Genetic algorithms for scheduling purposes

Mesman, B.

Award date:
1995

Eindhoven University of Technology
Department of Electrical Engineering
Design Automation Section (ES)

Genetic Algorithms for Scheduling Purposes

By B. Mesman

Master Thesis

performed: March 94 - August 95
by order of Prof. Dr. Ing. J.A.G. Jess
supervised by Ir. M.J.M. Heijligers

Abstract

Genetic algorithms (GAs) are a general purpose heuristic inspired on the principals of evolution and survival of the fittest. Heuristics are viewed from the perspective of the tradeoff between exploration and exploitation. The amount of exploration in a genetic algorithm is determined by the crossover mechanism and by the mechanism of selecting parents for creating a new individual. The schema theorem from [Holl75], the major theoretical result on GAs so far, guarantees good features (*schemas*) of an individual to spread in subsequent generations when they are not disrupted in crossover process. This result has lead researchers to a very exploitative attitude towards GAs. An important extension to the schema theorem by [DeJo92] suffers from ambiguity and errors. This report presents extensions of the ideas behind the schema theorem in two respects. First, a relation is analytically established between the statistics of two subsequent generations of a population. This relation predicts a number of phenomena that have been observed in practice and reported in literature. The relation especially emphasizes the importance of having a varied population, an effect that can be brought about by introducing exploration into the crossover mechanism, contrary to the traditional exploitative attitude. The crossover mechanism we use is uniform crossover because it does not suffer from the problem of *positional bias* and because the amount of exploration can easily be set by one parameter. From analysis it is concluded that this operator works best when set at its explorative extreme.

The second way we have extended the original schema theorem regards a *permutation* encoding of a solution and a *uniform order crossover* operator specifically targeted this encoding. A permutation encoding is often a very convenient way to express typical scheduling decisions in a scheduling problem, and it is with this type of problems in mind that the following research is conducted. The schema theorem is extended by detailed examination of how a schema (a combination of features) is created by crossover. We now take into account all possible ways a schema is created whereas the original schema theorem considers only the survival from one parent. Analyses are performed on the resulting schema theorem and it is again concluded that the parameters of uniform order crossover should be set at its explorative extreme. This suggests the possibility that the GA could perform even better when it is devised even more explorative by alteration of the selection mechanism. Using a *Boltzmann* selection scheme, it is even possible to adjust the amount of exploration each generation to match the statistics in the current population. This is done by updating the Boltzmann parameter in terms of a target variation and the actual variation in the population. Results show that this adaptive approach is promising. The behaviour of uniform order crossover is analyzed further with respect to the occurrence of a local optimum. It is shown how a local optimum can arise as a result of the *partition property* of uniform order crossover. Furthermore it is shown that some specific measures taken to reduce the solution space have no effect other than to increase the probability of encountering a local optimum, and should therefore not be used.

The genetic algorithm is used to tackle the problem of scheduling a *data-flow graph* (DFG) in high-level synthesis, which is an important part of designing a chip-layout from a behavioral description. A scheduling algorithm from [Heij95] is used to turn a permutation into a schedule in a way that matches the characteristics of the crossover operator. Efficiency is improved by computing a *distance-matrix* prior to running the genetic algorithm. For acyclic graphs, this computation is carried out in $O(VE)$ time by a new APLP-algorithm. Results on running the genetic algorithm show it to be superior to a number of different approaches.

Contents

Chapter 1

Introduction

The first step encountered in the translation of a behavioral description of a chip into a chip-layout, is high-level synthesis (HLS). HLS translates the behavioral description into a description of a network of modules and a controller for this network. Within this synthesis trajectory, scheduling is the problem of allocating operations to modules and cycle steps, given precedence constraints represented by a data-flow graph. Because this scheduling problem is NP-hard, exact solving methods are too expensive in terms of computation-time when considering large-size problems. When the requirement of optimality is relaxed however, heuristic methods are able to obtain a reasonable solution in a reasonable amount of time. A genetic algorithm is one of the few heuristics that has been tried successfully on the scheduling problem at the design automation section of Eindhoven University of Technology. There was however not much insight in the underlying mechanisms, and the available theory failed to provide a satisfying explanation. This thesis follows from the discontent with that situation.

Chapter 2

Genetic algorithms

2.1 Introduction

Since long has mankind struggled with the problem of optimizing functions, and one of the flourishing branches within this struggle is called combinatorial optimization. Let $N = \{1, \ldots, n\}$ be a finite collection and let $c = (c_1, \ldots, c_l)$ be an l-vector over N. For $F \subseteq N$, define $c(F) = \sum_{j \in F} c_j$. Suppose we are given a collection of subsets F of N. The *combinatorial optimization* problem is

$$\max\{c(F) : F \in \mathrm{F}\}$$

Note that the function to be maximized is assumed to be linear. A minimization problem can be incorporated in this model by maximizing the negative of the objective function. A binary vector $x = (x_1, \ldots, x_n)$ is used to indicate membership of F for each $j \in N$. A vast amount of problems are modeled in this way, mainly from the area of operations research. Techniques used within the area of combinatorial optimization to tackle such problems typically have **deterministic behaviour** and aim at **finding the optimal value.** A typical algorithm uses subroutines for finding a non-feasible solution by relaxing the integrality constraint and solving a *linear programming* problem. The purpose of this is to derive a lower (upper) bound on the solution, which is used to fix variables at a certain value which otherwise would exceed the bound. Many of the earlier graph-problems were tackled using algorithms that could readily solve large problem-instances using reasonable computational effort. With the rise of the theory of computational complexity (mid 70's) however, it became clear that there is a large amount of problems for which it is believed that no such algorithm exist. Any algorithm pretending to find the optimal solution to such a problem, probably would require an impractical amount of time or resources to solve even a disappointingly small instance of that problem. This initiated (early 80's) the development of *heuristics*. Heuristics derive from the optimality-feature and often from the deterministic-feature of traditional combinatorial optimization techniques, in order to find a reasonable solution with a constraint on the computational effort. Non-deterministic behaviour is used to handle a specific type of trap: the problem of getting stuck at a local suboptimal value. Heuristic algorithms appear as either *tailored* or general. A tailored algorithm has the advantage of having lots of problem specific knowledge incorporated. A general algorithm has the advantage of being applicable to a larger set of problems. It should be noted however that it is often possible to incorporate some problem specific knowledge into a general algorithm as well. A large group of general algorithms are characterized by their so called *local search* methods.

These methods use the notion of *neighbourhood* to iterate from one solution to another. Generally recognized as a local search method is the so called *genetic algorithm*.

2.2 Outline of the genetic algorithm

Nature has proved a fruitful source of inspiration for scientists looking for an 'easy' way out of the immense complexity that comes along with optimally solving certain types of large problems. Perhaps the most popular of the resulting heuristics is simulated annealing [Aart89], which is based on the principles of thermodynamics and has been successfully applied to many problems. Recently, interest in genetic algorithms experienced an explosive growth. Although the idea was already published in 1975 [Holl75], it seems that its popularity was subjected to that of heuristics in general, which has popularized mainly during the last decade. Genetic algorithms rely on the principle of survival of the fittest, and unlike most iteration (local search) methods, maintain a population of individuals (potential solutions) that evolve according to some measure reflecting the quality of a solution. Besides the obvious analogy in nature, different authors [Gold89b], [Hofs79] have suggested another interesting analogy: that of combining ideas to obtain better ones, an ability recognized as a landmark in the development of human intelligence. These analogies may convince the not-so-critical reader in believing that genetic algorithms are wonderful problem-solvers and there's no further theory required for applying them. Judging by the literature, there's a world full of not-so-critical readers. The only effect an analogy should have, however, is to arouse *interest* instead of belief. It is the aim of this thesis to develop some theory that can be of assistance in applying a genetic algorithm on a difficult problem. For the sake of easy reading, the genetics-analogy will be reflected in our terminology. Suppose A is an arbitrary alphabet not containing $*$. Without loss of generality we assume that the symbols in A are numbers.

Def 2.1 (chromosome) *a chromosome is a vector of l symbols from the alphabet A.*

l is called the *length* of the chromosome, and this will usually be a constant. Because most operations in the genetic algorithm work on vectors smaller than a chromosome x, it is convenient to talk about the elements $x[i]$:

Def 2.2 (gene) *a gene is an element of the chromosome.*

The genetic algorithm processes a number of chromosomes at a time. Therefore it is convenient to talk about a collection of chromosomes:

Def 2.3 (population) *a population P is a collection of chromosomes.*

The chromosome is a coded representation of something called a *phenotype*. The coding is done by way of the mapping $pheno$ from the set of chromosomes to the set of phenotypes. The code in the chromosome is also referred to as the *genotype*. In applying genetic algorithms to problem-solving, the phenotype will be a potential solution to the problem. There is some quality measure, called a *score*, assigning a value to each solution. Since all manipulation is done on chromosomes, we will assume the score $s(x)$ to be directly calculated from the chromosome x.

Def 2.4 (score s) *the score s is a mapping from the set of chromosomes to the set of real numbers. The score s(x) of a chromosome x is a numerical measure of the quality of a phenotype corresponding to x.*

In the process of running the genetic algorithm, new chromosomes are created by genetic operators:

Def 2.5 (genetic operator,parents,children,offspring) *a genetic operator is a mapping from the set of p-tuples of chromosomes to the set of c-tuples of chromosomes. The elements of the p-tuple are called parents. The elements of the c-tuple are called children or offspring.*

c and p are characteristics of a specific genetic operator. By far the most important type of operator (and implicant of the mentioned analogies) is called *crossover.*

Def 2.6 (crossover operator) *a crossover operator* cross *is a non-deterministic genetic operator operating on 2 parents and creating either 1 or 2 children by combining some features of the parents.*

Not all genotypes in a population are considered for creating offspring (*mating*), and some will be considered many times. The mechanism controlling this choice is called the *selection scheme.* In nature it takes some time for a new individual to mature to the point that it is able to mate, so in order to mate this individual has to **survive** till this point of maturity. If it survives even longer, it will be able to mate more than once. In this way, the number of offspring 'produced' by a certain individual depends for the larger part on this individuals' ability to survive. In analogy to the principle of 'survival of the fittest', the selection mechanism for a genetic algorithm chooses a parent x on the bases of its score $s(x)$. That's why the score is usually called a *fitness*-function.

Def 2.7 (selection scheme) *a selection scheme is a non-deterministic function sel from the set of chromosomes to the set of real numbers between 0 and 1. sel(x) is the probability of selecting parent x for a genetic operation, and is based on its score s(x).*

The main selection scheme is called *proportionate* selection (also known as roulette wheel selection) and its function is defined by:

$$sel(x) = \frac{s(x)}{\sum_{x' \in P} s(x')}$$

The genetic algorithm presented so far, proceeds by creating new populations iteratively (each of an increasing generation) by applying the crossover mechanism. Suppose x and y are parents, and z is one of their offspring. Most crossover operators have the property that genes conserve their location within the chromosome. That is, $z[i]$ equals either $x[i]$ or $y[i]$, and $x[j]$ and $y[j]$ for any $j \neq i$ have no bearing on $z[i]$. Suppose now, that we have a population with individuals all having an average score and, as a fortunate result of applying the crossover operator, one individual enters the population that has a much higher score. A proportionate selection scheme will consider this individual for mating far more often than all the other individuals in the population. The next generation will be crawling with its offspring, most of them expected to have a high score. In the next generation, an individual not descending from the high-score individual will be hard to find. When the whole population consists of all look-a-likes (descendent from the same chromosome), it is quite probable that there is a gene (with index i) the value of which is shared by **all** chromosomes in the population. So for some g, $x[i] = g$ for all $x \in P$. Because genes are location-conservative when it comes to crossover, the gene having the same value in all chromosomes, will **never** change value in coming generations in any of the chromosomes when crossover is the only genetic operator. Still, the optimal chromosome may carry a different value on that particular gene ($x_{opt}[i] \neq g$), and so the genetic algorithm will never find the optimal solution from that point on. The remedy for this inconvenience is the occasional use of a mutation-operator.

Def 2.8 (mutation operator) *a mutation operator mut is a non-deterministic genetic operator, operating on 1 parent and creating 1 child by arbitrarily changing the value of an arbitrary gene from the parent.*

For clarity, it is common to introduce another, trivial, genetic operator called *reproduction* corresponding to the identity-function. It is part of the set of operators used by a genetic algorithm, because usually an operator has an associated probability of appliance: p_c and p_m for crossover and mutation respectively, and reproduction is used to show what to do when no other genetic operator is applied. The corresponding probability is $p_r = 1 - p_c - p_m$. The genetic algorithm may be formulated in the following way:

(Let P_i be the population at generation i, and let n be the number of individuals in the population)

```
Fill P(0) with n randomly created individuals;
i:=0;
do until (stop criterion is met)
   do while (|P(i+1)|<n)
      choose a genetic operator;
      select the necessary number of parents from P(i);
      apply the operator;
      insert result in P(i+1);
   od
   i:=i+1;
od
```

Lots of variations are possible of course. For example instead of creating a whole new population we could insert a single new individual and throw another one out. The algorithm as presented above is the original one from [Holl75].

Example : Suppose we have a population of 8 chromosomes which are binary vectors of length 5, and the score s is such that the binary vector x is the binary representation of the integer $s(x)$. There are no genetic operators other than crossover, which is defined as follows. The mechanism takes two chromosomes and randomly chooses a crossover-point. The tails of the chromosomes (the parts after the crossover-point) are exchanged between the two chromosomes. Two offspring are created in this way. This process is illustrated in figure 3.1. The successive steps of the genetic algorithm are administrated in table 2.1. The first row represents the randomly generated population of generation 1. The second row denotes the scores corresponding to the individuals from the first row. The third row represents the individuals from the first row that

Table 2.1: An evolving population

generation 1	10011	01010	01110	00110	01011	10001	01101	00011
score	19	10	14	6	11	17	13	3
select	1-0011	1-0001	10-001	01-110	100-11	011-10	01-010	00-110
generation 2	10011	10001	10110	01001	10010	01111	01110	00010
score	19	17	22	9	18	15	14	2
select	1011-0	1001-1	1-0110	0-1111	100-01	100-11	1-0110	0-1110
generation 3	10111	10010	11111	00110	10011	10001	11110	00110
score	23	18	31	6	19	17	30	6

are selected for crossover. The crossover points are denoted by a "-". Note that some of the (below average) individuals from the first generation are not selected, whereas others (above average) are selected twice. Every pair of individuals between two double vertical lines are parent pairs. The pairs directly below the parent pairs are their offspring, which are members of the next generation. The optimal value (all one vector) is found in the third generation.

2.3 The schema-theorem and the building-block hypothesis

Why would a genetic algorithm work as a problem-solver? A disappointing amount of explanation can be found in the literature on genetic algorithms. Fortunately, [Holl75] provided an analysis of his genetic algorithm, which is the subject of this section.

Examination of the genetic algorithm and the example outlined in the previous section reveals some features of the underlying mechanism. Two chromosomes are selected from a population. Because they are selected, we expect these chromosomes to have a high score compared to the remaining chromosomes in the population. Although the selected individuals are no perfect solutions to the problem at hand, there are some *features* that makes them better than the not-selected individuals. When these features are combined, possibly an even better individual would be created. The crossover mechanism blindly combines bits and pieces from both chromosomes and most often the result has no better performance than its parents. Sometimes however, the crossover mechanism will combine exactly those features from the two chromosomes that are responsible for their above-average performance as a solution to the problem. This results in a major step in the development of the population towards a good solution the problem. The algorithm will proceed in this way when the features that are responsible for above-average performance are guaranteed to spread among the population. Only then can we be sure that sooner or later two of these features will be combined by the crossover mechanism to an even better feature. In the following, a good feature will be formalized as a *schema*. The *schema-theorem* guarantees that good features will spread among the population. Furthermore, the suggestion that this will lead the genetic algorithm to a good solution (under specified circumstances) is stated in the *building-block hypothesis*.

Suppose x is a chromosome as defined in section 2.2.

Def 2.9 (schema) *a schema is a vector of l symbols from the alphabet $A \cup \{*\}$.*

A schema and a chromosome have a lot in common. The only notational difference is that a schema may contain the symbol $*$. When k is a schema, and $k[i] \neq *$, then $k[i]$ is called a gene, as is the case with a chromosome. The conceptual difference is that a chromosome is a blueprint for a phenotype, and a schema is a *partial* blueprint for a phenotype. The schema is said to be defined on all i with $k[i] \neq *$. Because we assumed the symbols from the alphabet A to be numbers, a chromosome and a schema have a straightforward geometric interpretation: a chromosome corresponds to a point in R^l and a schema corresponds to a *hyperplain* H_k of dimension k, where the schema is defined on k genes (it has order k). In the text we will identify a schema with its corresponding hyperplane H_k.

Def 2.10 (containment) *A chromosome x is said to contain the schema H_k if x can be obtained from H_k by substituting for the $*$'s symbols from A.*

The geometrical interpretation of containment is that $x \epsilon H_k$. There's a score corresponding to a schema H_k, $s(H_k)$, which is simply the average score of all chromosomes that contain H_k:

$$s(H_k) = \frac{1}{|x \epsilon H_k|} \sum_{x \epsilon H_k} s(x) \tag{2.1}$$

Let P_i denote the population at generation i, and let $E_i[s] = \frac{1}{|P_i|} \sum_{x \in P_i} s(x)$,the average score of the current population (generation i). Suppose we don't do any crossover or mutation on our selected parents, but instead let the selection mechanism choose from the current population P_i the chromosomes for the next population P_{i+1}. Since the schema-theorem explicitly assumes proportionate selection, we expect that a chromosome containing H_k produces on the average $\frac{s(H_k)}{E_i[s]}$ chromosomes for the next population. If the current population P_i contains $N(H_k, i)$ chromosomes with schema H_k, we expect this number for the next generation to be multiplied by the mentioned fraction. So:

$$N(H_k, i+1) = N(H_k, i) \frac{s(H_k)}{E_i[s]} \tag{2.2}$$

Suppose we perform both mutation and crossover. If a chromosome from P_i contains the schema H_k and produces an offspring, either the schema will survive in the offspring, or it will be disrupted with a probability p_{dis}. We now update equation 2.2 by multiplying the right-hand side with $1 - p_{dis}$, and are left with the problem of finding an expression for p_{dis}. The schema H_k can be disrupted by either the mutation operator or the crossover-operator. The probability that a gene is mutated is p_m. The probability that a selected chromosome mates is p_c. In order for H_k to survive under the mechanism of these operators, it is required that the chromosome is **not** mutated on these k genes, and there is no disruption as a consequence of crossover. Let $p_d(H_k)$ denote the probability that H_k is disrupted as a result of the crossover operator. The value of $p_d(H_k)$ depends on the specific crossover operator chosen for the job. The analysis for obtaining an expression for $p_d(H_k)$ is called *disruption*-analysis. In the next chapter we will discuss several crossover operators and execute a disruption-analysis on them.

For the probability of survival, $1 - p_{dis}$, we now derive

$$\begin{aligned} 1 - p_{dis} &= (1 - p_m)^k (1 - p_c p_d(H_k)) \\ &\approx (1 - k p_m)(1 - p_c p_d(H_k)) \\ &= 1 - k p_m - p_c p_d(H_k) + k p_m p_c p_d(H_k) \\ &\approx 1 - k p_m - p_c p_d(H_k) \end{aligned} \tag{2.3}$$

Both approximations are a result of the fact that p_m is chosen to be small. The schema theorem is now obtained by knitting the probability of survival, formula 2.3 to the right-hand side of equation 2.2:

$$N(H_k, i+1) \geq N(H_k, i) \frac{s(H_k)}{E_i[s]} (1 - p_m k - p_c p_d(H_k)) \tag{2.4}$$

Note that if a schema H_k has above average score, and disruption effects are not too severe, the number of chromosomes containing schema H_k in population $i+1$, $N(H_k, i+1)$, is a factor > 1 larger than $N(H_k, i+1)$. On a restricted time scope this implies an exponential (in generation-number) increase in the amount of chromosomes containing H_k. Since this is true for every schema having above average score, we expect that sooner or later two individuals with different above average schemas will combine. The consequence is stated in the building-block hypothesis:

Building Block Hypothesis : If a good solution can be constructed from small (low-order) schemas having above-average score, then the genetic algorithm has a good chance of finding this solution.

The $\geq$ sign in equation 2.4 is a result of the fact that it is also possible for a schema H_k in P_{i+1} to be created from other schemas in P_i. An analysis on this effect is called *recombination analysis*. Dependent on the problem, the terms resulting from a recombination analysis may have a small or a much larger effect than the sole term in equation 2.4. When the probability of creation of a schema is included in the equation, the $\geq$ sign in 2.4 can be replaced by a $=$ sign. However, no analyses are known in the literature which effectively include all recombination-effects. This renders Hollands analysis somewhat superficial, and apparently not much progress has been made since then. The analyses in this section are however the basics of genetic algorithms, and now that they have been explained, we can compare the genetic algorithm with another optimization algorithm in a number of respects.

2.4 Convergence, exploration, and exploitation

The genetic algorithm has been introduced as a general-purpose problem solver. It has been successfully applied to problems in VLSI-design [Chan91], [Mart91], and many other domains of optimization. Whether or not these successes are predictable is a matter of debate, since the theory of GAs has not matured to the point that we are able to guarantee a certain degree of performance, and maybe we never will. This seems to be the most important drawback of the use of a genetic algorithm, and the lack of theory has major consequences to the way genetic algorithms are used in practice. Little is known about the 'right' adjustment of parameters, leading to rather arbitrary choices when implementing the algorithm. It is no surprise that this kind of programming at will, combined with the empirical nature of the type of research conducted, has led to the wildest conclusions and speculations with regard to the mechanisms involved in the evolution of a population of solutions. As a consequence, GAs and GA-research are not taken seriously by many people. To illustrate the lack of theory, we will compare the genetic algorithm with simulated annealing. This particular choice of comparison is motivated by the fact that simulated annealing has a lot in common with genetic algorithms:

- They are both intended as general-purpose problem solvers.
- They are both heuristic.
- They are both inspired on principles from nature.
- They are both probabilistic.

One of the major differences between the two has to do with *convergence*. Although the concept is usually only vaguely described, the following definition is sometimes used in GA-literature:

Def 2.11 (convergence) *The phenomenon that for each gene 95% of the chromosomes in a population have the same value, is called convergence.*

Tightly linked to the concept of convergence is that of a *local optimum*. We denote the set of globally optimal solutions by X_{opt}. A solution is a local optimum with regard to a subspace R' of the solution space R when it has the highest score of all solutions in R'. The set of local optima with regard to R' is denoted by $OPT(R')$. In the following we will also use the concept 'good' solution, meaning a solution having a score close enough to those of X_{opt}: solution y is 'good' iff $s(y) - s(x \in X_opt) \leq \epsilon$. In most optimization-algorithms it is very clear when a local optimum is encountered: No improvement can be made in the current solution given the available operators. (R' is the set of solutions attainable from the current solution by one of the available operators.) When a local optimum is also a global optimum, we are pleased by the result. However, what in practice is usually termed a local optimum, is the annoying case that it is not a global optimum. A local optimum is more difficult to define in connection with a genetic algorithm. This is because a local optimum is always used to refer to a single solution, whereas we also want to refer to a whole population of solutions. We therefore have to distinguish between a local optimum-solution, which is defined as usual with regard to the score function, and a local optimum-population. We say that a population is in a local optimum when it is 'trapped' in a subspace R' of the solution space R not containing a 'good' solution. We say that a population is trapped in subspace R' when a large part of its individuals are samples within R' such that the operators cannot produce a solution outside R' (when fed with parent solutions from R'). A convention easily derived from this condition for a local optimum-population, is that we speak of a local optimum when a large part of the population contains a schema not present in any 'good' solution. This however, is only valid when the operator(s) respect(s) the notion of a schema. With this I mean that whatever the operator(s) tend(s) to pass on from the parents to their children, consists of what we denote by schemas. Note that we didn't demand the whole population to be in the subspace R'. The *probability* of this restricted event would be small enough to leave out of consideration, yet the *effect* of having a large portion in R' is comparable. Some solutions have been proposed in literature for the problem of being trapped in a subspace: stochastic one-parent operators such as mutation and inversion (see section 3.1). However these operators only provide a solution *in principle* to the problem of having your entire population in a subspace. I make this claim because a local optimum-population is not just local, it is also full of locally optimal solutions. The probability that a random search-like operator yields above average individuals (which is a necessity for survival) is quite small. So although mutation and inversion can in principle provide the GA with an individual outside R', it is unlikely that it will survive to the point that it produces offspring itself. We conclude that a population can run into a local optimum in pretty much the same sense as, for example, steepest descent can run into a local optimum-solution.

In simulated annealing, the whole purpose is to achieve convergence towards a good solution (which is a local optimum). Convergence is directed according to a so called *cooling-schedule,* which is slow and regular enough to prevent the algorithm from getting stuck at a local optimum that is not good enough. The simulated annealing algorithm is stopped when the cooling schedule has ended and no improvement can be made within the neighborhood of the current solution. For a genetic algorithm however the best time to stop processing is unclear. Furthermore, the concepts of convergence and local optima are quite controversial, for a number of reasons:

- There is no easy mechanism for the detection of a local optimum in a genetic algorithm. Therefore it is difficult to use as a stop-criterion, and the algorithm will proceed.
- There is no easy mechanism to avoid a local optimum, and when processing continues despite the local optimum, the usage of time and resources is very inefficient, so we should avoid convergence from the perspective of efficiency.
- When there is no convergence at all however, good solutions have no opportunity to penetrate the population, and there is little chance that good solutions combine to better ones. Thus, the total abundance of convergence leads to an impediment of the underlying genetic mechanism, as suggested by the building block hypothesis.

The latter two points imply that a genetic algorithm without inefficient processing can only work when there is a proper balance of convergence. This balance can also be viewed from the perspective of the *exploration-exploitation tradeoff.*
An easy way to explain what this tradeoff is all about, is perhaps by way of the correlation between two subsequent states of a running algorithm. In a genetic algorithm a state is clearly formed by a population of a certain generation, and the 'amount' the next generation depends on the current one indicates where the exploration-exploitation balance lies. In an iterative algorithm one iteration is a step from one state to the other, and the correlation between states is defined by the step size. We say an algorithm is explorative when the correlation between two subsequent states is low. A random search algorithm is the extremum of exploration: there is no correlation at all between two subsequent states. We say an algorithm is exploitative when the correlation between two subsequent states is high. The algorithm literally exploits the result of the previous state to obtain the next one. A steepest descent approach is perhaps the (non-trivial) extremum of an exploitative algorithm. It is iterative in nature, and accepts a minor change in the current solution (state) that gives the best improvement within all possible minor changes.

The basic working of the simulated annealing algorithm is easily explained in terms of exploration and exploitation. In the beginning of the algorithm, it is most explorative. According to the cooling schedule (and its performance), the balance is shifted in time towards exploitation. What is left in the end is a steepest descent algorithm, and it stops when no improvement can be made by a minor change in the solution.

When we consider the genetic algorithm at the level of chromosomes, the concepts of exploration and exploitation are more difficult to grasp than was the case with the simulated annealing algorithm. The idea we have tried to give in the previous section was however that the real interesting processing was done on schemas (or building blocks) which is a very implicit sort of processing. From the perspective of schemas, a clearer picture of exploration and exploitation emerges. Exploitation is taken care of by the selection mechanism, allocating more resources (chromosomes in the next population) to schemas (in the current population) with above average score. Exploration on the other hand is provided by the crossover mechanism, assuming mutation is too infrequent to play a role. In literature, disruption is used as a measure of exploration. This tight connection is justified by the definition (above) of explorative: a low correlation between subsequent states (chromosomes). It is clear that more disruption results in a lower correlation between parent and offspring. We conclude that the amount of exploration is indicated by the amount of disruption occurring during the crossover process. This implies that controlling the amount of disruption is a way to control the balance of exploration and exploitation.

In summary: Genetic algorithms lack theory for both systematic engineering and (average) performance guarantees. It is difficult to avoid local optima without impediment to the implicit mechanism of schema synthesis. Empirical evidence suggests however, that the idea of a genetic algorithm is sound, and thus worth exploiting and analyzing.

2.5 Where to find more on genetic algorithms

For a short overview on genetic algorithms (GAs), the reader is referred to [Beas93a] and [Beas93b]. In the pioneering work [Holl75] the original schema theorem can be found. It also provides lots of thorough mathematical analyses of several problems (mainly the k-armed bandit problem), and examples from both nature and industry. A '92-edition is currently available, updated with topics as classifier systems, and artificial life. A widely used textbook on GAs is [Gold89a]. Common to all general treatments on GAs, it discusses the schema theorem, and the resulting Building-Block hypothesis. A number of examples are taken from [Gref85], to illustrate his Simple GA. The first serious discussion on representation-issues is found in this textbook. It is concluded that using the bitstring-representation, the greatest number of schemas per bit is processed. This argument in favour of the the bitstring-representation became known as the $O(n^3)$-argument. Some mathematical analysis on this argument is provided. In a small appendix the basics of Walsh-transforms are discussed. [Mich92] is a more up to date textbook, containing thorough overviews and an extensive bibliography. It provokes the traditional representation of genotypes by giving empirical evidence in favour of floating-point representations. An entire chapter on the travelling salesman problem is included, acting as a benchmark for even more exotic representations and all kinds of genetic operators. The reasons for the numerous alternatives for traditional GAs are somewhat superficially elucidated. As the title suggests, the book addresses implementational issues, thus aiming at the more practically inclined. Mathematical analysis however, is restricted to the part on the schema-theorem. In [Davi91] an effort is made to systematically instruct the reader how to engineer a genetic algorithm. An interesting paradigm within the framework of genetic algorithms is that of genetic programming. It applies the ideas of GA to programs. [Koza92] is considered the major representative of the genetic programming community. His book consists mainly of applications of genetic LISP-programming.

Chapter 3

Crossover operators for bitstring-representations

In the preceding chapter the concept of genetic algorithms (GAs) has been introduced. The building-block hypothesis was explained to contain the implicit mechanism of genetic algorithms. Central in this mechanism is the use of a *crossover*-mechanism, combining above average individuals in order to obtain even better solutions. The crossover mechanism is the central subject of this chapter. In chapter 2 a chromosome was very generally defined as a vector of symbols over an alphabet. In practice however the alphabet is usually quite restricted. In this chapter only one type of chromosome is treated: the bitstring.

In section 3.1 the basic mechanism that we already encountered in chapter 2, single point crossover, will be reviewed. The schema-theorem will be refined for this operator by specifying the disruption analysis. A possible cure for the major drawback of single-point crossover, inversion, will be explained. In sections 3.2 and 3.3 an intuitive analysis will be given for n-point crossover and parameterized uniform crossover resp. Section 3.5 is a discussion on the tradeoff between exploration and exploitation, which lies at the very core of every general-purpose problem solver. The last section treats, extends, and criticizes the state of the art in disruption-analysis with regard to uniform crossover.

3.1 Single point crossover and inversion

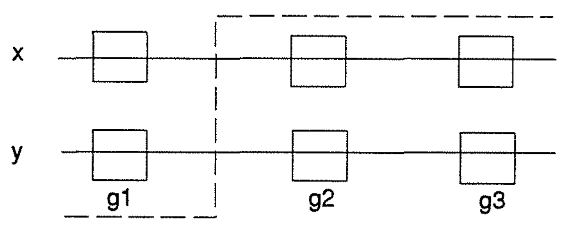

Figure 3.1: Single-point crossover

In single point crossover a *crossover-point* is randomly chosen between 1 and $l-1$, where l is the length of the chromosomes. Each of the parent chromosomes is split in two at this crossover-point, and one of the parts is swapped with the corresponding part from the other parent. Figure 3.1 illustrates single point crossover. In this figure, x and y are the parents, and g1,g2, and g3 are genes that together form a schema H_3 of order 3. The crossover-point is chosen between g1 and g2 with uniform probability, and the genes underneath the dotted line are swapped between parents, to produce the two offspring. As can be seen, the schema H_3 is disrupted as a result of the choice of crossover-point. As the matter a fact, *any* crossover point chosen between g1 and g3 would disrupt H_3, and the probability of disruption, p_d, is just the probability of the crossover-point being chosen between g1 and g3. Actually, p_d is just a bit smaller, because the swapped genes from the second parent coincidentally may be the same as the swapped genes from the schema from the first parent. We now see that this p_d gets higher with increasing distance between the genes of the schema that are furthest apart. This distance is called the *defining-length*, $\delta(H_k)$ of the schema H_k. If the crossover-point is chosen uniformly in the specified interval, the probability that it will fall within the defining length of H_k is just $\frac{\delta(H_k)}{l-1}$, so $p_d = \frac{\delta(H_k)}{l-1}$. In the pioneering work [Holl75] a schema-theorem using single point crossover is devised. Equation 2.4 is a generalization of the original schema theorem with respect to the crossover operator. Substitution of $p_d = \frac{\delta(H_k)}{l-1}$ in the generalized formula yields Hollands original formula:

$$N(H_k, i+1) \geq N(H_k, i)\frac{s(H_k)}{E[s_i]}(1 - p_m k - p_c\frac{\delta(H_k)}{l-1}) \tag{3.1}$$

The disruption analysis could be supplemented by including a recombination analysis and an analysis of the case that the disrupted genes from a schema equal the replacing genes. This is done in [Brid85]

Examination of the value of p_d, the probability of disruption, reveals a disadvantage of single-point crossover: schemas with a low defining length are favored to schemas with a high defining length, despite the fact that they may have the same order. In the previous chapter, we saw some of the implications of the schema theorem summarized in the building-block (BB) hypothesis. It is implied by the hypothesis that high order schemas are created by combining lower order schemas. The ideal mechanism using this principle, may favour schemas of a certain order, but does not distinguish between defining length. This is because schemas with the same order, but deferring on defining length, essentially carry the same amount of information and material, so one should not be favored detrimental to the other. It may even be the case that the schemas with the highest defining length just happen to be the ones with the highest score. The phenomenon, that schemas are favored according to the positions of their genes in the chromosome, is called *positional bias*.

To overcome this disadvantage of single-point crossover, Holland devised a third genetic operator called *inversion*. The mechanism of inversion requires a second representation for each chromosome. The actual phenotype is constructed from representation 1, which is the traditional bitstring. Representation 2 consists of two strings, the first of which is a gene-string, and the second a label-string. The first representation is obtained from the second by mapping each gene from the gene-string according to its corresponding label from the label-string, on a new string. This is illustrated in figure 3.2.

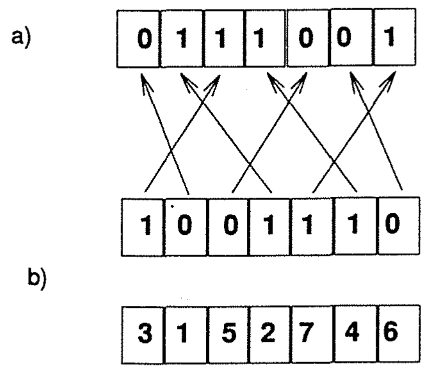

Figure 3.2: a) representation 1, b) representation 2

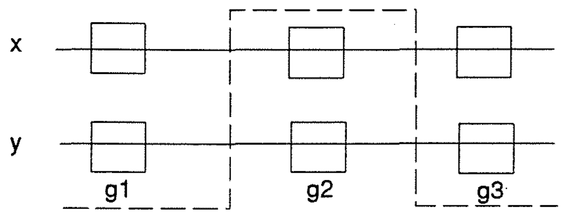

Figure 3.3: 2-point crossover

Now, mutation and crossover are performed only on the gene-string of representation 2, and this has the same effect on the phenotype as with the traditional representation, so there's nothing new here. However, inversion is performed on both strings from representation 2. It randomly chooses two points on the string and the part between these points is turned 180 degrees. Because the same operation is performed on both strings, the effect on the first representation is exactly nil! Inversion does not effect the phenotype. The catch is that inversion changes the way that genes are grouped in representation 2. Two genes, each on one extreme of the chromosome, may end up being neighbors! So, inversion severely affects the defining length of most schemas. Schemas of equal order now have equal probability on getting a favorable defining length for the next number of generations. Inversion is however, a long way from a GA without positional bias, and this bias is a major concern in the GA-community. In the next section, we will see the disruptive effect of positional bias when more than one crossover-point crossover is used.

3.2 n-point crossover

In the same year Hollands pioneering work was published, De Jong finished his doctoral thesis [DeJo75] on some variants of the traditional GA. These variants were GAs using n-point crossover. As the name suggests, n crossover points are chosen randomly instead of one. Figure 3.3 illustrates this idea for n=2.

Again, the genes underneath the dotted line are swapped between parents to produce the two offspring. It is clear that the disruptive properties of this operator are very different from single-point crossover. For instance, despite the presence of crossover points between g1 and g3 in figure 3.3, these genes are not separated by the operator. It is no longer true that a very high defining length guarantees disruption of the schema. As can easily be seen, any two genes in the chromosome will stay together in the offspring whenever there is an even number of crossover points between them. This idea was the basis for the disruption analysis in [DeJo75].

In [DeJo92] this disruption analysis is extended to schemas with order higher than 2. The paper also includes an analysis of uniform crossover which will be treated in section 3.6. The conclusion from [DeJo92], with regard to n-point crossover, is summarized as follows. As far as minimizing disruption, 2-point crossover gives the highest performance. With increasing order of the schema in question however, the difference in character between several n-point crossovers with regard to defining length, quickly fades. This helps explain the results of [Eshe89], in which the same conclusion was drawn from experiments from a perspective of positional and distributional bias. It should be noted that until about 1992, most people thought that minimization of the disruption was the way to an optimal genetic algorithm. In this context, it is not surprising that 2-point crossover was thought to perform relatively well. It still suffered from positional bias, tough.

In the next section, a crossover-operator is introduced that has no positional bias. It also shocked the GA-community in their belief that minimizing disruption should be a goal when designing a genetic algorithm.

3.3 parameterized uniform crossover

In [Sysw89] a new type of crossover operator, uniform crossover, has been introduced which provoked the traditional genetic operators with good experimental results. This operator made the drawback of n-point crossover especially clear, because uniform crossover does not suffer from a positional bias. The idea was to just randomly pick some genes from the first parent with a probability p_o and from the second parent with a probability $1 - p_o$. In the sequel of this report the parameter p_0 will be called r, and is referred to as the disruption parameter for reasons that will become clear later. The random choices made by this crossover scheme can be administrated using a mask h. Each of the bits in this 'mask' h, has a probability r of pointing at x. The first offspring is produced by taking the genes from the first parent according to the x in h. The remaining genes are taken from the second parent. This process is repeated with the complemented mask h_c to produce the second offspring. The process is illustrated in 3.4. In this figure, $o1$ and $o2$ are the first and second offspring resp.

That uniform crossover has several advantages over n-point crossover even convinced the main researchers in n-point crossover [Spea91]. Important virtues of uniform crossover are:

- There is no positional bias, so the disruption does not depend on the defining length of a schema, which reduces the representation-effects to nil.
- The disruptive potential is easily controlled by one parameter.
- This parameter can have a continuous value, so the crossover mechanism can be controlled to the highest level of refinement. This stands in sharp contrast to the discrete values of the number of points in n-point crossover.

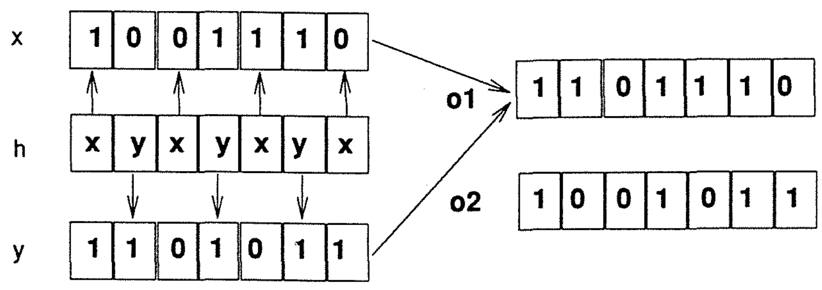

Figure 3.4: uniform crossover

One aspect of uniform crossover however, is a source of ambivalence in its interpretation. The operator is much more disruptive than n-point crossover, which is either a virtue, or a drawback. This disruptive potential will be clear after the analysis in the next section.

3.4 Introduction to disruption-analyses

This section serves two purposes. In the first place, it is intended as a practice for the reader to get a feeling of how disruption-analyses are performed. The result will be rather trivial, but strong enough for the second purpose, a demonstration of the disruptive potential of uniform crossover.
Suppose we have have a building-block H_k with an average score that is a times the average score of the population, $E[s_i]$, so $s(H_k) = aE[s_i]$. Furthermore we have a population P_i with cardinality n, containing one representative instance x of H_k, with $s(x) = s(H_k)$. We want to express the number of chromosomes containing schema H_k in population $i+1$, $N(H_k, i+1)$, in terms of $N(H_k, i)$.
Assuming proportionate selection, the probability that x is chosen for mating is $\frac{s(x)}{\sum_{x' \in P_i} s(x')} = \frac{aE[s_i]}{nE[s_i]} = \frac{a}{n}$, and the expected number of times x is selected for mating is $n\frac{a}{n} = a$, which is by the way independent of the population size. To calculate the number of chromosomes containing H_k in the next generation, we use formula 2.4 and multiply a with $(1 - p_d(H_k))$, assuming $p_m = 0$ and $p_c = 1$. For simplicity, we ignore the fact that H_k may be formed by recombination, and demand that H_k survives by complete replication. This implies that in the mask h all bits corresponding to H_k have to be set, so the number of set bits in h has to be at least as large as the order of H_k.
So assuming $i \geq k$ bits will be set in the mask h, what is the probability that H_k is left completely intact by the mask? There are $\binom{l}{i}$ ways to choose i bits from l. When k of these bits must be set for survival of H_k, $l-k$ bits remain, $i-k$ of which have to be set to complete the i set bits. This can be done in $\binom{l-k}{i-k}$ ways. If we let p_i denote the probability that i bits will be set in the mask h, the probability of survival is

$$1 - p_d(H_k) = \sum_{i=k}^{l} \frac{\binom{l-k}{i-k}}{\binom{l}{i}} p_i \tag{3.2}$$

There are several possibilities for choosing p_i. With an adaptive GA it is preferred that p_i can be changed along the way, so there's more anticipation on the order of the building blocks in process. However, it is not easy to design a scheme starting with this parameter. We stick to the scheme of uniform crossover shown in this section, and note that the stochast i is a Bernoulli-process repeated l times. This results in a binomial distribution: $p_i = \binom{l}{i} r^i (1-r)^{l-i}$. Substitution in 3.2 yields:

$$\begin{aligned}
1 - p_d(H_k) &= \sum_{i=k}^{l} \binom{l-k}{i-k} r^i (1-r)^{l-i} \\
&= \sum_{i=0}^{l-k} \binom{l-k}{i} r^{i+k} (1-r)^{l-i-k} \\
&= r^k \sum_{i=0}^{l-k} \binom{l-k}{i} r^i (1-r)^{l-k-i} \\
&= r^k (r + (1-r))^{l-k} \\
&= r^k
\end{aligned} \tag{3.3}$$

As I already mentioned, the result is rather trivial: in order for H_k to survive, k bits have to be set in the mask at the corresponding loci. The probability of this happening is just r^k. The above derivation was actually my first own analysis on this subject. Imagine my surprise when I saw the result. There is another reason why a recombination analysis is omitted: it is almost exactly the same as the disruption analysis itself. Instead of a building block from one parent, a building block distributed over two parents has to survive. As a result of the deletion of the recombination analysis and a tolerance towards imprecisions, the resulting schema-inequality 3.4 is extremely comprehensive.

$$\begin{aligned}
N(H_k, i+1) &\geq N(H_k, i) \frac{s(H_k)}{E[s_i]} (1 - p_m k - p_c p_d(H_k)) \\
&= N(H_k, i) a r^k
\end{aligned} \tag{3.4}$$

It is immediately clear how disruptive uniform crossover is: the probability of survival of schema H_k decreases exponentially with the order k. This means that in order to survive, a has to increase exponentially in k. Remember that $a = \frac{s(H_k)}{E[s_i]}$, so the score of the building block of order k has to be exponentially (in k) higher than the average of the population! When considering single-point crossover, the probability of disruption is only reversed proportionate with the defining length, so uniform crossover is much more disruptive than single-point crossover, especially for large schemas. The contrast with 2-point crossover is even stronger, because 2-point crossover is less disruptive than single-point crossover. Furthermore, the recombination analysis leads to a similar expression as 3.4, so the recombination or *exploration* potential of uniform crossover has the same contrast to n-point crossover. As mentioned in section 2.4, there is a very tight connection between disruption and both exploitation and exploration, and the balance between the latter two is drastically different in uniform crossover compared to any n-point crossover.

3.5 Controlling the exploration-exploitation tradeoff

Because uniform crossover disrupts more schemas than n-point crossover, it exploits less information on building blocks. Because it opens the way to even better schemas, it explores more of the search space. Which is better, exploration or exploitation? Where lies the ideal balance between these two?

These have been questions raised from the moment GAs were born, and nobody has ever been able to shed light on the issue. Furthermore, reported empirical results are contradictory. [Sysw89] concluded from his experiments that exploration was more desirable, arguing in favour of *his* uniform crossover. Others concluded otherwise. Adaptive strategies like [Spea92] let the GA (in this case: each chromosome) decide for itself which of two crossover operators should be used on them. The conclusion from this paper was that the preference for uniform crossover faded during search. This conclusion stands in contrast to the general consensus that more exploration is desired towards the end of a run, when the population converges and becomes more homogeneous. This consensus on its turn stands in contrast to more traditional methods such as simulated annealing, were large stochastic variations are allowed early in the search process, whereas no stochastic variations are allowed near the end of search. The contrast can be explained by the stop-criterion: traditional iterative methods stop when 'there is no direct improvement possible from the current state', and need zero stochastic variations in order to settle. GAs on the other hand are supplied with a stop criterion such as a restriction on the number of generations. Any processing on a converged homogeneous population is a waste of resources, and should be avoided using large stochastic variations.

Some results have been reported concerning the optimal disruption when coding scheme and population size are varied. The dependence of disruption on representation and the specific problem is made clear in [Huli91]. In this paper a method is given to calculate a suitability-value of a coding scheme for a certain problem. Population size is considered in [DeJo90]. Here, empirical evidence shows a preference for uniform crossover when using a small population size, and a preference for 2-point crossover when using a large population size. The authors explain this result by arguing that large stochastic variation is needed to deal with the stochastic imprecisions when a small population size is involved. The 'stochastic imprecisions' arise when the GA has to estimate the average score of a schema on the basis of just a few individuals carrying the schema.

In section 4.3 I will demonstrate a new result (formula 4.4) showing that improvement in average score over subsequent generations, is linearly proportional to the variation in score. However, a high variation in score does not necessarily imply a need for large stochastic variations in search: when interactions of high order are involved in a problem, schemas of high order are involved in the solution, and these can only be obtained and maintained in a population when disruption is not to severe on high order schemas. Under these circumstances, the highest scores (and thus the highest variation in score) can only be reached when the stochastic variation in search is not too large! This conclusion seems to contrast equation 4.4 at first sight. They are consistent however, when we assume that the desired disruption actually *changes* during search. The proper balance between exploration and exploitation depends on the order of schemas involved. This in turn, depends on the particular type of problem, and the progress made by the evolutionary process so far. The idea of a *dynamical* rather than constant level of disruption is supported in [DeJo92]. In this paper it is repeatedly argued that there is a "need for different levels of disruption at different points in the evolutionary process", suggesting the use of an adaptive crossover mechanism to even further balance exploration and exploitation.

At this point, the advantage of uniform crossover of being able to vary disruption by use of a single parameter, is made particularly clear. However, in order to use this advantage we should have a pretty good understanding of the relation between disruption and the parameter controlling the uniform crossover mechanism. In other words, we need an exact disruption-analysis.

3.6 An extensive disruption-analysis

The most extensive disruption analysis on uniform crossover so far, can be found in [DeJo92]. It discerns from previous analyses by taking *overlap* into account. Overlap is the phenomenon that parents have one or more genes in common. The probability that two genes (at location d) share their value is indicated by a parameter $p_{eq}(d)$. As mentioned in section 3.2, the population becomes more homogeneous and converges when p_{eq} rises, so that p_{eq} can be used to reflect the amount of convergence of the population. In summary, the idea in [DeJo92] is as follows. Suppose a subset $I \subset K$ (K is a schema) is chosen for crossover. The probability of this occurrence is $p_c(I) = r^{|I|}(1-r)^{|K/I|}$. Let $p_{s,1}(K)$, $p_{s,2}(K)$, and $p_s(K)$ denote the probability that the first, second, or any child contains K. The probability that the first child contains K is the probability that the *not*-chosen bits K/I correspond in both parents, so

$$p_{s,1}(K) = \prod_{d \in K/I} p_{eq}(d)$$

The second child is constructed using the complemented mask, so the schema-genes from the first parent are now K/I. The probability that the second child contains K is the probability that the not-chosen bits I correspond in both parents, so

$$p_{s,2}(K) = \prod_{d \in I} p_{eq}(d)$$

The probability of survival is obtained by adding these terms. One case however, is covered by both terms: both offspring contain K. The probability of this occurrence is subtracted from $p_s(K)$ in [DeJo92], because it is counted twice otherwise. When we are interested in the expected number of survivors, this case should be counted twice, and the subtraction can be omitted. So the probability that a schema K survives is

$$p_s(K) = \sum_{I \in K} r^{|I|}(1-r)^{|K/I|}\{\prod_{d \in I} p_{eq}(d) + \prod_{d \in K/I} p_{eq}(d)\} \tag{3.5}$$

To simplify, we assume that $p_{eq}(d)$ is the same for all d, so let $p_{eq}(d) = p_{eq}$ for all d. We now write i and k for $|I|$ and $|K|$, and note that the probability of survival, $p_s(K)$, depends only on k, the cardinality of K. Because there are $\binom{k}{i}$ subsets $I \in K$ equation 3.5 can be rewritten as

$$p_s(k) = \sum_{i=0}^{k} \binom{k}{i} r^i (1-r)^{k-i}\{p_{eq}^i + p_{eq}^{k-i}\} \tag{3.6}$$

This is as far as 3.6 goes, and the equation is still not very transparent. Furthermore, when we want to dynamically determine the optimal level of disruption at every point of the evolutionary process (as suggested in section 3.5), we should at least be able to evaluate expression 3.6 very quickly. We cannot do that when we have to evaluate a summation. The authors did not notice however, that 3.6 can be turned into a form, which is very easy to comprehend. An easy manipulation from my part did the job:

$$\begin{aligned}
p_s(k) &= \sum_{i=0}^{k} \binom{k}{i} r^i (1-r)^{k-i} \{p_{eq}^i + p_{eq}^{k-i}\} \\
&= \sum_{i=0}^{k} \binom{k}{i} (p_{eq} r)^i (1-r)^{k-i} + \sum_{i=0}^{k} \binom{k}{i} r^i (p_{eq}(1-r))^{k-i} \\
&= (p_{eq} r + (1-r))^k + (r + p_{eq}(1-r))^k \\
&= (1 - r(1-p_{eq}))^k + (p_{eq} + r(1-p_{eq}))^k \qquad (3.7)
\end{aligned}$$

This expression is easy enough to evaluate, so that an adaptive GA could easily determine a (close to) optimal value for r under stated restrictions. These restrictions include among other things, being able to estimate the schema-order k that is to be stimulated at the current point of the search process. A critical look however, reveals that 3.6 may not be a very valuable tool. Like every other disruption-analysis within the framework of the original schema theorem 2.4, it is merely a *survival*-probability. It was already remarked in chapter 2 that this may be insufficient for a large number of problems. Furthermore, there are some problems with the interpretation of the parameter p_{eq} as well as it's effective use in the schema equation. p_{eq} reflects the amount of convergence of the population. To what? It could be to K, but it is stated more general. So, suppose the population is not converging to K, then the interpretation of p_{eq} is the same, but it's potential use is different. The genes in K that differ from the value that the population is converging to, have no longer a probability p_{eq} of being equal in different chromosomes, but a probability of $1 - p_{eq}$! But in the analysis it is taken to be p_{eq}, so the calculated probability of survival is totally wrong, if the schema K is a lot different from the one the population is converging to. It would have been correct if p_{eq} reflected the convergence to K, but if this were the case than there would be two measurements of the same thing in one equation. This is because in the schema-equation 2.4 the number of chromosomes carrying K, $N(H_k, i)$, is also a reflection of the amount of convergence to K. Another point is that the reader may get the idea that p_{eq} is the fraction of the population that carries the gene-value the population is converging to, which is either p_1 or p_0. Since p_{eq} is defined independent from the gene value, it is an average value:

$$\begin{aligned}
p_{eq} &= \sum_i p(y = i | x = i) p(x = i) \\
&= p_1 p_1 + p_0 p_0 \\
&= p_1^2 + (1-p_1)^2 \\
&= p_0^2 + (1-p_0)^2
\end{aligned}$$

So p_{eq} is not the same as the fraction p_1 or p_0. The critical reader of [DeJo92] may have noticed this and treat p_{eq} correctly in this respect, so it is not incorrect but only deceiving. The last objection is, like the first one, an objection to all disruption-analyses within the framework of the original schema theorem: it assumes that the second parent is selected randomly, whereas all parents are selected with a probability proportionate to their score.

The conclusion from these objections is that the disruption-analysis in [DeJo92] is unreliable in most cases. Since this is the most extensive analysis up to this writing, it seems we have to devise our own analysis if we want to make reliable engineering decisions for the design of a genetic algorithm, based on such an analysis. The following two chapters are largely devoted to serve that purpose.

Chapter 4

GA-statistics

In the preceding chapters we made a clear choice for parameterized uniform crossover as our main genetic operator. It was shown that this operator, unlike n-point crossover, does not favour schemas which are grouped in a certain way within the genotype, so it has no positional bias. This has the advantage that schemas of a specified order are treated indifferently by the crossover-mechanism. Furthermore, the level of disruption can easily be adjusted with the bit-mask probability r, according to our wishes regarding the bias on schema-order. In this chapter we will calculate the dependence on r of some statistical values of the score within the population, and we will use this information to see whether there is an optimal value for r matched to the statistics of the current population in order to obtain maximum efficiency of the GA.

An overview of earlier work on adaptive GAs is given in section 4.1. In section 4.2 we will express the score-statistics from population P_{i+1} in terms of the score-statistics of an offspring given its parents x and y from population P_i. It is also shown that this expression remains the same under the assumption that the crossover-operator will produce 2 offspring (the second with a complemented mask) instead of one. In section 4.3 an expression is derived for the expected score in the $(i+1)$-th generation in terms of the expected score and its variance of the i-th generation, using a non-standard but reasonable assumption. The implications of this expression and its assumption will be discussed. In section 4.4 a simple linear problem will be analyzed using the expressions derived in previous sections.

4.1 Adaptive GAs

To enhance a GAs potential to find good solutions, GAs have been proposed which adapt some aspect of the GA-mechanism to either the search process or search space. This section gives an overview of the ideas found in the literature concerning adaptation in a GA. We will take a critical look at these approaches when they could serve our purpose, and we will learn from it how not to design an adaptive GA. The types of adaption encountered in the literature cover most aspects mentioned in the preceding chapter to have some influence on GA-performance. Since we are interested in an adaptive crossover operator, this section will treat a few influential ideas found in the literature covering crossover aspects. The larger part of the literature on genetic algorithms concentrates on either the schema-theorem or the optimal parameter settings for operator probabilities. It is no surprise then, that the aspects of GAs most researchers found suited for adaptation, are the operator probabilities. In a so-called *off line*-approach to adaptive GAs, [Gref86] tried to find the optimal operator probabilities as a global result. He coded the parameter-settings in a bitstring, and let a *meta*-GA optimize the parameters! A chromosome consisted of coded parameter-values, and the score of a certain parameter setting was simply the performance of a GA using these settings. We are however more interested in the *online* approach, were a GA during its own search is adapted to the current state in the search-process. Online approaches to adapting the operator-probabilities repeatedly turn up in the literature, most of them global in the sense that a parameter setting used at some point in the search process, is applied to all members in the population. The first report on applying the online approach (globally) to a GA with adaptive operator-probabilities, was [Davi87]. He used a set of 5 crossover operators with different characteristics. The probabilities of applying these operators were altered by a credit-assignment mechanism much like Hollands' bucket-brigade algorithm [Holl85]. In the approach of [Davi87], operators were given credit according to the performance of the offspring resulting from the use of this operator. [Davi87] took cleverly into account the interactive performance of all operators. This interaction arises when an offspring produced by operator a, itself does not perform well, but potentially produces high-performance offspring when processed by operator b. The credit-assignment mechanism not only increased the probability of an operator producing high-performance individual x, but also of the operator responsible for x's *parents*. In the previous chapter we have already looked at different operators and choose one single crossover-operator, which is flexible enough to provide us with a wide choice of tradeoffs between exploration and exploitation. It is of course possible to apply the approach in [Davi87] by regarding the crossover mechanism using different parameter settings as different crossover mechanisms. This approach is useful when there is very little knowledge of how to adapt the parameter setting to the structure of the problem and of the suitability of the different operators to this structure and the representation of solutions. However, it is our hope to deduce some of this knowledge in this thesis, and it would be efficient to use this knowledge in the adaptive mechanism.

Other ideas apply the adaptation to the crossover mechanism itself. In [Scha87], both the number and loci of crossover points is adapted. This is done locally for each individual chromosome by tagging each gene. The tag specifies whether or not the space to the right of the corresponding gene is a crossover point. When the n-point crossover is performed, the tags stay with their genes. An interesting consequence of this method is the possibility to observe as the search progresses, the GAs preference in the exploration versus exploitation tradeoff. The statistics, administrated during search, indicated an increasing total number of crossover points in the population. This implies that the GA prefers more disruption as the search progresses. In section 2.4 the tight connection between disruption and exploration was established, which implies a shift towards more exploration during the search process of the GA. This is in accordance with general consensus.

The last interesting idea we treat here, is the one found in [Spea92]. As in [Scha87], the adaptation is done locally for each individual chromosome. The difference is that Spears actually knits an extra bit at the end of each chromosome. indicating the type of crossover that is to be performed on this particular individual. He lets his chromosomes choose between two crossover operators: 2-point, and uniform, corresponding to a 0 and a 1 resp. When 2 individuals selected for mating did not have corresponding values of the last bit, a crossover operator was chosen with a 50% chance. At this point I already need to make a critical remark: Because the 2-point crossover has a positional bias, a reset last bit (corresponding to 2-point crossover) will always

be grouped with the last couple of bits in the chromosome, which may have its effect on the performance.
With the method in [Spea92] it was also possible to observe the GAs preference as the search progresses. The statistics showed that the 50%-50% distribution at the beginning of the search gradually transformed into a 80%-20% distribution at the end of the search in favor of 2-point crossover. This may suggest that 2-point crossover is generally better than uniform crossover. It also contrasts the general consensus that more exploration is desired towards the end of a run, when the population converges and becomes more homogeneous. No explanation for this surprising result was given in the paper, but it is obvious that Spears simply overlooked the interaction between the crossover operators, which was carefully dealt with in [Davi87] mentioned above. The interaction playing a role in this process is as follows. Uniform crossover, with its exploitative character, produces offspring with a large variety of scores. 2-point crossover on the other hand, conservatively produces offspring which do not radically differ in score. When selection takes place on the resulting population, say two third of the individuals produced with uniform crossover will *not* be selected, because they have the worst score of the whole population. The consequence is that the individuals carrying the 0-bit (2-point crossover) will dominate the next population. This is an indirect result of the different strategies of 2-point and uniform crossover, and a direct result of letting each chromosomes choose for itself. We now see that the choice of local adaptation in this case traps the uniform crossover operator. This being a wise lesson, we choose to take a global approach on our adaptive GA. The adaptation is with respect to the bit-mask probability r, and each r will be fixed for one generation. That doesn't mean that crossover will be identical for each mating: r is a stochastic parameter, so uniform crossover will probably be different each time it is used. The point is that using this method, the crossover-bias in [Spea92] is completely eliminated, for the individual chromosomes have no vote in designing the mask used for crossover. The value of r will be set dependent on the statistics of the score on the current population. How this dependence can be maximally exploited is the topic of this chapter.

4.2 Moments of score

How does the performance of a genetic algorithm depend on the bit-mask probability r ? And when this probability is changed each generation ? In order to shed some light on this issue, we need to relax the question to one that may be within our capacity to analyze: How does r influence the relation between the population of the i-the generation, P_i, and P_{i+1} ? because the first two (more difficult) questions involve performance, we will be mainly interested in the relation between P_i and P_{i+1} restricted to the scores within these populations. Thus we have to find a way to characterize the scores in a population. One way is by calculating the average score in the population. A better characterization can be found when we also consider variation of the scores in the population. These values are calculated using resp the first and second order *moments* of the scores in the population. The m-th order moment of a discrete stochast x is defined as $E[x^m] = \sum x^m p(x)$. It is the expected value (over a distribution p) of the m-th power of x. The reader may have noticed that there is no probability-distribution specified when we consider an existing population of individuals. Instead of a probability $p(s)$ of having an individual with score s we will consider the *fraction* $f(s)$ of individuals in the population having score s. This is obvious, since we do not wish to characterize a probability distribution, but the scores of individuals in a population. However, if the $i+1$-th population does not yet exist, we can only *estimate* its score-distribution. The best possible estimate on this distribution is the plain old probability distribution of the score, *given that all individuals in P_{i+1} are made from individuals in P_i*! So from this point on we assume that P_i is an existent population characterized by score-moments on fractions, and that P_{i+1} is a non-existent population characterized by score-moments on probabilities that are conditional on P_i in some way. From probability theory we know that having the values of an increasing number of moments of a probability distribution, we have an increasingly better characterization of this distribution. So the idea rises that we characterize the scores in populations P_i and P_{i+1} by as many score-moments as possible, and then try to relate the P_{i+1}-moments to the P_i-moments by way of the bit-mask probability r. This will be the goal for this and following sections.

The initial assumptions are that exactly two parents are selected for producing one offspring, and that our population is large enough so that the score distribution of P_{i+1} can be reliably estimated by the conditional probability distribution of the score. There are no assumptions on the type of crossover or the selection scheme. We start with an expression for the m-th score-moment of P_{i+1}:

$$E_{i+1}[s^m] = \sum_s p_{i+1}(s)s^m \tag{4.1}$$

where E_{i+1} and p_{i+1} denote the expectation and probability when observing the $i+1$-th population, P_{i+1}. As I said, p_{i+1} is a probability distribution, conditional on P_i in some way:

$$p_{i+1}(s) = \sum_{x,y \epsilon P_i} p(x,y)p(s(cros(x,y)) = s|x,y) \tag{4.2}$$

In this equation $p(x,y)$ is the probability that x and y will be chosen from the population P_i to mate. $p(s(cros(x,y)) = s|x,y)$ is the probability that x and y will produce an offspring with score s. In appendix A.1.1 the following equation is derived from 4.1 and 4.2:

$$E_{i+1}[s^m] = \sum_{x,y \epsilon P_i} p(x,y)E[s^m(cros(x,y))|x,y] \tag{4.3}$$

Here $E[s^m(cros(x,y))|x,y]$ is the m-th moment of the score s of the offspring given the parents x and y from the i-th generation, using the probabilities $p(cros(x,y) = x'|x,y)$. Given two individuals x and y, how $cros(x,y)$ can be made from individuals x and y depends only on the crossover-positions selected by the crossover-operator. Since these positions are selected at random (either uniform or by some n-points crossover) $p(cros(x,y)|x,y)$ describes a probability distribution on the crossover sites. So using very general assumptions on the GA, we have reduced the problem of finding the m-th moment of the score in a population to the problem of calculating the statistics of the crossover sites. The reduction in complexity becomes apparent when realizing the enormous difference between the number of possible populations (exponential in both the population size $|P|$ and the string length l) and the number of possible crossover sites (at most exponential in l). It states *at most* exponential in l, because using n-points crossover the complexity is only $O(l^n)$.

What if the crossover mechanism makes two children instead of one? This case is examined in section A.1.2. We conclude that it makes no difference for the score-moments to produce one or two offspring from two selected parents. In the past, variations on standard GAs have been tried, which produce only one offspring. The idea was that more stochastic variation was obtained from selecting more parents. Indeed, to produce the same of individuals in coming generations, twice the number of selections have to be made. We now can conclude that this has no effect on any moment of the score (and thus has no use), provided that the population is large enough. If the variation has had any effect in the past, then it should be due to the violation of this assumption.

4.3 The expected score

In the previous section we have reduced the problem of finding the m-th moment of the score in a population to the problem of calculating the statistics of the crossover sites. In this section we collect the fruits of that work by relating the first order statistics (the expected score) of two subsequent populations. This provides us with the knowledge of entities that influence the progress of the genetic algorithm. We will make a somewhat unusual assumption, which none the less is a reasonable description of the GAs mechanism in practice. We will assume that for all chosen parents, the sum of their scores equals the sum of the scores of the resulting offspring. Stated this way, the assumption places a heavy burden on the credibility of the analysis, but it only has to be valid on the average of all chosen parents. For all practical purposes, this is quite a useful approximation. The argument is that when combining genotypes, we take some of the material from both individuals for one offspring, and leave the rest of the material for the other. If we took the good material from one or both parents for the first offspring, we take the garbage for the second. This is not true in general, because of the numerous possible interactions among genes. Sometimes we will create two very good offspring, and sometimes we will create only garbage. The point is that they will approximately average out when taken over all chosen parents and the population is large enough, so the assumption yields a credible approximation on the average. The assumption may suggest that there will be no improvement in subsequent generations, because the total score remains the same. Of course the total score does not remain the same, because we only select the best individuals for producing the new population.
The second extra assumption concerns the selection scheme. We assume the most widely used scheme of proportionate selection. This means that an individual will be chosen for mating with a probability proportional to its score. In appendix A.1.3 the following result is derived:

$$E_{i+1}[s] = E_i[s] + \frac{\mathrm{var}_i[s]}{E_i[s]} \tag{4.4}$$

Equation 4.4 is interpreted as follows. The progress (in average score) a genetic algorithm makes each generation, is proportional to the variance in the population (and inversely proportional to the average score). To my knowledge, this is actually the first time a relation is formally established between progress ($E_{i+1}[s] - E_i[s]$) and variance. Let's see how equation 4.4 predicts the course of the expected score during search. The equation predicts that the increase ($\frac{\mathrm{var}_i[s]}{E_i[s]}$) in expected score will decline, exactly because of the increase in expected score. If you would draw the expected score as a function of generation, it would be a curve flattening (converging) with increasing generations. The flattening is increased because the converging population will become more homogeneous, which reduces variance, which by 4.4 reduces the increase in expected score, but even without the loss of variance the curve will flatten. The described curve is characteristic for the actual functions, obtained in all experiments with GAs. GA-researchers have always argued that attention should be paid to maintaining variance in the population, especially towards the end of search, because convergence implies homogeneity of population. Now we see that there is even more reason to emphasize variance, because by 4.4 the increase in score is proportional to variance.

The result is remarkable in its generality, for the expression is independent of the crossover-operator used, and thus also of any parameter fixing the crossover mechanism! The main assumption was on the selection scheme, and indeed, equation 4.4 is entirely due to this choice. This can be seen as follows: suppose we choose as our crossover bit mask h the all-one vector. Since no assumption is made on the crossover mechanism, this choice is allowed. The all-one vector doesn't do any combining, it merely assigns the first parent to the first offspring, and the second parent to the second offspring. So population P_{i+1} consists of all chosen parents from P_i. Because selection was done proportionally to score, it is expected that the average score increases although no actual improvement has taken place. This is confirmed by 4.4, which states that the expected score can only increase in following generations (unless negative scores are allowed). So equation 4.4 is entirely due to the selection scheme.
But if the crossover scheme has no direct influence on the expected score of the next generation, why should we even bother designing a complex crossover operator? The answer is that the higher statistical moments (in particular the variance) of the next generations score **are** dependent on the crossover mechanism, and by 4.4 this has its effect on the expected score of P_{i+2}. So it seems that we should design our crossover operator by maximizing the **variance** of the next generations score, instead of the expected score, which is more intuitive. Note that the idea of maximizing the variance pleads for the use of a disruptive operator such as uniform crossover. Now that we mention disruption, equation 4.4 pretty much reflects the ideas presented in section 2.4 on the exploration-exploitation tradeoff: When we want to exploit high-performance individuals, we should be very selective with the choice of partners for mating. Indeed, in the discussion above we concluded that the increase in average score (as reflected in equation 4.4) is entirely due to selection process. But being selective on parents tends to diminish the next populations variance, which (again by 4.4) has a bad long-term effect on the populations average score. We conclude that selection pressure (exploitation) should be balanced with disruptive crossover (exploration) to neutralize the effect on variance. Thus, in equation 4.4 we find support for our plea for a disruptive crossover mechanism such as uniform crossover.

4.4 The linear all one problem

In this section we will continue our effort to optimize the score-statistics of a population. In section 4.3 we found support for the decisions to use **uniform crossover** to optimize **variation** (and higher order statistics). Because the calculation of these statistics in general are far too complex, we apply the ideas to a simple problem. We know from experience that the result of applying a method to a simple instance can already lead to major conclusions on how to apply the method in general. This section is devoted to the analysis of such a simple problem: the linear all one problem. In this problem, the score of an individual is just the number of ones in its chromosome. The problem has frequently been used by GA-researchers because of its simplicity. Just like in this section, the idea was that a GA must perform well on a simple problem before it can be considered for use on a difficult problem. For the reader interested in benchmark-problems: more recently, functions are devised according to one's wishes [Gold90] using Walsh-functions.

In section 4.2 we reduced the analysis of statistics to the analysis of the conditional moments $E[s^m|x,y]$, the m-th moment of the score s of the offspring given a probability distribution $p(cros(x,y) = x'|x,y)$ on the possible offspring x'. That is, we need to know what offspring with what probability can be produced by the parents x and y from the i-th generation. By assumption, all we know are the statistical moments of the *score* of the current population, so our model of the current population consists of scores. From these scores we will derive the probability distribution $p(cros(x,y) = x'|x,y)$, or better $p(cros(x,y) = x'|s(x), s(y))$ as a result of the last remark. So it comes down to the question: How many ones has the offspring, given the number of ones in the parents. In order to answer this question we consider one gene from the offspring. The gene-value (allele) will be one in the following two cases:

- the allele in the first parent is one and the corresponding bit in the bit mask h is one
- the allele in the second parent is one and the corresponding bit in h is zero.

Since r is the probability of a one in the bit mask h, the probability $p_1(x')$ of a one on a specified gene of the offspring x' will be: $p_1(x') = p_1(x)r + p_1(y)(1-r)$. The probability $p_1(x)$ is approximated by the fraction of ones in a string of length l with $s(x)$ ones, so $p_1(x) = \frac{s(x)}{l}$. Thus using the abbreviated notion p_1 for $p_1(x')$, we get:

$$p_1 = \frac{s(x)}{l}r + \frac{s(y)}{l}(1-r) \tag{4.5}$$

So the evaluation of an offspring's score is just a Bernoulli-trial with probability p_1 specified by 4.5 repeated l times. As a result, the score has a binomial probability distribution with parameter p_1. In the following the probability $p(s(cros(x,y)) = s_c|s(x), s(y))$ is used frequently and abbreviated to $p(s_c|s(x), s(y))$.

$$p(s_c|s(x), s(y)) = \binom{l}{s_c} p_1^{s_c}(1-p_1)^{l-s_c} \tag{4.6}$$

This probability is used to calculate

$$E[s^m|s(x), s(y)] = \sum_{s_c} p(s_c|s(x), s(y))s_c^m \tag{4.7}$$

If we can calculate this expression, we have completed the work done in section 4.2, and we can derive (for the all one problem) a relation between the score statistics of two subsequent populations. This relation can then be optimized by proper choice of the bit-mask probability r. In appendix A.1.4 two approaches are given for trying to calculate expression 4.7, both ending up in hopeless recurrence-relations. This may not be a disaster, however. Remember that our interpretation of equation 4.4 was to get as large a *variation* as we can get. Although we expect higher other statistics to play an important role in this process as well, we could (as an approximation) restrict our scope to optimizing variation. In appendix A.1.5, an expression is derived (expression A.30) for the second order moment of the the $(i+1)$-th generations score $(E_{i+1}[s^2])$in terms of the moments of order up to 3 from the i-th generations score. In general, we expect the moment of order m from the $(i+1)$-th generations score to depend on the moments of order up to $m + 1$ from the i-th generations score. This has an obvious analog in queuing theory, where the main concern is the time a job has to wait in a queue for a server to become vacant, or the total time the job stays in the system. The generalized Pollazek-Khinchine relations express the m-th order moment of these waiting- or staying times in terms of the moments up to order $m + 1$ from the arrival times. Because we can consider the population from the i-th generation as the supplier of 'jobs' for the population from the $(i + 1)$-th generation, we expect a similar relation to hold for the moments of the score. GA-theory may be interwoven with lots of other mathematical theories, depending on one's own specific view on GAs.

In section 4.3 it was shown that we should maximize the expression for $E_{i+1}[s^2]$ with respect to the bit-mask probability r to get the highest possible average score in generations thereafter. We suggested in the previous chapter that this parameter r should be adaptive (dependent on the the statistics of the population) in order to obtain maximum efficiency. We maximize $E_{i+1}[s^2]$ by setting the derivative (with respect to r) of expression A.30 to 0. This is done in appendix A.1.5, and we simply obtain the value $r = \frac{1}{2}$. This is not at all what we expected, because there is no dependence on the statistics of the population. The best possible parameterized uniform crossover just randomly puts ones and zeros in the mask with equal probability! Is this a mistake?

No. The only mistake we made is the one everybody else made in GA community. It is the assumption that the most disruptive crossover we know, uniform crossover, provides us with all the disruption we could ever need. In fact, most people still think that uniform crossover provides too much disruption! What we have just done, constitutes strong evidence that all these people are wrong. We showed that for an *extremely simple problem,* the best way to use uniform crossover is in its *most disruptive* appearance. This problem is so simple, that we can say with absolute certainty that the building-block hypothesis holds for this problem: The all-one vector is constructed by all-one sub vectors (schemas), all of which have above average scores. You could hardly think of any problem that is more easily 'guided' to the optimum by schemas, than the linear all-one problem. Furthermore, harder problems (less easily guided by schemas) require an exploration-exploitation tradeoff that is shifted towards exploration. This is true because a hard problem is often characterized by a high risk of encountering a local optimum, an effect that can only be neutralized by allowing more exploration. Yet, the most easy problem we can think of, already requires the maximum amount of exploration obtainable with uniform crossover. This implies that for *every* problem disruption should be set to its maximum.

In this chapter we have analyzed score-statistics of a dynamic population of individuals. We derived formulas expressing the way a genetic algorithm finds improvements in these statistics in subsequent generations. These expressions lead to an important insight in the implicit mechanism of a genetic algorithm. At the beginning of this chapter we hoped that we could find an optimal setting for the bit-mask probability r that is matched to the current statistics. What we found is a revolutionary and counterintuitive result stating that this probability should *always* be set to its most disruptive value.

Chapter 5

Permutation representations

One of the most serious objections made to GA-theory, is that its fundamental theorems regard only bitstring-representations. These representations are indeed the most suited when one's interest is to analyze the mechanism of GAs, but its practical purpose remains quite limited as bit strings are rarely the most convenient form to express solutions to all but pet-problems. It is of course true that *any* type of information can potentially be expressed in bits, but this would introduce trivial interactions between genes, of which the GA is unaware. Thus a forced bitstring-representation would make life unnecessarily hard for a GA. Using other representations has its price as well: an appropriate crossover-operator has to be designed, and disruption-analyses for other-than-bitstring representations are very difficult. In this chapter we will address exactly these issues for a particular type of representation. A *permutation*-encoding is often a convenient way of expressing solutions to certain types of *scheduling*-problems. That is, we have to allocate a number of jobs to machines in time, subjected to some constraints (precedence, resource, timing, etc.). Well known examples of such problems are job-shop scheduling (JSS) and the scheduling of data-flow-graphs (DFG) in VLSI circuit-design (see [Heij95] and [Wehn91]). A solution can be expressed by giving for each job the machine it is allocated to and its starting time, or equivalently by giving for each machine the starting times for the jobs allocated to it. We now look for a representation of a solution suitable for processing by a genetic algorithm. After some puzzling we decide that for our specific scheduling problem, a list of absolute starting times is **not** a representation suitable for a GA. This is mainly because the crossover process may swap absolute starting times such that precedence constraints are violated. This problem can be dealt with by defining 'relative' starting times, from which absolute starting times can be derived that do not violate precedence constraints. This is still not satisfactory. The point is that we want the GA to process essential information. What is essential about most scheduling problems? *Priority*, or *order* is essential. Typical scheduling decisions are: 'put this operation before that one', 'when there is a conflict, give this job priority over that one'. If we want a GA to process essential information, we had better put this kind of scheduling decisions *directly* into the representation of a solution. This is not done satisfactory with relative starting times. However, it is done satisfactory when using a permutation as a representation, because priority and order is exactly what a permutation represents. Absolute starting times can be easily extracted from a specified order of operations (a permutation) using a fast simple scheduling heuristic that respects the partial order specified by the precedence constraints. This justifies our use of a permutation as a representation for (potential) scheduling-solutions.

Since scheduling problems are among the most difficult of problems [Gare79], some theory concerning a crossover-operator for a permutation-representation should be available for those wanting to tackle serious problems with GAs. The purpose of this chapter is to provide the reader with theoretical results that provide either insight in the implicit GA-mechanism, or methodology for designing a coherent genetic algorithm.
In the first section, a uniform order crossover-operator from [Sysw91] will be described which is analogous to the uniform crossover used for bit strings in the previous chapter. In the next section a characterization will be given of the parents responsible for producing an offspring with a certain schema. Section 5.3 will use this characterization for the design of a new schema-theorem for uniform order crossover, in order to guide the GA in its search. In section 5.4 some conditions are derived under which the GA will work as a problem-solver. We try to improve the GA even further by altering the selection mechanism in section 5.5. Section 5.6 is devoted to the phenomenon of local optima, and section 5.7 shows why a specific sort of search-space reduction should not be used in a genetic algorithm.

5.1 Uniform crossover for permutations

In this section, the mechanism of uniform crossover is extended to the domain of permutations. A permutation is a string over an n-ary alphabet, containing all the elements from the alphabet exactly once. Above, we explained that a permutation encoding is quite suitable for scheduling problems because the order in which operations have to be executed can easily be related to the order in which they occur in a permutation. In this section a crossover operator will be introduced that is especially designed for this representation. One of the difficulties in designing a crossover operator in general is to produce a legal chromosome. This is especially true when we consider a permutation. The problem is of course that traditional crossover-operators (for bitstring-representations), when provided with permutations, in principle can produce *any* string over the n-ary alphabet, whereas only a subset of these are permutations. There are n^n possible strings over an n-ary alphabet and only $n!$ permutations. The resulting redundancy must be filled by the crossover-operator in a meaningful way in order to obtain a legal solution, and that's why a crossover operator has to be suited for a certain representation of a solution. We will see that the crossover-operator used in this chapter always produces permutations when provided with permutations.

Another aspect to keep in mind is the type of information relevant with regard to the score. For the travelling-salesman problem (TSP) *adjacency* -information is relevant, because the distance to its neighbor (within the tour) is added in the total evaluation. However, it couldn't care less about cities other than the neighbors in a tour. In our discussion starting this chapter concerning some scheduling problems, we found a permutation encoding suitable for representing *order*-information. This is type of information is spread more evenly around all elements of the permutation (compared to a permutation encoding for the TSP). This is also due to the type of problem we try to tackle: The partial order, specified by the precedence constraints, dominates the order specified by the permutation. In fact, the order specified by the permutation plays a role only between elements that are not related by the precedence constraints, directly or indirectly. The dramatic result when adjacency and order are mixed up, is made particularly clear in [Star91]. In this article, the crossover-operator from [Sysw91] (also called order-crossover) described in this section is compared with several other crossover operators especially designed for scheduling-purposes. A necessary consequence of the importance of order is that order **must**

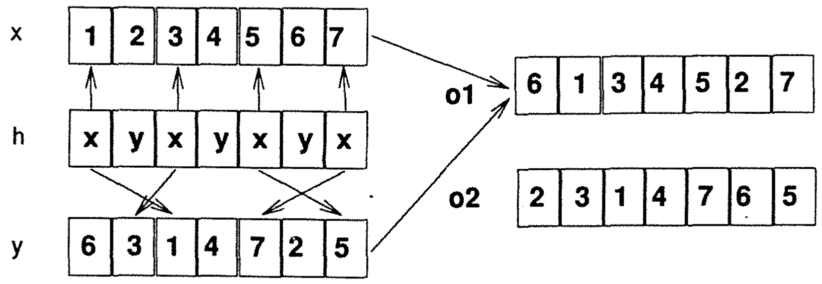

Figure 5.1: uniform crossover for permutation-representation

be reflected in our notion of *schema*. A schema will be given as an incompletely specified order of elements. Every permutation that can be obtained from this order by completing it, contains the schema. For example (1,5), (1,3,4), (2,5,6,7) are all schemas part of parent x in figure 5.1.

In uniform crossover for permutations, a mask h is constructed using a bit mask probability r just like in normal uniform crossover. The one's in the mask are used to identify the genes from the first parent used for crossover. The difference with uniform crossover for bit strings is that the relative order of the selected genes from the first parent is *forced* onto the corresponding elements in the second parent. This is essential for obtaining a permutation, because each gene must occur exactly once. The mechanism is illustrated in figure 5.1. In the figure, the process is repeated with the complementary mask h_c for producing the second offspring.

In the example, the order (1,3,5,7) from x is forced onto the corresponding elements from y to create $o1$. The absolute positions (and thus the relative order) of the remaining elements (6,4,2) from y stay intact. The order (2,4,6) from x is used next to create $o2$.

The uniform crossover-mechanism for permutations has not been an important focus of theoretical analysis. A thorough examination of literature has revealed that in [Karg92] some analysis is given, but the results reported are poor and very imprecise. The authors have only considered the case in which the bit mask probability $r = .5$ and (contradictory to their own interpretation) schemas consist of fixed positions instead of relative order. Further (unmentioned) assumptions are made with regard to the randomness of the population. Besides crucial mistakes, the analysis suffers from the same objection, made in section 3.5 to the analysis in [DeJo92].

5.2 Constructing parents

In the rest of this chapter a new analysis of uniform order crossover will be represented, including a new schema-theorem for permutations which is free of all objections made to the analyses mentioned. Proceeding in a way analogous to section 3.5, we might ask 'what is the probability that a schema survives under crossover ?'. However, one of the objections made to the entire methodology of section 3.5 is that *creation* of a schema is not sufficiently taken into account, whereas it may very well play an even bigger role than survival. It is therefore important to exactly determine the parental conditions for producing an offspring with a specified schema. We must realize that the situation here is radically different from that in section 3.3. For example, it is possible that 2 parent-permutations both contain a schema and are nonetheless *unable* (with a certain mask) to produce two children bearing that same schema! This is illustrated in figure 5.1 where schema (2,5) is contained in both parents but not in the child $o1$.

In the analysis we assume that one child is produced by crossover. Before we start with an analysis of uniform crossover for permutations, we have to define how the relevant elements are referred to. So let two chromosomes be selected for uniform crossover for permutations. We refer to these two chromosomes as the first parent and the second parent. Furthermore, let the mask h be a bitstring containing a '1' at those places which in the first parent correspond to elements chosen for crossover, and a '0' at all other places. Then

Def 5.1 *K is the set of elements constituting a schema*

Def 5.2 *C is the set of elements chosen for crossover*

It is also convenient to be able to talk about the intersection of some sets.

Def 5.3 *I is the set of elements from K that are chosen for crossover, so $I = K \cap C$*

Def 5.4 *J is the set of elements not in K that are chosen for crossover, so $J = C/I$*

Def 5.5 *O is the set of elements from K that are not chosen for crossover, so $O = K/I$*

For an example, observe figure 5.1, and assume that schema $K = \{1,4,6,7\}$. The elements chosen for crossover $C = \{1,3,5,7\}$. The intersection of K and C is $I = \{1,7\}$. Furthermore, $J = \{3,5\}$ and $O = \{4,6\}$.

We refer to the cardinality of each of these sets by their corresponding small print, so $k = |K|$, etc. From the above definitions it follows that I and O are a partition of K and that I and J are a partition of C. As a result, the following equalities hold with respect to the cardinalities of these sets:

$$k = i + o$$

and

$$c = i + j$$

The reader may have noticed that there are still some elements which have no name. These are the elements that are neither in C nor in K, and they have no name because they can not in any way affect the way the K-elements are ordered in the offspring, so

Observation 1 *Only the elements (in both parents) from the schema (K) and those selected for crossover (C) play a role in the survival or creation of schema K.*

Consequently, the remaining elements (the elements not in K and not in C) are omitted in all the following analyses. As an assumption for the analyses it is furthermore known what elements have been selected for crossover (C). All numbers and probabilities are thus conditional on the selected elements, and in the end we will have to average in order to lose this dependency.

Examination of the crossover-mechanism leads to three other observations, which together constitute a full characterization of possible parent-pairs of a child carrying the K-elements in the right order (the order defined by the schema). In each observation, the explicit goal will be to produce such a child, and when we say that something doesn't matter, it will be relative to this goal. The first observation arises from the fact that the information taken from the first parent describes only the relative order of the C-elements. This leads to

Observation 2 *In the first parent, neither the position nor the relative order of the O-elements matter.*

The relative order of the C elements from the first parent is forced to the same elements from the second parent, completely destroying that particular information in the second parent, so

Observation 3 *In the second parent, the relative order of the C-elements does not matter.*

Now, when this relative order is forced, it is important where the I-elements show up in the result, especially their position relative to the O-elements from the second parent, because they together constitute the schema K. However, the remaining J-elements may show up anywhere in the result. Their position is determined by the absolute position of the O-elements from the second parent and the relative position of the J-elements from the first parent. The former are fixed, but the latter can be varied randomly, so

Observation 4 *In the first parent, the relative order of the J-elements does not matter.*

These observations will now be used for a prescription for designing parents. The idea is to first design a template from the schema K. From this template both parents are constructed.

- Template: The template is the schema K in the right order with unlabeled (unspecified order) J-elements inserted randomly. Note that every order of the J-elements is considered equivalent w.r.t. the template. The number of ways in which this can be done is the same as the number of ways the reverse is done: randomly pick k elements from the total $k + j$ elements. The number of ways is $\binom{k+j}{k}$. Another way to see this is is follows: The first J-element can be inserted in the K-elements in $k + 1$ ways, the second in $k + 2$ ways, etc. The last (j-th) can be inserted in $k + j$ ways. This amounts to $\frac{(k+j)!}{k!}$. Since every order of the J-elements is considered equivalent, this fraction has to be divided by $j!$. The total number of ways is $\frac{(k+j)!}{k!j!} = \binom{k+j}{k}$

- First parent: Because of observation 4 the J-elements can be ordered randomly. This can be done in $j!$ ways. As a result of observation 2 the O-elements can be reinserted and ordered (labelled) randomly. There are $\binom{k+j}{o}$ ways to reinsert the O-elements, and $o!$ ways to label them. So reinserting and labelling O-elements can be done in $\binom{k+j}{o}o! = \frac{(k+j)!}{(k+j-o)!} = \frac{(k+j)!}{c!}$ ways, since $k - o = i$ and $i + j = c$. The total number of first parents corresponding to one specific template is thus $\frac{(k+j)!j!}{c!}$

- Second parent: Because of observation 3 the C-elements can be ordered randomly. This can be done in $c!$ ways, so the total number of second parents corresponding to one specific template is $c!$.

For example, suppose we are concerned with the schema (1,2,3,4) so $K = \{1,2,3,4\}$, and we know that the elements in $C = \{2,3,5,6\}$ have been chosen for crossover. Then $I = \{2,3\}$, $J = \{5,6\}$, and $O = \{1,4\}$. The template may be chosen as $(1,J_1,2,J_2,3,4)$ where the J-elements are unlabeled. To produce the first parent, the J-elements can be chosen randomly, so we may get the first parent (1,6,2,5,3,4). Furthermore, the O-elements can be reinserted and ordered randomly, so our first parent could be (6,4,2,1,5,3). The second parent is obtained from the template by randomly ordering the C-elements, so it may be (1,5,6,3,2,4). Uniform crossover for permutations forces the order (6,2,5,3) to their corresponding elements in the second parent and produces the child (1,6,5,2,3,4). Indeed, the schema (1,2,3,4) is contained in this child. In addition, because of observation 1 elements not in K and not in C may be inserted randomly in each parent, so suppose our first parent becomes (6,4,7,2,1,5,7,3,9) and our second parent becomes (8,1,5,6,9,3,2,4,7) then the child (8,1,6,2,9,5,3,4,7) is produced, which of course carries the schema.

The total number of parent-combinations is the product of the number of ways each step can be performed:

$$\#(\text{parent-pairs}) = \binom{k+j}{k} j! \frac{(k+j)!}{c!} c! = \frac{(k+j)!^2}{k!} \tag{5.1}$$

As a check, note that the probability of encountering a schema K in a *randomly* generated chromosome is the reciproke of the number of possible schemas with the elements from K: $\frac{1}{k!}$. There are $(k+j)!$ possible chromosomes (remember observation 1), and thus $(k+j)!^2$ possible chromosome-pairs. So the probability of creating a chromosome with schema K from *randomly* generated chromosomes is $\frac{\#(\text{parent-pairs})}{\#(\text{chromosome-pairs})} = \frac{1}{k!}$, and this equals (and should equal) the probability of encountering a schema K in a randomly generated chromosome.

Note that the number of possible parent-pairs is independent on *either* the number of selected schema-genes (i) *or* the total number of selected genes (c). They have however effect on the individual number of possible first or second parents.

As an example let's take $k = 3$ and $j = 1$. By equation 5.1 the number of possible parent-pairs is 96. Assume for example that $i = 2$ and thus $c = 3$. In table 5.1 the templates and (for each template) both parents are constructed by the steps given above. In the table, I-elements are indicated by capitals (A and B), O-elements by figures (1), and J-elements are in small print (a). The schema K is taken to be A1B. The number of templates is 4, and the numbers of first and second parents per template are 4 and 6 resp. Indeed the total number of configurations is $4 \times 4 \times 6 = 96$. The reader can verify the legality by constructing the offspring using a first and second parent from the same row. (they must be constructed from the same template) Note that not all possible first or second parents differ. In the parent 1 column, the second and third row are equal, as are the first and second row and the third and fourth row from the parent 2 column. No two constructed parent-*combinations* are the same however. Note also that you cannot randomly pick a first and second parent from different rows. In order to produce an offspring with schema K the parents need to have something in common, and that's why they were constructed from one and the same template!

In this section we have given a complete characterization of the parent-chromosomes when a certain schema is required in the offspring. Furthermore, the number of these parent-combinations is calculated. What we can do with these numbers to actually optimize a genetic algorithm will become clear in the following sections.

Table 5.1: construction of templates and parents for $k = 3, j = 1, i = 2$, and $c = 3$.

template	possible first parents	possible second parents
aA1B	1aAB, a1AB, aA1B, aAB1	aA1B, aB1A, Aa1B, AB1a, Ba1A, BA1a
Aa1B	1AaB, A1aB, Aa1B, AaB1	aA1B, aB1A, Aa1B, AB1a, Ba1A, BA1a
A1aB	1AaB, A1aB, Aa1B, AaB1	a1AB, a1BA, A1aB, A1Ba, B1aA, B1Aa
A1Ba	1ABa, A1Ba, AB1a, ABa1	a1AB, a1BA, A1aB, A1Ba, B1aA, B1Aa

5.3 A schema-theorem for permutation-encoding

When a distribution of chromosomes in a population is known, we can use the characterization given in the previous section, to calculate exactly the expected number of chromosomes carrying a specific schema in the next generation. In this section we will derive a closed-form expression incorporating this idea. The expression represents a schema-theorem for a permutation-encoding and can be used to predict the behaviour of a genetic algorithm when a specific distribution of chromosomes in a population is assumed.

A traditional disruption analysis consists of calculating the probability p_d that a proportionate selected parent carrying schema K, will produce an offspring with schema K. This probability would then be substituted in the schema-equation 2.4 to obtain a rough estimation on the expected number of individuals carrying K in the next population. The approach we take, is radically different: we do not calculate a probability of disruption, yet the result will be a major generalization of the schema-theorem 2.4. For clarity I repeat that we want a specific schema K to exist in the next generation. In the previous section we gave a way to enumerate all parent pairs that produce an offspring containing schema K, given a crossover-mask h. To get the expected number of chromosomes in the next generation that contain K, we could try to compute (and sum over) the *joint* probabilities of selecting parents these. An easier way is to sum over a single probability of selecting one parent and then compute the probability of selecting the 'corresponding' parent in table 5.1. This forces us to decide which parent (first or second) we take for the summation. It is convenient to choose the parent that is more easy to characterize from the schema K, since K is assumed known. We also assumed the bit-mask is known, and thus the sets I ($= \{A, B\}$, see definition 5.3) and O ($= \{1\}$, see definition 5.5). The sets I and O are a partition on the schema K, such that the order of the I-elements are inherited from the first parent, and the order of the O-elements are inherited from the second parent. It would be convenient to enumerate either all first or all second parents using either the set I or the set O. Examining table 5.1, we observe that all possible chromosomes containing I in the right order (in which the elements occur in the schema K) are present in the first-parent column, and thus *all* chromosomes containing I in the right order are enumerated. This is true in general. This is *not* the case with O in the second parent. There are $\frac{(k+j)!}{o!}$ chromosomes carrying O in the right order. For the example of table 5.1 this equals 24, yet there are only 12 different strings in the second-parent column. We conclude that we can enumerate all possible first parents with the set I, but enumeration of all chromosomes with the O-elements in the right order also gives chromosomes that cannot serve as a second parent. The best choice for enumeration is thus the first parent. We summarize our strategy for obtaining a schema-theorem as follows:

assumption We assume the sets K (def 5.1), C (def 5.2), I (def 5.3), and O (def 5.5) are known.

enumeration We sum over all possible first parents by enumerating all chromosomes carrying the set I in the right order.

template We calculate the average number of templates that produce a certain first parent.

second parent We calculate the average number of second parents per first parent.

The first two points have been explained above. We continue by computing the number of (possibly double) entries in the first-parent column of a table such as 5.1. In the previous section, we saw there are $\frac{(k+j)!j!}{c!}$ ways to construct a first parent from a given template. There are $\binom{k+j}{k}$ templates, so the total number of entries in the first-parent column is $\frac{(k+j)!j!}{c!} \times \binom{k+j}{k} = \frac{(k+j)!^2}{k!c!}$. Now, there are $\frac{(k+j)!}{i!}$ possible first parents (I-carriers), so the average number of entries of a first parent is $\frac{(k+j)!^2}{k!c!} / \binom{k+j}{k} = \frac{(k+j)!i!}{k!c!}$. That is, each first parent corresponds, on the average, to $\frac{(k+j)!i!}{k!c!}$ templates. There are $c!$ ways to construct a second parent from a given template, so per first parent, there are, on the average, $\frac{(k+j)!i!}{k!c!} \times c! = \frac{(k+j)!i!}{k!}$ mates for a first parent. There are a total of $\frac{(k+j)!}{o!}$ O-carriers, so the fraction of O-carriers suitable as mate for a certain first parent, is

$$\frac{(k+j)!i!}{k!} / \frac{(k+j)!}{o!} = \frac{i!o!}{k!} = \frac{i!(k-i)!}{k!} = \binom{k}{i}^{-1} \tag{5.2}$$

Note this is independent on the total number, c, of genes selected for crossover. For a schema-theorem we need just a few more additions. Let $N(I,g)$ denote the number of chromosomes carrying I in generation g of the population. The score of an arbitrary I-carrier (a chromosome carrying I in the right order) is estimated by $s(I)$ (see equation 2.1). The probability of selecting a *specific* I-carrier is $\frac{s(I)}{\sum_{x' \in P} s(x')} = \frac{s(I)}{|P| E_g[s]}$, where $E_g[s]$ is the average score of the individuals in generation g. The expected number of I-carrier selected for crossover is $N(I,g)$ times this probability: $\frac{s(I)}{E[s_g]} \frac{N(I,g)}{|P|}$. Similarly, the expected number of O-carrier selected for crossover is $\frac{s(K/I)}{E[s_g]} \frac{N(K/I,g)}{|P|}$ since $O = K/I$. Only a fraction $\binom{k}{i}^{-1}$ of these O-carriers are suitable for producing an offspring carrying schema K. So when the I-genes are selected for crossover, the probability of producing an offspring carrying schema K is $\frac{s(I)}{E[s_g]} \frac{N(I,g)}{|P|} \times \frac{s(K/I)}{E[s_g]} \frac{N(K/I,g)}{|P|} \binom{k}{i}^{-1}$. Because we want to fill a whole new population with offspring from the current population, this probability is applied to $|P|$ individuals. To get the expected number of K-carriers for generation $g+1$ the result has to be averaged over all possible I. Since the probability of choosing from the K-elements exactly I for crossover (plus an arbitrary number of elements outside K) is $r^{|I|}(1-r)^{|K/I|}$, our schema-theorem for permutations is

$$E[N(K, g+1)] = \sum_{I \subset K} r^{|I|}(1-r)^{|K/I|} \frac{s(I)s(K/I)}{E_g^2[s]} \frac{N(I,g)N(K/I,g)}{|P|} \binom{|K|}{|I|}^{-1} \tag{5.3}$$

To compare this equation with the original schema-theorem 2.4, note that 2.4 is just the term from 5.3 with $I = K$, since $N(K/K, g) = |P|$, $s(K/K) = E[s_g]$, $\binom{|K|}{|K|} = 1$, and $r^{|K|} = p_d$. Thus, our new schema-theorem is a true generalization of the original one. Also note that the inequality has been replaced by an equality. This is because all terms have been taken into account now. The equation can be aggravated in the following ways:

- $N(I,g)$ can be conveniently expressed in a way that reflects the occurrence of (parts of) the specific schema K in the current population.
- $s(I)$ can be expressed (for example by using a generating function, or Walsh-coefficients) in a way that reflects some features of the problem at hand.

Equation 5.3 can be simplified assuming both $s(I)$ and $N(I,g)$ depend only on $|I| = i$, instead of the specific genes. Because there are $\binom{k}{i}$ ways to choose i genes from the k schema-genes, 5.3 becomes

$$E[N(k,g+1)] = \sum_{i=0}^{k} r^i(1-r)^{k-i} \frac{s(i)s(k-i)}{E_g^2[s]} \frac{N(i,g)N(k-i,g)}{|P|} \tag{5.4}$$

This is the expression I promised to derive at the beginning of the section. It can be used to predict the behaviour of a genetic algorithm when a specific distribution of chromosomes in a population is assumed. The distribution of chromosomes is of course needed for the terms $N(i,g)$ and $N(k-i,g)$. We can use it for example, to calculate how 'easy' two schemas combine dependent on how well these schemas have invaded the population (reflected by peaks in the distribution). In the next section, we will use this new schema-equation to strike at the very core of the GA-mechanism.

5.4 When does the GA work?

It would be ambitious to try to obtain sufficient conditions for a GA to find a good solution (let alone the optimal solution), but we can state some *necessary* conditions intuitively. In order to state these conditions we have to consider the *implicit* search mechanism a genetic algorithm relies on. This mechanism is (implicitly) stated in the building-block hypothesis, and goes something like this: large schemas are made from smaller ones. We can easily derive necessary conditions for the building-block hypothesis to work. Surely, the search process must be *directed* in some way: random search does not suffice, and so we demand that above-average schemas from the current population occur more frequently in the next generation. If they don't, we cannot expect it to be very probable for two schemas to combine. Furthermore, the GA must be able to cope with the easiest type of problems, in the sense that is stated by the building-block hypothesis: large schemas actually *consist* of smaller ones. In this section we will use these necessary conditions, along with the schema-equation derived in the previous section, to come up with some very surprising result concerning the exploration-exploitation tradeoff.

To analyze the first condition, we assume the building-block consisting of K has not yet spread in the population. In this case we can assume a random population with respect to I. There are $\frac{(k+j)!}{i!}$ I-carriers, which is a fraction $i!^{-1}$ of the total number of chromosomes. The population contains on the average $N(i,g) = \frac{|P|}{i!}$ I-carriers, and from 5.4 it follows that

$$E[N(k,g+1)] = \sum_{i=0}^{k} r^i(1-r)^{k-i} \frac{s(i)s(k-i)}{E_g^2[s]} \frac{|P|}{i!(k-i)!} \tag{5.5}$$

We can rewrite equation 5.5 so that it can be interpreted more conveniently:

$$E[N(k,g+1)] = \frac{|P|}{k!}\frac{1}{E_g^2[s]}\sum_{i=0}^{k}\binom{k}{i} r^i(1-r)^{k-i}s(i)s(k-i) \tag{5.6}$$

Note that $\frac{|P|}{k!}$ is again the expected number of chromosomes carrying K in a random population. Equation 5.6 expresses the increase in $E[N(k,g+1)]$ relative to a random distribution, when all K-carriers are built from scratch. Because our optimum solution eventually has to be build from scratch, the random-population assumption is a practical one when analyzing the 'creativity' of the GA. We can now derive the conditions under which we expect a schema to thrive. From equation 5.6 it follows directly when a schema occurs more frequently than in a random population (the current population):

$$\frac{1}{E_g^2[s]}\sum_{i=0}^{k}\binom{k}{i} r^i(1-r)^{k-i}s(i)s(k-i) > 1 \tag{5.7}$$

When the inequality in 5.7 is replaced by an equality, the resulting equation describes **exactly** the borderline between random-search and directed search. When we have a mathematical model for the evaluation of a solution (that is, an expression for $s(i)$), we can use 5.7 to calculate the conditions under which the GA directs its search process. Unfortunately, such a mathematical model is not easy to find in general. At this point we can incorporate in our analysis the second necessary condition for a GA to work: it must be able to cope with problems that correspond to the building-block hypothesis. Perhaps the easiest non-trivial of such problems is a linear problem. We have already encountered the linear all-one problem in section 4.4 and found some surprising results. Lets see how the permutation-equivalent turns out. A linear model assumes $s(i) = a \times i$. In appendix A.2 we derive from equation 5.6 the following equation:

$$E[N(k,g+1)] \approx \frac{|P|}{k!}\frac{s^2(k)}{E_g^2[s]} r(1-r) \tag{5.8}$$

Actually the linear model $s(i) = a \times i$ is not correct, because $s(i)$ is the average score over all chromosomes having at least i elements in the right order. $s(0)$ doesn't state any restriction and is simply the average score over all possible chromosomes, however the linear model says $s(0) = 0$. It is better to use as a model $s(i) = E[s] + a \times i$. The calculation in appendix A.2 gets much more complicated this way, but the result is a minor change from equation 5.8, so we omit it. From equation 5.8 we derive that the GA will perform better than random search when

$$s(k) > \frac{E_g[s]}{\sqrt{r(1-r)}} \tag{5.9}$$

This condition is mildest when the GA is set to be most explorative, that is, when $r = .5$. We have obtained a result very similar to the one in section 4.4: The balance of exploration and exploitation should be set at the extreme in favor of exploration, at least as far as crossover is concerned. The reasons for this are already given in section 4.4: more difficult problems need more exploration, and for the easiest problem we can think of we already need the maximum amount of exploration. In the next section we try to push the balance towards exploration even further.

5.5 Adaptive selection pressure

Two factors are involved in the tradeoff between exploration and exploitation: crossover and selection. So far we have assumed roulette-wheel selection, and concluded that the explorative aspect of crossover should be maximally exploited. In order to push the balance towards exploration even further, we have little choice but to alter the selection mechanism. A scheme often used is *Boltzmann*-selection. The probability of selecting chromosome x using this scheme is:

$$sel(x) = \frac{e^{p_b s(x)}}{\sum_{x' \in P} e^{p_b s(x')}} \tag{5.10}$$

(the denominator is for normalization) In equation 5.10, p_b is a constant related to the selection-pressure. This highlights an important advantage of this selection-scheme: selection-pressure is easily adjustable to fit requirements in different situations. In this respect, the constant p_b is the analogous of the bit mask probability r in crossover. We will exploit this property and develop a method to choose appropriate parameter values.

The fundamental ground on which we base our method of choosing parameter values is variance within the population. In section 4.3 and especially equation 4.4 it was made clear just how important it is to have a varied population of chromosomes. On the other hand, when the population is forced too hard to be varied, a new above average individual may have no chance to spread its genes in the population, thus inhibiting the combining of good schemas. Our goal with an adaptive selection mechanism is to optimize this variation. That is, when the population is too homogeneous the Boltzmann variable should be set low, and when the population is too varied it should be set high. As an approximate measure for the variation in the population, we use the relative difference between the best and average score in the population: $var = \frac{best - E[s]}{best}$. The selection mechanism has no direct control over the variation in the population, because selected individuals first go through the process of crossover, which tends to enhance the variation in selected individuals. We will choose the Boltzmann parameter p_b in such a way that the variation in *selected* individuals tends to converge to a target variation v. This brings us to the first aspect of our scheme: an equilibrium.

The equilibrium is a situation without any pressure with regard to the variation, upwards or downwards. It corresponds to a value the variation in the population is converging to. With regard to the Boltzmann selection scheme, the potential point of equilibrium is easily computed. It corresponds to the value of p_b, the Boltzmann-variable, where individuals are selected with equal probability. This is the case when $p_b = 0$ (substitute in equation 5.10). When individuals are selected with equal probability, the variation in selected individuals equals the variation in the current population var_i. This is only a potential point of equilibrium because it is necessary for equilibrium that the crossover process does **not** enhance the variation in selected individuals, otherwise $var_{i+1} > var_i$, which does not correspond to an equilibrium. Thus, an equilibrium can only be obtained when the individuals in the population are such look a likes, that crossover is unable to introduce any disruption. This situation is much like a local optimum in local

search algorithms. It is also very improbable that any improvement is made from the point of equilibrium, so that the search process may stop. It is therefore convenient to detect an equilibrium, which is actually quite simple: Since crossover does not enhance variation in the case of equilibrium, variation in selected individuals equals variation in population. Furthermore, at the point of equilibrium the variation in selected individuals equals the target variation set by the user. To detect an equilibrium, one has only to check whether the variation in population equals the target variation set by the user.
When no such 'local optimum' is encountered, we may assume that crossover always enhances the variation, and therefore an equilibrium does not occur. This is fortunate, because the equilibrium implies that individuals are selected with uniform probability, which more or less reduces the GA-process to a random search process. However, it is still convenient to talk about a (potential) equilibrium, because the target variation v is tightly connected to it. We define the potential equilibrium (or equilibrium for short) as the point in the search process where individuals are selected with uniform probability, that is $p_b = 0$. The target variation v we define as the value of the variation in the population corresponding to $p_b = 0$ (according to some update rule). It also corresponds to the value the variation in selected individuals is converging to, but never reaches while crossover enhances the variation.

The first update rule we consider is simple:

$$p_b = a \times var + b \tag{5.11}$$

Equilibrium takes place at $p_b = 0$, such that $var = -\frac{b}{a}$. So when we think a certain amount of variation, say 20%, is required, we set a and b such that $b = -.2 \times a$. The second aspect of our scheme is *dominance*. It is the phenomenon that nearly all individuals in the population are offspring from one individual. When this is the case, schemas are so much alike that there is little chance of further improvement by combining different schemas. No need to look further, clearly a situation we want to avoid. Dominance occurs for example when a random population is infected with one good solution. Many people think that 'feeding' a GA with such useful information is a helping hand, but nothing could be further from the truth: in the second generation, almost all individuals are offspring from that one good solution. This is a severe restriction of the search space for a GA. Instead of helping a hand, you merely introduce dominance in the population, sabotaging the search process.

Dominance may also occur when selection pressure is too high, because differences in score are overemphasized, leading to a situation similar to the one sketched above. Progress in the search process of a genetic algorithm is typically highly nonlinear: improvements are sporadic and may take high values. Because of our measure of variation ($var = \frac{best-E[s]}{best}$), this results in sporadic large changes in the variation. When the sensitivity of the update rule is high (a small change in variation results in a large change in the Boltzmann variable), selection pressure may sporadically be set much too high, resulting in dominance. The occurrence of dominance thus depends on the sensitivity of the update rule. Because the step from selection pressure to probability of selection run through the Boltzmann selection scheme (equation 5.10), the sensitivity of the intermediate step is also of importance. This sensitivity is analyzed in appendix A.3. It is also concluded in this section that a simple variation of the Boltzmann selection probabilities does not improve sensitivity.

In section A.3.1 the sensitivity of the first update rule is analyzed. The analysis leads to expression 5.12.

$$\left|\frac{\Delta sel(x)}{sel(x)}\right| = a \times p_b \times var \left|\frac{\Delta var}{var}\right| \tag{5.12}$$

Notice that the sensitivity is quadratic in the variable a. Minimizing the effect of dominance implies the use of a small value of a. This however severely restricts the dynamic range of the Boltzmann-variable p_b, as can be seen in the update rule 5.11. A way around this problem is by introducing an element of integration. This leads to the second update rule.

$$p_b[i] = \alpha \times p_b[i-1] + (1-\alpha)(a \times var[i] + b) \tag{5.13}$$

Where $0 < \alpha < 1$. Notice that the element of integration forces us to make the various variables time-dependent. Also notice that this rule makes sure the equilibrium is the same as the equilibrium using the linear update rule. The sensitivity of this update rule is analyzed in section A.3.2, and leads to equation 5.14

$$\left|\frac{\Delta sel(x)}{sel(x)}\right| = (1-\alpha) \times a \times p_b[i] \times var[i] \left|\frac{\Delta var[i]}{var[i]}\right| \tag{5.14}$$

The sensitivity using the second update rule is a factor $(1-\alpha)$ smaller than the sensitivity using the first update rule (compare with equation 5.12), whereas the equilibrium is unchanged. This enables us to handle a relatively large dynamic range for the Boltzmann-variable while restricting the effect of dominance. The presentation of results, obtained by using adaptive selection pressure on a scheduling problem, is postponed to the next chapter where this problem is introduced and implementation issues are discussed.

In this section we have developed a method to design a genetic algorithm with an adaptive selection mechanism. Using the notions of equilibrium and dominance we were able to enforce a certain amount of variation in the population without jeopardizing the loss of too much information. In the next section we examine what will happen when schemas are 'overrepresented' in the population.

5.6 Local optima: how do they look like?

It is often assumed that uniform order crossover is so disruptive, that ending up in a local optimum is out of the question. Contrary to this belief, we showed in the former section that even in the mildest conditions (with regard to the score-function) disruption should be as high as possible. In this section we will show how a local optimum in connection with uniform order crossover looks like.

In the first chapter we discussed the concept of a local optimum in relation to genetic algorithms. The definition of a local optimum in this regard should encapsulate the suggestion that a good solution is very unlikely to be reached in a reasonable amount of time when a local optimum is encountered. This unconventional notion of a local optimum is motivated by the practical application of heuristics in general: finding a **good** solution (instead of optimal) in a **reasonable** amount of computation time. So for all practical purposes of a genetic algorithm,

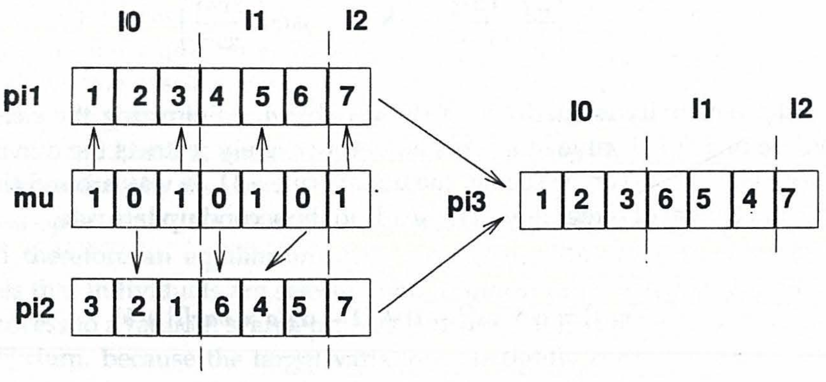

Figure 5.2: the partition property of uniform order crossover

this condition for a local optimum suffices. A convention easily derived from this condition, is that we speak of a local optimum when a large part (or the whole) of the population contains a schema not present in any good solution. A generalization of this degree of penetration of a schema, is the degree of penetration of a *partitioning*. In the following we will formally prove that uniform order crossover has the property of maintaining partitions. To define this property we need the following:

Let π denote a permutation of n elements, and let π_i denote the i-th element of this permutation. Let the index set $I = \{0, 1, \ldots, n\}$, and let this set be partitioned into m nonempty index subsets $I_1, I_2, \ldots, I_m$, such that each index subset contains only subsequent indices. For example $I_1 = \{5,6,7\}$, $I_2 = \{2,3\}$, etc. Let $\pi(I_i)$ be the projection of the index subset I_i under the mapping defined by permutation π. Now let $E_i = \pi(I_i)\ \forall i \leq m$. Then $E_1, E_2, \ldots, E_m$ is a partition on the set $V = \pi(I)$. We can now define the notion of the partition-property.

Def 5.6 (partition-property) *Let G be a genetic operator taking p parents and producing one child, and let $\pi^1, \pi^2, \ldots, \pi^p$ be permutations such that $\forall i : \pi^1(I_i) = \pi^2(I_i) = \ldots = \pi^p(I_i) = E_i$. Then G has the partition-property iff $G(\pi^1, \pi^2, \ldots, \pi^p) = \pi^{p+1}$ is such that $\forall i : \pi^{p+1}(I_i) = E_i$. That is, the result of G has the same partition $I_1, I_2, \ldots, I_m$ as all its parents have.*

Theorem 1 *Uniform order crossover has the partition-property*

proof: Let μ be a randomly generated binary mask of length n. Observe the index set I_1 in π^1, π^2, and the child π^3. Now assume there are a ones in the part of mask μ corresponding to the indexes of I_1, that is $a = \sum_{j \in I_1} \mu_j$. Now from the part corresponding to the indexes of I_1 from the first parent π^1, a elements are transferred to the corresponding part in the child π^3. The rest of the $|I_1|$ elements in the I_1-part of π^3 is filled with elements from the second parent π^2. Immediately after the transfer from π^1, there are $|I_1| - a$ empty spots in π^3, and there are $|I_1| - a$ elements from $\pi^1(I_1)$ and thus from E_1 that have not yet been used in π^3. Since $\pi^2(I_1) = E_1$, these $|I_1| - a$ elements from E_1 are exactly the elements from the I_1-part of π^2 that uniform order crossover uses to fill up the empty spots in the I_1-part of the child π^3. Now the I_1-part of π^3 is filled and we only used elements from $\pi^1(I_1)$, and $\pi^2(I_1)$, and thus from E_1. However, $|\pi^3(I_1)| = |I_1| = |E_1|$, and the elements in $\pi^3(I_1)$ are all different and in E_1. We conclude that $\pi^3(I_1) = E_1$. This proves the theorem for the I_1-part. Now we cut off the I_1-part from all permutations (this can be done safely because they all correspond to the same set of elements: E_1) and the mask μ, and consider the remainders as permutations of elements from V/E_1. Now the above argument can be repeated inductively for all parts $I_2, I_3, \ldots, I_m$. $\diamond$

The partition-property of uniform order crossover is illustrated in figure 5.2. We conclude that if a population only has individuals with the same partition, uniform order crossover will maintain this partition in all individuals it produces over all generations. Relaxing the condition of entire penetration of the partition to a high degree of penetration, we conclude that it is unlikely that a good solution can be found in a reasonable amount of time, and thus the genetic algorithm converges to a local optimum (unless there is a good solution that can be expressed using the same partition). It was already mentioned in section 3.5 that mutation and inversion are no answers to the problem: a local optimum is not just local, it is also an optimum, so the probability that a random search-like operator yields above average individuals (which is a necessity for survival) is quite small. We thus have to take seriously the problem that the population can converge to a locally optimal partition.

5.7 Partitioning resources: a reduction in search-space

Consider again the scheduling problem introducing this chapter. We have a number of lists and an algorithm constructing a solution from these individual lists. Now the way one list is interpreted by the construction-algorithm may depend on the order of jobs in another list, so we cannot decompose the scheduling problem in independent sub-problems. This means that all of the job-lists that together represent one solution, cannot be processed independently, so we want these lists to belong to one single chromosome, a permutation for example. The representation of these lists in one single permutation can be done in one of the following ways:

- Each of the lists may be spread more or less randomly across the chromosome.
- Each of the lists may be allocated to a specified part of the chromosome.

The latter case would imply an enormous reduction, and can be easily implemented using the partition property. Just how large would this reduction be? Well, the number of possible permutations using the first method is just $n!$. Using the second method, the number of possible part-permutations for each I_i is $|I_i|!$, so the total number of possible permutations is $\Pi_{i=1}^{m} |I_i|!$. The reduction in search space then amounts to

$$\frac{n!}{\Pi_{i=1}^{m} |I_i|!} \tag{5.15}$$

To get an impression as to how large this reduction can get, suppose that the chromosome of length n is partitioned in m equally large pieces. Using the Stirling approximation $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ the reduction in space is in $O(n^{-m/2} m^n)$, which gets quite large for large chromosomes (and a number of parts ≥ 2). It seems that allocating each priority list to a specified whole part of the chromosome is a logical decision. However, there are very good reasons to decide the opposite.

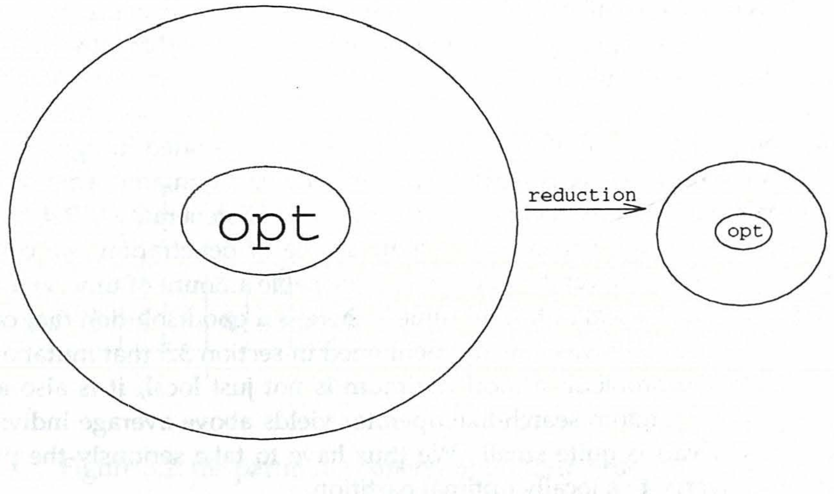

Figure 5.3: What good is the reduction in search space ?

First, we may very well doubt whether this reduction in search space is a practical aid for the genetic algorithm. The probability of finding an optimum at any point during search is for a large part determined by the **fraction** of the cardinality of the set of optimal solutions within the total set of solutions. The quotient of the sizes of the population and the total set of solutions has not much effect since the same chromosome may occur more than once in the population. So the reduction in search space only helps when the number of optimal solutions increases relative the the total number of solutions. Well, it doesn't. For each partitioned permutation, the number of corresponding non-partitioned permutations is given by formula 5.15, which is independent on the particular partitioned permutation. So, the reduction is the same for the number of optimal solutions and the the total number of solutions. This is illustrated in figure 5.3. It seems the genetic algorithm couldn't care less for the reduction in search space.

Second, the probability of getting in a local optimum increases for two reasons! In order to see this, it is important to have a clear understanding of the distinction between the partitioning deliberately forced upon the permutations for the sake of keeping job-lists together, and the unwanted partitioning that spontaneously occurs and gives rise to a local optimum. We thus claim that deliberately enforcing a partition stimulates in two ways the rise of an **unwanted** partition. The first reason is that enforcing a partition leaves us with with m permutations instead of one. Denote the probability of the spontaneous rise of an unwanted partition (within one single permutation) by p, and assume that p does not depend on the length of the permutation. The probability of not getting an unwanted partition is $1 - p$ using a single permutation, and (Bernoulli) $(1 - p)^m$ using m permutations. The probability of not getting an unwanted partition is thus a lot smaller. The second reason that we are more likely to get an unwanted partition, is that p **does** depend on the length of the permutation. When the length of a chromosome increases, it becomes more unlikely that a spontaneous partition arises. So p is a lot bigger for smaller permutations, which is exactly what we have when we enforce a partition on the original chromosome. So, there are two causes for an increased probability of obtaining an unwanted partition and thus getting in a local optimum.

We conclude that from the perspective of local optima, it is better to mix the individual priority lists altogether in one permutation than to make use of the partition-property.

Chapter 6

Scheduling in High Level Synthesis

6.1 Problem description and method of solving

High Level Synthesis (HLS) is part of the process of generating a chip-layout from a behavioral description of a chip. Perhaps the most important step within HLS is scheduling: the allocation of operations to resources in time. In a HLS scheduling problem, typical operations to be performed are multiplication, addition, and comparison. Operations are performed by resources called *modules*. In general it is possible that a module is able to perform different types of operations (ALU), but for our problem description we assume a one to one mapping of operation type and module type. A module performs an operation in a discrete number of *cycle steps*, which is the basic time-unit in a HLS scheduling problem. The number of cycle-steps a module needs to perform an operation is called *delay*. Typical delays used in this report are one cycle step for an adder-module and two cycle-steps for a multiplier-module. A HLS scheduling problem appears in different forms, dependent on the type of constraints. The basic problem has precedence constraints among operations, expressed in a directed acyclic graph (DAG) called the *Data Flow Graph* (DFG). The object is usually minimization of the *make-span*, the number of cycle-steps required to perform the schedule. A resource-constrained problem has additional constraints on the number of modules for each type of operation. The objective is minimization of the make-span. The type of scheduling-problem mostly encountered in practice is a time-constrained problem. In addition to the basic precedence-constraints it has a constraint on the make-span, and the objective is to minimize the number of modules needed to perform the schedule within the time-constraint. The last form of the HLS scheduling problem, has (in addition to the precedence constraints) both resource and precedence constraints. The objective is finding a feasible schedule. Because there is no actual *optimization* of any parameter, this problem is also called the feasibility problem. The feasibility problem is used in the method described in [Heij91] as an intermediate of solving the time-constrained problem. The idea is to use a heuristic to estimate the minimal set of modules (a lower bound) needed to schedule a DFG. This minimal set of modules is added as a constraint to the problems to obtain a feasibility problem. A scheduling heuristic is used to try to obtain a feasible schedule if there one. If the attempt succeeds we have obtained an *optimal* solution to the time-constrained scheduling problem because we used a minimal set of resources that nonetheless generates a feasible schedule. If the attempt fails, the minimal set of modules is extended and the loop is iterated. Note that in this case we can no longer guarantee optimality when a feasible solution is found.

We will use the genetic algorithm to solve the feasibility problem as an intermediate of solving the time-constrained problem as described above. Since the GA must handle an optimization problem, we have to find a way to turn the feasibility problem into an optimization problem. The reason we first turn a time-constrained (optimization) problem into a feasibility problem and then back into an optimization problem, is because we like to incorporate resource constraints. These type of constraints are handled quite easily by fast scheduling heuristics. Now a GA may not be a fast scheduling heuristic, but in order to turn a (chromosome) permutation into a schedule, we need some sort of scheduling heuristic, and resource-constrained scheduling heuristics are simply the fastest available. The way we turn a feasibility problem into an optimization problem is as follows: In the fast resource-constrained scheduling heuristic operations are fixed in time. During this process a lower bound on the make-span is derived. As soon as the lower bound exceeds the time-constrained, infeasibility is detected. The objective of the GA is to minimize the number of operations that are not allocated at the moment infeasability is detected. The next section introduces some fast scheduling heuristics that can be used either to turn a permutation into a schedule, or to derive a lower bound on the make-span in order to detect infeasibility.

6.2 Fast deterministic scheduling algorithms

In section 6.2.1 the fundamental and simplest scheduling algorithms are introduced that solve the basic form of the scheduling problem, constrained only by precedence relations. In section 6.2.2 the List scheduler is introduced, a resource constrained heuristic that is used in a GA in earlier research [Clui92]. Because this algorithm may exclude finding an optimal schedule for *any* given permutation, we develop a new resource constrained scheduler in section 6.2.3 that does not suffer from this disadvantage.

6.2.1 ASAP and ALAP scheduling

As soon as possible (ASAP) and as late as possible (ALAP) algorithms are schedulers that consider only precedence constraints. They minimize make-span, and the difference between the two is made clear by their names. The definitions of ASAP and ALAP-values include the notions $\delta(v)$, indicating the delay of operation v, and pred(v) and succ(v), indicating the set of predecessors and the set of successors resp. of operation v w.r.t. precedence relations. The asap-value asap(v) of an operation v is recursively defined as follows:

$$\text{asap}(v) = \begin{cases} 0 & \text{if pred}(v) = \emptyset \\ max_{v_i \in \text{pred}(v)}(\text{asap}(v_i) + \delta(v_i)) & \text{otherwise} \end{cases}$$

The alap-value alap(v) of an operation v is recursively defined as follows:

$$\text{alap}(v) = \begin{cases} t_{max} - \delta(v) & \text{if succ}(v) = \emptyset \\ min_{v_i \in \text{succ}(v)}\text{alap}(v_i) - \delta(v) & \text{otherwise} \end{cases}$$

The complexity of calculating the asap and alap values of all operations is $O(|V| + |E|)$, where V is the set of operations. In figure 6.1 an example is given of both types of schedule.

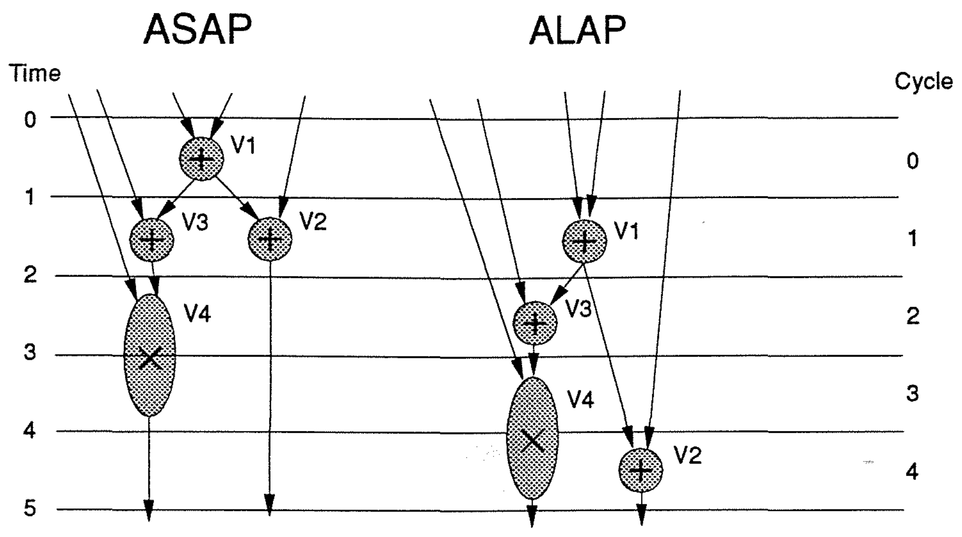

Figure 6.1: ASAP and ALAP schedule

ASAP and ALAP schedulers serve in general to obtain an approximation of an *execution interval*, which is the time-interval in which an operation can start execution. Because only precedence relations are taken into account, the asap and alap value of an operation are not more than a lowerbound and an upperbound resp. on the execution interval of the operation. The asap-alap interval will give an increasingly better approximation of the execution interval when more operations become fixed in time. When an operation becomes fixed in time (it is scheduled), the asap and alap value become equal, so that either the asap-value or the alap-value gets more tight. Because the asap and alap values are recursively computed, often a tighter bound is obtained on the execution-interval of potentially every operation. When a scheduling algorithm fixes an operation in time, it can use the recursive definitions of asap and alap to calculate the implications on the execution intervals of other operations. When the asap and alap value of other operations coincide as a result of the calculations, it is also fixed in time. When not enough resources are available at that specific time for that specific operation, infeasibility is detected. We conclude that a scheduler can use the asap and alap recursive computations each time an operation is fixed, in order to detect infeasibility. We will use this ability in some of the permutation schedulers explained in section 6.2.3.

6.2.2 List scheduling

A list scheduler handles the resource constrained scheduling problem. First reported in [Hu61], it is perhaps the most famous such algorithm in high-level synthesis today. The goal is to minimize the makespan while preserving precedence- and resource-constraints. Each time an operation is scheduled, the list scheduler updates the *free-list*, a list containing all operations that can be scheduled without violating the precedence constraints, that is, all operations having no unscheduled predecessors. Every cycle, for each type of operation a number of operations (at most equal to the resource constraint) is selected from the free list. Priority is given according to some properties of an operation such as a asap-value, or the number of successors, etc. An example of a list schedule is given in figure 6.2. In this example 2 adders and one multiplier

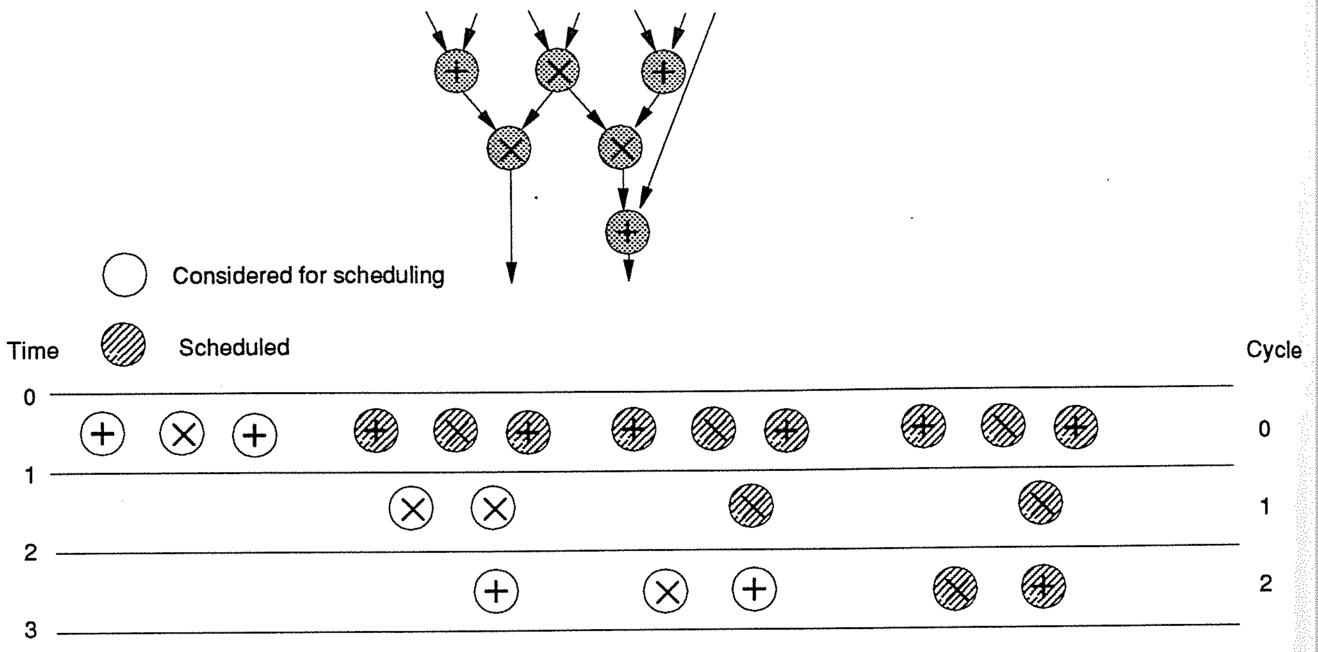

Figure 6.2: List scheduler

are available, all of which have a delay of one cycle. Priority is given according to the smallest alap-value of an operation. The make-span is 3 cycle-steps.

Priority can also be given according to the order in which the operations occur in a permutation, thus opening the way to use a list scheduler to produce a schedule from a permuttion. This is the way a list scheduler is used in the genetic approach of [Clui92]. To see the impact of the priority function on the makespan, consider the schedule when the left multiplication was scheduled before the right one, for example by priority of largest alap value: the makespan increases by one cycle. Judging from this impact, it makes sense to let a genetic algorithm find a suitable priority-function (a permutation). The disadvantage of using a list scheduler as permutation scheduler for our GA, is that for some DFGs there is no priority function that enables the algorithm to find an optimal schedule. This may occur when modules having different delays are available for scheduling. An example of this phenomenon is given in figure 6.3. The trouble is caused by the greedy nature of the list scheduler: Because it can schedule the left multiplication sooner than the right one, it does so despite the advantage of prolonging. We have to find a way to overcome this disadvantage, otherwise we cannot even guarantee the existence of a permutation yielding an optimal schedule.

6.2.3 Permutation-scheduling

In this section we will consider other fast scheduling heuristics that use information from a permutation in such a way that an optimal schedule is not excluded. The first such permutation scheduler that was used on the scheduling problem introduced in section 6.1 is straightforward: The permutation is searched from the head to the tail for an operation that is not yet scheduled. This operation is fixed at its asap. To calculate the implications of this on the asap-alap intervals

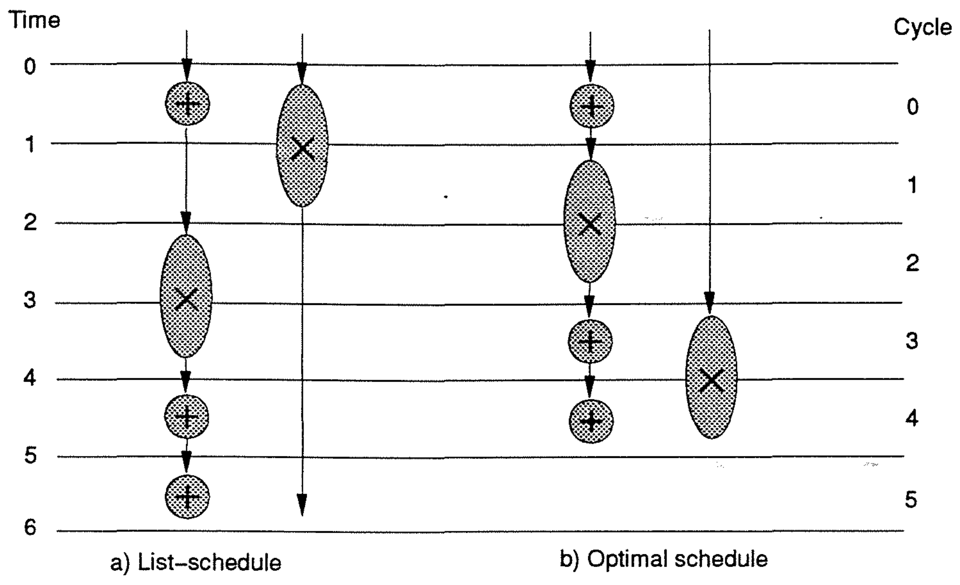

Figure 6.3: Greedy nature of List scheduler

of other operations, a *distance matrix* is used, as explained in section 6.3. Typically, a large fraction of the operations preceding this operation in the precedence graph, are fixed in time as a result. It is very well possible that some resource constraints are violated at the first operation processed in this way, so that infeasibility is detected immediately.

Because of the *order* in which the permutation is searched, it might seem at first sight that order-information in the permutation is relevant for this scheduler, and thus the algorithm obeys the conditions stated for a combination with uniform order crossover. However, when V_i is the set of operations preceding operation i in the data flow graph, and i is scheduled, most of the order-information w.r.t. V_i is rendered useless. The order (in the permutation) of two operations i and j usually only matters when they are not preceded (in the permutation) by an operation succeeding i and j in the precedence graph. The first conclusion is thus that this permutation scheduler is not a very good choice when used in combination with uniform order crossover. Another critical remark is the following. Because infeasibility can be detected very early in the process, for example when at most three operations in the permutation are chosen for scheduling, very little information is taken from the permutation. As a result, information is given by the scheduler on the quality of an extremely small portion of the permutation. One of the things that is appealing about a GA, as explained in the first chapter, is implicit parallelism: by evaluating a chromosome, a large number of schemas are evaluated (in parallel) as well. Surely this is not the case when only about three genes in a permutation are evaluated. We conclude that this permutation scheduler is not suitable in connection with a genetic algorithm. Empirical results lead to the same conclusion.

The second permutation scheduler we tried, is the following. An operation v_i is chosen based on earliest asap. The operation that gets scheduled is the first operation in the permutation that has a *resource conflict* with v_i. Two operations have a resource conflict when the following conditions are met:

- they are of the same type
- their execution intervals overlap
- they have no (transitive) relation in the precedence graph

Again, the asap-alap intervals of other operations are updated using a distance matrix. This scheduler is probably more suitable for uniform order crossover than the previous permutation scheduler, because order plays a bigger role now. We tried to overcome the last disadvantage of the previous permutation scheduler by first choosing an operation v_i based on earliest asap. This is no cure to the problem however: because execution intervals may be quite large, there is a large number of operations that have a resource conflict with v_i. This implies that the operation that is actually selected for scheduling, may be one much more near the bottom of the precedence graph than was intended. Fixing this operation at its asap typically fixes a number of other operations in time as well, so that we end up with the same problem we had with the previous permutation scheduler. Again, empirical results lead to the same conclusion.

The problem with both schedulers is that early in the schedule process, an operation can be selected for scheduling that lies deep within the precedence graph. To overcome this problem we would have to limit the choice to a set of operations that are not very remote (in the precedence graph) from operations already scheduled. This idea is already exploited in the list scheduler, and it is no surprise that the third permutation scheduler, a *path scheduler*, bears a large resemblance to the list scheduler. At any time, a list is kept (the *free list*), of all operations that have no unscheduled predecessors. The difference is that the list scheduler limits the choice of operation (within the free list) to one that could start execution at the current cycle-step. The path scheduler does not limit its choice in this way. It simply takes the first operation in the permutation that is also in the free list. The path scheduler is used in [Heij95]. In section 6.4 the results of applying the path scheduler are presented.

6.3 Producing a distance matrix

In section 6.2.3 we saw several ways to use information in a permutation to fix operations in time (to schedule operations). In section 6.2.1 we discussed that the asap-alap intervals of other operations are affected by fixing an operation. Infeasibility might even be detected in this way long before every operation is scheduled, so that useless computations can be avoided. Just how much the asap value or the alap value of an operation change as a result of scheduling another operation, is administrated in a *distance matrix*. The distance matrix D is a $V \times V$ matrix, such that each positive entry $D(i,j)$ represents the number of cycle-steps that operation j has to be scheduled after i in order to satisfy the precedence constraints. Before we continue to compute the values in the distance matrix, we should consider how to use them to update asap and alap values. When examining the recursive definitions of asap and alap values in section 6.2.1, it is obvious that fixing operation v_s has the following effect on the asap-alap interval of operation v:

- newasap(v) = MAX(asap(v), asap(v_s) + $D(v_s, v)$) when v succeeds v_s in the precedence graph.
- newalap(v) = MIN(alap(v), asap(v_s) - $D(v, v_s)$) when v preceeds v_s in the precedence graph.

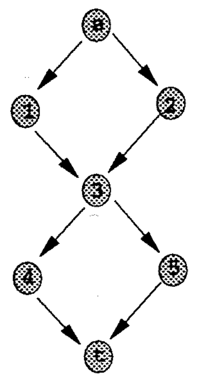

Figure 6.4: a DFG with 4 paths s->t

The update rules can be used unconditionally when $D(i,j)$ is set at $-\infty$ for i and j such that j does not succeed i in the precedence graph.

Obviously $D(i,j)$ must equal the maximum total delay of a sequence of operations that succeed operation i and precede operation j, according to the precedence constraints. When the branches in the precedence graph are labelled with the delay of the operation they branch from, the value of $D(i,j)$ corresponds to the value of a *longest path* in the graph from operation i to operation j. Computing the distance matrix thus amounts to finding the relative distances between every two operations, which is called an *all-pairs longest path* (APLP) problem. For general (cyclic) sparse graphs, the Johnson-algorithm [John77] is a very efficient way to solve this APLP problem in $O(V^2 \log V + VE)$ time. It is conceivable however, that an algorithm specifically targeted to acyclic sparse graphs can make efficient use of the DAG-structure. Such algorithms do exist for the *single source longest path* (SLP) problem of computing the longest path from a specified source to every other vertex in the graph. For the APLP problem for DAGs, no reference was found in literature. It is ofcourse possible to solve the APLP problem by repeating an SLP algorithm V times. However, efficient algorithms for the APLP problem for cyclic graphs are more efficient than algorithms that repeat an SLP algorithm V times, suggesting we could also do better for the acyclic case.

Let G be a DAG, and let G_i denote the subgraph of G induced by vertex i and all vertices reachable from i. Let V_i and E_i denote the vertex-set and the edge-set in G_i. $LP(G,i,j)$ denotes the length of the longest path from node i to j in graph G, and $arc(i,k)$ is the length of the arc from node i to k. We use the following recursive formula which holds for graphs in general:

$$LP(G,i,j) = \max_k\{arc(i,k) + LP(G,k,j)\} \tag{6.1}$$

When the recursion 6.1 would be used as a recursive algorithm, a superfluous amount of paths would be examined from i to j. For example, consider the DFG in figure 6.4. A recursive APLP algorithm is called for to find the longest path from s to t. This algorithm considers the paths (s,1,3,4,t), (s,1,3,5,t), (s,2,3,4,t), and (s,2,3,5,t). However, when the longest paths from vertex 1 and 2 are already known by earlier computation, only 2 possibilities need consideration: (s, LP(G,1,t)) and (s, LP(G,2,t). Constructing a longest path by using previously constructed longest paths is a technique called *memoization* [Corm90]. To make sure that assumed knowledge is indeed available, we have to process the vertices in a bottom-up fashion. This is accomplished by finding a *topological order*, an order of operations that respects the partial order implied by the DAG precedence graph. In [Corm90] an algorithm is described that finds a topological order in $O(E+V)$ time. We denote this algorithm by topological_sort(i). The topological order specifies the **reverse** order in which the vertices i are taken for determining the longest path to the vertices in V_i/i. For example, in the DFG of figure 6.4 first all longest paths from t are computed, followed by 4 and 5, then from 3, etc. Furthermore, each vertex i is generally connected to a limited number vertices, so we need to compute LP(G,i,j) only for some j. To accomplish this increased efficiency, we need to keep for each vertex i a list of the vertices in V_i/v_i. When traversing the reversed topological sorting to vertex j, the list of V_j is determined by uniting the lists of all its immediate successors. The list for the sink-vertex t is initiated as NULL. In summary, the computations performed for each vertex i is the following: First, list(i) is initiated as NULL. Second, for each immediate successor-vertex j and each vertex k in list(j), k is added to list (i), and the length of path (i,LP(G,j,k)) is computed. Of course, only the longest path is kept in memory. Third, the immediate successor-vertices are added to list(i). The computation of path (i,LP(G,j,k)) is done in $O(1)$ time, since both arc(i,j) and LP(G,j,k) are known. The addition of an element to a list may take $O(|list|)$ time in most implementations of a list, because the whole list should be checked whether or not the element is already in the list. A simple trick reduces the check-time to $O(1)$: All distances of longest paths (except single arcs) are initially set to $-\infty$. When a vertex k is added to the list(i), the distance of LP(G,i,k) is changed to a positive value. So when we want to add vertex k to list(i), we simply check whether or not LP(G,i,k) is still negative, and this is done in $O(1)$ time. We now formulate our APLP-algorithm for DAGs as follows.

Algorithm 1 *APLP for DAGs*

```
\\ determine topological order
Topological_sort(Source);

\\ initialization
for all i in V list(i).clear;
for all i in V  d_matrix(i,i) = 0;
for all (i,j<>i) in VxV {
   if (i,j) in E d_matrix(i,j) = delay(i);
   else          d_matrix(i,j) = MININF;
list(t) = NULL;

\\ APLP
for all i in V in reverse order of Topological_sort(Source) begin
    \\ make list(i) and update longest paths from i
    for all (i,j) in E begin
        for all k in list(j) begin
            if (d_matrix(i,k) == MININF)  list(i).add(k);
            d_matrix(i,k) = MAX(d_matrix(i,k), delay(i)+d_matrix(j,k));
        end
        list(i).add(j);
    end
end
```

Theorem 2 *Algorithm 1 solves the APLP problem for DAGs.*

Proof: The proof is by induction on a reversed topological sorting of G. The basis is at the sink: list(t)=NULL, and LP(G,t,k)=MININF for all k in V. The induction hypothesis for vertex i is that as a result of the reversed topological sorting, all immediate successors of i have already been processed, so LP(G,j,k) is determined for all immediate successors j of i, and all k reachable from j. But now, by formula 6.1, the algorithm computes LP(G,i,k) for all k reachable from i. Now vertex i has been processed and the next vertex in the reversed topological sorting is ready for processing.$\diamond$

Note that each arc (i,j) is observed exactly once, which is done in the computation for vertex i. For each arc (i,j) however, the number of computations equals $|$list(j)$|$, which is upper-bounded by V. Thus, the complexity of algorithm 1 is $O(EV)$. It should be noted that this worst-case bound is quite pessimistic, because $|$list(j)$|$ is typically much smaller than V. Furthermore, the amount of branches that go into a node, are typically one or two, thus making E having the same order as V. This effectively makes algorithm 1 run in $O(V^2)$ time, for data flow graphs.

6.4 Results

In this section we will review some of the results of running the GA on a a couple of benchmark problems on a HP 9000/735 workstation. The following additional implementation issues may be of importance. The population consists of 100 individuals. Another 100 individuals are created using crossover. No other operators are used. From the total of 200 individuals, 100 are selected to be discarded, based on their score. The remaining 100 individuals are subjected to selection for crossover, etc.
The benchmark used is the fast discrete cosine transform (fdct) with time constraints of 11, 14, and 18 clock cycles. Other benchmarks yielded trivial results (optimal solution in generation 0) and are therefore discarded. The GA is run until a population is obtained carrying at least one individual yielding a feasible schedule, or until 100 generations have passed without finding a feasible schedule. CPU-time is given in table 6.1 for each benchmark, using four (randomly chosen) different seeds. In one case (seed 4, fdct18) no solution was found. To compare these results with other algorithms, a time constrained list scheduler [Heij91], an improved force directed scheduler [Verh91], and a previous genetic scheduler [Clui92] were unable to find a feasible schedule for fdct14 and fdct18. In table 6.1 the stochastic character of the genetic algorithm is illustrated by the differences in the runs obtained by different seeds. This is a disadvantage of many probabilistic approaches, because it does not yield a very robust algorithm.

Table 6.1: CPU-times for GA with different seeds

fdct	1	2	3	4
11	4.65	2.82	2.82	5.21
14	4.84	14.57	9.89	20.32
18	10.74	11.69	12.44	∞

The Boltzmann adaptive selection mechanism explained in section 5.5 has also been run with various combinations of parameter values. Update mechanism 5.13 is used with the integration constant α equal to 0.25, the variation sensitivity a taken from the set $\{1, 1.5, 2, 2.5, 3\}$, and the target variation v taken from the set $\{0.2, 0.3, 0.5\}$ The results are averaged over four runs using different seeds and are given in table 6.2 for fdct14 and in table 6.3 for fdct18. In these tables,

Table 6.2: CPU-times for GA with Boltzmann adaptive selection, fdct14

$\downarrow$ v, a $\rightarrow$	1	1.5	2	2.5	3
0.2	11.9	21.85	11.23	12.51	10.53
0.3	13.63	22.98	13.0	10.12†	8.48
0.5	‡	18.63†	22.08	21.42	14.7†

Table 6.3: CPU-times for GA with Boltzmann adaptive selection, fdct18

$\downarrow$ v, a $\rightarrow$	1	1.5	2	2.5	3
0.2	11.4	20.23	8.33	9.26	11.84
0.3	16.39	43.13	6.35	12.22	8.71
0.5	‡	22.84†	14.91	16.71	12.01

† indicates that the average is taken over three runs because the fourth run did not result in a feasible schedule. ‡ indicates that none of the runs were successful.

From tables 6.2 and 6.3 we conclude that adaptive Boltzmann selection is certainly competitive with roulette wheel selection. For some combinations of parameter values, Boltzmann selection is even superior, especially for fdct18. Because this benchmark introduces more 'freedom' (operations are less restricted by timing constraints), more scheduling-decisions are due to information from the optimal permutation (instead of the scheduler). This suggests that for larger problems, Boltzmann adaptive selection is increasingly better (for some parameter values) than roulette wheel selection. The trouble however, is that no systematic increase in performance can be derived from these tables, as one of the parameters increases (or decreases). This suggests that either there is a lot of interaction between the two parameters, or the selection mechanism is very robust.

Chapter 7

Conclusions

The major conclusions drawn from the work leading to this thesis, are summarized as follows:

- For bitstring representations, a relation (equation 4.4) between the statistical values of two subsequent generations of a population is derived analytically. Extensive literature research has not revealed the analytical establishment of such a relation. A number of phenomena, observed in practice and reported in literature, can be directly derived from this relation. In particular, the importance of variance is made clear by the equation.
- For a permutation representation and uniform order crossover, a new schema theorem has been derived taking into account both survival and combination. This is in contrast to the traditional schema theorem from [Holl75], which has troubled the minds of many subsequent researchers by laying emphasis on survival.
- Analyses yield highly controversial results when considered in the perspective of exploration-exploitation. Analyses for both bitstring representations and a permutation representation suggest that the crossover mechanism should be as disruptive (explorative) as possible to obtain the best results. This stands in high contrast with the traditional view, caused by the traditional schema theorem laying emphasis on survival, which is a very exploitative idea. My suggestion is that many researchers in the GA-community should reconsider their ideas in this respect.
- A GA works best when interactions between search space and genetic operator are considered. Early implementations of our GA did not work because the characteristics of the scheduler did not match the characteristics of the crossover mechanism. Too often a genetic algorithm is considered as a black box by putting problems in and expecting solutions to come out. A genetic algorithm is a complex system that needs engineering before it works well.
- For DFG scheduling, uniform order crossover yields good results in our implementation. Run time efficiency has been improved by a new $O(VE)$ algorithm for solving an all pairs longest path problem (APLP) for acyclic graphs.
- Adaptive Boltzmann selection yields results competitive with traditional roulette wheel selection, by balancing the tradeoff between exploration and exploitation, dependent on the relative variation in the population. Results suggests that adaptive Boltzmann selection will yield increasingly better results when larger problems are considered.

Appendix A

Derivations

A.1 Statistical derivations

A.1.1 m-th order score moments

In the following we derive an expression of the m-th score-moment of the $i+1$-th population in terms of the expected performance of the crossover mechanism. We repeat equations 4.1 and 4.2 as A.1 and A.2:

$$E_{i+1}[s^m] = \sum_s p_{i+1}(s)s^m \tag{A.1}$$

$$p_{i+1}(s) = \sum_{x,y \epsilon P_i} p(x,y)p(s(cros(x,y)) = s|x,y) \tag{A.2}$$

Equations A.1 and A.2 combine to

$$\begin{aligned} E_{i+1}[s^m] &= \sum_s \sum_{x,y \epsilon P_i} p(x,y)p(s(cros(x,y)) = s|x,y)s^m && \text{(A.3)} \\ &= \sum_{x,y \epsilon P_i} p(x,y) \sum_s p(s(cros(x,y)) = s|x,y)s^m && \text{(A.4)} \end{aligned}$$

The last step is due to exchange of the summations. Because

$$p(s(cros(x,y)) = s|x,y) = \sum_{x':s(x')=s} p(cros(x,y) = x'|x,y)$$

(where $s(x')$ is the score of genotype x'), it follows that

$$\sum_{s} p(s(cros(x,y)) = s|x,y)s^m = \sum_{x'\epsilon G} p(cros(x,y) = x'|x,y)s^m(x') \tag{A.5}$$

where G is the collection of all genotypes. Substitution of A.5 in A.4 yields:

$$\begin{aligned} E_{i+1}[s^m] &= \sum_{x,y\epsilon P_i} p(x,y) \sum_{x'\epsilon G} p(cros(x,y) = x'|x,y)s^m(x') & \text{(A.6)} \\ &= \sum_{x,y\epsilon P_i} p(x,y)E[s^m(cros(x,y))|x,y] & \text{(A.7)} \end{aligned}$$

Here $E[s^m(cros(x,y))|x,y]$ is the m-th moment of the score s of the offspring given the parents x and y from the i-th generation, using the probabilities $p(cros(x,y) = x'|x,y)$.

A.1.2 Two offspring

In the following we prove that it makes no difference to let crossover make one or two offspring. We assume that within the pair (x,y), x and y are the first and second parent resp. An assumption on the selection scheme will be that $p(x,y) = p(y,x)$. Although formally a restriction, no selection schemes are known in which the assumption is violated. The crossover scheme will choose the crossover positions and these positions will be notated by the binary vector h.The first child will be produced by choosing the genes according to h from x and according to the complementary mask h_c from y. The first child is denoted $cross_h(x,y)$. The second child will be produced just the other way around, so it is denoted $cross_{h_c}(x,y)$. Instead of summing once over all individuals in the population, we have to sum twice, and each probability on an offspring-pair has to be multiplied with the m-th power of the score of both the first and second child. In the derivation we use $p(x',y'|x,y)$ as shorthand for $p(cross_h(x,y) = x', cross_{h_c}(x,y) = y'|x,y)$

$$E_{i+1}[s^m] = \frac{1}{2} \sum_{x,y\epsilon P_i} p(x,y) \sum_{x',y'\epsilon G} p(x',y'|x,y)[s^m(x') + s^m(y')] \tag{A.8}$$

The factor $\frac{1}{2}$ is due to double summation: Each sum $[s^m(x') + s^m(y')]$ is evaluated both in the term $p(x,y)p(x',y'|x,y)[s^m(x') + s^m(y')]$ and in the term $p(y,x)p(y',x'|y,x)[s^m(y') + s^m(x')]$. Since we assume that $p(x,y) = p(y,x)$ and we know that $p(y',x'|y,x) = p(x',y'|x,y)$ (because they use the same mask), the context of $[s^m(x') + s^m(y')]$ is the same in both cases. Thus each sum is evaluated twice in exactly the same context.

To reduce the two-offspring case to the one-offspring case, we need to express $p(x',y'|x,y)$ in terms of $p(cross_h(x,y) = x'|x,y)$. This relation is determined as follows: First we use the chain rule to obtain

$$p(x',y'|x,y) = p(cross_h(x,y) = x'|x,y) \times p(cross_{h_c}(x,y) = y'|x',x,y) \tag{A.9}$$

Observe $p(cross_{h_c}(x,y) = y'|x',x,y)$. When the parents and the first child are known, we can try to reconstruct the mask h. This cannot be accomplished completely, however: when a gene has the same value in both parents the value of the corresponding mask-bit does not matter. So we construct an equivalent mask h' in the following way. When a gene in x' has the same value has the gene in x, the corresponding mask-bit is set to one, otherwise it is zero. Now we can also construct the complemented mask h'_c. So we derive

$$\begin{aligned} & p(cross_{h_c}(x,y) = y'|x',x,y) \\ = \quad & p(cross_{h_c}(x,y) = y'|h',x,y) \\ = \quad & p(cross_{h_c}(x,y) = y'|h'_c,x,y) \end{aligned}$$

But because the parents and the complemented mask are known, the second child is determined. Thus, the probability of the second child being equal to y' is either zero (when they are not equal) or one (when they are equal). We use this information in equation A.9 to obtain

$$p(x',y'|x,y) = \begin{cases} p(cross_h(x,y) = x'|x,y) & \text{if } \exists h : x' = cross_h(x,y) \text{ and } y' = cross_{h_c}(x,y) \\ 0 & \text{otherwise} \end{cases} \tag{A.10}$$

Substitution of A.10 in A.8 gives

$$E_{i+1}[s^m] = \frac{1}{2} \sum_{x,y \epsilon P_i} p(x,y) \sum_{x' \epsilon G} p(cross_h(x,y) = x'|x,y)[s^m(x') + s^m(y')] \tag{A.11}$$

where y' is now taken as $cross_{h_c}(x,y)$ for h such that $x' = cross_h(x,y)$. We split the sum $[s^m(x') + s^m(y')]$ and examine the following resulting term:

$$\begin{aligned} & p(x,y) \sum_{x' \epsilon G} p(cross_h(x,y) = x'|x,y) s^m(y') \\ = \quad & p(y,x) \sum_{y' \epsilon G} p(cross_h(x,y) = y'|y,x) s^m(y') \\ = \quad & p(x,y) \sum_{x' \epsilon G} p(cross_h(x,y) = x'|x,y) s^m(x') \end{aligned}$$

The first step is valid because $p(x,y) = p(y,x)$, summation over x' covers the same elements as summation over its 'complement' y', and $p(x'|x,y) = p(y'|y,x)$ (same mask). The last step is due to exchange of the variables x' and y'.
So the two terms we split up from A.11 turn out to have the same value! Now, using this equality we can directly obtain A.6 and thus A.7 from substitution of the equality in A.11.

A.1.3 Expected scores

In the following we derive a relation between the expected scores of two subsequent populations. We assume proportionate selection: $p(x) = \frac{s(x)}{S_i}$, where the normalizing constant S_i is the total score of the i-th generation, $\sum_{x' \in P} s(x')$. This gives the following expression for $p(x, y)$:

$$p(x,y) = p(x)p(y) = \frac{s(x)s(y)}{S_i^2} \tag{A.12}$$

Using A.12 in equation A.11 with $m = 1$ we get:

$$E_{i+1}[s] = \frac{1}{2} \sum_{x,y \epsilon P_i} \frac{s(x)s(y)}{S_i^2} \sum_{x' \epsilon G} p(cross_h(x,y) = x'|x,y)[s(x') + s(y')] \tag{A.13}$$

Now, using our assumption on the sum of scores, it follows:

$$\begin{aligned}
E_{i+1}[s] &= \frac{1}{2S_i^2} \sum_{x,y \epsilon P_i} s(x)s(y) \sum_{x' \epsilon G} p(cross_h(x,y) = x'|x,y)[s(x) + s(y)] \\
&= \frac{1}{2S_i^2} \sum_{x,y \epsilon P_i} \{s^2(x)s(y) + s(x)s^2(y)\} \sum_{x' \epsilon G} p(cross_h(x,y) = x'|x,y) \\
&= \frac{1}{2S_i^2} \sum_{x,y \epsilon P_i} \{s^2(x)s(y) + s(x)s^2(y)\} \\
&= \frac{1}{S_i^2} \sum_{x,y \epsilon P_i} s^2(x)s(y) \\
&= \frac{1}{S_i^2} \sum_{x \epsilon P_i} s^2(x) \sum_{y \epsilon P_i} s(y) \\
&= \frac{1}{S_i} \sum_{x \epsilon P_i} s^2(x) \\
&= \frac{1}{E_i[s]|P_i|} \sum_{x \epsilon P_i} s^2(x) \\
&= \frac{E_i[s^2]}{E_i[s]} \qquad \text{(A.14)} \\
&= \frac{\text{var}_i[s] + E_i^2[s]}{E_i[s]} \\
&= E_i[s] + \frac{\text{var}_i[s]}{E_i[s]} \qquad \text{(A.15)}
\end{aligned}$$

A.1.4 Crossover-statistics for the all-one problem

We want to calculate the following expression.

$$E[s^m|s(x), s(y)] = \sum_{s_c} p(s_c|s(x), s(y))s_c^m \tag{A.16}$$

In this expression, $p(s_c|s(x), s(y))$ has a binomial distribution:

$$p(s_c|s(x), s(y)) = \binom{l}{s_c} p_1^{s_c}(1-p_1)^{l-s_c} \tag{A.17}$$

Our first approach is to use a normal distribution instead. The binomial distribution is closely approximated by the normal distribution for large l, and in practice the latter is used instead of the former because of mathematical convenience. So we use the probability density function

$$f(s) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(s-\mu)^2/2\sigma^2} \tag{A.18}$$

for $p(s_c|s(x), s(y))$ where $\mu = E[s|s(x), s(y)] = lp_1$ and $\sigma = lp_1(1-p_1)$. The summation in A.16 now becomes an integration, and we derive:

$$\begin{aligned}
E[s^m|s(x), s(y)] &= \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} s^m e^{-(s-\mu)^2/2\sigma^2} ds \\
&= \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} (s+\mu)^m e^{-s^2/2\sigma^2} ds \\
&= \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} (s+\mu)^{m-1} s e^{-s^2/2\sigma^2} ds \\
&+ \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} (s+\mu)^{m-1} \mu e^{-s^2/2\sigma^2} ds \\
&= \frac{1}{\sigma\sqrt{2\pi}} \sigma^2 \left[(s+\mu)^{m-1} e^{-s^2/2\sigma^2}\right]_{-\infty}^{\infty} \\
&+ \frac{1}{\sigma\sqrt{2\pi}} \sigma^2 (m-1) \int_{-\infty}^{\infty} (s+\mu)^{m-2} e^{-s^2/2\sigma^2} ds \\
&+ \mu E[s^{m-1}|s(x), s(y)] \\
&= \sigma^2(m-1)E[s^{m-2}|s(x), s(y)] + \mu E[s^{m-1}|s(x), s(y)]
\end{aligned} \tag{A.19}$$

Equation A.19 defines a recursive relation for $E[s^m|s(x), s(y)]$. Using generating functions, this recursive relation can be solved by solving a differential equation of the first order. This however comes down to finding the primitive of e^{-x^2}, which doesn't exist. Other ways of trying to calculate $E[s^m|s(x), s(y)]$ using the normal distribution reduced to the same recursive relation A.19. Other ways of solving the recursive relation were not found. We conclude that this approach fails in calculating $E[s^m|s(x), s(y)]$.

In our second approach we stick to the binomial distribution, and use the *moment generating function*. The moment generating function $\Gamma(z)$ is a transformation on a probability distribution p_n defined by

$$\Gamma(z) = E[z^n] = \sum_{n=-\infty}^{\infty} p_n z^n \tag{A.20}$$

Differentiating A.20 k times gives

$$\Gamma^{(k)}(z) = E\left[\frac{n!}{(n-k)!} z^{n-k}\right] \tag{A.21}$$

Substitution of $z = 1$ in A.21 gives

$$\Gamma^{(k)}(1) = E\left[\frac{n!}{(n-k)!}\right] \tag{A.22}$$

This is an exact formula for a sum of moments, which can be rewritten to an expression for the moment $E[n^k]$ in terms of all lower order moments. Note that $E[n^k] = 0$ if $k > n$. If we neglect the lower order terms, A.22 gives an approximation for $E[n^k]$. If the sum is not too large however, it can be calculated without too much effort. Let's first compute the moment generating function $\Gamma(z)$ for our binomial distribution:

$$\Gamma(z) = \sum_{s=0}^{l} \binom{l}{s} p_1^s (1-p_1)^{l-s} z^s = (p_1 z + (1-p_1))^l \tag{A.23}$$

Where p_1 is calculated by 4.5. Now differentiating A.23 m times and substituting $z = 1$ gives

$$\Gamma^m(1) = \frac{l!}{(l-m)!} p_1^m \tag{A.24}$$

Substituting A.22 (conditional on x and y) with $n = s$ and $k = m$ in A.24 gives the equality

$$E\left[\frac{s!}{(s-m)!} | x, y\right] = \frac{l!}{(l-m)!} p_1^m \tag{A.25}$$

Comparing this expression to the one derived in our first approach, equation A.19, we find that things have only gotten worse: instead of a recurrence relation for the $m+1$-th moment in terms of the m-th moment we now implicitly have a a recurrence relation for the $m+1$-th moment in terms of *all* lower order moments.

A.1.5 Crossover variation for the all-one problem

We try out formula A.25 on the second order moment. The left-hand side of equation A.25 equals

$$E\left[\frac{s!}{(s-2)!}|x,y\right] = E[s(s-1)|x,y] = E[s^2|x,y] - E[s|x,y] = E[s^2|x,y] - lp_1 \qquad \text{(A.26)}$$

The right-hand side of equation A.25 equals

$$\frac{l!}{(l-2)!}p_1^2 = l(l-1)p_1^2 = (l^2-l)p_1^2 \qquad \text{(A.27)}$$

Combining A.26 and A.27 gives

$$E[s^2|x,y] = lp_1(1-p_1) + l^2p_1^2 \qquad \text{(A.28)}$$

Now we have calculated the conditional second order moment in terms of the bit-mask probability r (implicit in p_1). Substitution of expression A.28 in A.4 with $m = 2$ gives the second order moment of the $(i+1)$-th generations score:

$$E_{i+1}[s^2] = \sum_{x,y \epsilon P_i} p(x,y)\left[lp_1(1-p_1) + l^2p_1^2\right] \qquad \text{(A.29)}$$

We substitute 4.5 for p_1 and, assuming proportionate selection (equation A.12) for $p(x,y)$:

$$\begin{aligned} E_{i+1}[s^2] &= \sum_{x,y \epsilon P_i} \frac{s(x)s(y)}{S_i^2}\left[lp_1(1-p_1) + l^2p_1^2\right] \\ &= \sum_{x,y \epsilon P_i} \frac{s(x)s(y)}{S_i^2}\left[(s(x)r + s(y)(1-r))^2 + (s(x)r + s(y)(1-r))(1 - \frac{s(x)}{l}r - \frac{s(y)}{l}(1-r))\right] \\ &= \frac{E_i[s^2]}{E_i[s]} + (1-\frac{1}{l})\left[\frac{E_i[s^3]}{E_i[s]}(r^2 + (1-r)^2) + \frac{E_i[s^2]}{E_i[s]^2}2r(1-r)\right] \end{aligned} \qquad \text{(A.30)}$$

We maximize A.30 by setting its derivative (with respect to r) to 0:

$$\frac{dE_{i+1}[s^2]}{dr} = 2(1-\frac{1}{l})(2r-1)\left[\frac{E_i[s^3]}{E_i[s]} - \frac{E_i[s^2]}{E_i[s]^2}\right] \qquad \text{(A.31)}$$

Setting A.31 to 0, we get $r = \frac{1}{2}$.

A.2 A necessary condition for directed search

A linear model assumes $s(i) = a \times i$. We derive from equation 5.6:

$$\begin{aligned}
E[N(k,g+1)] &= \frac{|P|}{k!}\frac{1}{E_g^2[s]}\sum_{i=0}^{k}\binom{k}{i}r^i(1-r)^{k-i}s(i)s(k-i) \\
&= \frac{|P|}{k!}\frac{1}{E_g^2[s]}\sum_{i=1}^{k-1}\binom{k}{i}r^i(1-r)^{k-i}aia(k-i) \\
&= \frac{|P|}{k!}\frac{a^2}{E_g^2[s]}\sum_{i=1}^{k-1}\frac{k!}{i!(k-i)!}i(k-i)r^i(1-r)^{k-i} \\
&= \frac{|P|}{k!}\frac{a^2}{E_g^2[s]}\sum_{i=1}^{k-1}\frac{k!}{(i-1)!(k-i-1)!}r^i(1-r)^{k-i} \\
&= \frac{|P|}{k!}\frac{a^2k(k-1)}{E_g^2[s]}\sum_{i=1}^{k-1}\frac{(k-2)!}{(i-1)!((k-2)-(i-1))!}r^i(1-r)^{k-i} \\
&= \frac{|P|}{k!}\frac{s(k)s(k-1)r(1-r)}{E_g^2[s]}\sum_{i=1}^{k-1}\binom{k-2}{i-1}r^{i-1}(1-r)^{k-2-(i-1)} \\
&= \frac{|P|}{k!}\frac{s(k)s(k-1)r(1-r)}{E_g^2[s]}\sum_{i=0}^{k-2}\binom{k-2}{i}r^i(1-r)^{k-2-i} \\
&= \frac{|P|}{k!}\frac{s(k)s(k-1)r(1-r)}{E_g^2[s]}(r+(1-r))^{k-2} \\
&= \frac{|P|}{k!}\frac{s(k)s(k-1)r(1-r)}{E_g^2[s]} \\
&\approx \frac{|P|}{k!}\frac{s^2(k)}{E_g^2[s]}r(1-r) \qquad \text{(A.32)}
\end{aligned}$$

A.3 Sensitivity analysis for adaptive boltzman-selection

We start the analysis by rewriting equation 5.10, the probability of selection using the boltzmann-selection scheme.

$$\begin{aligned} sel(x) &= \frac{e^{p_b s(x)}}{\sum_{x' \in P} e^{p_b s(x')}} \\ &= \frac{1}{\sum_{x' \in P} e^{p_b (s(x') - s(x))}} \end{aligned} \tag{A.33}$$

We differentiate expression A.33 with respect to p_b, the variable controlling selection pressure.

$$\begin{aligned} \frac{\delta sel(x)}{\delta p_b} &= \frac{1}{\sum^2_{x' \in P} e^{p_b (s(x') - s(x))}} \times p_b \sum_{x' \in P} e^{p_b (s(x') - s(x))} \\ &= \frac{p_b}{\sum_{x' \in P} e^{p_b (s(x') - s(x))}} \end{aligned} \tag{A.34}$$

We multiply left and right-hand side with δp_b, and we divide the left and right-hand side by the left and right-hand side resp. of equation A.33. Furthermore the δ is replaced by Δ.

$$\begin{aligned} \left| \frac{\Delta sel(x)}{sel(x)} \right| &= |p_b \Delta p_b| \\ &= p_b^2 \left| \frac{\Delta p_b}{p_b} \right| \end{aligned} \tag{A.35}$$

At first sight it might seem that the following variation improves the sensitivity: Use $-p_b e^{s(x)}$ in the nominator of equation A.33 instead of $e^{-p_b s(x)}$. When comparing the taylor expansion of these two terms, the alternative indeed depends less on the Boltzmann variable p_b. However, the denominator of equation A.33 is also affected by this variation, and alas, p_b is completely cancelled in the whole expression. This simple variation does not diminish the sensitivity.

A.3.1 Linear updating

The linear update rule (equation 5.11) is repeated as equation A.36.

$$p_b = a \times var + b \tag{A.36}$$

Both sides are differentiated with respect to var:

$$\frac{\delta p_b}{\delta var} = a \tag{A.37}$$

From this equation we derive:

$$\left|\frac{\Delta p_b}{p_b}\right| = \frac{a \times var}{p_b}\left|\frac{\Delta var}{var}\right| \tag{A.38}$$

Substitution of equation A.38 in equation A.35 gives:

$$\left|\frac{\Delta sel(x)}{sel(x)}\right| = a \times p_b \times var\left|\frac{\Delta var}{var}\right| \tag{A.39}$$

A.3.2 Linear updating with integration

The second update rule (equation 5.13) is repeated as equation A.40.

$$p_b[i] = \alpha \times p_b[i-1] + (1-\alpha)(a \times var[i] + b) \tag{A.40}$$

Now we assume that before a sudden change in variation, as a result of the discovery of a new score-champion, the process was at equilibrium: $p_b[i-1] = 0$. It also implies that the variation was at $var[i-1] = -\frac{b}{a}$. Substitution in equation A.40 now gives

$$\begin{aligned}\Delta p_b[i] &= p_b[i] - p_b[i-1]\\ &= (1-\alpha)(a \times var[i] + b)\\ &= (1-\alpha)(a \times (\Delta var[i] + var[i-1]) + b)\\ &= (1-\alpha)a \times \Delta var[i]\end{aligned} \tag{A.41}$$

Finally, substitution of equation A.41 in equation A.35 gives:

$$\left|\frac{\Delta sel(x)}{sel(x)}\right| = (1-\alpha) \times a \times p_b[i] \times var[i]\left|\frac{\Delta var[i]}{var[i]}\right| \tag{A.42}$$

[Eshe89] L.J. Eshelman, R.A. Caruana, and J.D. Schaffer. Biasses in the crossover landscape. In J.D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 10--19, George Mason, June 1989. Morgan Kaufmann.

[Gare79] M.R. Garey and D.S. Johnson. *Computers and intractability : a guide to the theory of NP-completeness*. Freeman, 1979.

[Gold89a] D.E. Goldberg, editor. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[Gold89b] D.E. Goldberg. Zen and the art of genetic algorithms. In J.D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 80--85, George Mason, June 1989. Morgan Kaufmann.

[Gold90] D.E. Goldberg. Construction of high-order deceptive functions using low-order Walsh coefficients. Illigal Report 90002, University of Illinois, Urbana-Champaign, 1990.

[Gref85] John J. Grefenstette, editor. *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications*, Pittsburgh, July 1985. Erlbaum Associates.

[Gref86] J.J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):122--128, January 1986.

[Gref87] John J. Grefenstette, editor. *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*, MIT, Cambridge, July 1987. Erlbaum Associates.

[Heij91] M.J.M. Heijligers. Time Constrained Scheduling for High Level Synthesis. Master's thesis, Eindhoven University of Technology, May 1991.

[Heij95] M.J.M. Heijligers, L.J.M. Cluitmans, and J.A.G. Jess. High-level Synthesis Scheduling and Allocation using Genetic Algorithms. In *submitted to Asia and South Pacific Design Automation Conference*, September 1995.

[Hofs79] D.E. Hofstadter. *Godel, Escher, Bach*. Vintage, 1979.

[Holl75] J.H. Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

[Holl85] J.H. Holland. Properties of the Bucket Brigade. In John J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications*, pages 1--7, Pittsburgh, July 1985. Erlbaum Associates.

[Hu61] T.C. Hu. Parallel sequencing and assembly line problems. *Operation Research*, 9(6):841--848, November 1961.

[Huli91] M. Hulin. Analysis of schema distributions. In R.K. Belew and L. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 204--209, San Diego, July 1991. Morgan Kaufmann.

[John77] D.B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1--13, 1977.

[Karg92] H. Kargupta, K. Deb, and D.E. Goldberg. Ordering genetic algorithms and deception. Illigal Report 92006, University of Illinois, Urbana-Champaign, 1992.

[Koza92] J.R. Koza. *Genetic Programming; on the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[Mart91] R. San Martin and J.P. Knight. Genetic Algorithms for Optimization of Integrated Circuits Synthesis. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 432--438. Morgan Kaufmann, 1991.

[Mich92] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Artificial Intelligence. Springer Verlag, 1992.

[Scha87] J.D. Schaffer and A. Morishima. An adaptive crossover distribution mechanism for genetic algorithms. In John J. Grefenstette, editor, *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*, pages 36--40, MIT, Cambridge, July 1987. Erlbaum Associates.

[Scha89] J.D. Schaffer, editor. *Proceedings of the 3rd International Conference on Genetic Algorithms*, George Mason, June 1989. Morgan Kaufmann.

[Spea91] W.M. Spears and K.A. DeJong. On the virtues of parameterized uniform crossover. In R.K. Belew and L. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 230--236, San Diego, July 1991. Morgan Kaufmann.

[Spea92] W.M. Spears. Adapting crossover in a genetic algorithm. Report AIC-92-025, Naval Research Laboratory AI Center, Washington, 1992.

[Star91] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Withley. A Comparison of Genetic Sequencing Operators. In R.K. Belew and L. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 69--76, San Diego, July 1991. Morgan Kaufmann.

[Sysw89] G. Syswerda. Uniform crossover in genetic algorithms. In J.D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 2--9, George Mason, June 1989. Morgan Kaufmann.

[Sysw91] G. Syswerda. Schedule optimization using genetic algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms*, pages 332--349. Van Nostrand Reinhold, 1991.

[Verh91] W.F.J. Verhaegh, E.H.L. Aarts, J.H.M. Korst, and P.E.R. Lippens. Improved Force-Directed Scheduling. In *Proceedings of the European Conference on Design Automation*, pages 430--435. IEEE Computer Society Press, February 1991.

[Wehn91] N. Wehn, M. Held, and M. Glesner. A Novel Scheduling and Allocation Approach for Datapath Synthesis based on Genetic Paradigms. In *IFIP Working Conference on Logic and Architecture Synthesis*, pages 47--56, Paris, 1991.