MASTER

Genetic algorithms for input support minimalisation

Ederveen, N.H.

Award date:
1996
Master's Thesis:

**Genetic Algorithms for Input Support Minimalisation**

By Niek Ederveen

Coach : Dr. ir. L. Jóźwiak
Supervisor : Prof. ir. M.P.J. Stevens
Period : January 1996 - August 1996

The Faculty of Electrical Engineering of Eindhoven University of Technology does not accept any responsibility regarding the contents of Master's Theses.
Summary

Many problems, especially in the technical fields, include optimization. Whether IC’s are developed (high clock frequency, small area) or planes have to be constructed (strong, light materials, aerodynamic), one or more objectives have to be optimized. Even in forecasting the weather or the stock exchange index, the goal is to minimize the differences between the predictions and reality.

Frequently, no algorithms are available to find the best solution to such problems in an acceptable amount of time. Therefore special algorithms have to be developed to find satisfactory solutions (so called heuristic algorithms). Two drawbacks of this approach are the development time and the limited reusability.

Recently, there has been a growing interest in the optimization possibilities offered by genetic algorithms. Although they cannot guarantee to find the truthfully best solution, they have shown their very powerful characteristics in a wide variety of problems. As the basic elements of the genetic algorithm remain the same for various problems, this approach overcomes the earlier mentioned drawbacks.

These features sound very promising. The reason for the Section of Digital Information Systems at Eindhoven University of Technology to investigate some aspects of the genetic algorithms is the need to check whether they would be useful for solving some major design problems and how they should be used in such applications. Because the practical performance, instead of just theoretical background, is of high interest, the genetic algorithm had to be tried at some practical design problems.

The minimal input support problem was chosen, because the problem has a clear structure, the comparison material from some other algorithms is accessible and there is a need to have a very effective and efficient algorithm for this particular problem. The task of my graduation assignment was to analyze the features of genetic algorithms, to develop and implement an effective and efficient genetic algorithm for the input support problem and to characterize the algorithm using practical instances of the problem.

The minimal input support problem is concerned with reducing the number of inputs needed, while maintaining the performance of the system. Without loss of generality, this problem has to realise a predefined Boolean function with as few input bits as possible. In the same section of the university, extensive research has been done on this topic and a very powerful heuristic method was developed. Therefore, results of the genetic algorithm can be compared with the results from this method.

Genetic algorithms have borrowed some aspects from natural populations. A population contains several elements, each defining a possible solution. The worse the defined solutions are, the lower the corresponding elements are valued. Every generation a new population is formed. Elements of this new population (offspring) are based upon selected elements of the old population (parents). High valued elements have a higher chance to be selected for reproduction than low valued ones. There are several ways to create offspring.

The basic genetic algorithm has three operators to choose from: a parent can be copied to form offspring, a very small part of the parent can be changed to form mutated offspring, and some equal parts of two parents can be interchanged in order to get two new solutions - this way of
creating elements is called crossover. Every operator has its own probability to be chosen, even the involved parts are randomly selected by the operators.

The new population will be treated just like the old one; elements will be selected, operators will be applied on them and a new population will be created. This process continues until a predefined stop condition is met.

The genetic algorithm we have developed contains two extra operators, to incorporate problem specific knowledge. The repair operator can be applied when an element defines a solution that is actually so bad, that the proposed input bits are not sufficient to maintain the Boolean function. This operator will add just a few input bits to create an element that enables realising the functionality. The merge operator will combine the information of two elements and find a local optimal solution, that is added then to the new population.

This genetic algorithm contains many probabilities, all of them influencing the final results of the tool. Therefore, we have tried to adjust the probabilities to fit the problem well. First, the probabilities of the crossover and mutation operator have been tuned in order to get the best result using just those operators (including copying). Thereafter the merge operator has been tuned, leaving the settings of the others the same. Finally, the repair operator has been adjusted. Some additional settings have been taken into account too, like a new immigration principle in order to bring new elements into the population, the population size and changing the merge probability over time.

Using a test set of 15 Boolean functions, the probabilities for the operators to be applied did not proof themselves to be critical. As might be expected from a stochastic process, results varied within some ranges, making it very time consuming to find one setting superior to another one. With the final settings, we have always found solutions with an equal or higher quality than the heuristic methods.

For all but two of the 15 Boolean functions checked, we found the minimum solution (for ten functions this meant an improvement of one input bit over the heuristic method). For one Boolean function we never found the best result; we found always a solution with one input bit more than was needed. But even then, the solution found still contained one bit less than the solution found by the already developed heuristic method. For the other function we needed mostly one bit more, but sometimes actually we found the best solution (magnitude 20%). For this instance the solution of the heuristic algorithm contained also one bit too many.

The repair operator has not been applied in the final algorithm, unless for elements with very promising solutions, but just a few input bits too short. The merge operator on the other hand, has shown to be a very powerful operator. As a matter of fact, using this operator without the genetic algorithm, just one time on the whole problem, we found it generating better results for six of the problems where the heuristic method did not found the optimum. It never realized an inferior solution. In one case it constructed even the best solution ever found.

The original genetic algorithm has been developed for test purposes; speed was not of main concern. Now the algorithm has shown its good performance, it would be useful to make it faster. As said before, the algorithm can be easily applied on new problems and therefore the benefits could also be found in future projects. The developed heuristic algorithm is much faster; as a matter of fact it often concludes within the preprocess time needed by the genetic algorithm.

There are still some other improvements possible. The probabilities to use an operator are fixed now. It would be useful to test the changing of the probabilities over the time. In the ideal
situation, dependent on some population characteristics (diversity, average value of the solution), the probabilities should be adapted. A real stop condition has to be added too, as the current tests were just runs over 60 generations.

For the minimal input support problem, it would be useful to extend the current test set, so the performances of the possible algorithms can be researched better. The parameters of the genetic algorithm are especially set for the current test set, and the heuristic method almost always finds a near optimal solution. There is also a possibility to use the genetic algorithm quite differently in order to make the heuristic method perform as good as possible.
Acknowledgements

I would like to thank all the people who have made this project possible and contributed to its final results. A few of them have to be named for their big contributions to this research work. Hans van Pinxteren has been a major help. Often I had his ear for my problems with the computer. He always tried to find some suggestions or concrete solutions. He has been interested in my project and even when I thought I didn't have any troubles, he had some very useful tips. Gerald Hansink, Mark van Felius and Pieter Kuppens have contributed to a very good atmosphere, in which our projects could grow, even during the work overtime. Rian van Gaalen has taken care of the facility support. She got notice of things possibly influencing the progress of the work, also on the social level. She even corrected parts of this report, in order to improve the English sentences. She thinks it is her job, I think she is far too modest.

The person I owe many, many thanks is my coach, Lech Jóźwiak. He has been really interested in my project, in the progress I made, in the solutions I proposed and in everything else in the world. We have had many long discussions, starting with problems concerning my research, but quite often ending with social or cultural issues. He has been very patient with me and has stimulated me to look with a very critical eye on both whatever I proposed and whatever he proposed. I always have had the feeling we co-operated as colleagues, and I am very grateful for that.
1. Introduction

For many problems the aim is not to find just a solution, but to find the best solution. Suppose schedules have to be made for a school. It will not be very difficult to plan all lessons correctly. On the other hand, when everything has to be scheduled in order to create as few free periods as possible (for both teachers and students) with a limited number of classrooms (and probably many personal preferences added), it will be hard to find the best solution.

In the technical fields, goals often can be translated into optimization problems. When integrated circuits are developed, the clock frequency has to be as high as possible, while at the same time the chip area and the power dissipation have to be as small as possible. These objectives are just the most obvious ones (costs and reliability for example have not been taken into account). Or take for instance a train that has to be designed. The following optimization dimensions have to be taken into account (among others): the maximum speed, the possible number of passengers, the power consumption, the life expectancy and ergonomic aspects.

Real life problems are often too complex to test all possibilities in order to obtain the optimal solution. Therefore, many heuristic search algorithms have been developed to acquire a very good (if not the best) solution within an acceptable time period. Very often these heuristic algorithms are problem dependent; a new algorithm will be invented for just a relatively small class of problems.

Genetic algorithms overcome this problem in a way. The basic principle is the same for every genetic algorithm. The only two (but far from trivial) things someone should do is to code the dimensions (including the boundaries) and to quantify proposed solutions within these dimensions. This feature together with the very promising results, have made these algorithms popular both among scientists and engineers.

In the Section of Digital Information Systems at Eindhoven University of Technology I have been given an opportunity to inquire the power of genetic algorithms on a specific problem: the Minimal Input Support Problem (MISP). In brief, solutions to MISP define the minimum number of input variables required to correctly implement the behaviour of a given discrete (logical) function or sequential machine. This is a very important problem, because a substantial reduction in costs can be accomplished in many disciplines. For example, chip designers will always try to reduce the area involved and the number of pins needed. When there are less input variables, there will be less pins and less wires, so a smaller area will be needed.

The aim of my master project reported here was to develop and implement an effective and efficient genetic algorithm for MISP, to analyze and improve it and to compare it with the previously developed heuristic algorithms for MISP. To accomplish this, a set of benchmarks had been collected on which the algorithm should be tested. These hard to solve problems make a balanced process necessary, so the basic operators of the genetic algorithm should be easily to adapt. Results of my project should serve two main purposes:

• the developed prototype tool for solving MISP should be used for research purposes;
• more insight should be obtained about the usefulness of genetic algorithms for solving design problems in the field of digital system design.

Because MISP has been subject of many research projects within the fore-mentioned section, there is much information about its solutions. The solutions created and the matching time
needed by genetic algorithms can therefore easily be compared to the respectively solutions and
time of the previously developed heuristic algorithms.

In Chapter 2 the Minimal Input Support Problem will be formulated more precisely. Some
examples will be given, and parts of the previous research will be described. A state of the art
heuristic algorithm will be explained and the test set will be looked at in some detail.

Genetic algorithms (GAs) make use of a survival of the fittest principle. An initial generation is
constructed semi-randomly. This generation consists of various possible solutions (however,
mostly not very useful ones). The better solutions have the higher chances to mate and create
offspring (solutions in the next generation). Sometimes solutions do even mutate spontaneously
to propagate. Of course, there are some difficulties and some decisions to be taken. To some
extent, even explicit knowledge about a specific problem can be taken into account. The
(dis)abilities of GAs will be dealt with in Chapter 3.

In Chapter 4 time has come to merge the knowledge of MISP with that of GAs. The concepts of
the genetic algorithm for MISP will be explained. MISP has a very hierarchical solution space
topology. To incorporate problem specific knowledge, the GA makes use of this topology.
Especially for the GA solving MISP, we have designed a new operator. Analogous operators can
be designed for GAs solving other design problems. Also in Chapter 4, the outer side of the
algorithm's implementation will be showed. As the algorithm is written in the object oriented
language C++, the public part (the interface) of each object will be explained.

The interior of the implementation, the parts that make the interface actually provides the
required computations, will be described in Chapter 5. Although this is a chapter primarily meant
for maintenance purposes, here it will also become clear why some parts take very much time to
execute and therefore should be used with some reserve.

A lot of tests have been performed. Not only to check the correctness of the C++ code, but also
to adjust the probabilities of the genetic operators and afterwards to compare the genetic
algorithm solutions with those produced by the previously developed heuristic algorithms.
Because of the probabilistic character of the GA, multiple tests have to be performed in order to
draw some useful conclusions. Chapter 6 describes the process in more detail. Some conclusions
and recommendations can be found in Chapter 7.
2. The minimal input support problem

2.1 Introduction

Information processing systems are very important for our modern society. Machines have taken over many jobs and the information necessary to fulfil the task properly has to be their input. Of course, it would not be desirable to bother the system with other data - costs and time would be wasted for no good reason. To make a flight reservation, identification information and some preferences (smoker, vegetarian) will be asked, but it would be very strange when besides the number of brothers and sisters has to be mentioned. It should be obvious that information reduction is an economical issue. Databases with too many attributes, seeming to make decisions on all of them, are not wanted.

A discipline with growing interest in input minimisation is the logic design field. Recent advances in microelectronic technology have created opportunities for building very complex digital circuits at relatively low costs. These circuits include often many co-operating logic building blocks (for instance PLA's, PAL's, look-up tables and gates), each having a limited number of input pins. Therefore, a single building block enables in most cases to implement only a certain (sub-)function of the required system. Presently, much research is performed in the field of general decomposition of combinational circuits and sequential machines [Józ, 95]. Finding a minimal set of inputs for a small subcircuit is crucial for constructing effective circuit decompositions, and typically involves a lot of reduction comparing to the original input set.

As the importance of minimizing the input support is hopefully clear by now, in the remainder of this chapter the problem will be described more formally and its topology will be presented. The major heuristic algorithm will be discussed and its performances on a test set will be taken into consideration. The interested reader is referred to [Kon, 95a], which describes the problem more formally and in more detail.

2.2 The Problem

Although the Minimal Input Support Problem (MISP) is a very general problem, this thesis will only describe the problem in relation to the Boolean switching functions. Other applications are very similar to this one, and in the final project the power of the Genetic Algorithm (GA) has to be compared to other algorithms on these specific functions.

A Boolean switching function can be defined by its On- and Off-set. The members of these two sets are \( n \)-dimensional vectors, representing patterns of Boolean values. Each of the \( n \) elements of a vector can take the value zero (0), one (1), or don't care ("-"). All members of the On-set will cause the function value to become 1, while the members of the Off-set will result in a value 0 of the function. Of course, the sets have to be disjunct. MISP is defined as the problem of finding a minimal number of inputs \( m (m \leq n) \), so that the On- and Off-sets can be transformed into two new ones by just removing the same \( n-m \) input variables of every vector, while preserving the functionality. To accomplish the latter it has to be possible to distinguish every reduced element of the original On-set from every reduced element of the original Off-set.

Take for instance the following, very simple, function. The On-set consists of \{110, 101\} and the Off-set equals \{00, 011\}. It can be seen that the last two bits are essential for the functionality. So, \( n \) equals 3, \( m \) equals 2, and the new sets become: On = \{10, 01\}; Off = \{00, 11\}. 

3
More formally defined: Given two sets \( \text{On}, \text{Off} \subset \{0, 1\}^k \), \( \text{On} \cap \text{Off} = \emptyset \), find a minimal input support \( U, U \subset \{1, \ldots, n\} \) such that

\[
\forall \text{on} \in \text{On}, \text{off} \in \text{Off} \ [ \exists i \in U (\text{on}^i \neq \text{off}^i) ]
\]

MISP can be translated into a row covering problem (as both problems are \( NP \)-complete, the problem does not become harder to solve) in this way:

Suppose the \( \text{On} \)-set contains \( k \) members \( \text{on}_i \) and the \( \text{Off} \)-set contains \( l \) members \( \text{off}_i \). This defines a matrix with \( k \times l \) rows and \( n \) columns. Every row \( r \) corresponds with the differences between a unique pair \(( \text{on}(r), \text{off}(r) )\). These differences are represented with \( n \) values, each representing whether \( \text{on}(r) \) and \( \text{off}(r) \) can be distinguished by the corresponding input bit. Define Matrix \( M \) with \( k \times l \) rows and \( n \) columns:

\[
M(r, c) = 1 \quad \text{iff} \quad ((\text{on}(r)_c = 1 \land \text{off}(r)_c = 0) \lor (\text{on}(r)_c = 0 \land \text{off}(r)_c = 1))
\]

\[
M(r, c) = 0 \quad \text{iff} \quad (\text{on}(r)_c = 0 \lor \text{off}(r)_c = 0)
\]

Take the \( \text{On} \)- and \( \text{Off} \)-set of the previous example. The four rows defined by those sets are:

\[
\begin{align*}
(\text{on}_1, \text{off}_1) & \quad [0 \ 1 \ 0] \\
(\text{on}_2, \text{off}_1) & \quad [1 \ 0 \ 1] \\
(\text{on}_3, \text{off}_2) & \quad [0 \ 0 \ 1] \\
(\text{on}_4, \text{off}_3) & \quad [1 \ 1 \ 0]
\end{align*}
\]

To solve the row covering problem a minimum number of columns has to be selected, so that every row contains at least one \( 1 \) at a position of a selected column. These selected columns correspond with the minimum input support.

In order to fulfill the requirements for the first row in the previous example, we have to select the second column. Based on the same principle, the third column has to be selected for the third row. Fortunately, row two and four contain both a \( 1 \) at a selected column. So it is clear those two columns are essential to be selected, and it is sufficient to select only them. Hence, this is the solution to the row covering problem.

As can be seen from the previous example, in some cases it can be very easy to find the solution to MISP. There are three bases to reduce the cover matrix, making the task of finding the minimum number of columns an easier one. First, there can be an essential column (as in the example given). For this to happen, there has to be at least one row containing just a single \( 1 \). The corresponding column has to be selected (added to the solution). Because every row has to have (at least) one \( 1 \) selected, all rows containing a \( 1 \) at the same position can be deleted from the matrix (they do not have to be taken into account any more, as the requirements are fulfilled), together with the selected column.

\footnote{The don't care is not taken into account. Every don't care can be expanded to both a \( 0 \) and a \( 1 \), which can be translated back into the don't care after the new sets have been formed, so it does not really matter. For a more general formulation, the reader is once again referred to [Kon, 95a].}
Second, **dominated columns** can simplify the problem. If all those rows consisting of a 1 at a specific position \( d \) also have a 1 at another specific position \( e \), column \( d \) is dominated by column \( e \) and the last one dominates the first one. If a dominated column would be a member of the solution to the row cover problem, it could always be replaced by the dominating column, resulting in the same number of selected columns. Therefore the dominated column can be deleted from the matrix without influencing the solution.

Define a matrix \( M \) containing three rows:

\[
\begin{bmatrix}
0 & 1 & 1 \\
1 & 1 & 0 \\
1 & 1 & 1 \\
\end{bmatrix}
\]

Both columns one and three are dominated by the second column. Deleting them creates a new matrix \( M' \) with just one column, which is the essential one and forms the solution.

Third, **dominated rows** can ease the task. If, for a row \( d \) another row \( e \) can be found, so that for every position row \( e \) contains a 1, row \( d \) has a 1 too, row \( d \) is called to be dominated by row \( e \); the latter dominates the former. When a column \( c \) is selected to cover a dominating row \( e \) (the value of the matrix at position \((e,c)\) equals 1), automatically all rows dominated by row \( e \) will also be covered. So dominated rows do not have to be taken into account, and thus can be deleted.

All these three possibilities to reduce the problem can result in new possibilities to perform one of these actions. Therefore, these are very powerful tools.

Define a matrix \( M \) containing six rows:

\[
\begin{bmatrix}
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Notice \( M \) contains neither essential nor dominated columns. Row two is dominated by the first row and both row four and five are dominated by third row. Deleting them creates a new matrix \( M' \):

\[
\begin{bmatrix}
1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Now the second and third column are dominated by respectively the first and the last one. deleting them leaves us with both columns becoming essential; so there are only two inputs needed.

As long as essential or dominated columns, or dominated rows can be found, the matrix will reduce and the problem will take less time to come to an end. When those cannot found any more, the **preprocessing phase** stops and the real NP-complete problem starts.

The solution space of MISP contains all subsets of the input set, that is \( 2^n \) points. To represent an input support it is convenient to use a bit pattern. A 1 represents an input belonging to the input support, while a 0 corresponds with a not needed one. Hence, the bit pattern corresponding to the solution of the previous example equals 1001. It will be obvious the number of 1's in the input support is a measure for the quality of the reduction; the less 1's, the better the reduction is (and the higher the chance the input support is a solution to MISP).

When a bit pattern is actually an input support, each 0 in this pattern can be substituted by a 1 and that new bit pattern will again be an input support (albeit of less quality). The opposite is also true: each 1 in a bit pattern that is not an input support can be changed into a 0 without ever creating an input support. Figure 2.1 shows the relations between the possible bit patterns, next
to the level of quality for each pattern if it is an input support (the more to the bottom in the picture, the better the solution is) for several problem instances.

Figure 2.1 The search space of problem instances with $n = 1, 2, 3$ and $4$.

There is a potential of $k$ input support solutions with $k$ 0's. For large $n$, the search space has the form of Figure 2.2. It should be clear that for large $n$ the search space becomes as large that heuristics have to be used to find (with a high probability) the solution or a near-minimal one to the Minimal Input Support Problem.
A very smart and fast algorithm to find an input support of high quality, called QuickScan, has been developed in the Section of Digital Information Systems at Eindhoven University of Technology ([Kon, 95a]). Although it was originally created to measure the "depth" of a particular instance of the problem, the solutions it computed turned out to be of such a high quality, that all other previously developed heuristics became useless. The algorithm traverses one time through the search space, going from input support to input support.

QuickScan has two modes: a down-mode and an up-mode. In the down-mode the algorithm seeks the left-most (according to the lexicographic order) local optimum. During the search in the up-mode, the algorithm constantly tries to discover a new local minimum to enter, before it makes a next step up. In order to prevent from revisiting a local minimum, in the up-mode the algorithm considers only supports to the right of the previously visited support. When the heuristic starts, the starting position will be the bit pattern containing all inputs (guaranteed to be an input support), and the starting mode is the down-mode. The best local minimum is stored and makes the result of the algorithm. Figure 2.3 shows a search path of the QuickScan algorithm. It is clear the algorithm is deterministic and cycle-free. Because the deeper a local minimum is, the more possible ways lead to it, there is little chance the scan will skip a solution which is much better than the best visited so far.
Figure 2.3 QuickScan's search path (gray nodes represent unfeasible solutions, black arrows the down-scan, gray arrows the up-scan.

2.4 The test set

The test set contains 15 files, each defining an On- and Off-Set. Some of them contain a few essential and/or dominated columns, but for none of them the solution can be found by just preprocessing. The file name, the number of input bits, the number of product terms (usually they are evenly distributed over the On-set and the Off-set), the number of 1's in the minimum input support, the number of different solutions of the same quality and the number of essential and dominated columns all can be found in appendix A. The reader interested in the way the benchmarks have been produced is referred to [Bos, 94].

The time QuickScan needed to find a solution and the solution's quality can be found in Appendix B. The time includes the time needed to preprocess. It may be clear that the algorithm combines a good performance with a fast execution time.
3. Genetic Algorithms

3.1 An Overview

Genetic Algorithms (GAs) have been introduced by Holland in the mid-70s [Hol, 92]. Since then this topic has become of rising interest for numerous researchers, mainly because of the simple structure of GAs, the ease of translating the GAs to new fields of problems, and the relatively good performance they have shown through the years. Although GAs do not generally outperform specific problem-oriented heuristic algorithms, some improvements still are possible by incorporating problem specific knowledge into the genetic operators. Because GAs implement some stochastic processes, the theory behind GAs is mainly based on statistics.

GAs are used for optimization tasks. The basic algorithm searches the problem range without any knowledge about the problem characteristics. There is no guarantee the optimum will be found, but empirically the power has been shown. GAs have been used for several very different areas of interest. Among them are function optimization, image processing [Gol, 89], design of a communication network, semiconductor layout [Dav, 87], and machine learning [Bea, 93a]. This is by no means listed to be an exhaustive enumeration of the fields in which GAs are used, it is just to give an impression of the possibilities they offer.

The ideas GAs have been based upon follow from some characteristics of natural populations (hence the term 'genetic'). The strength of evolution, with survival of the fittest, is simulated. Mating, chromosomes, offspring - they are all meaningful for GAs too. A striking property is the ability to perform a well balanced combination of exploration and exploitation [Hol, 92]. Exploration is necessary to investigate new and unknown areas in the search space. Exploitation is necessary to dig the promising areas.

The basic GA uses a collection of samples, called a population. These samples are evaluated, to determine the promising areas, using a fitness function. Good samples (relative to the other ones in the same population) have a higher chance to generate offspring than bad samples. Two operators are used for generating offspring; crossover and mutation. The crossover operator needs two parents. The two children are constructed by combining the chromosomes of the parents using a sort of cut and paste technique. The mutation operation generates one child out of one parent by changing gene values. A new population is formed by the children. Dependent on a stop criterion, it is decided whether a new population has to be made or the algorithm terminates. Figure 3.1 depicts a schematic overview of these methods.

<table>
<thead>
<tr>
<th>Population</th>
<th>Selection</th>
<th>Operator</th>
<th>Offspring</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>0.4</td>
<td>0.111</td>
<td>crossover</td>
</tr>
<tr>
<td>1011</td>
<td>0.5</td>
<td>1110</td>
<td>after 2nd gene</td>
</tr>
<tr>
<td>1110</td>
<td>0.6</td>
<td>1011</td>
<td>crossover</td>
</tr>
<tr>
<td>0111</td>
<td>0.8</td>
<td>0111</td>
<td>after 1st gene</td>
</tr>
<tr>
<td>1001</td>
<td>0.2</td>
<td>1100</td>
<td>3d gene mutation</td>
</tr>
</tbody>
</table>

Figure 3.1 Schematic representation of the main body of the algorithm.
The remainder of the chapter describes the different parts of the algorithm in more detail. Moreover, some extra operators are handled, just as the possibility of adding some explicit knowledge to the algorithm. A brief outline of the theory behind the power of the GAs is also presented.

3.2 Genes and the fitness function

The two most important aspects of each GA are the coding and the evaluation of a sample. These are the critical aspects of a GA [Bea, 93a]. Several very different ways of coding are available, and the fitness function has to be an adequate cost function; a (relatively) good fitness should correspond to a (relatively) good solution and vice versa. Some important aspects and examples will be given below.

Coding is performed to translate the problem into a representation appropriate for processing by the computer. The coding should be in a one to one relation to the problem. Heavily related points should be coded as close as possible. This is to support the exploration and exploitation aspects. Because the coding generally just allows finite representations, continuous functions should be represented with enough coding elements, in order to be able to recover the original function (sampling rate).

A code defines a chromosome. It consists of genes (the different parts of the code), each at a locus (a specified position). So, when a code is chosen, the genes are defined (the number, the possible values, etc.) [Gol, 89]. A specified gene is called an allele, a specified chromosome (just alleles) is called a genotype. Often the code is just a vector of bits; each gene can take the value zero or one. In that case, with \( n \) bits (loci), there are \( 2^n \) genotypes.

A word of caution has to be made. Using just bits in coding does not imply using the natural binary code. As stated earlier, related points in the search space have to be similar. However, two successive points in the search space can be very different (big Hamming distance) using the natural binary code. Therefore, the Gray coding can be often a good alternative [Mat, 94].

Consider the (trivial) function \( f(x) = 4 - (x - 2)^2 \) and suppose the optimum has to be found within the range \( 0 \leq x < 4 \). When we are using the natural binary code, with four bits the coding would be 0000, 1.75 \( \sim \) 0111, 2.0 \( \sim \) 1000 and 3.75 \( \sim \) 1111. Using this code implies there are four genes, and all of them can take either the value 0 or 1. When the first allele is 1, this represents a value of 2. When the last allele on the other hand becomes 1, a value of \( \frac{3}{4} \) is represented. Now, near the optimum the genotypes differ maximal (see Figure 3.2). When the algorithm has found the second best solution at genotype 0111, it is not very likely the optimum will be found at point 1000. Using Gray code overcomes this problem.

Various problems require very different codings. Take for example the Travelling Sales Person problem (TSP). In this problem there is a map with several cities. Every city has to be visited
exactly once, except for the first one. The start point for the route has to be the end point too, so this city is visited twice. The goal is to find the shortest route for a given map. The code for the TSP can be a sequence of city numbers. Every city number corresponds with a special city. The corresponding numbers of successive cities in the tour are coded successively in the sequence. Thus, every gene can take the value of a city number, as long as none of the other genes equals that value [Gre, 87].

When such a genotype for the TSP looks like [1 5 4 2 3], it means city 1 is the start city, from there the tour leads to city 5, thereafter cities 4, 2 and 3 will be visited and finally one will return to city 1.

Other codings are used as well. Every problem has to be analyzed and the best suited coding must be used. Little changes (a decimal alphabet instead of a binary one) [Bea, 93a] and bigger changes (a variable-elements list instead of a sequence of numbers) [Dav, 87] have been tested and their results show they can be very fruitful.

The fitness function determines the quality of a genotype. The fitness of a genotype is just a single numerical value. The higher this value is, the higher the quality of the solution coded within this genotype is. A genotype together with its fitness is called a phenotype.

A fitness function will be best supporting the GA when it has not too many local optima and for sure no isolated global optimum. As a matter of fact, most optimization algorithms cannot handle such functions containing a huge amount of different steep surfaces. Because the selection function (see next section) depends highly on the fitness function, the exploration and exploitation power is influenced badly by a poor evaluation.

Changing a genotype results in another value for the fitness function. The gradient of the fitness dependent on the change of an allele is used to discriminate between three levels of difficulty [Bea, 93b].

- Level 0 - No interaction. A particular change in a gene always produces the same change in fitness.
- Level 1 - Mild interaction. A particular change in a gene always produces a change in fitness of the same sign, or zero.
- Level 2 - Epistasis. A particular change in a gene produces a change in fitness which varies in sign and magnitude, depending on the values of other genes.

It is obviously that level 2 contains the most difficult problems. As a matter of fact, level 0 and level 1 problems do not require GAs to solve them; simple algorithms like hill climbing suffice. So, before using GAs, it has to be checked the problem characteristics correspond to epistasis.

For some problems the fitness function is evident. Of course, when there is just a single objective it is easier to measure quality than when several objectives have to be optimized at the same time. Priority and relative importance have to be formulated to insure the quality of the solutions found instead of turning the balance in favour of one objective. To tackle the problem of formulating a good fitness function for the multi-objective problem, a thoroughly understanding of the problem and solution space is necessary - whether GAs will be used or not. Sometimes a fitness function suffers from noise. If a good set of rules for playing a game has to be evolved and the quality is tested by using them to play against an opponent, then only an approximation of the fitness function can be determined.
The examples already given have clear objectives; the maximal value within a small range of a function and the shortest tour for some cities have to be found. In both cases just a single objective has to be satisfied. The function to evaluate the quality can be straightforward. The fitness function for optimizing \( f \) can be \( f(x, \text{an}, k) \), for the quality is proportional to this value. For the TSP a very bad tour can be constructed with length \( l_{\text{worst}} \). Now the fitness function can be defined as \( f_{\text{best}} = l_{\text{worst}} \), with \( l_{\text{worst}} \) the length for the tour specified by the genotype.

### 3.3 Population and selection

The GA is generating a new population over and over, until the stop criterion has been satisfied. A population is just a collection of phenotypes. Often the population contains a fixed number of phenotypes. Generally the first generation is created at random. Every allele has the same chance to become part of a genotype. This population forms a good starting point for the search. The GA gets an impression of the promising areas.

It is clear that the bigger the size of the population is, the more detailed the impression will be, but on the other hand the more computational effort and time will be spent by the algorithm. Therefore it is interesting what the ideal size of a population should be. Unfortunately, not much is said about this topic. Sizes of 50 or 100 elements are very common, but seem especially inspired by the beauty of the number. [fbi, 94] proposes a number of elements equal to two times the chromosome’s number of loci. Just like many parameters used in GAs, empirical results rules.

The quality of a population can be measured in several ways. The best phenotype within the population is an important indication, for this is probably the solution to the problem when the program finishes. Another indication is the on-line performance. This performance is an average (sometimes nonuniformly weighted) of all function evaluations up to the most recent one. The off-line performance is the (weighted) average of the best phenotype’s fitness at a particular time, over the whole time range available. In formulac:

\[
\begin{align*}
\text{On-line performance} &= \frac{1}{T} \sum_{t=1}^{T} a_{\text{sample}} \cdot \text{fitness}(x_{\text{sample}}) \\
\text{Off-line performance} &= \frac{1}{T} \sum_{t=1}^{T} b_{\text{sample}} \cdot \text{fitness}(\text{best}_{\text{sample}})
\end{align*}
\]

\[
\text{With } \frac{1}{T} \sum_{t=1}^{T} a_{\text{sample}} = \frac{1}{T} \sum_{t=1}^{T} b_{\text{sample}} = 1
\]

\[
x_i = \text{genotype at time } i \\
\text{best}_i = \text{best genotype at time } i
\]

The difference between the names on-line and off-line are strongly related to their applications. An on-line application demands for a quickly offered solution. In other words, the GA has to find as fast as possible an acceptable genotype. In an off-line application it is appreciated too to have a short computing time, but the main thing is to get the best solution [Gol, 89].

In a well implemented GA the population will evolve over successive generations to increase both the best phenotype’s fitness and the average fitness within the population towards the value of the global optimum. A gene is said to have converged when 95% of the population contains
the same allele at that specified locus. The population is said to have converged when all of the
genes have converged [Bea, 93a]. For an off-line application this is often the stop criterion.

As already stated the initial population is often constructed at random; every allele has an equal,
fixed, probability to be created. It can also be constructed quasi-randomly in order to create a
maximum amount of diversity. Alleles not already covered by some members of the population
are now preferred to be selected (so the probability is adapted dynamically) [Gre, 87].

Selection should discriminate upon expectation of area's potentials. The idea behind this
operation is simple: good phenotypes have to be used for creating new phenotypes, bad
phenotypes have to be ignored. Many different implementations have been developed. Describing all of them here would fall behind the scope of this report, especially because the
imagination is the limit.

A selection operation has to fulfil three goals:

- the population size has to stay constant (maybe vary within some range, see [Hin, 96]);
- the population has to find the optimum (or at least a near optimal solution) within an
acceptable time;
- the alleles within the population has to be various as long as searching has to be performed.

Maintaining the mixture of the alleles is not necessarily just the task of the selection operator (as
the mutation operator has been especially developed to create some other allele values), but it
plays an important role in it. If a gene has been converged (almost all values at that specified
locus equal each other), the GA gets a hard job in finding a solution with another allele at that
locus, so the balance will turn in favour of exploitation over exploration. Therefore it is very
important not to converge (too) early.

A very commonly used operator for this purpose is the roulette wheel selection. Every
phenotype gets a slot in an imaginary roulette wheel. The size of the slot is linear dependent on
the fitness function. A phenotype is selected by turning the roulette wheel. This have to be done
a number of times equal to the size of the population. The process is also called stochastic
sampling with replacement.

This method is very straightforward. The expectation for every phenotype to be selected equals
the relative fitness (individual fitness divided by the summoned fitnesses of all phenotypes).
There is a drawback though. When the fitness of one genotype is giant compared with the other
fitnesses, this will probably be the only selected phenotype. This results in premature
convergence.

Another method is called ranking. This method sorts the phenotypes, best fitnesses first. Every
ranking corresponds to a number of selections; it is a deterministic process. When differences
between fitnesses are very small, this operation generates bad results.

The last, frequently used selection method described in detail here is the stochastic tournament.
Two (sometimes more) phenotypes are chosen randomly (all have the same chance to be picked
up) to be compared. The best phenotype is selected. This process continues until enough
phenotypes have been selected. A big advantage of this method is the short computing time
involved. This is a sort of stochastic ranking method. The expected number of selections is not
depending on the fitness, just on the ranking position. We have performed a statistical analysis on
this topic (Appendix C). The chance for the best phenotype to be not selected in big populations equals almost $e^{-2}$.

Many other selection methods have been developed. These include roulette wheel selection with adapting slots (when a phenotype is selected, the size of the slot is made smaller), called stochastic sampling without replacement, and remainder stochastic sampling methods. In this last method every phenotype is deterministic selected a number of times equal to the integer of the expected roulette wheel selection. The empty spaces are filled with a stochastic sampling method applied on the remainder value. These methods can be varied to other tastes, but have the disadvantage using a lot of computational power.

Several fitness scaling methods are available to overcome the problems with extreme big or small differences between the fitnesses within a population. Linear or exponential scaling, with or without shifting, the GA designer can add it all to the design. The interested reader is referred to [Bea, 93a] and [Gol, 89] for this topic.

Another selection criterion includes maintaining different alleles in successive populations to prevent the GA from genetic drift (a form of premature convergence). This is done by adjusting the fitness of very similar genotypes (e.g. dependent on the uniqueness of alleles or the averaged Hamming distance to other genotypes) (see for example [Kan, 95]).

One thing should always be kept in mind. In developing very advanced operators and methods the computational time needed to carry them out should be related to the profits they bring. Often it is more beneficial to have an extra population evaluated than to make sure a population is created on the best way. The power of a GA is the easy but efficient way of exploration and exploitation of the function through successive populations. Therefore it should be weighed whether very special operators are worth their time to spend.

3.4 Crossover and mutation

The two main operators in a GA are crossover and mutation. The purpose of both of them is to create offspring. Crossover is a sexual operator; it needs two parents. Mutation on the other hand needs just one phenotype and is therefore called an asexual operator.

The basic form of the crossover process uses cut and paste. The oldest form is called 1-point crossover. Both the parent's genotypes are cut at the same, randomly chosen, locus. The first part of parent one is connected with the second part of parent two. This newly formed genotype is child one. Child two is constructed similar. Now the first part of parent two is connected with the second part of parent one.

Suppose genotype 1 has the form [0 0 1 1 0 0] and genotype 2 looks like [1 0 1 0 1 0] (see Figure 3.3). These two genotypes are selected to create offspring using crossover. The cutting point is chosen between loci 3 and 4. Now the genotype of child 1 becomes [0 0 110 1 0] ("I" marks the cutting point), while the genotype of the other child now has the form [1 0 1 0 0].

14
Of course, this simple cut and paste technique is not always possible. The offspring might violate the gene specifications. Take for instance the TSP. When two genotypes specify two different tours, the crossover operation in this form might create tours including the same city twice. There are many solutions proposed [Gol, 89]. A simple crossover operation for this type of problems chooses a cross point, and rearranged the alleles behind that point in the same relative sequence as could be found within the other parent. This method always generates correct genotypes.

Take for example tour 1 [1 2 3 4 5] and tour 2 [5 2 4 1 3]. Suppose the cross point is between locus 2 and 3. Now child 1 becomes [1 2 1 5 4 3]. The first two alleles of parent 1 have been unchanged. The last three have been ordered like the relative ordering within parent 2. Similar, child 2 becomes [5 2 1 1 3 4].

The basic crossover operator described is called the 1-point crossover, because it uses just one cross point. Result of this technique is that the first and the last allele of a selected genotype never propagate together. As these two loci are nothing more special than the other ones, the crossover operator has been refined; the 2-point crossover treats the chromosome as a ring structure. Two points are randomly chosen. All alleles within this range specified by these points are changed over the genotypes.

Again, genotype 1 ([0 0 1 1 0 0]) and genotype 2 ([1 0 1 0 1 0]) are selected to create offspring (Figure 3.4). The 2-point crossover cutting points are chosen between loci 1 and 2 and between loci 4 and 5. Now the genotype of child 1 becomes [0 1 0 1 0 0] and the genotype of child 2 becomes [1 1 0 1 1 0].

Inspired by the success of the 2-point crossover, n-point crossover has been developed (n a fixed integer). These operators swap after each chosen point the parent they use to construct the offspring. As a matter of fact, these operators are identical to n times using the 1-point crossover.
The ultimate n-point crossover technique is the uniform crossover. At every locus, a coin is tossed to decide which parent's allele is inserted in child 1 (the other parent's allele is inserted at that locus in child 2). Hence, every combination is possible. The probability of the parent chosen to deliver an allele has not to be $\frac{1}{2}$. For instance, [Hua, 94] uses a probability of 0.6. Still every mixture is possible, but on the average child 1 will resemble parent 1 for 60 percents and parent 2 for 40 percents (by just non identical alleles), while for child 2 this is just the opposite.

Theoretically, the 2-point crossover is better than uniform crossover [Bea, 93b]. However, this depends highly on the coding used (a good coding ensures heavily related genes to be close to each other). Empirically, both operators have shown to be the better one, dependent on the application, but differences were only small. A major disadvantage for 2-point crossover is the big chance to produce children which are identical to the parents for converging populations. Therefore a new 2-point crossover operator has been developed.

The new 2-point crossover operator assures that both children are different from their parents. Parents which are different from each other at one locus or not different at all are not allowed to cross. One of the two cross points have to be in a section that contains different alleles, while the other has to be outside that region. This method was found to perform slightly better than uniform crossover [Bea, 93b].

Take for example genotype 1 [0 0 0 0 0 1] and genotype 2 [0 0 0 0 1 0]. They differ at two positions (5 and 6). So one cross point has to be between loci 5 and 6. The other has to be somewhere else, and the resulting offspring will become: [0 0 0 0 0 0] and [0 0 0 0 1 1].

The other operator is called mutation. Basically, mutation changes the values of one or more alleles within the parent's genotype. This operator is meant to maintain a mixture of alleles within one population. Dependent on the coding requirements, there are different appearances. For the bit sequence, this operation only has to swap a bit (0 to 1 or vice versa), while for the TSP, mutation (for example) has to turn around a sub tour in order to create a valid genotype.

Take for example genotype [0 0 1 1 0 0] as a bit sequence. Suppose the operation has chosen locus 2 to mutate, the child then becomes [0 1 1 1 0 0]. At the other hand, let the genotype for the TSP be [1 2 3 4]. In this situation 2 loci have to be chosen. Suppose mutation selects positions 2 and 4. The mutated genotype becomes [1 4 3 2]. It is clear that this creates correct genotypes.

It should be clear that the probability to use the mutation operator ($p_m$) has to be small. When we choose $p_m$ to be $\frac{1}{2}$ for every allele, the GA would degrade to a very foolish random search routine. On the other hand, crossover has to be operated often (the probability to use the crossover operator $p_c$ typically has a value above 0.6). A comparison between a crossover-only and a mutation-only based GA has been made. As could be expected, the crossover-only regime gives much faster evolution. At the end, however, mutation becomes more productive and generally finds better solutions [Bea, 93b] (mutation can be seen as a sort of random, hill climbing search).

Therefore, it could be worthwhile to adapt operator probabilities during a run [Bea, 93b]. However, this is not usually done. Often $p_c$ and $p_m$ stay constant during a run. Typical values vary from 0.6 to 1.0 for $p_c$ [Gol, 89], [Hua, 94]. For $p_m$ these vary from very small (for instance 0.001) to $(\text{number of loci})^{-1}$ [Bac, 94] or $(\text{population size})^{-1}$ [Gol, 89]. Mostly, for both operators it is checked whether they have to be used, but [Hin, 96] proposes to produce new individuals by either crossover or mutation, but never by both.
Another adaptation of the crossover probability is proposed by [Boo, 87] to overcome the problem of genetic drift. By measuring the percentage of the current population producing offspring - also called the percent involvement - one can anticipate this rapid convergence and have a chance to prevent it. A method to improve performance is to adapt crossover rate to this change in percent involvement with an opposite percentage. However, the crossover probability should always be kept in the range between 0.2 and 1.0.

The diversity can also be guaranteed by using a uniqueness value. Now, a child has to be sort of unique; it has to differ a minimum number of alleles (Hamming distance) with all other offspring. When this is not the case, some alleles are changed to satisfy the requirement. To come to a converged population the uniqueness value is lowered as the process continues. It can be seen as an adaptive mutation operator.

This prevention from loss of diversity improves the rate at which the search converges toward a good solution [Boo, 87]. On the other hand, for big population sizes it is a time expensive mechanism. Good solution parts may be randomly broken too (see also section 7), when no knowledge is used for changing gene values.

The very different parameter settings together with overall good performance might indicate the robustness of the GA. As stated earlier, these operators are not the most critical aspects.

3.5 Penalty and repair function

The fitness function is a measure for the quality of a genotype. Some functions have to deal with hard constraints. Therefore, a few parts within the problem domain can happen to be not feasible. Two ways are commonly used to handle this problem. The fitness function can be negatively modified by a penalty function to make that specific genotype less likely to be selected. The other possibility is to use a repair function when a genotype is constructed which violates the constraints. In this way, only correct genotypes are present in a population.

The penalty function can only be applied when not too many incorrect genotypes will be constructed, elsewhere there is a big chance to come at the end with an infeasible solution (though this can often be repaired by a local search algorithm). The penalty function should also lead into the good direction; less penalty should be given to almost correct solutions than to solutions far from the feasible region [Bac, 94], [Hor, 94]. This implies making the fitness function just zero for constraint violating genotypes, would not be a smart strategy.

A changing penalty function is also possible [Orv, 94]. The penalty function at the start is kept very small, but as time goes by this function is growing to extreme values. The genotypes will drift away from the unfeasible regions as the GA proceeds.

The other method is to repair these genotypes. The most extreme form is to leave the genotypes just out of the population, but this would favour an optimum within a feasible region above an optimum at the edge. Repair possibilities depend highly on the problem at hand and the coding chosen (for the TSP the crossover and mutation operators have been constructed to generate only correct genotypes - this can be seen as a sort of repair function).
According to [Orv, 94], if the repair operation constructs only points at the edges of feasible regions, and the best points are at the centre of these regions, then repairing could be a good strategy, otherwise it is not likely to make things better. Therefore it might be a good alternative to use the repair operator with a probability \( R \). Tests have shown this hybrid strategy has an optimum at \( R=0.05 \) (marginally better than for \( R=0 \), a few percentages better than for \( R=1 \)) [Orv, 94].

3.6 Other operators

Many ideas GAs use, have been inspired by nature. But there are much more mechanisms and concepts out there, not implemented in the GA discussed so far. This does not mean there are no thoughts on this topic. In this section some of them will be described briefly. The interested reader is referred to [Bea, 93b], [Gol, 89] and the newsgroup on internet called comp.ai.genetic².

Of course, it is strange that all parents die together and make room for a new generation. The steady state GA has not this property. Now every time just one (pair of) parent(s) dies, not an entire population. Another alternative has been developed. Every selected pair generates offspring. The best two phenotypes are selected out of this four and are added to the new generation [Thi, 94]. When the optimum has been found, all successive generations will contain this phenotype (as long as is is selected).

Many organisms have two sets of genes (diploidy) instead of just a single one (haploidy). In order to know which allele should be judged by the phenotype (both sets contain alleles at the same loci - what if the position for eye colour contains an allele indicating blue for the first set and brown for the other?), there is a mechanism called dominance (brown dominates blue, so the colour becomes brown). This idea have been implemented in GAs too. However, this involves a significant overhead. It is likely organisms have this property to adapt to an environment that fluctuates. So this is an interesting method when the fitness function is going to change over time, or a lot of noise is added, but for many applications diploidy and dominance are not very useful.

In nature, many species influence each other, as long as they are in the same niche. Speciation is the process whereby a single specie differentiates into two (or more) different species occupying different niches. In GAs, the maxima in the fitness function can be seen as the niches. The GA has been constructed to search for the optimum, but due to genetic drift the population may end up in a local maximum. Even worse becomes the situation when all peaks have to be detected. Still the GA converge to one of them. There are some well known solutions to this problem. The first solution prevents from crowding. Offspring replaces highly identical parents in the population (an overlapping population is used). Diversity is maintained; the population contains some species. Another solution is based on a sharing analogy; when many individuals share the same niche, they all share the same resources. In the GA, the fitness function for every phenotype is modified according to its uniqueness. When three phenotypes are (almost) identical,

²This newsgroup was a very interesting place for me during my final project. I have found here interesting discussions, new ideas and many references.
their fitness would be three times smaller. This method results in a number of species within one niche according to the corresponding peak of the fitness.

The last topic to mention here is restriction of mating³. The view behind this process is that in nature there is no mating between different species. When phenotypes located in different niches mate together, the offspring could become very unfit (the low fitness region between two niches). This is the reason some GAs do not allow every phenotype to be selected as second one for the crossover operation.

Again, and it cannot be stressed enough, implementation of these advanced procedures should be considered carefully. The time spent on a very sophisticated method cannot be spent on the evaluation of a very straightforward created population. The time lost by the extra procedures has to be (empirically) justified.

3.7 Schemata and building blocks

The theory behind GAs mainly has been involved with schemata [Hol, 92] or similarity templates [Gol, 89]. The theory has been based on statistics, because GAs use many decisions dependent on a probability factor. The formulas will not be repeated here.

The most important aspect of the theory will be explained only for chromosomes with binary genes. This can easily be translated to other chromosomes, but the main point here is to get the idea. For quantitative material, the reader is referred to the already mentioned publications.

For the type of chromosomes containing genes defined over the alphabet \{0, 1\} schemata have been defined over the alphabet \{0, 1, #\}. The meaning of the #-symbol is it matches anything; it is the don't care of the schema theorem. A genotype contains every schema with at every position # or the same symbol as the allele at that locus. So, every genotype contains \(2^l\) schemata (/ the number of loci).

Consider the genotype \([0 0 1 1 0 0]\). Some schemata contained by this type are \([0 0 1 1 0 0]\), \([# # # # # #]\) (the identical and the universal one respectively), \([0 1 0 0 # 0]\), \([# 0 1 # # #]\) and \([0 # # # # 0]\).

A schema has two characteristics; the order and the defining length. The order is the number of non-# symbols, and the defining length is the distance between the outermost non-# symbols.

So, the order of the previous example's schemata are 6 (\([0 0 1 1 0 0]\)), 0 (\([# # # # # #]\)), 3 (\([0 1 0 0 # 0]\)), 2 (\([# 0 1 # # #]\)) and 2 (\([0 # # # # 0]\)). The corresponding defining lengths are 6, 0, 5, 2 and 6 respectively.

A population contains more schemata than phenotypes. The power of a GA lies in the fact the selection operator not just selects phenotypes but also good schemata. As a matter of fact, the number of schemata which are effectively being processed in each generation is of the order \(n^2\) (n the population size). Although [Hol, 92] called this property intrinsic parallelism, the term implicit parallelism has been more frequently used.

³The reason inversion has not been described, is that this operator implies the coding has been chosen badly. A better remedy is to use uniform crossover in that case.
Now the importance of well coding becomes clear. When the fitness function has neither many local maxima nor an isolated optimum, the schemata contained by the best phenotype has on the average a high fitness, so these schemata has a better change to propagate. The GA will converge towards the optimum.

Taking the crossover operator into account, schemata with short defining lengths will be more likely to stay intact in the next generation\(^4\). That is the basis for the building block hypothesis. Building blocks are such schemata, consisting of alleles working well together, and tend to lead to improved performance when incorporated into an individual. This can be seen as the innovative thoughts of mankind; often high-performance notions (the building blocks) are combined to speculate on new ideas (the highly fit phenotypes).

Two aspects for a successful coding scheme encouraging the use of building blocks can now be formulated:

- related genes are close together on the chromosome;
- there is a little interaction between genes (the extent the fitness is influenced by an allele depends on the other alleles, too).

These two aspects cannot always be met. It has, except for the code, also to do with the problem. As is outlined in section 2, epistasis is an essential feature of a problem to justify GAs. Therefore, the second point has to be fulfilled as good as possible. But the coding has to be done carefully.

3.8 Problem specific knowledge

As said before, GAs are very robust and problem independent. This means GAs can be quickly applied to a new research area. However, when a problem has been analyzed and problem-specific heuristics have been developed, they seldom will be outperformed by GAs. It is not said that GAs are not useful for these fields (although GAs adds little to problems with very powerful heuristics available - [Eib, 94]). As stated by [Bea, 93b]: adding problem information to the GA, makes the GA less robust. On the other hand it may improve performance significantly. There are various techniques to add problem specific knowledge, but it should always be kept in mind that the GA uses some principles not supported by deterministic algorithms.

Generally, there are three different methods [Gol, 89]:

- using hybrid schemes;
- approximating the fitness function;
- constructing problem-specific operators.

\(^4\)Actually, this is just true for the 1-point crossover. For the 2-point crossover this is only true when the defining length definition follows the ring structure; outermost has to be defined at the edges of the largest #-sequence. This might also be the reason the uniform crossover sometimes seems to work better; when on the average the best schemata are of a big defining length, the uniform crossover will be less likely to disrupt them.
Hybrid schemes:
There is no guarantee the GA will find the optimum, or even a local optimum (it does often, however). Therefore, the GA result can be used as input for a local optimizer. This form of hybrid schemes is used widely; a GA to find the promising area(s) and heuristics to postprocess. Another possibility would be to use some preprocessing techniques, but this is not normally done. Using parallel operating machines, some GAs could be running independent of each other, reporting intermediate results to a (heuristic) algorithm. On account of these reports, this algorithm can adapt for instance the populations. Of course, combinations can be implemented, too.

In this method, the GA is not really changed. The results are judged by heuristics, maybe the input is already preprocessed, but the GA is still the robust process it used to be.

Approximating the fitness function:
When an exact evaluation of genotypes is a costly process, the fitness function may return an approximation of the exact answer. The time saved this way can be used for creating new generations. As the GA is to be expected robustly under error and noise, this method will not cause the GA to malperform. There are a number of approximation techniques, usually dependent on the fitness of the parents [Gol, 89]. Of course, once in a while the exact evaluation of genotypes has to be made.

Problem-specific operators:
The third method changes the GAs internally. The application of heuristics improves the performance of the blind random mechanism and the random component can compensate the strong bias introduced by the heuristics [Eib, 94]. Of course, the GA has not to be dominated by these heuristics; else the random component is not able to compensate anything at all. Creating the initial population only by heuristics might result in premature convergence, for instance. Heuristically adapting probabilities of alleles to be produced during initialising however, if done within some ranges, may result in better performance.

Another approach is to repair heuristically every genotype, which does not satisfies all constraints. This new genotype does not replace the existing genotype, but the corresponding fitness function does. Hence the phenotype is the combination of a genotype within an unfeasible region and a fitness function of a genotype in a feasible region close to the impossible one [Hua, 94].

Domain knowledge may also be used to prevent obviously bad phenotypes, or genotypes which would violate problem constraints, from being produced in the first place. Local improvement operators can also be designed based on this specific knowledge, so exploration can be improved [Bea, 93b].

Mostly, the mutation and crossover operations are changed. The mutation operation selects some alleles to be changed. For the problem specific GA, the loci and the new values can be chosen by either a uniform random selection or a heuristic-based selection. The crossover mechanism can be changed to construct two children using the pool of alleles defined by the parents. Now, a specific allele of one parent may be used twice (and hence the corresponding allele of the other parent is not used at all), to produce better offspring (genotypes, which do not violate the objectives). This should be done with care, because the diversity within the population might easily be broken.
As a matter of fact, the crossover and mutation operations for the TSP described already, has been provided to some level with problem specific knowledge. Both of the operations have been constructed to generate feasible offspring. Much experimentation is done in this field, and an important research on the topic has been described in [Gre, 87]. Although the main objective of the GA was to locate promising areas in order to fine-tune the results using efficient local optimizers (hybrid GA), some operators have been changed to reduce the need for this additional postprocessing. The mutation operator was constructed to perform local search. A number of different crossover mechanisms have been tried, all based more or less on problem specific information. The results found support one of the philosophical axioms of the genetic approach: probabilistic choices are usually preferable to deterministic ones.

This is also an important aspect for constructing problem conditioned operators. Dependent on the problem specific knowledge, not only the operators can be altered, but also their probabilities. The mutation operator probability distribution for instance can be made different for changing some kind of alleles within genotypes violating the constraints than for changing the same kind of alleles within feasible ones [Hua, 94]. It can be a good strategy when the problem structure is clear and the optimum has to be found on the edge of a feasible region.

So, when operators have to be made or adapted to include some heuristics, there are two aspects to keep in mind; diversity should be maintained within the population and heuristics can be applied stochastically, or can contain some probabilistic decisions in order to improve the GA.
4. A Genetic Algorithm for MISP

4.1 An Introduction

The Minimal Input Support Problem is clear by now, and the basics of GAs has been dealt with, too. Time has come to combine the two things to get (hopefully) a minimal input support answer. Although we have devised an algorithm for this purpose, the many different (and often contradictory) recommendations found in the literature on GAs (see also Chapter 3) indicate the algorithm has to be adapted easily to new insights.

The choice for the programming language C++ is an easy one. All the parts can be treated like objects, and when something has to be changed (for instance the possible gene values, or the crossover operator), this can be done right within the object, without affecting the others. This language provides also dynamic allocation, so neither the number of alleles contained by a chromosome nor the population size has to be known beforehand.

The matrix containing the support information (see Chapter 2) has to be traversed every time a possible solution has to be checked. Therefore this matrix should be kept as small as possible. Moreover, the recommendations described in [Kon, 95a] suggest strongly to use a preprocessing algorithm. This resulted in a preprocessing algorithm including detection (and elimination) of dominated rows (as can be read at the end of Section 4, after some tests this algorithm has been modified to perform just once the dominated rows check). For this stripped matrix the genetic algorithm should find the minimal input support.

In Section 4.2 the genetic algorithm will be outlined. The operators will be described and elucidated. In the remainder of this chapter the interfaces of all the objects will be presented. By interface, public procedures are meant. The implementation of objects (the protected and private parts of the objects) will be discussed in the next chapter. All code has been written in C++ (using the Gnu compiler, Version 2.7.0) in a UNIX (HP-UX 9.05) environment. The reader of this report is assumed to be familiar with both of them.

4.2 The ideas behind the algorithm

The coding of the genes is binary; a 1 means the input bit is considered to be a part of the support, a 0 on the other hand means the specific bit will not become a solution member. Hence, the number of alleles equals the number of input bits. This is a naturally way of encoding, and there are no Hamming cliffs (see the problem topology in Chapter 2).

The problem now is how to ensure that heavily related genes are close to each other. The more two columns are disjunctive (to be measured by the number of rows covered by exactly one of the columns), the more likely it is that those columns are heavily related. Hence, a good coding algorithm should rearrange the columns to get a maximal hamming distance between successive columns. Unfortunately, this would be very time consuming. Therefore we have chosen not to do this. This implies nothing can be said about the influence adjacent genes will have on each other, so the 2-point crossover might not be the best choice (see also Chapter 3).

The population size has been fixed on two times the number of alleles within a chromosome, following the proposal of [Thi, 94]. The initial population is generated semi-randomly. Every
input bit has a probability to appear (become 1) in the initial chromosome dependent on the so-called importance.

The importance of an input bit should be based on the number of rows covered by that input bit and the difficulty to cover those rows (when only two input bits can cover a row, it is harder to cover that row, than it is to cover a row that could be selected by all but one input bit). Therefore, we have formulated an importance factor as follows:

$$\text{Importance}(col) = \sum_{row \,=\, 1}^{\text{#rows}} M_{row,col} \cdot \frac{1}{S(\text{row})^2}$$

$$S(\text{row}) = \sum_{\text{column} \,=\, 1}^{\text{#columns}} M_{\text{row},\text{column}}$$

$$M_{i,j} = \text{matrix element at row } i \text{ and column } j \text{ (0 \lor 1)}$$

The total of all the importance factors now becomes:

$$\sum_{\text{col} \,=\, 1}^{\text{#columns}} \text{Importance}(\text{col}) = \sum_{row \,=\, 1}^{\text{#rows}} \frac{S(\text{row})}{S^2(\text{row})} = \sum_{row \,=\, 1}^{\text{#rows}} \frac{1}{S(\text{row})}$$

The average importance factor should lead to a fifty percent chance to become a support member. So every importance factor lead to a probability for an allele to become 1 by dividing the factor by 2 times this average factor:

$$p(\text{allele } \text{col=1}) = \frac{\text{Importance}(\text{col})}{\sum_{row \,=\, 1}^{\text{#rows}} \frac{2}{\text{#columns} \cdot S(\text{row})}}$$

However, this formula does not guarantee the probability value will be in the range [0, 1]. To repair this (and prevent alleles from being just 1 or 0 - what might lead to premature convergence) an upper and lower bound have to be chosen as maximum and minimum probabilities. The initial population can now be simply constructed by generating the appropriate amount of chromosomes (for every allele the chance to be selected is known).

The fitness function has to promote better chromosomes. So two possibilities have to be distinguished. In case of a support (the set of input bits cover all the rows) the less input 1-alleles the chromosome contains, the better the fitness function should be. For the non-support situation (some rows are not covered by the set of input bits) there has to be a different approach. Now, the potential of the chromosome has to be tested. For instance, when only one row has not been covered by the chromosome, a full cover can be reached by changing just one 0-allele into a 1-allele. Therefore, we have decided to judge such a chromosome like it was repaired this way. More in general, for every chromosome failing to cover all rows, the fitness is
computed for a chromosome containing an extra number of 1-alleles equal to the number of rows not covered by the original chromosome\(^5\), and realising a support.

Using this method it is possible to use virtually chromosomes containing more 1-alleles than genes. However, this does not matter, as long as the fitness function values the chromosomes in accordance with their potential. The formulae for the fitness function have been chosen to be:

\[
\text{Fitness}(\text{Chromosome}_{\text{cover}}) = 1 - \frac{\#1\text{-alleles}}{2(\#\text{alleles})}
\]

\[
\text{Fitness}(\text{Chromosome}_{\text{nocover}}) = 1 - \frac{\#1\text{-alleles} + \#\text{rows notcovered}}{2(\#\text{alleles})}
\]

The meaning of \#1\text{-alleles}, \#\text{alleles} and \#\text{rows notcovered} is the number of alleles containing a 1, the total number of alleles in the chromosome and the total number of rows not covered respectively. Using these formulae a chromosome covering very few rows will have a lower fitness than the worst chromosome covering all rows (the chromosome containing just 1-alleles). The fitness should never drop between zero, thus if the number of rows not covered together with the number of 1-alleles exceeds twice the number of alleles, the fitness should become zero. Figure 4.1 shows the fitness function schematically.

![Fitness function](image)

Figure 4.1 Schematic overview of the fitness function.

The selection mechanisms of both the tournament and the roulette wheel procedures have advantages and disadvantages. The tournament selection requires less computation time, but the distribution is independent of the actual fitness values (it only depends on the ranking, see also Appendix C). The advantage of this is that one superfit phenotype does not capture the next generation. The advantage of the roulette wheel mechanism is that phenotypes will be selected proportional to their relative fitness, so when the phenotypes have just very small differences between their fitnesses, the expectation is that they all will be selected the same number of times. On the other hand there is a drawback of the superfit phenotype leading to premature convergence.

---

\(^5\)To be more correctly, after four uncovered rows have been found, the result will be estimated.
When a selection mechanism is required to overcome both the superfit problem and the almost equal fitness problem, both selection methods can be combined. Doing a part of the selection by the tournament principle and the other part of the selection with the roulette wheel, the result will be a next generation not captured by superfit phenotypes. It is also expected that almost equal phenotypes will be selected a comparable number of times. With the tested settings the tournament selection has been used 60% of the time.

The **crossover** operator has been chosen to operate only on the different alleles\(^6\). This implies that if both the selected chromosomes differ only at 1 locus, both chromosomes are copied. The operator is extended for the situation where both the chromosomes are identical. In that case one instance will be mutated, while the other will nevertheless be copied. As long as disjunctive columns are not rearranged to become adjacent, the uniform crossover operator is expected to be superior.

The **mutation** operator will be different for chromosomes representing supporting input bit sets and the chromosomes unable to cover all the rows. In the support case the chance to change a 1-allele into an allele containing a 0 has to be higher than the chance for a 0-allele to change. Of course, in the non-support case this has to be the other way around. The reason behind this is to create chromosomes that offer better solutions.

**Another possible mutation** option is to change a number of alleles into semi-randomly chosen values. This operator is useful when most of the genotypes cannot easily be changed to gain fitness. The traditional mutation operator would create inferior genotypes compared to the parent in this situation. Therefore alleles would still not be prevented from dying off. This operator selects a part (which size is at most one fourth of the total size) of the genotype and computes the ONE fraction (the number of ONE alleles divided by the total number of alleles). This fraction is modified to function as the probability for the alleles to become ONE. Dependent on the parent's supporting capability this probability will be slightly smaller (support) or bigger (non-support). For all alleles in the selected part a random number is tested against this probability and the allele will get the correct value.

We have developed yet another operator especially for MISP. It is called the **merge** operator. This operator has been included to examine good sections of the problem. As this is a sexual operator, two chromosomes are needed. The set of all 1-alleles (the union of the two sets) define the solution space. So only these alleles might stay 1, the others stay at 0 for sure. Within this reduced matrix it is checked if there are any essential or dominated columns corresponding with the set elements.

Using the same principles as in the preprocessing phase, the matrix is reduced. When no further reduction can be made and there are still columns left neither essential nor dominated, one of these columns is chosen based on the importance factors of the involved columns, and selected to become part of the input set (hence added to the essential set). This is repeated until all columns are dominated or essential. The essential set is translated to become a new chromosome.

\(^6\)Of course, when the uniform crossover operator is used, the results will be the same (although the computational time can be shortened when there will be less decisions to make based on random chosen numbers).
The Matrix: | The Phenotypes: | After the first reduction: |
---|---|---|
0 0 0 1 0 0 0 0 1 | Chromosome: 0 0 0 1 0 0 0 0 1 | 0 0 0 1 0 0 0 0 1 |
1 0 0 1 1 0 0 1 0 | [0 1 0 1 1 0 0 0] | 1 0 0 1 1 0 0 1 0 |
0 1 0 0 1 1 1 0 0 | Fitness: 14/18 | 0 1 0 0 1 1 1 1 1 |
1 1 0 0 0 0 0 0 1 0 | 1 0 1 0 1 1 1 0 0 | 1 0 1 0 1 1 1 0 0 |
1 1 1 0 0 0 0 1 0 | Chromosome: 1 1 1 1 1 0 0 0 0 | 1 1 1 1 1 0 0 0 0 |
0 0 1 0 0 1 1 0 0 | [1 0 1 0 1 0 0 1 0] | 0 0 1 0 1 0 1 0 1 |
0 1 1 0 1 0 0 1 0 | Fitness: 14/18 | 0 0 1 0 1 0 1 0 1 |
1 0 1 0 0 0 0 1 1 | 0 1 0 0 1 1 0 0 1 | 0 0 1 0 1 0 1 0 1 |
0 0 1 0 0 0 1 0 0 | Fitness: 13/18 | 0 0 1 0 1 0 1 0 1 |

Figure 4.2 Example of the merge operator.

Consider the matrix given in Figure 4.2a (there are neither essential columns, nor dominated rows or columns). Suppose this matrix is the result of the preprocess. The two phenotypes selected for the merge operator contain the chromosomes [0 1 0 1 1 1 0 0 0] and [1 0 1 0 0 0 1 0 0] (Figure 4.2b). The union of 1-alleles could be represented by [1 1 1 1 1 1 0 0 0], so the last two rows of the matrix may be left out (Figure 4.2c). Now the 4th column has become an essential one (see the A in the figure). After reducing (only the bold faced rows have now to be considered), the first column is dominated by the second. Because one row is only covered by the first and the second columns (see b), after the domination detection the second column becomes essential. Reducing the matrix (removing the underlined rows) leaves us with three rows to be covered, and all three of them are actually covered by the third column (what makes the other columns automatically dominated ones, see " in the figure). So the merging operator returns the chromosome [0 1 1 1 0 0 0 0 0]. In this example the reduce procedure generated a set of essential columns, while the supplementary set equalled the set of dominated columns. Therefore, there was no need for preferring one column over another one.

The last operator we have added to the genetic algorithm is the repair operator. This operator can only be applied on a phenotype that does not cover the matrix completely. It adds a number of 1-alleles to the chromosome (hopefully the lowest number possible), to make it an input support.

The principle of this operator has something in common with the merge operator. First, all rows covered by the chromosome are deleted from the matrix. Thereafter the matrix is reduced, using the importance factors when neither essential nor dominated columns can be found. The essential set together with the original 1-alleles are the new 1-alleles of the chromosome.

The Matrix: | The Phenotypes: | After the first reduction: |
---|---|---|
0 0 0 1 0 0 0 0 1 | Chromosome: . 0 1 0 . 0 0 0 0 | . 0 1 0 . 0 0 0 0 |
1 0 0 1 1 0 0 1 0 | [0 0 0 1 0 0 0 0 0] | . 0 1 0 . 0 0 0 0 |
0 1 0 0 1 1 1 0 0 | Fitness: 13/18 | . 0 1 0 . 0 0 0 0 |
1 1 0 0 0 0 0 0 1 0 | 1 0 1 0 1 1 1 0 0 | . 0 1 0 . 0 0 0 0 |
1 1 1 0 0 0 0 1 0 | Chromosome: 1 1 1 1 1 0 0 0 0 | . 0 1 0 . 0 0 0 0 |
0 0 1 0 0 1 1 0 0 | [1 0 1 0 1 0 0 1 0] | . 0 1 0 . 0 0 0 0 |
0 1 1 0 1 0 0 1 0 | Fitness: 14/18 | . 0 1 0 . 0 0 0 0 |
1 0 1 0 0 0 0 1 1 | 0 1 0 0 1 1 1 0 0 | . 0 1 0 . 0 0 0 0 |
0 0 1 0 0 0 1 0 0 | Fitness: 13/18 | . 0 1 0 . 0 0 0 0 |

Figure 4.3 Example of the repair operator.

Consider the same matrix as in the previous example (repeated in Figure 4.3a). The phenotype selected for the repair operator contains the chromosome [1 0 0 1 0 0 1 0 0] (Figure 4.3b). Looking at the corresponding columns, it can be seen that there are just two rows not covered (the bold face 1s in the matrix of Figure 4.3a represent a covered row by a selected column, and the underlines rows are the set of covered rows). So, after the first reduction there are just two rows left, and six columns (see Figure 4.3c - the dots indicate the already selected columns). There is one column dominating
the other ones (see the \( A \) in the figure), so this column will be selected too. The repair operator will therefore return the chromosome \[1 0 1 1 0 0 1 0 0\].

Of course, these operators should not be used with the same frequency. Every operator has its specific function. At the very first start, the mutation operator should only have a very small chance to be called (as it is mainly meant to maintain the diversity within the population), just as the merge operator (else it will proceed with almost all of the bits), while the crossover operator has to be called often (new areas have to be discovered). When the process advances, the merge operator should become more important at the cost of the crossover operator (the promising sections have been found and have to be exploited). The mutation operator also should gain influence.

As a matter of fact, we have chosen to call just one operator per selected phenotype. In this way, the effects of an operator are not disturbed by another operator. The only exception is for the repair operator, as this operator will only be applied to just (by another operator) created phenotypes that do not cover the matrix. So, when crossover has been applied, no mutation nor merge operators will be called thereafter. When neither crossover, nor merge, nor mutation has been executed, the selected phenotype will just be copied.

One other idea we have decided to add to the genetic algorithm is a sort of immigrate aspect. After a while the population will more or less converge. To create new offspring the mutation operator has been invented. Another possibility is to create new elements 'from scratch'. When the algorithm has run for some time the major part of the next generation will be formed by the normal procedure, but a small amount of phenotypes will be created just like during the first generation. This fresh blood can stimulate a new exploration.

The algorithm can go on endlessly. In the long run there will be the perfect answer, but that is obviously not intended with genetic algorithms. We decided not to investigate a highly sophisticated stop condition at this point, but to test the algorithm first. To do this, the program has to be developed in such a way it would be able to create as many generations as wanted, and to use the newest generation as a base to make offspring again.

These ideas are just meant to form a starting point from where the algorithm should grow towards a good performance. The final program should rely on empirical evidence.

4.3 Some general classes

Many parts of the complete implementation are only concerned with 1s and 0s. As a result, there are many procedures operating on bit level. Because some other researchers involved in related projects needed similar possibilities, such a class has been constructed already\(^7\). The name of the class is `BitStr`. An instance is able to contain a set of bits. The range of the set is given as the argument with the constructor. Some very powerful operations are implemented for two instances, like the union and intersection operators. The class will not be described in this report.

\(^7\)Hans van Pinxteren improved a class made by Marcel Kolsteren, who adapted a class described in [Kon, 95b] for this task.
At many points in the program the size of the problem has to be known. Therefore a class has been added. Its only purpose is to return this size (or in other words the number of loci in the chromosomes). As can be seen in Figure 4.4, in this class called *OverallSize* there is just one public function, Loci, returning the needed integer.

```cpp
class OverallSize
{
public:
    static unsigned Loci( );
};
```

Figure 4.4 The public part of the class *OverallSize* (defined within *generic.h*).

Another tool we needed for the project were some almost identical, very simple, list structures. Therefore, we have constructed a general basic class, which could be used to create the more specific classes. The interface can be found in Figure 4.5.

```cpp
template <class Class>
class ClassList
{
public:
    ClassList( const unsigned n);
    virtual void Add( Class Element);
    void MakeEmpty( );
    unsigned Count( ) const;
    unsigned MaxCount( ) const;
    Class Position( unsigned i) const;

    ~ClassList( );
    ClassList< Class>& operator=( const ClassList< Class> &XCL);
    ClassList( const ClassList< Class> &XCL);
};
```

Figure 4.5 The public part of the class *ClassList* (defined within *liststruct.h*).

The class *ClassList* can be used for other classes to store any type in a list, by inheritance this class with the appropriate type. We will focus now on the possibilities it offers in respect to the data. The constructor needs an integer to allocate enough memory. This integer namely decides the number of instances that can be stored. The data can be stored using the procedure Add. When there is not enough room to store the data, the procedure does not do anything (not generating an error message, too). All data can be flushed using MakeEmpty. The number of instances the class can contain remains constant, when using this routine.

The procedures Count and MaxCount will return respectively the values of the number of instances already stored, and the number that can be stored in this list. To read an element of the list, Position is called with the number of the position as the argument. This number should be minimal 0 and maximal the value returned by MaxCount minus 1 (the list is treated like an ordinary array).

The copy constructor, assignment operator and destructor are really implemental aspects and will be described in the next chapter.
class SupportList: public ClassList<unsigned>
{
public:
    SupportList(const unsigned un): ClassList<unsigned>(un);
    friend ostream& operator <<
        (ostream &out, const SupportList &SList);
};

Figure 4.6 The class SupportList (defined within liststruct.h).

In Figure 4.6 a derived class has been presented. Just unsigned integers can be stored in this class. The main purpose of this class is to represent the members of the BitStr. At position 0 the lowest position with a bit containing a 1 can be found, the next lowest position can be found at position 1, and so on. All procedures implemented within the class are the same as those of the basic class. The only new aspect is that instances of this class can be written to standard output. This will result in a sequence of 0s and 1s, to represent the underlying bits.

class CrossoverList: public SupportList
{
public:
    CrossoverList(const unsigned un): SupportList(un) {}
    void Remove(unsigned first, unsigned last);
    friend ostream& operator <<
        (ostream &out, const CrossoverList &XList);
};

Figure 4.7 The class CrossoverList (defined within liststruct.h).

Another similar class is called CrossoverList (see Figure 4.7). It is derived from SupportList, and one procedure has been added. The purpose of this class is to store all alleles that can or will be changed. When two genotypes are compared, all different values can be stored in the list. This list forms a base for the crossover operator (hence the name). Never will more alleles have to change between the involved chromosomes than the number of elements stored in this list.

The only new procedure has been called Remove. Its effect is that the elements of the list at the positions between the arguments (included) are removed from the list. This is especially valuable for the 2-point crossover operator, in order to remove the selected positions and to perform a swap operation on the alleles indicated by the elements of the remaining list.

4.4 Preprocessing modules

Although algorithms to solve MISP have been very well implemented already with a preprocessing module (see [Kon, 95a]), a new code has been written for this task. There were some reasons to do this. First, and most important, a number of functional elements had to be added to make the old implementation suit the requirements of the genetic algorithm. Second,

\footnote{It is obvious that the results of the crossover operator will be exactly the same whether the selected part is used or the list without the selected part.}

30
the code of the existing implementation has never been documented elsewhere than in the program itself. Together this means the time needed to get to the bottom of the old program would be probably more than the time needed to build the new code. Besides, as long as the algorithm has not proved to generate good results, the compatibility is of minor concern.

The implementation has been split into two classes. The class MakeMatrix is used to read an input file containing the Boolean function that indirectly formulates the minimal input support problem, into memory. The class Matrix contains just a matrix and has some procedures to manipulate it and compute some vital functions.

class MakeMatrix
{
public:
    MakeMatrix( ifstream &in);
    ~MakeMatrix( );
    Matrix Do( ) const;

    unsigned NumOutp( ) const;
    unsigned NumInp( ) const;
};

Figure 4.8 The public part of the class MakeMatrix (defined within makematx.h).

In Figure 4.8 the available procedures of the class MakeMatrix are showed. There is just one constructor, and it needs an input file reference. The corresponding input file should contain the Boolean function MISP has to reduce. The format of this file is just a little bit more restrictive than other, similar, programs allow. The restriction is that just one of the outputs of a line may be specified (a 0 or a 1). Besides, all lines specifying the first output bit should come first, thereafter the lines specifying the second output bit, etc. All test files described in [Kon, 95a] fulfil these restrictions.

The constructor reads all the lines and put them into memory. This indicates the reason a destructor has been made - but as this is more an implementation aspect, it will be described in more detail in the next chapter.

The Do procedure returns a Matrix, containing the coverage matrix defined by the Boolean function within the file. The other two available functions of this class are (Figure 4.8) NumOutp and NumInp. These functions return respectively the number of output and input bits defined by the file.

Hence, every instance of the class MakeMatrix can just read one input file to form a matrix. When several input files should be read, a corresponding number of instances should be created. Of course, when some identical matrices have to be made based on a single input file, the Do procedure can be called a number of times.
The class *Matrix* contains some more public procedures, as can be seen in Figure 4.9. The constructor creates the boundaries for the matrix; the maximum number of rows is given as the first argument, while the number of columns is read as the second one. The boundaries cannot be adapted later for the same instance without losing the information already put into the matrix, so during declaration the maximum number of rows and columns should be known. These numbers should minimally equal 1 (as a matter of fact a smaller matrix is meaningless).

Rows can be added by using the Add procedure. A row is represented by a *BitStr*. Essentially, this is a class containing a set of bits. This class has been described in the previous section a little bit deeper. When a number of rows equal to the row boundary (as given in the first argument of the constructor call) has been added, new rows cannot be added (the *Matrix* has already been filled). However, the procedure can be called, and will not return an error, but will not try to do anything with the row either. The data will just be lost.

A very important procedure, Reduce, will construct a new matrix containing only non-essential and non-dominated columns, and non-dominated rows (see for more details Chapter 2). To do this, the matrix within the class is searched on essential and dominated columns. These columns are removed from the matrix. As long as new columns are removed, this searching is repeated. Thereafter is checked on dominated rows. If they are found, they are removed and the whole
process is started all over again\(^9\), else the process terminates and fills the new matrix with the rows and columns left.

Because the reduced matrix does not have automatically the same number of columns, the minimal input support corresponding to that matrix cannot be applied to the original matrix (and hence the problem). The function Construct overcomes this problem. The class SupportList (as has been explained in the previous section) contains the positions of the 1s of a binary representation. So, when the solution has been found for the reduced matrix (a set of 1s corresponding to the columns being part of the solution), the SupportList representation is given as the argument for the Construct procedure and the corrected solution (with the essential columns added and the positions translated into the original ones) will be returned.

The procedure Merge executes the merge operator for the genetic algorithm (see also Section 2). A SupportList containing the combined on-sets (logical OR) of the two selected chromosomes determines what columns have to be taken into account. On these columns the procedure will be applied. The best columns are selected in order to cover all the rows. It is not checked whether the combined on-sets actually cover all the rows in the matrix, but this function considers only the rows that are covered. The return value of the procedure is also a SupportList, containing the necessary bits. The procedure will not affect the matrix (it makes a copy to operate upon), as can be seen actually from the const behind it (Figure 4.9).

To repair chromosomes (see Section 2), the on-set of the chromosome is needed. This is given as the SupportList argument in the procedure Repair. All rows not covered by these columns have to be covered by as few extra columns as possible. Therefore, the matrix will be reduced (considering only the not covered rows and not selected columns). When neither essential nor dominated columns can be found, dependent on the importance factor one column is selected and added to the essential set. This process is continued until all rows have been covered, and the essential set is added to the given on-set. This new set will be returned by the procedure.

The function UncoveredRows also is primarily made for the genetic algorithm. An on-set is given, and the number of rows that are not covered by this set is returned. A 0 indicates all the rows are covered by the set of columns, where the other values means this is not the case (hence the higher this value, the less useful the on-set will be). This is an important feature in order to compute the fitness. To prevent the algorithm from taking too long computing uncovered rows, after four uncovered rows have been found, the result will be estimated (based on searched rows and number of uncovered rows found).

The functions MaxRowNow, MaxRow and MaxColumn return respectively the number of rows inserted into the matrix, the maximal number of rows and the number of columns for the matrix. The functions ColumnSum and RowSum require the column number or row number respectively, on which the function has to be applied. The value returned equals the number of 1s in the column or row.

A specific row can be got by using RowSup (returning the SupportList). Similarly, a column can be returned in this format using ColumnSup. The set of essential or dominated columns can be

\(^9\)Due to test results the procedure has been changed to never start over again - see also the subsection Note at the end of this section.
accessed by using \texttt{EssentialList} respectively \texttt{DominatedList}. These calls are useless when the \texttt{Reduce} procedure has not been applied (there are neither essential nor dominated columns detected at that moment).

The \texttt{friend} operator realizes that the matrix can be put on the screen (on the standard output, actually). The other procedures will be described in more detail in the next chapter. Those procedures are to assure that instantiations can be copied or assigned, and that memory will be deallocated properly.

\textit{Note}

The execution time of the algorithm to only perform the merge operator took very long, as can be read in Section 6.2. Therefore, the procedure \texttt{Reduce} has been changed. The procedure will search the matrix on essential and dominated columns and will remove them, just like the original procedure did. The reduced matrix is checked once on dominated rows, and thereafter the new matrix is filled, no matter whether actually dominated rows had been removed or not.

4.5 Chromosomes and phenotypes

As the description of the genetic algorithms were full of chromosomes and phenotypes, it is clear the program should use these entities too. In Figure 4.10 the interface of the chromosomes is represented.
enum Allele { ZERO, ONE };

class Chromosome
{
public:
    Chromosome( );
    Chromosome( const SupportList &SupList);
    void Initialise( );

    void Support( SupportList &SupLi) const;
    unsigned SupCount() const;
    Allele Locus( unsigned i) const;
    void SpecifyGene( unsigned i, Allele allele);

    void MutateSome( bool IsCover);
    void Mutate( bool IsCover);

    void Uniqueness( const Chromosome &Second,
                    CrossoverList &Xov) const;
    bool Identical( const Chromosome &Second) const;
    void SwapDifferent( unsigned i, Chromosome &That);
    void Merge( const Chromosome &Second, SupportList &SupLi) const;

    Chromosome& operator = ( const Chromosome &C);
    Chromosome( const Chromosome &C);
    friend ostream& operator <<
                   (ostream &out, const Chromosome &Chrom);
};

Figure 4.10 The public part of the class Chromosome (defined within genetic.h).

There are two constructors. The constructor without an argument creates a chromosome with all allele values set to ZERO. When a SupportList (details in Section 3) has been given, the chromosome will have alleles containing ONEs at the positions defined in the list. The Initialise routine can be called to attach some more or less randomly chosen values to the alleles. The probability to assign a ONE depends on the position of the allele and the matrix containing the representation of the problem (see also Section 2). The values the alleles might have, do not influence the result of this routine.

A Support call will put all the chromosome's alleles with the value ONE in a SupportList. This list has to be given as an argument to prevent the program from generating an identical list for every Chromosome endlessly. When just the number of ONE alleles is needed, SupCount can be used. A specific allele's value can be claimed using Locus with the position. An allele can be changed (actually, awarded a value) using the SpecifyGene command. The arguments have to be the position (locus) and the new value.

---

10To be more correct, this is a genotype, but for the sake of simplicity we will use the term chromosome in the description of this class also to indicate the specified contents.
The MutateSome procedure will affect a randomly chosen set of alleles. Minimal two and maximal a quarter of the total number of alleles will be affected. The number of ONE alleles within this set is count and leads together with the argument (indicating whether the chromosome contains a solution to the input support problem) to a probability factor. Every involved allele has a chance equal to this factor to become ONE, else it will become ZERO (see also Section 2). When the argument is TRUE (cover), on average there will be 10 percent less ONE alleles in the chosen set. On the other hand, for FALSE (no cover) this will be 15 percent more (hence the difference is 25 percent).

The Mutate operator will either call the previous described procedure or will change a single allele into its opposite. This depends on the probability settings, as will be described later (Section 8). Suppose just one allele will be changed. When the chromosome represents a solution to the input support problem, there will be more chance to mutate an allele containing a ZERO than an allele containing a ONE. When the chromosome does not represent such a solution, the opposite is true. The argument of the function indicates whether the chromosome is a solution or not.

Some procedures require a second chromosome. The procedure Uniqueness traces the differences between the two chromosomes and those positions will be put into a list called CrossoverList (almost identical to SupportList, as described in Section 3), which is given as the second argument. The bool returned from Identical indicates whether the two chromosomes are identical or not. If two chromosomes have different alleles at a specific position, SwapDifferent can be called and will change the alleles at that specified position. The Merge procedure fills the list containing all loci at which minimal one chromosome contains an allele with a ONE value. This list is given as the second argument. For these procedures it does not matter which chromosome is given as an argument (they are kind of associative).

For the Chromosome class an output operator has been developed, too. The values of the alleles will be printed in the same order as the corresponding numbers of their loci would suggest. The other two procedures are added to copy Chromosomes correctly.

```c++
class Phenotype
{
public:
    Phenotype( );
    Phenotype( const Chromosome &Ge);
    float Fitness( ) const;
    Chromosome Genotype( ) const;
    void Mutate( );
    Chromosome MutateInv( ) const;

    Phenotype& operator = ( const Phenotype &P);
    Phenotype( const Phenotype &P);
    friend ostream& operator<< (ostream &out, const Phenotype &Phe);
};
```

Figure 4.11 The public part of the class Phenotype (defined within genetic.h).

The class Phenotype consists of a genotype and the corresponding fitness. Again there are two constructors, as can be seen in Figure 4.11. The one without an argument creates a genotype with
just ZERO alleles and the fitness will become $0^{11}$. The other one needs a Chromosome and computes the accompanying fitness. The function Fitness returns the fitness value, while Genotype leaves us with the Chromosome containing the genotype.

The procedure Mutate mutates the genotype by calling the Chromosome::Mutate procedure. To keep the Phenotype consistent, the fitness will be computed thereafter again, of course. The procedure MutateInv does not affect the Phenotype, but returns the mutated genotype using the Chromosome::MutateSome procedure. In order to copy Phenotypes correctly, two procedures are added. Finally, there is a procedure to write the Phenotype to the standard output, too; both the genotype and the fitness will be printed.

4.6 The population

```cpp
class PopElement
{
public:
    PopElement( );
    PopElement( Phenotype phen, const float Extra);
    Phenotype View( ) const;
    float Cumulative( ) const;

    PopElement& operator = ( const PopElement &P);
    PopElement( const PopElement &P);
    friend ostream& operator <<
    (ostream &out, const PopElement &PE1);
};
```

Figure 4.12 The public part of the class PopElement (defined within genetic.h).

The population contains a number of phenotypes, but for implemental convenience we have made a class called PopElement (see Figure 4.12), that keeps track of the cumulative fitness as well. This comes in handy when the roulette wheel selection mechanism has to be performed.

Except for the empty constructor, there is a constructor that needs the Phenotype and the cumulative fitness before this PopElement has been added. The two functions View and Cumulative return the stored Phenotype and the float equalling the cumulative fitness. The stream out operator outputs the genotype, the corresponding fitness and the cumulative fitness. The other two procedures take care of the copying of PopElements.

---

$^{11}$The correct corresponding fitness might be higher, but as this is not of interest it will not be taken into account.
class Population: public ClassList< PopElement>
{
public:
    Population( const unsigned size = 1);
    float BestFitness( ) const;
    unsigned IndexBest( ) const;
    Phenotype BestEver( ) const;
    void NextPopulation( const unsigned TwoPointXover = 1);

    Population& operator = ( const Population &P);
    friend ostream& operator «(ostream &out, const Population &Popu);
};

Figure 4.13 The public part of the class Population (defined within genetic.h).

The class Population, as partly shown in Figure 4.13, has been derived from ClassList with the type PopElement as the elements of the list. The constructor needs the population size and then fills the population with the correct number of semi-randomly constructed elements (see also Section 2). The functions BestFitness and IndexBest will return the value of the highest fitness and the position of the corresponding element within the population respectively. When two or more phenotypes share the highest fitness, the position of the first in the list will be returned. The best element found in all the generations of the Population is kept also, and can be claimed using BestEver.

The procedure NextPopulation selects the most promising elements in the current population and constructs the new population using the crossover, merge, mutation and copy operators. The new generation contains exactly the same number of individuals, and replaces the old generation. The argument decides whether uniform crossover should be applied (0) or how often 2-point crossover should be applied. Only the different alleles are involved with the crossover operator. When for instance the value 2 is taken to be the argument, two times 2-point crossover is executed. Most of the time this will lead to 4-points crossover, but when either the start or the stop point is chosen to be the same the two times the operator is executed, the net result is a 2-point crossover. Seldom both the start and the stop point will be chosen exactly the same two times, but when this happens, unfortunately no crossover at all has been applied effectively.

The other procedures take care of assigning and displaying of a Population.

4.7 Making the genetic algorithm work

With the classes described in the previous sections, the genetic algorithm can already almost be performed. Some things we are still missing are the link between the matrix and the population and the adaptation of the probabilities. To overcome the problems, a new class has been created, called GeneticAlgorithm. The public part is listed in Figure 4.14.
class GeneticAlgorithm
{
public:
    GeneticAlgorithm( const Matrix &Matr, unsigned SizePop = 1);
    void Continue( const unsigned Generations = OverallSize::Size);
    void StartOverAgain( unsigned Ab);
    void StartOverAgain( float Rel = RelSizePop);
    Phenotype BestOne( ) const;
    Population Pop() const;
    void AdaptCrossProb( const float NewXProb);
    void AdaptUnifXProb( const float NewUXProb);
    void AdaptPointCross( const unsigned NewPoint);
    void AdaptMutateProb( const float NwMtProb);
    void AdaptMutateSomeProb( const float NewSomeProb);
    void AdaptMutateRightProb( const float NewRightProb);
    void AdaptMergeProb( const float NewMrgProb);
    void AdaptRepairProb( const float NewRprProb);
    void AdaptHeuristProb( const float NewHrstProb);
    void AdaptSizeOfNew( const unsigned NewSize);
~GeneticAlgorithm();
};

Figure 4.14 The public part of the class GeneticAlgorithm (defined within ga.h).

The constructor needs the matrix containing the problem. At best this matrix has already been reduced. The genetic algorithm will not reduce the matrix any more. As the genetic algorithm will access the matrix very often (every constructed phenotype has to be checked against the matrix to compute its fitness, and the merge operator will operate directly on a subset of the matrix), the computing time will be unnecessarily high when the matrix has not been reduced beforehand.

The constructor initialises some classes now (for example, the class Chromosome needs to know the number of alleles involved). It thereafter calls Population to form the first generation. The second argument defines the number of elements within the population. Hence, when this unsigned integer is omitted, the size of the population will equal 1.

The procedure Continue has the responsibility to create more generations. The number of iterations can be specified. When no argument is passed, the number will equal again the number of alleles (the class OverallSize is a general class containing just the number of alleles - see also Section 3).

To start with a new population, the procedure StartOverAgain has to be called. This procedure will do the same as the constructor, except for the initialise part. However, there is one big difference. Instead of calling the procedure with an unsigned integer to define the size of the new population, the procedure can also be called with a floating point number. In that latter case the number of elements within the population will be based on the number of input bits (alleles) of the problem. To be more precisely, the size of the population will equal the argument times
the number of input bits. If no argument is given, \texttt{RelSizePop} is used to define the (relative) size (see next section).

The function \texttt{BestOne} returns the best phenotype seen in all the generations. However, when \texttt{StartOverAgain} has been called, this value will be reset. The current population can be claimed using \texttt{Pop}.

The procedures \texttt{AdaptCrossProb}, \texttt{AdaptMergeProb} and \texttt{AdaptMutateProb} can be called to adapt some probabilities, namely the crossover, the merge and the mutate probability respectively. It should be kept in mind that the probabilities have to be translated to the values given in order to perform properly, because the algorithm tests sequentially whether the operators have to be performed, and when one has been applied, the others will not be checked\footnote{Using the notation $P_i$ for the probability an operator will be chosen and $A_i$ for the given argument (with $i = x$, $me$ or $mu$ to indicate respectively crossover, merge and mutation), the following equations will hold: $P_x = A_x$, $P_{me} = (1-P_x)A_{me}$ and $P_{mu} = (1-P_x-P_{me})A_{mu}$. The remainder $(1-P_x-P_{me}-P_{mu})$ is the probability no operator will be applied and a phenotype will just be copied.}

To adapt the behaviour of the \textbf{crossover operator} \texttt{AdaptPointCross} can be called. The number given as argument prescribes whether uniform crossover has to be applied ($0$), or in the other cases the number of times the 2-point crossover has to be performed. For uniform crossover, the behaviour can be further specified by calling \texttt{AdaptUniFracProb}. The argument defines the fraction of the parent's alleles that will not be copied into the child (the other alleles will be copied from the other parent).

The procedure \texttt{AdaptMutateSomeProb} changes the probability that a part of the \textit{Chromosome} will be affected by the mutation operator instead of just a single one (see also Section 5). To change the probability for the mutation operator to mutate a $1$ into a $0$ for an input cover genotype or a $0$ into a $1$ for a non cover, \texttt{AdaptMutateRightProb} can be called.

To change the probability a non-cover genotype will be repaired, \texttt{AdaptRepairProb} can be called with the new probability. As described in Section 2, a heuristic algorithm computes the new values when the merge or repair operator have to be applied. However, the genetic algorithm allows to choose random columns too, instead of taking only the column with the highest important factor as defined within the heuristic algorithm (see also Section 2). The probability can be adapted using \texttt{AdaptHeuristProb}. The higher the argument, the less the columns will be picked at random.

To create the immigrate aspect as described in Section 2, \texttt{AdaptSizeOfNew} has been added to the genetic algorithm. The number of genotypes created 'from scratch' instead of by selection and operations is given as the argument. In subsequent generations there will still be immigration; this number will be halved every next generation (rounded down), until it becomes five or less. From then per generation one will be subtracted from this number until it reaches zero.

The destructor will end some memory claims. It will be described in the next chapter.
4.8 The default settings

The program has some default settings. As has been described in the previous section, a few can be adapted during the execution. All these settings have been collected in the same file, so there will be just one place to change them.

```c
float ProbAlleleUniformCross = 0.5;
float ProbMutNotJustOne = 0.5;
float ProbMutRightSide = 0.9;

float ProbCrossover = 0.6;
float ProbMerge = 0.2;
float ProbMutAfterX = 0.1;

float RepairProb = 0.0;
float HeuristProb = 0.5;

unsigned SizeOfNews = 0;

double UpperAlleleProb = 0.8;
double LowerAlleleProb = 0.3;

unsigned DefaultXPoint = 0;

float RouletteProb = 0.4;

float RelSizePop = 2.0;
```

Figure 4.15 The default settings (defined within frandom.cxx).

The variables and their default values are shown in Figure 4.15. Most of them identify probabilities. The use of these variables will be depicted here.

The ProbAlleleUniformCross will only be used when uniform crossover is applied. It represents the part of the differences between the two parent genotypes that will be expected to stay together after crossover. So, when alleles 0, 1, 3 and 4 do not have the same value in the two genotypes and this probability value is 0.75, it is to be expected the children will differ at just one allele from their most similar parent. The effect of this parameter will be the same for value \(x\) and \(1-x\) (assumed \(x\) is in the range \([0, 1]\)). The program will swap less when \(x\) is chosen to be in the range \([0, 1/2]\).

The question whether just one single allele will be changed or a number of alleles will be affected, depends partly on ProbMutNotJustOne. This represents the probability the procedure `Chromosome::MutateSome` will be called by the `Chromosome::Mutate` procedure.

The value ProbMutRightSide represents the probability for a ONE allele to change into a ZERO allele when the genotype happens to be a cover, or vice versa for a non-cover. This probability is only used when just a single allele will be mutated.

The probabilities to have an operator applied, are called ProbCrossover, ProbMerge and ProbMutAfterX. Actually, these are not the real probabilities (see also the previous section). First is tested whether the crossover operator has to be called. Because two different operators
(except for the repair operator) never will be applied to the same selected genotype(s), the test for the merge operator will only be executed when the crossover operator has not been called. For the same reason the mutate operator can only be applied when neither crossover nor merge has been applied. The formula to compute the real probabilities can be found in the footnote in the previous section.

A warning about the probabilities will be justified. To compute the expected number of offspring created by each operator, one should keep in mind that only the crossover operator generates two children. So there will be a difference between the probability an operator will be applied and the probability a child has been generated by that operator.

As described in the previous section, the probability a non-cover genotype will be repaired is not fixed. The initial setting can be changed by altering the value of RepairProb. Also HeuristProb is responsible for the probability the importance factor will be used to compute the results of the merge and repair operators (see also Section 2). The initial immigrate aspect can be found in SizeOfNews.

The two variables UpperAlleleProb and LowerAlleleProb indicate the maximum and minimum probabilities for an allele to become ONE in the semi-randomly initialised population (see also Section 2 at the part describing the importance factor and Section 5 at the procedure Initialise). The values should be chosen in the range $0 \leq \text{LowerAlleleProb} \leq \text{UpperAlleleProb} \leq 1$. The closer their values will come to 0 and 1 respectively, the bigger the chance is the initial population could have converged already.

The next variable indicates the default crossover setting. If DefaultXPoint equals 0, uniform crossover will be applied, else the value equals how many times the 2-point crossover has to be applied.

Whether the selection within a Population is made using the roulette method or the tournament ranking depends on the value of RouletteProb. If a random number is bigger than this value, the tournament ranking will be used.

As can be read in the previous section, RelSizePop defines the size of the population relative to the problem dimension when GeneticAlgorithm::StartOverAgain has been called without an argument. Default this value has been set to 2.0, according to the ideas behind the algorithm as outlined in Section 2.
5. Implementation

5.1 Class ClassList

The class ClassList contains two unsigned integers and a pointer to an arbitrary class, next to some procedures (see Figure 5.1).

```cpp
template <class Class>
class ClassList
{
private:
    void construct( const unsigned n);
protected:
    unsigned MaxNumber;
    Class *List;
    unsigned CurrentPosition;
public:
    ~ClassList( ) { delete[ MaxNumber] List;}
    ClassList< Class>& operator=( const ClassList< Class> &XCL);
    ClassList( const ClassList< Class> &XCL);
};
```

Figure 5.1 The implemental aspects of the class ClassList (defined within liststruct.h).

The pointer addresses an array using the new command. The width equals the number given as an argument for the constructor, which will also been stored in the variable MaxNumber. The number of values stored already is kept in CurrentPosition (at the start this value will equal 0, of course). When the constructor has been called, the private procedure construct is used to assign the initial values. The classes using this class as a basis do not have to access this procedure, although they have to access the variables (of course). Hence the difference between the private and the protected part.

The need for the destructor, assignment operator and copy constructor is now clear, too. The new command requires a delete command, which will be performed when the destructor is called. The assignment operator and copy constructor make the program copy the data instead of just copying the memory address.

Both classes SupportList and CrossoverList are derived from this class, as can be read in Chapter 4. The implementation of these classes is very straightforward. As a matter of fact, they do not have private nor protected parts (apart from the derived ones). Therefore, they do not have to be explained in detail.

5.2 Classes MakeMatrix and Matrix

The class MakeMatrix has to read a file, store input bits separately for every output bit value and make a matrix based on, roughly speaking, the output bit wise differences between the on and off sets (see for more details Chapter 2). To prevent the class from reading the file over and over again, much information has to be stored. The class contains therefore a number of variables (see Figure 5.2).
Both the numbers of input and output bits in the file are stored in the variables InputBits and OutputBits respectively.

All the pointer types are actually used as arrays. The OnSet keeps the input bits leading to an output bit having the value 1. The OfSet is used for the other input bits. As an input may contain several don't cares and the class BitStr can hold only a set of bits, every two subsequential values in the array together will code the input. At the places they do not differ, that value indicates the value of the corresponding input bit. At the positions the values do differ, the corresponding input bit is not specified (don't care).

To handle both the OnSet and OfSet arrays correctly, the size of both of them have to be known, just like the number of values stored already. The corresponding variables are OnSize and OfSize respectively. The input bits have to be kept apart for every output bit, so the arrays OnSeparation indicate the start/stop positions in the input array for every output bit.

When a BitStr array does not have enough space to keep all the data, a new one is constructed, and the values are copied. This ensures a flexible program. It is done in the procedures ExpandOnSet and ExpandOfSet.

The main goal of the constructor is to fill these variables properly. Some procedures have been written to make the task more convenient. The procedures CharAfterPointAtLineStart, SkipSpaces and SkipLine will skip meaningless characters and make the code more robust. When a number has to be read (number of input bits for example), ReadNumber does the job.

The Error procedure is called when something in the input file format is not right. The code will get rid of the stored values and put a message on the standard error.
The procedure Compare is only called by the Do procedure. The Do procedure computes the number of rows needed in the matrix. For every row bit for bit (column for column) is checked what value has to be put in it. For this purpose Compare will be called. Just in case the OnSet differs from the OfSet at the specified position and no don't cares are involved, the column has to contain a 1. The procedure needs therefore the positions in the two arrays and the bit position. The Do procedure is not very fast (it checks bitwise bit sets in two arrays at two positions). Probably, there can be much speed gained here. But, as has been mentioned before, this project has to be seen as a pilot project to test whether GAs would be useful when applied to MISP. The step to make it more efficient has to be taken when the program has showed its usefulness.

The destructor will execute the delete commands. The total code for this class can be found within the files makematx.h and makematx.cxx.

The class Matrix relies heavily on the class BitStr (see also Sections 4.3 and 4.4). As a matter of fact, the rows and columns are types of this class. The matrix contains two arrays to hold them (see Figure 5.3).

class Matrix
{
    private:
        unsigned NumberOfRows, NumberOfColumns,
            *CheckSumRow, *CheckSumColumn;
        BitStr *Rows, *Columns, Essential, Dominated;
        unsigned CurrentRow, MaxCheckSumColumn;
        SupportList ReturnFromRow, ReturnFromColumn;
        unsigned NewRows, NewColumns;

        void Row( unsigned Number);
        void Column( unsigned Number);

        void EssentialAdd( const unsigned ColumnNumber);
        void EssentialCol( const unsigned RowNumber);
        void MakeRowEmpty( const unsigned RowNumber);
        void MakeColumnEmpty( const unsigned ColumnNumber);

        SupportList Heuristic( ) const;
        void Reducing( );

        void construction( const unsigned Row, const unsigned Column);
        void FillConstruct();
        void CopyFrom( const Matrix &Mat);
        void DeleteAll();

    public:
        ~Matrix( );
        Matrix& operator=( const Matrix &Mat);
        Matrix( const Matrix &Mat);
};

Figure 5.3 The implemental aspects of the class Matrix (defined within matrix.h).
The variables `Rows` and `Columns` both contain the whole matrix. This means there is much redundancy in the class. This can slow things down. On the other hand, the code will become more naturally, since the columns can be checked without tracing the separate rows, and vice versa. When the program has to be sped up, probably a possibility can be found here.

The `Checksum...` arrays contain the total number of 1 bits in the corresponding row or column. Testing on essential columns and dominated rows or columns is based on these numbers and their differences. This holds also for the merge operator. Therefore it is kept up to date during the program.

A call to `Row` or `Column` needs an argument indicating the number of the row or column and will result in the `SupportList ReturnFromRow` or `ReturnFromColumn` to be filled with the positions of the 1 bits of the corresponding row or column, respectively. Those `SupportLists` are both primarily meant to pass the 1 values through the class itself. The value of `MaxCheckSumColumn` is used to determine the maximal size of the `SupportList` needed to contain the 1 bits of any column. This is important as a matrix can contain up to 10,000 rows, but not any column will cover nearly that many rows.

The positions of the 1 bits in `Essential` and `Dominated` represent the found essential and dominated columns respectively. The number of rows stored into the matrix is kept by `CurrentRow`.

The procedure `EssentialAdd` is used by both the public procedure `Merge`, based on the merge operator, and the private procedure `Heuristic`. It adds a column to the set of essentials. This call will be based on either essential columns in the matrix after reducing based on the combined on-set of the two genotypes used by the merge operator or on columns ranked best in the matrix (see also Section 4.2).

The procedures `EssentialCol`, `MakeRowEmpty` and `MakeColumnEmpty` are used in the publicly available `Reduce` procedure. The first one needs the number of a row as the argument. Actually, this row may only contain one 1 bit. The corresponding column will be found by `EssentialCol`. The column is added to the set of essentials, and the column and covered rows are made empty afterwards. This will result in calling the procedures `MakeColumnEmpty` and `MakeRowEmpty`.

The procedure `Heuristic` will select one column to become essential. Dependent on the probability `HeuristProb` this column will be chosen randomly or based on the highest importance factor of the involved columns (the definition of an importance factor has been given in Section 4.2). The chosen column and the covered rows are deleted, and the remainder matrix is reduced using the procedure `Reducing`. As long as the matrix contains non-empty columns, the procedure `Heuristic` will go on selecting columns and reducing the matrix. All collected essential columns are returned. These columns (input bits) are sufficient to cover the original matrix. Calling this procedure will affect the matrix!

The procedure `Reducing` accomplish the reduce task, without creating a new `Matrix` and without testing for dominated rows. This procedure will be called by `Reduce` (of course), and by `Heuristic`. This prevents the class from generating constantly new matrices when the merge or repair operator has been called. The two unsigned integers `NewRows` and `NewColumns` have been added to prevent the procedure `Reduce` from computing the size of the new `Matrix`, as these values are known within the `Reducing` procedure.
When the matrix is being created, *FillConstruct* fills the rows and the columns with just 0 bits. The other private procedures are included to make the constructor, destructor and assignment or copy operator execute properly. To make a *Matrix* construction has to be called to allocate the memory, like *DeleteAll* has to be called by the destructor, doing the opposite. When a *Matrix* has to be made based on an already existing one, *CopyFrom* is needed. This explains the need for the three public procedures too.

5.3 Classes *Chromosome* and *Phenotype*

In spite of what their functionality might look like, the implemental aspects of both the classes *Chromosome* and *Phenotype* are really very straightforward. The private part of *Chromosome* is shown in Figure 5.4.

```cpp
class Chromosome
{
private:
    static unsigned NumberOfAlleles;
    static SupportList ChromSupLi;
    friend class GeneticAlgorithm;
    BitStr Geno;
    void NoSupport( ) const;

    bool MutateNotJustOne( ) const;
    bool MutateRightSide( ) const;
    void Mutate0tol1( );
    void Mutatelto0( );
};
```

Figure 5.4 The implemental aspects of the class *Chromosome* (defined within *genetic.h*).

The main variable in this class is called *Geno*. It contains all the alleles. The number of alleles is stored in *NumberOfAlleles*. This is a static integer, because all chromosomes have the same size. The class *GeneticAlgorithm* has to put a value into this integer during its initialise procedure. That is the reason that class has been made friend.

The other private procedures have something to do with the mutation operator. The procedure *NoSupport* fills the static *SupportList* containing the positions of ZERO alleles. That will be useful for the *Mutate0tol1* procedure, because this procedure has to pick one ZERO allele at random, and changes it into a ONE allele. The opposite will be done by *Mutatelto0*. The *MutateNotJustOne* procedure returns a TRUE when not just a single allele has to be changed (and hence *MutateSome* has to be called). The procedure *MutateRightSide* will return a TRUE when an allele has to be changed which may lead to a better fitness (for a support, a ONE allele has to be changed, as for a non-support the opposite holds).
class Phenotype
{
private:
    float fitness;
    Chromosome Geno;
    void ComputeFit( );
};

Figure 5.5 The implemental aspects of the class Phenotype (defined within genetic.b).

The class Phenotype is a very simple one. The class contains the genotype and the fitness, as can be seen in Figure 5.5. The other thing is the procedure to compute the fitness actually, called ComputeFit. First it computes the number of rows in the Matrix not covered by the genotype. If this number equals zero, a cover has been found. On the other hand, when no cover has been found, it is checked whether the genotype has to be repaired or not, checking a random value against the repair probability (see also Chapter 4, Section 8). Also, when the number of uncovered rows added to the number of 1s in the genotype does not exceed the number of 1s in the best genotype found so far, the repair operator will be called. Afterwards, the fitness for the (maybe repaired) genotype is computed.

5.4 Classes PairOfChromosomes, Selection, PopElement and Population

The class Population needs the classes PairOfChromosomes and Selection to operate properly. The first one has been created for the crossover operator, the second one is made to execute the selection parts. Figure 5.6 shows the class PairOfChromosomes (defined in the files genetic.h and genetic.cxx).

class PairOfChromosomes
{
private:
    Chromosome XXY[ 2];
    void MakeCrossoverList( CrossoverList &XLi);
    bool RandomCross( );
public:
    PairOfChromosomes( const Chromosome &XX, const Chromosome &XY);
    void Crossover( const CrossoverList &MaXLi, const unsigned& n);
    Chromosome One( ) const;
    Chromosome Two( ) const;
};

Figure 5.6 The class PairOfChromosomes (defined within genetic.b).

The class contains two Chromosomes. These two are needed when the constructor is called and are stored in the private array.

The procedure crossover needs the CrossoverList and the number indicating whether uniform crossover (0) or multiple 2-point crossover (>0) has to be applied. It may sound strange that the list has to be added, but as it is useless to call the procedure with just 0 or 1 allele being different, this list has been made before deciding to use the class. It would be a shame to throw it away, and start making it again, then.
When uniform crossover has to be applied, for every position in the list RandomCross will be called. The returned Boolean indicates whether the allele at that position will swap between the two chromosomes. The probability of returning a TRUE depends on the default settings and the adaptation done in the class GeneticAlgorithm (see also Section 4.7).

The procedure MakeCrossoverList will be called to execute a 2-point crossover. The argument is a copy of the list and will be adjusted to indicate the positions that have to be swapped. The procedure selects a part from the list to be removed. Because a part has to be removed, one point will be chosen out of the list. The other point will be chosen from the total number of loci. Hence there will be a lower probability to select two different alleles with a big distance between them than to select two different ones close together.

The new chromosomes can be claimed using the procedures One and Two.

The class Selection is meant to store the selected Phenotypes and therefore has been derived from the class ClassList with the appropriate class (see also Figure 5.7). The selected members can be put into the standard output. The Remove procedure makes it possible to delete a number of selected Phenotypes, but has not been used in the program yet. The other usable procedures have been described for ClassList already. The class Selection can be found within the file genetic.h.

```
class Selection: public ClassList< Phenotype>
{
public:
    Selection( unsigned un): ClassList< Phenotype>( un);
    void Remove( unsigned first, unsigned last);
    friend ostream& operator<< (ostream &out, const Selection &Sel);
};
```

Figure 5.7 The class Selection (defined within genetic.h).

Having described the way crossover is treated and the selection will be stored, one more class has to be explained before a population can be made: PopElement (Figure 5.8). As can be read in the previous chapter, this class is mainly used to store a Phenotype and keeps track of the cumulative fitness. This explains the very simple private part.

```
class PopElement
{
private:
    float CumulativeFitness;
    Phenotype pheno;
};
```

Figure 5.8 The implemental aspects of the class PopElement (defined within genetic.h).

Now time has come to take a look at the class Population (Figure 5.9). This class uses the ClassList base with the PopElements.
class Population: public ClassList< PopElement>
{
private:
    float TopFitness, TotalFitness;
    unsigned Index;
    Phenotype BestSeen;
    SupportList MergeList, ResultList;
    CrossoverList MaXLi;

    void Initialise( );
    void Add( const PopElement PopEl);
    Phenotype Tourn();
    Phenotype Roulet();

    bool Merge();
    bool Mutation();
    bool Crossover();
};

Figure 5.9 The implemental aspects of the class Population (defined within genetic.h).

The variables used to store the best phenotype ever seen, the fitness of the best phenotype in the current population and the corresponding position are respectively BestSeen, TopFitness and Index. To keep the value of the cumulative fitness correct, TotalFitness is used.

Both SupportLists and the CrossoverList are included to ease the handling of arguments and to prevent the program from initialising these lists over and over again.

To make the initial generation Initialise will be called. It will assign the correct values to the variables and fill the population with semi-randomly filled chromosomes (see also Section 4.2). Because the code has to keep TotalFitness up to date, the procedure Add had to be rewritten. That was the reason the original procedure in ClassList had to be virtual.

The two procedures Tourn and Roulet both are part of the selection operator. The first one selects a phenotype from the population by the tournament method, the other one does the same by using the roulette wheel. The selected phenotypes are placed in the above-described Selection. The number of the elements equals the size of the population (minus the part reserved for the immigrate aspect - see also Chapter 4, Section 7). Dependent on the probability defined in ProbTourn a part of it is filled by the tournament selection, while the remainder is filled by the roulette selection.

The three procedures Merge, Mutation and Crossover each return a Boolean value indicating whether the corresponding operator has to be called. The probability of returning TRUE is defined in the default settings and can be adjusted in the class GeneticAlgorithm.

If the two selected phenotypes differ at no more than one locus when the Boolean indicates that crossover has to be applied, the crossover operator will not be called. The offspring would just not differ from the parents. When the two involved genotypes are twins, one will be mutated, else (one different allele) the phenotypes are just copied.
One thing to keep in mind is that the merge operator needs two elements, and only creates one new genotype. Hence, every time this operator is called, there will be one genotype that helps creating two new genotypes.

5.5 Classes OverallSize, InitialAspects, CoverAspects and GeneticAlgorithm

The class GeneticAlgorithm initialises some important classes. These classes (OverallSize, InitialAspects and CoverAspects) all have a static variable to be filled by the friend class GeneticAlgorithm. The classes are very straightforward.

class OverallSize // this class contains the size of the chromosomes
{
private:
    static unsigned Size;
    friend class GeneticAlgorithm;
};

Figure 5.10 The private part of class OverallSize (defined within genetic.h).

As can be seen in Figure 5.10 the class OverallSize contains only one unsigned variable indicating the size of the problem. This size equals the number of alleles within the genotypes or, in other words, the relevant (neither essential nor dominated) input bits in the matrix. This number is often required within the program.

class InitialAspects
{
private:
    static double *IniCol;
    static unsigned NumberOfCols;
    friend class GeneticAlgorithm;
public:
    static double Col( const unsigned Num);
};

Figure 5.11 The class InitialAspects (defined within inias.h).

The probabilities for each allele to become ONE in the initial population (see also Chapter 4, Sections 2 - importance factors - and 8) are kept in the class InitialAspects (Figure 5.11). An array will be made during the initialising of the genetic algorithm. The size is stored in NumberOfCols, and at every position of the array the probability to become ONE can be found for the allele with that locus. When the public procedure is called with a number exceeding the array size, a zero will be returned. The array will be deleted by the destructor of the GeneticAlgorithm.
class CoverAspects
{
private:
    static unsigned BestNumberAlleles;
    static Matrix Mtrix;
    friend class GeneticAlgorithm;
public:
    static unsigned TestThreshold( );
    static SupportList MergeOperator( const SupportList &SupLi);
    static SupportList Repair( const SupportList &SupLi);
    static unsigned Uncovered( const SupportList &SupLi);
};

Figure 5.12 The class CoverAspects (defined within covas.h).

Figure 5.12 shows the class CoverAspects. This class contains the matrix defining the problem and an unsigned integer indicating how many ONE alleles are contained in the best genotype found so far. The procedure TestThreshold returns this threshold value.

Another publicly available procedure is called MergeOperator and performs the merge operator indeed. The returning SupportList will cover the same set of rows as the argument did, but the number of columns has been reduced (when possible) by the heuristic algorithm outlined in Section 4.2.

On the opposite, the Repair procedure creates a SupportList that contains the SupportList given as the argument as a subset. This procedure executes the repair operator, as described in Chapter 4. Therefore, a call to this procedure will only be useful when the columns defined within the SupportList do not cover all rows of the Matrix.

The Uncovered procedure computes the number of rows not covered by the SupportList given as an argument. Any number between 0 and 3 equals the exact number of uncovered rows, where a 0 means the SupportList defines an input support. When 4 uncovered rows have been found, the number is extrapolated (using the total number of rows to be covered and the number of rows already checked). The returned value is essential for the computation of the fitness function.

All procedures MergeOperator, Repair and Uncovered execute just a call to the corresponding public procedures of Matrix. This class can be seen as an interface between the genetic algorithm and the matrix.

The class GeneticAlgorithm (Figure 5.13) is just meant to make everything cooperate. It can adapt probabilities, it takes care of initialising the classes (as described already in this section), it creates a population and it lets it run. It can be seen as the world where the population lives, bringing environmental aspects in.
class GeneticAlgorithm
{
private:
    Population Popu;
    unsigned TwoXPoint;
    void DefineSize( const unsigned &i);
    void InitialiseParts( const Matrix &Matr);
public:
    ~GeneticAlgorithm();
};

Figure 5.13 The implemental aspects of class GeneticAlgorithm (defined within ga.h).

The class contains a population and the crossover operator modifier (if TwoXPoint equals 0, uniform crossover will be applied, else the integer decides how many times 2-point crossover will be executed).

The procedure InitialiseParts will be called by the constructor. All statics will be properly filled and the random generator (see next section) will be initialised. The procedure will use DefineSize to assign the correct value to the private variable of OverallSize.

The destructor has been added in order to deallocate the memory claimed by InitialAspects (see above).

5.6 The random generator

Genetic algorithms do often need randomly chosen numbers. Other researchers have also looked for good generators, witness the little thread on the newsgroup comp.ai.genetic on this topic starting at the end of February 1996. I found the code for the algorithm by Fuat C. Baran at this place. As the code has been written in C and it had a number of recommendations, the code has been used in our project.

void frandinit(int s);
static float frandom();

unsigned Random( const unsigned &Max);
float Random( const float &Max = 1.0);

bool RandomAllele( const unsigned &Position);
bool RandomTresh( const float Tre = 0.5);

Figure 5.14 The stochastic functions specified within frandom.h and frandom.hxx.

\footnote{Based on code in "Random Number Generators: Good Ones are Hard to Find", by Stephen K. Park and Keith W. Miller in Communications of the ACM, 31, 10 (Oct. 1988) pp. 1192-1201.}
The stochastic functions can be found in Figure 5.14. Before the random generator can be used, the function has to be initialised. This has to be done by calling frandinit with an integer. The constructor of *GeneticAlgorithm* will take care of this. Thereafter, the stochastic function frandom will return a randomly number between 0 (inclusive) and 1 (exclusive). These two functions have been found on the newsgroup.

The other functions will use this function as the base. When a random integer is needed, Random can be called. The returned value will be at least 0 and at most the argument minus one. A similar function can be called when floating points are needed, also called Random. Now the argument has to be a float too, indicating the ceiling for the possible values to return. When no argument has been given, the function is exactly the same as the base function frandom.

The stochastic function RandomAllele needs a locus of an allele and will return a Boolean indicating whether the allele should become ONE (true) or ZERO (false). This function relies on the class *InitialAspects*. See for further details the previous section.

When in the program there has to be made a decision based on a probability distribution, RandomTresh can be called. The chance to make the decision equals the argument (when no argument is given, the default setting of fifty percent will be used). The returning Boolean indicates whether to do it (true) or not (false).

5.7 The display functions

Many classes can be displayed at the standard output. This can be seen at several figures in Chapter 4, where the publicly available parts were discussed. To simplify maintenance and extensions, the code has been put together in one file, called *gaout.cxx*. There are eight different classes that can be displayed using the outstream operator, see Figure 5.15. To describe the implementation would be getting too far into detail.

```
ostream& operator << (ostream &out, const CrossoverList &XList);
ostream& operator << (ostream &out, const SupportList &SList);
ostream& operator << (ostream &out, const Chromosome &Chrom);
ostream& operator << (ostream &out, const Selection &Sel);
ostream& operator << (ostream &out, const Phenotype &Phe);
ostream& operator << (ostream &out, const PopElement &PEl);
ostream& operator << (ostream &out, const Population &Popu);
ostream& operator << (ostream &out, const Matrix &Mat);
```

Figure 5.15 The specified outstream operators within *gaout.cxx*.

5.8 Making the executable

Now all classes have been described, time has come to make them cooperate. In order to create an executable, two parts have to be added:

- a file with the code for the main function;
- a Makefile, containing the compile commands.
Every aspect to test needs a specific code for the main function. However, many parts can stay the same. For example, the check whether a file can be read or not has to be performed every time the program is called. The standard part of the code has been listed in Appendix D. The specific code of every test has to be placed inside the block starting with the comment line '//SPECIFIC CODE GOES HERE'.

In order to run the genetic algorithm on a file (the only way the code can be used now) the filename has to be given as an argument. In fact, as many files as needed can be minimized in one single run, as the function will perform the genetic algorithm on every given argument.

In the beginning the code has been slightly different. The (re)initilizer for the random number generator has been changed in order to get more different results. The processing times are written to a file now (in the code TIMEFILE has to be replaced by the appropriate name and USER has to be changed into the login name of the user).

The Makefile has been listed in Appendix D too. The location of this file has been in the parent directory of the *.cxx files listed in the src directory, the *.h files listed in the include directory, the *.o files listed in the obj directory, and the executable file itself listed in the bin directory. The name of the executable can be chosen by changing EXENAME into the desired filename.

Of course, everything compiles without errors when the make command is executed.
6. Testing the algorithm

6.1 Introduction

The genetic algorithm as described in the previous two chapters is a stochastic process with many parameters that serve for adaptation of the probabilities of operators and other computation parts. Therefore, a simple test run of the program that implements the algorithm is not sufficient to judge the power of the algorithm. Testing the program has the three following goals:

• validation of the program by checking it for programming faults;
• adjustment of the probabilities to optimize the performance;
• comparison of the algorithm's performance to that of QuickScan (see Appendix B).

The genetic algorithm contains more than 8 probability parameters. To optimize their default values, a test run has to be set in an 8 dimensional space, with every single run at least performed twice, as it is a stochastic process. This would be very time consuming. Therefore, we decided to search for a local optimum in a single dimension, and to take that optimum as a starting point for another dimension. In that way we can find good settings for the algorithm in a reasonable time. Of course, we could have developed a meta genetic algorithm just to find the (near-)optimal settings.

There are three different types of parameters:

• frequency of operators;
• behaviour of operators;
• additional aspects.

The frequency of an operator is defined by the probability whether or not that operator will be used in order to create offspring. Four operators can be used for this purpose directly: crossover, mutation, merge and copy (which will be used when none of the others has been chosen). The fifth one, the repair operator, can be applied in conjunction with any of the previously mentioned operators, but only if the genotype can be repaired (i.e. the genotype does not represent a solution to the problem).

The behaviour of an operator has not completely been fixed. The crossover operator can produce a uniform or an \textit{n times 2-point} crossover. The mutation operator can mutate just a single allele or a complete set. The merge operator can select the alleles from the set of combined alleles at random or using heuristics. The same holds for the repair operator.

The additional aspects contains first of all the size of the population. The immigrate aspect will be included into this group too, although it could be seen as an operator with a very rigid mutation behaviour. Also the duration of the process, or in other words the number of generations, can be seen as part of this group. Even the choice of the initialiser of the random number generator could be inside of this category.

First, we will try to find for every operator the best application frequency in relation to its behaviour. We will start with the crossover operator, then take the mutation operator, the merge operator and finish with the repair operator. Thereafter we will look at the immigrate aspect and take a brief look at the size of the population.
After we have found the settings this way, the frequency and behaviour of the crossover and mutation operators will be tested once again. Some aspects of a more dynamic approach of the settings will be examined and we will end this chapter with some thoughts on additional research.

However, before examining the operators and their influence on the total algorithm, we will investigate the merge operator a little bit closer by testing it on the total input domain without using the genetic aspects of the genetic algorithm. When the merge operator has some potential, a useful solution must be discovered by making a reduced cover matrix, and thereafter the merge operator must be called with just 1 alleles (this is the smallest support set that for sure contains the minimal input support solution).

6.2 The power of the heuristic merge operator

As can be read throughout Chapters 4 and 5, the genetic algorithm has been implemented by creation of a flexible program. Many parameters can be changed in order to modify the behaviour of the algorithm. It can even be adapted to turn off the genetic aspects. That is exactly what we need to test the power of the merge operator on its own.

The initial generation has to contain just two members (as the merge operator is a sexual operator), both equalling just 1 alleles. Therefore, the minimal probability for an allele to become 1 has to be exactly 1 to ensure this. The probabilities of all the operators have to equal 0, except for the merge operator; it has to be called for sure. Furthermore, the merge operator has to be executed using the pure heuristic algorithm as described in Chapter 4. Only one next generation is needed to test the result. The specific code for this test can be found in Appendix E.
The results can be found in Table 6.1, and as can be seen, they are a little bit better than QuickScan's results. The solution found for the problem defined in i100.pla is even the best ever found. On the other hand, the execution time is high. The time needed for this algorithm to create only the reduced matrix (thus to preprocess) extends often the total time needed by QuickScan to terminate (which includes the preprocess module).

The problems defined in i40.pla and i40s10a.pla lead to matrices with over 10,000 rows. Checking for dominated rows can result in comparing 50,000,000 times two rows with each other. This can take an enormous amount of time. On the other hand, it would be a waste to check every dominated row for every created genotype in order to decide whether it contains an input support or not. Therefore, we have decided to change the algorithm to check only once on dominated rows during the Matrix:::Reduce before creating a new reduced Matrix, and to check not at all on dominated rows during the heuristic merge and repair operators. The new test resulted in the same input support qualities, but remarkable shorter execution times, as can be found in Table 6.2.
Table 6.2 Comparing the new and the old algorithm's execution times.

<table>
<thead>
<tr>
<th>File</th>
<th>T(new)</th>
<th>T(old)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>58</td>
<td>1:43</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>19</td>
<td>43</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>19</td>
<td>44</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>1:47</td>
<td>4:01</td>
</tr>
<tr>
<td>i40.pla</td>
<td>1:47</td>
<td>4:02</td>
</tr>
<tr>
<td>i50.pla</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>i100.pla</td>
<td>1:02</td>
<td>1:12</td>
</tr>
</tbody>
</table>

T in seconds.

As we have seen, the merge operator is a powerful operator indeed. As a matter of fact the operator on its own combines an acceptable execution time with good solutions.

6.3 The genetic algorithm using only the crossover operator

The most important operator of a standard genetic algorithm is the crossover operator. It serves as the main exploration and exploitation operator. Therefore we have chosen to use only the crossover operator in the first genetic test; the probabilities for all other operators will be set to zero.

Two settings will have their influences on the final result: the probability whether the crossover operator will be applied or not (else the selected genotype will be just copied) and whether uniform crossover will be applied or (n times) 2-point crossover. To try to find the best settings for the operator, we have explored this two dimensional space. The probability has been changed from 0.1 to 1.0 in 7 equal steps and for all these settings uniform and 1 and 2 times 2-point crossover have been applied.

As long as the other settings do not affect operator probabilities, they have been assigned the default value, what means:

- the roulette wheel method will be used 40% of the time (RouletteProb equals 0.4);
- when the repair operator is called (only when the repaired phenotype will for sure have a higher fitness than the best found yet) new columns are added at random instead of using importance factors (HeuristProb equals 0.0);
- the probability for each allele in the first population to be set to 1 is in the range [0.3; 0.8] (LowerAlleleProb equals 0.3 and UpperAlleleProb equals 0.8);
- no immigrate aspect is used (SizeOfNews equals 0);
- the size of the population is chosen to be 2 times the number of alleles in each genotype (RelSizeProb equals 2.0);
• when uniform crossover will be applied half of the different alleles will change genotype (ProbAlleleUniformCross equals 0.5).

The code for the algorithm can be found in Appendix F. The results can be found in Table 6.3 - 6.5.

**Table 6.3 Results obtained using the genetic algorithm with uniform crossover.**

<table>
<thead>
<tr>
<th>File</th>
<th>Q(0.1)</th>
<th>Q(0.25)</th>
<th>Q(0.4)</th>
<th>Q(0.55)</th>
<th>Q(0.7)</th>
<th>Q(0.85)</th>
<th>Q(1.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7.5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.5</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5.5</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>7</td>
<td>6</td>
<td>6.5</td>
<td>6.5</td>
<td>7</td>
<td>6.5</td>
<td>6</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5.5</td>
<td>5.5</td>
<td>5.5</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
<td>9.5</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>9.5</td>
<td>9.5</td>
<td>9.5</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9.5</td>
<td>9.5</td>
<td>9.5</td>
<td>10</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11.5</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>12</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i40.pla</td>
<td>12</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8.5</td>
<td>8.5</td>
</tr>
<tr>
<td>i100.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Q in number of input bits (average over 2 runs), in brackets the probability of the crossover operator.
Table 6.4 Results obtained using the genetic algorithm with 2-point crossover.

<table>
<thead>
<tr>
<th>File</th>
<th>Q(0.1)</th>
<th>Q(0.25)</th>
<th>Q(0.4)</th>
<th>Q(0.55)</th>
<th>Q(0.7)</th>
<th>Q(0.85)</th>
<th>Q(1.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4.5</td>
<td>4</td>
<td>4</td>
<td>4.5</td>
<td>4</td>
<td>4</td>
<td>4.5</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>7</td>
<td>6.5</td>
<td>6</td>
<td>6.5</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>6.5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9</td>
<td>10</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9.5</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>10</td>
<td>9.5</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11.5</td>
<td>11.5</td>
<td>12</td>
<td>11</td>
<td>12</td>
<td>11</td>
<td>11.5</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>10.5</td>
<td>9.5</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10.5</td>
<td>9.5</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11.5</td>
<td>12</td>
<td>11.5</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i40.pla</td>
<td>11.5</td>
<td>12</td>
<td>11.5</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i50.pla</td>
<td>9</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8.5</td>
</tr>
<tr>
<td>i100.pla</td>
<td>12</td>
<td>10.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10.5</td>
<td>10.5</td>
</tr>
</tbody>
</table>

Q in number of input bits (average over 2 runs), in brackets the probability of the crossover operator.
Table 6.5 Results obtained using the genetic algorithm with 2 times 2-point crossover.

<table>
<thead>
<tr>
<th>File</th>
<th>Q(0.1)</th>
<th>Q(0.25)</th>
<th>Q(0.4)</th>
<th>Q(0.55)</th>
<th>Q(0.7)</th>
<th>Q(0.85)</th>
<th>Q(1.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4</td>
<td>4</td>
<td>4.5</td>
<td>4</td>
<td>4.5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>7</td>
<td>7</td>
<td>6.5</td>
<td>7</td>
<td>6.5</td>
<td>6</td>
<td>6.5</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
<td>5.5</td>
<td>5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9.5</td>
<td>9.5</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
<td>8.5</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9.5</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
<td>9.5</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>12</td>
<td>11.5</td>
<td>11.5</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11.5</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10.5</td>
<td>10</td>
<td>10.5</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10.5</td>
<td>10</td>
<td>10.5</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11.5</td>
<td>11.5</td>
<td>11.5</td>
<td>11</td>
<td>11.5</td>
<td>11.5</td>
<td>11.5</td>
</tr>
<tr>
<td>i50.pla</td>
<td>11.5</td>
<td>11.5</td>
<td>11.5</td>
<td>11</td>
<td>11.5</td>
<td>11.5</td>
<td>11.5</td>
</tr>
<tr>
<td>i100.pla</td>
<td>11.5</td>
<td>11</td>
<td>10.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Q in number of input bits (average over 2 runs), in brackets the probability of the crossover operator.

It is clear from these results that the differences between the behaviour settings of the crossover operator are very small. On the other hand, the choice of the frequency is an important one; especially for the large problem instances the solutions found tend to be better when crossover is more often called.

A new test run have been performed on the frequency spectrum in the range [0.5; 0.9]. The results can be found in Table 6.6 - 6.8.
Table 6.6 Results obtained using uniform crossover in the frequency range \([0.5; 0.9]\).

<table>
<thead>
<tr>
<th>File</th>
<th>Q(0.5)</th>
<th>Q(0.6)</th>
<th>Q(0.65)</th>
<th>Q(0.75)</th>
<th>Q(0.8)</th>
<th>Q(0.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4.5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
<td>5.5</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6</td>
<td>6.5</td>
<td>7</td>
<td>6.5</td>
<td>6</td>
<td>6.5</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5.5</td>
<td>5</td>
<td>5.5</td>
<td>5.5</td>
<td>5</td>
<td>5.5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9.5</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>9.5</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>9</td>
<td>8.5</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9</td>
<td>10</td>
<td>9.5</td>
<td>9.5</td>
<td>9.5</td>
<td>9.5</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11</td>
<td>12</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10.5</td>
<td>10</td>
<td>10.5</td>
<td>10</td>
<td>9.5</td>
<td>10</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>9.5</td>
<td>10.5</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i40.pla</td>
<td>12</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i50.pla</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i100.pla</td>
<td>11</td>
<td>10.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Q in number of input bits (average over 2 runs), in brackets the probability of the crossover operator.
Table 6.7 Results obtained using 2-point crossover in the frequency range [0.5; 0.9].

<table>
<thead>
<tr>
<th>File</th>
<th>Q(0.5)</th>
<th>Q(0.6)</th>
<th>Q(0.65)</th>
<th>Q(0.75)</th>
<th>Q(0.8)</th>
<th>Q(0.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4</td>
<td>4.5</td>
<td>4.5</td>
<td>4.5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5.5</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6.5</td>
<td>6.5</td>
<td>6</td>
<td>6</td>
<td>6.5</td>
<td>6</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>10</td>
<td>9</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>12</td>
<td>11.5</td>
<td>12</td>
<td>11.5</td>
<td>11.5</td>
<td>11.5</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>i40.pla</td>
<td>12</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8.5</td>
<td>9</td>
<td>9</td>
<td>8.5</td>
<td>8.5</td>
<td>8.5</td>
</tr>
<tr>
<td>i100.pla</td>
<td>10.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Q in number of input bits (average over 2 runs), in brackets the probability of the crossover operator.
Table 6.8 Results obtained using 2·2-point crossover in the frequency range [0.5; 0.9].

<table>
<thead>
<tr>
<th>File</th>
<th>Q(0.5)</th>
<th>Q(0.6)</th>
<th>Q(0.65)</th>
<th>Q(0.75)</th>
<th>Q(0.8)</th>
<th>Q(0.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>5</td>
<td>4.5</td>
<td>4.5</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9</td>
<td>9.5</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9.5</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11.5</td>
<td>11</td>
<td>12</td>
<td>11.5</td>
<td>11</td>
<td>11.5</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
</tr>
<tr>
<td>i40.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
</tr>
<tr>
<td>i50.pla</td>
<td>9</td>
<td>8.5</td>
<td>8.5</td>
<td>8.5</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i100.pla</td>
<td>11</td>
<td>10.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Q in number of input bits (average over 2 runs), in brackets the probability of the crossover operator.

The results obtained from the new test are very comparable to the results shown in Tables 6.3 - 6.5. To delve somewhat deeper into the potential of the uniform crossover operator, another test has been performed in the frequency range from 0.55 to 0.9.

Three different settings have been applied for the probability of an allele of offspring1 (offspring2) to be defined by the value of the corresponding allele of parent1 (parent2). Of course, when that parent is not chosen to be responsible for the value, the allele of the other parent will be used. These so called swap chance has been set to the values 0.2, 0.3 and 0.4 (0.5 has already been tested on - see Table 6.6). The results can be found in Table 6.9, 6.10 and 6.11, respectively.
Table 6.9 Uniform crossover with swap chance 0.2 in the frequency range [0.55; 0.9].

<table>
<thead>
<tr>
<th>File</th>
<th>Q(0.55)</th>
<th>Q(0.6)</th>
<th>Q(0.65)</th>
<th>Q(0.7)</th>
<th>Q(0.75)</th>
<th>Q(0.8)</th>
<th>Q(0.85)</th>
<th>Q(0.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4</td>
<td>4.5</td>
<td>4.5</td>
<td>4.5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6.5</td>
<td>6</td>
<td>6.5</td>
<td>6</td>
<td>6.5</td>
<td>7</td>
<td>6.5</td>
<td>6.5</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
<td>5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9.5</td>
<td>9.5</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>10</td>
<td>8.5</td>
<td>9.5</td>
<td>9.5</td>
<td>9</td>
<td>10</td>
<td>9.5</td>
<td>9</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>12</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11.5</td>
<td>12</td>
<td>12</td>
<td>11.5</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>10.5</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10.5</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10.5</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
</tr>
<tr>
<td>i40.pla</td>
<td>11.5</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i50.pla</td>
<td>9</td>
<td>8.5</td>
<td>9</td>
<td>8.5</td>
<td>9</td>
<td>8.5</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i100.pla</td>
<td>10.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Q in number of input bits (average over 2 runs), in brackets the probability of the crossover operator.
Table 6.10 Uniform crossover with swap chance 0.3 in the frequency range [0.55; 0.9].

<table>
<thead>
<tr>
<th>File</th>
<th>Q(0.55)</th>
<th>Q(0.6)</th>
<th>Q(0.65)</th>
<th>Q(0.7)</th>
<th>Q(0.75)</th>
<th>Q(0.8)</th>
<th>Q(0.85)</th>
<th>Q(0.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>il0.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4</td>
<td>4.5</td>
<td>4</td>
<td>4.5</td>
<td>4</td>
<td>4</td>
<td>4.5</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
<td>5.5</td>
<td>5.5</td>
<td>5.5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>7</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>6</td>
<td>6.5</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5.5</td>
<td>5</td>
<td>5.5</td>
<td>5.5</td>
<td>5</td>
<td>5.5</td>
<td>5.5</td>
<td>5.5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9.5</td>
<td>9.5</td>
<td>9.5</td>
<td>9.5</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
<td>9.5</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11.5</td>
<td>11.5</td>
<td>11.5</td>
<td>11.5</td>
<td>12</td>
<td>11.5</td>
<td>11.5</td>
<td>11.5</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10.5</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10.5</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
<td>12</td>
</tr>
<tr>
<td>i40.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i50.pla</td>
<td>9</td>
<td>8.5</td>
<td>9</td>
<td>8.5</td>
<td>8.5</td>
<td>8.5</td>
<td>8.5</td>
<td>8.5</td>
</tr>
<tr>
<td>i100.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10.5</td>
<td>11</td>
<td>11</td>
<td>10.5</td>
</tr>
</tbody>
</table>

Q in number of input bits (average over 2 runs), in brackets the probability of the crossover operator.
Table 6.11 Uniform crossover with swap chance 0.4 in the frequency range [0.55; 0.9].

<table>
<thead>
<tr>
<th>File</th>
<th>Q(0.55)</th>
<th>Q(0.6)</th>
<th>Q(0.65)</th>
<th>Q(0.7)</th>
<th>Q(0.75)</th>
<th>Q(0.8)</th>
<th>Q(0.85)</th>
<th>Q(0.9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4.5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5.5</td>
<td>5</td>
<td>5.5</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>7</td>
<td>6.5</td>
<td>6.5</td>
<td>6</td>
<td>6.5</td>
<td>7</td>
<td>6.5</td>
<td>6</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
<td>5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9</td>
<td>9.5</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>9</td>
<td>8.5</td>
<td>9</td>
<td>9.5</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11.5</td>
<td>11.5</td>
<td>11.5</td>
<td>11.5</td>
<td>12</td>
<td>12</td>
<td>11.5</td>
<td>11</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>10</td>
<td>9.5</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11</td>
<td>11.5</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i40.pla</td>
<td>11.5</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i50.pla</td>
<td>9</td>
<td>9</td>
<td>8.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8.5</td>
<td>9</td>
</tr>
<tr>
<td>i100.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Q in number of input bits (average over 2 runs), in brackets the probability of the crossover operator.

Still, there are minor differences between the settings. Settings in the frequency range [0.7, 0.9] seem to be slightly preferable over settings in the range [0.0, 0.65]. Because other operators have to be included too (and the genetic algorithm has been build using one operator per offspring\(^{14}\)), it would not desirable to choose a value bigger than 0.85 for the probability.

We have fine tuned that setting in the range [0.725, 0.8], as can be found in Table 6.12 and 6.13. Six different types of crossover have been applied (uniform crossover with a swap chance of 0.2, 0.3, 0.4, and 0.5 respectively, as well as 1 time and 2 times 2-point crossover).

---

\(^{14}\)With the exception of the repair operator.
Table 6.12 Testing the crossover operator in the frequency range [0.725; 0.75].

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>0.725</th>
<th></th>
<th></th>
<th></th>
<th>0.75</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.5</td>
<td>1</td>
<td>2</td>
<td>0.2</td>
<td>0.3</td>
</tr>
<tr>
<td>i10.pla</td>
<td></td>
<td>7</td>
<td>7.5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td></td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td></td>
<td>4.5</td>
<td>4</td>
<td>4.5</td>
<td>4.5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.5</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td></td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6.5</td>
<td>6</td>
<td>6.5</td>
<td>7</td>
</tr>
<tr>
<td>kaz.pla</td>
<td></td>
<td>5.5</td>
<td>5.5</td>
<td>5.5</td>
<td>5.5</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5.5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td></td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9.5</td>
<td>10</td>
<td>9.5</td>
<td>9</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td></td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td></td>
<td>9</td>
<td>9.5</td>
<td>9.5</td>
<td>9.5</td>
<td>8.5</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td></td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
<td>11.5</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td></td>
<td>11</td>
<td>11</td>
<td>11.5</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
</tr>
<tr>
<td>i40.pla</td>
<td></td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11.5</td>
</tr>
<tr>
<td>i50.pla</td>
<td></td>
<td>8.5</td>
<td>9</td>
<td>8.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8.5</td>
<td>9</td>
</tr>
<tr>
<td>i100.pla</td>
<td></td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10.5</td>
<td>10.5</td>
</tr>
</tbody>
</table>

Type defines crossover behaviour; values $x < 1$ define a uniform crossover operator with swap chance equal to $x$; integer values $n$ define an $n$ times 2-point crossover. $Q$ in number of input bits (average over 2 runs).
Table 6.13 Testing the crossover operator in the frequency range \([0.775; 0.8]\).

<table>
<thead>
<tr>
<th>File</th>
<th>i10.pla</th>
<th>i20s04a.pla</th>
<th>i20s05a.pla</th>
<th>i20s06a.pla</th>
<th>kaz.pla</th>
<th>i30s08a.pla</th>
<th>i30s08b.pla</th>
<th>i30s08c.pla</th>
<th>i30s10a.pla</th>
<th>i40s09a.pla</th>
<th>i40s09b.pla</th>
<th>i40s10a.pla</th>
<th>i40.pla</th>
<th>i50.pla</th>
<th>i100.pla</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>0.775</td>
<td>0.8</td>
<td>0.775</td>
<td>0.8</td>
<td>0.775</td>
<td>0.8</td>
<td>0.775</td>
<td>0.8</td>
<td>0.775</td>
<td>0.8</td>
<td>0.775</td>
<td>0.8</td>
<td>0.775</td>
<td>0.8</td>
<td>0.775</td>
</tr>
<tr>
<td>Type</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.5</td>
<td>1</td>
<td>2</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.5</td>
<td>1</td>
<td>2</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7.5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>4.5</td>
<td>4</td>
<td>4.5</td>
<td>4</td>
<td>4.5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>6.5</td>
<td>6</td>
<td>6.5</td>
<td>6</td>
<td>6.5</td>
<td>6.5</td>
<td>6</td>
<td>6.5</td>
<td>6.5</td>
<td>6</td>
<td>6.5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>9.5</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>12</td>
<td>11.5</td>
<td>11</td>
<td>11.5</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9.5</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9.5</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>8.5</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8.5</td>
<td>8.5</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

Type defines crossover behaviour; values \(x < 1\) define a uniform crossover operator with swap chance equal to \(x\); integer values \(n\) define an \(n\) times 2-point crossover. Q in number of input bits (average over 2 runs).

It can be seen from these tables that a frequency of 0.8 is a little bit worse than the other settings, which are very similar to each other. When the results at that frequency found in Table 6.13 are compared with the results found in Table 6.6 - 6.8, the problem of testing settings for stochastic processes is showed; although the settings are identical, the results are not. At the other hand, the results at frequency 0.75 seem to be more reliable, witness Table 6.12 and Table 6.6 - 6.8.

Therefore, the crossover probability had been fixed at 0.75. Just one more test had to be performed to get a good behaviour setting. In order to get better results, the tests had been performed on five successive runs. The average solutions created by the genetic algorithm are put together in Table 6.14.
Table 6.14 A last test on the crossover operator at the fixed frequency 0.75.

<table>
<thead>
<tr>
<th>File</th>
<th>Type 0.15</th>
<th>0.2</th>
<th>0.25</th>
<th>0.3</th>
<th>0.35</th>
<th>0.4</th>
<th>0.45</th>
<th>0.5</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4.4</td>
<td>4.4</td>
<td>4.2</td>
<td>4.2</td>
<td>4.2</td>
<td>4.2</td>
<td>4.4</td>
<td>4.8</td>
<td>4.2</td>
<td></td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5.4</td>
<td>5.4</td>
<td>5.2</td>
<td>5.2</td>
<td>5</td>
<td>5.2</td>
<td>5.4</td>
<td>5.4</td>
<td>5.2</td>
<td></td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6.2</td>
<td>6.2</td>
<td>6.6</td>
<td>6.4</td>
<td>6.4</td>
<td>6.2</td>
<td>6.4</td>
<td>6.6</td>
<td>6.2</td>
<td></td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5.2</td>
<td>5.6</td>
<td>5.6</td>
<td>5.4</td>
<td>5</td>
<td>5.2</td>
<td>5.6</td>
<td>5.2</td>
<td>5</td>
<td>5.2</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9</td>
<td>9</td>
<td>9.2</td>
<td>9.2</td>
<td>9.4</td>
<td>9.2</td>
<td>9.2</td>
<td>9.4</td>
<td>8.8</td>
<td></td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>9.6</td>
<td>9</td>
<td>9.2</td>
<td>9.2</td>
<td>9</td>
<td>8.6</td>
<td>9.4</td>
<td>9</td>
<td>9.4</td>
<td>9.2</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9</td>
<td>9.4</td>
<td>9.4</td>
<td>9</td>
<td>9.4</td>
<td>9.4</td>
<td>9.2</td>
<td>9.6</td>
<td>9.4</td>
<td>9.2</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11.4</td>
<td>11.6</td>
<td>11.2</td>
<td>11.6</td>
<td>11.6</td>
<td>11.4</td>
<td>11.4</td>
<td>11.4</td>
<td>11.4</td>
<td>11.4</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>9.8</td>
<td>10</td>
<td>9.8</td>
<td>10</td>
<td>10.2</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9.8</td>
<td>10</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11</td>
<td>11.2</td>
<td>11.2</td>
<td>11.4</td>
<td>11.4</td>
<td>11.2</td>
<td>11.2</td>
<td>11.2</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i50.pla</td>
<td>11</td>
<td>11.2</td>
<td>11.6</td>
<td>11</td>
<td>11.2</td>
<td>11.4</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i50.pla</td>
<td>9</td>
<td>8.8</td>
<td>8.8</td>
<td>9</td>
<td>9</td>
<td>8.8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8.8</td>
</tr>
<tr>
<td>i100.pla</td>
<td>11</td>
<td>10.8</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10.8</td>
<td></td>
</tr>
</tbody>
</table>

Type defines crossover behaviour; values $x < 1$ define a uniform crossover operator with swap chance equal to $x$; integer values $n$ define an $n$ times 2-point crossover. $Q$ in number of input bits (average over 5 runs).

The best behaviour setting for the crossover operator with a frequency of 0.75 found in the table is the 2 times 2-point crossover. Even when we take the results earlier found into account (see Table 6.6 - 6.12), this setting is preferred to the other ones.

Now the settings for the first operator have been found, the other operators have to be included into the genetic algorithm.

6.4 Creating a standard genetic algorithm by adding the mutation operator

A standard genetic algorithm consists of the crossover operator, the mutation operator and the copy operator to create offspring. So, when the setting for the mutation operator has been optimized, the results found so far are an indication for the potential of a genetic algorithm for the Minimal Input Support Problem.

The mutation operator contains three aspects:
- the probability it will be applied (frequency);
- the probability not just one allele will be affected (behaviour)$^{15}$;

$^{15}$We will call this the multi-allele aspect.
the probability the allele(s) will change in a way that the offspring has a chance to become better (measured by fitness) than the original (behaviour). The simple translation of the last aspect is as follows: when the original phenotype had been a solution to MISP, create offspring using no more 1-alleles than the original had, and when the original phenotype did not happen to be a solution, the offspring should contain no more 0-alleles than the original.

To get a global impression of the best probability regions of the first two aspects, a test has been performed using the frequencies \{0.00125; 0.0125; 0.0625; 0.125\} and three settings for the multi-allele probability \{0.0; 0.5; 1.0\}. The setting of the probability of the right-side aspect was set (by default) on 0.9. All twelve combinations have been made, and for every setting three runs have been performed. Appendix G contains the code of the program. The results can be found in Table 6.15.

<table>
<thead>
<tr>
<th>Frequency 'Multi'</th>
<th>0.00125</th>
<th>0.0125</th>
<th>0.0625</th>
<th>0.125</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 0.5 1</td>
<td>0 0.5 1</td>
<td>0 0.5 1</td>
<td>0 0.5 1</td>
</tr>
<tr>
<td>i10.pla</td>
<td>7 7 7</td>
<td>7 7 7</td>
<td>7 7 7</td>
<td>7 7 7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4.3 4 4</td>
<td>4.3 4 4</td>
<td>4 4 4</td>
<td>4 4 4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5 5.3 5.3</td>
<td>5 5.7 5</td>
<td>5 5 5.3</td>
<td>5 5 5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6.7 6.3 6.3</td>
<td>6.3 6.3 6.7</td>
<td>6.3 6.3 6.3</td>
<td>6.3 6.3 6.7</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5 5 5</td>
<td>5.3 5 5.7</td>
<td>5.3 5 5.3</td>
<td>5 5 5.3</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9 9.3 9</td>
<td>9.3 9.7 9.3</td>
<td>9.3 9.3 9</td>
<td>9 9 9</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>9 9.3 9.3</td>
<td>9.3 8.7 9.3</td>
<td>9.3 9.3 9</td>
<td>9.7 9 9</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9.3 9 9.3</td>
<td>9.7 9.3 9</td>
<td>9 9 9</td>
<td>9.3 9 9</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11.7 11.3 11</td>
<td>11.3 11.3 12</td>
<td>11.7 11 11</td>
<td>11 11 11</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10 10 10</td>
<td>10 10 10</td>
<td>10 10 10</td>
<td>10 10 10</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10 10 10</td>
<td>10 10 10</td>
<td>10 10 10</td>
<td>10 10 10</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11.3 11 11.3</td>
<td>11 11.3 11.3</td>
<td>11 11 11</td>
<td>11 11 11</td>
</tr>
<tr>
<td>i40.pla</td>
<td>11 11.7 11.3</td>
<td>11 10.7 11.3</td>
<td>11 11 11</td>
<td>11.7 11 11</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8.7 8.7 8.7</td>
<td>9 9 8.7</td>
<td>9 9 8.3</td>
<td>8.7 8.7 9</td>
</tr>
<tr>
<td>i100.pla</td>
<td>11 11 10.7</td>
<td>11 11 10.7</td>
<td>11 11 11</td>
<td>11 11 10.7</td>
</tr>
</tbody>
</table>

'Multi' defines the 'multi-allele' behaviour. Q in number of input bits (average over 3 runs).

Based on these results the area around frequency 0.0625 and multi-allele setting 1.0 has been explored in more detail. The frequencies 0.05, 0.0625 and 0.0825 were used combined with multi-allele probabilities 0.8, 0.9 and 1.0. Now the right-side aspect has been also taken into account. Those settings varied from 0.7 to 0.9 in steps of 0.1. The results can be found in Table 6.16 - 6.18.

\[16\text{We will call this the right-side aspect.}\]
Table 6.16 Mutation operator test for 'multi-allele' probability 0.8.

<table>
<thead>
<tr>
<th>File</th>
<th>0.05</th>
<th>0.0625</th>
<th>0.0825</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
</tr>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6.3</td>
<td>6.7</td>
<td>6.7</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5.3</td>
<td>5</td>
<td>5.3</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9.3</td>
<td>9.3</td>
<td>9</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9.3</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11</td>
<td>11.7</td>
<td>11</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10</td>
<td>9.7</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11.3</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i40.pla</td>
<td>11</td>
<td>11</td>
<td>11.3</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8.7</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i100.pla</td>
<td>10.7</td>
<td>11</td>
<td>10.7</td>
</tr>
</tbody>
</table>

'Right' defines the 'right-side' behaviour. Q in number of input bits (average over 3 runs).
<table>
<thead>
<tr>
<th>File</th>
<th>Frequency</th>
<th>'Right'</th>
<th>0.05</th>
<th>0.05</th>
<th>0.0625</th>
<th>0.0625</th>
<th>0.0825</th>
<th>0.0825</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
</tr>
<tr>
<td>i10.pla</td>
<td></td>
<td></td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.3</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td></td>
<td></td>
<td>6.3</td>
<td>6.7</td>
<td>6</td>
<td>6.3</td>
<td>6.3</td>
<td>6</td>
</tr>
<tr>
<td>kaz.pla</td>
<td></td>
<td></td>
<td>5.7</td>
<td>5.3</td>
<td>5.3</td>
<td>5.3</td>
<td>5.3</td>
<td>5.3</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td></td>
<td></td>
<td>9</td>
<td>9</td>
<td>9.3</td>
<td>9</td>
<td>9</td>
<td>9.3</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td></td>
<td></td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9.3</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td></td>
<td></td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td></td>
<td></td>
<td>11.3</td>
<td>11.3</td>
<td>11</td>
<td>11.3</td>
<td>11.7</td>
<td>11.3</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
<td>9.7</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
<td>9.7</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td></td>
<td></td>
<td>11.3</td>
<td>11</td>
<td>11.3</td>
<td>11</td>
<td>11</td>
<td>11.3</td>
</tr>
<tr>
<td>i40.pla</td>
<td></td>
<td></td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11.7</td>
<td>11.3</td>
<td>11.3</td>
</tr>
<tr>
<td>i50.pla</td>
<td></td>
<td></td>
<td>8.3</td>
<td>9</td>
<td>8.7</td>
<td>9</td>
<td>8.7</td>
<td>8.7</td>
</tr>
<tr>
<td>i100.pla</td>
<td></td>
<td></td>
<td>10.3</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

'Right' defines the 'right-side' behaviour. Q in number of input bits (average over 3 runs).
Table 6.18 Mutation operator test for 'multi-allele' probability 1.0.

<table>
<thead>
<tr>
<th>File</th>
<th>'Right'</th>
<th>Frequency</th>
<th>0.05</th>
<th>0.0625</th>
<th>0.0825</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
<td>0.7</td>
</tr>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4</td>
<td>4</td>
<td>4.7</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5</td>
<td>5.3</td>
<td>5.3</td>
<td>5.3</td>
<td>5.3</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6.3</td>
<td>6.3</td>
<td>6.3</td>
<td>6.3</td>
<td>6.3</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5.7</td>
<td>5.3</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9</td>
<td>9</td>
<td>9.3</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>9</td>
<td>9.3</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9</td>
<td>9.3</td>
<td>9.3</td>
<td>9</td>
<td>9.7</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11.7</td>
<td>11</td>
<td>11.3</td>
<td>11</td>
<td>11.7</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9.7</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11</td>
<td>11.7</td>
<td>11</td>
<td>11.3</td>
<td>11.3</td>
</tr>
<tr>
<td>i40.pla</td>
<td>11</td>
<td>11.3</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i50.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9.7</td>
</tr>
<tr>
<td>i100.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

'Right' defines the 'right-side' behaviour. Q in number of input bits (average over 3 runs).

The performance of the genetic algorithm seems to be a little bit improved over the genetic algorithm with only the crossover operator. The multi-allele probability of 0.9 in conjunction with the 0.05 frequency setting shows the best result. On these settings the last check has been performed. The frequency 0.0375 has also been tested, and the right-side probability 1.0 has been added. The results for this test can be found in Table 6.19.
In Table 6.19 the average time per genetic algorithm run on a specific file added to the preprocessing time for that file has been listed too.

Based on this and the previous test run, we have decided to use the mutation operator with a probability of 0.05, a multi-allele probability of 0.9 and also a right-side probability of 0.9.

Now the standard genetic algorithm has been created, time has come to put some extra operators in it to enhance the power.

6.5 Adding the merge operator to the standard genetic algorithm

The merge operator will operate on two selected parents and will produce just a single offspring. The combined 1-alleles are used as maximum set and as long as some alleles are not required to cover the rows, they are stripped one by one. This can always be done at random, or by using a heuristic algorithm exclusively, or by a hybrid technique. Therefore, there are two aspects related to the merge operator:

- the probability for the operator to be called (frequency);
- the probability the heuristic algorithm will be applied (behaviour).
The first test run has explored the behavioural space on five evenly distributed points in the range \([0, 1]\). Simultaneously, three frequencies have been tested; 0.025, 0.05 and 0.1. The code for this test can be found in Appendix H. Table 6.20 and Table 6.21 contain the results.

Table 6.20 First test of merge operator on frequency and behaviour (part one).

<table>
<thead>
<tr>
<th>File</th>
<th>Heuristics</th>
<th>0.25</th>
<th>0.05</th>
<th>0.1</th>
<th>0.25</th>
<th>0.05</th>
<th>0.1</th>
<th>0.25</th>
<th>0.05</th>
<th>0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6</td>
<td>6.3</td>
<td>6</td>
<td>6.3</td>
<td>6</td>
<td>6</td>
<td>6.3</td>
<td>6</td>
<td>6</td>
<td>6.3</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>8.7</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8.7</td>
<td>9</td>
<td>9</td>
<td>8.7</td>
<td>9</td>
<td>8.3</td>
<td>9</td>
<td>9</td>
<td>8.7</td>
<td>8.7</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9</td>
<td>9</td>
<td>8.7</td>
<td>9</td>
<td>8.7</td>
<td>8.7</td>
<td>8.7</td>
<td>9</td>
<td>8.7</td>
<td>8.7</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11</td>
<td>11.3</td>
<td>11</td>
<td>11.3</td>
<td>11</td>
<td>11.1</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9.7</td>
<td>9.3</td>
<td>9</td>
<td>9.3</td>
<td>9.3</td>
<td>9.3</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9.3</td>
<td>9.3</td>
<td>9</td>
<td>9.3</td>
<td>9.3</td>
<td>9.3</td>
<td>9.3</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10.7</td>
<td>10.7</td>
<td>11</td>
<td>10.7</td>
<td>10.7</td>
<td>10.7</td>
</tr>
<tr>
<td>i40.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10.7</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10.7</td>
<td>10.7</td>
<td>10.3</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8.3</td>
<td>8</td>
<td>8.7</td>
<td>8.7</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i100.pla</td>
<td>10.7</td>
<td>10</td>
<td>10.3</td>
<td>10.3</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9.7</td>
</tr>
</tbody>
</table>

'Freq.' defines the frequency, heuristics the behaviour. Q in number of input bits (average over 3 runs).
These tables show very clear the superiority of the merge operator using the heuristic algorithm over the random one. But even the pure random behaviour shows better results for the genetic algorithm. As a matter of fact, the performance of the genetic algorithm has been improved so much, the results obtained using the pure heuristic algorithm and a 0.1 frequency (see Table 6.21) are even better than the results for the one run merge operator without genetic algorithm (see Section 2).

Especially the last run has been a very time consuming process, so it would be desirable to find a faster way to achieve the same results. Table 6.21 provides us with a clue; the runs for bigger problem instances seem to converge faster towards the optimal solution than the test runs for the smaller ones. Therefore, a new test has been performed on the merge operator using relative frequencies (relative to the problem size). We have explored three different settings; two times, three times and four times one divided by the number of alleles in each genotype (the size of the problem space).

As a side note, the expected offspring created by the merge operator cannot be computed by only taking into account the problem size and the merge probability. Of course, instead of the problem size the population size will be needed, but still one aspect has to be known too: the crossover probability. Why? Because every operator creates one offspring, except for the crossover operator. As that operator creates two new phenotypes and the total population size will be kept constant, less decisions about which operator to choose will be made than the number of elements in the population.

<table>
<thead>
<tr>
<th>Heuristics 'Freq.' 0.75</th>
<th>0.025</th>
<th>0.05</th>
<th>0.1</th>
<th>0.025</th>
<th>0.05</th>
<th>0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6.3</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9</td>
<td>8.7</td>
<td>8.7</td>
<td>8.3</td>
<td>8.3</td>
<td>8</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>9</td>
<td>8.7</td>
<td>8.7</td>
<td>8.3</td>
<td>8.3</td>
<td>8</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8.7</td>
<td>8</td>
<td>8.7</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11</td>
<td>10.7</td>
<td>11</td>
<td>11</td>
<td>10.7</td>
<td>10.7</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11</td>
<td>10.7</td>
<td>10.7</td>
<td>11</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40.pla</td>
<td>11</td>
<td>10.7</td>
<td>10.3</td>
<td>10</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i100.pla</td>
<td>9.7</td>
<td>9.7</td>
<td>9.3</td>
<td>9.7</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

'Freq.' defines the frequency, heuristics the behaviour. Q in number of input bits (average over 3 runs).
Suppose the probabilities for the crossover, merge, mutation and copy operator are respectively $P_c$, $P_{mer}$, $P_{mu}$ and $1 - P_c - P_{mer} - P_{mu}$. The average operator will create $(2P_c + P_{mer} + P_{mu} + (1 - P_c - P_{mer} - P_{mu})) \frac{1}{1 + P_c}$ new elements. Hence, the expected number of operators to be used equals the population size divided by $(1 + P_c)$.

As the default settings have been chosen to take a population size as big as twice the problem size, and the crossover probability has been set to 0.75, the real expected number of merge calls equals $\frac{8}{7}$ times the relative value per generation.

The results for these three relative frequencies are listed in Table 6.22, together with the average process time for the last setting.

### Table 6.22 Merge operator test on relative frequency, with pure heuristic behaviour.

<table>
<thead>
<tr>
<th>File</th>
<th>'Freq.'</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>T(min:sec)</th>
<th>File</th>
<th>'Freq.'</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>T(min:sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td>8.3</td>
<td>8.3</td>
<td>8.3</td>
<td>1:12</td>
<td>i40s09a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>3:30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>1:02</td>
<td>i40s09b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>3:14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>1:04</td>
<td>i40s10a.pla</td>
<td>10.3</td>
<td>10</td>
<td>10</td>
<td>10:54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11</td>
<td>10.7</td>
<td>11</td>
<td>3:51</td>
<td>i40.pla</td>
<td>10.7</td>
<td>10</td>
<td>10</td>
<td>10:41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i50.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>1:24</td>
<td>i100.pla</td>
<td>9.3</td>
<td>9.3</td>
<td>9</td>
<td>16:59</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

'Freq.' defines the relative frequency. $Q$ in number of input bits (average over 3 runs). $T$ includes preprocessing time, and is the average run time for the relative frequency of 4.

The influence of the stochastic process is clear; the column showing the relative frequency of 3 of Table 6.22 corresponds with the frequency $= 0.1$, heuristic $= 1$ column of Table 6.21 for the problems with 30 alleles (files beginning with '30s'). The latter column can be seen as a 'lucky shot'. Hence the relative frequency of 4 can be chosen as a good setting.

The process takes much more time now than the pure genetic algorithm did. A trade-off is possible between quality and execution time. The last test is still much faster than the test described in Table 6.21 - the time needed for an average run for 'i100.pla' with merge frequency 0.05 has been 21'08", while the merge frequency 0.1 needed even more time; 37'26".

### 6.6 Involving the repair operator

A number of tests have been performed and the settings for the genetic algorithm have become by now:

- Crossover: frequency 0.75; behaviour 2-point crossover.
- Mutation: frequency 0.05; behaviour multi-allele and right-side probability both 0.9.
- Merge: frequency relative 4; behaviour pure heuristic algorithm based.

The application of the repair operator depends on two aspects:

- Frequency: the probability a phenotype not covering all the rows will be repaired.
- Behaviour: the probability the algorithm is heuristic (opposite to random) based.

As the behaviour of the repair operator is defined identically to the behaviour of the merge operator, it has been set already to 1. Thus the frequency aspect is the only part we have to investigate. The initial test includes four frequencies, evenly distributed in the range [0.25; 1.0]. The results can be found in Table 6.23, while the code has been listed in Appendix I.
Table 6.23 Global repair operator test on frequency, with pure heuristic behaviour.

<table>
<thead>
<tr>
<th>File</th>
<th>'Freq.'</th>
<th>.25</th>
<th>.50</th>
<th>.75</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td></td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td></td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>kaz.pla</td>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8.7</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8.3</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td></td>
<td>8</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td></td>
<td>11</td>
<td>11</td>
<td>10.7</td>
<td>11</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td></td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td></td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td></td>
<td>10.3</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40s10b.pla</td>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i50.pla</td>
<td></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i100.pla</td>
<td></td>
<td>9.3</td>
<td>9</td>
<td>9.3</td>
<td>9.3</td>
</tr>
<tr>
<td>i100c.pla</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

'Freq.' defines the frequency. Q in number of input bits (average over 3 runs).

The repair operator gives no real improvement; certainly not if the extra processing time is taken into account. The incorporation of the repair operator with frequency 0.75 (the best setting shown in Table 6.23) results in an average run time taking just more than 2 hours for the file named 'i100.pla'.

A test run for the frequencies 0.05 and 0.15 resulted in similar values (however, the average time needed for 'i100.pla' in this case was 41'43"), as can be seen in Table 6.24.

Table 6.24 Repair operator test on small frequencies, with pure heuristic behaviour.

<table>
<thead>
<tr>
<th>File</th>
<th>'Freq.'</th>
<th>.05</th>
<th>.15</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td></td>
<td>8</td>
<td>8.3</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td></td>
<td>8</td>
<td>8.3</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td></td>
<td>11</td>
<td>10.7</td>
</tr>
<tr>
<td>i50.pla</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File</th>
<th>'Freq.'</th>
<th>.05</th>
<th>.15</th>
</tr>
</thead>
<tbody>
<tr>
<td>i40s09a.pla</td>
<td></td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td></td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td></td>
<td>10.3</td>
<td>10.3</td>
</tr>
<tr>
<td>i100c.pla</td>
<td></td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

'Freq.' defines the frequency. Q in number of input bits (average over 3 runs).

Based on these results, it has been decided the repair operator will not been used in the first settings of the genetic algorithm (this does not mean the repair operator will not been called in the algorithm; when a phenotype can be repaired to become for sure best phenotype ever, this will be done - see also Chapter 4).

6.7 The immigrate aspect

With the current settings (see the previous section's first paragraph for an overview), an extensive test has been performed on the influence of the immigrate aspect. All genetic algorithm tests described in Section 6.3 - 6.6 have contained 60 generations. For this test these 60 generations have been split into ten parts of six generations. Hence, after every six generations we can apply the immigrate aspect (as can be read in Chapter 4, Section 7, the influence of this aspect will not be restricted to just one generation, but for every next generation the immigrate effect is halved).
The size of the population reserved for new phenotypes has been related to the total population size; for every setting both 1/16 and 1/8 of this size has been used. The immigrate aspect has been applied after every six generations, after every odd part of six generations, after 12 and 30 generations, and six settings included only once the immigrate aspect; after 12, 18, ..., 42 generations. The two times nine corresponding results have been stored in Table 6.25 - 6.26. The specific main function part can be found in Appendix J.

Table 6.25 Using the immigrate aspect with size 1/16 times population size.

<table>
<thead>
<tr>
<th>File</th>
<th>All</th>
<th>Odd</th>
<th>3,6</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td>8.3</td>
<td>8</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>10.7</td>
<td>11</td>
<td>10.7</td>
<td>11</td>
<td>10.7</td>
<td>11</td>
<td>10.7</td>
<td>11</td>
<td>10.3</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40.pla</td>
<td>10.3</td>
<td>10</td>
<td>10.3</td>
<td>10</td>
<td>10.3</td>
<td>10</td>
<td>10.3</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i100.pla</td>
<td>9.3</td>
<td>9.3</td>
<td>9.3</td>
<td>9.3</td>
<td>9.3</td>
<td>9</td>
<td>9.3</td>
<td>9.3</td>
<td>9.3</td>
</tr>
</tbody>
</table>

When defines after how many times 6 generations the immigrate aspect has been applied. Q in number of input bits (average over 3 runs).

Table 6.26 Using the immigrate aspect with size 1/8 times population size.

<table>
<thead>
<tr>
<th>File</th>
<th>All</th>
<th>Odd</th>
<th>3,6</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8.7</td>
<td>8.3</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>10.7</td>
<td>11</td>
<td>11</td>
<td>10.7</td>
<td>11</td>
<td>10.7</td>
<td>11</td>
<td>10.7</td>
<td>11</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10.3</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i100.pla</td>
<td>9.3</td>
<td>9.3</td>
<td>9.3</td>
<td>9.3</td>
<td>9.3</td>
<td>9.7</td>
<td>9</td>
<td>9.3</td>
<td>9.3</td>
</tr>
</tbody>
</table>

When defines after how many times 6 generations the immigrate aspect has been applied. Q in number of input bits (average over 3 runs).

At first glance the immigrate aspect for 1/16th of the population size after 42 generations seems to do a good job, especially for the file named 'i30s10a.pla'. Unfortunately, two other runs with the same settings (not listed here) showed respectively an average quality of 10.7 and 11 bits,
while they needed for other files more bits too. Therefore this good result has not to be taken too seriously.

The immigrate aspect has not showed itself to be an improvement over the genetic algorithm without it, so the settings still stay the same.

6.8 The population size

Before further exploring the local optimum of the settings, an important aspect for the speed of the algorithm has to be taken into account; the size of the population. It has been set equal to two times the size of the problem space (after removing essential and dominated parts). In this section the influence of a smaller population will be researched.

Three settings have been tested; .5, 1, and 1.5 times the problem dimension (the merge probability has been corrected on these values in order to perform this operator on average equally often). Appendix K lists the code. The results can be found in Table 6.27.

<table>
<thead>
<tr>
<th>File</th>
<th>Size 0.5</th>
<th>Size 1</th>
<th>Size 1.5</th>
<th>T(min:sec)</th>
<th>File</th>
<th>Size 0.5</th>
<th>Size 1</th>
<th>Size 1.5</th>
<th>T(min:sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td>9 8.7 8</td>
<td>0:50</td>
<td>i40s09a.pla</td>
<td>9 9 9</td>
<td>2:28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8 8 8</td>
<td>0:48</td>
<td>i40s09b.pla</td>
<td>9 9 9</td>
<td>2:34</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8 8.3 8</td>
<td>0:49</td>
<td>i40s10a.pla</td>
<td>10.3 10 10</td>
<td>8:38</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11 11 11</td>
<td>3:15</td>
<td>i40.pla</td>
<td>10 10.7 10</td>
<td>8:30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i50.pla</td>
<td>8 8 8</td>
<td>0:57</td>
<td>i100.pla</td>
<td>10 10 9</td>
<td>13:24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Size defines the relative population size. Q in number of input bits (average over 3 runs). T includes preprocessing time, and is the average run time for the relative population size of 1.5.

As can be seen from this table, 1.5 is a superior setting over the other two. This will become the new population size for the genetic algorithm, because compared to the results found using 2 times the problem dimension for the population size, this setting leads to comparable results. However, the total processing time for an average run with this setting is roughly speaking a factor 3/4 of the time needed by the old setting for many benchmarks. This could be expected, because the total algorithm now performs 3/4 of the previous performed tasks except for the preprocessing part.

Now the settings for the algorithm have been tuned one by one, it is very well possible the early fixed settings for the operators could be improved by the new environment. So, a new tuning will be performed, although this will be only done in promising regions found by the previous results.

6.9 Fine tuning the crossover and mutation operators

The first test performed to create as good a genetic algorithm as possible by using only a genetic operator cannot be compared to the much more sophisticated algorithm we are testing now. Therefore, the setting for the crossover operator, which has not been changed since then, should be reconsidered. The promising frequency region was around 0.75, and the choice for the
behaviour setting was a very hard one, as there were almost no differences between them. Hence, the new test includes the frequencies 0.65, 0.75 and 0.85, as well as five different behaviours; both 1 and 2 times 2-point crossover, and 3 settings using uniform crossover (with swap chance 0.2, 0.4 and 0.5). The results can be found in Table 6.28 - 6.29.

Table 6.28 Tuning the uniform crossover operator in the frequency range [0.65; 0.85].

<table>
<thead>
<tr>
<th>File</th>
<th>Type 0.2</th>
<th>Type 0.4</th>
<th>Type 0.5</th>
<th>Type 0.2</th>
<th>Type 0.4</th>
<th>Type 0.5</th>
<th>Type 0.2</th>
<th>Type 0.4</th>
<th>Type 0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11</td>
<td>11</td>
<td>10.7</td>
<td>11</td>
<td>11</td>
<td>10.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>10.7</td>
<td>10.3</td>
<td>10</td>
<td>10.7</td>
<td>10.3</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i50.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i100.pla</td>
<td>9</td>
<td>9.3</td>
<td>9</td>
<td>9.3</td>
<td>9</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Type defines the swap chance of the uniform crossover. Q in number of input bits (average over 3 runs).

Table 6.29 Tuning the n*2-point crossover operator in the frequency range [0.65; 0.85].

<table>
<thead>
<tr>
<th>File</th>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 1</th>
<th>Type 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td>8.3</td>
<td>8.7</td>
<td>8.3</td>
<td>8.7</td>
<td>8.3</td>
<td>8.7</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8</td>
<td>8</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
<td>8.3</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10.7</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>10.3</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10.3</td>
<td>10.3</td>
</tr>
<tr>
<td>i40.pla</td>
<td>10.3</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10.7</td>
<td>10.7</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i100.pla</td>
<td>9</td>
<td>9</td>
<td>10.7</td>
<td>9.7</td>
<td>9.7</td>
<td>9.7</td>
</tr>
</tbody>
</table>

Type defines the value n of the n times 2-point crossover. Q in number of input bits (average over 3 runs).

Remarkable are the relatively poor results obtained by the settings obtained in the previous section; frequency 0.75 and 2 times 2-point crossover behaviour. The superior setting is frequency 0.65 with uniform crossover using a swap chance of 0.5. Both the good results of the setting using the same frequency, but a swap chance of 0.4, as well as the setting using the same behaviour, but a frequency of 0.75 gives the impression this is a reliable result.
With this setting, again a test on the mutation operator settings has been done. The frequency dimension has been tested on three locations; 0.025, 0.05 and 0.1. Two settings for the right-side aspect have been performed; 0.7 and 0.9. Three multi-allele probability settings have been applied; 0.3, 0.6 and 0.9. The results are listed in Table 6.30 - 6.31.

Table 6.30 Tuning the mutation operator for the right-side probability 0.7.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>0.025</th>
<th>0.05</th>
<th>0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Multi'</td>
<td>0.3</td>
<td>0.6</td>
<td>0.9</td>
</tr>
<tr>
<td>File</td>
<td>0.3</td>
<td>0.6</td>
<td>0.9</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>8</td>
<td>8.3</td>
<td>8</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8</td>
<td>8.3</td>
<td>8</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11</td>
<td>11</td>
<td>10.7</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40.s08c.pla</td>
<td>10.3</td>
<td>10</td>
<td>10.3</td>
</tr>
<tr>
<td>i50.s08a.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i100.pla</td>
<td>9.3</td>
<td>9.3</td>
<td>9.3</td>
</tr>
</tbody>
</table>

'Multi' defines the 'multi-allele' behaviour. Q in number of input bits (average over 3 runs).

Table 6.31 Tuning the mutation operator for the right-side probability 0.9.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>0.025</th>
<th>0.05</th>
<th>0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Multi'</td>
<td>0.3</td>
<td>0.6</td>
<td>0.9</td>
</tr>
<tr>
<td>File</td>
<td>0.3</td>
<td>0.6</td>
<td>0.9</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>8.3</td>
<td>8.3</td>
<td>8.3</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8.3</td>
<td>8.3</td>
<td>8.3</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8</td>
<td>8</td>
<td>8.3</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>10.7</td>
<td>10.7</td>
<td>11</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40.s08c.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i50.s08a.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i100.pla</td>
<td>9.7</td>
<td>9.3</td>
<td>10</td>
</tr>
</tbody>
</table>

'Multi' defines the 'multi-allele' behaviour. Q in number of input bits (average over 3 runs).

These results are probably very 'luck' dependent, but as it will not be a worse choice than the others, it has been decided to switch the setting of the right-side aspect to 0.7 (leaving the frequency to 0.05 and the 'multi-allele' probability to 0.9).
The settings have now become:
Crossover: frequency 0.65; behaviour uniform crossover with swap chance 0.5.
Mutation: frequency 0.05; behaviour multi-allele 0.9 and right-side probability 0.7.
Merge: frequency relative 4; behaviour pure heuristic algorithm based.

6.10 Changing the merge operator during the algorithm

In this section the merge operator will be taken into account, sometimes in combination with the repair operator or immigrate aspect, but always varying the frequency of the merge operator when the genetic algorithm is running. There are two goals for these tests; improving the solutions found by the algorithm and bringing down the execution time.

The disappointing results produced by the repair operator (see Section 6) might be explained by the good results of the merge operator (maybe those results are hard to improve). To check again its influence, four new tests have been performed. As the genetic algorithm is producing 60 generations divided by ten blocks, changing the settings of the operators always takes place after \( n \) times six generations.

Test 1: the frequency of the merge operator is set at 0 during the first six generations, while the repair frequency is set at 10% during the same interval.
Test 2: the first six times six generations are normal, but thereafter the merge frequency is set to 0 and the frequency of the repair operator becomes 10%.
Test 3: idem as test 2, but now also the immigrate aspect is set to .25 times the population size after 36 generations.
Test 4: the first three times six generations are normal, thereafter the immigrate aspect is set to .25 times the population size, the repair frequency is set to 10% and the merge frequency is set to 0, but after four times six additional generations, all is set back to normal.

The reason for the immigrate aspect is to prevent the algorithm from converging - the repair operator would become useless.

Besides these four tests, one additional test has been performed during this run to get a better comparison of the results; the run where all settings stay normal. The results can be found in Table 6.32. The code has been printed in Appendix L.
Table 6.32 Testing the repair - merge combination.

<table>
<thead>
<tr>
<th>File</th>
<th>Test 1 (rep⁺)</th>
<th>Test 2 (rep⁻)</th>
<th>Test 3 (rep⁺)</th>
<th>Test 4 (rep⁻)</th>
<th>normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td>8</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8.3</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8.7</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8.7</td>
<td>8.7</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>10.7</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>10.7</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i40.pla</td>
<td>10.3</td>
<td>10</td>
<td>10.3</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i100.pla</td>
<td>10</td>
<td>9.7</td>
<td>9</td>
<td>10</td>
<td>9.3</td>
</tr>
</tbody>
</table>

Test defines the test number (see text, here cryptic displayed). Q in number of input bits (average over 3 runs).

The differences are very small. All settings seem to do a good job. It would probably be smarter to try to reveal the power and weaknesses for the merge operator separately before trying to combine it with the repair operator (and improve the genetic algorithm).

To research the influence of the merge operator during a run, three sorts of tests have been performed; using the merge operator not until after some generations have been made, changing between using the merge operator and not using it, and starting in the normal way, but removing the merge operator before the algorithm has finished. The results are showed in Table 6.33.

Table 6.33 Revealing the power and weakness of the merge operator.

<table>
<thead>
<tr>
<th>Type</th>
<th>Starting at generation</th>
<th>Start switching at</th>
<th>Stopping at generation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7 19 31 43</td>
<td>7 19 31 43</td>
<td>7 19 31 43</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>8.3 8.7 9 8.7</td>
<td>8.3 9 9 9</td>
<td>9.7 8.7 8.3 8</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8 8.3 8.7 9</td>
<td>8.3 8.7 8.3 9</td>
<td>9 8 8 8.3</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8 8.7 9 9</td>
<td>8 8.3 8.3 8.3</td>
<td>9.3 8 8 8.3</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11 11 11 11</td>
<td>11 11 11 11</td>
<td>12 10.7 10.7 10.3</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>9 9 9 9.3</td>
<td>9 9.3 9.3 10</td>
<td>10 9 9 9</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>9 9 9 9.7</td>
<td>9 9.3 9.3 9.3</td>
<td>9.7 9 9 9</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>10 10.7 11 11</td>
<td>10.3 10.3 11 11</td>
<td>11.3 10 10 10.3</td>
</tr>
<tr>
<td>i40.pla</td>
<td>10 10.7 11 10.7</td>
<td>10.3 10 11 10.7</td>
<td>11 10.7 10 10.7</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8 8 8 8</td>
<td>8 8 8 8</td>
<td>9 8 8 8</td>
</tr>
<tr>
<td>i100.pla</td>
<td>9.3 10 10 10</td>
<td>9.3 9.7 10 10</td>
<td>11 9 9.7 10</td>
</tr>
</tbody>
</table>

Type 'switching' starts using merge at the indicated generation and switches every 6 generations thereafter. Q in number of input bits (average over 3 runs).
There are big differences between these settings. It has not come as a surprise that the less the merge operator has been used, the worse the solution has become. It is a surprise, however, that the influence of the first 18 generations is remarkable high. Using the merge operator until 18 generations (of the 60) is a much more better choice than start using it by then for the remainder of the algorithm.

With this knowledge, we can further try to exploit the merge operator. The relative frequency of the merge operator has been kept constant throughout the algorithm (its value is set to 4). The time for the algorithm will be roughly the same when the probability value would be set twice as high during half the execution time, and to zero during the other half. So we can experiment for better settings.

A new test uses this idea. The relative frequency of the merge operator decreases in 4 steps (each step taking 6 generations) from 10 to 2, and sticks for the remainder 6 steps at that value. Four variations have been tested:

1. the pure form;
2. the pure form including a repair probability of .33;
3. the pure form including the immigrate aspect of relative size .25 after 18 and 36 generations;
4. the pure form including both a repair probability of .33 and the immigrate aspect of relative size .25 after 18 and 36 generations.

The results are listed in Table 6.34.

Table 6.34 Decreasing the relative merge probability from 10 to 2.

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>File</th>
<th>Type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td>8.25</td>
<td>8.25</td>
<td>8</td>
<td>8</td>
<td>8.25</td>
<td>i40s09a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8.25</td>
<td>i40s09b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>i40s10a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>10.5</td>
<td>10.5</td>
<td>10.75</td>
<td>10.75</td>
<td>10.75</td>
<td>i40.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>i50.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>i100.pla</td>
<td>9</td>
<td>9</td>
<td>9.25</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Type defines the exact settings (see text for details). Q in number of input bits (average over 4 runs).

All these results are very good. Unfortunately, the addition of the repair operator costs an enormous amount of execution time; the time difference between variant 1 and 2 for the file named 'i100.pla' is more than 26 minutes! This means that the execution time for variant 2 in this case is three times that of variant 1. Adding the repair operator seems a bad idea.

On the other hand, it still is not clear whether the immigrate aspect is a good choice or not. To gain some more insight, two more rigid versions of the immigrate aspect have been tested besides the pure form (once again, to tackle the stochastic influence):

5. the pure form including the immigrate aspect of relative size 0.25 after every 12 generations;

With the relative merge frequency the merge probability multiplied by half the population size is meant, see also Section 5 and 8.
6. the pure form including the immigrate aspect of relative size 0.25 after every 6 generations. Table 6.35 shows the results.

Table 6.35 Decreasing the relative merge probability from 10 to 2 (+ immigrate aspect).

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>1</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td>8.25</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8.25</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>10.25</td>
<td>10.5</td>
<td>10.75</td>
<td></td>
</tr>
<tr>
<td>i50.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>1</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>i40s09a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>i40.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>i100.pla</td>
<td>9.25</td>
<td>9.5</td>
<td>9.25</td>
<td></td>
</tr>
</tbody>
</table>

Type defines the exact settings (see text for details). Q in number of input bits (average over 4 runs).

As can be seen, the first pure form results (Table 6.34) were somewhat better than the second. The immigrate aspect looks promising and more stable.

The last test on changing the merge operator during the algorithm we have performed decreases more smoothly the relative merge probability. Starting at 8, it decreases in 4 steps to 4 (each setting holding for 6 generations), whereafter it is set to 2 for the remainder of the algorithm. In two variants there is no remainder, because the algorithm will stop after 30 generations (instead of the usual 60). The four tested variants are:

1. the new pure form;
2. the new pure form including the immigrate aspect of relative size 0.25 after every 6 generations;
3. the new pure form, ending after 30 generations;
4. the new pure form including the immigrate aspect of relative size 0.25 after every 6 generations, ending after 30 generations.

The results of these four variants are listed in Table 6.36. Within this test, after 18 generations the algorithm reported an interim solution. Generally spoken, the result did not improve thereafter (the interim result improved in 5% of the long runs with one bit - this did not happen for the short runs, however).

Table 6.36 Decreasing the merge probability, including immigrate and stop aspects.

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td>8.5</td>
<td>8</td>
<td>8.5</td>
<td>8.25</td>
<td></td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>10.5</td>
<td>10.75</td>
<td>11</td>
<td>10.75</td>
<td></td>
</tr>
<tr>
<td>i50.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i40s09a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>10</td>
<td>10</td>
<td>10.25</td>
<td>10.25</td>
<td></td>
</tr>
<tr>
<td>i40.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10.25</td>
<td></td>
</tr>
<tr>
<td>i100.pla</td>
<td>9.25</td>
<td>9.25</td>
<td>9</td>
<td>9.25</td>
<td></td>
</tr>
</tbody>
</table>

Type defines the exact settings (see text for details). Bold printed numbers indicate settings for which the final solution was better than the intermediate after 18 generations. Q in number of input bits (average over 4 runs).

The average execution time for each of the runs can be found in Table 6.37. Except for the two files 'i40s10a.pla' and 'i40.pla', there are no big differences between the execution times of the algorithm with the immigrate aspect and the one without it. The exception for these two files can
probably be explained by the fact the population of the algorithm without the immigrate aspect converges early, and thereafter the merge operator has an easy job, while for the other population there are still new regions to research. The fact the execution times for the short algorithms for these files do not differ significantly support this hypotheses.

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i30s08a.pla</td>
<td>1</td>
<td>0:53</td>
<td>0:53</td>
<td>0:43</td>
<td>0:43</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>2</td>
<td>0:51</td>
<td>0:55</td>
<td>0:41</td>
<td>0:44</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>3</td>
<td>0:49</td>
<td>0:52</td>
<td>0:40</td>
<td>0:44</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>4</td>
<td>4:02</td>
<td>3:30</td>
<td>3:16</td>
<td>3:09</td>
</tr>
<tr>
<td>i50.pla</td>
<td>5</td>
<td>1:09</td>
<td>1:10</td>
<td>0:52</td>
<td>0:53</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>6</td>
<td>2:40</td>
<td>2:50</td>
<td>2:10</td>
<td>2:08</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>7</td>
<td>2:46</td>
<td>2:37</td>
<td>1:59</td>
<td>2:07</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>8</td>
<td>8:55</td>
<td>10:06</td>
<td>7:32</td>
<td>7:14</td>
</tr>
<tr>
<td>i40.pla</td>
<td>9</td>
<td>8:30</td>
<td>10:05</td>
<td>7:09</td>
<td>7:35</td>
</tr>
<tr>
<td>i100.pla</td>
<td>10</td>
<td>13:53</td>
<td>13:39</td>
<td>10:37</td>
<td>11:22</td>
</tr>
</tbody>
</table>

Table 6.37 Average execution times corresponding with the results of Table 6.36.

Still it is a question whether the incorporating of the immigrate aspect has been an improvement or not. Unfortunately, the current test set is too small to do further research on the settings. It would be wise to extend the set by many new benchmarks, so that only a part has to be used for adjusting the settings, and the others can be used to test the actual algorithm (on the moment it is not clear whether the algorithm does a good job because it has been tuned to the test set or the results are good because the algorithm has been set well).

6.11 A last look at the stand alone merge operator

During the tests of the algorithm the unbalanced behaviour of the heuristic algorithm of the merge operator came to mind. The merge operator computes repeatedly the importance factors of each column in the matrix, adding the one with the highest factor to the 'essentials', reducing the matrix and removing dominated columns (which could be created after the reducing). It does not matter whether there is a big difference between two importance factors or not, the highest is chosen.

On the other hand, it is very well possible the lowest importance factor is really low compared to the others. Adding the corresponding column to the set of 'dominated' is a wise strategy in such a case. The matrix can be reduced, and it is possible an essential column will be found, or, almost as good as that, the differences between the high factors could increase, making the choice easier and better.

To incorporate this idea in the algorithm, three variables have been added to the merge method. First, the value of the lowest importance factor is needed, but after that, both the second to highest and second to lowest variables are needed too. A test has been performed using these new variables in three different ways:

1. if the difference between the two lowest is bigger than the difference between the two highest (Δl > Δh), treat the to the lowest importance factor corresponding column as a dominated one, else do as always (treat the 'highest' as an essential one).
2. the same as the previous one, but now half the difference of the two lowest is used (Δl > 2Δh).
3. the same as the previous one, but now twice the difference of the two lowest is used ($2\Delta l > \Delta h$).

**Table 6.38** Using only the merge operator in a new way, testing order dependent.

<table>
<thead>
<tr>
<th>File</th>
<th>$Q(1, \Delta l &gt; \Delta h)$</th>
<th>$Q(2, \Delta l &gt; 2\Delta h)$</th>
<th>$Q(3, 2\Delta l &gt; \Delta h)$</th>
<th>$Q(\text{old})$</th>
<th>$Q(\text{min})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>i40.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>i100.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

See text for explanation of the settings. $Q$ in number of input bits.

In the implementation the lowest and highest values were assigned initially impossible values (the number of columns and zero, respectively) whereafter the importance factor of each column was checked against these values. If necessary they were adjusted (and the second to highest or lowest value as well, of course).

The results of this test are stored in Table 6.38. Execution time is very similar to the new times given in Table 6.2. The setting using $\Delta l > 2\Delta h$ seems to have only advantages above the old setting (it never needs more bits, but for one file it needs one bit less).

The implementation of this test has been changed to check each importance factor against the second to highest and second to lowest values. Now the order in which the columns are checked is not important any more. The new results are listed in Table 6.39.
Table 6.39 Using only the merge operator in a new way, testing order independent.

<table>
<thead>
<tr>
<th>File</th>
<th>$Q(1, \Delta l &gt; \Delta h)$</th>
<th>$Q(2, \Delta l &gt; 2\Delta h)$</th>
<th>$Q(3, 2\Delta l &gt; \Delta h)$</th>
<th>$Q(\text{old})$</th>
<th>$Q(\text{min})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10.pla</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a.pla</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a.pla</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a.pla</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>kaz.pla</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>i30s08a.pla</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i30s08b.pla</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>i30s08c.pla</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>i30s10a.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>i40s09a.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s09b.pla</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>i40s10a.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>i40.pla</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>i50.pla</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>i100.pla</td>
<td>10</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>$&lt;9$</td>
</tr>
</tbody>
</table>

See text for explanation of the settings. $Q$ in number of input bits.

Now it is less obvious whether it is advantageous to use the setting $\Delta l > 2\Delta h$ (although it is the best choice of the three). Again, it is clear the current test set is too small to draw a conclusion about the performance differences between small changes in the algorithm. Therefore, it would not be useful to perform related tests, like using the differences between the mean importance factor and the highest or lowest one, nor trying to find a bias to decide whether or not the highest importance factor has to be added to the 'essentials', as long as the current test set will be used.

6.12 Ideas about new tests to be performed

Because all of the tests have been performed on a small test set and most of the tests included stochastic processes, the most important aspect will be extending the current test set. When a new test set will be generated, it would be desirable to create benchmarks not only in the 'lower region of the topology map'. In other words, minimal input support problems of size $n$ should equally often have solutions needing more than $n/2$ bits as ones needing less than $n/2$. Especially the region $n/2$ can be a hard job for this algorithm, as well as for QuickScan, because in that region many local optima can be possible and there are less paths leading to points within this region.

With the above mentioned limitations, there are some insights gained from the total test path. The genetic algorithm is very robust. The settings are not crucial for its behaviour. Because it is a stochastic process, many runs are needed to validate conclusions (often the chosen setting of a
previous test - based on its good performance - turned out to be worse than expected in a next test. The generic genetic algorithm (using only crossover and mutation) did a good job; execution time is acceptable and it could be implemented very fast (especially with the growing interest in the topic it is to be expected public packages will become available quickly making it possible to do such a task instantly). It has been possible to incorporate problem specific knowledge, improving the algorithm at the expense of execution time. Unfortunately, the repair operator did not show itself to be useful.

Having created a new test set, it would be interesting to test the current settings of the full static genetic algorithm (Section 9), which are:
Crossover: frequency 0.65; behaviour uniform crossover with swap chance 0.5.
Mutation: frequency 0.05; behaviour multi-allele 0.9 and right-side probability 0.7.
Merge: frequency relative 4 (see Section 8); behaviour pure heuristic algorithm based.
Repair: frequency 0.
Immigrate aspect: not used.
Population size: 1.5 times the problem size.
Number of generations: 60.

Some dynamic genetic algorithms (Section 10) could be performed again to gain insight too, especially those using the same settings as the static algorithm except for the merge operator (decreasing from 10 to 2 or 8 to 4), testing on the immigrate aspect and number of generations (as can be found in Table 6.35 and 6.36).

Some regions look promising for further exploration. Besides more research on dynamically adaptation of the operator frequencies and behaviour, especially the merge operator can probably be improved. The test on the highest importance factor can be extended (see also previous section). It could be an advantage to make use of offsets. When an importance factor is higher than a specified offset, the corresponding column will always be added to the 'essentials', no matter whether $\Delta l > \lambda x \Delta h$ holds or not. Using this model, there should also be an offset for the lowest factor; when it is less than the offset the corresponding column will be treated as a dominated one.

Another possible improvement of the merge operator within the genetic algorithm could be the way it is called. Now the union of the 1-alleles of two selected genotypes is computed, whereafter the merge operator will return a subset as a result. This could be seen as a top down approach. The alteration could be to take the intersection of the 1-alleles, and making the merge operator return a superset (like the repair operator does), which can be seen as a bottom up approach. A first, small test run (not included in this report) looked promising. Even the implementation of a merge operator needing the union and intersection to compute a set in between as a result could come to mind.

Many other ideas are possible. Incorporating an adapting merge operator in the genetic algorithm, sometimes using this, then using that algorithm. New ways the importance factors are computed. And so on, and so on. It will be a time consuming task. Besides I do not have the time to do it, I have probably become 'problem blind'. I am thinking in a specific direction, not using the many rich opportunities laying there to take. Somebody with a fresh mind can improve this project more than I can at this point. I hope he will be found and I wish him as much pleasure as I have had throughout this research.
7. Conclusions and recommendations

We have developed a robust genetic algorithm that computes nearly optimal values for the Minimal Input Support Problem. The algorithm using only the standard genetic operators comes with acceptable solutions. As a matter of fact, these solutions are almost identical to the solutions found by QuickScan, although the execution time of the latter is remarkably shorter, especially for large problem instances.

Incorporating a problem specific operator (merge) into the genetic algorithm had been successful; for the more complicated problems the solutions found needed mostly one bit less. Unfortunately, the extra computations had quite much lengthened the execution time. The introduced operator did not only turn out to be powerful within the genetic algorithm, but also as a stand alone heuristic algorithm it delivered good results. Generally spoken the quality of those results are between the QuickScan and the genetic algorithm, and the same holds for length of the execution time. In single cases, the results of merge alone are even better than that of the genetic algorithm with the merge operator included.

Some extra tests on the merge operator showed us that its use is especially important during the first generations of the algorithm. Some opportunities have been found to possibly improve the stand alone merge operator, as its behaviour is a little bit asymmetric. Incorporating this new implementation might also improve the genetic algorithm.

The results we have found using the standard genetic operators and the results found by other researchers on other fields, show us the power of genetic algorithms. The totally different approach of handling the problem can lead to new insights. On the other hand, a good understanding of the problem structure and its specific aspects can result in new operators for the genetic algorithm, making it less general but more powerful.

It has become clear that the implementation of the code can be improved in order to speed it up. Compared to QuickScan, the new code often needs more time to just perform the preprocess module than the old code needed to complete the task (which includes the preprocessing). When faster code could be implemented, the genetic algorithm can really become a very powerful tool. As a matter of fact, as there has been growing interest in the subject, in the near future there might be developed a very fast general genetic algorithm, that easily can be customized to the considered problem. In the meantime, it should be considered to rewrite the classes MakeMatrix and Matrix based on the preprocess code in the QuickScan algorithm.

Some aspects of the genetic algorithm have not been tested yet. The probabilities for the operator selection and their behaviour have been set using a one or two dimensional search. During the search, the other settings have stayed fixed. Besides a first test on changing the frequency of the merge operator throughout the algorithm, the powerful property of dynamically adapting the probabilities has not been tested by this approach. It would be even more interesting to develop an algorithm that adapts the behaviour of the genetic algorithm based on the diversity of the population or the (lack of) improvement of the fitness of the best genotype found so far. The stop condition of the algorithm has to be based on the same information.

The main reason we did not test these aspects was a shortage of time. But even if we had time enough, it would not be useful to try testing those properties on a test set of fifteen elements. Even during the project we encountered the limitation, as we made choices about settings,
although the differences between them often were minimal (which of course stresses the robustness of the algorithm). Hence, it would be worthwhile to extend the current test set. In this way it will become possible both to improve and research the behaviour and test it on problems for which they were not optimized.

When new elements will be added to the test set, it would be a good idea to take the topology of MISP. If solutions to those problems contain about half of the input bits, it would be much harder to escape from an inferior local optimum. Differences between algorithms would become more clear.

The quality of QuickScan depends on the sequence of the input bits. Opposite to the genetic algorithm or the sole merge operator, differences in coding can lead towards the global optimum. It would be interesting to combine these two algorithms. It is possible to design a genetic algorithm that judges the sequence of the bits. Starting from the left most position, the genetic algorithm takes as many bits as needed to create an input support. The less bits needed, the better the solution will be. Just as in the Travelling Sales Person problem, the genetic operators change the ordering of the bits. In this way, related bits will come close to each other. After a few populations, the best sequence found so far can be made the starting position for QuickScan.

Another way to combine algorithms in order to improve QuickScan can be found by using the importance factors. The importance factors are used by the sole merge operator to select the most promising column to be put into the solution. The results of that operator show the usefulness of these factors. However, when a column has been chosen to be used, the importance factors of the other columns have to be computed again (two input bits covering almost the same rows in the matrix will compete for the position, but will not both be picked). The differences in importance factors can be computed for every column on the other ones. Now we can make another ordering of the input bits. Two columns having a huge influence on each other (the difference of the importance factor of the column is high when the other column has been chosen) have to become far away from each other as two columns with a small influence have to become neighbours. Now QuickScan should probably find better results than previously.
## Appendix A Benchmark set

### Table A.1 The characteristics of the MISP testset.

<table>
<thead>
<tr>
<th>File</th>
<th>#inputs</th>
<th>#terms</th>
<th>Q(min)</th>
<th>#min</th>
<th>#essential</th>
<th>#dominated</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10</td>
<td>10</td>
<td>180</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>i20s04a</td>
<td>20</td>
<td>20</td>
<td>4</td>
<td>31</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i20s05a</td>
<td>20</td>
<td>30</td>
<td>5</td>
<td>66</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i20s06a</td>
<td>20</td>
<td>40</td>
<td>6</td>
<td>72</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>kaz</td>
<td>21</td>
<td>31</td>
<td>5</td>
<td>35</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i30s08a</td>
<td>30</td>
<td>120</td>
<td>8</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i30s08b</td>
<td>30</td>
<td>120</td>
<td>8</td>
<td>17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i30s08c</td>
<td>30</td>
<td>120</td>
<td>8</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i30s10a</td>
<td>30</td>
<td>240</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i40s09a</td>
<td>40</td>
<td>160</td>
<td>9</td>
<td>1632</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i40s09b</td>
<td>40</td>
<td>160</td>
<td>9</td>
<td>1632</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i40s10a</td>
<td>40</td>
<td>240</td>
<td>10</td>
<td>716</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i40</td>
<td>40</td>
<td>240</td>
<td>10</td>
<td>716</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i50</td>
<td>50</td>
<td>100</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i100</td>
<td>100</td>
<td>200</td>
<td>?</td>
<td>?</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

All filenames have the extension ".pla".

- **#inputs**: The number of inputs defined in the file.
- **#terms**: The number of term lines defined in the file.
- **Q(min)**: The minimal number of inputs needed to preserve functionality.
- **#min**: The number of different solutions to MISP.
- **#essential**: The number of essential inputs defined in the file.
- **#dominated**: The number of dominated inputs defined in the file.
## Appendix B QuickScan's performance on benchmark set

Table B.1 Results of the QuickScan algorithm on the MISP testset.

<table>
<thead>
<tr>
<th>File</th>
<th>T(qs)</th>
<th>Q(qs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i10</td>
<td>&lt;1</td>
<td>7</td>
</tr>
<tr>
<td>i20s04a</td>
<td>&lt;1</td>
<td>4</td>
</tr>
<tr>
<td>i20s05a</td>
<td>&lt;1</td>
<td>5</td>
</tr>
<tr>
<td>i20s06a</td>
<td>&lt;1</td>
<td>6</td>
</tr>
<tr>
<td>kaz</td>
<td>&lt;1</td>
<td>6</td>
</tr>
<tr>
<td>i30s08a</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>i30s08b</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>i30s08c</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>i30s10a</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>i40s09a</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>i40s09b</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>i40s10a</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>i40</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>i50</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>i100</td>
<td>15</td>
<td>11</td>
</tr>
</tbody>
</table>

| T(qs): Time in seconds needed by QuickScan to finish (includes preprocessing). |
| Q(qs): The number of inputs needed by QuickScan's solutions. |
Appendix C Statistical analysis of the tournament ranking

Assumption: to select a phenotype two phenotypes are chosen randomly and the best one is selected.

The crossover operator needs two parents and creates two children, while the mutation operator generates one child using one parent. To create the next generation for a population with \( n \) members, \( n \) parents are needed and \( 2n \) phenotypes have to be chosen randomly.

Without losing generality, all phenotypes can be ordered according to their fitnesses, using an integer index between 0 and \( n-1 \). For every two phenotypes \( P_i \) and \( P_j \) chosen holds \( i < j \) \( \iff \) \( \text{Fitness}(P_i) > \text{Fitness}(P_j) \)\(^\text{18}\). Thus, \( P_{\text{worst}} \) refers to the worst phenotype (lowest fitness). Now, for every \( P_i \) there are exactly \( n-i \) phenotypes having the same fitness function or worse.

Consider the notation for the two randomly chosen phenotypes \( P_x \) and \( P_y \): \( C(x, y) \). If \( x < y \) then \( P_x \) is selected, else \( P_y \). The probability for a specific phenotype \( P \) to be selected in a single tournament round (just one pair is evaluated) (notation \( p(P) \)) can now be formulated:

\[
p(P_i) = p(C(i, y > i)) + p(C(x \geq i, i)) - p(C(i, i))
\]

\[
= \frac{1}{n} \cdot \frac{n-i}{n} + \frac{n-i}{n} \cdot \frac{1}{n} - \frac{1}{n^2}
\]

\[
= \frac{2(n-i) - 1}{n^2}
\]

Because \( n \) pairs are chosen, the expected number of selections is \( n \) times this probability.

The chance for a specific phenotype \( P_i \) to be not selected during all tournament rounds (notation \( p(\neg P_i) \)) can be formulated too:

\[
p(\neg P_i) = (1 - \frac{2(n-i) - 1}{n^2})^n
\]

\[
= \left( \frac{(n-1)^2 + 2i}{n^2} \right)^n
\]

For the best phenotype this formula becomes:

\[
p(\neg P_{\text{best}}) = \left( \frac{n-1}{n} \right)^{2n}
\]

For big population sizes this formula converges to \( \epsilon^2 \).

\(^{18}\)When there are two (or even more) phenotypes with equal fitnesses, the only change in the story is they have equal chances to be selected. The corresponding selection chances have to be averaged.
Appendix D Making an executable for the genetic algorithm

D.1 The standard code for the main function

```c
#if !defined( __GA_H )
#include "ga.h"
#endif

#if !defined( __MATRIX_H )
#include "matrix.h"
#endif

#if !defined( __MAKEMATRIX_H )
#include "makematx.h"
#endif

#include <fstream.h>
#include <stdlib.h> // for system

// TIMEFILE can be changed into a more convenient filename
// USER has to be changed into the login name - in my case: ederveen
int main (int argc, char *argv[]) {
    char* WrFileCl = "nohup date >> TIMEFILE";
    char* WrFileC2 = "nohup ps -u USER >> TIMEFILE";
    char* WrFileC3 = "echo ""**********************************" >> TIMEFILE";
    int RetInt = 1;
    for( unsigned ArgNumber = 1; ArgNumber < argc ; ArgNumber++) {
        if( ArgNumber == 1) RandomInit = 1997;
        if( ArgNumber > 1) RandomInit = Random( (unsigned) 997 + 1);
        ifstream in;
        cout << ArgNumber << 1 << argc << " 
        cout << argv[ ArgNumber] << \
        in.open(argv[ ArgNumber]);
        if (!in) cerr << "cannot open input file" << endl;
        else {
            if( !in.eof()) {
                int Sysi = system( WrFileCl);
                MakeMatrix MM( in);
                in.close( );
                Matrix Total( MM.Do());
                Matrix NewMat = Total.Reduce();
                SupportList SupEs = Total.EssentialList( );
                SupportList SupDo = Total.DominatedList( );
                if( SupEs.Count() + SupDo.Count() < Total.MaxColumn() ) {
                    // SPECIFIC CODE GOES HERE
                }
                else
                    cout << "The solution has been found!\n";
                SupportList Empty( 0);
                cout << "Support: " << Total.Construct( Empty);
            }
        }
        else cerr << "The file " << argv[ ArgNumber] << \
```
cout << "*******************************" << "*******************************\n";
int Sysi = system( WrFileC3);
RetInt *= ( in != 0);
}
return RetInt;

D.2 The Makefile

GNU = /usr/gnu/lib/g++-include
AFLAGS = +U -Aa -W0,-Aa -Wb,-Aa -D__STDC__ -D_HPUX_SOURCE
IFLAGS = -Iinclude
DFLAGS = -D__DEBUG=0
TFLAGS = -ptr./ptrepository/u

CXX = CC
CXXFLAGS = $(AFLAGS) $(IFLAGS) $(DFLAGS) $(FLAGS)

MODULES = obj/bitstr.o \
    obj/covas.o \
    obj/ga.o \
    obj/gaout.o \
    obj/main.o \
    obj/frandom.o \
    obj/matrix.o \
    obj/makematx.o

bin/EXENAME : $(MODULES)
    $(CXX) -0bin/EXENAME $(CXXFLAGS) $(MODULES) /lib/libM.a

obj/bitstr.o: src/bitstr.cxx include/bitstr.h
    $(CXX) -c -o$@ $(CXXFLAGS) src/$(@F:.o=.cxx)

obj/frandom.o: src/frandom.cxx include/frandom.h include/inias.h
    $(CXX) -c -o$@ $(CXXFLAGS) src/$(@F:.o=.cxx)

obj/genetic.o: src/genetic.cxx include/genetic.h
    include/listruct.h obj/frandom.o obj/bitstr.o include/covas.h
    $(CXX) -c -o$@ $(CXXFLAGS) src/$(@F:.o=.cxx)

obj/ga.o: src/ga.cxx include/ga.h
    include/matrix.h include/genetic.h include/frandom.h include/inias.h
    $(CXX) -c -o$@ $(CXXFLAGS) src/$(@F:.o=.cxx)

obj/gaout.o: src/gaout.cxx
    $(CXX) -c -o$@ $(CXXFLAGS) src/$(@F:.o=.cxx)

obj/matrix.o: src/matrix.cxx include/matrix.h
    include/listruct.h include/bitstr.h
$(CXX) -o$@ -c $(CXXFLAGS) src/$(@F:.o=.cxx)

obj/makematx.o: src/makematxcxx include/makematx.h
   include/matrix.h include/bitstr.h
   $(CXX) -o$@ -c $(CXXFLAGS) src/$(@F:.o=.cxx)

obj/covas.o: src/covascxx include/covas.h
   include/matrix.h include/liststruct.h
   $(CXX) -o$@ -c $(CXXFLAGS) src/$(@F:.o=.cxx)

obj/main.o: src/maincxx $(MODULES)
   $(CXX) -o$@ -c $(CXXFLAGS) src/$(@F:.o=.cxx)
Appendix E The specific code for the pure heuristic merge algorithm

//this code has to be inserted into the code described in Appendix D.1
UpperAlleleProb = 1.0;
LowerAlleleProb = 1.0;
GeneticAlgorithm GenAl( NewMat);
GenAl.AdaptSizeOfNew( 0);
GenAl.AdaptCrossProb( 0.0);
GenAl.AdaptMutateProb( 0.0);
GenAl.AdaptHeuristProb( 1.0);
GenAl.AdaptMergeProb( 1.0);

GenAl.StartOverAgain( (unsigned)2);
GenAl.Continue( 1);
SupportList AnsL( NewMat.MaxColumn() );
GenAl.BestOne().Genotype().Support( AnsL);
cout << "Support: " << Total.Construct( AnsL);
cout << "( " << Total.Construct( AnsL).Count() << ")\n";
Appendix F The specific code for the first crossover test

//this code has to be inserted into the code described in Appendix D.1
GeneticAlgorithm GenAl( NewMat, 1 );
GenAl.AdaptSizeOfNew( 0 );
GenAl.AdaptRepairProb( 0.0 );
GenAl.AdaptHeuristProb( 0.0 );
GenAl.AdaptMergeProb( 0.0 );
float MutProb = 0.0;
GenAl.AdaptMutateProb( MutProb );
GenAl.AdaptUnifXProb( 0.5 ); // 'swap' probability for uniform crossover

for( unsigned XP = 0; XP < 3; XP++ ) {
    GenAl.AdaptPointCross( XP );
    if( XP == 0 ) cout << "Uniform crossover\n";
    else cout << XP << " times 2-point crossover\n";

    for( float TestCros = 0.1; TestCros <= 1.0; TestCros += 0.15 ) {
        GenAl.AdaptCrossProb( TestCros );
        cout << "CrossoverProb: " << TestCros << "\n";

        for( unsigned testcount = 0; testcount < 2; testcount++ ) {
            GenAl.StartOverAgain( );
            GenAl.Continue( 60 );

            SupportList AnsL( NewMat.MaxColumn() );
            GenAl.BestOne().Genotype().Support( AnsL );
            cout << "Support: " << Total.Construct( AnsL );
            cout << "\n";
            RandomInit = Random( (unsigned) 997 + 1 );
            Sysi = system( WrFileC2);
            Sysi = system( WrFileC1 );
        }
    }
}
Appendix G The specific code for the first standard genetic algorithm test

//this code has to be inserted into the code described in Appendix D.1
GeneticAlgorithm GenAl( NewMat, 1 );
GenAl.AdaptSizeOfNew( 0 );
GenAl.AdaptRepairProb( 0.0 );
GenAl.AdaptHeuristicProb( 0.0 );
GenAl.AdaptMergeProb( 0.0 );
GenAl.AdaptPointCross( 2 ); //2 times 2-point crossover
float CrossProb = 0.75; //crossover probability 0.75
GenAl.AdaptCrossProb( CrossProb );
GenAl.AdaptMutateRightProb( 0.9 ); //default right-side aspect
float CrossProb = 0.75; //crossover probability 0.75
GenAl.AdaptCrossProb( CrossProb );
GenAl.AdaptMutateRightProb( 0.9 ); //default right-side aspect
float MutProb = 0.000125; //the algorithm starts with multiplying
for( unsigned type = 0; type < 4; type++ ) {
    MutProb *= 10;
    if( type == 3) MutProb = 0.0625;
    /* because the check on applying the crossover operator will be
    * performed before the mutation operator will be called, the
    * probability has to be corrected.
    */
    GenAl.AdaptMutateProb( MutProb/(1 - CrossProb) );
    cout << "Mutation probability: " << MutProb << "\n";
}
for( float NotJustOne = 0.0; NotJustOne <= 1.0; NotJustOne += 0.5 ) {
    GenAl.AdaptMutateSomeProb( NotJustOne);
    cout << "Multi-allele: " << NotJustOne << "\n";
for( unsigned testcount = 0; testcount < 3; testcount++ ) {
    GenAl.AdaptMutateSomeProb( NotJustOne);
    cout << "Support: " << GenAl.BestOne().Support( AnsL );
    cout << "(" << Total.Construct( AnsL ).Count() << ")
";
    RandomInit = Random( (unsigned) 997 + 1); Sysi = system( WrFileC2);
    Sysi = system( WrFileC1);
} }
Appendix H Specific code for the first genetic algorithm with merge

//this code has to be inserted into the code described in Appendix D.1
GeneticAlgorithm GenAl( NewMat, 1 );
GenAl.AdaptSizeOfNew( 0 );
GenAl.AdaptRepairProb( 0.0 );
GenAl.AdaptPointCross( 2 ); //2 times 2-point crossover
float CrossProb = 0.75;  //crossover probability 0.75
GenAl.AdaptCrossProb( CrossProb );
GenAl.AdaptMutateRightProb( 0.9 );  //right-side aspect
GenAl.AdaptMutateSomeProb( 0.9 );  //multi-allele

for( float Hrst = 0.0; Hrst <= 1.0; Hrst += 0.25 ) {
    GenAl.AdaptHeuristProb( Hrst );
    cout << "Heuristics probability: " << Hrst << "\n";
    for( float MrgP = 0.025; MrgP <= 0.1; MrgP *= 2 ) {
        /* The first check on applying an operator will be for the
        * crossover operator, second comes the merge operator and
        * third the mutation operator.
        * Therefore both merge and mutation probabilities have to
        * be corrected.
        */
        GenAl.AdaptMergeProb( MrgP/(1 - CrossProb) );
        GenAl.AdaptMutateProb( 0.05/((1 - MrgP) * ( 1 - CrossProb)) );
        cout << "Merge probability: " << MrgP << "\n";
    }
    for( unsigned testcount = 0; testcount < 3; testcount++) {
        GenAl.StartOverAgain( );
        GenAl.Continue( 60 );

        SupportList AnsL( NewMat.MaxColumn() );
        GenAl.BestOne().Genotype().Support( AnsL);
        cout << "Support: " << Total.Construct( AnsL);
        cout << "(" << Total.Construct( AnsL).Count() << ")\n";
        RandomInit = Random( (unsigned) 997 + 1);
        Sysi = system( WrFileC2);
        Sysi = system( WrFileC1);
    }
}
Appendix I The specific code for the first test on the repair operator

//this code has to be inserted into the code described in Appendix D.1
GeneticAlgorithm GenAl( NewMat, 1 );
GenAl.AdaptSizeOfNew( 0);
GenAl.AdaptPointCross( 2 ); //2 times 2-point crossover
float CrossProb = 0.75; //crossover probability 0.75
GenAl.AdaptCrossProb( CrossProb);
GenAl.AdaptHeuristProb( 1.0); //pure heuristic algorithm
float MrgP = 4.0 / NewMat.MaxColumn(); //relative probability 4.0
MrgP /= (1 - CrossProb); //corrected probability
if( MrgP > 0.75) MrgP = 0.75; //Don't let merge capture the process
GenAl.AdaptMergeProb( MrgP);
GenAl.AdaptMutateProb( 0.05 / ((1 - MrgP) * (1 - CrossProb)) );
GenAl.AdaptMutateRightProb( 0.9 ); //right-side aspect
GenAl.AdaptMutateSomeProb( 0.9 ); //multi-allele

for( float Rep = 0.25; Rep <= 1.0; Rep += 0.25) {
    GenAl.AdaptRepairProb( Rep);
    cout << "Repair probability: " << Rep << "\n";

    for( unsigned testcount = 0; testcount < 3; testcount++) {
        GenAl.StartOverAgain( );
        GenAl.Continue( 60);

        SupportList AnsL( NewMat.MaxColumn() );
        GenAl.BestOne().Genotype().Support( AnsL);
        cout << "Support: " << Total.Construct( AnsL);
        cout << " (" << Total.Construct( AnsL).Count() << ")\n";
        RandomInit = Random( (unsigned) 997 + 1);
        Sysi = system( WrFileC2);
        Sysi = system( WrFileC1);
    }
}
Appendix J: The specific code for the first test on the immigrate aspect

// this code has to be inserted into the code described in Appendix D.1
GeneticAlgorithm GenAl( NewMat, 1 );
GenAl.AdaptPointCross( 2 ); // 2 times 2-point crossover
float CrossProb = 0.75; // crossover probability 0.75
GenAl.AdaptCrossProb( CrossProb);
GenAl.AdaptHeuristProb( 1.0 ); // pure heuristic algorithm
float MrgP = 4.0 / NewMat.MaxColumn(); // relative probability 4.0
MrgP /= (1 - CrossProb); // corrected probability
if( MrgP > 0.75) MrgP = 0.75; // Don't let merge capture the process
GenAl.AdaptMergeProb( MrgP);
GenAl.AdaptMutateProb( 0.05 / ((1 - MrgP) * (1 - CrossProb)));
GenAl.AdaptMutateRightProb( 0.9 ); // right-side aspect
GenAl.AdaptMutateSomeProb( 0.9 ); // multi-allele
GenAl.AdaptRepairProb( 0.0);
for( float Imm = 1/16; Imm <= 1/8; Imm *= 2) {
cout << "Relative size of immigrate aspect: " << Imm << "\n";
    for( int type = 0; type < 9; type++) {
cout << "Immigrate type " << type << "\n";
        for( unsigned testcount = 0; testcount < 3; testcount++) {
GenAl.StartOverAgain( );
        unsigned ImmSize = (unsigned)(Imm * GenAl.Pop().MaxCount());
        for( unsigned GoOn = 0; GoOn < 10; GoOn++) {
            GenAl.Continue( 6);
            if( type == 0 || (type == 1 && GoOn % 2 == 0))
                GenAl.AdaptSizeOfNew( ImmSize);
            if( type == 2 && (GoOn == 2 || GoOn == 3))
                GenAl.AdaptSizeOfNew( ImmSize);
            if( type > 2 && (GoOn == type - 1))
                GenAl.AdaptSizeOfNew( ImmSize);
        }
SupportList AnsL( NewMat.MaxColumn() );
GenAl.BestOne().Genotype().Support( AnsL);
cout << "Support: " << Total.Construct( AnsL);
cout << " (" << Total.Construct( AnsL).Count() << ")
";
RandomInit = Random( (unsigned) 997 + 1);
Sysi = system( WrFileC2);
Sysi = system( WrFileC1);
}
}
Appendix K The specific code for the test on the population size

// this code has to be inserted into the code described in Appendix D.1
GeneticAlgorithm GenAl( NewMat, 1 );
GenAl.AdaptPointCross( 2 ); // 2 times 2-point crossover
float CrossProb = 0.75; // crossover probability 0.75
GenAl.AdaptCrossProb( CrossProb);
GenAl.AdaptHeuristProb( 1.0 ); // pure heuristic algorithm
GenAl.AdaptMutateRightProb( 0.9 ); // right-side aspect
GenAl.AdaptMutateSomeProb( 0.9 ); // multi-allele
GenAl.AdaptRepairProb( 0.0 );

for( float PopSz = 0.5; PopSz <= 1.5; PopSz += 0.25 ) {
    cout << "Relative population size: " << PopSz << "\n";
    float MrgP = (4.0) / ((1 - CrossProb) * (PopSz * NewMat.MaxColumn() / 2 ));
    // the relative merge frequency is corrected on the population size
    if( MrgP > 0.75 ) MrgP = 0.75;
    GenAl.AdaptMergeProb( MrgP);
    float MutProb = 0.05 / ((1 - CrossProb) * (1 - MrgP));
    GenAl.AdaptMutateProb( MutProb );

    for( unsigned testcount = 0; testcount < 3; testcount++ ) {
        GenAl.StartOverAgain( PopSz );
        GenAl.Continue( 60 );

        SupportList AnsL( NewMat.MaxColumn() );
        GenAl.BestOne().Genotype().Support( AnsL);
        cout << "Support: " << Total.Construct( AnsL );
        cout << " ( " << Total.Construct( AnsL ).Count() << " )\n";
        RandomInit = Random( (unsigned) 997 + 1 );
        Sysi = system( WrFileC2 );
        Sysi = system( WrFileC1 );
    }
}
Appendix L The specific code for the repair - merge combination test

//this code has to be inserted into the code described in Appendix D.1
GeneticAlgorithm GenAl( NewMat, 1 ) ;
GenAl.AdaptPointCross( 0 ); //uniform crossover
float CrossProb = 0.65; //crossover probability 0.65
GenAl.AdaptCrossProb( CrossProb);
GenAl.AdaptUnifXProb( 0.5 ); //swap chance 0.5
GenAl.AdaptHeuristProb( 1.0 ); //pure heuristic algorithm
GenAl.AdaptMutateRightProb( 0.7 ); //right-side aspect
GenAl.AdaptMutateSomeProb( 0.9 ); //multi-alleles
GenAl.AdaptSizeOfNew( 0 );

float MrgP = (4.0)/((1 - CrossProb)*(1.5 * NewMat.MaxColumn()/2 ));
if( MrgP > 0.75) MrgP = 0.75;
float MutProbMrg = 0.05/((1 - CrossProb)*(1 - MrgP));
float MutProbNoMrg = 0.05/(1 - CrossProb);
unsigned ImmSize = (unsigned) (0.25 * 1.5 * NewMat.MaxColumn() );

for( unsigned type = 0; type < 5; type++ ) {
    cout << "Processing type " << type << "\n" ;
    for( unsigned testcount = 0; testcount < 3; testcount++ ) {
        if( type == 0 ) {
            GenAl.AdaptRepairProb( 0.1);
            GenAl.AdaptMergeProb( 0.0);
            GenAl.AdaptMutateProb( MutProbNoMrg);
        } else {
            GenAl.AdaptRepairProb( 0.0);
            GenAl.AdaptMergeProb( MrgP);
            GenAl.AdaptMutateProb( MutProbMrg);
        }
        GenAl.StartOverAgain( 1.5f);
        for( unsigned GoOn = 0; GoOn < 10; GoOn++ ) {
            if( GoOn == 1 && type == 0 ) {
                GenAl.AdaptRepairProb( 0.0);
                GenAl.AdaptMergeProb( MrgP);
                GenAl.AdaptMutateProb( MutProbMrg);
            }
            if( GoOn == 3 && type == 3 ) {
                GenAl.AdaptSizeOfNew( ImmSize);
                GenAl.AdaptRepairProb( 0.1);
                GenAl.AdaptMergeProb( 0.0);
                GenAl.AdaptMutateProb( MutProbNoMrg);
            }
            if( GoOn == 6 && ( type == 1 || type == 2 ) ) {
                GenAl.AdaptRepairProb( 0.1);
                GenAl.AdaptMergeProb( 0.0);
                GenAl.AdaptMutateProb( MutProbNoMrg);
                if( type == 2 ) GenAl.AdaptSizeOfNew( ImmSize);
            }
            if( GoOn == 6 && type == 3 ) {
                GenAl.AdaptRepairProb( 0.0);
                GenAl.AdaptMergeProb( MrgP);
            }
        }
    }
}
GenAl.AdaptMutateProb( MutProbMrg);

GenAl.Continue( 6);

SupportList AnsL( NewMat.MaxColumn() );
GenAl.BestOne().Genotype().Support( AnsL);
cout << "Support: " << Total.Construct( AnsL);
cout << " ( " << Total.Construct( AnsL).Count() << " )\n";
RandomInit = Random( (unsigned) 997 + 1);
Sys1 = system( WrFileC2);
Sys2 = system( WrFileC1);
References

[Bac, 94] Bäck, T. and S. Kuri
An Evolutionary Heuristic for the Maximum Independent Set Problem
Proceedings of the 1st IEEE Conference on Evolutionary Computation, Orlando,

An overview of genetic algorithms: Part 1, fundamentals

An overview of genetic algorithms: Part 2, research topics

[Boo, 87] Booker, L.
Improving Search in Genetic Algorithms
In: Genetic Algorithms and Simulated Annealing - Research Notes in Artificial

[Bos, 94] Bosch, W.F.A.
Implementation of an OR-BDD based TPG algorithm for combinational circuits
Master Thesis Report, Section of Digital information systems, Faculty of Electrical
Engineering

[Cou, 95] Coudert, O. and J.C. Madre
New Ideas for Solving Covering Problems
Proceedings of the 32nd Design Automation Conference, San Francisco, CA., 12-16
June 1995.

[Dav, 87] Davis, L. and M. Steenstrup
Genetic Algorithms and Simulated Annealing: An Overview
In: Genetic Algorithms and Simulated Annealing - Research Notes in Artificial

[Eib, 94] Eiben, A.E. et al
Solving Constraint Satisfaction Problems Using Genetic Algorithms
Proceedings of the 1st IEEE Conference on Evolutionary Computation, Orlando,

Genetic Algorithms in Search, Optimization, and Machine Learning
[Gre, 87] Grefenstette, J.J.
Incorporating Problem Specific Knowledge into Genetic Algorithms

[Hua, 94] Huang, Wen-Chih; Cheng-Yan Kao and Jorng-Tzong Horng
A Genetic Algorithm Approach for Set Covering Problems

[Hin, 96] Hinterding, R.H. et al
Self-Adaptive Genetic Algorithm for Numeric Functions
(found on internet http://www.vut.edu.au:80/~rhh/papers.html)

[Hol, 92] Holland, J.H.
Adaptation in natural and artificial systems - An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence

[Hor, 94] Horng, Jorng-Tzong; Cheng-Yan Kao and Baw-Jhiune Liu
A Genetic Algorithm for Database Query Optimization

[Jóź, 95] Jóźwiak, L.
General Decomposition and Its Use in Digital Circuit Synthesis

[Kan, 95] Kanoh, H. et al
Genetic Algorithms for Constraint Satisfaction Problems

Minimal Input Support Problem and Algorithms to Solve It
EUT Report 95-E-289

[Kon, 95b] Konieczny, P.A.
General Decomposition of Sequential Machines - Algorithms and Programs
IVO report
Mathias, K.E. and L.D. Whitley
Transforming the Search Space with Gray Coding

Orvosh, D. and L. Davis
Using a Genetic Algorithm to Optimize Problems with Feasibility Constraints

Sun, Yanfeng and Zhongtuo Wang
The genetic algorithm for 0-1 programming with linear constraints

Thierens, D. and D.E. Goldberg
Elitist Recombination: an integrated selection recombination GA

Pál, K.F.
Genetic algorithm with local optimization
Berlin: Springer-Verlag, 1995