Applying Adaptive Evolutionary Algorithms to Hard Problems

J.I. van Hemert[1]
jvhemert@wi.leidenuniv.nl

August, 1998

[1]Department of Computer Science, Leiden University, P.O. Box 9512, 2300 RA Leiden, The Netherlands

Abstract

This report is based on the work I have done for my Master Thesis project. The project as a whole consists of research done in the field of evolutionary computation, and it is split into two distinct parts. The main theme is adaptive evolutionary algorithms.

The first part covers the research done on solving binary constraint satisfaction problems using adaptive evolutionary algorithms. This involves a comparative study on three algorithms, each of which incorporates a different adaptive fitness measure to guide its search to a solution for an instance of a binary constraint satisfaction problem.

The second part mainly consists of the development of a library. Its use is aimed at evolutionary algorithms in general. Furthermore, a genetic programming algorithm is contructed, that incorporates an adaptive fitness measure. This construction served as a test of the usability of the library. The genetic programming algorithm has been used for experiments on different data sets from the data mining field.

Contents

Chapter 1

Introduction

1.1 What to expect

This report has been written for the conclusion of my Master Thesis. During the time I wrote this, I have been assisted and guided by A.E. Eiben and E. Marchiori. They both did a great job in correcting my mistakes and even more important, they frequently pointed me into the right direction. The research preceding and described in this report has also resulted in two articles:

A.E. Eiben, J.K. van der Hauw, J.I. van Hemert. Graph Coloring with Adaptive Evolutionary Algorithms. *Journal of Heuristics*, 4(1):25–46.

A.E. Eiben, J.I. van Hemert, E. Marchiori, A.G. Steenbeek. Solving Binary Constraint Satisfaction Problems using Evolutionary Algorithms with an Adaptive Fitness Function. In A.E. Eiben, Th. Bäck, M. Schoenauer, H.-P. Schwefel editors, *Proceedings of the 5th Conference on Parallel Problem Solving from Nature*, number 1498 in LNCS, pages 196–205, Springer, Berlin.

The whole project, and therefore this whole document, has evolutionary computation as a leading theme. In the field of evolutionary computation one uses evolutionary algorithms to solve some kind of problem. The process of getting the answer forms the computation part. The next section provides some inside into evolutionary algorithms for those who have never heard about it.

To make things even more difficult, the document is split into two parts. Both have some things in common, which comes down to adaptive evolutionary algorithms and me doing something with it. But because both are separate research fields and because both were independently researched, we decided to split the report into two parts. The order of these parts is the same as the order in which both researches were carried out.

The first part is about solving constraint satisfaction problems with evolutionary algorithms. In short a constraint satisfaction problem is a bunch of variables and a bunch of constraints. Each of the variables have to be assigned a value, this value is from a fixed and predefined domain. But some combinations of assignments of values are prohibited. These combinations are defined in the constraints. Solving a constraint satisfaction problem means finding values for all variables without violating the constraints.

The second part is focused on the construction of a library for programming evolutionary algorithms. The library aims at evolutionary algorithms as a whole, providing a broad spectrum of techniques from different fields of evolutionary computation. Its usability is tested on the construction of genetic programming algorithm that makes use of an adaptive fitness measure. The genetic programming algorithm is then tested on different data sets from the field of data mining.

1.2 Evolutionary Algorithms

As often is observed when a new field of research is maturing a clutter of names starts to form. After a while the names that have been formed will be connected with the founders and their own research. The common divisor of all these smaller research fields is then given a much broader name and it will be seen as an umbrella for all all these fields.

The same goes for the *evolutionary computation* field. When someone talks about evolutionary algorithms he implicitly says: ‘The field concerning genetic programming, genetic algorithms, evolutionary programming, evolutionary strategies, simulated annealing and classifier systems’. Evolutionary computation is all about evolutionary algorithms, search algorithms based on the theory of evolution by Charles Darwin. All of the research done inside this field is based upon a driving force we can witness in nature: *evolution*. But what is evolution about? It uses a basic principle which guides it towards a goal; ‘Survival of the fittest’. By selecting those individuals from a population that are closest to the goal, a pressure is created that drives the population to the goal which seems best for survival of the individuals and the population.

This principle would not be as powerful as we see today if things like genetic operators would not exists. These operators, such as mutation and crossover, make sure that during the lifetime of a population new individuals are formed that resemble their parents, but have some new information of their own. The operators provide the new ideas, while the natural selection makes sure, the best ideas survive.

An evolutionary algorithm tries to mimic this behavior by creating an artificial environment in which artificial individuals try to survive. By carefully constructing the environment it is possible to let a population of these individuals pursue a special goal. A goal which we want it to reach; the solution of a problem. The optimizing power of an evolutionary algorithm is so strong that it can solve all kind of optimizing problems. After all, it is evolution that is responsible for the creation of the most powerful problem solver known to mankind: ‘the human brain’.

Now that evolutionary computation is becoming increasingly popular, the number of practical applications is growing rapidly. The optimizing property of evolutionary algorithms is used for scheduling transportation of goods, planning lectures, finding the best shape of all kind of constructions and data mining purposes. Evolutionary algorithms are even used in an application that finds the best coffee by mixing several different blends. A panel of experts grades the coffee, which is then used as a fitness measure.

The field of evolutionary computation is still in the process of getting recognition of the business community. Looking at the great number of successes it has witnessed, undoubtedly this recognition will eventually be there. Who knows what evolutionary algorithms will and can do for us in the future?

Part I

Solving Constraint Satisfaction Problems with Adaptive Evolutionary Algorithms

Chapter 2

Introduction

This part of the report handles the experiments in solving constraint satisfaction problems. These constraint satisfaction problems will be represented in a binary form, which allows for a random generation using a set of parameters. The parameters have already been investigated in other experiments, and therefore allow us to make a comparison on results.

To solve the constraint satisfaction problems we use three evolutionary algorithms, where each algorithm makes use of a form of adaptivity. The three algorithms are the co-evolutionary method of Paredis (Paredis, 1994; Paredis, 1995b; Paredis, 1995a), the heuristic-based microgenetic method of Dozier, G. et al. (Dozier et al., 1994) and the stepwise adaptation of weights technique by Eiben et al. (Eiben et al., 1995; Eiben and Ruttkay, 1996; Eiben and van der Hauw, 1996; Eiben and van der Hauw, 1998; Eiben et al., 1998a).

The performance of these algorithms in successfully solving the constraint satisfaction problems is measured and compared. Furthermore we will see that the empirical results will conform with the theoretical assumptions made on the hardness of these kind of problems.

We compare the different algorithms using a test suit consisting of 625 generated problems. The results are then evaluated for 25 different parameters settings. The two best algorithms will be compared in a scale-up test, where we look at how the performance changes when the size of the problem grows.

The following chapter will explain more about what constraint satisfaction problems are. Chapter 4 will show the difficulty of solving constraint satisfaction problems and how to generate them randomly. The three algorithms will be explained in Chapter 5. The experiments and the results will be shown in Chapter 6, followed by the conclusions in Chapter 7. The last chapter will discuss future research.

Chapter 3

Constraint Satisfaction Problems

3.1 What is a CSP

When facing the task of solving a *constraint satisfaction problem* (CSP), the problem is to find values for a given set of variables, without violating constraints that exist between those variables. The number of variables is fixed and each variable gets its value from a finite domain. In a CSP, a constraint can exist between any set of variables, but here we only examine *binary constraint satisfaction problems*, i.e., constraints between two variables. This does not restrict our research because any CSP can be transformed into a binary CSP (Tsang, 1993).

Beside CSPs, another well known problem class is that of *constraint optimization problems* (COP). These two classes are similar except that a COP has an additional function that has as input the values of the variables in a possible solution. In addition to satisfying the constraints in the problem, this function has to be minimized as well.

On the first sight it looks like a problem from the class of COPs will generally be harder to solve than a problem from the class of CSPs. But this is not the case. Just like any other optimization problem, a COP can be transformed into a decision problem, in this case a CSP. The decision problem will be as hard to solve as the optimization problem.

Looking at the theoretical base of CSPs and COPs, there is an extra difficulty in solving CSPs using evolutionary algorithms. To be able to solve a COP, an evolutionary algorithm will try to solve the problem using a fitness objective, this function can be based or maybe even completely the same as the function that comes with the COP. An evolutionary algorithm needs a function to optimize, i.e., to base its selection mechanism on. It is therefore quite naturally to provide the COP's function as the fitness function. When an evolutionary algorithm has to solve a CSP, there is no function to base the fitness of a possible solution on. Therefore a completely new function has to be made up for the evolutionary algorithm to optimize.

3.2 An example: Eight Queens

A well known example of a binary CSP is the *Eight Queens Problem*. Here we want to place eight queens on a chess-board, such that they can not check each other. We know that if we find a solution, every column will contain precisely one queen, therefore each queen is assigned to one column. Every queen gets a number corresponding to the row it is placed in.

Speaking in terms of binary CSPs, between every pair of queens there is a set of two-tuples, which determine the combinations of values that are not allowed. For the first and second queen the following combinations may not occur: {(1,1), (1,2), (2,1), (2,2), (2,3), (3,2), (3,3), (3,4), (4,3), (4,4), (4,5), (5,4), (5,5), (5,6), (6,5), (6,6), (6,7), (7,6), (7,7), (7,8), (8,7), (8,8)}. For example, the (3,4) tells us that it is not allowed to place a queen in the first column, third row and at the same time place a queen in the second column, fourth row. But there is no objection to place a queen in the first column of the first row and a queen in the third row of the second column, because the tuple (1,3) is not in the set of forbidden combinations. The tuples in the set of forbidden combinations represent the conflicts. Remember that for this problem we need a set of forbidden combinations for each pair of queens. Every queen is a potential danger to any other queen. The amount of sets will therefore be $\frac{1}{2} \cdot 8 \cdot (8-1) = 28$.

A solution of the Eight Queens Problem is a vector of eight values, these values are the row number where the queen in that column is placed. One of the solutions looks like this: $(2, 4, 6, 8, 5, 7, 1, 3)$, the board-representation can be found in Figure 3.1.

1							◇	
2	◇							
3								◇
4		◇						
5					◇			
6			◇					
7						◇		
8				◇				

Figure 3.1: A solution to the Eight Queens Problem.

3.3 A formal definition

A constraint satisfaction problem is defined as a tuple $\langle S, \phi \rangle$, where S is called the search space and ϕ is a Boolean function taking as input the values of the variables. The search space S is defined as the Cartesian product of variable domains; $S = D_1 \times \cdots \times D_n$, for a CSP with n variables. We define the *solution* of a constraint satisfaction problem s in terms of ϕ, where $s \in S$. $\phi(s)$ returns `true` if and only if s is a solution for the CSP.

We will only consider binary CSPs, which enables us to define ϕ as a conjuncture $c_1 \wedge c_2 \wedge \cdots \wedge c_m$ of binary constraints c_k. A binary c_k can be defined as a three tuple $c_k = \langle v_i, v_j, C_k \rangle$. Here the constraint c_k is upon the variables v_i and v_j with $i < j$, without loss of generality. C_k is a matrix of size $|D_i| \cdot |D_j|$, where each element $C_k(x, y) \in \{0, 1\}$. If $C_k(x, y) = 1$ we speak of a conflict, which means the instantiation $v_i = x$ and $v_j = y$ may not occur, thus resulting in $\phi(v) =$ `false`.

Through this report we will also use $m(v_i)$ to denote the size of the domain D_i of variable v_i, instead of writing $|D_i|$. Furthermore m is used when we mean the average size of the domain size over all the variables, it is defined as follows:

$$m = \frac{\sum_{i=1}^{n} m(v_i)}{n}$$

3.4 A general binary CSP

To illustrate what a binary CSP looks like, we give at an example. We take five variables $v_1, \ldots, v_5$, where each variable has a domain $D_i = \{1, 2, 3, 4\}$. Note that the domains of the variables do not have to be the same in a CSP, nor do they have to have the same size. There are three constraints c_1, c_2 and c_3, each of which relates to two variables v_i and v_j, and with a matrix C_k representing the conflicts. This is denoted as the three-tuple $\langle v_i, v_j, C_k \rangle$. Here are the constraints in the example: $c_1 = \langle v_1, v_2, C_1 \rangle$, $c_2 = \langle v_2, v_3, C_2 \rangle$ and $c_3 = \langle v_2, v_4, C_3 \rangle$.

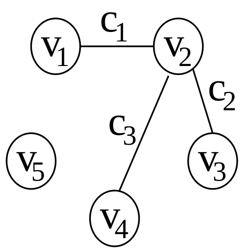

Figure 3.2: The variables and constraints in the binary CSP represented as a graph.

The variables and the way in which the constraints connect these variables can be represented as a graph. The nodes of the graph will represent the variables, the edges will represent the relations as defined by the constraints (Figure 3.2). More precisely, there is an edge (v_i, v_j) in the graph between variable v_i and v_j with $i < j$ iff $\langle v_i, v_j, C \rangle$ is a constraint of CSP for some C. Figure 3.3 shows the conflicts between the values of the variables in the constraints. For instance the matrix C_1 belonging to the constraint c_1 tells us that the variables v_1 and v_2 can not be assigned the value 2 at the same time.

C_1	1	2	3	4
1	0	0	1	0
2	1	1	0	1
3	0	0	1	0
4	0	1	1	1

C_2	1	2	3	4
1	0	1	0	1
2	1	1	0	0
3	0	0	0	0
4	0	0	1	0

C_3	1	2	3	4
1	1	0	0	0
2	0	1	1	0
3	0	0	0	0
4	0	0	0	1

Figure 3.3: Matrices for C_1, C_2 and C_3.

Chapter 4

Generating Random CSPs

4.1 Introduction

In the field of constraint satisfaction problems, a lot of experiments have been done on specific instances of CSPs, for instance on n-queens problem (Dozier et al., 1994; Paredis, 1995a; Homaifar et al., 1992; Crawford, 1992), on graph coloring (Davis, 1991; Eiben and van der Hauw, 1996; Eiben et al., 1998a; Fleurent and Ferland, 1996a; Fleurent and Ferland, 1996b), and on satisfiability (Selman et al., 1992; Eiben and van der Hauw, 1997; Selman and Kautz, 1993; Bäck et al., 1997; Bäck et al., 1998). These experiments mostly consist of comparisons with other techniques that solve the same problem. The drawback of this method of testing is that it is not possible to say what kind of problems is easily tackled by a method and what kind is not. It is only possible to speculate about the performance of a method on other problems than those used in the experiments.

These speculations can only be based upon a comparison between the domains[1] of the problems on which the technique was tested and the domains of the problems where we want to say something about the expected performance of the technique. However, this approach is not very effective, since most problems have a search space that is very hard to get a grip on, let alone compare it to other search spaces.

Nevertheless, it would be very interesting to know if a method will in general perform better on a class of problems than other known techniques. With this knowledge it would be easier to decide which technique to use when facing the task to solve a new problem from this class. Especially when the task is to come up with a method that solves the problem faster then the currently used method. The best performing techniques within the class could then be compared, and the winner could do the job. This way no extensive study into the search space of a new problem is needed to find good techniques to solve it.

[1] Here the domain as defined by a problem, not the domain of the variables of a problem.

We have to point out that in general it is not possible for any algorithm to have a better performance than another algorithm. Wolpert and Macready (Wolpert and Macready, 1995; Wolpert and Macready, 1997) have shown that all algorithms that try to optimize a function, will perform equally when averaged over all possible cost functions. This result, which they have named the *No Free Lunch Theorem*, is informally discussed by Culberson (Culberson, 1996), where he states that solving a collection of problem instances from a NP-complete problem does not solve the NP-problem itself. Often a NP-complete problem has a set of instances which are very easy to solve. Our results will show that with binary CSPs this claim also holds.

This gives rise to the question of how to collect information about a method such that it has a general content. One way would be to test the method on a test case consisting of problems from the whole class of problems we want to test. This can be achieved by constructing a problem representation that can be used to represent any problem in the class of problems. In the case of a CSP this is always possible, because every CSP can be represented by an equivalent binary CSP (Tsang, 1993). But the transformation can result in very complex domains for the variables. Binary CSPs with variables over a finite domain can be generated using a so called random method. As every binary CSP can be produced in the process, every CSP we are able to model using the formalism from Chapter 3, can be produced this way. A binary constraint satisfaction problem created this way is called a *random binary constraint satisfaction problem*. In section 4.3 two methods for generating CSPs will be shown.

4.2 Difficulty of a CSP

4.2.1 Measuring hardness

To compare techniques, information about the hardness of a problem is needed. Two sorts of comparing measures lie beforehand, firstly the amount of space needed to solve the problem, and secondly the amount of time needed to solve the problem. The focus will lie on the second measure, mainly because small binary CSPs take far more time to solve compared to the amount of space they need, especially because of the simple representation of the solution inside of the evolutionary algorithms we will use. The representation mainly consists of a possible solution, i.e., an instantiation of the variables in the CSP. Furthermore the problem needs to be accessible so we can check the possible solutions for validity.

The size of the problem increases quadratically with the number of variables and also quadratically with the size of the domain of the variables. The search space, i.e., the number of possible solutions grows exponentially with regard to the number of variables n: m^n where m represents the average domain size of all the variables.

4.2.2 NP-completeness

The reason that solving CSPs is a very difficult task, whether randomly generated or specific ones, lies in the fact that CSPs are member of a class known as the class of NP-*complete* problems. The problems inside this class are characterized by two facts. Firstly, given some solution to an instance of an NP-complete problem, it should be possible to verify that it is correct in polynomial time. Secondly, all problems in this class can be reduced to any other NP-complete problem, using a function that can be evaluated in polynomial time as well.

One of the problems from the NP-complete class, the famous satisfiability problem (SAT), has been proven to be NP-complete. There is no known algorithm that solves this problem in polynomial time. If it would exist, every NP-complete problem would be solvable in polynomial time as well, because of the reduction mechanism.

Proving that binary CSPs belong to the NP-complete problems class, would imply the need for a verification algorithm and a reduction. It is easy to see that the verification algorithm will work in polynomial time, because a binary CSP has at most $\frac{1}{2}n(n-1)$ constraints. Each constraint will have to be checked once, which can be done in quadratic time, with respect to the domain sizes of the variables. By carefully choosing a problem, the reduction becomes easy as well. If we take a specific problem from the class of binary CSPs that is known to be NP-complete, like graph coloring or the n-queens problem, we can almost copy the problem into the binary CSP model.

4.2.3 Where are those hard CSPs?

Parameters of a problem class

If a problem has been identified as belonging to the class of NP-complete problems, this does not mean that every instance of this problem is hard to solve. The parameters of a NP-complete problem, i.e., the parameters that define an instance from a class of problems, determine in a sense the difficulty of this instance. When the parameters are changed another instance may be created that could be more, less or just as difficult to solve.

A trivial example of this is when we look at an instance consisting of just a few variables. A binary CSP with only two variables that have a domain size of one can be easily solved. And even if there is no solution we can verify that this is the case. Here there are only three possibilities:

1. No constraint present — solution
2. A constraint, without a conflict — solution
3. A constraint, with a conflict — no solution

When looking at binary CSPs with more variables and with larger domain sizes, the number of possible solutions grows exponentially when one of these parameters is increased. It gets much harder to find a solution for particular problem instances and it also gets much harder to find out if a solution does exist.

The landscape of solvability

If the number of variables and the domain sizes are fixed, experiments reveal an interesting phenomenon. When randomly generating binary CSPs, some instances appear easy to solve, while others are quite a hard nut to crack. This raises the question "What makes a binary CSP hard to solve?".

In a number of articles Prosser et al. (Prosser, 1994; Prosser, 1996; Gent et al., 1995) uses a method of exploring the landscape of the chance on finding a solution for a binary CSP. By using four parameters he creates a three dimensional view on this landscape. From these four parameters, two are fixed; the number of variables and their domain sizes. He defines the density and the tightness of a problem and uses these to show the probability that a problem instance has a solution. The first parameter, the *density of a binary* CSP, is defined as the probability that a constraint exists between two variables. The second parameter, the *tightness of a binary* CSP, is defined as the probability of having a conflict between two given variables in a constraint.

Setting the number of variables to 15 and the domain size of each variable to 15, the two parameters density and tightness are varied throughout their real valued domains, varying from zero to one. This gives a very remarkable landscape. For low values of the parameters the chance a solution exists is almost one. This remains so when the values are increased, until a certain point. When this point is reached a steep curve takes the chance of having a solvable problem from almost one to almost zero. This area is called the *mushy region* or *phase transition*, it can be observed in Figure 4.1. It marks the region where pairs of the parameters density and tightness create problems which have a chance of having a solution somewhere between zero and one.

The results from Prossers' work confirm what Smith (Smith, 1994) conjectures. Smith estimated the expected number of solutions given a binary CSP, knowing that there are $\frac{1}{2}dn(n-1)$ constraints and m^n possible instantiations, with the following equation:

$$E = m^n(1-t)^{\frac{1}{2}dn(n-1)} \tag{4.1}$$

Smith thinks that the hardest problems would occur when $E = 1$, i.e., when the problem probably has one solution. This enables us to find the critical value for the tightness: $\hat{t}_{crit}$:

$$\hat{t}_{crit} = 1 - m^{-2/d(n-1)} \tag{4.2}$$

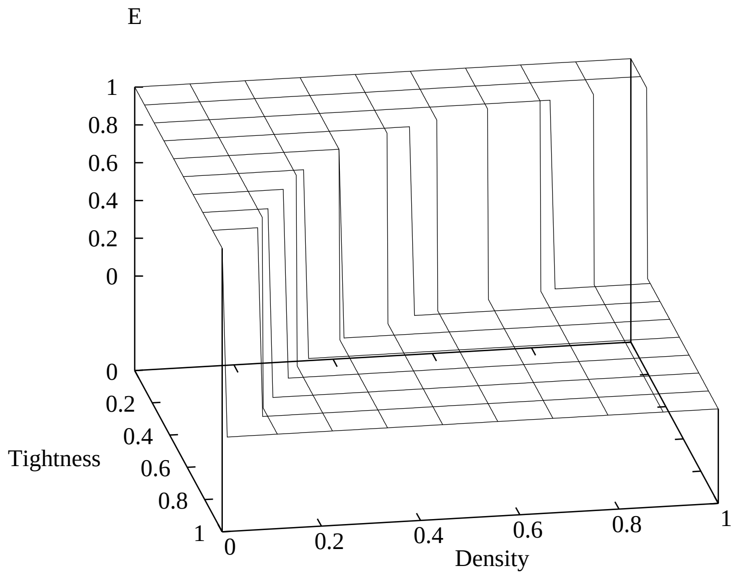

Figure 4.1: Mushy region as predicted theoretically.

Prosser (Prosser, 1996) shows that the values found in his experiments agree with the values predicted with Equation 4.2, except for low values of d ($d < 0.3$).

4.3 Two methods for generating CSPs

The following sections describe two slightly different methods for generating random binary CSPs. Both method have been implemented in C++ and are available as a library or problem instance generator. The package is called *RandomCsp*, and can be downloaded from the Internet (see Appendix A). Also documentation on the library and the problem instances generator is available (van Hemert, 1998a).

4.3.1 Method by Prosser

The first method for creating random binary CSPs is developed and used by Prosser in a number of articles (Prosser, 1994; Gent et al., 1995) where some exact algorithms, i.e., algorithms that are able to tell if a problem has a solution, for solving binary CSPs are compared. The method starts by calculating the exact number of constraints that will be produced.

$$\textit{number of constraints} = \frac{n(n-1)}{2} \cdot d, \tag{4.3}$$

where n is the number of variables and d is the constraint density

The method generates a random number between zero and one for every combination of two variables. When this number is below the density of the graph, a constraint is added. When the number of constraints calculated with Equation 4.3 is reached the algorithm terminates.

For every constraint that is generated, a matrix of conflicts has to be produced. This matrix is build by generating a random number in the range of [0..1) for every possible pair of values of the two variables. When that number is lower then the chosen tightness, a conflict is produced and stored in the matrix. An overview of the method is given in Algorithm 1.

Algorithm 1 Generating binary CSPs using Prossers' method.

```
constraints = density * variables * (variables - 1) * 0.5;
while (constraints > 0)
{
  i = 0;
  while ((constraints > 0) && (i < variables - 1))
  {
    j = i + 1;
    while ((constraints > 0) && (j < variables))
    {
      if ((random(1.0) < density) && no_edge(i, j))
      {
        constraints--;
        add_constraint(i, j);
        for (x = 0; x < domainsize; ++x)
          for (y = 0; y < domainsize; ++y)
            if (random(1.0) < tightness)
              add_conflict(i, j, x, y);
      }
      i++;
    }
  }
}
```

4.3.2 Method by Dozier

The method developed by Dozier has been used in his research on solving random binary CSPs with a heuristic-based microgenetic algorithm (MID) (see Section 5.2). It uses roughly the same technique as the method in section 4.3.1, but there are two differences:

1. The method for choosing between which variables constraints are added.
2. The way in which conflicts are produced.

The individual constraints are produced by randomly selecting two distinct variables, if no constraint exists between them a constraint is created. This is repeated until we have created the number of constraints determined in Equation 4.3.

Just as the number of constraints is determined in advance, so is the number of conflicts that are generated for each constraint. When a constraint has been produced, the number of conflicts is determined with the following equation, where v_i and v_j are two distinct variables and t is the tightness:

$$\begin{gathered} \textit{number of conflicts}(v_i, v_j) = m(v_i) \cdot m(v_j) \cdot t, \\ \text{where } m(v_i) \text{ is the domainsize of variable } i \end{gathered} \tag{4.4}$$

The same procedure to generate conflicts is used as in generating constraints. Two random values are chosen, one for each variable. If no conflict exists between them, a conflict is added to the matrix of conflicts for this constraint. This is repeated until there are as much conflicts as calculated with Equation 4.4. An overview of the method can be found in Algorithm 2.

Algorithm 2 Generating binary CSPs using Dozier's method.

```
constraints = density * variables * (variables - 1) * 0.5;
while (constraints > 0)
{
  do
  {
    i = random_integer() % n;
    j = random_integer() % n;
  }
  while (i != j);

  if (!constraint_exists(i, j))
  {
    add_constraint(i, j);
    conflicts = tightness * domainsize(i) * domainsize(j);
    while (conflicts > 0)
    {
      x = random_integer() % domainsize(i);
      y = random_integer() % domainsize(j);
      if (!conflict_exist(i, j, x, y))
      {
        add_conflict(i, j, x, y);
        conflicts--;
      }
    }
  }
  constraints--;
}
```

Chapter 5

Three Algorithms

This chapter consists of the explanation of the three algorithms that were tested in the experiments. All three are evolutionary algorithms, but each one employs a different technique to help improve its performance on solving CSPs.

The three algorithms share one common feature, they all make use of *adaptation*. Which means that the population has to work with a fitness objective that changes throughout the lifetime of the algorithm. Each algorithm introduces its own way of adaptation. The co-evolutionary algorithm uses two populations, that are entangled in a constant arms race. MID uses two techniques, it provides information to offspring from their parents. And it maintains a mechanism which evolves during the lifetime of the system, it is used to help individuals escape local optima. The SAW-ing technique directly alters the fitness function thereby influencing the pressure of selection in the population.

5.1 Co-evolutionary Algorithm

5.1.1 What makes up a co-evolutionary algorithm

This method uses a technique that is based on a natural phenomenon. When a population evolves in an environment, this environment often changes, partly because of the way the population interacts with its environment. An environment will for instance contain other species. One of the interactions will be the way in which the population reacts with one of the other species inside the environment. Such as the way in which a population of predators evolves better techniques to get its prey, and in return the prey will try to counter these techniques and evolve its own methods for a better chance of survival. This kind of interaction, which resembles an arms race, is a form of co-evolution.

This is the process on which Paredis has based his *Co-evolutionary approach to Constraint Satisfaction* (CCS) (Paredis, 1994; Paredis, 1995b; Paredis, 1995a). This approach uses a system called *Lifetime Fitness Evaluation* (LTFE).

A co-evolutionary algorithm based on CCS consists of two populations, the *solutions population* and the *constraints population*. The solutions population consists of individuals that represent possible solutions to the CSP that is being handled by the algorithm. The constraints population consists of all constraints that are in the CSP.

To let these two populations interact the LTFE is used. It provides both populations with a fitness value to base selections on. The fitness of an individual in either of the populations is based on a history of *encounters*. An *encounter* occurs when an individual from one of the populations is paired with an individual from the other population. In this co-evolutionary algorithm two things can happen. An individual from the solutions population is paired with a constraint, and the individual either violates the constraint or does not violate the constraint. If it does not violate the constraint it receives one point, otherwise it gets nothing. Likewise, the individual representing the constraint respectively gets nothing, otherwise it receives one point. The results of the encounters an individual makes are saved in its *history*, this history has a finite size, thus only part of the results of all encounters are registered.

Algorithm 3 The co-evolutionary algorithm.

```
i = 0;
while (i < encounters)
{
  solution = select(solutions_population);
  constraint = select(constraints_population);
  result = encounter(solution, constraint);
  update_history_and_fitness(solution, result);
  update_history_and_fitness(solution, 1 - result);
  i++;
}
parents = select(solutions_population);
offspring = crossover_and_mutate(parents);
evaluate(offspring);
insert(solutions_population, offspring);
```

The fitness of an individual is calculated from the history of the individual. It is simply the sum of all the points, i.e., the number of non-violated constraints in the history of this individual. Because of the inverse effect the encounters have on individuals from the opposite population an arms race is created where individuals that are able to withstand the most individuals from the other population gets the most chance of survival. The better individuals will be selected more often for an encounter, which results in better individuals getting paired more often with better individuals from the other population.

Before the co-evolutionary algorithm starts, the two populations will have to be initialized. This is done by letting all individuals in both populations have a fixed number of encounters. When this is done the main algorithm will start its work, as illustrated in Algorithm 3.

5.1.2 Techniques used in the implementation

The fitness function

As described in Section 5.1.1 the fitness function is calculated from the history of an individual. This history consists of the results of the encounters the individual has had over some finite time. It is represented as a vector of zeros and ones, where a one means the individual had success in its encounter and a zero means the it has failed to succeed. The history has a fixed and finite size $size(h)$. The fitness of an individual with history $h = \langle h_1, \ldots, h_{size(h)} \rangle$ becomes:

$$fitness(h) = \sum_{i=0}^{size(h)} h_i$$

To be successful in being selected, an individual has to obtain a high fitness value. When we talk about the *best* individuals, we mean the individuals with the highest fitness value, the *worst* individuals are those that have the lowest fitness value. This value can never exceed the size of the history and it can never be lower than zero.

The selection mechanism

The selection mechanism used in the implementation is called *linear ranked based selection*. In the implementation the linear function for selective pressure from an article of Whitley is used (Whitley, 1989).

$$linear() = \frac{population\_size \cdot \left(bias - \sqrt{bias^2 - 4\,(bias - 1) \cdot random()} \right)}{2/(bias - 1)},$$

where $random()$ generates a real number between 0 and 1.

The *linear* function is repeatedly used to determine which individual gets selected for crossover. The population is sorted, from best to worse fitness. The place an individual occupies in the sorted population is called its *rank*. Thus the first individual from the top has a rank of one, the second has a rank of two, and so on. Every time an individual needs to be selected, the linear function is called and the individual with the rank equal to the value returned by the linear function is selected. If the bias is higher than one, individuals at the top of the population get selected more often than individuals at the bottom. When the bias is one, the selection is completely random.

More precisely when the bias equals 1.5, it means that the best individual, i.e., the one at the top of the population, has one and a half as much chance of getting selected for reproduction than the median one. Whitley showed that a bias higher then 1.5 leads to premature convergence, while lower values did not push the algorithm they tested to a good optimum.

The crossover

The crossover used in the co-evolutionary algorithm is called *two point reduced surrogate parents crossover* (Whitley, 1989; Booker, 1987). It uses a special technique that tries to minimize the chance of generating offspring that look much alike. The operator is the same as the standard two point crossover operator, except for the way in which the crossover-points are chosen. The operator finds the first position such that the values at that position in both parents are different. It repeats this starting at the other side of the individual. The part that lies between these two points is where the two crossover-points will be chosen in.

An example can be found in Figure 5.1. Here the first difference in values is at position 3 (position numbering starts from left and with one as the first position), and looking from the other side, the first position is 6. The bottom line shows the positions. The part in which the crossover-points have to be chosen is marked bold in both parents. In the example two points are chosen and the offspring is created.

$$
\begin{array}{cc|ccc|cccccccccccccc}
\multicolumn{9}{c}{\text{Parents}} & & \multicolumn{9}{c}{\text{Offspring}} \\
9 & 8 & \mathbf{7} & \mathbf{6} & \mathbf{5} & \mathbf{4} & 3 & 2 & 1 & \rightarrow & 9 & 8 & \mathbf{6} & \mathbf{5} & \mathbf{4} & \mathbf{4} & 3 & 2 & 1 \\
1 & 2 & \mathbf{6} & \mathbf{5} & \mathbf{4} & \mathbf{3} & 7 & 8 & 9 & \rightarrow & 1 & 2 & \mathbf{7} & \mathbf{6} & \mathbf{5} & \mathbf{3} & 7 & 8 & 9 \\
{\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9} & & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} & {\scriptstyle 9}
\end{array}
$$

Figure 5.1: Two point reduced surrogate crossover.

The idea behind this operator is to prevent the generation of offspring that look almost the same, or worse are exact duplicates. If this would happen it could result in premature convergence. The operator also enables the algorithm to examine more possible solutions in the same number of runs, because more different individuals will be generated.

Another interesting aspect of this operator is that the two parents create offspring by exchanging information in the parts they do not agree on. The other two parts are copied without alteration, this ensures that the offspring does not lie very far from their parents in the search space.

The mutation

The mutation is a simple operator called *uniform mutation*. It is applied to freshly created offspring, and has one additional feature regarding the mutation probability p_m. A random number between zero and one is generated for each variable in the vector v. If this number is lower than the p_m the value is replaced with a random value from the domain of this variable. The mutation probability depends on the parents of the individual. If the parents are different, it is set to $p_m = 0.001$. If the parents are duplicates, the mutation probability is raised to $p_m = 0.01$.

5.1.3 Parameters of the algorithm

The co-evolutionary algorithm has a number of parameters. These parameters can be modified and may yield better or worse results. When an algorithm has a couple of parameters without knowledge of their exact influence on the performance, it is already very difficult to choose the right setting. For the parameters of the co-evolutionary algorithm the values are taken from articles of Paredis (Paredis, 1994; Paredis, 1995b; Paredis, 1995a). The parameters can be found in Table 5.1.

Parameter	Value
Solutions population size	50
Constraints population size	# constraints in CSP
Initial number of encounters	20
Number of encounters when running	20
History size	25
Parent selection	linear ranking
Bias for linear ranked selection	1.5
Replacement strategy	replace worse
Mutation probability with different parents	0.001
Mutation probability with duplicate parents	0.01
Crossover	2-point surrogate parents
Representation	integer-based
Stopcondition	solution found or maximum # evaluations

Table 5.1: Fixed parameters of the co-evolutionary algorithm.

5.2 Microgenetic Method

5.2.1 General idea

This algorithm gets its name from the small population size it uses. Together with the incorporation of the *Iterative Descent Method* (IDM) as described by Dozier et al. (Dozier et al., 1994; Dozier et al., 1995) its adopted name is *Microgenetic Iterative Descent* (MID).

The basic operator in this method is called the *value reassignment*, which is a kind of hill-climber algorithm. It works on one variable at a time, by reassigning a new value to this variable using a heuristic mutation operator. Two different objectives exist, the minimization of the number of constraint violations this variable causes, and the minimization of the total number of constraint violations the whole individual causes. In this research IDM is used, which minimizes the total number of constraint violations, it includes a Breakout Management Mechanism for escaping local optima.

The *Breakout Management Mechanism* (BMM) stores a list of breakouts. A *breakout* consists of two parts:

1. A 2-tuple called 'nogood' — this is really just a pair of values that violates a constraint. Recall that we are dealing with binary constraints.

2. A weight — also called the value of the breakout.

When IDM gets trapped in a local optimum (see Section 5.2.2), it invokes the Breakout Management Mechanism. For each pair of values from the individual that violates a constraint the BMM either creates a breakout or changes the breakout if it already exists. The weight of a newly created breakout is set to one. If the breakout already exists, the weight of the breakout is incremented by one. These weights are used in the fitness function, which will be explained in the next section.

5.2.2 Techniques used in the implementation

The representation

Before revealing the overall structure of an individual, we first describe a single allele within this structure. Each allele consists of four elements (Figure 5.2), the name (or number) of the variable, the assigned value, the number of constraint violations this variable is involved in and an h-value. This *h-value* is used in choosing the pivot variable of an individual. It is explained in the section on reproduction.

Variable name		
Value	$\longleftarrow$	undergoes heuristic mutation
Constraint violations	$\longleftarrow$	number of constraint violations this value assignment causes
h-value	$\longleftarrow$	used for choosing the pivot allele

Figure 5.2: Representation of an allele from an individual within MID.

The overall structure of an individual is made up of n alleles, one for each variable in the problem. The structure also consists of the fitness value and a pivot. The *pivot* determines which variable should be used for the value reassignment process and it is also used for a single point mutation operator.

Variable name	Variable name		Variable name
Value	Value		Value
Constraint violations	Constraint violations	$\cdots$	Constraint violations
h-value	h-value		h-value
Fitness			
Pivot			

Figure 5.3: Representation of an individual within MID.

The fitness function

To compute the fitness of an individual $x = \langle x_1, \ldots, x_n \rangle$, it counts the number of constraint violations each variable is involved in and then sums these all up. Added to this is the weight of each breakout that is violated by the individual.

$$\mathit{fitness}(x) = \sum_{i=1}^{n} V(i) + \sum_{i=1}^{n} \sum_{j=1}^{n} B(x_i, x_j)$$

Where $V(i)$ gives the number of violated constraints involving the i-th variable, and $B(x_i, x_j)$ returns either the weight belonging to the breakout of tuple $\langle x_i, x_j \rangle$, or zero when the breakout is not defined. We are trying to minimize the fitness function, which has an optimum at zero. The fitness of an individual is better than that of another individual if its fitness value is lower.

Initializing the population

Instead of randomly generating all the individuals to create the first generation for the population, a different scheme is adopted here. A first individual is created by randomly choosing values for all the variables. The remaining individuals are created by copying the first individual, after which a mutation is done on one variable in each individual. A random variable is chosen and then a random number is assigned from the domain of the variable.

When an individual is created, i.e., the variables have been assigned a value, the other items inside the individual have to be initialized. The h-value for each allele is set to zero and the number of constraint violations the allele is involved in is counted. Then the fitness is calculated and the pivot allele is chosen. The allele involved in the most constraint violations is chosen as pivot, where ties are broken randomly.

Heuristic mutation

After selecting an individual for reproduction the reassignment process will be run. This is a *single-point heuristic mutation* where a parent copies itself to produce an offspring and then mutates one allele. The pivot of the offspring points to the variable that will undergo the mutation. This variable is assigned a value chosen randomly from its domain. No other genetic operators are used.

The reproduction

The offspring that is created by the reassignment process is then compared to its parent. If the fitness of the parent is better than or equal to that of the offspring, the h-value of the corresponding pivot allele of the offspring is decremented by one. And the individual is inspected to see if the pivot should point to another allele. This is done by computing the *s-value* of each allele, which is calculated by summing the number of constraint violations of this allele and its h-value. The allele with the highest s-value will be appointed as the new pivot. If there is a tie between the current pivot and one or more other alleles, the current allele stays pivot. Ties between other alleles are broken randomly. If the fitness of the parent is not better than that of the offspring, the h-values and thus the pivot is left unchanged.

Using this method of inheriting information for choosing which allele is to be mutated provides two interesting mechanisms for the algorithm to exploit. First of all, a consecutive line of successful offsprings can optimize the number of constraint violations related to one variable. Secondly, it allows the algorithm to switch to other variables when this optimizing stops or when other variables have higher s-values.

On the other hand, the method also poses a problem, after a while it is possible that the h-value causes the system to choose an allele that is not involved in any constraint violations. This happens when the h-values of the variables that are involved in constraint violations get lower than the actual number of constraint violations. If the algorithm would reach this state, no further progress will be made. In order to prevent this from happening, the h-values will be reset to zero using a probability function r_x for an individual x:

$$r_x = \frac{1}{|O_x| + 2}$$

where O_x is the amount of variables involved in constraint violations caused by individual x.

In addition to the heuristic mutation operator mentioned here before, the algorithm makes use of one genetic operator as well. It is a single-point uniform mutation operator, that is applied to the freshly generated offspring. A value from the domain of the variable of the allele the pivot is pointing at is randomly generated. This value is than assigned to the variable belonging to the pivot allele.

Calling the Breakout Management Mechanism

As mentioned in Section 5.2.1, the algorithm makes use of a breakout system to escape local optima. This implies that the algorithm must have a way of deciding when to invoke this system. The system used here is identical to that used by Morris (Morris, 1993). Let $m(i)$ give the domain size of variable i, and let V be the set of variables involved in constraint violations caused by the best individual in the population. If we define y as the number of consecutive offsprings created which do not have a better fitness than the fitness of their parents, the BMM will be invoked when the following equation $y > \sum_{i \in V} m(i)$ holds.

Once invoked, the BMM starts creating breakouts using the best individual in the population. When a breakout already exists, its weight is incremented by one. When the BMM is finished, all the individuals inside the population have to be reevaluated. Furthermore the counter y has to be reset to zero again.

5.2.3 Parameters of the algorithm

This algorithm does not have a lot of parameters, the ones mentioned in Table 5.2 are mostly taken from articles of Dozier et al. (Dozier et al., 1994; Dozier et al., 1995). Some of them are slightly different, to boost performance, but they are still close to the originals.

Parameter	Value
Population size	8
Parent selection	roulette-wheel
Replacement strategy	replace worse (8+1)
Mutation	one point with heuristic
Crossover	none
Representation	integer-based
Stopcondition	solution found or max. evals.

Table 5.2: Fixed parameters of the Iterative Descent Method.

5.3 Stepwise Adaptation of Weights

5.3.1 Where SAW-ing is based on

The *Stepwise Adaptation of Weights* (SAW) mechanism first appeared in (Eiben et al., 1995; Eiben and Ruttkay, 1996) and is invented by Eiben et al. It has been used in numerous experiments to solve different kind of problems, such as satisfiability problems (Eiben and van der Hauw, 1997; van der Hauw, 1996; Bäck et al., 1997), bin-packing (Vink, 1997) and graph coloring (van der Hauw, 1996; Eiben and van der Hauw, 1996; Eiben and van der Hauw, 1998; Eiben et al., 1998a).

The SAW-ing mechanism is an adaptive technique, which uses the idea that after some generations, the constraints that are violated by the best solution, i.e., the individual with the best fitness value, must be hard. By increasing the penalty for violating these constraints, it gets more rewarding for all individuals to make sure these constraints are not violated. Thus focusing the search on the harder constraints.

Previous research from Eiben et al. (van der Hauw, 1996; Eiben and van der Hauw, 1996; Eiben and van der Hauw, 1997; Eiben et al., 1998a) show that the best performing algorithm with the SAW-ing mechanism makes use of an order-based representation with a decoder. The decoder takes as input an individual in order-based representation and delivers a partial solution for the CSP. As we will show in Section 5.3.2 this makes it impossible to count the number of violated constraints, to overcome this we will count the number of unassigned variables. Thus the fitness function works on variables, not on constraints.

To represent the penalties a system of weights is used. Every variable i is assigned a *weight* w_i. Initially these weights are all set to one. After some generations the algorithm is interrupted and the best individual is evaluated. The weight of every variable that has not been assigned a value by this individual is increased with Δw. The fitness function will incorporate the w_i values, such that when an individual cannot assign a value to a variable i, the w_i will have a negative influence on the fitness function. This technique closely resembles the original idea where every constraint is paired with a weight, and where a weight is changed when the corresponding constraint is violated.

One of the consequences of this technique is that we will have to decide when to interrupt the algorithm to update the weights belonging to the variables. In the first version called *Offline*-SAW (Eiben and Ruttkay, 1996; Eiben et al., 1995), the algorithm was stopped and the best individual is then used to make changes to the weights. The algorithm then had to be restarted by hand using the new weights. The second version, which is used in this experiments, is called *Online*-SAW. It uses a parameter T_p to determine when to interrupt the evolutionary algorithm to update the weights. One of the results in (Eiben et al., 1998a) is that varying T_p in a range of 1 to 10000 has not much effect on the performance of the algorithm when tested on hard problem instances of the graph coloring problem for three colors. Another result shows that varying Δw from 1 to 30 also does not alter the performance of the algorithm.

5.3.2 Techniques used in the implementation

The representation

Instead of just representing the values of each variable, this algorithm uses an order-based representation. An *order-based representation* provides data that is then fed into a decoder. This decoder transforms this data into an actual solution. The individual is nothing more than a permutation of the variables. The decoder used is a greedy algorithm, which tries to assign every variable with the first possible value, starting with the lowest value, without violating constraints. If no value can be assigned without violating constraints, the variable is left unassigned.

The *decoder* tries to assign a value to each variable without violating a constraint. It does this by looking in turn at each variable and then for each variable tries every possible value from the domain of the variable until it finds a value that does not violate any constraint. Unfortunately this is quite a costly operator, because for every variable, and every value from its domain size, every other constraint between this variable and any other variable will have to be checked. For one variable this amounts to checking quite a lot of possible conflicts. The worst case is when there is no possible assignment, then the amount of checked conflicts for variable v_i is approximately:

$$\textit{checked conflicts}(v_i) = \frac{\frac{1}{2}n(n-1)d}{n}m = \frac{1}{2}(n-1)dm$$

An example of an individual that is decoded can be found in Figure 5.4. After the decoding we end up with values for the variables $v_1, \ldots, v_7$, except for variable v_6, it has a cross, which means no value could be assigned by the decoder without violating a conflict.

$$\begin{array}{ccccccccccccccc} \multicolumn{7}{c}{\text{Individual}} & & \multicolumn{7}{c}{\text{Partial solution}} \\ 2 & 3 & 7 & 4 & 1 & 5 & 6 & \overrightarrow{decoder} & 9 & 1 & 3 & 5 & 4 & \times & 4 \end{array}$$

Figure 5.4: Example of decoding an individual.

The fitness function

The use of the decoder means that violated constraints cannot be counted, because they will not occur. Instead we first decode the individual x to obtain a partial solution. Every variable v_i that has not been assigned a value in the partial solution is multiplied by its weight w_i. The results of these multiplications are then summed to produce the fitness of the individual x.

$$fitness(v) = \sum_{i=1}^{n} w_i \cdot \chi(v, i) \tag{5.1}$$

$$\text{where } \chi(v, i) = \begin{cases} 1 & \text{if variable } v_i \text{ is not assigned a value,} \\ 0 & \text{otherwise.} \end{cases}$$

To find the minimum of this function, the evolutionary algorithm will have to generate an individual that has no variables with unassigned values. This is the only case where the fitness function is zero. If the weights w_i would not be changed during the lifetime of the evolutionary algorithm, but instead would stay constant, we would end up with a conventional fitness function that only counts the number of variables with unassigned values. Such a fitness function always prefers an individual with fewer variables that have unassigned values, i.e., give a better fitness value. The fitness function in Equation 5.1 however can give a worse fitness value to a individual with less unassigned variables, this happens when the weights of the unassigned variables have, relative to other weights, a high value.

The selection mechanism

During a great number of experiments, involving numerous operators, selection mechanisms and other techniques, Eiben et al. (Eiben et al., 1998a; Eiben and van der Hauw, 1997; Eiben and van der Hauw, 1996; van der Hauw, 1996) found that he best strategy involved a populations size of one. Our experiments will be done using the same preservative selection strategy, denoted as $(1 + 1)$.

The mutation

Just as in the previous section, the mutation will be the same as in the experiments of Eiben et al. This mutation is called *swap*. It is an order-based operator that swaps pairs of genes. The parameter p_m determines the probability that an allele is used in a swapping session. If it is used, the second allele for swapping is selected randomly from the individual. This parameter will be set to $p_m = 1/n$ as results by Eiben et al. show that this value (or higher ones) give optimal results.

An example of a swap operation is in Figure 5.5. First the allele at position 6 is chosen for the session, then the allele at position 2 is chosen. They are swapped resulting in a new individual.

Before		After
2 **3** 7 4 1 **5** 6	$\rightarrow$	2 **5** 7 4 1 **3** 6

Figure 5.5: Example of the swap operation.

5.3.3 Parameters of the algorithm

The SAW-ing algorithm has a number of parameters. These parameters can be modified and may effect the quality of the results. When an algorithm has a couple of parameters without knowledge of their exact influence on the performance, it is already very difficult to choose the right setting. For the parameters of the SAW-ing algorithm the values are taken from articles of Eiben et al. Some effort was taken into optimizing the parameters, but these ended into the same results as these articles (Eiben et al., 1998a; Eiben and van der Hauw, 1997; Eiben and van der Hauw, 1996; van der Hauw, 1996). The parameters that were used in the experiments can be found in Table 5.3.

Parameter	Value
Δw	1
Initial w for each variable	1
T_p	250
Selection	$(1+1)$
Population size	1
Representation	order-based
Crossover	none
Mutation type	swap
Mutationrate (p_m)	$1/n$
Stopcondition	solution found or T_{max} evaluations

Table 5.3: Fixed parameters of the Stepwise Adaptation of Weights algorithm.

Chapter 6

Experiments and results

6.1 Measuring performance

6.1.1 Success rate

One of the measures of performance is the *success rate* (SR). It is defined as the average number of times a method finds a solution, during a number of runs:

$$\mathrm{SR}(\#runs) = \frac{\sum_{i=1}^{\#runs} success(run_i)}{\#runs},$$

$$\text{where } success(run_i) = \begin{cases} 1 & \text{when a solution is found in } run_i, \\ 0 & \text{otherwise.} \end{cases}$$

Some care has to be taken when results are interpreted concerning the success rate. When the success rate of an algorithm is lower then one, this does not mean it did not find all the solutions. When the experiments involve problems that are not solvable we have no way in telling how much of the problems actually did find a solution. That is except if one of the algorithms has a success rate of one. As all algorithms are tested on the same test suit, if one of them has a success rate of one, we can say something extra on the performance of the algorithms that did not find all solutions.

6.1.2 Average evaluations to success

If the performance of several algorithms has to be compared, one of the most obvious measures seems to be the measure of *time complexity*, i.e., the time an algorithm needs to complete a task. But this measure has some disadvantages. It highly depends on the hardware and the implementation of the algorithm itself. Not only that, but other less obvious things like compiler optimizations can disturb the measure. A more robust measure is required, one which does not depend on external factors. One such measure is *computational complexity*. It measures the performance of an algorithm by counting the number of basic operations.

This of course leads to the question: "What are basic operations?". In search algorithms the basic operation is often defined as the creation and evaluation of a new candidate solution. This definition is not perfect, because it does not define how much time can be spend on the creation and evaluation of a candidate solution. Also it does not measure work done outside these operations. But the advantages are numerous, first of all its independence of external events. Furthermore, every search algorithm needs to create and evaluate candidate solutions. And best of all, when we look at scale-ups of results, i.e., how the performance varies when changing certain parameters, the comparison becomes independent of the absolute amount of time an algorithm has spend in the experiment. A scale-up test compares the change in performance between algorithms, measured as a change in the number of basic operations, which is much more fair than trying to compare algorithms on fixed parameter settings.

The measurement used in the experiments is called *average number of evaluations to solution* (AES). It is defined as the sum over all runs of the total number of evaluations needed to reach a solution divided by the number of successful runs (Section 6.1.1). Consequently, when the SR $= 0$, the AES is undefined.

$$\mathrm{AES}(\#runs) = \frac{\sum_{i=1}^{\#runs} evaluations(run_i)}{\mathrm{SR}(\#runs) \cdot \#runs},$$

$$\text{where } evaluations(run_i) = \begin{cases} 0, \text{ if no solution has been found,} \\ \text{evaluations in } run_i \text{ otherwise.} \end{cases}$$

On very few occasions the AES can lead to strange results. When the number of successful runs is relatively low, lets say 1 or 2 out of a hundred, and the algorithm has found a solution in relatively few evaluations (it got lucky), the AES will report a low number. It is therefore important to always report the SR and AES together, so that it is possible to make a clear interpretation.

6.2 Comparing three algorithms

To compare the three algorithms, we need a test suite of problem instances. This test suite will be created with one of the random generators described in Section 4.3. For our experiments we choose the generator by Dozier. Although the number of constraints and conflicts are more fixed then in the method of Prosser, the choice of constraint placements seems to be more random in Dozier's method. The difference lies in the selection of pairs of variables where a constraint will be added. Prosser's method uses a double loop that generates a random number between zero and one for every combination of two variables. If this number is below the density parameter a constraint is added. When the density parameter is quite high, i.e., near one, chances are that the constraints are all added in the first iterations of the second loop. This would result in an unfair distribution of constraints over the variables, the variables looked at first in the first loop will get more constraints assigned than the variables that have not been looked at. Dozier's method assigns all constraints by randomly selecting two variables, preventing any ordering in the variables and their chance of assignment.

The test suite consists of a total of 625 problem instances. It is divided into 25 combinations of the density and tightness parameter. Thus every combination consists of 25 problem instances. Every algorithm did 10 runs on each problem instance. When an algorithm had not found a solution after 100000 evaluations it was terminated and the run was marked as unsuccessful. The parameters for the number of variables and the domain size of all the variables are set to $n = 15$ and $m = 15$.

The results of the experiments can be found in Table 6.1. The table shows the success rate for each algorithm for every combination, together with the AES (in brackets).

The first thing that is clear from the results is that CCS is not able to compete with the other two algorithms. There is no combination of the parameters where SAW and MID perform worse then CCS. Therefore CCS is left out of further comparisons and experiments. However the conclusions will contain speculations on the worse performance of CCS.

Only the comparison between SAW and MID remains. For the less difficult problems, i.e., with low values for the connectivity and tightness parameters, SAW seems to be at the winning hand. But on the harder problems (($d = 0.1, t = 0.9$), ($d = 0.3, t = 0.7$) and ($d = 0.5, t = 0.5$)) the success rate of SAW drops down to a lower value than that of MID. However, SAW is has lower values for the AES on the problem instances. For two of the combinations (($d = 0.1, t = 0.9$) and ($d = 0.5, t = 0.5$)) even two and a half time as fast.

6.3 Scaling up of MID and SAW

Based on results from Section 6.2 the next results will not include information about CCS. The experiments described in this section are called scaling-up tests. Here the size of a problem, in this case the number of variables, is increased in a number of steps. This gives us insight into the robustness of the algorithms, which is very important, because most real life problems are very large.

Before revealing the experiments and the results first some words on the fairness of this kind of testing. We will look at how the AES of the algorithms changes when the problem size is increased. This kind of measure is more fair

density	alg.	tightness				
		0.1	0.3	0.5	0.7	0.9
0.1	CCS	1.00 (3)	1.00 (15)	1.00 (449)	1.00 (2789)	0.62 (30852)
	MID	1.00 (1)	1.00 (4)	1.00 (21)	1.00 (87)	0.96 (2923)
	SAW	1.00 (1)	1.00 (1)	1.00 (2)	1.00 (9)	0.64 (1159)
0.3	CCS	1.00 (96)	1.00 (11778)	0.18 (43217)	0.00 (-)	0.00 (-)
	MID	1.00 (3)	1.00 (50)	1.00 (323)	0.52 (32412)	0.00 (-)
	SAW	1.00 (1)	1.00 (2)	1.00 (36)	0.23 (21281)	0.00 (-)
0.5	CCS	1.00 (1547)	0.08 (39679)	0.00 (-)	0.00 (-)	0.00 (-)
	MID	1.00 (10)	1.00 (177)	0.90 (26792)	0.00 (-)	0.00 (-)
	SAW	1.00 (1)	1.00 (8)	0.74 (10722)	0.00 (-)	0.00 (-)
0.7	CCS	1.00 (9056)	0.00 (-)	0.00 (-)	0.00 (-)	0.00 (-)
	MID	1.00 (20)	1.00 (604)	0.00 (-)	0.00 (-)	0.00 (-)
	SAW	1.00 (1)	1.00 (73)	0.00 (-)	0.00 (-)	0.00 (-)
0.9	CCS	0.912 (28427)	0.00 (-)	0.00 (-)	0.00 (-)	0.00 (-)
	MID	1.00 (33)	1.00 (8136)	0.00 (-)	0.00 (-)	0.00 (-)
	SAW	1.00 (1)	1.00 (3848)	0.00 (-)	0.00 (-)	0.00 (-)

Table 6.1: Success rates and the corresponding AES values (within brackets) for the co-evolutionary EA (CCS), the Microgenetic algorithm with Iterative Descent (MID), and the SAW-ing EA (SAW). In this experiment the number of variables is set to $n = 15$ and the domain size of each variable is set to $m = 15$.

than the previous comparisons, because the AES only measures the number of search steps an algorithm does. It does not show how much work an algorithm performs for each search step, but when a scale-up is performed, the behavior of the performance is compared, which is by far more fair than only comparing AES values for a number of fixed parameter settings.

The experiment consists of varying n from 10 to 40 with a step size of 5, while keeping the other parameters constant ($m = 15, t = 0.3, d = 0.3$). Again 25 problem instances were generated and 10 runs were done on each instance. All runs for both algorithms were successful, therefore we will only show the AES values (Figure 6.1).

n	MID	SAW
10	10	1
15	52	2
20	163	5
25	410	30
30	1039	190
35	3462	1465
40	17252	18668

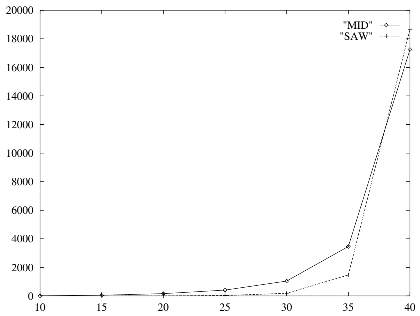

Figure 6.1: AES results for the scale-up tests of MID and SAW.

For the first part until $n = 35$, SAW scales up much better then MID. Between $n = 35$ and $n = 40$, the lines cross each other and MID gets the upper hand. Because neither of the algorithms performs better than the other on the whole range of n, and because the difference between both algorithms decreases as n grows, we are not able to conclude if one of the algorithms performs significantly better than the other.

Chapter 7

Conclusions

This research consists of two major experiments in which we compare three algorithms that incorporate adaptive mechanisms to improve their performance. The comparison is based upon results that were produced by having the algorithms solve randomly generated binary constraint satisfaction problems.

7.1 First experiment

The first experiment consists of comparing the algorithms on different combinations of two parameters: the connectivity and density of binary constraint satisfaction problems. The outcome is that the results closely match the theoretically estimated landscape of solvability. Prosser (Prosser, 1996) also concluded this, which again gives us a stronger feeling that this theoretical estimated landscape is precise enough in pointing out where to find hard instances of binary constraint satisfaction problems.

An interesting conclusion can be drawn. The co-evolutionary method, at least the one as implemented and used here, does not have a satisfactory performance compared to the other two algorithms; MID and SAW. It was often observed during runs of the algorithm that after a while the fitness values of all the members in the solution population were almost the same. By this time the fitness value of the members of the constraint population were all zero, except for one or two. Once in a while the algorithm is able to change which individual occupies the top of the constraint population, showing that the algorithm is still exploring the search landscape. But it seems that the pressure inside both of the populations is gone. The low performance of CCS made us decide to cut the algorithm from further experiments, some suggestions on improvement will be made in Section 8.

The performance of the algorithms MID and SAW differ significantly on three combinations of the density and tightness parameters, all of which are situated in the mushy region. On two of these combinations SAW is two and a half times as fast as MID, but SAW has a lower success rate on all three combinations. In one case the success rate of SAW is half that of MID's. On all combinations SAW has a lower AES value, but mostly the difference between MID and SAW are not to large.

7.2 Second experiment

The second experiment is only performed on MID and SAW, this is due to the low performance of CCS in the first experiment. The experiment consists of a scale-up test, where we increase the size of the problem and observe the behavior in performance of the two algorithms. We observed that SAW scales up much better than MID in the first part of the test where the number of variables lies between 10 and 35. But when the number of variables is 40, the difference in performance of both algorithms is not very high. Here SAW performs only slightly worse than MID.

Chapter 8

Future research

8.1 Dynamic constraints

An interesting research area is that of constraints that change during the runtime of the algorithm. A lot of problems in the real world have constraints that can change, because of a change in the environment. If an algorithm is robust enough to cope with such a change and continue its search to a new optimum, this can be seen as a very important feature for an algorithm. Evolutionary algorithms with adaptive fitness measures are promising candidates for such robust methods.

8.2 Adapting history size

As we observed during experiments with the co-evolutionary algorithm, it often happened that the fitness value of all the members from the solution population became equal to the maximum size of the history. This leads to a drop in the selective pressure, because individuals do not have to compete to stay inside the population.

One way of sustaining pressure is to increase the size of the history and the number of encounters. This way the better individuals can use the larger history to show that they are actually better then the rest. Which leaves us with the decision when to change the history size and the number of encounters. A good starting point is to do it when the fitness value of the member with the highest rank is (almost) equal to the fitness value of the member with the lowest rank. This is the point where we observe that the number of changes inside the solution population drops and that the constraints population almost exclusively consists of individuals with a fitness value of zero.

8.3 Penalizing the constraints

8.3.1 Using a different representation

Because the same representation for SAW was used in this experiment as in the those done on graph coloring by Eiben et al. (Eiben and van der Hauw, 1996; Eiben and van der Hauw, 1998; Eiben et al., 1998a), we also used the same way of penalizing variables that were not instantiated. It could be interesting to see what the performance of the SAW-ing technique would be if we would stick to the authentic idea of penalizing violated constraints. The easiest way of implementing is to use an integer-based representation where the fitness would be calculated as weighted constraint violations.

8.3.2 Another approach using permutations

The permutation-based representation for the SAW-ing techniques, as used in the experiments seems less appropriate for counting violated constraints, because there are no constraints violated once the individual is decoded. Instead we have uninstantiated variables. But when the process of decoding is doing its work we can still observe which constraints are responsible for the constraint violations. Instead of counting uninstantiated variables we can keep track of all the constraints that were the cause of an uninstantiated variable. When decoding is finished the weights of these constraints can be adjusted. The fitness function will still only return zero when a solution to the CSP is found, on the simple idea that a solution is found when there are no constraints that prevent us on instantiating variables.

Because we are dealing with binary CSPs every constraint can be the cause of one of two uninstantiated variables. This technique ones again give the SAW-ing mechanism a chance to focus on the constraints. For high values (near one) of the density of a graph this is especially important, because there will be more constraints then variables.

8.4 Improving the problem instance generator

8.4.1 Disconnected graphs

It is possible that the techniques explained in Section 4.3 for generating random CSPs create graphs that are disconnected, i.e., the graph can be divided into more than one group of vertices, such that there are no edges between vertices that reside in different groups. The chance that such a graph is generated increases when the density of a graph is low. Prosser recognizes this too in (Prosser, 1996) and uses it as a probable reason for justifying the difference between the obtained results and the theoretical estimations. Further research by Kwan et al. (Kwan et al., 1996) showed that the estimations made by Smith are not accurate for sparse graphs.

This has lead to the development of a better model for predicting the phase transitions of binary CSPs. But this model has not yet resulted into a way of generating random binary CSPs.

Because of the results mentioned previously, it may be wise to prevent the random generator for binary CSPs to generate disconnected graphs. We could try to change the generator such that it is forced to generate constraints that at least make the graph fully connected. But this could give the generated graphs some unknown and unwanted property that changes its solvability. Another approach is to discard graphs that are disconnected and to perform experiments only with connected graphs. This should be done rather carefully, because a loop like mentioned in Algorithm 4 could be infinite if the parameters are set such that it is not possible to generate connected graphs ($n > constraints + 1$).

Algorithm 4 Generating connected binary CSPs.

```
while (number_of_binary_csps > 0)
{
  binary_csp = generate_binary_csp();
  if (connected(binary_csp))
  {
    add_to_list(binary_csp);
    number_of_binary_csps--;
  }
}
```

8.4.2 Random domain sizes

It would be interesting to know what happens with the difficulty of solving binary CSPs, when the domain sizes of the variables would not be equal throughout the problem. These domain sizes could, for instance, be drawn from a random uniform distribution. This sounds more natural, because real life problems are not restricted to having one constant size for every variable's domain.

8.5 Combining different techniques

Sometimes the combination of two techniques can lead to an even better one. When such a combination can easily be created it is always worthwhile to have a look at it. If we look closely at MID and SAW, it is clear that these methods can be combined quite easily. MID keeps the penalty term caused by violated constraints in the fitness function identical throughout its lifetime, and focuses the adaptation on the penalty caused by breakouts. The

SAW-ing mechanism can thus be applied on the penalty term for violating constraints.

Combinations where CCS is involved are a little bit more complicated to produce. Its LTFE system causes much difficulty in adapting the fitness function. Another aspect is that by changing the interactions between the two populations, we could just be throwing away the careful balance that is introduced by the system, maybe causing it to converge to a non-global optimum from which it cannot escape.

Part II

Construction of a Library for Evolutionary Algorithm Programming

and

Genetic Programming on Data Mining

Library for Evolutionary Algorithm Programming

Chapter 9

Introduction

This part of the report will give an non-technical overview of a library for programming evolutionary algorithms. Because we are aiming at a framework that can be used in a broad field, we will have to incorporate techniques from all the different subareas that reside within the field of evolutionary computation. One of these subareas is *genetic programming*, a method where programs are evolved to perform certain tasks.

The incorporation of a general genetic programming frame within the whole framework will be necessary to complete the second goal: testing the extensibility of the framework by using it for the construction of a genetic programming algorithm, which will be added to the library in the future. This genetic programming algorithm will be provided with an adaptive fitness function. We have chosen the *Stepwise Adaptation of Weights* method to fulfill the role of adaptive measurement.

The genetic programming algorithm will be tested on different classification test sets. These sets are taken from standard data sets from the field of data mining. The idea here is to breed rules that should classify the records from the data set as good as possible, and then take the best rules and test how well they perform on the classification of another data set that holds the same sort of information.

The goals we want to reach in this part are highly integrated and they have a chronological ordering. First the framework needs to be build, before we can use it for the construction of a genetic program. Although the emphasis in this part lies on these two goals, we will apply the adaptive genetic programming algorithm to data mining problems, because we are interested in the performance of it, and because we want to test the implementation on real problems. The same ordering is used in this report. We will start by explaining the basic idea that was used in the construction of the framework and we will give information on the availability of the framework and its documentation. We will continue with information on the genetic programming algorithm and finish with the experiments on data mining.

Chapter 10

A library for evolutionary algorithm programming

10.1 A general overview

Many libraries for programming evolutionary algorithms exist, but most of these have two major drawbacks. Firstly, these libraries are so called toolkits, which means that they provide a collection of handy functions and objects that can be used to construct a new evolutionary algorithm. This is similar to programming toolkits that contain things such as string handling and hash tables. Generally speaking, these toolkits consist of containers and functions, that put together, provide the user with a library of building blocks for the construction of an evolutionary algorithm. Thus the user has to determine which building blocks are needed, and how to use them to build a new algorithm. To do this job, a user needs to know exactly which blocks should be connected and how they should be connected. This takes a lot of time to learn. Secondly, these libraries are aimed at a specific area within the field of evolutionary computation. Using methods from other areas means that the user will have to make the necessary changes him or herself, basically doing work that was already done in other libraries of this kind.

The problems with using toolkits occurs when the library or the program that was build with it, has to be changed. For instance, when a new selection mechanism has been added to the library, incorporation of this new mechanism into a program that was designed for an older version of the library, often requires a step by step change of the source code. This is hard work, for just trying out a new selection mechanism.

By using a *framework* we can overcome the problem of having to dig through hundreds or even thousands of lines of source code, just for trying out a new or other mechanisms. A framework does not supply the user with loosely connected building blocks, instead it provides an almost running algorithm. The user only has to put in the last pieces of the puzzle and maybe has to change the parts of the framework that are not appropriate for the problem. The framework will then provide a running evolutionary algorithm using the provided pieces, substituting the changed parts.

When additions are made to the library, programs made with it can easily make use of these additions, by only changing some lines of code in a specific and predetermined place. As long as the new method is compatible with the old one, in a high level specification sense, the library will produce a new evolutionary algorithm without the need of much work. One can imagine things going horribly wrong when trying to use a new genetic algorithm operator on a genetic program for instance. On the other hand a selection mechanism could easily be tested on different kind of algorithms, thus providing an easy way of sharing techniques between different areas of research.

To provide a running algorithm, the framework needs strong connections between the different parts that make up the library. These parts are called *objects*, taken from the *Object Oriented Paradigm*. The connections should not require that an object knows what the actual implementation behind another object is. By showing only the interface of an object, i.e., the way an object requires others to make use of it, an object can hide its implementation. The interface part of an object is called its *abstract* part. Together, these parts make up the *abstract layer* of the library.

A framework can only produce running programs when it has an actual implemented algorithms inside. The pieces of implemented code that provide the functionality of the different methods that are described in the abstract layer form another layer in the library: the *implementation layer*. Eventually this layer should grow when the library is equipped with new methods. The two layers and the parts into which they are divided are shown in Figure 10.1.

To put the abstract and implementation layer together a system is needed such that objects know only of the interfaces of other objects and such that changing objects within an implementation is made easy. This is accomplished using a technique called an *abstract factory method* taken from the software engineering paradigm called *Design Patterns* (Gamma et al., 1994). By forcing every object in the library to use factories for the creation of new objects, we can easily control which implementation is used in a program. If a user wants to make use of another method, for instance another selection mechanism, only the factory that creates this object has to be changed. The factory is part of the package for which it creates objects, it is an object as well, that has to be provided by the user of the library. A default factory is provided for each package, that creates basic objects from the library.

The construction of the library is an ongoing project where multiple people use the library for their own purpose and afterwards donate their material to the main authors of the library. These in turn incorporate it into the library. This way the library can be extended faster and it can

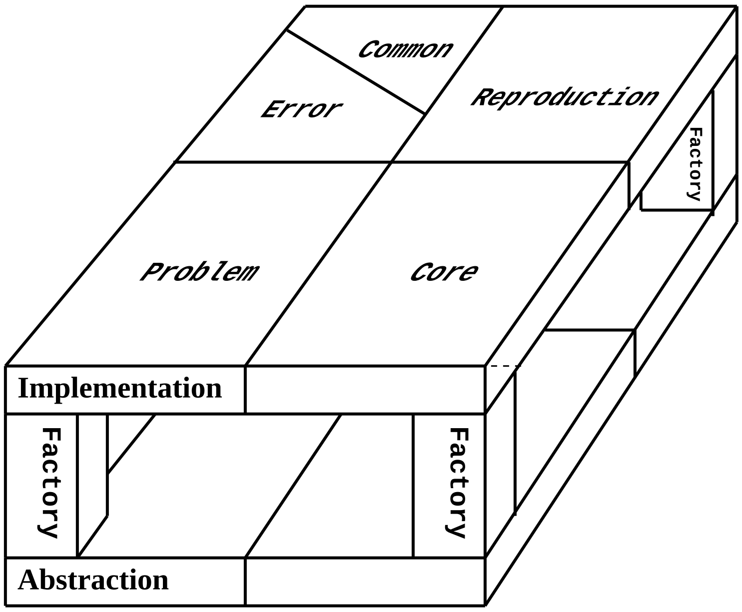

Figure 10.1: Overview of the Library for Evolutionary Algorithm Programming.

benefit from the diversity of the provided material.

As the library is distributed under the GNU Public License (GPL2), everyone can use the library for its own personal projects, and everyone is free to modify the library as long as they do not violate the GPL2. This license is provided with the distribution package for the library, and users of the library are advised to read it.

For technical information on the library we point you to the programming manual (van Hemert, 1998b), please note that as the development of the library continues, the current state of the library can differ from the information provided in this chapter.

10.2 Structure of the library

This section zooms in on the library to reveal the structure of the individual objects. Before we continue, some words on the meaning of the terminologies used in this report. When talking about libraries in general we could be talking about a toolkit, a framework, or even on a combination of the two. A library is nothing more than a collection of objects that together provide a kind of functionality. Our framework is a kind of library as well. We will use the words framework and library intertwined throughout the text.

The objects in the library, such as a population and a fitness function, should really be called classes. As the library is build using object oriented techniques, a definition (or type) is called a *class* and an instance of such a class is called an *object*. Because there will be no technical information on programming issues in this report, we will continue to name these essential items *objects*.

To provide some more structure, the library is divided into a number of packages. A *package* holds a collection of objects, where every object only belongs to this package. The contents of a package is based upon the functionality it should provide. The package consists of objects that together provide the functionality of the package. It could be the case that one object is needed in different packages, the decision where to put this object is then based on intuitive and maintenance grounds. Intuitive as in: "Where would I look first when trying to find this object?" And from a maintenance point of view; "What will happen when an object is changed within the library?"

If an object is changed, the objects should be distributed over all the packages such that most of the objects that have to be changed with it are in the same package. Note that packages work from a different angle on the layers, i.e., if an object is derived from an object of the abstract layer, both objects will be put in the same package.

10.3 A more detailed view

This section describes the objects that make up the library. Again no technical information is given, just a small overview of every object. The objects are grouped in smaller sections, where each section equals a package in the library, except for the first one. The library has no package named 'main', it is provided here for grouping the objects in the root of the library. Note that abstract and implemented objects are not discussed differently. Only the structure of the library is described here, which leaves out the details of the inner workings of the implemented objects.

An overview of all the different objects can be found in Figure 10.2. The objects are placed in a striped box, representing the package. The three objects, that are not part of a striped box, placed at the top of the picture belong to the main package. To better understand this figure we provide some notational remarks. The notation used here is taken from the *Unified Modeling Language* (UML) (Burkhardt, 1997; Fowler and Scott, 1997), with some modifications to keep the picture from getting cluttered with arrows.

Arrows with diamonds The object with the diamond attached holds the object where the arrow points to and is responsible for its creation and destruction.

Striped box around smaller box A package with its name in the top left corner of the box. This is not the usual way of representing packages in UML.

Borders of objects

Continued lines — An object that plays a role in providing the actual evolutionary algorithm.

Striped lines — A factory that produces objects in the same package.

Every package has its name in the top-left corner, and its factory in the top-right corner. Objects that have access to a factory are able to produce instances of objects from that package. For example the `Population` object has access to the `ReproductionFactory`, thus it can produce objects from the Reproduction package. The factories have to be provided by the user of the system, they determine which implemented version of an object from the abstract layer will be created. Default versions of the factories are provided by the library, they choose the basic objects that are present in the library. How these default objects look can be found in the technical documentation.

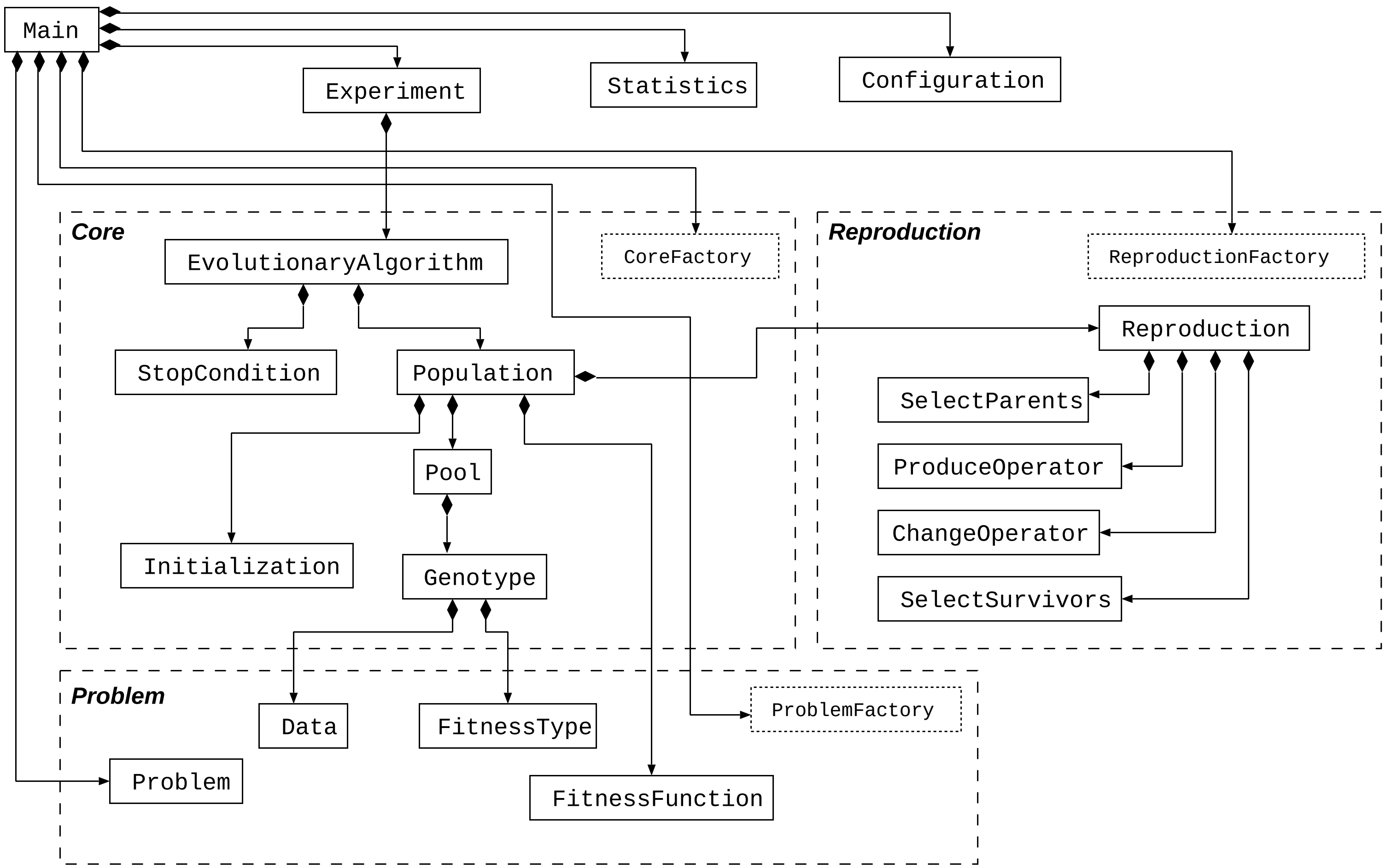

Figure 10.2: Overview of the objects in the Library for Evolutionary Algorithm Programming.

10.3.1 Main package

This package is not present as a real package in the library, it is the collection of objects that are often used in other packages or that have a great responsibility. In the library these objects can be found in the root of the directory tree. Note that the first object is not included in the library, but used here for the purpose of explanation.

Main

The library does not contain an object called main, instead this should be provided by the user. It is included here to show how some of the objects in the library have to be used in an actual implementation.

Configuration

The configuration is responsible for reading in the configuration file. Most of the other objects in the library have access to this object, and they can use it to obtain information from the configuration file.

Experiment

This object provides the actual experiments that have to be performed. It sets up the `Statistics` object and makes sure problem instances are produced before starting the evolutionary algorithm.

Statistics

A couple of objects have access to the statistics. It stores the data provides by these objects, and uses this data to provide information during and after the experiments. Data comes in the form of success rate, average evaluations to solution and average fitness.

10.3.2 Core package

Objects that reside in this package make up the inner workings of an evolutionary algorithm. Often they require little or no change when a new algorithm is being made. They either represent some sort of container for other objects or they provide a very simple algorithm.

Evolutionary Algorithm

Although this object is what gives the library its name, it does not provide much functionality. It has one population and a stop condition, and uses these to run the main evolutionary algorithm loop until the stop condition is true. The loop consists of asking the population to go to the next generation.

Population

This object is responsible for the creation and destruction of quite a lot of other objects. It has a pool where it stores all the genotypes that reside in the population. After the population is initialized using the `Initialization` object its main functionality is provided in the building of new generations. To accomplish this it has a `Reproduction` object, that given a pool of genotypes generates a new pool using predefined genetic operators.

Pool

The pool is just a storage container for genotypes. It is mentioned here because it can have a functionality of its own that can be useful for other parts of the library. For instance when the selection method used in an implementation is rank based, it could be handy to have a pool that stores its genotypes sorted on fitness.

Genotype

The genotype basically provides two functionalities. Firstly, it stores the data that is used by the genetic operators. Secondly. it stores the fitness value after it has been calculated by the fitness function.

Initialization

Before the evolutionary algorithm can begin its main loop, an initial population has to be generated. This object is used for initializing the data part of a genotype.

StopCondition

Some objects in the library have access to the stop condition. These objects can request the stop condition to terminate the evolutionary algorithm before the next loop starts. It handles two types of requests: termination because a solution was found, and termination because of an event other than finding a solution.

10.3.3 Reproduction package

One of the most important packages, because this package takes full responsibility in providing facilities for the selection mechanisms and the genetic operators. It has one main object that controls which objects are called and in which sequence. All other objects provide either a selection mechanism or a genetic operator.

Reproduction

Here the main control procedure for the reproduction takes place. This object decides which other objects in the reproduction package are used, and in which order. The default is a steady state evolutionary algorithm, where we first select a list of parents. These parents then produce offspring by recombination, which are mutated afterwards. The last operation is selecting who will survive and go to the next generation.

SelectParents

This object gets as input the current pool and returns a list of parents, which are used in one or more genetic operators.

ProduceOperator

The `ProduceOperator` gets its name from the fact that given some parents, it creates new genotypes using these parents. The new genotypes, called offspring, are returned as a list. Depending on the reproduction algorithm, this list can then be given to the `SelectSurvivors` object or to the `ChangeOperator` object. Examples of genetic operators that fall into this category are crossovers and mutation operators that copy existing genotypes.

ChangeOperator

This operator does not create new genotypes, instead it takes existing genotypes and alters their data. Examples of such an operator are normal mutation operators and heuristic repair methods.

SelectSurvivors

The final step in reproduction is determining the contents of the new pool. This is the responsibility of the `SelectSurvivors` object, which given a list of new genotypes and the old pool, outputs the new pool.

10.3.4 Common package

When constructing a new algorithm one of the most important thing is to have good building blocks. Although this library is aimed at evolutionary computation and not on data structures, a few building blocks are provided.

Array

This is a very simple array, with two advantages over other implementations. It always checks its input, and provides the user with useful information when something went wrong. Furthermore, it is possible to add a name to the array, which will be used when an error is reported.

Tree

The tree object provides the basic functionality of an n-ary tree. It hides most of its internal structure, and it has basic features for the manipulation and traversal of a tree. The tree object has already been added to the library and will be used for the implementation of the adaptive genetic program.

String

This class should be replaced with the string class from the standard C++ because it is much more efficient and easier to use. For now this object provides basic string manipulation.

Random

It is very hard to find a good random generator that works on different platforms and compilers, and at the same times guaranteeing the same results. We have chosen for the standard random generator that comes with the libraries provided by the Egcs compiler. This object merely raps the generator with a nice interface.

10.3.5 Error package

During the development of a computer program mistakes are almost inevitable. By providing a good system for detecting anomalies during the runtime of a program, errors can be exposed more quickly and more easily. But a program often needs more than debugging facilities, it needs a proper way of handling errors caused by not using the program correctly.

To report errors in a program, the library provides a small collection of different exceptions that can be used for reporting errors. Such an exception can be thrown anywhere in the library, it can than be handled by objects that were responsible for calling this code, or if this is not the case, it can be handled by the main program.

Warning

This object should only be used if something strange is found, but which does not cause immediate danger for further execution of the program. For example a parameter that has not been given in the configuration file, but which can be provided with a default value. In other words, it should report on actions taken by the program that may not be anticipated by the user of the program.

Error

An error should be the result of wrong usage of the program, such as failing to read in a file or trying to set a parameter to an unacceptable value.

InternalError

This exception helps a programmer in locating problems in a program. It reports on wrong usage of objects within the library. For instance, supplying a genetic operator with data it was not designed to manipulate should result in an internal error. Hypothetically speaking, this error should only occur during the development of a program.

LibraryError

The library error is not used much, and is not to be used in objects outside of the library. If this error occurs it means something is wrong inside the library. This could be something innocent like a function that has not been implemented yet, but it can also mean a bug has been found.

10.4 Availability

During the development of LEAP, a tool was used to produce a document containing technical information on the inner workings of the library. This tool is called DOC++, and it is freely available on the Internet. The Library for Evolutionary Algorithm Programming is free as well, it is made available to the public under the GNU Public License (GPL2). This license can be found inside the main archive, in the file '`COPYING`'. LEAP has its own page on the World Wide Web from which the latest version can be obtained. For the address of this page we point you to Appendix A.

Archives of the technical documentation can be downloaded from the LEAP page in Postscript and HTML format, in addition the HTML format has been put online. The documentation is also included in the main archive, but it has to be compiled by DOC++ before it can be used. This documentation will have to be maintained throughout the development of the library, it is therefore important to have matching versions of the documentation and library.

The library has been developed using the experimental *Egcs* compiler (version 1.0.3a), this was necessary because of the lack of good C++ support in older compilers. No attempt to test the library on other systems and compilers has been made so far, but it is known not to work on versions of the GNU compiler equal or lower than 2.7.2.1.

As development continues the library will become more dependent on the Standard Template Library (STL). Without this library LEAP will be useless. STL is freely available for almost every platform and compiler. A good implementation of STL is provided by Egcs, but for those who want to download their own version or who want to have a look at the documentation, we point you to the Internet address in Appendix A.

Chapter 11

An Adaptive Genetic Programming algorithm for Data Mining

11.1 Genetic programming and data mining

11.1.1 What is genetic programming?

Genetic programming (GP) started out as a new kind of genetic algorithm. One that does not operate on bit strings, but instead uses trees to represent individuals. Because of the special nature behind the idea of genetic programming and due to the success of the applications, genetic programming has become a field on its own. The special nature lies in the interpretation of the individuals. In genetic algorithms the individual is a static solution for a problem that was supplied to the genetic algorithm. However in genetic programming, an individual is a function or program that can be stored and used on different input than that during the execution of the genetic programming algorithm.

When genetic programming first started the structure of an individual was represented as a tree (Koza, 1992). Later other representation were adopted, such as linear and graph representations. These structures still represent a kind of program or function. Although using these representations have produced promising results, we will use the tree representation here, because it easily represents the functions we are going to breed. For the reader who is interested in the more advanced topics of genetic programming, we point you to a book written by Banzhaf et al. (Banzhaf et al., 1998) and a book edited by Kinnear (Kinnear, 1994).

Genetic programming has produced numerous successful applications, in a wide variety of fields. It has been applied in robotics, function approximation, creation of jazz melodies, construction of randomizers, cellular automata rules, data mining, and much more. When genetic programming is applied for data mining purposes, it is often used to breed rules that provide information on a data set. Different sorts of information can be gathered from data, we could be looking for rules that tell us something about the relations of the different data fields or we could be looking for rules that are able to predict an unknown data field from new data sets. We will test our GP algorithm on this last sort of information.

11.1.2 What is data mining?

The most important goal of data mining is to find new information that is potentially interesting and useful by looking at data. This search can be on a lot of different kinds of information. Examples of useful information are summary of data, classification rules, analysis of changes, detection of anomalies and clustering of data. The basic idea in all these different types of information is the discovery of information in data that gives a higher form of knowledge on the complete data set.

With so many different types of information, the field where data mining can be applied is very large. Some areas are medicine, finance and marketing. In these areas data mining has been used to analyze genetic sequences, make predictions on the stock market and discover buying patterns.

As the main focus of this research is on the development of a usable library, and not on the construction of a very efficient algorithm that outperforms presuccessors, we will not try any advanced data mining techniques. Instead the GP algorithm will be tested on a number of data sets that are commonly used to compare classification algorithms.

11.2 A genetic program for classification

11.2.1 Representation

The genetic program contains individuals that represent classification rules. These rules will have to work with records, $r = \langle r_1, r_2, \ldots, r_n \rangle$, with n the number of records, i.e. the number of data fields. The attribute r_i, with $1 \leq i \leq n$, can have any value. Most of the time a value is from one of these three domains: boolean, real or nominal. In the data sets used in the experiments later on, the attributes have been scaled to the real valued domain such that $0 \leq r_i < 1$. The rules will have to classify the records from the provided test set into two classes.

Given this information, we choose the building blocks for the classification rules. There will be two types of building blocks, functions and atoms. The set of functions will consist of Boolean functions with two arguments from this set: $\{\texttt{and}, \texttt{or}, \texttt{nor}, \texttt{nand}\}$. Note that this set is functionally complete. The atoms will read the value from one attribute, compare it to a

$$rule(r) = (A_>(r_1, 0.347) \texttt{ nor } A_<(r_0, 0.674)) \texttt{ and } A_>(r_1, 0.240)$$

Figure 11.1: An example of a classification rule.

fixed constant c, and return a boolean value. Two types of atoms will be provided, one that checks if the value from an attribute is greater than c: $A_>(r_i, c)$, and one that checks if the value is less than c: $A_<(r_i, c)$. Here c is chosen from the same domain as r_i. Figure 11.1 shows an example of a rule that can be produced using these functions and atoms. This rule returns either true or false depending on the record r. The GP is build such that the syntax as described here can easily be extended.

The atoms return a boolean value, therefore they are able to serve as input to the Boolean functions. Note that normally genetic programming uses a set of terminals, instead of atoms, these terminals represent the information as found in the data it receives as input. If the standard way of encoding would be used we would have to add two new functions, '<' and '>' and add two types of terminals, one for reading a field from a record, and one representing a constant. But as we will show next this would complicate the structure of individuals in the GP algorithm.

The rules are easily represented as trees, especially because we already made a distinction between functions and atoms. The functions become the nodes of the tree and the atoms will be represented as leaves. When the rule from Figure 11.1 is represented as a tree, it will look like Figure 11.2. We can evaluate the rule on a record by doing a postorder tree traversal which returns the class as false or true at the root of the tree.

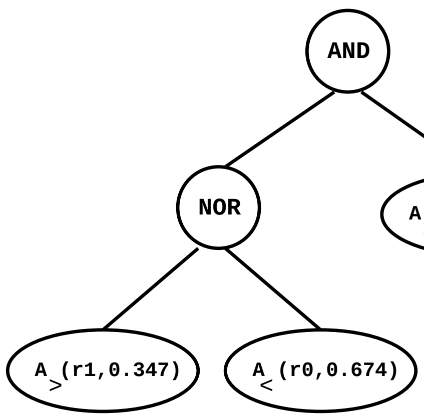

Figure 11.2: Representation of a classification rule as a tree.

If standard representation for genetic programming would be used, i.e., terminals instead of atoms, we would have to make sure that the tree is kept in a certain form. The reason for this is that not all functions understand the same type of arguments. The `and` function for instance only accepts boolean values, but the terminals are real-valued. The '<' function requires two real-valued arguments, but the `and` functions returns a boolean value. When constructing a tree, we would have to make sure these restrictions are not violated. To overcome this problem, the idea of atoms is introduced, where some of the functions are combined with the terminals to produce a set of atoms. When creating trees, the leaves of the tree will be selected from this set. In the next chapter we will show that this idea can be easily extended to other representations.

11.2.2 Initialization

Before the GP algorithm can start its main loop, a population of individuals has to be created randomly. Thus we need a method for randomly creating trees. Three methods are commonly used in genetic programming.

Full method

Starting with a function at the root of the tree, this method continues to fill the tree with randomly chosen members of the function set, up to a predefined maximum depth of the tree. When this depth is reached, the method chooses the leaves from the tree from the set of atoms. An atom is further initialized by randomly selecting an attribute r_i and randomly selecting a value for the constant c from the domain of r_i. Note that trees created this way will always be completely filled.

Grow method

This method starts out with a function for the root and then keeps selecting members from the conjunction of the functions and atoms set randomly. The initialization of an atom is the same as with the full method. To keep the method from running forever, a maximum depth is introduced. When it reaches this depth it will only select from the set of atoms.

Ramped half-and-half

This is not a new method for creating trees, but it is a combination of the other two commonly used methods. This technique generates half of the population using the full method and half of it using the grow method. This way a wide variety of trees are created.

11.2.3 Reproduction

Every generation two parents are selected using rank-based selection (Section 5.1.2), which breed two offspring using crossover or by copying themselves, this is determined using a crossover probability. Another probability, the mutation probability, determines whether the offspring is mutated. The offspring is put into the population using a worst rank replacement strategy.

Crossover

The crossover used by the GP algorithm is the basic subtree swap as defined by Koza(Koza, 1992). When two parents have been selected from the population for sexual reproduction, a random process chooses a crossover point in both parents. The choice is biased to give functions a higher probability of being selected than atoms. The offspring is created by swapping the subtrees between the two crossover points in both parents, an example is shown in Figure 11.3.

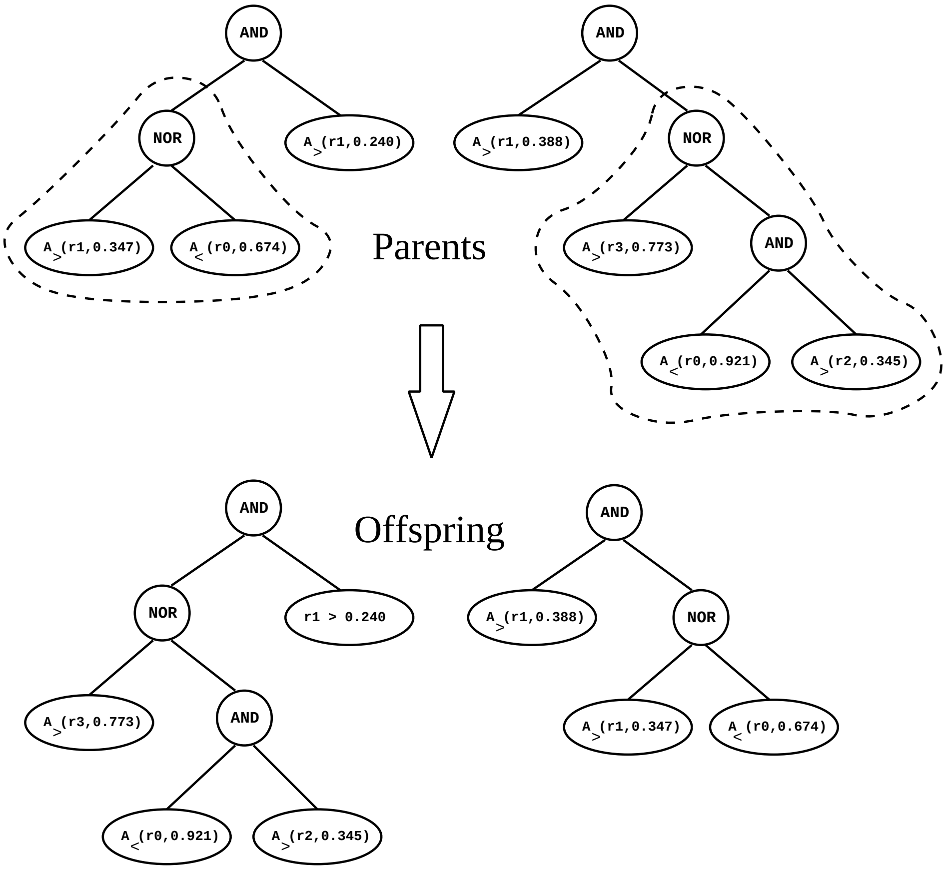

Figure 11.3: Crossover in genetic programming by swapping subtrees.

An exception is made when the size of a child is larger than the predetermined maximum number of nodes an individual is allowed to have. When this happens, the offspring will be disposed and a copy of the parent will be used as offspring.

Mutation

Because the GP algorithm uses atoms instead of terminals, we will have to make sure that the mutation operator has effect on the whole rule that is represented by an individual. To ensure this two mutation operators are used independently. Whenever an individual has been selected for mutation, a node (function) or leaf (atom) is chosen randomly from the tree for participation in the operation. If a node (function) is selected, the subtree mutation operator is applied. If a leaf is chosen, a random choice is made to determine whether we will do a subtree mutation or a subatomic mutation. Both operators have an equal chance of being chosen.

The subtree mutation replaces the selected node and the complete subtree underneath it with a subtree that is generated using the grow initialization method described in Section 11.2.2. It also uses the same maximum depth for the complete tree. Figure 11.4 gives an example of this operator.

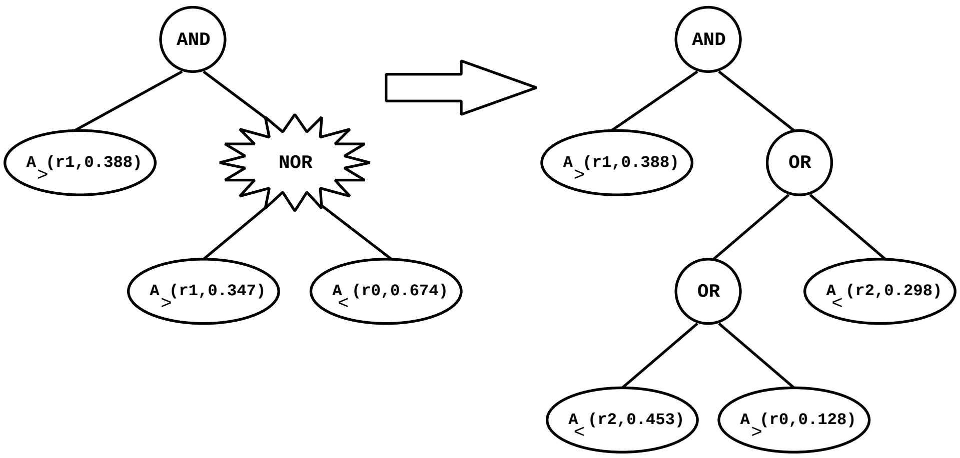

Figure 11.4: Mutation in genetic programming by replacing subtrees with randomly generated ones.

The subtree crossover only works on full building blocks, it only replaces a set of functions and atoms with a new set. This can be very destructive, especially when a node is chosen at a low depth of a tree. Here we want to introduce a less disruptive mutation, which we will call subatomic, because it operates on items in an atom.

When an atom is selected for subatomic mutation we randomly pick either the attribute r_i or the constant c, again with an equal chance. If the attribute r_i is selected, it will be replaced by randomly selecting an attribute from the record. If the constant is selected, a small random number is generated $-d < \Delta c < d$, which is then added to the constant c to form the new constant c‘. This is shown in Equation 11.1.

$$c' = \begin{cases} 0, & \text{if } c + \Delta c < 0, \\ 1, & \text{if } c + \Delta c > 1, \\ c + \Delta c, & \text{otherwise.} \end{cases} \tag{11.1}$$

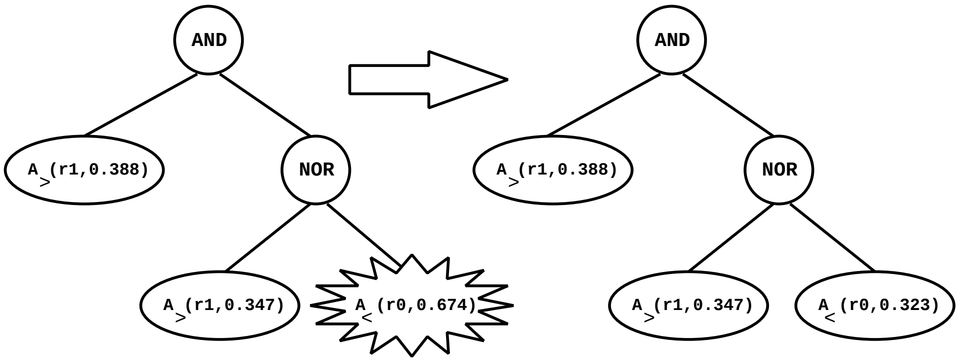

Figure 11.5: Example of a subatomic mutation.

11.2.4 Fitness function

The fitness of an individual in the population is defined as the number of wrongly classified records from the data set. This value has to be minimized by the GP algorithm, and if an individual with a fitness value of zero is found, this would mean a perfect classification rule for this data set. To evaluate an individual means that its prediction has to be verified on every record in the data set. If we define D as the set of all the records in the data set we can state the fitness of individual x as:

$$\mathit{fitness}(x) = \sum_{r \in D} |\mathit{evaluate}(x, r) - \mathit{class}(r)|$$

Because the function set consists of Boolean functions, the GP algorithm can only predict two classes. The tests will also be restricted to data sets with two classes, thus $\mathit{class}(r)$ will give a one or a zero, depending on which class record r belongs. The $\mathit{evaluate}(x, r)$ function traverses the tree of individual x and calculates the value of the rule, returning a one if the outcome is true and a zero if it is false. In the next section the fitness function is slightly changed to incorporate an adaptive technique.

11.3 Applying an adaptive fitness measure

More and more evolutionary algorithms are using an adaptive fitness measure to guide their search to an optimum solution. As one major aim of this project is to build a modern and extensible library, it would be good to include such adaptive techniques. The only problem is that some of them are quite specifically build for tackling certain problems. The Microgenetic Method from Dozier (Dozier et al., 1994; Dozier et al., 1995) (see also Section 5.2) is an example of such a technique. One of its basic components, the Breakout Management System, requires a problem where we can identify pairs of values (nogoods) that cause the algorithm to get stuck in local optima. If a problem does not have such a feature, which is the case in classification of data sets, the method can not be used.

To improve the extensibility of the library we want to include an adaptive fitness measure that can easily be used in many evolutionary algorithms working on different sorts of problems. A promising candidate for this is the SAW-ing technique by Eiben et al. (van der Hauw, 1996; Eiben and van der Hauw, 1996; Bäck et al., 1997; Bäck et al., 1998; Eiben et al., 1998b; Eiben et al., 1998a). It is also described in Section 5.3. This technique changes the fitness function using a vector of weights. These weights depend on the problem at hand, but with some creativity they can be added to almost every problem.

In a classification problem, the problem solver is presented with a number of records from a data set. It is up to the problem solver to find one or more rules that classify this list of records as good as possible. For an evolutionary algorithm this means that every time an individual needs to be evaluated, the rule it represents has to be checked on this list of records. In other words, when if we order a list of records of size n such that every record gets a unique number from $1, \ldots, n$. To introduce the SAW-ing technique into the genetic programming algorithm, a vector $w = \langle w_1, \ldots, w_n \rangle$ is defined, where every $w_i, 1 \leq i \leq n$ is paired with record i. The weights are used in the fitness function as follows:

$$\mathit{fitness}(x) = \sum_{r \in D} w_r \cdot |\mathit{evaluate}(x, r) - \mathit{class}(r)|$$

All weights in the vector w are set to one before the algorithm starts. After a number of generations, determined by the parameter T_p, the genetic programming algorithm is interrupted. The fitness of the best individual in the population is determined, and for every rule it does not classify correctly, the corresponding weight is incremented by one. The whole population then needs to be reevaluated, because of the changed fitness function. This may take a lot of time, because of the large population sizes used in genetic programming and because we are doing classification problems. As the data set does not change, we can speed up this reevaluation by storing a vector of fails and successes for each individual. This vector is updated the first time an individual is evaluated, and stores boolean values, one for each record. Instead of having to evaluate an individual all over again, we can look at the vector when we need to calculate the fitness value of an individual. When the whole population has been reevaluated the GP algorithm continues its run. Note that the reevaluation process is not the same as a fitness evaluation, because there is no need for a complete tree evaluation on all records in the training set. These reevaluation are not counted as evaluations, and therefore, will have no effect in the maximum number of evaluations.

The idea behind this way of applying SAW to data mining is that some records might be hard to classify, i.e., when trying to find a general rule from a set of records, some records differ much from the others. To help focus the search of the GP into finding a rule that is also able to classify these records, individuals get an extra reward if they classify such a record. In other words, when an individual succeeds in classifying a record that has got a high weight because it often was not classified correctly by the best individual in the population, that individual can benefit from the extra high penalty the other individuals get by not classifying it correctly.

11.4 Parameters of the algorithm

The GP contains quite a lot of parameters, as can be seen from Table 11.1. Most of the values were taken from default or recommended values from the appropriate literature.

Parameter	Value
Representation	trees
Initial maximum tree depth	5
Maximum number of nodes	200
Function set	$\{\texttt{and}, \texttt{or}, \texttt{nand}, \texttt{nor}\}$
Atom set	attribute greater or less than a constant
Population size	1000
Initialization method	ramped half-and-half
Parent selection	linear ranking
Bias for linear ranked selection	1.5
Replacement strategy	replace worse
Mutation type	1. subtree replacement 2. subatomic mutation
Subatomic d parameter	0.1
Mutation probability	0.1
Crossover	swap subtrees
Crossover probability	0.9
Crossover functions:atoms ratio	4:1
Stop condition	perfect classification or 10000 created individuals
SAW-ing update interval T_p	250
SAW-ing Δw parameter	1
SAW-ing initial weights	1

Table 11.1: Fixed parameters of the adaptive genetic algorithm programming.

Chapter 12

Experiments and results

In this chapter we will first describe the construction of a genetic program using the LEAP library. The results will consist of an overview of the development path that was followed and of the structure of the GP. The second section consists of experiments on data mining. Also, comparisons will be made with other data mining techniques.

12.1 Building a genetic programming algorithm

The genetic programming algorithm as constructed here contains three separate packages. One package for reading and using the data set, another package that holds all the components of the GP, and another one that makes up SAW. This section will describe some details of these packages and how they were constructed, beginning with the data set. An overview of these packages and of the objects that they contain can be found in Figure 12.1. To keep the figure from getting cluttered, all connections to the library are left out. Most of these are obvious, as almost all objects showed are inherited from objects from the library.

As the GP will be used for classifying data, it is necessary to construct an object that reads in the problem, i.e., the data set. The fitness function, represented by the `Classification` object, will have to traverse a list of records to evaluate each freshly created classification rule. The object that stores this list is called `DatasetProblem`, it contains a list of records. For the experiments the data sets will have to be split up into two distinct parts. One part is used for the *training phase*. This is one run of the GP, from which the best individual is taken and tested on the other part of the data set, this is called the *test phase*. The distinct parts are called respectively *training set* and *test set*.

One of the core objects needed in a tree-based GP is of course a tree. Eventually, our aim is to put all the objects constructed here into the library, we therefore make an n-ary tree that can easily be reused for other

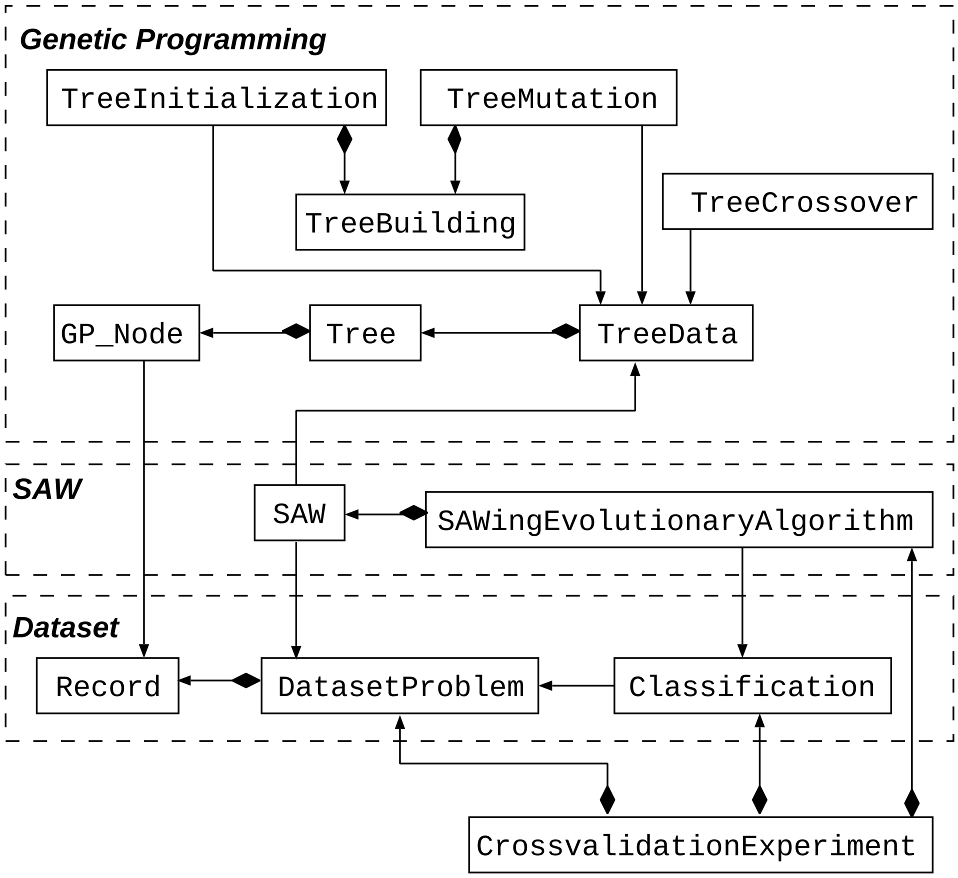

Figure 12.1: Overview of the new objects, and their relations, that make up the GP.

projects. The tree object will hide its internal structure, and it will only show basic information to other objects. Its interface is handled by *iterators* (Gamma et al., 1994) and functions working on these iterators. For example, two iterators, both pointing at a node in a tree, have to be swapped, such that a crossover as shown in Section 11.2.3 is performed. To get the job done, we just have to call the appropriate function `SwapSubtree` with the iterators as arguments. The tree object will be wrapped inside a `TreeData` object, which is handled by four other objects in the GP structure, these are: `TreeCrossover`, `TreeMutation`, `TreeInitialization` and `Classification`.

In genetic programming, the nodes of a tree have to represent something. In this GP all nodes are all of the same type, called `GP_Node`. The different types of nodes, such as functions and atoms, are all inherited from this node. This allows for an easy extension of the nodes set. This inheritance tree[1] is showed in Figure 12.2. Some work into extending this set has already begun, but in this version of the GP we will use the objects presented in Figure 12.2, as this set implements the functionality needed for the GP described in Chapter 11.

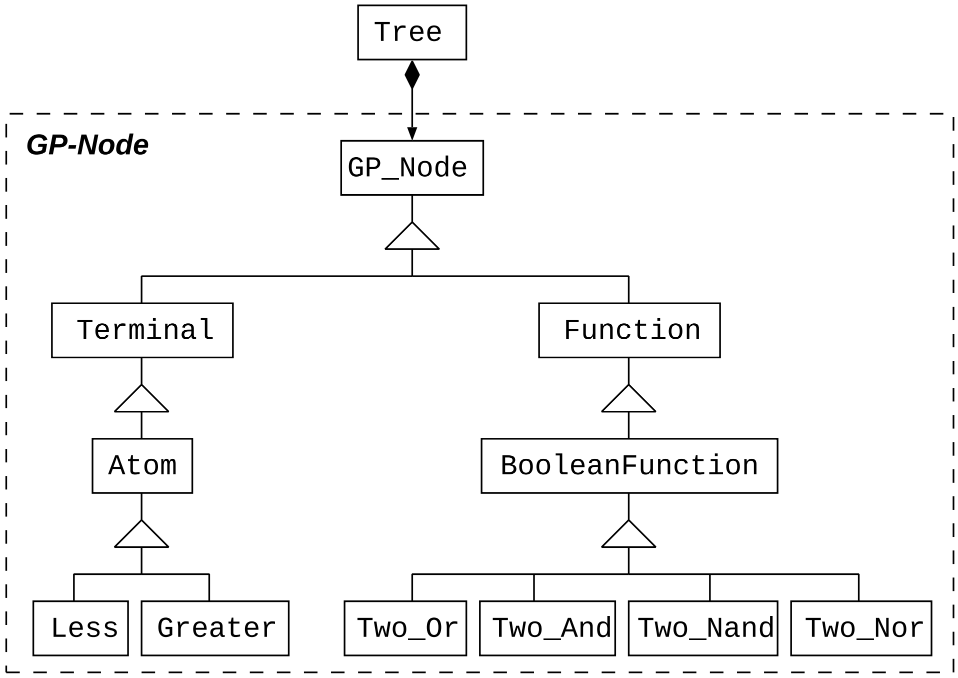

Figure 12.2: The inheritance tree for `GP_Node`.

The connection between the GP and the data set is made by the `GP_Node` and the `Classification` objects. The `GP_Node` needs the `Records` object to read the individual data fields in a record when it is being evaluated. For example when an atom is being evaluated, it needs to know the value of one of the attributes of the current records. The `Classification` object iterates through all the records and tries each record on the individual which it is evaluating. For this it needs the `DatasetProblem`. Both relations can be seen in Figure 12.1.

The `TreeInitialization` implements the ramped half-and-half tree initialization method, by calling calling the appropriate function from the `TreeBuilder` object. This object implements the first two methods as described in Section 11.2.2. The mutation operator also uses one of the functions from `TreeBuilder`. If the `TreeMutation` object has to perform a subtree mutation, it uses the grow method to generate a new subtree. The last object from the GP package, `TreeCrossover`, uses only the `TreeData` object.

[1] The word tree is taken from the terminology used in Object Oriented programming.

At this point, we almost have a running GP that tries to classify data into two classes. There is one object missing that has to control these two packages. We need an experiment manager that performs cross-validation tests using the `GeneticProgramming` and `Dataset` package. This object is called `CrossvalidationExperiment`. It runs the GP the required number of times on the provided training sets and for every run it calculates the number of wrongly classified records in the test set.

The GP is extended with the SAW-ing technique by replacing the default evolutionary algorithm object with the `SAWingEvolutionaryAlgorithm` object. It differs from the default evolutionary algorithm object, because it interrupts the main evolutionary loop every T_p number of generations, and then the SAW-ing mechanism will update its weights. This process is described in Section 11.3. The SAW-ing mechanism is implemented by the `SAW` object, which looks similar to the `Classification` object, but differs in two respects. It does not alter the fitness value stored by a genotype. Secondly, it contains a vector of weights that can be read by the `Classification` object. We will denote this algorithm as GP+SAW, and version without SAW as GP.

12.2 Data mining with a genetic program

This section describes three experiments conducted on different data sets, where the GP has to classify the records into one of two distinct classes. These data sets are taken from existing collections used for research on neural networks and machine learning. The first collection, aimed at neural networks, is called *Proben1* (Prechelt, 1994). The second collection is called *Statlog*. It has mainly been used for research on machine learning. The Internet addresses of these collections are in Appendix A.

The data set collections are accompanied with results gathered using different techniques. We will compare the performance of the GP and GP+SAW with these results. Unfortunally, both collections use different kinds of tests to acquire their results. To provide a fair comparison, we will use the same technique when we are comparing with the performance of an algorithm from one of these collections.

The results in the Proben1 collection are produced by splitting the data set into two equal parts. The first part is used as a training set and the second part is used for testing. We call this the '50% training set – 50% test set' experiments, or '50/50 test' in short. The comparisons in this section with Proben1 are all based on the percentage of wrongly classified records in the test set. Proben1 provides three permutation of each data set, therefore each table of results on a 50/50 test gives three columns of data and one column with the average over these three values.

Statlog uses n fold cross-validation tests to compare the performance of algorithms. This involves partitioning the data set into n distinct parts. The experiment consists of doing n runs of the algorithm, each run selecting one part to act as test set, while the remaining parts are used in the training phase. The results over all runs are averaged over n. Statlog also reports the percentage of wrongly classified records in the training set, we will do the same in all the n folded cross-validation tests. The different algorithms are sorted by their performance on the test set first. If this performance does not differ, further sorting is done based on the performance on the training set.

12.2.1 Test set 1: Breast cancer

The first data set comes from the Proben1 data set collection. We will only compare the results with Proben1, because Statlog does not contain this data set. This data set is originally provided by the UCI machine learning data set, and it is called the 'Wisconsin breast cancer database'. The objective is to classify the set into two types of cancer cells; benign and malignant. The class distribution is shown in Table 12.1. A record consists of nine input attributes, all having a nominal value between 1 and 10. This values are linearly scaled to a number between 0 and 1. There were 16 missing values of one attribute, they were all encoded with the average (0.35) of that attribute over all the other records.

class	quantity	percentage
Benign	458	65.5%
Malignant	241	34.5%
	699	100.0%

Table 12.1: Class distribution of the Wisconsin breast cancer data set.

50% Training set – 50% test set

The results as shown in Table 12.2 show that the difference between the best neural network and GP is only 0.002. The GP+SAW has a bit lower performance, 0.014 less than the best neural network.

12.2.2 Test set 2: Diabetes

This data set originates from the National Institute of Diabetes and Digestive and Kidney Diseases, it contains information on female Pima Indians, all at least 21 years old. Comparisons will be made on results from both Proben1 and Statlog. The set has to be classified into two classes; tested positive for diabetes and tested negative for diabetes. The input consists of eight attributes, they are all scaled to a number between 0 and 1.

	algorithm	cancer1	cancer2	cancer3	average
	Pivot NN	0.015	0.045	0.034	0.031
	No-shortcut NN	0.014	0.048	0.037	0.033
☞	GP	0.032	0.040	0.029	0.033
	Linear NN	0.029	0.050	0.052	0.044
☞	GP+SAW	0.026	0.054	0.054	0.045

Table 12.2: Percentage of wrongly classified records in a 50/50 test during the test phase on the Wisconsin breast cancer data set.

class	quantity	percentage
No diabetes	500	65%
Diabetes	268	35%
	768	100.0%

Table 12.3: Class distribution of the Pima Indians Diabetes data set.

50% Training set – 50% test set

The results of of this test clearly show (Table 12.4) that both genetic programming algorithms are beaten by the neural networks. The GP+SAW performance is better than GP, the difference is 0.01.

	algorithm	diabetes1	diabetes2	diabetes3	average
	No-shortcut NN	0.241	0.264	0.226	0.244
	Linear NN	0.258	0.247	0.229	0.245
	Pivot NN	0.246	0.259	0.231	0.245
☞	GP+SAW	0.234	0.281	0.302	0.276
☞	GP	0.271	0.294	0.294	0.286

Table 12.4: Percentage of wrongly classified records during the test phase in a 50/50 test on the diabetes data set.

12 Fold cross-validation

The performance results in Table 12.5 have been sorted by performance on the test set. For algorithms that have an equal result, a second ordering has been done on the performance on the training set. After inserting GP and GP+SAW into the list, the list was truncated after Ac2. Both genetic programming algorithms perform average, with a difference of 0.01 between them.

algorithm	training	test
LogDisc	0.219	0.223
Dipol92	0.220	0.224
Discrim	0.220	0.225
Smart	0.177	0.232
Radial	0.218	0.243
Itrule	0.223	0.245
BackProp	0.198	0.248
Cal5	0.232	0.250
Cart	0.227	0.255
Castle	0.260	0.258
QuaDisc	0.237	0.262
Bayes	0.239	0.262
☞ GP	0.255	0.263
C4.5	0.131	0.270
IndCart	0.079	0.271
BayTree	0.008	0.271
LVQ	0.101	0.272
Kohonen	0.134	0.273
☞ GP+SAW	0.246	0.273
Ac2	0.0	0.276

Table 12.5: Percentage of wrongly classified records in a 12 fold cross-validation test on the diabetes data set.

12.2.3 Test set 3: Credit cards

This data set is taken from the UCI machine learning database. Both Statlog and Proben1 contain this data set, but they have transformed it into a different encoding. We have conducted both type of tests to both encodings, thus every table of results contains two entries for each algorithm, one for Statlog and one for Proben1. The class distribution for this data set is given in Table 12.6. The data set consists of information on clients, from a Australian credit card firm. It is divided into two classes; those who are granted credit and those who are denied credit.

The data set encoding in Statlog uses 14 attributes, consisting of continuous and categorical. All attributes were scaled linearly to a value between 0 and 1. The Proben1 set uses 51 attributes, encoding any categorical attribute using a vector the size of the category and assigning a 1 to only one position in the vector.

class	quantity	percentage
Granted	307	44.5%
Denied	383	55.5%
	690	100.0%

Table 12.6: Class distribution of the Australian Credit Approval data set.

50% Training set – 50% test set

On this data set, the GP shares the first place with the Linear neural network (Table 12.7), followed closely by GP+SAW. The same algorithms perform much worse on the same data set with a different encoding, having a difference of 0.072 between both GP+SAW results and a difference of 0.113

between both GP results. Note that Statlog does not have different permutations of its data sets. Entries in the table for Statlog are therefore marked as '—'.

	algorithm	card1	card2	card3	average
	Linear NN	0.134	0.192	0.144	0.157
☞	GP[a]	0.157	0.197	0.116	0.157
	Pivot NN	0.136	0.192	0.174	0.167
☞	GP+SAW[a]	0.119	0.174	0.191	0.161
	No-shortcut NN	0.141	0.189	0.188	0.173
☞	GP+SAW[b]	—	—	—	0.243
☞	GP[b]	—	—	—	0.270

[a]Proben1
[b]Statlog

Table 12.7: Percentage of wrongly classified records during the test phase in a 50/50 test on the credit card data set.

10 Fold cross-validation

The results from this test (Table 12.8), are similar to those from the 50/50 test, in a sense that both genetic programming algorithms perform better using the Proben1 encoded data set, than using the Statlog encoded data set. The difference between the results on the different encodings is 0.096 for GP and 0.077 for GP+SAW. The performance on the Statlog encoding is worse than most other algorithms, with a difference of 0.120 between GP and the best algorithm.

	algorithm	training	test		algorithm	training	test
	Cal5	0.132	0.131	☞	GP[a]	0.149	0.155
	Itrule	0.162	0.137		Smart	0.090	0.158
	LogDisc	0.125	0.141		BayTree	0	0.171
	Discrim	0.139	0.141		KNN	0	0.181
	Dipol92	0.139	0.141		Ac2	0	0.181
	Radial	0.107	0.145		NewId	0	0.181
	Cart	0.145	0.145		LVQ	0.065	0.197
	Castle	0.144	0.148		Alloc80	0.194	0.201
	Bayes	0.136	0.151		Cn2	0.001	0.204
	IndCart	0.081	0.152		QuaDisc	0.185	0.207
☞	GP+SAW[a]	0.138	0.152	☞	GP+SAW[b]	0.231	0.229
	BackProp	0.087	0.154	☞	GP[b]	0.254	0.251
	C4.5	0.099	0.155		Default	0.440	0.440

[a]Proben1 — Runs on all three permutations of the data set are averaged.
[b]Statlog

Table 12.8: Percentage of wrongly classified records in a 10 fold cross-validation test on the credit cards data set.

Chapter 13

Conclusions

13.1 Success of the library

13.1.1 Ease of extensibility

One disadvantage of this library is its age, as it is just a few months old, the library is only equipped with a few basic objects. This led to a lot of work that had to be done in the construction of the genetic program. But as long as the library is growing, these exceptionally high amounts of work will probably decrease. However, this 'growing principle' brings additional work as well, we will discuss this in Section 13.1.2.

A very nice feature derived from the idea of having a running framework, is the fast response a developer gets during the construction of a new algorithm or technique. Because it is easy to take a default program and then to start changing this program to get the desired algorithm, the program can be executed from a very early state. This enables the developer to test the implementation from a very early start.

When an algorithm has to be implemented, a lot of work will go into designing the structure of the program. Things like storage of data, reading in of parameters, efficient usage of time and space, and hopefully re-usability and comprehensibility all take part in making this process of designing more difficult. As algorithms get larger and more complex, the construction of programs gets slower. A library, such as this one, where the developer gets a running system that already has a structure, can help in speeding up the time it takes to implement new ideas.

13.1.2 Amount of work

As the library grows, users of it will get more chances of trying out new ideas that have been implemented by others, without having to go through the developing process themselves. Thereby saving a lot of time and work. But to make this work two essential items are needed. Users of the system that have developed something new will have to donate their efforts to the library, and the maintainers of the library will have to keep it up to date.

Not only the maintainers of the library will have to do a lot of work, users will have to do quite a lot of work too. Just as with everything that is new, using a new library will require some time to get use to. But as the library has examples and as it is a running system, a user can begin by changing small parts of the library at a time. For instance a user could start by constructing a new genetic operator and use it to replace it in an example program.

The problem in maintaining a growing library is that it has quite a lot of different areas that require work. A handful of them are listed here:

Squashing bugs Whenever there is programming, there will be bugs. A good design and the usage of software engineering techniques can help prevent bugs or speed up the time in finding bugs. But errors will always be made, and one of the tasks of the maintainers is to correct these errors.

Adding new material As users donate their material it is vital that it gets incorporated into the library. If this does not happen, users could lose the motivation to turn in new material or even lose interest in using the library.

Testing new material Including new material, especially if it has been made by others, is very difficult and time consuming. The code will have to be checked to see if it works correctly, and to fully grasp its functionality.

Keeping the library consistent One of the most important issues is to keep the library consistent. This is best illustrated with an example. Lets assume that a new feature has been added to the library such that it can now handle trees represented as linear arrays. Although the library already consists of operators that work on arrays, it might not be a good idea to let them work on these particular arrays.

Keeping the documentation up-to-date One of the most frustrating moments when using a system is to find out that the contents of manual is not in line with the system. Especially for users that just started using a system it is very important that the documentation actually tells what the system does.

Maintaining a change-log When a system gets larger, it gets more difficult to identify updates and new features. Users should have a way of making sure if it is interesting to download a new version.

13.1.3 Debugging facilities

Having good debugging facilities helps users and maintainers in identifying problems and errors much earlier. It is always more informative to see an error message such as 'Tried to read beyond the end of a list' than just a plain 'Segmentation fault'. The library provides an easy way of handling errors that for most checks only takes two lines of code. It is very important for the amount of development time that these checks keep getting built in.

A good point on these checks is that when a system is running they take up precious computing time. Although this can not be denied, two good reasons block this argument. Firstly, the amount of time that was spent on fixing a system could have been spent on doing experiments if good checks were built in, such that the error in the system was identified much quicker. Secondly, it is much easier to remove checks, than it is to remove bugs.

13.1.4 Difficulty of C++

C++ is a strongly typed computer language, which means that the type of an object will have to be known at compilation time. This helps speed up the execution time of a program and it can also help in the prevention of errors. The compiler can check if types are consistently used in the source code. Because C++ is this strongly typed, it is not easy to build a library that is usable for every type, as soon as we want to introduce a variable we will have to define its type, making it impossible for a user of the library to change it.

To overcome this problem templates were introduced. These enable a developer to defer the exact type of some of the variables until a user wants to use the code. The user then has to specify the types of these variables. This works very well with containers for instance. As containers only store things, it is not necessary to know anything about the functionality of a variable. But the problem of templates is that they will have to be compiled when the user has specified the type of variable. This prevents the construction of a life system, because without knowing the type of variables we can not produce code.

Templates are very popular in the construction of libraries, such as the Standard Template Library (STL). Most of the time they are easy to use and very efficient, because they are compiled with the knowledge of the type of variables. We will try to use templates in the library where suitable, but the main goal will still be a running system.

13.2 Experiments on data mining

Looking at the results presented in Section 12.2, it is clear that we did not produce a killer algorithm that beats all the competition. But this was not our primary goal. We wanted to construct a library and test its usability on genetic programming. To observe if the GP, as constructed in the experiment, actually worked, it was used in experiments on different data sets.

Taking another look at the experiments on data mining, it is quite safe to say the GP and GP+SAW did not perform all too bad. Although neither one ever had the best results, they competed reasonably well with other algorithms in the 50/50 tests on the Proben1 data sets. On the Statlog data sets however, the performance varied widely, with a largest difference of 0.12 measured between GP and the best algorithm on the credit cards data set.

The worst difference measured between a GP and a GP+SAW on the same data set is 0.027. Showing that both algorithms have about the same performance on these data sets. Furthermore, GP+SAW had a better performance than GP in 4 of the 7 tests. An interesting point is the difference in performance on the different encodings of the credit card data set. This difference is smaller for GP+SAW than for GP in both tests. the This could mean that GP+SAW is less sensitive to a different encoding than GP.

Because of the small differences in performance of both algorithms we can not select one of the algorithms as being better than the other. However, the GP+SAW seems to be les sensitive to different encodings that GP.

Chapter 14

Future work

14.1 On the library

14.1.1 Maintenance

As already discussed in the conclusions, the library will have to be continually maintained. The first maintenance task is to clean up the library by removing unnecessary code, adding more checks and by finishing some of the interfaces.

Another important issue is the amount of STL that is used in the library. As STL is getting more popular, and because it is very easy to use it will be used more in the library. It is therefore a good idea to try to make use of the same syntax. This allows for easier use of the library, especially for those already familiar to STL.

One major point is the technical documentation, which is still not even half finished. Furthermore, the genetic programming package still has to be documented. But can better be delayed until the library has been cleaned up.

14.1.2 Genetic programming

The genetic programming algorithm has been constructed such that its parts can be easily reused. Of course it would be better if the whole package would be incorporated into the library. This would be a good test too see how difficult it is to incorporate new material into the library itself.

The genetic programming package would be a good extension to the library as it opens up a path to another field of evolutionary computation. As soon as it is in the library, work could start to extend this package, keeping it compatible with other parts of the library where possible. The syntactical restriction of functions that handle only one type of argument should be removed. Furthermore, other techniques and methods that have proven successful in boosting performance should be included.

14.1.3 New implementation objects

Every new object in the implementation layer makes the library more interesting. They provide the library with more functionality, as long as the object is usable by other objects from the implementation layer.

Sometimes it would be desirable to extend the abstract layer of the library. A good example here might be repair operators. Although they look as if they could easily be added as a kind of change operator, some of them could be doing a lot of exotic work. Maybe working on whole populations at once, or maybe even on multiple populations at once. There is no limitation to fantasy. Work has already begun on repair operators, and when they will be added to the library, the abstract layer will probably be extended.

14.2 On data mining

One interesting idea of handling a data mining problem is to use the coevolutionary model. By using two populations, one for the rules and one that holds the records from the data set, we can try a system such as the CCS as described in Section 5.1. This would save a lot of computing power when evaluating an individual, because the LTFE system only works on a subset of the data set. Instead of the normal procedure that calculates the fitness of an individual using the whole data set.

A more straight forward method of research is to change the GP by extending it with other function, atoms and terminal sets. It still lacks some basic features, such as the possibility to use it for classifying data that has more than two classes. Other improvements could be possible by incorporating more advanced genetic programming ideas, such as automatically defined functions, recursion and different representations.

Acknowledgements

A big thanks goes to A.G. Steenbeek. He implemented the Microgenetic Iterative Descent algorithm and did all the experiments with it. Without him it would not have been possible to get our article (Eiben et al., 1998b) finished in time.

We thank M. Lamers for providing some space to place the compiler that was used to develop the library. This resulted in getting him a fourth place in the list of people that use the most space on the entire system.

One of the main testers of the library, without doubt, is B.G.W. Craenen. He willingly put his own project at risk by using LEAP for the constructions of his programs. Without him I would not have spotted quite a number of errors and design flaws.

Last, but not least, the people that have taken the time to read earlier versions of this report, and who have pointed out mistakes. Here is the list: H.F. Dechering, A.E. Eiben, E. Marchiori, J.C.M. Roks, and B. Warmerdam.

Appendix A

Addresses on the Internet

CSP & EA `http://www.wi.leidenuniv.nl/~jvhemert/csp-ea`

This page contains the algorithms as described in Part 1. It contains a lot of programs that were used for doing research on different constraint satisfaction problems. It also contains the problem instance generator used in the experiments, called *RandomCsp*.

DOC++ `http://www.imaginator.com/doc++`

DOC++ is the documentation tool that has been used for generating the LaTeX and HTML based technical documentation. It is free for usage.

Egcs `http://egcs.cygnus.com`

The Egcs compiler is used for the development of LEAP. It is free for downloading and comes standard with most Linux distributions. Eventually it will supersede the GNU G++ compiler. Egcs comes equipped with a good implementation of the Standard Template Library.

LEAP `http://www.wi.leidenuniv.nl/~jvhemert/leap`

For downloading of the *Library for Evolutionary Algorithm Programming* we point you to this page. It also offers the complete documentation in Postscript and HTML format, and it has an online version of the HTML documentation.

Proben1 `ftp://ftp.ira.uka.deas/pub/neuron/proben1.tar.gz`

Proben1 consists of twelve learning problems using real data. Along with the data comes a technical report describing a set of rules and conventions for performing and reporting benchmark tests and their results.

Statlog `http://www.ncc.up.pt/liacc/ML/statlog`

The project Statlog offers data sets that were used in research on machine learning. The data sets are all accompanied with test results and a detailed description of the data and the experiments.

STL `http://www.sgi.com/Technology/STL`

This is the main site for the Standard Template Library as developed by Silicon Graphics. Beside good documentation about its contents, it offers links to many other informative pages as well.

Bibliography

Bäck, T., Eiben, A., and Vink, M. (1997). A superior evolutionary algorithm for 3-SAT. In van Marcke, K. and Daelemans, W., editors, *Proceedings of the 9th Dutch/Belgian Artificial Intelligence Conference*, pages 47–57. NAIC.

Bäck, T., Eiben, A., and Vink, M. (1998). A superior evolutionary algorithm for 3-SAT. In Porto, V. W., Saravanan, N., Waagen, D., and Eiben, A., editors, *Proceedings of the 7th Annual Conference on Evolutionary Programming*, number 1477 in LNCS, pages 125–136. Springer, Berlin.

Banzhaf, W., Nordin, P., Keller, R., and Francone, F. (1998). *Genetic Programming: An Introduction.* Morgan Kaufmann Publishers.

Booker, L. (1987). Improving search in genetic algorithms. In *Genetic Algorithms and Simulated Annealing*, pages 61–73. Korgan Kaufmann.

Burkhardt, R. (1997). *UML: Unified Modeling Language.* Addison-Wesley.

Cohn, A. G., editor (1994). *Proceedings of the European Conference on Artificial Intelligence.* Wiley.

Crawford, K. (1992). Solving the N-queens problem using genetic algorithms. In Berghel, H., Deaton, E., Hedrick, G., Roach, D., and Wainwright, R., editors, *Proceedings of the Symposium on Applied Computing. Volume 2*, pages 1039–1047, New York, NY, USA. ACM Press.

Culberson, J. (1996). On the futility of blind search. Technical Report TR 98-18, The University of Alberta. also available at `http://www.cs.ualberta.ca/~joe/Abstracts/TR96-18.html`.

Davis, L. (1991). *Order-Based Genetic Algorihms and the Graph Coloring Problem*, pages 72–90. Van Nostrand Reinhold.

Dozier, G., Bowen, J., and Bahler, D. (1994). Solving small and large constraint satisfaction problems using a heuristic-based microgenetic algorithm. In *Proceedings of the 1st IEEE Conference on Evolutionary Computation*, pages 306–311. IEEE Press.

Dozier, G., Bowen, J., and Bahler, D. (1995). Solving randomly generated constraint satisfaction problems using a micro-evolutionary hybrid that evolves a population of hill-climbers. In *Proceedings of the 2nd IEEE Conference on Evolutionary Computation*, pages 614–619. IEEE Press.

Eiben, A. and van der Hauw, J. (1996). Graph coloring with adaptive evolutionary algorithms. Technical Report TR-96-11, Leiden University. also available as `http://www.wi.leidenuniv.nl/~gusz/graphcol.ps.gz`.

Eiben, A. and van der Hauw, J. (1997). Solving 3-SAT with adaptive Genetic Algorithms. In *Proceedings of the 4th IEEE Conference on Evolutionary Computation*, pages 81–86. IEEE Press.

Eiben, A. and van der Hauw, J. (1998). Adaptive penalties for evolutionary graph-coloring. In Hao, J.-K., Lutton, E., Ronald, E., Schoenauer, M., and Snyers, D., editors, *Artificial Evolution'97*, number 1363 in LNCS, pages 95–106. Springer, Berlin.

Eiben, A., van der Hauw, J., and van Hemert, J. (1998a). Graph coloring with adaptive evolutionary algorithms. *Journal of Heuristics*, 4(1):25–46.

Eiben, A., van Hemert, J., Marchiori, E., and Steenbeek, A. (1998b). Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function. In Eiben, A., Bäck, T., Schoenauer, M., and Schwefel, H.-P., editors, *Proceedings of the 5th Conference on Parallel Problem Solving from Nature*, number 1498 in LNCS, pages 196–205. Springer, Berlin.

Eiben, A., Raué, P.-E., and Ruttkay, Z. (1995). Constrained problems. In Chambers, L., editor, *Practical Handbook of Genetic Algorithms*, pages 307–365. CRC Press.

Eiben, A. and Ruttkay, Z. (1996). Self-adaptivity for constraint satisfaction: Learning penalty functions. In *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*, pages 258–261. IEEE Press.

Fleurent, C. and Ferland, J. (1996a). Genetic and hybrid algorithms for graph coloring. In G. Laporte, I. H. O. and Hammer, P. L., editors, *Annals of Operations Research*, number 63 in Metaheuristics in Combinatorial Optimization, pages 437–461. J.C. Baltzer AG, Science Publishers.

Fleurent, C. and Ferland, J. (1996b). Object-oriented implementation of heuristic search methods for graph coloring, maximum clique, and satisfiability. In Trick, M. A. and Johnson, D. S., editors, *Second DIMACS Challenge, special issue*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. to appear.

Fowler, M. and Scott, K. (1997). *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley.

Gent, I., MacIntyre, E., Prosser, P., and Walsh, T. (1995). Scaling effects in the CSP phase transition. In *First International Conference on Principles and Practice of Constraint Programming*. also available at `http://www.cs.strath.ac.uk/~apes/apepapers.html`.

van der Hauw, J. (1996). Evaluating and improving steady state evolutionary algorithms on constraint satisfaction problems. Master's thesis, Leiden University.

van Hemert, J. (1998a). *Documentation of the RandomCsp library*. Leiden University, randomcsp version 1.1 edition. Also available at `http://www.wi.leidenuniv.nl/~jvhemert/csp-ea`.

van Hemert, J. (1998b). *Library for Evolutionary Algorithm Programming (LEAP)*. Leiden University, LEAP version 0.1.2 edition. Also available at `http://www.wi.leidenuniv.nl/~jvhemert/leap`.

Homaifar, A., Turner, J., and Ali, S. (1992). The N-queens problem and genetic algorithms. In *Proceedings of IEEE SOUTHEASTCON'92*, volume 1, pages 262–267, Birmingham, AL. IEEE, New York, NY.

Kinnear, K., editor (1994). *Advances in Genetic Programming*. The MIT Press.

Koza, J. (1992). *Genetic Programming*. MIT Press.

Kwan, A., Tsang, E., and Borrett, J. (1996). Predicting phase transitions of binary CSPs with local graph topology. In Wahlster, W., editor, *12th European Conference on Artificial Intelligence*, pages 185–189.

Morris, P. (1993). The breakout method for escaping from local minima. In *Proceedings of the 11th National Conference on Artificial Intelligence, AAAI-93*, pages 40–45. AAAI Press/The MIT Press.

Paredis, J. (1994). Co-evolutionary constraint satisfaction. In Davidor, Y., Schwefel, H.-P., and Männer, R., editors, *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, number 866 in Lecture Notes in Computer Science, pages 46–56. Springer-Verlag.

Paredis, J. (1995a). Co-evolutionary computation. *Artificial Life*, 2(4):355–375.

Paredis, J. (1995b). The symbiotic evolution of solutions and their representations. In Eshelman, L., editor, *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 359–365. Morgan Kaufmann.

Prechelt, L. (1994). Proben1 — a set of neural network benchmark problems and benchmarking rules. Technical Report 21/94, Universität Karlsruhe.

Prosser, P. (1994). Binary constraint satisfaction problems: Some are harder than others. In (Cohn, 1994), pages 95–99.

Prosser, P. (1996). An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109.

Selman, B. and Kautz, H. (1993). Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In Bajcsy, R., editor, *Proceedings of IJCAI'93*, pages 290–295. Morgan Kaufmann.

Selman, B., Levesque, H., and Mitchell, D. (1992). A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence, AAAI-92*, pages 440–446. AAAI Press/The MIT Press.

Smith, B. (1994). Phase transition and the mushy region in constraint satisfaction problems. In (Cohn, 1994), pages 100–104.

Tsang, E. (1993). *Foundations of Constraint Satisfaction*. Academic Press.

Vink, M. (1997). Solving combinatorial problems using evolutionary algorithms. Master's thesis, Leiden University.

Whitley, D. (1989). The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In Schaffer, J. D., editor, *Proceedings of the Third International Conference on Genetic Algorithms (ICGA'89)*, pages 116–123, San Mateo, California. Morgan Kaufmann Publishers, Inc.

Wolpert, D. H. and Macready, W. G. (1995). No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute.

Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

Index