and N tells something about the quality of the found solution.

3.4.2 Used Graphs for HCP

Table 3.3 shows the parameters used to create the training set for the genetic program while trying to solve HCP. The column labelled with '#' shows the index number for the graph. The $|V|$ -column shows the amount of vertices. The EDGE-column is used for determining which vertices are connected. All sets are used three times to create a graph, except for the set labelled with ' \emptyset ' which is only used once.

#	V	EDGE	#	V	EDGE	#	V	EDGE	#	V	EDGE
0	10	0.40	3	10	0.60	6	15	0.40	9	15	0.60
1	10	0.50	4	10	0.70	7	15	0.50	10	15	0.70
2	10	0.60	5	10	0.80	8	15	0.60	11	15	0.80
12	20	0.40	15	20	0.60	18	25	0.40	21	25	0.60
13	20	0.50	16	20	0.70	19	25	0.50	22	25	0.70
14	20	0.60	17	20	0.80	20	25	0.60	23	25	0.80
24	35	0.40	27	35	0.70	29	50	0.40	32	50	0.70
25	35	0.50	28	35	0.80	30	50	0.50	33	50	0.80
26	35	0.60				31	50	0.60			

Table 3.3: Parameters for the creation of the training set for HCP

For this problem first random graphs were created by using the parameters in the table. Then a Hamilton cycle was added to be sure that there was at least one in it. This was done by creating a random permutation of the vertices and made sure that every consecutive pair of vertices was connected with an edge.

3.5 Minimum Spanning Tree

Consider a cable company that is in charge of providing cable television for everyone. One of their projects is to connect a sparsely populated region which contains only a set of farms spread out criss-cross all over the place most of the time miles apart of each other. This cable company is of course highly interested in the way to connected every farm to the cable while using as less cable as possible.

This problem can be formalized using graph theory. Consider a weighted graph where every vertex represents a farm and the weight on the edges stand for the length of the cable that has to be laid to connect them. We define a spanning tree of that graph as a subgraph which is a tree and connects all the vertices together. A minimum spanning tree (MST) or minimum weight spanning tree is then the spanning tree with the lowest total weight of all the edges. When a graph is unconnected we speak of a minimum spanning forest.

MST is solvable in polynomial time. The first algorithm which was able to find a minimum spanning tree was developed by the Czech scientist Otakar Borůvka in 1926 [9] and is nowadays called Borůvka's algorithm. It can be shown to run in time $\mathcal{O}(|E| \log |V|)$ with $|E|$ the amount of edges and $|V|$ the amount of vertices. Today we mostly use Kruskal's algorithm [27] which runs in $\mathcal{O}(|E| \log |E|)$ or $\mathcal{O}(|E| \log |V|)$ or we use Prim's algorithm [33] which runs in $\mathcal{O}(|E| \log |V|)$ or $\mathcal{O}(|E| + |V| \log |V|)$. The difference in runtime depends on which data structure is used. Prim's algorithm is advised for very dense graphs when $|E|$ is $\Omega(|V| \log |V|)$.

#	V	EDGE	#	V	EDGE	#	V	EDGE	#	V	EDGE
0	10	0.40	3	10	0.60	6	15	0.40	9	15	0.60
1	10	0.50	4	10	0.70	7	15	0.50	10	15	0.70
2	10	0.60	5	10	0.80	8	15	0.60	11	15	0.80
12	20	0.40	15	20	0.60	18	25	0.40	21	25	0.60
13	20	0.50	16	20	0.70	19	25	0.50	22	25	0.70
14	20	0.60	17	20	0.80	20	25	0.60	23	25	0.80
24	35	0.40	27	35	0.70	29	50	0.40	32	50	0.70
25	35	0.50	28	35	0.80	30	50	0.50	33	50	0.80
26	35	0.60				31	50	0.60			

Table 3.4: Parameters for the creation of the training set for MST

3.5.1 Greedy Algorithm for MST

The greedy algorithm for MST goes like this:

1. Let v_1, v_2, \dots, v_N be the permutation of the vertices (with N the amount of vertices).
2. Let $S_1 = \{v_1\}$ be a set of vertices.
3. Let $E = \emptyset$ be a set of edges and $x = 0$ be a counter.
4. for i is 2 to N :
 5. let e be the edge with lowest weight connecting v_i to S_1 .
 6. if e exists, raise x with weight of e and add e to E .
 7. else raise x with a penalty of a certain size.
 8. put v_i in S_1 .

When this algorithm is finished, the found spanning forest is in (S_1, E) . The value of x determines the quality of the found spanning forest and it is used to calculate the fitness of the individual. The size of the penalty determines how important it is to find vertices that are connected. In this case with the weight of the edges ranging from 1 to 100 the penalty was set to be 1000.

3.5.2 Used Graphs for MST

Table 3.4 shows the parameters used to create the training set for the genetic program while trying to solve MST. The column labelled with ‘#’ shows the index number for the graph. The |V|-column shows the amount of vertices. The EDGE-column is used for determining which vertices are connected. All sets are used three times to create a graph, except for the set labelled with ‘0’ which is only used once. For this problem there was nothing needed to add to the graphs to ensure the program would perform well. Therefore a wide variety of nodes and densities for the graphs was chosen.

3.6 Bipartite Graph Problem

A graph is bipartite when all its vertices can be divided into two disjoint sets where two vertices of the same set are not connected by an edge. These types of graphs are commonly used for matching problems. For instance the heterosexual section of a coupling agency who wishes to make as many matches as possible. In comparison to a graph every person represents a vertex. The vertices are divided into two groups (male and female) so every vertex is in one group, no more and no less. A possible match is an edge between the two people and hence all edges go from one group to the other.

When we consider a graph we might want to know whether it is bipartite or not. This can be easily done with a slightly modified Breadth First Search (BFS). For more information about BFS check *Introduction to Algorithms* of Thomas H. Cormen et al [12]. In BFS every vertex that is evaluated is at a certain depth from the first evaluated node. If that depth is odd all adjacent vertices which are already evaluated must be even and vice versa. If not, then the graph is not bipartite. BFS is known to have running time $\mathcal{O}(|V| + |E|)$ where $|V|$ is the amount of vertices and $|E|$ is the amount of edges. Since the modification to check if the graph is bipartite does not change this, this algorithm has also running time $\mathcal{O}(|V| + |E|)$ and therefore we can say the question is solvable in polynomial time.

3.6.1 Greedy Algorithm for BGP

The greedy algorithm for BGP goes like this:

1. Let v_1, v_2, \dots, v_N be the permutation of the vertices (with N the amount of vertices).
2. Let $S_1 = \{v_1\}$, $S_2 = \emptyset$ and $S_3 = \emptyset$ be sets of vertices.
3. for i is 2 to N :
4. if v_i is not connected to a vertex in S_1 , put v_i in S_1 .
5. else if v_i is not connected to a vertex in S_2 , put v_i in S_2 .
6. else put v_i in S_3 .

The algorithm tries to divide the vertices into two sets (S_1 and S_2) where vertices in the same set are not connected. Vertices that cannot be placed in either of the two sets will be placed in S_3 . When the algorithm is finished, the size of S_3 tells something about the quality of the found solution.

If you use this greedy algorithm and the graph is not bipartite it does not matter which permutation you present. Since it is not bipartite you will not be able to correctly divide the vertices into two sets. Therefore testing your individuals on graphs that are not bipartite does not help to distinguish good individuals from bad ones. So in order to make things efficient, the individuals will only be tested on bipartite graphs. To be precise, we can say that the individuals are not tested on their ability to tell whether a graph is bipartite but on their ability to show that a graph is bipartite. In practice, on the other hand, when we have found an individual that does that perfectly it can be used easily to distinguish bipartite graphs from non-bipartite ones.

#	V	EDGE	RATIO	#	V	EDGE	RATIO
0	25	0.30	50/50	6	30	0.30	50/50
1	25	0.30	60/40	7	30	0.30	60/40
2	25	0.50	50/50	8	30	0.50	50/50
3	25	0.50	60/40	9	30	0.50	60/40
4	25	0.70	50/50	10	30	0.70	50/50
5	25	0.70	60/40	11	30	0.70	60/50
12	35	0.30	50/50	18	40	0.30	50/50
13	35	0.30	60/40	19	40	0.30	60/40
14	35	0.50	50/50	20	40	0.50	50/50
15	35	0.50	60/40	21	40	0.50	60/40
16	35	0.70	50/50	22	40	0.70	50/50
17	35	0.70	60/40	23	40	0.70	60/50
24	45	0.30	50/50	29	50	0.30	50/50
25	45	0.40	60/40	30	50	0.40	60/40
26	45	0.50	50/50	31	50	0.50	50/50
27	45	0.60	60/40	32	50	0.60	60/40
28	45	0.70	50/50	33	50	0.70	50/50

Table 3.5: Parameters for the creation of the training set for BGP

3.6.2 Used Graphs for BGP

Table 3.5 shows the parameters used to create the training set for the genetic program while trying to solve BGP. The column labelled with ‘#’ shows the index number for the graph, the |V|-column shows the amount of vertices, the EDGE-column is used for determining which vertices are connected and the RATIO-column shows the ratio between the two groups of vertices. All sets are used three times to create a graph, except for the set labelled with ‘0’ which is only used once.

As seen before, for this problem we need to test the individuals on graphs that are bipartite. To make a bipartite graph, first, the vertices are divided into two groups. Every vertex had either a 50% or a 60% chance to be in group ‘one’, otherwise it would be in group ‘two’. Then for every pair of vertices it was certified first whether they were in different groups and if that was the case there was a chance, which is determined by the number in the EDGE-column, that there was an edge between the two.

3.7 Connected Component Problem

An undirected graph is said to be connected when there is a path from every vertex to every other vertex. A maximally connected component of a graph is a sub graph for which it is impossible to add a vertex and still have a connected graph. The connected components of a graph is the set of maximally connected components.

Finding all the maximally connected components can be done with Breadth First Search (BSF, [12]) which is known to have running time $\mathcal{O}(|V| + |E|)$ where $|V|$ is the amount of vertices and $|E|$ is the amount of edges. Therefore the Connected Components Problem (CCP), that is finding all the connected components of a graph, is solvable in polynomial time.

3.7.1 Greedy Algorithm for CCP

The greedy algorithm for CCP goes like this:

1. Let v_1, v_2, \dots, v_N be the permutation of the vertices (with N the amount of vertices).
2. Let $S_1 = \{v_1\}, S_2 = \emptyset, S_3 = \emptyset, \dots, S_N = \emptyset$ be sets of vertices.
3. Let a be the index of the actual set (S_a), initially 1.
4. for i is 2 to N :
 5. if v_i is not connected to a vertex in S_a , raise a with 1.
 6. put v_i in S_a .

When the algorithm is finished all the found sets of connected components are stored in the different sets S_1, S_2, \dots, S_N . These sets will always contain connected sub graphs, no matter what permutation is presented. They do not necessarily need to be maximally connected. The quality of an individual depends on the amount of nonempty sets there are at the end of the algorithm.

3.7.2 Used Graphs for CCP

Table 3.6 shows the parameters used to create the training set for the genetic program while trying to solve CCP. The column labelled with ‘#’ shows the index number for the graph, the $|V|$ -column shows the amount of vertices, the EDGE-column is used for determining which vertices are connected and the RATIO-column shows the ratio between the several components. All sets are used three times to create a graph, except for the set labelled with ‘0’ which is only used once.

For this problem the individuals needed to be tested on graphs that have different components. To create a graph with several components, first, the vertices were divided into several, maximal four, groups, each corresponding with a component. The ratio between those groups is shown in the corresponding column. Again here they can be seen as the chance that a certain vertex is in that component. Then for every pair of vertices, first it was certified whether they were in the same component and if that was the case there is a chance, which is determined by the number in the EDGE-column, that there is an edge between the two.

#	V	EDGE	RATIO	#	V	EDGE	RATIO
0	10	0.50	1.0	6	15	0.50	1.0
1	10	0.70	1.0	7	15	0.70	1.0
2	10	0.70	0.50/0.50	8	15	0.70	0.50/0.50
3	10	0.85	0.50/0.50	9	15	0.85	0.50/0.50
4	10	0.85	0.65/0.35	10	15	0.85	0.65/0.35
5	10	0.85	0.34/0.33/0.33	11	15	0.85	0.34/0.33/0.33
12	20	0.50	1.0	18	25	0.50	1.0
13	20	0.70	1.0	19	25	0.70	1.0
14	20	0.70	0.50/0.50	20	25	0.85	0.50/0.50
15	20	0.85	0.50/0.50	21	25	0.85	0.65/0.35
16	20	0.85	0.65/0.35	22	25	0.85	0.34/0.33/0.33
17	20	0.85	0.34/0.33/0.33	23	25	0.85	0.40/0.25/0.20/0.15
24	35	0.85	0.50/0.50	29	50	0.85	0.50/0.50
25	35	0.85	0.65/0.35	30	50	0.85	0.65/0.35
26	35	0.85	0.34/0.33/0.33	31	50	0.85	0.34/0.33/0.33
27	35	0.85	0.25/0.25/0.25/0.25	32	50	0.85	0.25/0.25/0.25/0.25
28	35	0.85	0.40/0.25/0.20/0.15	33	50	0.85	0.40/0.25/0.20/0.15

Table 3.6: Parameters for the creation of the training set for CCP

Chapter 4

Setup of the Program

While developing Genetic Programming, Cramer [13] and Koza [25], suggested that a tree structure should be used as the program representation in a genome. In designing the individuals for this thesis their suggestions were adopted. The most important element of each individual is its algorithm which has the proposed tree structure. There are two main advantages of using a tree structure. First of all its working is very intuitive and can easily be comprehended easily, due to its comparison with a parse tree which is used while compiling a computer program. For more about parse trees the reader is recommended to read Aho et al. [2]. The second main advantage is the fact that this structure makes it very easy to create syntactically correct algorithms which stay correct after crossover or mutation.

4.1 Working of an Individual

Before taking a look at the working of the genetic program, we first take a close look at the individuals. A better understanding of the individuals will help clarify some of the procedures in the program.

4.1.1 The Function Set

Every individual is capable of executing an algorithm. This algorithm is stored in a tree structure which is created using the grammar described in Table 4.1. When using this grammar to create a tree one starts with the `Begin`-element which can only be replaced by '`List_of_Stats`'. From here we can choose a variety of replacements. Some of them cannot be replaced any more. These are called the terminal elements. In Table 4.1 the elements in the replacements which are labelled with 6, 7, 20, 21, 22 and 40 are the terminal elements. We are finished when there are no more replacements possible.

As an example of how these replacements form, an algorithm is shown in Table 4.2. The numbers in this table labelling the replacements correspond to the labels in Table 4.1. Every replacement can be seen as adding a node in the tree at that position. The tree that corresponds to the replacements as described in Table 4.2 can be seen in Figure 4.1. Here the numbers in the `List_of_Stats`-elements are labels which label the entire statement. The numbers in the other

```

Begin -->
( 1) List_of_Stats

List_of_Stats -->
( 2) Statement;
( 3) Statement; List_of_Stats

Statement -->
( 4) SWAP(Vertex,Vertex) // Swaps the two vertices in the permutation
( 5) PUSH(Vertex,Vertex) // Puts vertex 1 at place of 2 and pushes vertices in between
( 6) NEXT_G(Pointer) // Let pointer point to next vertex (acc. to labels) in graph
( 7) NEXT_P(Pointer) // Let pointer point to next vertex in permutation
( 8) SET(Pointer,Vertex) // Set pointer to point at given vertex
( 9) IF(Test,List_of_Stats) // IF-statement
(10) WHILE(Test,List_of_Stats) // WHILE-statement

Test -->
(11) EXIST(Vertex,Vertex) // Is there an edge between the vertices?
(12) PREV(Vertex,Vertex) // Is first vertex previous to the other in the permutation?
(13) EQNODE(Vertex,Vertex) // Are the two vertices the same?
(14) EQUAL(Number,Number) // Are the two numbers equal to each other?
(15) LESS(Number,Number) // Is the first number smaller than the second?
(16) GREATER(Number,Number) // Is the first number greater than the second?
(17) NOT(Test) // The logical NOT for the input
(18) AND(Test,Test) // The logical AND for the inputs
(19) OR(Test,Test) // The logical OR for the inputs

Number -->
(20) INTEGER // A random integer
(21) NODES // Total amount of vertices
(22) EDGES // Total amount of edges
(23) DEGREE(Vertex) // Degree of the vertex
(24) DEG_HINB(Vertex) // Highest degree of all the neighbours of the vertex
(25) DEG_LONB(Vertex) // Lowest degree of all the neighbours of the vertex
(26) DIS_CLONB(Vertex) // Distance to the closest neighbour
(27) DIS_FARNB(Vertex) // Distance to the farthest neighbour
(28) COMNB(Vertex,Vertex) // Amount of common neighbours
(29) DEG_HICONB(Vertex,Vertex) // Highest degree of all the common neighbours
(30) DEG_LOCONB(Vertex,Vertex) // Lowest degree of all the common neighbours
(31) PATH(Vertex,Vertex) // Length of path between both vertices
(32) DIS_SHOCONB(Vertex,Vertex) // Distance of the shortest path through a common neighbour
(33) DIS_LONCONB(Vertex,Vertex) // Distance of the longest path through a common neighbour
(34) PLUS(Number,Number) // Both numbers added to each other
(35) MIN(Number,Number) // One number subtracted from the other
(36) MULTI(Number,Number) // Both numbers multiplied with each other
(37) DIVIDE(Number,Number) // One number divided by the other (division by 0 gives 0)
(38) MAX(Number,Number) // The biggest of both numbers
(39) MIN(Number,Number) // The smallest of both numbers

Vertex -->
(40) POINTER(Pointer) // The vertex to which the pointer points
(41) FIRSNB(Vertex) // The neighbour which is first in the permutation
(42) LASTNB(Vertex) // The neighbour which is last in the permutation
(43) PREVNB(Vertex) // The neighbour which is the previous in the permutation
(44) NEXTNB(Vertex) // The neighbour which is the next in the permutation
(45) HINB(Vertex) // The neighbour which has the highest degree
(46) LONB(Vertex) // The neighbour which has the lowest degree
(47) CLONB(Vertex) // The closest neighbour
(48) FARNB(Vertex) // The farthest neighbour
(49) HICONB(Vertex,Vertex) // The common neighbour which has the highest degree
(50) LOCONB(Vertex,Vertex) // The common neighbour which has the lowest degree
(51) SHOCONB(Vertex,Vertex) // The common neighbour that creates the shortest path
(52) FARCONB(Vertex,Vertex) // The common neighbour that creates the longest path

```

Note: Pointer stands for a pointer to a vertex. They are chosen randomly.

Table 4.1: The Grammar used

```

Begin
( 1): List_of_Stats
( 3): Statement; List_of_Stats
( 8): SET(Pointer,Vertex); List_of_Stats
(45): SET(Pointer,HINB(Vertex)); List_of_Stats
( 3): SET(Pointer,HINB(Vertex));
Statement; List_of_Stats
(10): SET(Pointer,HINB(Vertex));
WHILE(Test,List_of_Stats); List_of_Stats
(15): SET(Pointer,HINB(Vertex));
WHILE(LESS(Number,Number),List_of_Stats); List_of_Stats
(23): SET(Pointer,HINB(Vertex));
WHILE(LESS(DEGREE(Vertex),Number),List_of_Stats); List_of_Stats
(24): SET(Pointer,HINB(Vertex));
WHILE(LESS(DEGREE(Vertex),DEG_HINB(Vertex)),List_of_Stats); List_of_Stats
( 3): SET(Pointer,HINB(Vertex));
WHILE(LESS(DEGREE(Vertex),DEG_HINB(Vertex)),
Statement; List_of_Stats); List_of_Stats
( 8): SET(Pointer,HINB(Vertex));
WHILE(LESS(DEGREE(Vertex),DEG_HINB(Vertex)),
SET(Pointer,Vertex); List_of_Stats); List_of_Stats
( 2): SET(Pointer,HINB(Vertex));
WHILE(LESS(DEGREE(Vertex),DEG_HINB(Vertex)),
SET(Pointer,Vertex);
Statement;); List_of_Stats
( 7): SET(Pointer,HINB(Vertex));
WHILE(LESS(DEGREE(Vertex),DEG_HINB(Vertex)),
SET(Pointer,Vertex);
NEXT_P(Pointer)); List_of_Stats
( 2): SET(Pointer,HINB(Vertex));
WHILE(LESS(DEGREE(Vertex),DEG_HINB(Vertex)),
SET(Pointer,Vertex);
NEXT_P(Pointer));
Statement;
( 4): SET(Pointer,HINB(Vertex));
WHILE(LESS(DEGREE(Vertex),DEG_HINB(Vertex)),
SET(Pointer,Vertex);
NEXT_P(Pointer));
SWAP(Vertex,Vertex);
(40, several times): SET(Pointer,HINB(PONTER(Pointer)));
WHILE(LESS(DEGREE(PONTER(Pointer)),DEG_HINB(PONTER(Pointer))),
SET(Pointer,PONTER(Pointer));
NEXT_P(Pointer));
SWAP(PONTER(Pointer),PONTER(Pointer));

```

Table 4.2: Example of how to use the grammar to create an algorithm.

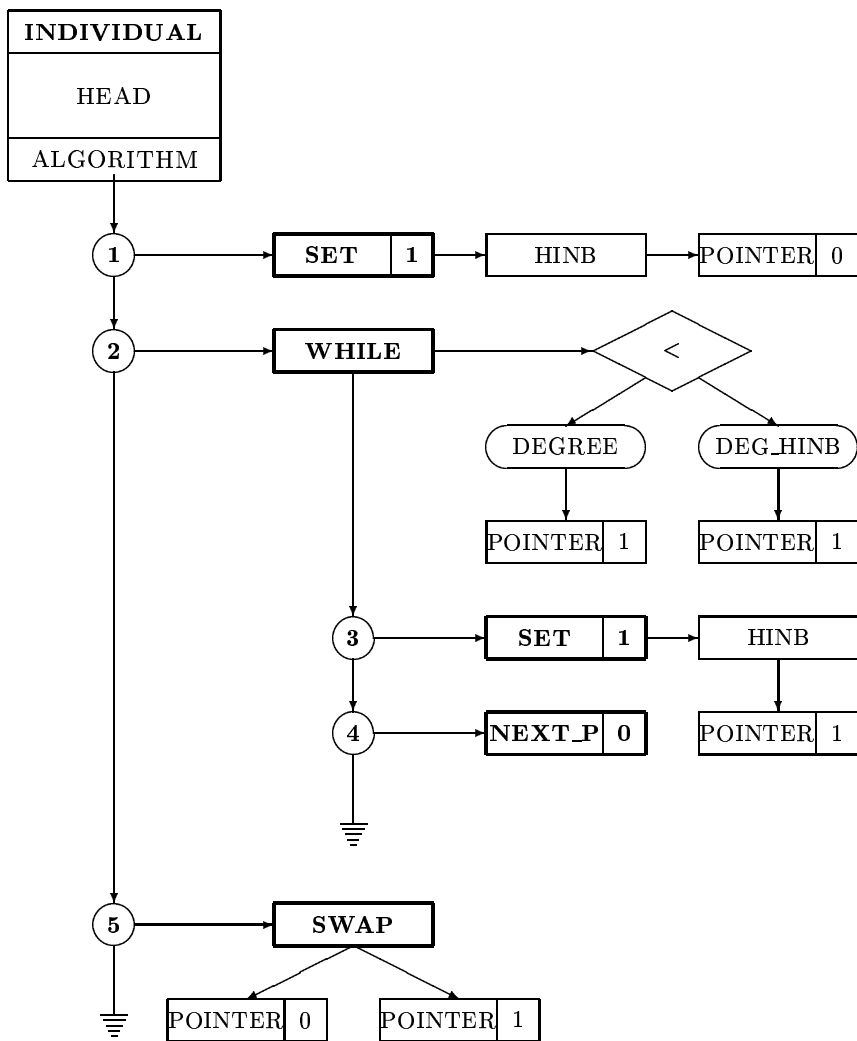
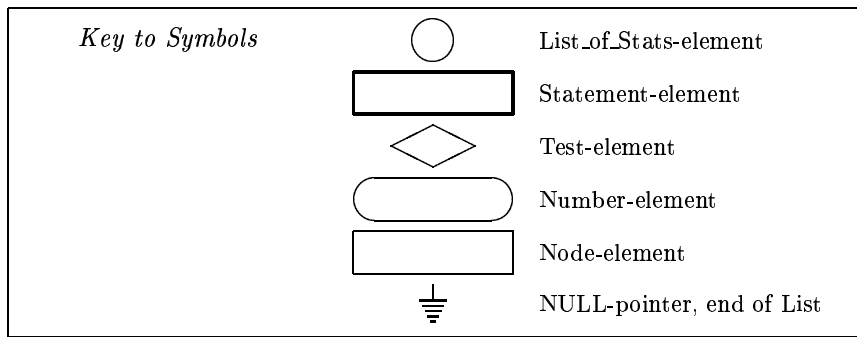


Figure 4.1: Example of an individual in the genetic program

```

1  Pointer1 = HighestNeighbour(Pointer0);
2  WHILE (Degree(Pointer1) < Degree_HighestNeighbour(Pointer1))
   {
3      Pointer1 = HighestNeighbour(Pointer1);
4      Pointer0 = Next_in_Permutation();
   }
5  SWAP(Pointer0, Pointer1);

```

Table 4.3: Pseudo-code of the algorithm described in Figure 4.1. This algorithm does the following: At first it sets `Pointer1` to point to the neighbour with the highest degree (cardinality) of `Pointer0`. Then, the `WHILE`-loop checks if `Pointer1` has a neighbour with a higher degree than itself and while this is true, it will point to the neighbour with the highest degree. In the meantime `Pointer0` will make steps through the permutation. In the final step, the vertices to which `Pointer0` and `Pointer1` point will be swapped. So, line 5 is the only line with a statement that actually changes the permutation. If you examine this algorithm it will probably not score very good, which will result in a low fitness value.

elements are the indices of the pointers needed. The pseudo code corresponding to this individual is shown in Table 4.3. Here the labels of the statements correspond to the labels of the `List_of_Stats`-elements in Figure 4.1.

The spine of the tree is formed with the use of the `List_of_Stats`-elements. They can create a chain of any number of nodes each holding one `Statement`-element. Any `IF`- or `WHILE`-statement can itself hold a chain of `List_of_Stats`-elements which form the body of that conditional statement or, as we can say, the nested statements.

Besides the `IF`- and `WHILE`-statements there are also statements which perform actions. The first group contains the `SWAP`- and `PUSH`-statements and directly influence the permutation. The other group contains the `NEXT_G`-, `NEXT_P`- and `SET`-statement which change the value of the pointers. These can be seen as variables and are discussed more detailed in Section 4.1.2.

Every statement might need one or more arguments which are also nodes of the tree and can be several different types. `Test`-elements will return a Boolean value; `Number`-elements will return a real number and `Vertex`-elements return a vertex of the input graph. These elements themselves might need one or two arguments. For instance the `Number`-element `DEGREE` needs as argument a `Vertex`-element. When being called it returns a number which is the cardinality of the vertex provided by the `Vertex`-element.

4.1.2 Processing a Graph

To execute its algorithm, an individual needs more than the tree structure where the algorithm is stored in. This is the existence of some sort of memory and it serves two purposes. First of all it is used to store the needed variables. These are pointers to vertices in the graph. The individual may need to use a certain amount of pointers which is stored in an integer called `AMOUNT_INDICES`. These pointers can be seen as an array with length equal to `AMOUNT_INDICES`. Every element of the array is a pointer and its value is the index of the vertex it points at. Initially they all point to vertex ' v_0 '.

The second thing it needs in its memory is the current permutation. It is a permutation of all the vertices and when the algorithm terminates its present state is the answer generated by the algorithm. This is an array with length V with V the amount of vertices in the graph. Initially it contains the permutation $1, 2, \dots, V$. Throughout the entire execution of the algorithm this array contains the current state of the permutation, which can be altered by several `Statement`-elements.

Every time an individual needs to execute its algorithm it gets a graph as input. After initialising its pointers and its permutation it can start to execute the algorithm by executing the tree node by node. Some statements will alter the variable which can later on be used by other statements. Some statements alter the permutation by swapping or pushing elements in it. Some nodes in the tree might not be executed. For instance when the `Test`-element of an `IF`-statement returns *false*, the nodes in its body are not executed. After all nodes in the tree are executed (when needed) the individual has its answer, as in its permutation, ready to present to the greedy algorithm that will use it further on.

4.2 Energy Bound Genetic Programming

In the previous section we have seen the structure of an individual and the way in which it processes a graph. In the program is made use of ‘Energy Bound Genetic Programming’ or ‘Energy Bound GP’. The ideas behind this are first explained in an intuitive way. Later on, it is explained how these ideas are implemented in the program and they are formalised more.

4.2.1 Explosive Intron Growth

When we look carefully at the grammar described in table 4.1, we can see that it is easy to construct an infinitely big individual. Consider, for instance, the step where we replace a `List_of_Stats`-element. If we would choose to replace it with “`Statement; List_of_Stats`” all that happens is that the tree grows with a `Statement`-element while the `List_of_Stats`-element remains. This step could be repeated an infinite number of times letting the tree grow infinitely big. A similar procedure can be created with `Test`-, `Number`- and `Node`-elements. Although this allows to create every potential algorithm, it has to be taken into account that the capacity of the computer used for the experiment provides a practical boundary. Crossing this boundary will terminate the program too early. This could already happen at initialization. To prevent the individuals from expanding to enormous size we must put a halt to them. Koza [26] suggested setting a maximum depth for an individual during initialisation. When this depth is reached initialization is only allowed to choose terminals. The depth you choose depends on both your grammar and your computer system. It must be balanced between good computer performance and powerful individuals.

By making use of this idea, it is possible to initialise the program. Unfortunately, many genetic programs encounter this problem later on in their runs again. Consider the crossover mechanism where we swap pieces of DNA (in this case algorithm) between individuals. When we swap a small piece from one in-

dividual with a big piece from the other we create one very big and one very small offspring. Most of the time, the small individual is not so good in solving the problem, thus it gets a low fitness value and therefore cannot reproduce a lot. On the other hand, the bigger child has bigger chance of a better fitness and therefore bigger chance of reproducing children. These children themselves have a good chance to become even bigger. This can go on and on resulting in excessive growth of the individuals.

Most of the time, this rapid growth mainly results in an explosive growth of the occurrence of introns. Introns are pieces of ‘useless’ information e.g., the body of an IF-statement where the test will always return false. The benefit of introns is the fact that they protect the good pieces of code to be cut in halves at crossover. The more introns there are the bigger the chance is that the randomly chosen point where the cut for the crossover will be, is in such an intron. This leaves the functional code in one piece. Introns occur in both natural DNA [3] and in the DNA of a Genetic Program [38]. The big difference is that at the end of a run of a genetic program the number of introns tends to grow exponentially, undermining the evolutionary process [37] where the amount of introns in natural DNA remains the same.

4.2.2 Energy Bound GP, a Practical Solution

In his Master’s Thesis Nabuurs proposed in 2004 a reason and a solution for this difference in behaviour for introns in natural DNA and in GP [32]. Nabuurs claimed that this explosive growth does not occur in natural individuals and their DNA simply because they lack the resources to make it happen. He called these resources ‘energy’. Every cell of a natural creature contains DNA so every time a cell is created, this DNA needs to be copied and this costs energy. As long as the chromosomes of the DNA are useful for the individual it is worth spending the energy on copying them. For a certain amount of introns this is also true, because they protect the useful parts. At a certain point, though, more introns do not significantly contribute to more protection and, with nature’s energy limited, it is better for the individual to not spend energy on it. When we compare this with Genetic Programming, we see that the amount of available computer resources, the equivalence for nature’s energy, is also limited for GP. Although this limitation exists, a genetic program will not notice it and can let its individuals evolve the way it wants until the moment the program crashes due to a lack of them. Nabuurs suggested that by adding an analogy for this energy in the program one could limit the intron growth. He called this ‘Energy Bound Genetic Programming’ and found that it performed better than a Koza-style algorithm [26].

4.2.3 Principles of Energy Bound GP

In order to redeem this lack of analogy for energy in ordinary GP, Energy Bound Genetic Programming adds a certain, total amount of energy to the genetic program. It obeys the first law of thermodynamics which says that the total amount of energy will always remain constant. Every piece of energy can be assigned to a specific individual. When it is not assigned it is part of the available energy.

When being created, each individual gets a certain amount of energy which is the same for every individual. This energy will be deducted from the amount of available energy, and therefore an individual can only be created when there is enough energy available. This energy is used for the actual creation of the algorithm. The bigger this algorithm is, the more energy it requires. If an individual does not have enough energy, it cannot create its algorithm and therefore cannot exist. This limits the size to which an individual can grow.

Then, the energy an individual has left, is used to evaluate the fitness cases. As time passes by, this energy will be released to the population. Once an individual has ran out of energy it dies. An individual that needs little energy to evaluate the fitness cases, will live much longer than an individual that wastes everything at once.

Every time there is enough energy available, a new individual will be created. Its parents are chosen from the current population using tournament selection (see Section 4.3.4). An individual that lives longer therefore has more chance of being selected to participate in such a tournament than an individual that has a shorter life. On the other hand, when the short living individual uses all the energy it consumes wisely and performs very well, it will have a much better fitness value than an individual that spends almost no energy but, as a result, performs poorly. Therefore the long living individual will have a better chance to participate in a tournament, but the good performing individual will have a better chance to actually win this tournament and become a parent.

All this will prevent the individuals from growing exponentially. Although the population size is not fixed, it will not be able for the population to grow to enormous size because of the limit in the total energy. Therefore Energy Bound Genetic Programming is an easy way of controlling the size of your genetic program and its usage of computer resources.

4.2.4 A Flaw in the Theory

Energy Bound GP works better than a Koza-style GP [26], as shown by Nabuurs [32]. In addition to the fact that it performs better it also prevents the enormous growth in the amount of introns. The latter is good, as we have seen, because this explosive growth can result in a crash of the program since there is a limit on the amount of available computer memory.

Unfortunately, there is something odd about the working of Energy Bound GP. Nabuurs claimed that he had invented it to prevent the explosive intron growth. Introns are pieces of code that do not do anything. When we, for instance, consider an IF-statement where the `Test`-element will always return *false*, its body will never be executed and therefore it is considered to be an intron. Now, if introns do not do anything when the algorithm is processing a graph, why then, does Energy Bound GP make use of the consumption rate of an individual?

When an individual processes a graph, the amount of introns does not have any effect on either the performance of an individual or on the amount of nodes evaluated in the algorithm-tree. The first determines its fitness while the latter determines its energy consumption rate and this, in its turn, determines the life expectancy of the individual. The chance that an individual is allowed to produce offspring is caused by both the fitness and the life expectancy of an individual. Thus, the amount of introns an individual has does not have any in-

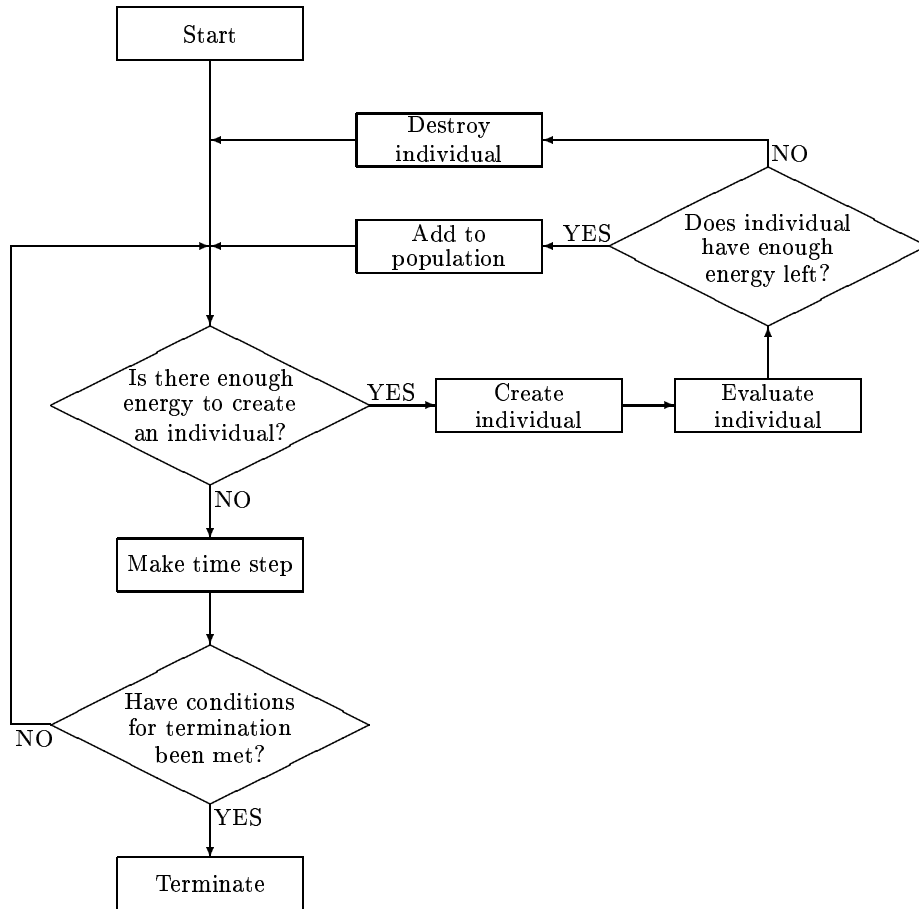


Figure 4.2: General working of Energy Bound GP.

fluence on the chance it is selected for reproduction. When the selection method is not influenced by the amount of introns it cannot control the amount of introns.

In conclusion, an important part of Energy Bound Genetic Programming is used to control the behaviour of the individual while introns, on the other hand, do not have anything to do with the behaviour of an individual. Therefore it looks like Energy Bound GP is a very laborious way to put a hold on explosive intron growth. Maybe this part is just useless, or maybe it does something else which in an unknown way improves the genetic program. It is not clear yet and more research on the subject is recommended. In the mean time it is used as it is.

4.3 Working of the Genetic Program

We have seen how the algorithm of an individual is constructed and how it processes a graph. We have also seen the principles of Energy Bound Genetic Programming. Now we can have a better look at the exact working of the program. The schedule for the general working of an Energy Bound GP is shown in Figure 4.2. The central question is: ‘Is there enough energy to create an individual?’. The precise meaning of this question is described in Section 4.3.1. When there is enough energy, a new individual will be created and we will return to this question.

The way in which an individual is created depends first of all at what point in the program this is done. At initialisation an individual will be created randomly, which is described in Section 4.3.2. During the run a new individual will be created using mutation or crossover. First, one or two parents need to be selected. The selection procedure is described in Section 4.3.4. Then the new individual is created using crossover or mutation (described in Section 4.3.5). After an individual has been created it will be evaluated immediately and checked for viability. This entire procedure is the same for every individual and is described in Section 4.3.3.

When there is not enough energy the program will make a time step. What happens there is explained in Section 4.3.6. Then the program checks if the termination conditions have been met and this is explained in Section 4.3.7. If they are met, the program terminates and otherwise it returns to the central question about whether there is enough energy to create an individual.

4.3.1 Is there enough Energy?

When we start translating the principles for Energy Bound GP mentioned previously in practical computer code, we first need a constant integer to hold the total amount of energy. This is called E_{total} . Every time an individual is created it gets a fixed amount of energy, again a constant integer and this time called E_{birth} . This amount is subtracted from the total amount of energy which leaves an amount of available energy, called $E_{available}$. The program will create individuals until there is not enough energy left.

Then, as time passes by (see Section 4.3.6) the individuals will release energy which is returned to $E_{available}$. Every time when there is enough energy left ($E_{available} > E_{birth}$), a new individual will be created. In this way the population changes gradually and generations are mixed throughout the population. This looks quite like the steady-state genetic programs described in Section 2.1.1 and [24], but there is one important difference. At a steady-state genetic program the age of an individual is decided by chance. It depends on whether an individual is chosen at all for a tournament and the quality of the chosen competitors. At Energy Bound GP the age an individual can reach is solely determined by its initial energy rate and its energy usage.

4.3.2 Initialisation

An individual consists of a header with general information and a tree with the algorithm. The header contains several data structures needed to process

a graph (as explained in Section 4.1.2) which are always the same for every individual. Besides that, an individual has an array called the `CHANCE_TABLE` which is created first, because it influences the way the algorithm can develop. The `CHANCE_TABLE` determines the chance that a certain element is chosen when the tree is being constructed. In this table for every kind of element in the tree there is an integer number. The rate between these numbers for certain types of elements determine the chance they will be chosen. Therefore they need to be initialised prior to the initialisation of the algorithm. When a child is created using crossover or mutation, the values in the `CHANCE_TABLE` of the parent(s) will be used to choose the values for the table of the child. The exact way in which this is done is described in Section 4.3.5.

All individuals are created using the grow-method described by Koza [26]. Following the grammar in Table 4.1, the tree will begin with a `List_of_Stats`-element. From there for every element in the tree that is not a terminal element, random children are chosen using the chances in the `CHANCE_TABLE`. In this recursive way the entire tree will be constructed until a certain maximum depth has been reached. When this depth is reached, only terminal nodes can be chosen.

4.3.3 Evaluation and Viability Check

The actual creation of an individual will cost energy for that individual and the amount of energy depends on the size of the individual. This energy is subtracted from the amount of energy it got when being born (E_{birth}) and immediately returned to the population ($E_{available}$). This gives the following equation for the initial energy-level of an individual, called $E_{initial}$:

$$E_{initial} = E_{birth} - (|\text{algorithm}| \cdot E_{node}) \quad (4.1)$$

where E_{node} is a constant determining the amount of energy it costs to create a single node and $|\text{algorithm}|$ is the size of the algorithm, the amount of nodes in the tree. If the individual does not have any energy left, it ceases to exist and all its energy will be returned to the population. This enables us to immediately try to create another individual.

Immediately after an individual has been created, whether this is randomly at initialisation or by using crossover or mutation later on in the run, it will be evaluated, provided, of course that it has energy left. There are 100 fitness cases which it needs to try to solve to calculate its fitness score. The fitness of an individual depends on its score on the different fitness cases and does not depend on its energy level. How these scores are calculated exactly, depends on the problem type.

After a graph is created, the value of the *Randomly Found Score* (RFS) is determined. To do this 10,000 random permutations are created which are used as input for the greedy algorithm. The average value of all the found scores is the RFS for that graph. This value is used to determine the fitness of an individual. Every time an individual evaluates a graph it gets a certain score for that graph. The difference between the found score and the RFS is squared and added to the total fitness of that individual when the found score is better, but subtracted from it when the found score is worse than the RFS. The sum of all these is

the fitness of that individual. Formalized, when s_{ij} is the score that individual i found for graph j , we can calculate the fitness of individual i by:

$$\text{FITNESS}_i = \sum_{j=0}^{100} (s_{ij} - \text{RFS}_j) \cdot |s_{ij} - \text{RFS}_j| \quad (4.2)$$

During this evaluation we keep track of the amount of energy the individual uses. Every time it visits a node in its tree, it costs one unit of energy. This amount is stored in $E_{fitness}$ instead of being released back into the population right away. When $E_{fitness}$ exceeds $E_{initial}$, the individual needs more energy to evaluate all the fitness cases than it has, which results in this individual to be terminated immediately and returning $E_{initial}$ to the population.

When an individual evaluated all the fitness cases and $E_{fitness}$ did not exceed $E_{initial}$, the individual was able to do all its work with the amount of energy it had. Every individual for which this holds true is said to be *viable* and will be added to the population. Formalized, we could say that for every individual in the population the next equation holds:

$$0 < E_{fitness} \leq E_{initial} < E_{birth} \quad (4.3)$$

4.3.4 Selection

When the program is running and we are to create a new individual, we first need to decide whether this will be done using crossover or mutation. The method is chosen randomly and the rate between the two may differ between one run and the other. Depending on whether we choose crossover or mutation we may need two parents or one. Every single parent is always chosen in the same manner which is done once for mutation and twice for crossover.

A parent is chosen using *tournament selection*. A number of individuals, called the tournament size, are chosen randomly from the population. The individual with the best fitness will become parent of the to be created individual. The difference between the regular tournament selection (see [5]) is that in Energy Bound GP the weak performing individuals are not replaced. When the tournament size is big, there is tough competition in the tournament. In this case the *selection pressure* is high.

4.3.5 Creation of Offspring

When the program is running, new offspring will only be created using mutation or crossover. Random creation is only done at initialisation. When offspring is created first the header of the individual is created. This is for both crossover and mutation done in an analogue way. Then the algorithm is created. This is where the genetic operators have their impact.

Creation of the Header

When the header of an individual is created, we first need to choose the values for the elements of the CHANCE_TABLE. Let us assume there are two parents. The difference when there is only one parent is explained later. Every element in the table is an integer. All of these integers for a new individual are created in the same manner. Let us see what happens when we want to determine the value for such an integer.

The value for the integer of the child we want to choose we will call V_c (Value for child). It needs to be between 0 and a defined maximum (MAX). This maximum is 100 for an element in the CHANCE_TABLE and 20 for AMOUNT_INDICES. The corresponding integers of the father and the mother are called V_f (Value for father) and V_m (Value for mother) respectively. Then we define the smallest of V_f and V_m as V_s (smallest value) and the largest as V_l (largest value). V_s and V_l are used to create a probability distribution function ($p(x)$) for crossover according to:

$$p(x) = \begin{cases} \frac{h}{V_s} \cdot x & \text{when } 0 \leq x < V_s; \\ h & \text{when } V_s \leq x \leq V_l; \\ h - \left(\frac{h}{\text{MAX} - V_l}\right) \cdot (x - V_l) & \text{when } V_l < x \leq \text{MAX}. \end{cases} \quad (4.4)$$

where h is the height of the function. How to calculate h is described later. Figure 4.3 shows an example. In this probability distribution we can see that the chances for any number between V_s and V_l are equal, but the chances for numbers outside the range $[V_s, V_l]$ are smaller. In Figure 4.3 it is easy to see how this works. The further a number is away from the range $[V_s, V_l]$, the smaller is its chance of being chosen.

Now, since the function is either linear or even constant the value of h does not have any influence on the ratio of the chances for two different numbers. If h would be multiplied by a certain factor, the same is true for every chance in the distribution function. Therefore we could choose h to have any value we want, without consequences for the ratio of the chances. But, if we want it to be a probability distribution, the size of the surface under the function must be exactly 1 because the sum of all chances is always 1. Therefore we can calculate the value of h . The surface (S) beneath the function can be described by:

$$S = \frac{1}{2} \cdot V_s \cdot h + (V_l - V_s) \cdot h + \frac{1}{2} \cdot (\text{MAX} - V_l) \cdot h \quad (4.5)$$

and when we fill in 1 for S then we can get the formula to calculate h :

$$\begin{aligned} 1 &= \frac{1}{2} \cdot V_s \cdot h + (V_l - V_s) \cdot h + \frac{1}{2} \cdot (\text{MAX} - V_l) \cdot h \\ 1 &= \frac{1}{2} \cdot h \cdot \left(V_s + 2 \cdot (V_l - V_s) + (\text{MAX} - V_l) \right) \\ 1 &= \frac{1}{2} \cdot h \cdot \left(\text{MAX} + V_l - V_s \right) \end{aligned}$$

which finally gives the equation:

$$h = \frac{2}{\text{MAX} + V_l - V_s} \quad (4.6)$$

So, every time we need to choose an integer in the header of an individual we create the probability function as described. Then we choose a random number

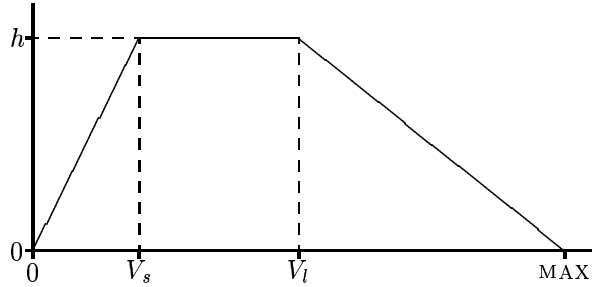


Figure 4.3: Probability distribution created with equation 4.4.

using this function. In this way the chances for certain values are determined by the parents, but it can still take any value. Take into account that the exact chances may vary a bit since the fact that some numbers need to be rounded. When there is only one parent, say, the father, things will go similar except for the fact that since there is no mother there is no V_m and the distinction between V_s and V_l is useless. Therefore if we substitute V_f for V_s and V_l we will get the probability distribution function for mutation.

Creation with Crossover

The crossover operator chosen for the initial run is ‘subtree exchange crossover’ as described in [5]. First a father and a mother would be selected. Then two random points in both their trees would be chosen. As we have seen in Section 4.1.1 we could consider the tree representation as a list of statements. The crossover points in the chosen parents are pointers between the `List_of_Stats`-elements which separate the various statements. Here, nested statements are not taken into account. For instance, if we consider the individual drawn in Figure 4.1 there are only two points that can be chosen as a crossover point. The first is the arrow from the node labelled ‘1’ to the node labelled ‘2’ and the other is the arrow from the node labelled ‘2’ to the node labelled ‘5’.

After the points are chosen in both the father and the mother we create a new individual by taking the first part (from the root to the crossover point) of the father and the last part (from the crossover point to the leaves) of the mother. It is easy to see that the created individual remains syntactically correct.

This new individual will be evaluated and checked for viability as described in Section 4.3.3. If the new individual is viable it will be added to the population. If the new individual is not viable, two new points are chosen to create a new child, which is also checked. This will go on until a viable individual is found. If all possibilities for crossover have been tried and no viable child has come out of it, two new parents will be chosen.

Creation with Mutation

When a new individual will be created using mutation, one parent, called the father, is selected. After the header for the new individual has been created as explained previously, one of the statements is selected to be mutated. This statements is chosen randomly from the ones that are not nested. Assume, for instance, that the individual drawn in Figure 4.1 is selected to be the father for a new individual that is to be created using mutation. We see it has five statements which all are linked to a `List_of_Stats`-element labelled with a number. Let us say that these labels apply for the entire statement. Then the statement which is chosen to be mutated is selected randomly from the statements ‘1’, ‘2’ and ‘5’. The statements ‘3’ and ‘4’ cannot be selected.

Once a statement is selected it will be removed entirely. This means that when, in the example of the individual in Figure 4.1, statement ‘2’ is selected, the entire subtree hanging at the right of the `_List_of_Stats`-element labelled ‘2’ will be removed, including the statements ‘3’ and ‘4’. After removing the statement it will be replaced by a new one. This new statement is created randomly in a way that is the same as it would be at initialisation as described in Section 4.3.2. This will, of course, guarantee that the new individual will be syntactically correct.

This new individual will be evaluated and checked for viability as described in Section 4.3.3. If the new individual is viable it will be added to the population. If the new individual is not viable, the entire procedure will be repeated with the original father. This will continue until a viable individual is created or until after five attempts still no viable offspring has been created. When the latter is the case, a new father is selected and the entire procedure starts over again from there.

4.3.6 Making a Time Step

We have seen how an individual is created, let us now take a look at how the population develops through time. We could say the population in the program has two states: one is when time is running and the other when time is standing still. If we look at Figure 4.2 the mentioned states correspond to the two branches from the question “Is there enough energy to create an individual?”. The branch labelled ‘NO’ corresponds to the state when time is running and the branch labelled ‘YES’ corresponds to the state where time stands still.

When time stands still, the program uses this ‘break’ to create new individuals as we have seen in the previous section. When time is running, or, said differently, when the program makes a time step, each individual consumes energy thereby decreasing its current energy level. This current energy level is called $E_{current}$ and at birth it is initialised at $E_{initial}$. Every time step each individual will consume energy according to its own consume rate. This consumption rate is called $E_{consume}$. At first glimpse it would look logical to just take $E_{fitness}$ for it. Unfortunately, this has two disadvantages.

First of all, it stimulates the occurrence of very small trees. When a tree is very small (size < 5) it has poor fitness, but also a low $E_{fitness}$ which gives it a great advantage over larger trees. The disadvantage of a poor fitness is not in proportion with the advantage of the low $E_{fitness}$. This will result in a severe

dropping of the average size which, in its turn, causes the fitness values to drop and the evolution to halt. To compensate for this, Nabuurs [32] suggested to add a very small penalty to $E_{fitness}$, which he called E_{fixed} . Because the penalty is very small it will hardly have any influence on the consumption rate of normal sized individuals, but it has a big influence on that of small individuals.

The second disadvantage is the fact that releasing all the energy $E_{fitness}$ at once will result in big jumps through time. This causes it to be very difficult for the program to make a distinction between individuals that have a different consumption rate. Compare, for instance, an individual that has still 90% of $E_{fitness}$ left after the first time step and an individual that wasted all but 10% of $E_{fitness}$ after the same time step. Although the first individual is more economical and therefore, due to the principles of Energy Bound GP, should live longer and have more chance to reproduce, both individuals will be deleted after the second time step. For the program there is no distinction between the consumption rate of the two individuals.

To solve this problem, we divide the consumption rate into T steps. This also has as effect that the moments when new individuals will be created will be spread out more evenly through time. Both refinements lead to the following equation:

$$E_{consume} = \frac{E_{fitness} + E_{fixed}}{T} \quad (4.7)$$

For choosing the exact values for E_{fixed} and T there are no rules yet, but the programmer should easily find good enough values while fine-tuning his program.

4.3.7 Termination

Sometimes it is difficult for a programmer to determine when to terminate a run. The programmer could, for instance, choose to set the amount of generations it needs to evaluate to 500 and later, when examining the results, he realizes that after about 70 or 80 generations there was no significant improvement in the quality of the individuals, resulting into the useless evaluation of over 400 generations. Energy Bound GP gives us the ability to use ‘smart termination’ as invented by Nabuurs [32].

As we have seen for an individual the chance of reproduction is determined by both its fitness and its age, determined by its size and consumption rate. The second influences the chance of being chosen for a tournament, while the first influences its chance of actually winning it. The changes in the rate between the two can tell us something about when to terminate the run.

When we use Energy Bound GP we can see that at first the fitness of the individuals improves. After a while we have found a (local) maximum and the fitness value of the best individual will not improve anymore. Instead of that, we see that the individuals become smaller and more efficient (lower consumption rate) so they will live longer. Therefore if we can detect when this happens, we can terminate the run at the best moment.

Now, how can we do this? At regular intervals (size $T/4$) the fitness F and size S of the smallest best individual is measured. We take a new variable which we call `SIZE_DROP` (or short d), which starts at 0 and cannot be negative. If there is an increase in fitness, `SIZE_DROP` is reset to 0. If there is no increase in the fitness, a constant is added to `SIZE_DROP`. The size of the constant is determined

by the difference in size. If we consider a case where we want the fitness to be as high as possible, at every i^{th} interval (with $i > 0$) we can calculate `SIZE_DROP` in this way:

$$d_i = \begin{cases} 0 & \text{when } F_i > F_{i-1} \\ \max(d_{i-1} - 1, 0) & \text{when } F_i \leq F_{i-1} \text{ and } S_i > S_{i-1} \\ d_{i-1} + 1 & \text{when } F_i \leq F_{i-1} \text{ and } S_i = S_{i-1} \\ d_{i-1} + 3 & \text{when } F_i \leq F_{i-1} \text{ and } S_i < S_{i-1} \end{cases} \quad (4.8)$$

When `SIZE_DROP` reaches a certain threshold, the run is terminated.

The benefit of smart termination is that it does not take any assumptions about the problem, fitness values, characteristics of an individual or the size distribution. Therefore you can always use smart termination which has little or no overhead. A disadvantage is the fact that it does not guarantee termination. When the fitness and size of the smallest best individual fluctuates it is quite easy to construct situations where smart termination will never reach the threshold. Therefore to be absolutely sure your run will always terminate it is good to also set a (very large) maximum to the amount of time steps your run can take. Experimental result by Nabuurs [32] have shown that these fluctuations hardly ever occur.

Chapter 5

Fine-tuning the Program

The basic strategy for this thesis starts with finding good parameters for the genetic program. This is done by tuning the settings for the program in such a way that it will perform well for the Maximum Clique Problem (MCP) as described in Section 3.2. MCP is chosen for this fine-tuning since, as described in Section 3.1.1, this was the original problem that inspired the idea.

The program needs quite a few parameters. Some of them will be chosen based on recommended settings from literature or by trying them while programming. Others, however, need to be determined by researching a range of possibilities. After this research several sets of parameters will be chosen and for these sets the program will try to solve all six problems to see if they differ. The parameters for which good values need to be found are E_{total} , E_{birth} , E_{fixed} and the rate between mutation and crossover.

5.1 General Settings

Some parameters are selected without the support of well-founded research. These parameters were found during the programming and debugging of the program and showed satisfactory compromises between good results for the program and good computer performance. Table 5.1 shows these parameters and their values.

MAX_NODES first was 500, but this resulted in serious problems with the computer memory. The runs hardly ever finished because there simply was not enough memory. This has to do with the fact that the graphs are implemented as adjacency matrices and all the values for all the different operators on the graphs (see the grammar in Table 4.1) are calculated in advance to speed up the processing of the graphs. All these values had to be stored in arrays or double arrays which used a lot of memory. For example, a graph with 50 nodes costs about 25 kB and a graph with 500 nodes costs about 2450 kB. Since graphs with 50 nodes still provide an adequate amount of information, the maximum amount of nodes is set to 50.

MAX_EDGE, EVA_GRAPHS, INDICES, MAX_CHANCE and MAX_INT have just been chosen while debugging the genetic program. During the testing and debugging phase several options were tried until values came up which resulted in satisfying

name	description	value
MAX_NODES	maximum amount of nodes in a graph	50
MAX_EDGE	maximum weight of an edge in a graph	100
EVA_GRAPHS	amount of graphs used for the evaluation	100
INDICES	amount of pointers an individual may use	4
MAX_CHANCE	maximum value for element in CHANCE_TABLE	100
MAX_INT	maximum value for a number in the grammar	32
MAX_STAT	maximum depth for Statement-elements	25
MAX_TEST	maximum depth for Test-elements	4
MAX_NUMB	maximum depth for Number-elements	4
MAX_NODE	maximum depth for Node-elements	4
E_NODE	energy it costs to create one node in the tree	5
E_PROCESS	energy it costs to evaluate one node in the tree	1
TIME_STEP	value for T (as described in Section 4.3.6)	20
SIZE_DROP	value for SIZE_DROP (as described in Section 4.3.7)	100
TOUR_SIZE	tournament size during selection	7

Table 5.1: Values for the parameters that are chosen without extended research.

runs. The same holds true for the maximum depths in the trees describing the algorithm, but there is a difference in MAX_STAT and the others. The crossover operator can make a tree exceed the maximum amount of Statement-elements. Energy Bound GP will prevent it to grow extraordinary big.

The values for E_NODE, E_PROCESS, TIME_STEP, TOUR_SIZE and SIZE_DROP have been chosen in correspondence with the results from Nabuurs' Master's Thesis about Energy Bound Genetic Programming [32].

5.2 Tuning E_{fixed} and the Genetic Operators Rate

First some test runs were done and the results were examined to see how the parameters had to be found. Careful consideration of these test runs determined that first a good value for E_{fixed} had to be found. An average parameter setting was chosen ($E_{total} = 1,000,000$ and $E_{birth} = 5,000$ and a mutation rate of 50% which means that 50% of the offspring is created using mutation and the rest using crossover). Figure 5.1 shows the values of the average fitness of the population and the best found fitness at the end of the run. The values are the mean over 250 runs. Also shown are the ranges from the mean -1 SD to the mean $+1$ SD. In this phase of the thesis still only 40 evaluation graphs were used.

What can be seen here, is that the value for E_{fixed} does not have any influence on the score of the program. Even when E_{fixed} is 0, the fitness-values are not really different, especially in consideration of the fact that the SD is very big, indicating a wide spread of results. This is odd, since Nabuurs [32] inserted E_{fixed} to prevent the run from creating very small individuals which would result in an early termination of the run (see also Section 4.3.6). If this would be true then it is expected that without E_{fixed} , so when it is 0, the program will not be able to reach the same fitness values. Instead it can be seen that the value for E_{fixed} does not have any influence at all.

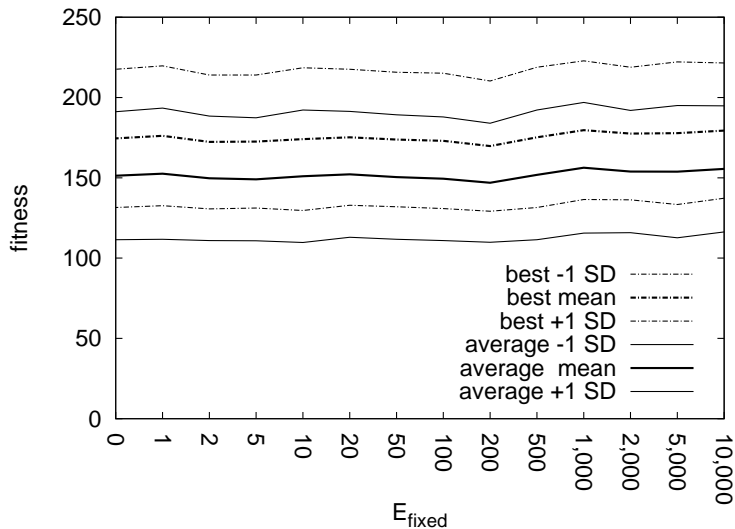


Figure 5.1: Values for both the average fitness and the best fitness of the population at the end of the run for different values of E_{fixed} . Values are the mean after 250 runs together with the mean -1 SD and the mean $+1$ SD.

Another thing to notice is that the fitness value for the best individual and the average fitness value of the population appear to be highly correlated. Taking into consideration a number of 250 runs it has been discovered that the correlation coefficient between the fitness value for the best individual and the average fitness value is 0.999. Therefore it is safe to only use the fitness value of the best individual to determine the values for the parameters.

It is of course possible that for different values for the other parameters (E_{total} , E_{birth} and the mutation rate) there is an influence of E_{fixed} . Therefore several different values of E_{total} and E_{birth} were applied, but they did return any change. There could be an important role for the rate between mutation and crossover. When the depth of an individual, which is the main concern for inserting E_{fixed} , is considered, it is clear that this cannot be influenced by mutation. This is because mutation only exchanges one **Statement**-element for another. Crossover on the other hand, can create both smaller and bigger individuals.

To see if this is true the same experiment as above was repeated for different values of the mutation rate. The results can be seen in Figure 5.2. Here the values of the mean -1 SD and $+1$ SD are left out to make the figures more clear. The sizes of the SD's were comparable with that of the first experiment. Again it shows that the value for E_{fixed} is not of any influence on the results of the genetic program. On the other hand it shows that the mutation rate does have some influence. Since the SD of any of the runs is much greater than the differences reflected by these figures, it is impossible to state that these results are in fact different. However, in order to be safe, from now on, a relatively high value for the mutation rate of 80% will be chosen. This means that in 80% of the cases offspring will be created using mutation, leaving only 20% of the offspring being created by crossover.

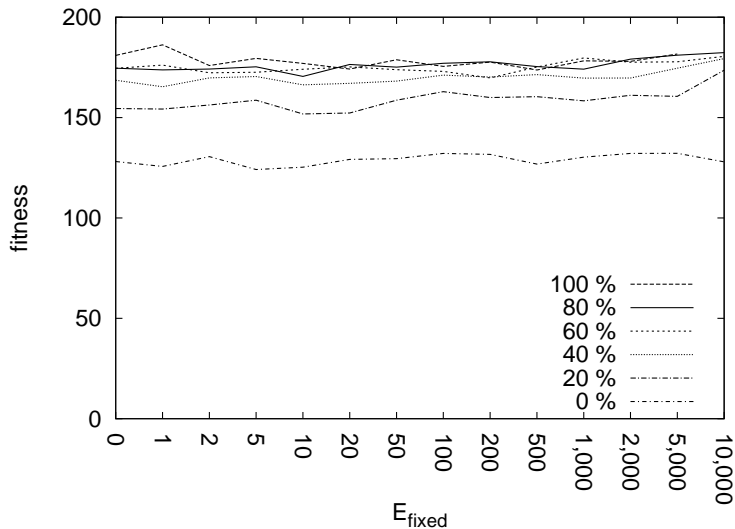


Figure 5.2: Values for the best fitness of the population at the end of the run for different values of E_{fixed} while taking different values for the mutation rate. Values are the mean after 250 runs.

As it is demonstrated that E_{fixed} does not have any influence, any value can be chosen for it. Here also a “better safe than sorry” approach will be chosen by setting its value to 1. With this, Nabuurs’ advice to choose a small, non negative number for E_{fixed} [32] is complied to.

5.3 Tuning E_{total} and E_{birth}

The ratio between E_{total} and E_{birth} has a great influence on the way the population acts. If E_{birth} is very small the individual does not have much energy to solve all the fitness cases. This will result in poor fitness values for the individuals. When E_{birth} is too small it will even be impossible for the individual to solve all the cases. For this progra, the value for E_{birth} must be at least 2, 500 to achieve a minimum performance. On the other hand, when the value for E_{birth} gets bigger and bigger, and E_{total} stays the same, the population size will become smaller and smaller. This will also result in a drop in the fitness values.

In order to find good settings for the program several combinations for E_{total} and E_{birth} were evaluated. E_{total} was evaluated for the values 1, 2.5, 5, 7.5 and 10 million and E_{birth} was evaluated in the range from 2, 500 to 100,000. For every combination 250 runs were done with the settings as shown in Table 5.1 and the values for E_{fixed} and the mutation rate as found in Section 5.2. The following aspects of the genetic program were examined: the average time for a run, the average and best fitness value of the population at the end of the run and the size of the population at the end of the run.

Figure 5.3 shows the average amount of time it will take to do a run with the given settings. It is very obvious that for low values of E_{birth} it will take a lot

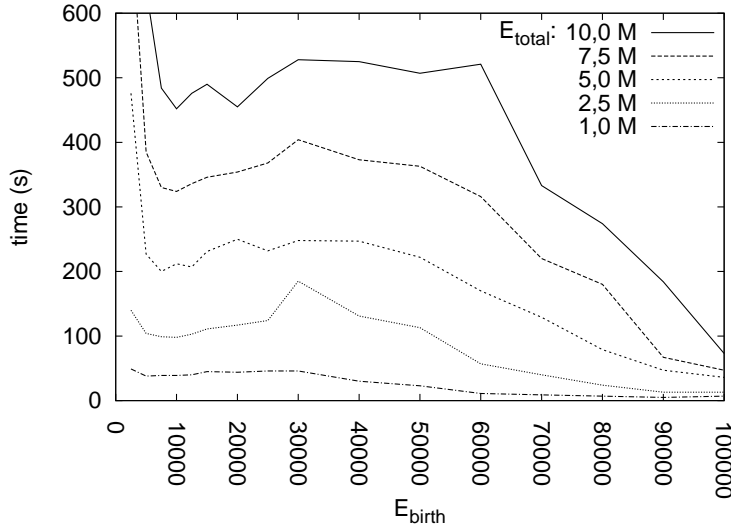


Figure 5.3: The average amount of time for a run.

of time for the program to finish. This is because it is very difficult to find individuals that can solve all fitness cases with very little energy. When E_{birth} has a high value, the program finishes early. This is probably caused by the fact that a high value of E_{birth} results in a low population size. When there is only a small amount of individuals, the genetic program cannot develop well and will terminate early.

Figure 5.4 shows the fitness of the best individual. It is easy to see that for every value of E_{total} there is an optimal value for E_{birth} which will result in the best fitness values.

5.4 The Chosen Settings

As seen in Chapter 4, when using Energy Bound Genetic Programming, the size of the population will vary throughout the run. New individuals are only added when there is enough energy and the amount of available energy is determined by the consumption rate ($E_{consume}$) of the individuals. Eventually, after a while the population size varies around a certain value. This is when on average the amount of individuals that die is approximately the same as the amount of new individuals that are created. Although the population size still varies, there is a rough correlation between the values of E_{total} , E_{birth} and the population size at the end of the run. For this connection there is the following equation which can at least give us an estimation of the population size:

$$\text{Population size} \approx c \cdot \frac{E_{total}}{E_{birth}} \quad (5.1)$$

where c is a constant. If c is calculated at the end of all the runs, it can be found that it has a value of 2.039 with a standard deviation of 0.069.

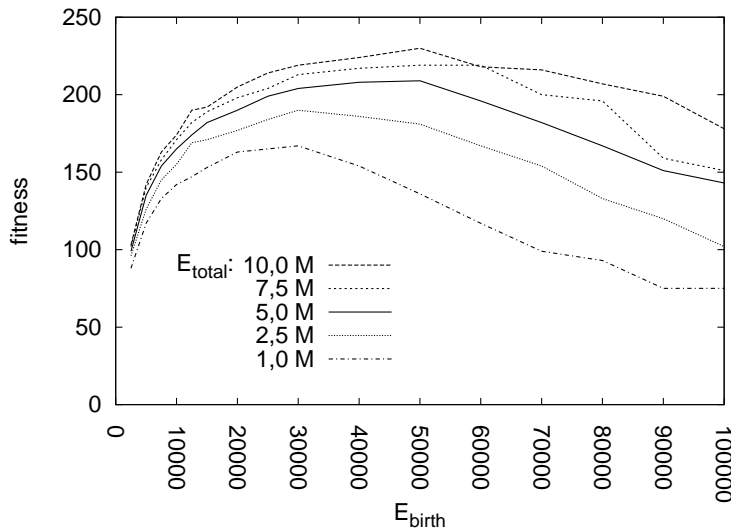


Figure 5.4: Values of the best fitness of the population at the end of the run. Values are the mean after 250 runs.

The fact that the population size is about twice the value of E_{total} divided by E_{birth} is easy to explain. All the energy in the population (E_{total}) is divided over the individuals. There is a little amount left ($E_{available}$), but this is even less than E_{birth} , otherwise a new individual could have been created. Each individual starts with 100% of the energy E_{birth} , releases this energy as time passes by and dies when it has no energy left. After the program has been running for a while, it is most likely that the energy levels of all the individuals are distributed equally in the range from 0% to 100% of E_{birth} . This means that on average the individuals will have an energy level of 50% of E_{birth} which explains why the population size is approximately twice the value of E_{total} divided by E_{birth} .

Every one of the six problems will be tried to be solved with three different parameter settings. The parameters in Table 5.1 will naturally stay the same, together with the value for E_{fixed} which will be 1 and the mutation rate which will be 80%. The three settings will be determined by different values for E_{total} and E_{birth} . Besides the fact that the ratio between the two gives an estimate for the population size, E_{birth} also determines the maximum size of the individual, together with its available energy to solve the fitness cases.

Taking all this into account including the fact that these runs should not last more than ten minutes, three parameter settings for the genetic program emerged. Table 5.2 shows them. They were chosen because they were considered to have a good variation in both the population size and the size of an individual. Furthermore, these settings are close to the maxima in Figure 5.4.

Setting	E_{total}	E_{birth}	population size
A	1,000,000	20,000	≈ 100
B	5,000,000	50,000	≈ 200
C	10,000,000	40,000	≈ 500

Table 5.2: The three parameter settings for the runs.

5.5 Testing the Results on a Series of Test Graphs

Now the program is almost ready to solve the six problems. There are only a few problems remaining to be solved. Two of these are well-known problems in Genetic Programming and Artificial Intelligence and are called *sampling error* and *overfitting*. The other has to do with the setup of this thesis and is the problem that there still has to be found a way of comparing the results of the different runs. All three can be solved in the same manner. First, we take a closer look at the problems.

Every run has its own training set (see Section 3.1.2). Different training sets have different values for their solutions. This means that in one run the individuals have the possibility to exceed the average much more than in another run where the possibilities are much more limited. When the way the fitness of an individual is calculated (see Section 4.3.3) is looked upon closely, it is clear that this will have a big effect on the possible fitness values. Eventually this will average out, but it will result in a wide spread of final scores for each run, making it hard to compare different sets of runs. This is called *sampling error*, and more information about it can be found in [5].

When an individual has a very high fitness, it could mean that it is very good in solving the problem for which it is designed. Unfortunately, it can also mean that it is very good in producing the correct answers for the cases in the training set. This means that the individual is too “specialized” for the training set and did not find a general rule for solving the problem. When this is the case we speak of *overfitting*, and more information about it can be found in [35].

The six different problems all have their own range of solutions. This means that individuals in a program solving one problem have totally different fitness values than individuals in a program solving another problem. If these different genetic programs are to be compared, there needs to be a way to generalize their results.

These three problems can all be solved with one simple addition: As proposed by Russell and Norvig [35], the problem of overfitting can be solved by introducing a *test set* which is a set of test cases that is different from the *training set*. If the same test set for every run of a problem is taken, the problem of sampling error can also be reduced. The only thing that needs to be done is to find a way to generalize the results so they can be compared. This asks for a new way to calculate the quality of individuals at the end of a run. For this purpose, the *Final Score of an Individual* (FSI) was introduced. It is calculated for every individual at the end of each run.

For each of the six problems a set of 2,000 graphs is made. These graphs are made using the same parameters as described in Chapter 3 where they were used 20 times. As described in Section 4.3.3 for every graph the RFS (Random Found Score) can be calculated. This is the average score that you get when

you feed 10,000 random permutations into the greedy algorithm for that graph. To calculate the FSI of an individual, it will try to solve the problem for all the graphs in the test set. Whenever it finds a better score for a graph than the RFS, it will score 1 point and else it does not get a point. The total amount of points scored is the FSI of an individual. It can range from 0 (very poor) to 2,000 (very good). When it scores 1,000 it is just as good as random search, since it has beaten the RFS as many times as it has been beaten by it.

In this way, the problems of sampling error and overfitting are solved and it enables us to compare the results for the programs across the different problems. A shortcoming of this method is that the individuals are not rewarded more when they perform much better on a single graph. Consider, for instance, a graph with 10 vertices where the RFS for MCP is 3.14. An individual that finds a clique of size 4 gets 1 point, which is exactly the same as an individual that found a clique of size 6 in the same graph, although the second individual performed much better.

Unfortunately, the second individual cannot be given any bonus points. When this would be done, it would become impossible to compare its score to that of an individual which had to solve a different problem as, for instance, HCP. An individual that performed outstandingly on HCP could be given a bonus as well, but it is impossible to know in what relation the two bonuses should be. Since you cannot compare the quality of a found Hamilton cycle to that of a found clique, you cannot give them a bonus which is in proportion. The only way in which you can compare them is whether they perform better or worse than the RFS. In this way you lose the extra information of extremely good performing individuals, but you will keep the possibility of comparing the scores.

Chapter 6

Results

6.1 Results for the Test Set

For each of the six problems there were three settings. They are shown in Table 5.2 and called setting A, B and C. From now on, to make things easier, a combination of a problem and a setting will be called a PS-combination (Problem/Setting-combination) Every PS-combination has its own Genetic Program which made 1,000 runs. The most important was of course their scores on the test set. At the end of each run both the best score of an individual and at the average score of all individuals is examined. Figure 6.1 shows the average score of the individuals. For every PS-combination it shows the mean over 1,000 runs and the spread from -1 SD to $+1$ SD.

It is clear to see that the three problems which are known to have polynomial complexity (MST, BGP and CCP) performed significantly better than the three NP-complete problems (MCP, GCP and HCP). The same holds true if we only look at the best individual.

There is a wide variety in spread for the scores of the PS-combinations. This can partly be explained. When it is difficult for a program to solve a certain problem, the found scores will stay close to 1,000, resulting in a small range of scores and thus a low standard deviation. When a problem is relatively easy to solve, the scores will be farther above 1,000, which will result in a wider range for these scores and thus a bigger standard deviation. Considering the average score of all individuals, there is a correlation of 0.70 between the mean and the standard deviation. And considering the best score of an individual, there even is a correlation of 0.89 between them.

6.2 Influence of the Population

As seen in Figure 5.4, for every value for E_{total} the fitness-values of the individuals first increase and then decrease when E_{birth} increases. This means that there will be an optimal population size for every PS-combination. However, Equation 5.1 showed a direct relation between E_{total} , E_{birth} and the population size. The setup with only three settings for every problem therefore did not allow to examine what that optimal population size is.

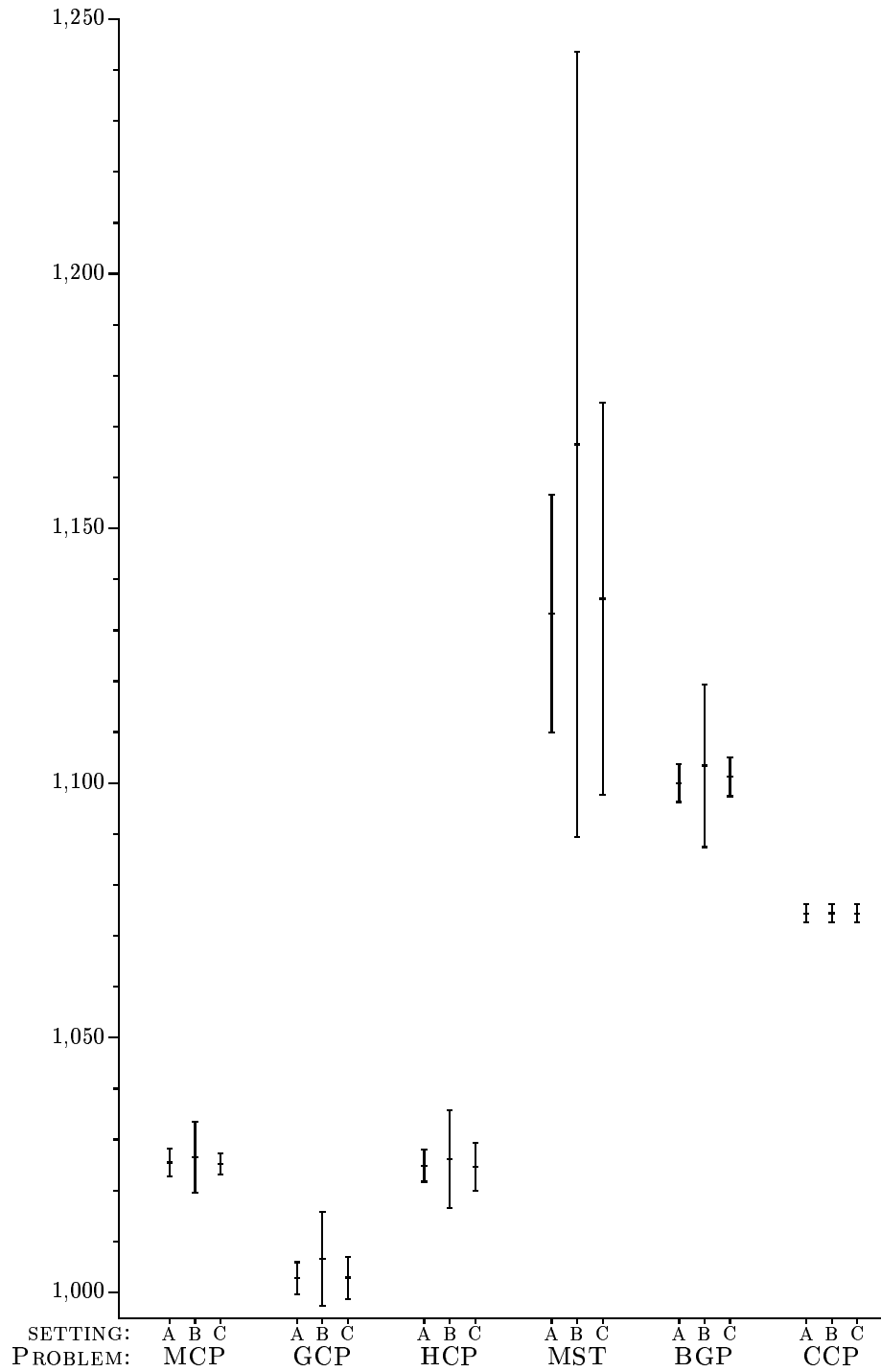


Figure 6.1: Average score for the FSI on all individuals in the population for all six problems and all three settings. Values are the mean over 1,000 runs with their standard error.

	MCP	GCP	HCP	MST	BGP	CCP
A	1503	1446	1592	1414	1206	2313
B	2330	2122	2562	1983	1180	4010
C	1847	1989	1859	1959	769	2735
AVERAGE	1893	1852	2004	1785	1052	3019

Table 6.1: Average number of time steps a run made before termination.

Another important aspect of the population is the amount of time it needed to terminate. Table 6.1 shows the average amount of time steps to the point where the program terminated. The number of steps that BGP and CCP needed, is very different from the other four problems. Unfortunately, since one is much higher and the other much lower and the amount of steps needed by MST is roughly the same as that of the three NP-complete problems, we cannot draw conclusions about difference in behaviour between the two types of problems. Comparing the score of a run to its running time, shows something interesting. One would expect that the longer a program is running, the better its score on the test set will be, but the opposite is true. Runs that took a long time to terminate score worse than runs that terminate early. This is more or less true for all PS-combinations. Figure 6.2 shows the average score of the individuals according to the time it took the program to terminate for setting B for GCP and Figure 6.3 shows the same for MST.

The exact way in which this decrease in performance for longer running runs shows, is not the same for every PS-combination. Nevertheless, they all show this decrease. Figure 6.4 shows, for example, how this is for BGP. Despite these differences between the PS-combinations there is no significant difference between the NP-complete problems and the problems solvable in polynomial time.

6.3 Influence of the Individuals

Before the influence of the aspects of the individuals on the test set score can be examined, a few problems need to be solved. First of all, all runs work with different training sets. This directly influences the range of possible fitness values a lot, but it might also have some influence on the other aspects of the individual. This makes the individuals from different runs difficult to compare. Second, the populations in the runs do not have the same size which also makes it more difficult to compare them.

There is a solution for these problems. The individuals are sorted with decreasing value of the aspect that is to be examined. Then, they are given a rank starting with '1' for the individual that has the highest value for it. Finally, the average score on the test set is calculated for all individuals with the same rank. In order to solve the problem of the different population sizes, the highest ranked individuals are not taken into account. More precisely, for setting A only the first 100 individuals are taken, for setting B only the first 200 individuals and for setting C only the first 500 individuals. This procedure is applied further on throughout this entire section, where all averages are over 100 runs.

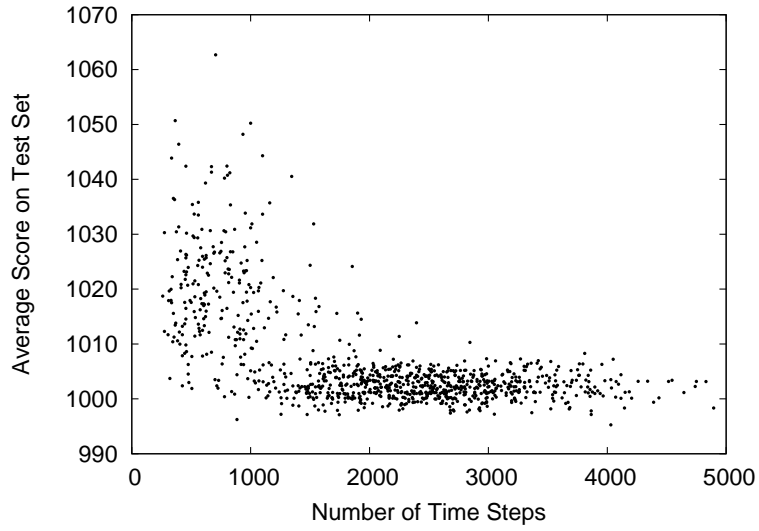


Figure 6.2: Average score of all individuals for setting B for GCP.

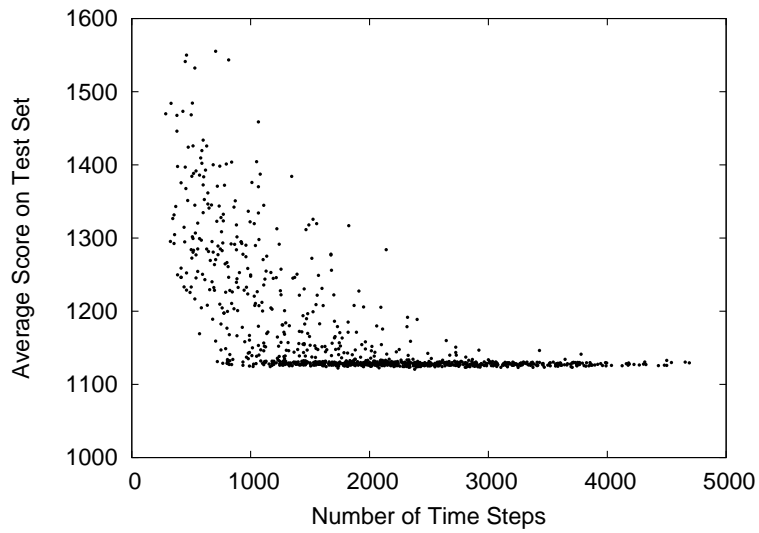


Figure 6.3: Average score of all individuals for setting B for MST.

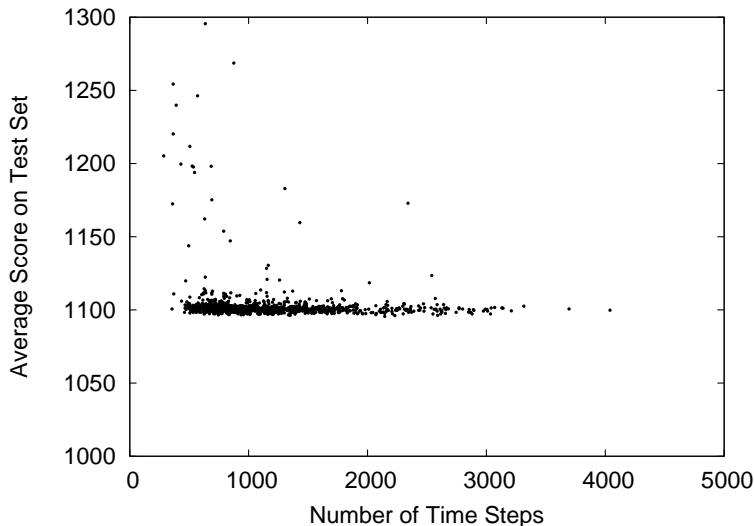


Figure 6.4: Average score of all individuals for setting B for BGP.

6.3.1 Fitness of the Individuals

The most interesting question, of course, is: “Does an individual that performed better on the training set, also perform better on the test set?”. For some PS-combinations we can see that individuals that score better on the training set indeed score better on the test set. For instance, this is true for setting B for MST as Figure 6.5 shows. On the other hand, for the most PS-combinations this is not the case. Figure 6.6 shows the same for setting B for HCP. The first five ranked individuals indeed score better, but further along the ranking the individuals score very good as well. This effect could be caused by overfitting as discussed in Section 5.5. Both situations appear for both NP-complete problems and for problems solvable in polynomial time and therefore this cannot help us in distinguishing the two.

Another interesting thing to look at is the influence of $E_{fitness}$ on the score of the test set. $E_{fitness}$ is directly determined by the amount of nodes in the algorithm tree of the individual that are visited when the training set is solved. It shows clearly that the individuals with higher ranking perform better on the test set. For some PS-combinations this is only true for the top ranked individuals. This is, for instance, the case for setting A for MCP as Figure 6.7 shows. For other PS-combinations this effect can be seen more throughout the entire ranking. This can be seen in Figure 6.8 which shows this for setting B for GCP. Again here, there is no difference between the NP-complete problems and the problems that are solvable in polynomial time.

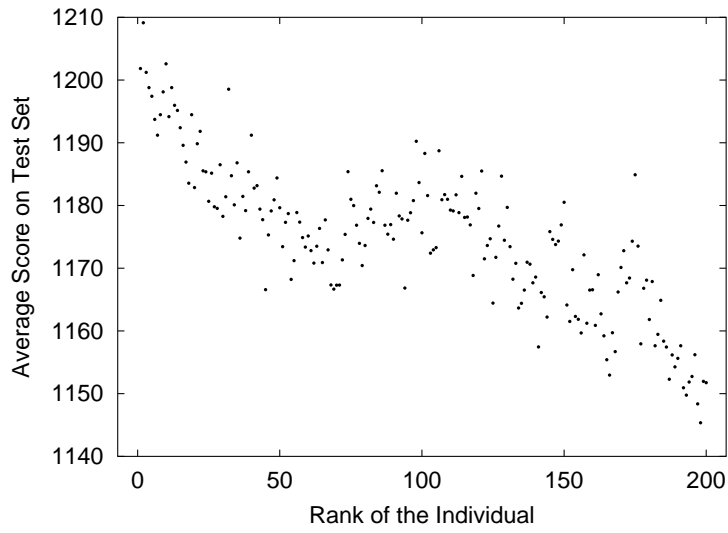


Figure 6.5: Average score on the test set for the ranked individuals sorted descending on the fitness value for setting B for MST.

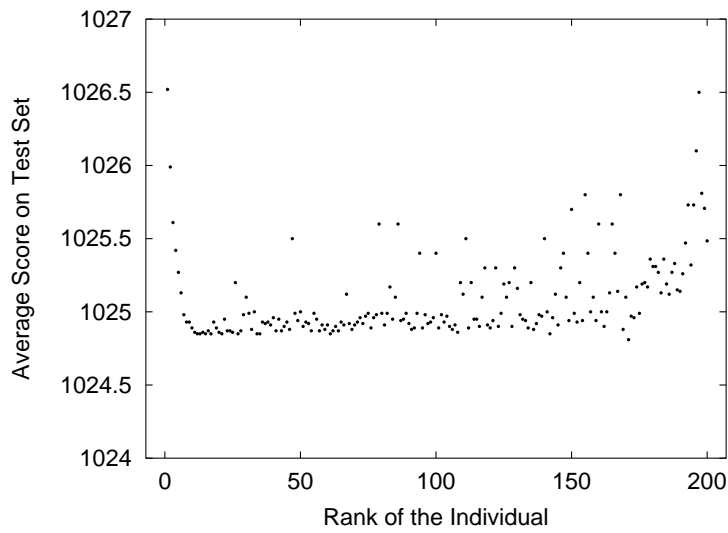


Figure 6.6: Average score on the test set for the ranked individuals sorted descending on the fitness value for setting B for HCP.

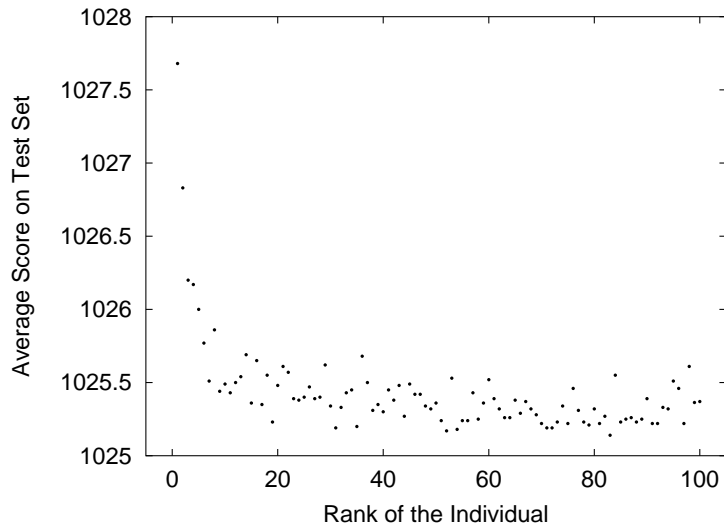


Figure 6.7: Average score on the test set for the ranked individuals sorted descending on the value of $E_{fitness}$ for setting B for MCP.

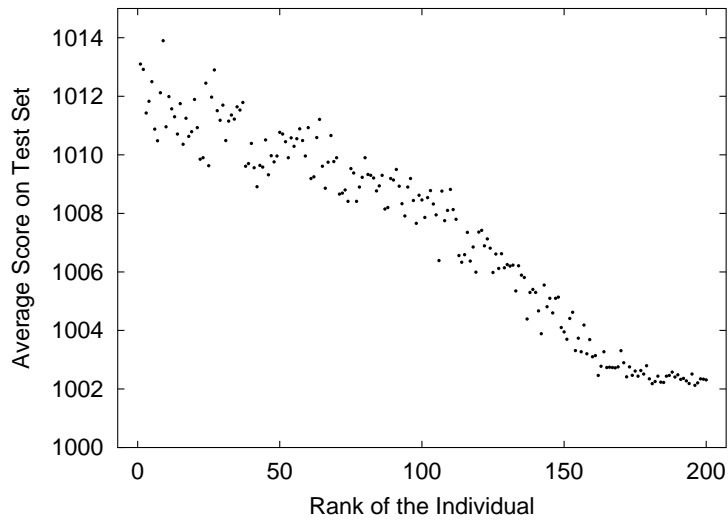


Figure 6.8: Average score on the test set for the ranked individuals sorted descending on the value of $E_{fitness}$ for setting B for GCP.

6.3.2 Size of the Individuals

Another important aspect is the size of the individual. Table 6.2 shows the average sizes of individual for every PS-combination, for every type of grammar-element (see Table 4.1) and in total. There is no significant difference in the amount of **Statement**-elements, the amount of **Vertex**-elements and the total amount of elements used for the different problem types. The only difference you can see is caused by the setting where a higher E_{birth} leaves more room for bigger individuals.

There is a big correlation (0.99) between the amount of **Test**-elements and **Number**-elements used. The reason why this is, is easy to see when we look at the grammar described in Table 4.1. A **Number**-element can only occur when there is a **Test**-element directly above it, or when it is linked with other **Number**-elements to a **Test**-element above it. The amounts of **Test**- and **Number**-elements per individual are roughly the same for MCP, HCP, MST and BGP. On the other hand, the amounts of these elements for GCP and CCP stand out significantly. Unfortunately, since this is not the case for the four other problems, of which two are NP-complete and two are solvable in polynomial time, it is impossible to say that it is a significant difference between the two types of problems.

Since there is quite a difference between the **Test**- and **Number**-elements on one side and the **Statement**- and **Node**-elements on the other side, they are evaluated separately. When we look at the influence of the amount of **Statement**- and **Node**-elements on the performance of the individual on the test set, we must conclude there is hardly any. Figure 6.9 show this for setting A for BGP. The only thing noticeable is the fact that there is a little increase in score for the very small individuals. This is more or less the case for all PS-combinations.

When regarding the influence of the amount of **Test**- and **Node**-elements on the score of the individual on the test set for most PS-settings, it seems there is no influence at all. Only for CCP these graphs do not show flat lines. Figure 6.10 shows the average score of the individuals ranked by the amount of **Test**-elements for setting A for CCP. Here it is plain to see that the highly ranked individuals perform better. Besides that it is interesting to notice that for almost every run of CCP no more than 10 individuals have any **Test**-elements.

When an individual is solving a graph, each element in the grammar tree of the individual can be evaluated 0, 1 or more than 1 times. Table 6.3 shows the average amount of times an element in the tree is evaluated while solving a graph. Again there is no significant difference between the two problem types. The only thing we can see, is that GCP and CCP stand out again with GCP having the lowest usage rate and CCP the highest. If we compare these results with the results on the amount of **Test**-elements used, we could draw the conclusion that the genetic programs for GCP probably use **Test**-elements to create in-trons. The genetic programs for CCP are much more effective with their usage of **Test**-elements.

PROBLEM	SETTING	STATEMENTS	TESTS	NUMBERS	VERTICES	TOTAL
MCP	A	21.9	2.2	2.0	147.1	173.2
	B	63.7	14.3	15.3	444.9	538.2
	C	50.7	8.1	7.6	339.4	405.8
GCP	A	25.2	6.1	5.2	175.9	212.4
	B	65.9	24.5	24.7	513.0	628.1
	C	63.5	20.6	19.3	464.1	567.5
HCP	A	23.2	1.8	1.5	143.7	170.2
	B	67.1	12.5	12.8	417.2	509.6
	C	56.8	6.1	5.7	302.8	371.4
MST	A	24.6	3.7	3.5	146.0	177.8
	B	52.8	13.4	15.4	353.3	434.9
	C	57.1	12.2	12.1	345.8	427.2
BGP	A	22.8	1.4	1.3	141.2	166.7
	B	53.9	10.6	13.4	401.4	479.3
	C	44.9	7.4	9.1	315.2	376.6
CCP	A	28.9	0.1	0.1	126.2	155.2
	B	60.6	0.8	0.8	342.5	404.7
	C	52.6	0.3	0.2	263.8	316.9

Table 6.2: Sizes of the individuals, divided by type of element and in total.

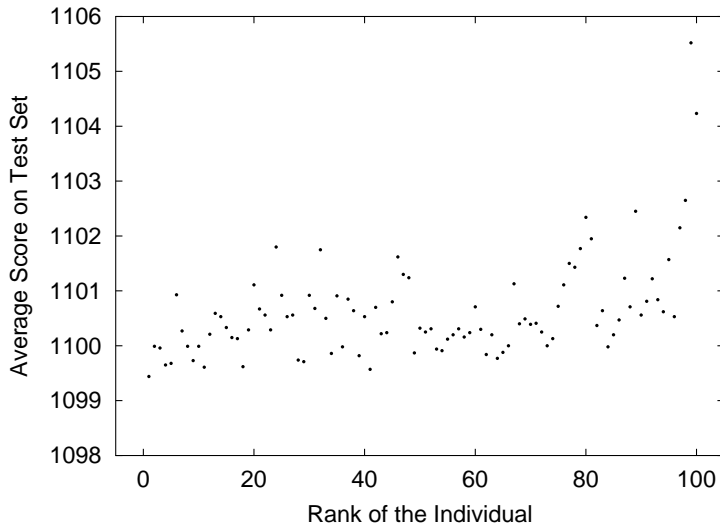


Figure 6.9: Average score on the test set for the ranked individuals sorted descending on the amount of **Statement**-elements for setting A for BGP.

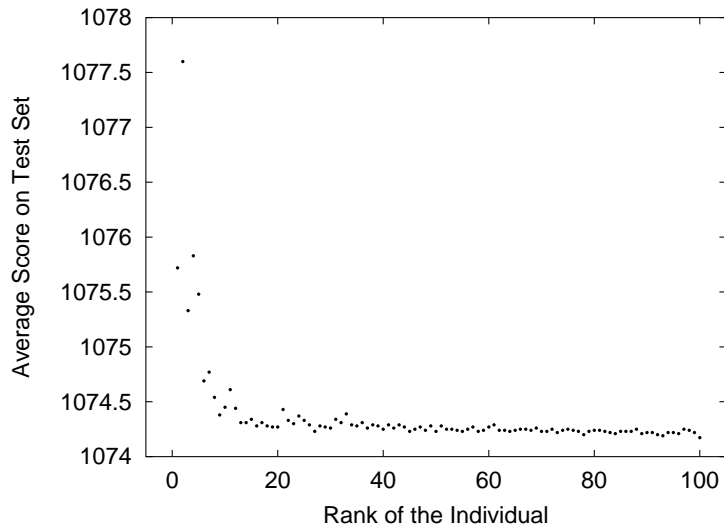


Figure 6.10: Average score on the test set for the ranked individuals sorted decending on the amount of Test-elements for setting A for CCP.

	MCP	GCP	HCP	MST	BGP	CCP
A	1.01	0.76	1.04	0.90	1.04	1.19
B	0.79	0.54	0.84	0.72	0.81	1.14
C	0.88	0.59	0.98	0.77	0.85	1.17
AVERAGE	0.89	0.63	0.95	0.80	0.90	1.17

Table 6.3: Average number of times an element is used in the evaluation of a graph.

Chapter 7

Conclusions

7.1 Main Conclusion

Regarding the results described in Chapter 6, the main conclusion to draw is the fact that there is indeed a big difference in performance between problems that are NP-complete and problems that are solvable in polynomial time. Figure 6.1 shows this clearly. On the other hand, there is no aspect of the Genetic Program that can explain where this difference is originated. Features like running time, fitness values, and individual sizes can have an influence on the performance of the genetic programs, but they do not show any significant difference between the two types of problems.

It has not been possible to show a difference in behaviour for the genetic programs solving NP-complete problems and the programs solving the problems which are known to be solvable in polynomial time. Despite this lack of difference in behaviour, the genetic programs to solve the problems which are known to be solvable in polynomial time still performed much better than the genetic programs to solve the NP-complete problems. This indicates that NP-complete problems are more difficult in nature.

This cannot yet lead to the conclusion that $P \neq NP$. That conclusion would be too premature and there are two reasons for that. First, this thesis did not prove anything. It only showed that in this environment there is a difference in performance between the two types of problems. This could make it a little more plausible that $P \neq NP$, but no more than that. Second, the fact that NP-complete problems have a higher complexity does not necessarily mean they are not in P. They could have complexity $\Theta(n^c)$ with a high value for c , which makes them more difficult, but still in P. Further research on the subject is therefore needed.

7.2 Something to Think About

The question “Is P equal to NP?” originated from the need of people to qualify problems as ‘solvable’ or ‘unsolvable’. If a problem is not solvable, a programmer who needs to deal with this problem has to write an algorithm that approximates the answer, instead of writing one that can solve it. A big group of Computer

Scientists has been trying to solve this question for many years now and still they did not succeed. From one point of view, one could say that they at least have quite a lot of difficulty in trying to solve it. One could even say that maybe this question itself is 'unsolvable' and therefore we need to approximate it. In this light we could say that this thesis is a first approximation of it.

7.3 Further Research

First of all, there need to be further research on the question "Is P equal to NP?". Considering this thesis, it could be very interesting to see if the same results can be found in different environments and for different problems. Furthermore, it is recommended that there will be more research on Energy Bound Genetic Programming to make it the powerful tool which it in potential could be for the field of Genetic Programming.

Bibliography

- [1] *Millenium Prize Problems*. Clay Mathematics Institute, Cambridge, MA, USA, <http://www.claymath.org/millennium/>.
- [2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, USA, 1986.
- [3] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J. Watson. *Molecular Biology of the Cell*. Garland Publishing, Inc., New York, NY, USA, third edition, 1994.
- [4] S. Aurora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 14–23, Pittsburgh, PA, USA, 1992. IEEE Computer Society Press.
- [5] W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone. *Genetic Programming, an Introduction, On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1998.
- [6] M. Bellare, S. Goldwasser, and M. Sudan. Free bits, PCPs and non-approximability - towards tight results. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 422–431, Milwaukee, WI, USA, 1995. IEEE Computer Society Press.
- [7] Paul E. Black ed. Complexity. In *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology, Gaithersburg, MD, USA. 17 December 2004. Available from: <http://www.nist.gov/dads/HTML/complexity.html>.
- [8] I.M. Bomze, M. Budinich, P.M. Pardalos, and M. Pelillo. The Maximum Clique Problem. In D.Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization (suppl. Vol. A)*, pages 1–74, Dordrecht, The Netherlands, 1999. Kluwer.
- [9] O. Borůvka. O jistém problému minimálním. (English: On a certain minimal problem). *Práce Moravské Přírodovědecké Společnosti*, 3:37–58, 1926.
- [10] R.L. Brooks. On colouring the nodes of a network. *Proceedings of the Cambridge Philosophical Society*, 37:194–197, 1941.

- [11] S. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, OH, USA, 1971. ACM Press.
- [12] Th.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, Cambridge, MA, USA, second edition, 2001.
- [13] N.L. Cramer. A representation for the adaptive generation of a simple sequential programs. In J.J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Pittsburgh, PA, USA, 1985. Carnegie-Mellon University.
- [14] C. Darwin. *On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*. Murray, London, United Kingdom, 1859.
- [15] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating clique is almost NP-complete. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 2–12, San Juan, Puerto Rico, 1991. IEEE Computer Society Press.
- [16] L. Fogel, A. Owens, and M. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, USA, 1966.
- [17] R. Friedberg. A learning machine, part i. *IBM J. Research and Development*, 1958.
- [18] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.
- [19] J. Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 627–636, Burlington, VT, USA, 1996. IEEE Computer Society Press.
- [20] J. Holland. *Adaption in Natural and Artificial Systems*. MIT Press, Cambridge, MA, USA, 1992.
- [21] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, USA, 1979.
- [22] W. Johannsen. The genotype conception of heredity. *The American Naturalist*, 45:129–159, 1911.
- [23] R. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, NY, USA, 1972. Plenum Press.
- [24] K.E. Kinnear Jr. Generality and difficulty in genetic programming: Evolving a sort. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 287–294, San Francisco, CA, USA, 1993. University of Illinois at Urbana-Campaign. Morgan Kaufmann.

- [25] J.R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N.S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers, Inc.
- [26] J.R. Koza. *Genetic Programming, On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
- [27] J.B. Kruskal. On the shortest spanning subtree and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.
- [28] C.E. Lemke. Bimatrix equilibrium points and mathematical programming. *Management Science*, 11:681–689, 1965.
- [29] M. Locatelli, I.M. Bomze, and M. Pelillo. The combinatorics of pivoting for the maximum weight clique. *Operation Research Letters*, 32:523–529, 2004.
- [30] A. Massaro, M. Pelillo, and I.M. Bomze. A complementary pivoting approach to the maximum weight clique problem. *SIAM J. Optim.*, 12:928–948, 2002.
- [31] T.S. Motzkin and E.G. Straus. Maxima for graphs and a new proof of a theorem of turán. *Canadian Journal of Mathematics*, 17:533–540, 1965.
- [32] R. Nabuurs. Energy-bound Genetic Programming. Master’s thesis, Leiden Institute of Advanced Computer Science, 2004.
- [33] R.C. Prim. Shortest connection networks and some generalisations. *Bell System Technical Journal*, 36:1389–1401, 1957.
- [34] I. Rechenberg. *Evolutionsstrategie '93*. Frommann Verlag, Stuttgart, Germany, 1994.
- [35] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, USA, second edition, 2002.
- [36] H.-P. Schwefel. *Evolution and Optimum Seeking, Sixth-Generation Computer Technology Series*. John Wiley & Sons, New York, USA, 1995.
- [37] T. Soule, J.A. Foster, and J. Dickinson. Code growth in genetic programming. In J.R. Koza, D.E. Goldberg, D.B. Fogel, and R.L. Riolo, editors, *Genetic Programming 1996: Proceedings of the first Annual Conference*, pages 215–233, Stanford University, CA, USA, 1996. MIT Press.
- [38] A.S. Wu and R.K. Lindsay. A survey of intron research in genetics. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving From Nature IV. Proceedings of the International Conference on Evolutionary Computation*, volume 1141, pages 101–110, Berlin, Germany, 1996. Springer-Verlag.