# Evolutionary Algorithms and Scheduling Problems

Sieuwert van Otterloo

August 7, 2002

# Contents

# Preface

This thesis has been written in the academic year 2001-2002 as a final project for obtaining the title Master of Science in Cognitive Artificial Intelligence at the Universiteit Utrecht. This project has been done mainly by Sieuwert van Otterloo, with much help from Peter Bosman. Dirk Thierens and Vincent van Oostrom have also been involved as supervisors and referees. I would like to thank Renze Steenhuisen and Jan Kok for helping me collect hardcopies of the literature, the organization of BNAIS 2002 for giving me the chance to present my work in a workshop, and all people in my environment who supported me during my presence at the Universiteit Utrecht.

We would like to thank all authors listed in the bibliography who have made their articles available on the internet. This has made it much easier for us to obtain their work because the Universiteit Utrecht does not have all titles in its library. We would also like to apologise for the incompleteness of the information on some sources. These sources have been downloaded from the internet and you can probably find them by using a search engine. These are the bibliographic data for this source, in case you would like to refer to it.

| | |
|---|---|
| author | Sieuwert van Otterloo |
| title | Evolutionary Algorithms and Scheduling Problems |
| institute | Universiteit Utrecht |
| year | 2002 |
| web-address | www.bluering.nl |

I hope you enjoy reading this text. If you feel something needs to be corrected or if you have other ideas you would like to share with me, do not hesitate to contact me. The best way to reach me will probably be by email.

Sieuwert van Otterloo
Reinoutsgaarde 9
3436 RA Nieuwegein
sieuwert@bluering.nl

# Chapter 1

# Introduction

## 1.1 Motivation

In this thesis two subjects are combined: evolutionary algorithms and the field of scheduling problems. The order of terms in our title is significant. Our primary goal is to learn how we can successfully apply evolutionary algorithms and we use scheduling as a source of benchmark problems in our experiments. Much work in both areas already exists and unfortunately much has also been written on the combination of these subjects. It has not been possible to read and do justice to all previous work so completeness is not one of the qualities of this thesis. We do hope you find it readable, interesting, self-contained, that you are occasionally surprised by the results and that you will have the feeling that you have learned something valuable after reading.

The goal of the research project that resulted in this thesis has been to test the newly developed $\mathbb{ICE}$ algorithm on some hard-to-solve scheduling problems. We have performed a series of experiments that we have used to answer the following questions:

- What are the important aspects of an evolutionary algorithm in order to solve scheduling problems?
- How effective is $\mathbb{ICE}$ compared to less advanced algorithms and previous results in solving real world problems like scheduling problems?

## 1.2 Contents in brief

### 1.2.1 Evolutionary algorithms

This chapter starts by introducing the more restricted idea of Genetic Algorithms and the basic crossover operators that we use. In the next section an idea on how to use evolutionary algorithms on permutations is explained. After that the $\mathbb{ICE}$ algorithm is explained in detail. It is contrasted with similar algorithms and finally we compare evolutionary algorithms to more traditional methods of problem solving.

### 1.2.2 Scheduling

The goal of a scheduling problem is to make a schedule for a certain situation. In the problem it is stated what must take place, and the solution describes exactly what should happen at what time.

These problems occur in factories, personnel planning, airfields, railroad stations etcetera and have been studied by Operations Research people for a long time (our bibliography goes back to 1965).

Since mathematical descriptions of scheduling problems are often distilled from practical situations there are many variants of scheduling problems. A selection had to be made which problems we wanted to study. We have chosen problems with a simple definition in which a lot of practical details have been left out.

In the chapter on scheduling we give definitions of the problems we are trying to solve, introduce important concepts (properties, bounds, definitions) from the field of scheduling and we discuss the schedulers that we use to convert the output of the evolutionary algorithms into schedules.

The last section takes a small detour into theoretical computer science in order to make precise that scheduling problems are hard to solve.

### 1.2.3 Experiments

A great number of plots and figures are used in the experimental chapter to explain what new experiments we have done and what the results are. The chapter is divided in three sections according to three different classes of scheduling problems. The order approximates the order in which the experiments were performed. The largest section is the one on job shop scheduling because it explains additional theory like forcing, parallel evolutionary algorithms and local search.

### 1.2.4 Conclusion

As usual everything said will be rephrased and put together in the final chapter. Our findings for each of the problems are summarized in their own section. At the end some general results regarding how to benchmark evolutionary algorithms are listed.

# Chapter 2

# Evolutionary Algorithms

## 2.1 Introduction

`Evolutionary Algorithms` (`EA`) is the term used in this text to describe search methods based on the mechanics of natural selection and evolution. This field is often divided into three subfields: evolution strategies, genetic algorithms and genetic programming. We will omit describing genetic programming, which is concerned with the stochastic search for good programs, because knowledge of that subfield is not necessary for understanding this paper.

The term `evolution strategies` was introduced by Rechenberg and Schwefel around 1960 [1]. Evolution strategies are often used to find optima of real-valued functions. Applications were for instance optimization of the shape of a bended pipe or drag minimization of a joint plate. It works mainly by applying changes on a candidate solution. The actual application of these changes are self-adaptive. The first evolution strategies maintained only one candidate solution at the same time, but very soon variants emerged maintaining a pool of two or more solutions.

John Holland independently developed `Genetic Algorithms`(`GA`) in the 1960s [21]. They are not simply optimization techniques but are also claimed to mimic certain aspects of natural evolution. Like in evolution strategies random mutation is used and selection, but an important difference to previous approaches is that information from different individuals is combined using so-called crossover or recombination operators. The biological analogue is sexual reproduction, and biological terminology is often used. Two parent strings are combined to form a new child string. Genetic Algorithms are often defined on binary strings, so in order to use standard crossover operators a binary encoding has to be found for each problem. The operators only work on strings of single bits, not on more advanced structures like graphs or trees. Since there is interaction between candidate solutions by means of crossover the population size and structure are very important in GAs, while they are less important in Evolution Strategies.

In the basic genetic algorithm a population of bitstrings $P_0$ is generated randomly. For a certain number of rounds a new population $P_{i+1}$ is generated by selecting strings from $P_i$ as parents for the next generation, copying those parents or crossing them over, and possibly mutating. Because selection is based on the fitness of the string as a solution for the problem at hand, it is hoped that the average fitness of each new generation is higher than the average fitness of the previous population. Whether this actually happens depends on the suitability of the representation and the recombination for the current problem.

One has a lot of freedom when implementing genetic algorithms. Important decisions to be made, for which there is no standard solution, are the following:

**selection:** Which strings are selected for contributing to the next generation is very important for the performance. For instance: in roulette wheel selection individuals are assigned a probability to be chosen proportional to their fitness. In rank based selection the probability only depends on the relative ordering of the fitnesses. In tournament selection two strings are picked at random, compared to each other and the winner is chosen to be a parent. See for instance the books [16] or [27].

**crossover operator:** For bit strings common crossover operators are uniform and one-point crossover. Each operator is good at preserving certain patterns, but can easily destroy other patterns. In our work we have compared uniform and one-point crossover against $\mathbb{ICE}$, a more advanced crossover operator explained on page 15.

**mutation operator:** A mutation operator introduces a small difference in a string, allowing the algorithm to walk to the optimum in small steps. Again general operators and specific operators for certain problems exist.

**population size:** What quality of solution is reached in what time depends heavily on the size of the population.

**termination criterion:** If one does not know what the optimum value is, or if one is just looking for an acceptable solution, it is often hard to say how many rounds the algorithm should be allowed to run. In theory the population should converge to one solution and the algorithm should terminate exactly after this has happened. unfortunately it is not sure that convergence will happen within reasonable time(or even whether it will happen at all) and in that case the algorithm must be stopped at a certain point.

EA research tries to develop theories that can guide implementers of Evolutionary Algorithms in making choices for the above and other aspects[36][17]. If it would be possible to predict beforehand how good a certain combination of choices and settings work it would certainly increase the value of EAs as a tool for problem solving.

### 2.1.1 Crossover operators

Two crossover operators have received much attention because they are very different and both have good theoretical properties.

One-point crossover is designed to combine two parent strings into a new string such that many properties of the parents are inherited by the child. The underlying assumption is that properties are represented by groups of adjacent bits in the parent strings. It would be unfortunate to break up these groups, so one-point crossover tries to make the least number of cuts in each string. It cuts both strings at exactly one place, and chooses the first half of the first string, the second half of the second parent and glues them together. The position where the cut is made is determined at random.

Let $p(0)p(1)\dots p(n-1)$ and $q(0)q(1)\dots q(n-1)$ be two parent strings that are input to the operator and $t(0)t(1)\dots t(n-1)$ the outcome of one-point crossover. One-point crossover is then best described in the following procedure:

```
1. $s = \text{random}(n-1)$
2. for $i = 0$ to $n$
```

```
    (a) if $i \leq s$ $t(i) = p(i)$ else $t(i) = q(i)$
```

The procedure random($x$) returns a number randomly chosen from $\{0, 1, \ldots, x-1\}$.

$$P(\text{random}(x) = i) = \frac{1}{x} \text{ if } 0 \leq i < x$$

Uniform crossover is designed to mix the properties of the two parents as much as possible. It assumes a property is represented by a combination of a small number of bits, not necessarily adjacent. For each position in the output string it chooses randomly a parent whose value at that position is copied. In a procedure it reads like this:

```
1. for $i = 0$ to $n$
    (a) if random(2)=0 $u(i) = p(i)$ else $u(i) = q(i)$
```

Note that given two parent strings the set of possible outcomes is much larger for uniform crossover than for one-point crossover. The parent strings 000000 and 111111 give an extreme example. In this case uniform crossover can produce any string as outcome, and thus has $2^6$ possible outcomes, while one-point crossover only has strings of the form $0^a1^{6-a}$ for $0 < a < 6$.

One-point crossover is sensitive to the position in a string of certain bits, while uniform crossover is not. This can be both an advantage and a disadvantage, and is something to take into account when designing a representation.

The above operators are formulated for working on bit strings, but the definitions are also sound for strings of different type, for instance strings of real numbers.

### 2.1.2 Schemata Theory

One attempt to explain the workings of GAs that has been very successful is schemata theory. It is described in [21] and [27]. Recall that GAs operate on fixed length bit strings. A `schema` (plural `schemata`) is a pattern of the same length as the strings, containing 0,1 and *. A schema $S$ matches a string $T$ if $T$ contains a 0 where $S$ contains a 0 and a 1 where $S$ contains a 1. Either value is allowed on the positions where $S$ contains *. For example: The schema 0**1 matches 0011 and 0101, but not 1001. Each string is matched by a large number of schemata and one can say that it represents all matching schemata at the same time. This means that a population of strings can also be seen as a multiset of schemata. During the running time certain schemata will be represented more often, while others are not represented at all. The successful schemata probably describe combinations of bits also present in the optimal solution.

Good schemata are often called `building blocks` [16]. Each schema has a `defining length` and an `order`. The order of a schema is the number of fixed positions, while the defining length is the maximal distance between two fixed positions minus one (it is the number of cut positions breaking up the schema). For instance the schema *1**1 has order 2 and defining length 3, while **100 has order 3 and defining length 2. Most crossover operators are designed to be good at preserving 'short' schemata. The word `short` for a schema can refer to both size measures, depending on the problem, the crossover operator and the intentions of the designer of the algorithm.

In a run of a Genetic Algorithm only short schemata are present more than once at the beginning. The good ones among those will be become more common because of selection and crossover. The algorithm will try to combine those schemata into larger schemata. For instance if 11**** 10****, *10*** and *01*** are good schemata it is very likely strings will emerge matching 101*** and 110***, because they can be created when crossing over two instances of the popular schemata. If one of these has a very good fitness it will become dominant in the population and thus a large order schema has been found. This process of finding larger building blocks will continue until the algorithm terminates. Which larger schemata will emerge depends on the crossover operator because schemata with a larger chance of undisrupted inheritance from one parent have an advantage.

This theory explains the efficiency of Genetic Algorithms. A large number of schemata are competing through a relatively small number of strings, so a lot of work is done in few evaluations. On the basis of this analysis one can create problems that are easy and hard to solve for GA. In the next table examples are given of a trivial problem, bitcounting, and a hard problem, called a `deceptive problem`.

| String | bitcounting fitness | deceptive fitness |
|---|---|---|
| 000 | 0 | 4 |
| 001 | 1 | 2 |
| 010 | 1 | 2 |
| 011 | 2 | 0 |
| 100 | 1 | 2 |
| 101 | 2 | 0 |
| 110 | 2 | 0 |
| 111 | 3 | 5 |

For both problems 111 is the optimal solution. In the bitcounting problem schemata matched by 111 are always better than schemata not matched by it. This means the algorithm is directed in the right direction by the short schemata and will 'easily' find the right solution in a small number of evaluations and with a small population size. This is true for both one-point and uniform crossover.

In the deceptive problem all schemata of order 1 and 2 matched by 000 are at least as good as other schemata, and the same is true for all schemata of defining length 1 and 2. This means the GA can only find the solution if it discovers the point 111 by random exploration. The schemata do not help the GA. Genetic algorithms can solve deceptive problems but need a larger population and a larger number of evaluations. The encoding of the problem is in this case very important. Problems are called deceptive if the lower order schemata (all schemata up to a certain order $k$) do not point to the global solution.

Using this theory one can say something about the suitability of one-point crossover and uniform crossover. The chance a building block is inherited completely from one of the parents during uniform crossover is not dependent on the defining length of the schema. It only depends on the order. So a short ordered building block like 1*****1 has a fair chance of survival in uniform crossover, while a large order building block like 10101**** has not.

In one-point crossover things are the other way around. Large order schemata can be inherited completely as long as they have a small defining length. The schema 1*****1 has a negligible chance[1] of being inherited in one-point crossover. The schema 10101**** does have a good chance, namely 25%, to survive crossover. So which crossover is best depends on the nature of the successful building blocks, whether they tend to have a small order or not. This is not always clear from the problem description so it can be worthwhile to try both forms of crossover (or other kinds of crossover).

[1]the chance is zero in our definition of one-point crossover, but in different textbooks it is allowed to cut strings before the first symbol, in which case the child is an exact copy of one of the parents

Note that in the bit counting problem there is in fact no interaction between the three variables: The fitness function can be written as the sum of the fitness of three functions each taking one position as input. We call this kind of fitness function an `additively decomposable function`. A function $f(\vec{x})$ is additively decomposable if certain sets $a$ and $b$ non-overlapping exist and functions $g$ and $h$ such that

$$f(x_1, x_2, \ldots, x_n) = g(x_{a_1}, x_{a_2}, \ldots, x_{a_k}) + h(x_{b_1}, x_{b_2}, \ldots, x_{b_l})$$

The components of the decomposable function can be decomposable itself. The formula above can be applied recursively to decompose a function in nondecomposable factors $f^i$:

$$f(x_1, x_2, \ldots, x_n) = \sum_{i=0}^{m-1} f^i(x_{a_0^i}, x_{a_1^i}, \ldots, x_{a^i_{|a^i|-1}})$$

## 2.2 Density Estimation Algorithms

A genetic algorithm is one way to model evolution. In can also be done in different fashions and these alternatives can be more effective in dealing with certain problems. The most important point of this section is explaining the crossover operator used in $\mathbb{ICE}$. In order to do so first ID$\mathbb{E}$As are explained because that is where $\mathbb{ICE}$ is derived from. At the end several other algorithms similar in structure or motivation are discussed.

### 2.2.1 ID$\mathbb{E}$A

Iterated Density Estimation Evolutionary Algorithms (ID$\mathbb{E}$As) are based on a different way of looking at search algorithms. They are designed to find a probability distribution that models the set of selected solutions. This distribution is represented by a `probability density function` (`pdf`). A GA can be said to implicitly model a pdf that is represented by the population of strings, but the pdf is never explicitly calculated nor are any samples drawn from it. In an ID$\mathbb{E}$A a population of individuals is generated from the current pdf. Selection is applied, leaving only 'better' strings. Then a new pdf is learned from the selection and this is the next current pdf and is used to generate new strings. This process is repeated until a termination criterion is satisfied. The formal description of an ID$\mathbb{E}$A(based on [6]) is given below. The parameters are a population size $n$, a selected fraction $\tau \in [0, 1]$, a number of new samples $m$, a replacement strategy $R(O, N)$ (where $O$ is an old population and $N$ a set of new individuals, and the outcome a subset of $N \cup O$), a termination criterion $T$, a function to be optimised $f : D \to E$ and a procedure that returns a random sample from $D$ based on the probability model $M$ called ran$(M, D)$.

**ID$\mathbb{E}$A**

```
1. Let $M$ be the uniform pdf on $D$
2. $P = \emptyset$
3. while $|P| < n$
    (a) $P \leftarrow P \cup \{ran(M, D)\}$
4. while $\neg T$
```

```
    (a) Select $P'$, $P' \subseteq P$, $|P'| = \lfloor \tau n \rfloor$ based on $f$
    (b) Estimate a new model $M$ based on $P'$
    (c) $O = \emptyset$
    (d) while $|O| < m$
        i. $O \leftarrow O \cup \{ran(M, D)\}$
    (e) $P = R(P', O)$
5. return $\text{argmax}_{x \in P}\{f(x)\}$
```

The performance of an IDEA depends heavily on the restrictions based on the model $M$. A very common form for $M$ is a model with a fixed number of parameters. For instance one can assume the model to be a Gaussian function. To fully specify this function only a mean vector $\mu$ and a set of variances $V$ must be estimated from the samples and that is relatively easy. The formula for the density function $f$ over $\mathbf{R}^n$ of such a model $M$ is:

$$f(\vec{x}) = \frac{1}{(2\pi)^{d/2}|V|^{1/2}} e^{-\frac{1}{2}(\vec{x}-\mu)'V^{-1}(\vec{x}-\mu)}$$

$M$ consists of a mean vector $\mu$ and a $n$ by $n$ matrix of covariances $V$, where $V_{ii}$ is the variance of the $i$th coordinate and $V_{ij}$ the covariance between the $i$th and $j$th coordinate.

This model is flexible and in many cases too flexible: If the dimensionality $n$ is high it is hard to estimate all the parameters right. It seems reasonable to assume some of the coordinates $x_i$ and $x_j$ independent (which means that their covariance is zero). At the same time you do not want to assume all of the variables independent because that might yield a too simple model. A solution is to split the model learning phase in two stages. These phases are called `model selection` and `model fitting`. In the model selection phase the algorithm tries to find a parameterised model $M$ with a good trade-off between complexity of $M$ (number of parameters) and the suitability of $M$ for describing the current set of samples. After a model has been selected model fitting takes place. In this phase the algorithm estimates the best values for the parameters of the model. Because the models considered often have some nice mathematical properties the best values for the parameters can often be found by evaluating quite simple formulas. For instance a mean vector $\mu$ when modelling a set $P$ is calculated by $\mu = \frac{1}{|P|}\sum_{\vec{x} \in P} \vec{x}$.

In the implementation of an IDEA the separation in two phases is not always present because one needs to have the right values for certain parameters to say something about the suitability of a certain model.

In the model selection phase of our IDEA the coordinates of the domain $\mathbf{R}^n$ are divided into smaller groups, and only covariances inside groups are considered. A model $M$ consists under these assumptions of $M = (\mu, F, V)$. $F$ is a factorization of the numbers $1, \ldots, n$ into groups and V is a covariance matrix with zeroes at the right places:

$$\cup f \in F = \{1, \ldots, n\}$$

$$\forall f, g \; (f, g \in F \wedge f \neq g) \Rightarrow f \cap g = \emptyset$$

$$\forall i, j \; (\neg \exists f \in F \; i \in f \wedge j \in f) \Rightarrow V_{i,j} = 0$$

In the model selection phase finding the right factorization is done by choosing the model with the greatest penalized likelihood. The model $M$ is chosen for which the $\prod_{x \in P'} f(x) - E(|M|)$ is maximal, where $E$ is some sort of penalty term depending on the number of parameters of $M$. Without a penalty term the most complicated model would always be chosen.

### 2.2.2 $\mathbb{ICE}$

$\mathbb{ICE}$ stands for IDEA Induced Chromosome Elements Exchanger and is described in [8] and [7]. It is a variant of the IDEA algorithm described above that is more like a genetic algorithm. It functions quite like ID$\mathbb{E}$A, except that the model learned is not used to generate new samples. The factorization contained in the model is used to do crossover on two selected elements. The complete algorithm is described below, where $X(F, x, y)$ is a crossover operator for the elements $x$ and $y$ that uses the factorization $F$.

```
1. Let $M$ be the uniform pdf on $D$
2. $P = \emptyset$
3. while $|P| < n$
   (a) $P \leftarrow P \cup \{ran(M, D)\}$
4. while $\neg T$
   (a) Select $P'$, $P' \subseteq P$, $|P'| = \lfloor \tau n \rfloor$ such that $\forall a \in P'$ $\forall b \in P$
       $P'$ $f(a) \geq f(b)$
   (b) Estimate a new model $M$ based on $P'$ where $M = (\mu, F, V)$
   (c) $O = P'$
   (d) while $|O| < n$
       i. $O \leftarrow O \cup \{X(F, x \in P', y \in P')\}$
   (e) $P = O$
5. return $\text{argmax}_{x \in P}\{f(x)\}$
```

The crossover operator $X$ applies uniform crossover on the groups $f$ in the factorization $F$. This means that all coordinates belonging to the same group will be inherited from the same parent. Which group is inherited from which parent is determined at random and independently, like in uniform crossover. For $\vec{r} = X(\vec{p}, \vec{q})$ this is described in the following formula:

- For each $f \in F$
  - $\text{if}(\text{random}(2) = 1)$ then $\forall i \in f$ $r_i = p_i$ else $r_i = q_i$

An example of this operator on the parent strings $p = 000000$ and $q = \mathbf{111111}$ where the model uses a factorization $F = \{\{0, 2, 3\}, \{1\}, \{4, 5\}\}$ could have yielded the result $r = 0\mathbf{1}00\mathbf{11}$.

The kind of model used for $\mathbb{ICE}$ can be the Gaussian model described above. Note that in $\mathbb{ICE}$ no samples are drawn from the model. Only the factorization is needed so the model fitting phase can be omitted. It is necessary to specify the kind of pdf used because it matters for comparing the suitability of factorizations.

Permutation $\mathbb{ICE}$ is a variant of $\mathbb{ICE}$ that can only be applied to permutation problems. It makes use of the random keys representation described on page 17. In the search to find the best factorization it does not assume a Gaussian model, but it looks at the represented permutations directly. It creates permutation frequency tables to count how often certain orderings occur. Permutation $\mathbb{ICE}$ can outperform normal $\mathbb{ICE}$ on a certain class of additively decomposable functions.

It also has certain drawbacks: because the frequency tables are quite large a limit on the group size inside a factorization must be set. This algorithm can also be very time consuming.

The permutation frequency tables count correlations of a subset of positions of random keys. The algorithm for finding a factorization starts with the factorization with a separate group for each position:

$$f = \{\{1\}, \{2\}, \ldots, \{n-1\}\}$$

The algorithm tries to improve upon this factorization by considering joining two groups together or swapping two elements of different groups(this is ofcourse only useful if one of the groups contains more than one element). To do this it calculates the entropy of the new group(s) and compares it with the entropy of the old groups. In order to give the reader some idea of how this works the procedure for calculating the entropy is given. In the next algorithm $G$ is a group of positions and $G_p$ is the permutation of the elements of $G$ inside the larger permutation $p$. For example let $p = (521043)$ and $G = \{1, 3, 4\}$. We are only interested in the permutation of the elements 1,3 and 4 and they appear with the last two numbers swapped, so $G_p = (021)$, which is the permutation of size 3 with the last two number swapped. We want to count these permutations and to do that a function $\rho$ is needed that assigns an integer to each permutation of the size of $G$. Which function you take is not important, as long as it is a bijection between permutations of size $|G|$ and the numbers $\{0, 1, \ldots, |G|! - 1\}$.

The following procedure estimates the entropy in a population $P$ in the elements at the position indicated by the group $G$.

- For all $i \in \{0, 1, \ldots, |G|! - 1\}$ $A[i] = 0$
- For each $p \in P$ do $A[\rho(G_p)] = A[\rho(G_p)] + \frac{1}{|P|}$
- $E = \sum_{i=0}^{|G|-1} -A[i] * \log_2(A[i])$

A low entropy means that there is some structure in the population at the indicated position, which could mean that these positions form a building block. If that is the case the group should be preserved in crossover. In the actual implementation an error term must again be introduced because otherwise the factorization with only one group would again be chosen. The table $A$ needs to be very large if $G$ is large and for this reason a limit of 6 or 7 must be set on the size of $G$.

### 2.2.3 Previous work

The use of probabilistic models in evolutionary algorithms has been used in several approaches. The motivation is often to overcome the difficulties that simple GAs have in solving certain problems because of the destruction of building blocks in crossover. In [33] several algorithms are briefly mentioned and referred to. EDA (Estimation of Distribution Algorithm) is a general scheme for algorithms that estimate a distribution for the current population and use this distribution to generate new strings for replacing some of the current strings[28]. Several algorithms implement this scheme but using estimates in which each variable is considered independent: the population based incremental learning algorithm (PBIL)[2] the compact genetic algorithm (cGA)[18] and the univariate marginal distribution algorithm (UMDA)[2].

The selfish gene algorithm [23][10] has been inspired by Richard Dawkins famous book *The selfish gene*. It is inspired by the idea that evolution is about genes, not about individuals. The individuals are only vehicles for the genes to survive. The selfish gene algorithm (abbreviated FGA for unstated reasons) maintains a table $T$ with probabilities for each value of each variable:

$T_{i,a} = P(x_i = a)$. During the run two samples are drawn from this distribution, they are compared and the probabilities used to generate the winner are increased, the losers chances are reduced. In this model also all variables are considered independent, so it is similar to the algorithms described above.

The algorithms mentioned above are not very well suited for solving problems with interactions between the variables. Several algorithms therefore try to detect pairwise interaction, like the bivariate marginal distribution algorithms (BMDA)[32], the MIMIC algorithm and the incremental algorithm using dependency trees (these are also mentioned in [33]). The class of problems solvable by these algorithms is larger than the class of problems solved by the proposals in the previous paragraph, but is still very limited.

The Bayesian Optimization Algorithm (BOA) defined in [33] fits the description of an EDA and uses a Bayesian Network as a model. This network encodes relations between variables. In the BOA asymmetric relations (one-way) between variables are learned instead of the symmetric relations encoded in a factorization.

MIMIC[12] is an acronym for Mutual-Information-Maximizing Input. It uses only pairwise probabilities so the pdf is factorised like this

$$\hat{p}(X) = p(X_{i_1}|X_{i_2})p(X_{i_2}|X_{i_3})\dots p(X_{i_{n-1}}|X_{i_n})P(X_{i_n})$$

This algorithm is defined for binary strings so $p(A|B)$ is fully defined by $p(1|0)$ and $p(1|1)$. The pdf can be represented in table form.

In [19] the importance of linkage learning is discussed and the extended compact genetic algorithm (ECGA) is proposed. The factorised distribution algorithm (FDA) [29] uses a factorization that in this case is described by:

$$p(X) = \prod_{i=1}^{l} p(X_{b_i}|X_{c_i})$$

In this formula the $b_i$ and $c_i$ are sets of indices so the factorization is more general than the one used in MIMIC. Because it is asymmetric (directed) it is also more general than what is used in IDEA. The authors conjecture that finding an optimal factorization for a certain population is NP-hard and propose a practical algorithm for finding good (not necessarily optimal) factorizations.

## 2.3 Random Keys for permutation problems

Permutations are bijective functions from a finite set to itself. If we take for a finite set the set of numbers $\{0, \dots, n-1\}$ a permutation is a new ordering for this set of numbers. A permutation $P$ can then be written as the list $P(0)P(1)\dots P(n-1)$. In many optimization problems the goal is to find the best permutation for a certain set of input objects. These problems are permutation optimization problems(POP). In order to apply Evolutionary Algorithms to POP one must have a crossover operator and sometimes a mutation operator for permutations. Finding such operators is not trivial, since the standard operators do not work on permutations written as lists. This is demonstrated in the next example, where one-point crossover on two lists generates a new list that does not correspond to any permutation. It corresponds to a function, but the function is not a bijection.

- parent 1:(9403726815)
- parent 2:(**2071538649**)

- result: (94037||**38649**)

In order to overcome this problem several alternative crossovers have been proposed. Which crossover is best depends on the properties of the problem that must be solved. For scheduling problems the ordering of elements is often important (and not just the neighbour information like in the travelling salesman problem). A representation for permutations that respects ordering is proposed in [3] and is called random keys. In the random keys framework a permutation of $n$ elements is represented by a vector $v \in [0,1]^n$. Each element or key $v_i \in v$ is initially randomly chosen. In order to find the list representation of $v$ the elements of $v$ are sorted in increasing order, and the original positions of the elements are output in this new order. For example

- input: [0.5 0.33 0.8 0.7]
- append the original positions to each number: [(0:0.5) (1:0.33) (2:0.8) (3:0.7)]
- sort this list on the keys [(1:0.33) (0:0.5) (3:0.7) (2:0.8)]
- output the original positions in this new order (1 0 3 2)

This representation transforms a permutation problem to a real-valued optimization problem in $[0,1]^n$. Note that the function to be optimised is piecewise flat with a lot of sudden jumps. This means the function does not have a meaningful derivative (it is 0 or undefined), so many (gradient-based) optimization methods for this space will not work. Another bad property is that every permutation is represented by an infinite number of points in $[0,1]^n$. This seems very bad because people often try to design representations with a one to one correspondence. If such correspondence is present the search space of the algorithm is not larger than the space of all permutations. The good news is that one can use standard crossover operators on this representation. An example for one-point crossover is given:

- parent 1: [0.1 0.3 0.5 0.7 0.9] (0 1 2 3 4)
- parent 2: [0.8 0.2 0.6 0.4 0.0] (3 1 4 2 0)
- result: [0.8 0.2 | 0.5 0.7 0.9] (1 2 3 0 4)

In the article [3] various applications of this representation are given, including scheduling, routing, travelling salesman and quadratic assignment problem. While the one algorithm designed around the random keys representation does not beat all previous results, it is shown that the random keys approach is effective on a wide range of problems.

## 2.4 Traditional solution methods

Scheduling and other optimization problems were formulated independently from evolutionary algorithms and it is no surprise that different methods can be used to solve them.

The simplest algorithm for optimization is `exhaustive search`: evaluate all schedules(points in search space) and return the schedule with the best fitness. This method is practical for small problems but becomes very slow if the problem size increases. The time needed for exhaustive search grows exponentially with the size of the problem.

`Branch and bound` is a general principle often used in Operations Research problems. It is a slightly more efficient way to search than exhaustive search and it also finds the optimal schedule.

In practice it can be much faster than exhaustive search but it does not run in polynomial time (see the section of page 30 for a definition of polynomial time).

When searching for a schedule in a branch and bound fashion you try to evaluate all schedules by summing them up in a tree structure. Each leaf is a schedule, each inner node a partial schedule. A child is created by adding an unscheduled operation to the schedule. In each inner node one can estimate a lower bound on the achievable makespan (by summing up the time all remaining operations take on each machine and each job) of each schedule that can be obtained from that partial schedule. If that lower bound is already higher than the best result so far, all schedules containing the current partial schedule can be skipped (this is an entire branch of the search tree).

A good stochastic way of solving scheduling problems is `simulated annealing`. Stochastic means that the method uses guessing or choice based on probability. A stochastic method never guarantees to find an optimal solution (or even a good solution). In simulated annealing for each point in the search space $x$ a neighbourhood $N(x)$ is defined. $T(t)$ is the temperature function depending on the current iteration $t$. The temperature decreases through time. The path $x_i, i = 0, \ldots, t_{max}$ is then generated by the following procedure:

```
1. $t = 1$
2. Select $x_0$ at random
3. while $t < t_{max}$
    (a) Select $x_t$ at random according to $P(x_t = y) = S(f(y), T(t))$ if $y \in N(x_{t-1})$ and 0 otherwise.
    (b) $t = t + 1$
4. return $x_{t_{max}}$
```

The function $S$ defines a probability depending on the fitness $f(y)$ and the temperature of the system. If the temperature is low the probability is very dependent on the fitness, so that the next $x_t$ with the highest fitness must be chosen. If the temperature is high the choice is more at random. This gives the algorithm the opportunity to escape a local optimum in the beginning, while convergence to a solution in a later stage is assured. Simulated annealing is inspired by the physical process of cooling down of metal objects.

Another successful stochastic method is `tabu search`[15]. In tabu search again a neighbourhood $N(x)$ is used and a path through the search space is again generated. To avoid walking in circles a history $H(t)$ containing information about the points $x_{t-1}, x_{t-2}, \ldots, x_{t-i}$ is maintained. A point is chosen that is not described by the information in the history.

```
1. $t = 1$
2. Select $x_0$ at random
3. while $t < t_{max}$
    (a) Select $x_t$ from $N(x_{t-1})$ such that $x_t$ is not in or described by $H(t)$
    (b) $t = t + 1$
4. return $x_{t_{max}}$
```

The descriptions of the two methods above are intentionally vague because these are only algorithm templates that can be implemented in very different ways. What the neighbourhood is and how the history is calculated depends on the problem that must be solved.

These methods differ from evolutionary algorithms because they use a population of only one candidate and only mutation, not crossover. Both methods are `local search` methods. They explore the space 'close' to the points they already know. It is essential for evolutionary algorithms that they combine solutions, allowing them to jump to an entire new point that is promised to be very good, not because it is close to a good point, but because it is a combination of two good points. Crossover can be seen as some sort of `global search` method.

Another important distinction is the one between `blind methods` and `problem-specific methods`. A blind method needs to know nothing about the search space. It takes a function, evaluates a number of different points, uses its bias to discover structure on its own (this is often done implicit, since GAs for instance do not output any structural description) and returns a hopefully optimal point. This blindness is what makes it easy for programmers to implement GAs and EAs.

The traditional methods above can be seen as problem-specific methods. The general description can be seen as a template where a lot of details need to be filled in. In order to implement it for a specific problem, neighbourhoods, steps and other terms have to be translated into the domain of the problem by an experienced researcher. This makes these methods much harder to apply.

The distinction sketched above is a little bit artificial. To implement successfully an EA it is certainly useful to have some idea of the problem at hand, and standard methods and representations for use in traditional methods are often at hand. The distinction is also artificial because of the existence of evolutionary algorithms designed for very specific problems. One can even say that the use of non-blind EAs is a current development in EA research and predict that the different fields in optimization will eventually merge. Nevertheless it is useful to keep this distinction in mind when comparing different methods. It also helps to understand what makes evolutionary algorithms interesting for Artificial Intelligence.

# Chapter 3

# Scheduling

Scheduling problems come in different flavours. In the introduction we have already mentioned that we concentrate on abstract problems not containing to much detail. We have made another decision to focus on problems with easy constraints because they appeared to be most suitable for solving by stochastic methods. In problems with difficult constraints it is very hard to come up with even one schedule meeting all demands stated in the problem. For this thesis we concentrate on problems where this is not very hard, and we are searching for the best schedule according to a certain criterion.

In our terminology we have adopted a factory setting. Our resources are called machines. These machines do operations on jobs. Each job thus consists of a number of operations. One can also think of a job as a car needing a number of repairs(the operations) or a student needing to attend a certain number of class meetings.

## 3.1 Definition

Scheduling problems are problems concerned with the allocation of resources over time to perform a set of tasks. Each scheduling problem is characterised by three components:

- A number of machines $M$, a number of jobs $J$, and the processing time for each job on each machine in a matrix $A$.
- A set of constraints $C$ that must be satisfied for each schedule.
- A target function $f$ that must be minimized.

Scheduling problems are interesting from both a practical and a theoretical viewpoint. Since we have selected these problems to benchmark evolutionary algorithms we have selected problems that are relatively easy to describe. They form a very abstract model of a real world situation.

To simplify the description each job contains exactly one operation on each machine. The processing times are given as a matrix $A$ of size $J * M$, where $A_{j,m}$ is the processing time for job $j$ on machine $m$. All processing times are strictly positive integers. The goal of a scheduling problem is to find a schedule. Such a schedule is completely described by a matrix $S$ of size $J * M$ where $S_{j,m}$ is the starting time of the operation from job $j$ on machine $m$. Feasible schedules must not use jobs or machines at two places at the same time, so all schedules must satisfy the following constraints:

$$\forall i, m \; S_{i,m} \geq 0$$

$$\forall i, j, m \ (i \neq j \wedge S_{i,m} \leq S_{j,m}) \Rightarrow S_{i,m} + A_{i,m} \leq S_{j,m}$$

$$\forall i, m, n \ (m \neq n \wedge S_{i,m} \leq S_{i,n}) \Rightarrow S_{i,m} + A_{i,m} \leq S_{i,n}$$

The function $f$ to be minimized is the total length of the schedule, or the time the last operation finishes. In a formula:

$$f(S) = \max_{i,m}(S_{i,m} + A_{i,m})$$

This is often called the `makespan` criterion, and it is a very reasonable measure. Alternatives include flow time, lateness and tardiness (see [5] for definitions) but some of these notions require additional information such as expected delivery times for jobs.

The situation above describes `static scheduling` situations. In the static situation all jobs are available from the beginning. In a `dynamic scheduling` problem each job is released at a time specified by a vector $r$ of dimension $J$. Instead of demanding all operation to start after the zero moment the following constraint must hold.

$$\forall i, m \ S_{i,m} \geq r_i$$

Most methods designed for one variant of scheduling can also be applied to the other. The main difference is that the makespan is not a very suitable notion for dynamic scheduling because it gives too much importance on the latest released job.

Another limitation is that this situation describes `deterministic scheduling`, because everything about the problem is known and nothing unexpected can happen. In practice machines break down and things can be delayed. In `stochastic scheduling` these things also happen. Schedules performing good even when unexpected things may happen are called robust or flexible schedules[22]. This branch of scheduling has a better practical value but is less suited to act as benchmark.

For the constraints $C$ we have three different solutions, because we have studied three different scheduling problems. The choice for these problems has been influenced by their simultaneous appearance in [35].

### Permutation flow shop scheduling

In permutation flow shop all machines in the factory are standing in a line. This means that the jobs can be freely ordered for the first machine, but then they must visit the rest of the machines in the same order in which they entered the first machine. This can be formalized like this:

$$S \sim C \Leftrightarrow \forall i, j, m, n \ (S_{i,m} \leq S_{j,m} \Rightarrow S_{i,n} \leq S_{j,n})$$

### Job shop scheduling

In job shop scheduling each job must enter the machines in a specified order. In the specification of the problem an extra matrix $B$ of size $J * M$ is given. Each row of $B$ is a permutation of the machines, and job $j$ must visit the machines in the order given by row $j$ of the matrix. Formally:

$$S \sim C \Leftrightarrow \forall i, x, y \ (x < y \Rightarrow S_{i,B_{i,x}} < S_{i,B_{i,y}})$$

**Open shop scheduling**

In open shop scheduling no extra constraints exist.

$$C = \emptyset$$

Every job shop or permutation flow shop schedule is also a feasible open shop schedule. Note that jobs and machines can be freely interchanged, yielding an equivalent problem. So we can think of open shop as the problem of making a schedule for a dancing competition given $J$ boys and $M$ girls and the times they want to spend together.

### 3.1.1 Job shop example

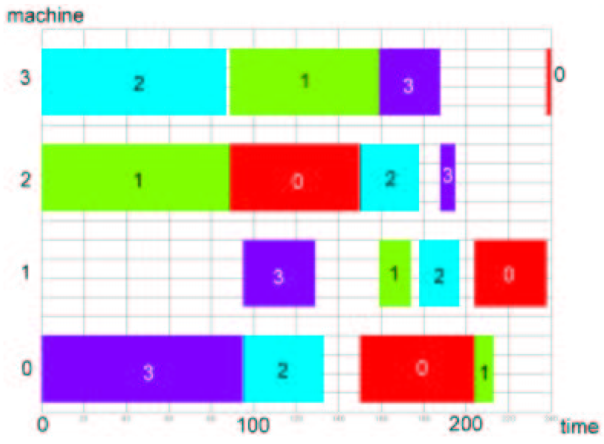
An example Job shop problem is defined by the two matrices $A$ (for the processing times) and $B$ (for the machine order of each job) below. It is a problem with four jobs and four machines.

$$\mathbf{A} = \begin{pmatrix} 54 & 34 & 61 & 2 \\ 9 & 15 & 89 & 70 \\ 38 & 19 & 28 & 87 \\ 95 & 34 & 7 & 29 \end{pmatrix}$$

$$\mathbf{B} = \begin{pmatrix} 2 & 0 & 1 & 3 \\ 2 & 3 & 1 & 0 \\ 3 & 0 & 2 & 1 \\ 0 & 1 & 3 & 2 \end{pmatrix}$$

A solution to this problem is given in the illustration. This illustration is called a `Gantt chart`. Each row corresponds to a machine. The operations on each machine are depicted as a coloured block. On the x-axis the time is given, so each block is as large as the time it takes. The number inside the block is the job it belongs to. The makespan of this schedule is 240 time units.

Solution to the example problem



Gantt charts can be used to check schedulers. By drawing the solutions we have found we can verify that our code respects all constraints posed in the problem.

## 3.2 Lower bounds

For each scheduling problem a lower bound on the makespan can be calculated by looking at the minimal time needed for one machine or one job. This lower bound can be used to prove certain schedules optimal: If the makespan of a certain schedule is equal to this lower bound, no other schedule can do better. In many cases however the optimal schedule does worse than this lower bound and in that case it is very hard to prove that that schedule is optimal. The lowerbound $L(A)$ takes the matrix $A$ of processing times as input and is defined as:

$$L(A) = \max(\min_j \sum_{m=0}^{M-1} A_{j,m}, \min_m \sum_{j=0}^{J-1} A_{j,m})$$

This lower bound is valid for all three kinds of scheduling. For permutation flow shop scheduling a sharper bound can be found, described in [35]. Recall that for each job the order in which it passes the machines is known. For each machine $m$ the time $b_m$ denotes the minimal time it takes a job to get at machine $m$, and $a_m$ denotes the time it takes a job minimally to pass through all machines after $m$. The set of machines in these formulas is $\{1, \dots, M\}$.

$$b_m = \min_j (\sum_{k=1}^{i-1} A_{j,k})$$

$$a_m = \min_j (\sum_{k=i+1}^{M} A_{j,k})$$

The lower bound $\hat{L}$ for the makespan of permutation flow shop schedules is again based on the worst job or the worst machine, but in determining the bottleneck machine it can use $a$ and $b$.

$$\hat{L}(A) = \max(\max_m (b_m + (\sum_{j=1}^{J} A_{j,m}) + a_m),\ \max_j (\sum_{m=1}^{M} A_{j,m}))$$

This bound $\hat{L}$ is at least as good as $L$ so for permutation flow shop this one is preferred. For job shop and flow shop problems the bound $L$ is best.

## 3.3 Schedule properties

The search space of all schedules is very large. Several restrictions on this space have been invented to make the search for a good schedule easier. All schedules matching the constraints as stated in the problem are called `feasible schedules`. Any feasible schedule can be made semi-active by starting all operations as early as possible, without changing the order of operations on one machine or one job. So `semi-active schedules` $S$ have the property that for all $i, m$ the schedule $T$ defined by.

$$T_{i,m} = S_{i,m} - 1$$

$$T_{j,n} = S_{j,n} \text{ if}(j,n) \neq (i,m)$$

is not a feasible schedule. If a schedule is semi-active the starting time of each job must be limited by a certain constraint. Since the extra constraints specific for the three different kinds of scheduling are only concerned with the order of operations, it must be the case that in a semi-active schedule each operation starts exactly when the machine or the job becomes available, or at time 0. This is expressed in these three conditions. For each operation $(j, m)$ in a semi-active schedule at least one of the following propositions is true (this is a second definition equivalent to the first).

$$S_{j,m} = 0$$

$$\exists n\ S_{j,m} = S_{j,n} + A_{j,n}$$
$$\exists k\ S_{j,m} = S_{k,m} + A_{k,m}$$

A certain fitness measure for schedules is called `regular` if it cannot become worse by starting a job earlier. Makespan has this property. For regular fitness measures it is enough to look at semi-active schedules, since for each schedule with a makespan $m$ a semi-active schedule can be derived having an equal or smaller makespan.

A schedule is an `active schedule` if no single operation can be preponed (preponing is the opposite of postponing) without conflicting with other operations. This form of preponing allows the order of operations to be changed. Formally a schedule is active if for all $i, m$ and $0 \leq t < S_{i,m}$ the schedule $T$ defined by:

$$T_{i,m} = t$$
$$T_{j,n} = S_{j,n}\ \text{if}(j,n) \neq (i,m)$$

is not a feasible schedule. Every schedule can be transformed into an active schedule in a computationally easy manner without affecting its length, so an optimal schedule that is active exists. All active schedules are semi-active.

An even stronger notion is that of a `non-delay schedule`. In a non-delay schedule, if a machine m is not doing anything, and a job j is not operated on, and j can visit m next according to the constraints, it must be scheduled. Let $B^M(S,m,t)$ mean that machine $m$ is busy at time $t$, and let $B^J(S,j,t)$ mean that job $j$ is operated on at time $t$ in the schedule $S$. Formally

$$B^M(S,m,t) \Leftrightarrow \exists j\ S_{j,m} \leq t < S_{j,m} + A_{j,m}$$
$$B^J(S,j,t) \Leftrightarrow \exists m\ S_{j,m} \leq t < S_{j,m} + A_{j,m}$$

We also need to define $S \sim_t T$ which means that $S$ is equal to $T$ up to $t$.

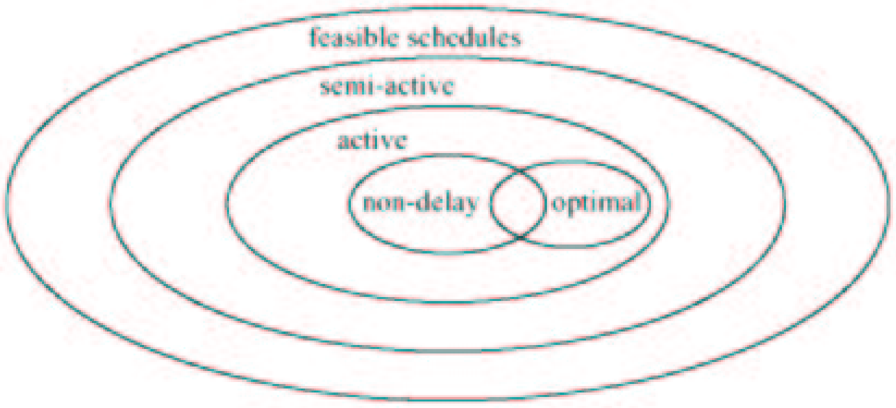
$$S \sim_t T \Leftrightarrow (\forall j,m\ (S_{j,m} < t \vee T_{j,m} < t) \Rightarrow S_{j,m} = T_{j,m})$$

Now we can define that if it is possible to start a job it must start. $S$ is a non-delay schedule if this property is true for $S$:

$$\forall t,j,m \neg \exists T\ T \sim_t S \wedge T_{j,m} = t \wedge \neg B^M(S,m,t) \wedge \neg B^J(S,j,t)$$

The relations between all schedule properties are given in the next figure.

Venn diagram of schedule restrictions



It can be the case that all optimal schedules are not non-delay. The class of non-delay job shop schedules does have a lower average makespan than the class of active schedules. Because for realistic problem sizes one cannot always expect to find an optimal schedule it can be fruitful only to search for non-delay schedules.

### 3.3.1 critical path

One of the interesting aspects of scheduling is that many changes to the schedule do not affect the schedule length at all. Methods for making small improvements to schedules can be more efficient if they can single out the operations that do influence the schedule length. For this purpose the term `critical path` has been invented (see for instance [5] and [22]). The notion appears mostly in relation to job shop scheduling but can be used for other kinds of scheduling as well.

According to an informal definition[22] the critical path should consist of the operations whose delay would increase the makespan. This is a sensible definition if you are interested in isolating the operations you want to monitor extra carefully in a real world situation. In the context of this thesis we want to use critical paths to find shorter schedules and to make this easy we use a different definition that is more concerned with decreasing the makespan. Let $S$ be a semi-active schedule for the scheduling problem $(J, M, A, C, f)$. An operation $(j, m)$ is *critical* in the schedule $S$ if the makespan of the semi-active schedule $S'$ derived from the schedule $S$ by evaluating $S$ on the problem $(J, M, A', C, f)$ has a shorter makespan, where $A'$ is defined as follows:

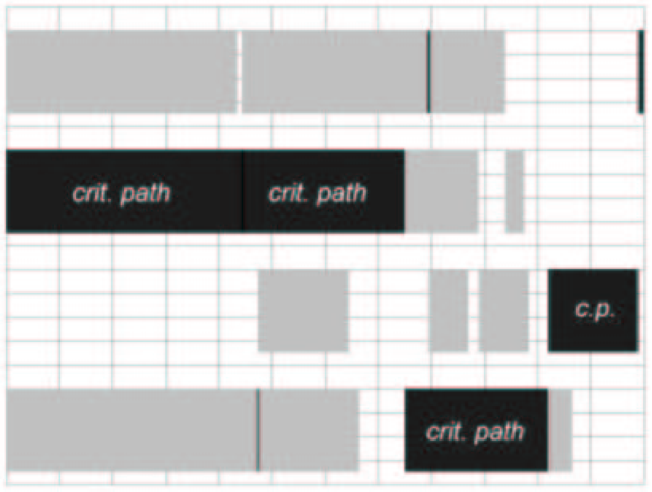
$$A'_{j,m} = A_{j,m} - 1$$

$$A'_{k,n} = A_{k,n} \; (k, n) \neq (j, m)$$

In this definition we have assumed $A_{j,m} > 1$. If not a more difficult definition, multiplying all values by 2 and then subtracting 1, can be used. Note that deriving $S'$ from $S$ is done by shifting operations forward in time. Informally the definition says that operation $(j, m)$ is *critical* if the makespan depends on the length of $(j, m)$.

The figure shows the critical path of the schedule on page 23.

The critical path



Recall that for each operation $(j, m)$ in a semi-active schedule at least one of the following conditions hold:

$$S_{j,m} = 0$$

$$\exists n \; S_{j,m} = S_{j,n} + A_{j,n}$$

$$\exists k \; S_{j,m} = S_{k,m} + A_{k,m}$$

It is possible for the last two conditions to be true at the same time. This would be a rare coincidence if the processing times were random real numbers, but in the integer case assumed here it is a possibility that cannot be neglected. If this coincidence happens in a schedule for a critical operation, our definition and the competing definition from [22] could differ. In the competing definition both predecessors (the operation before on the same machine and the job before the current one in the same job) are critical, because delaying either of them would affect the makespan. In our definition neither of them is critical, because shortening either of them would not decrease the makespan.

If the coincidence described above does not happen and there is a unique last finishing job the critical operations form a path, hence the name critical path. The critical path $P$ can then be found by marking the unique operation with the latest finish time as critical and work backwards in the schedule using the following rules:

$$\forall j, m, n\ ((j, m) \in P \wedge S_{j,m} = S_{j,n} + A_{j,n}) \Rightarrow (j, n) \in P$$

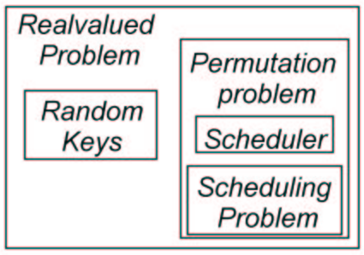
$$\forall j, k, m\ ((j, m) \in P \wedge S_{j,m} = S_{k,m} + A_{k,m}) \Rightarrow (k, m) \in P$$

If the coincidence happens where $S_{j,m}$ is a critical operation or there are multiple last operations , there is not a unique critical path. In this case there are several paths consisting of critical operations according to [22]. In that situation both rules could be applied and we would have a critical tree. The solution is to make an arbitrary choice between the two rules. A path will then be obtained containing all critical operations, and possibly some noncritical operations as well.

A `critical block` is a maximal sequence of consecutive critical operations on the same machine. In the example on page 26 the indicated critical path consists respectively of a critical block of two operations on machine 2 and then three critical blocks of one operation on machines 0,1,3 respectively. The notion of a critical block is important in job shop scheduling because the elements of a critical block can be exchanged, while the consecutive critical operations in the same job cannot, because it would violate the job shop specific constraints. In [13] it is shown that a local change involving only small changes cannot improve the makespan if it does not makes changes at the beginning or end of a critical block (reference copied from [22]). This is used in certain (local) search techniques described on page 47.

## 3.4 Schedulers

Problem transformations



So far several evolutionary algorithms have been described, a method for converting permutation problems into realvalued optimization problems and definitions of scheduling problems have been given. The missing links between all of this are methods for building schedules from permutations. These methods are called schedulers. A scheduler is a function that takes a permutation and returns a schedule. We have used two versions of each scheduler: one scheduler written in java that draws the schedule in a bitmap image, another method written in C that given a permutation returns the makespan of the schedule. Since neither version is very pretty in this text only mathematical definitions for the schedulers are given.

### 3.4.1 Permutation flow shop

Let $(J, M, A, C, f)$ be a permutation flow shop scheduling problem, and let $P$ be a permutation of $J$ elements. The following formulas define a schedule $S$.

$$S_{P(j),m} = \max\{0, S_{P(j-1),m} + A_{P(j-1),m}, S_{P(j),m-1} + A_{P(j),m-1}\}$$

with the following assumptions for the undefined values:

$$\forall m \in \{-1, \dots, M-1\}\ S_{P(-1),m} + A_{P(-1),m} = 0$$

$$\forall j \in \{0, \dots, J-1\}\ S_{P(j),-1} + A_{P(j),-1} = 0$$

This method defines the unique semi-active schedule where $P$ is the order in which jobs pass each machine.

### 3.4.2 Job shop schedulers

For job shop scheduling two scheduling algorithms are described here. The first is what is called the *naive algorithm* and it constructs semi-active schedules. The other one is called the `Giffler-Thompson algorithm`. The latter algorithm produces active schedules.

The naive algorithm for a job shop scheduling problem $(J, M, A, C, f)$ takes a permutation $P$ of the numbers $0, \dots, JM-1$ as input. At the start all entries of the schedule to build $S$ are undefined, denoted by $S_{a,b} = \emptyset$.

```
1. for $i = 0$ to $JM - 1$
    (a) $j = \lfloor P(i)/M \rfloor$
    (b) $m = B_{j,(\min x)}$ such that $S_{j,B_{j,x}} = \emptyset$
    (c) $S_{j,m} = \max_{k,n}\{S_{k,m} + A_{k,m}, S_{j,n} + A_{j,n}, 0\}$
```

The above formulas say that each permutation is used to define a multipermutation (see page 41 for a definition) in which each element of $0, \dots, J-1$ appears $M$ times. The algorithm runs through the multipermutation and for each job number it occurs it schedules the next operation if that job. It schedules it right after the other operations it has already scheduled.

The Giffler-Thompson algorithm, published in [14], also takes a permutation $P$ of the elements $0, \dots, JM-1$. It does not walk through the permutation in order, but searches for the first place in the schedule where a decision must be made. It carefully defines the set of operations suitable for scheduling on that machine, and uses the permutation to choose an element from that set.

$$\text{available}(j, m) \Leftrightarrow S_{j,m} = \emptyset \wedge \exists x m = B_{j,x} \wedge (x = 0 \vee S_{j,B_{j,x-1}} \neq \emptyset)$$

An operation is available if the previous operation on a job is completed and the operation is not yet scheduled.

$$\text{start}(j, m) = \max\{0,\ max_k(S_{k,m} + A_{k,m}),\ max_n(S_{j,n} + A_{j,n})\}$$

The start of an operation is the moment it could start if it were scheduled.

```
1. For $i = 1$ to $JM$
```

```
    (a) $C = \{(j,m) | \text{available}(j,m)\}$
    (b) $(j,m) = min_{(j,m)\in C}\, \text{start}(j,m) + A_{j,m}$
    (c) $D = \{(k,n) | \text{available}(k,n) \wedge n = m \wedge \text{start}(k,n) < \text{start}(j,m) + A_{j,m}\}$
    (d) $(j,m) = \text{argmin}_{(j,m)}(q)$ such that $P(jM+m) = q \wedge ((j,m) \in D)$
    (e) $S_{j,m} = \text{start}(j,m)$
```

This algorithm has been designed only to create active schedules. This stringent condition means that the permutation has less influence on the schedule created. In fact it is possible that several values of the permutation are not used. For other entries only their relative pairwise ordering matters.

The GT algorithm can be modified in order to generate only non-delay schedules[4]. In order to do that one must search for the operation with the first start time instead of the first completion time. This results in the non-delay (ND) scheduling algorithm:

```
1. For $i = 1$ to $JM$
    (a) $C = \{(j,m) | \text{available}(j,m)\}$
    (b) $(j,m) = min_{(j,m)\in C}\, \text{start}(j,m)$
    (c) $D = \{(k,n) | \text{available}(k,n) \wedge n = m \wedge \text{start}(k,n) < \text{start}(j,m)\}$
    (d) $(j,m) = \text{argmin}_{(j,m)}(q)$ such that $P(jM+m) = q \wedge ((j,m) \in D)$
    (e) $S_{j,m} = \text{start}(j,m)$
```

The GT and the ND algorithms can be mixed into an mixed algorithm[4] with a parameter $0 \leq \delta \leq 1$ where $\delta = 0$ behaves identical to the ND algorithm and $\delta = 1$ identical to the GT algorithm. We have not used the ND algorithm or this variant because we wanted to compare our results to other GT based work.

### 3.4.3 Open shop schedulers

In the article [23] two schedulers for Open shop scheduling are described. It is interesting to note that again one of them is very intuitive, while the other restricts itself and makes less use of the permutation. In the article one of them is simply called makespan and the other is called hybrid_makespan.

For the formulation of the schedulers we need one helper function $g$ that calculates what the earliest time is an operation $(j,m)$ can be inserted. Unlike in previous algorithms it searches for open slots inside the schedule.

$$g(j,m) = \min_t(\neg\exists n,k\ t \leq S_{n,k} < t + A_{j,m} \vee\ S_{n,k} \leq s < S_{n,k} + A_{n,k})$$

The first algorithm schedules the jobs in the order they appear in the permutation.

**Algorithm 1**

```
1. for $i = 0$ to $JM - 1$
    (a) $j = \lfloor P(i)/M \rfloor$
    (b) $m = P(i)\, \text{modulo}\, M$
    (c) $S_{j,m} = g(j,m)$
```

The second algorithm uses a heuristic: It schedules the long operations first. The permutation determines which job is processed and the operation from that job is taken such that its time is as long as possible. The procedure needs an extra $J$ by $M$ matrix $B$ with for each row $j$ a permutation of the machines, such that the machines are ordered according to the time for job $j$.

$$\forall j, a \; A_{j,B_{j,a}} > A_{j,B_{j,a+1}}$$

**Algorithm 2**

```
1. for $i = 0$ to $JM - 1$
    (a) $j = P(i)/M$
    (b) $m = \text{argmin}_m(x)$ such that $m = B_{j,x} \wedge S_{j,m} = \emptyset$
    (c) $S_{j,m} = g(j,m)$
```

The permutation is again used as a multipermutation. Note that this algorithm cannot generate all schedules. There is no fancy name for the exact class of schedules generated by this algorithm, since it is hard to find what property they exactly have. They tend to have the large jobs early and a number of machines must start with the largest operation of a job.

## 3.5 NP-completeness

Scheduling problems are interesting for EA research because they are believed to be hard to solve. In computer science a lot of effort has gone in making precise what it actually means that a problem is hard and in this section some of the terminology is introduced, in order to state that indeed job shop scheduling and open shop scheduling are hard problems for which approximation methods are needed. The definitions in this paragraph are taken from [9].

Each instance of a certain problem can be assigned a size. For a scheduling problem the number of entries in the input matrix $A$ is taken as the size of the problem, so the size of a problem is $JM$.

Each algorithm for solving a certain problem needs a number of computation steps to solve an instance. For most algorithms one can estimate an upperbound for the number of steps based on the size of the problem. For instance sorting a list of numbers $L$ of size $n$ can be done in less than $n\log(n)$ steps. The class of easily solvable problems is then often taken as the class of problems solvable in *polynomial time*. This is the case if there is a method such that the number of steps $t(S)$ for an instance $S$ is bounded by a polynomial function. Formally:

$$\forall S \exists a, b \; t(S) < |S|^a + b$$

The theory of NP-completeness works best for decision problems: problem in which a question must be answered by yes or no. Scheduling problems are optimisation problems, in which a certain function must be optimised. We can define for each optimisiation problem a corresponding decision problem in which the input is the input of the optimisation problem and a certain value $x$. The question is: Can a better value than $x$ be reached? For scheduling problems the corresponding decision problem thus is: Is there a schedule with a lower makespan than $x$?

Checking that a certain schedule has a certain makespan can be done in polynomial time (most schedulers run in linear or quadratic time) for all reasonable scheduling problems and certainly for the problems in this thesis.

A decision problem is called *nondeterministic polynomial time* or simply `NP` if a combination of an instance (the scheduling problem and the value x) and a proposed solution (a schedule) for that instance can be checked in polynomial time in the size of the instance. The problem of deciding whether a scheduling problem can be solved with a makespan lower than a given value is NP because calculating the makespan of a schedule is doable in polynomial time.

It is completely unknown whether NP problems can be solved in polynomial time. Most researchers however believe it is not the case. The reason is that certain NP problems are `NP-complete`. This means that any instance of any NP problem can be translated into an equivalent instance of the NP-complete problem within polynomial time. One example of an NP-complete problem is whether a certain formula of propositional logic can be solved. It is considered highly unlikely that NP-complete problems can be solved within polynomial time because it would mean that a wide range of problems would become easily solvable at once simultaneously.

It turns out that many more problems can be proven to be NP-complete. This includes many variants of decision problems based on job shop scheduling. The variant in this thesis is NP-complete and open shop scheduling is considered to be even harder than job shop scheduling [22][34]. The NP-completeness of permutation flow shop scheduling is not known to us.

If the corresponding decision problem is NP-complete then an optimisation problem is called `NP-hard`. So job shop scheduling and open shop scheduling are NP-hard.

# Chapter 4

# Experiments

## 4.1 General framework

In these experiments evolutionary algorithms are tested on scheduling problems. To test them the following framework for an evolutionary algorithm is used (similar to the algorithm on page 15)

```
1. Let $M$ be the uniform pdf on $[0,1]^n$
2. $P = \emptyset$
3. while $|P| < n$
   (a) $P \leftarrow P \cup \{ran(M, [0,1]^n)\}$
4. while $\#\,\text{eval} < \text{Maxeval} \wedge |\{f(p)|p \in P\}| > 1$
   (a) Create $P'$ by selecting the best elements from $P$. $|P'| = \lfloor \tau n \rfloor$
   (b) If needed a model $M$ is created
   (c) $O = P'$
   (d) while $|O| < n$
       i. $O \leftarrow O \cup \{X(M, x \in P', y \in P')\}$
   (e) $P = O$
5. return $\text{argmax}_{x \in P}\{f(x)\}$
```

For the crossover operator $X$ uniform, one-point, $\mathbb{ICE}$ or permutation $\mathbb{ICE}$ is taken. The model $M$ is only estimated in case of $\mathbb{ICE}$ or permutation $\mathbb{ICE}$. $\#\,$eval denotes the number of evaluations done.

We have chosen to work with $\tau = 0.3$ since this value has given good results in previous experiments. This value is suggested as a rule of thumb for the FDA[29]. We do not believe a different value would have changed our conclusions. All experiments are done at least 10 times and the average over these runs is used. We would have like to do more runs for each experiment but the required computer time was already high.

In this algorithm the best $\lfloor \tau n \rfloor$ strings are copied directly to the next generation. This makes this algorithm an `elitist` algorithm: if an optimal solution is discovered halfway in the running of the algorithm, it is still available at the end.

The algorithm terminates if all the strings have the same fitness or if a specified number of evaluations is reached. Note that the number of evaluations is not equal to $n$ times rounds because strings that remain in the population are not re-evaluated. The number of evaluations is $(n + (n - \lfloor \tau n \rfloor)(\text{rounds} - 1))$. Note that the actual number of evaluations can be slightly higher than Maxeval because the algorithm will not terminate halfway in a round.

Three aspects of each run are of interest:

**Fitness:** Our fitness function is the makespan of each schedule, and it must be minimized. We do not have the optimal value for each benchmark problem and even if we had it it could take ages to reach it. Therefore we try to get the best results relative to the other parameters.

**Population size:** The size of the initial population must be chosen beforehand and influences the termination speed of the algorithm.

**number of evaluations:** Given enough evaluations one can always reach the optimal solution. Often the available time is limited, an optimal solution hard to find and you want to know what makespan you can reach for each number of evaluations.

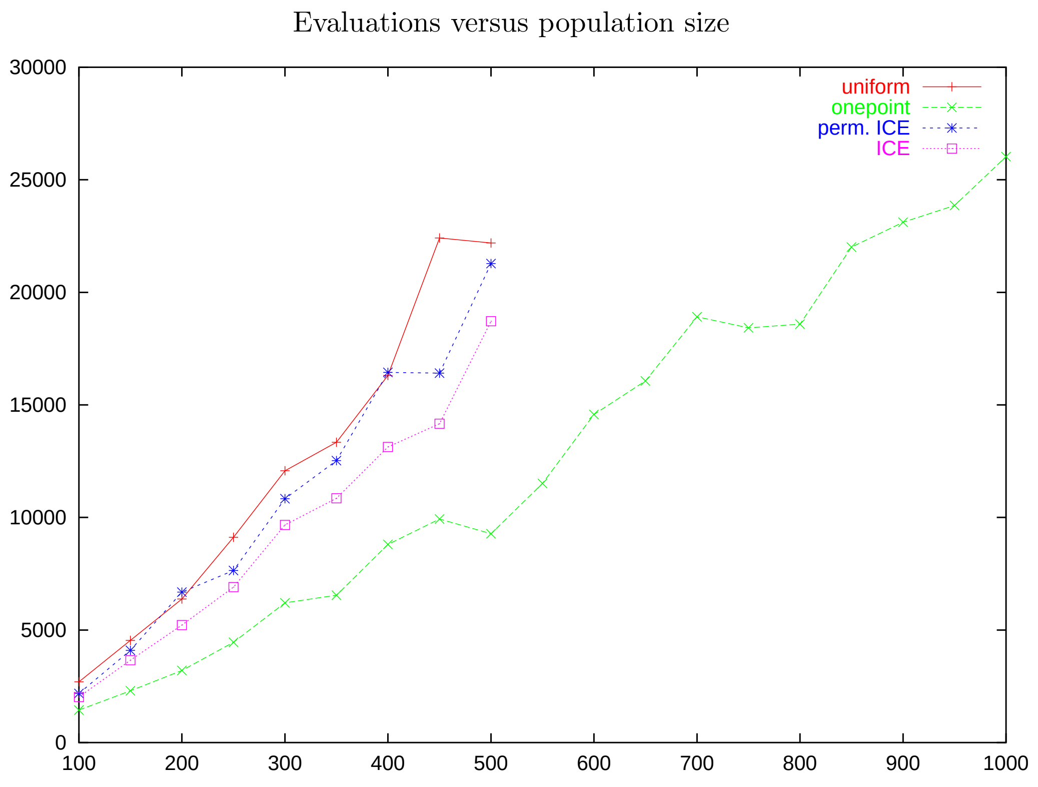
The central question is: Given a number of evaluations what is the best fitness I can reach, and which population size, crossover and other settings do I have to use. We cannot conclude anything about the suitability of the framework above because we have not tried alternatives. Using this framework has been a design decision.

## 4.2 Permutation flow shop

Taillard [35] has given several benchmark problems for three classes of scheduling problems that have been widely studied and the first class is permutation flow shop scheduling. The problems vary in sizes, from 20 jobs and 5 machines to 500 jobs and 20 machines. The number of jobs determines the dimensionality of the search space, while the number of machines influences the chaotic behaviour of the fitness function. With more machines there is more room for complex interaction between jobs.

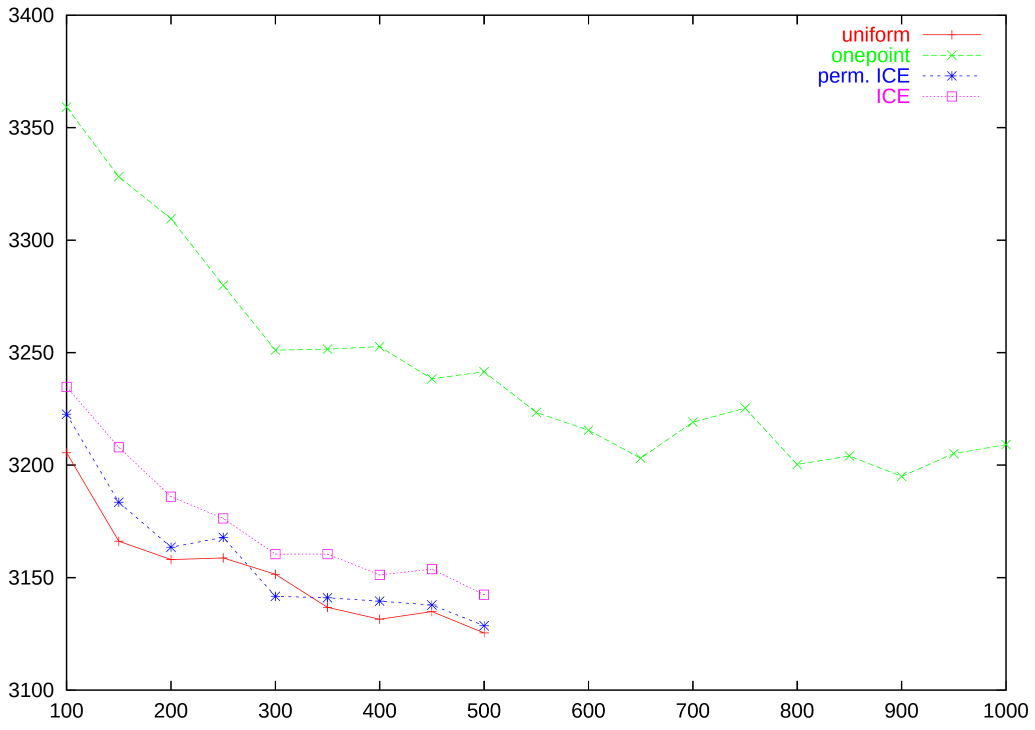
Taillard chooses his problem size to match the real dimensions of industrial problems. Each problem is generated at random (by filling the problem matrix $A$ with numbers uniformly chosen from $\{1, 2, 3, \ldots, 99\}$). After generation a lowerbound for that instance is calculated and the problem is solved by a Tabu search method. If this method easily finds an optimal schedule the instance is discarded. This ensures that the benchmark problems are not trivial to solve.

Given the three aspects one can create three different figures and we have done that. The figures show result on a test problem with 50 jobs, 10 machines and a best known solution of 3037. All three figures are based on exactly the same data, that has been obtained by running our algorithms ten times for each population size. The makespan and evalutation are an average over these ten runs. The maximum number of evaluations is set to infinity, so the algorithms terminate if all strings have the same fitness.



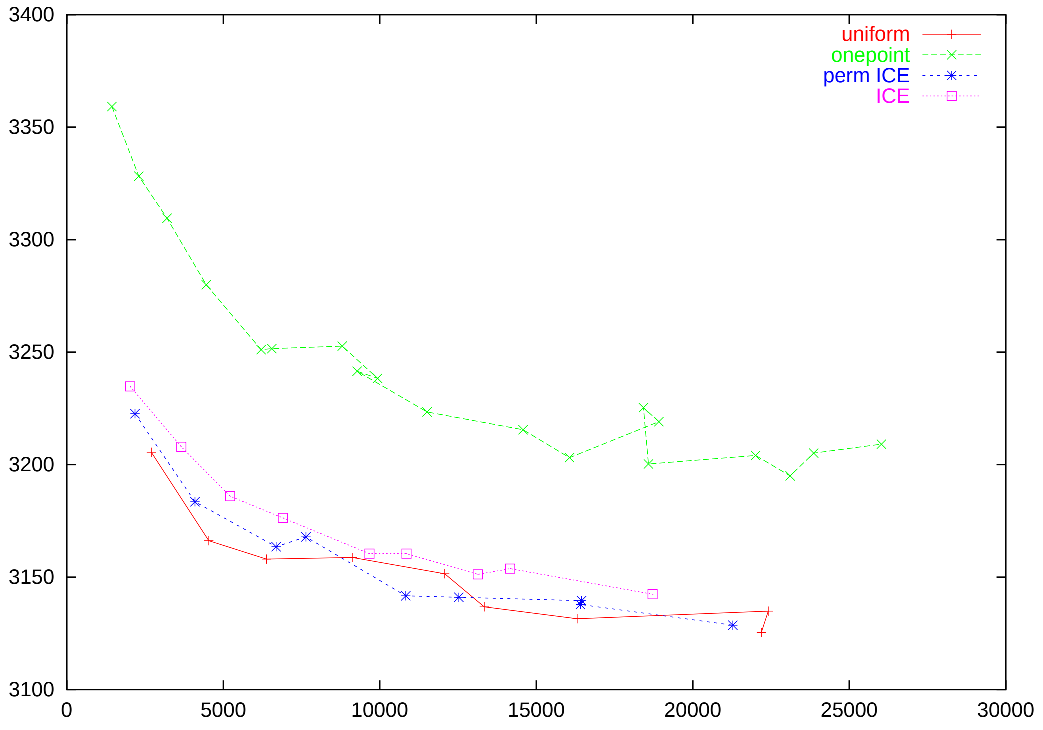
The first figure shows that for the same population size one-point crossover converges faster. An explanation for this is that one-point crossover can generate a much smaller number of children for two given parents. Our algorithm gets a fixed amount of diversity at the beginning of the run, and it decreases faster if only one-point crossover is used. The relative smoothness of the figure indicates a predictable convergence speed, which is always nice to have in an EA.

Makespan versus population size



The second figure shows how good our kinds of crossover score for each population size. It is clear that one-point crossover gives inferior results for the same population size, but this is understandable: it also takes less time. For the real comparison we have to take a look at the third figure.

Makespan versus evaluations

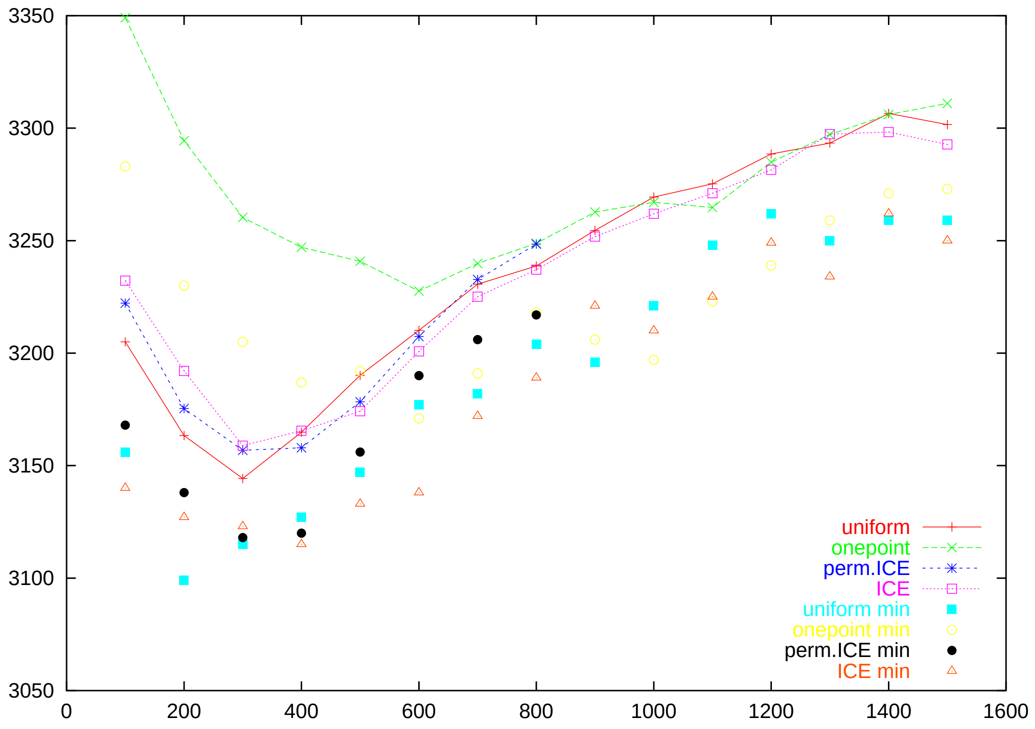


This figure contains two anomalies due to incidental variations in both makespan and number of evaluations. It shows that the makespan decreases for all three methods when using more evaluations, as expected. The two winners are permutation 𝕀ℂ𝔼 and uniform crossover, since their difference in performance is very small. One-point crossover is not very suitable for this problem. 𝕀ℂ𝔼 is a little bit worse than the two winners. Apparently the Gaussian model it uses is not right for this permutation problem.

We have also tested our algorithms also with a clear limit on the number of evaluations. This penalizes the algorithms with a great variance in convergence speed, because it will stop some runs before they have fully converged. On the other hand it favours algorithms that find good solutions early in the process, because the algorithm can be stopped after good solutions are discovered but before it has fully converged to save some evaluations.

If the number of evaluations is fixed different population sizes must be tried and we have done so. Each plot of makespan versus population size will show a U-shaped curve. With a very small population size the algorithm will end before the total number of evaluations is done. With a very large population size the algorithm will be terminated after just a few runs without having had the chance to discover something useful. Somewhere inbetween an optimal population size exists. The next figure gives the makespan at different population sizes for, uniform, one-point, 𝕀ℂ𝔼 or permutation 𝕀ℂ𝔼 crossover on the same benchmark problem.

Average result versus population size



This figure shows results on the same problem with 10000 evaluations. The best known value for this problem is 3037. On the horizontal axis the population size is depicted, on the vertical axis the makespan. The lines represent average results over ten runs, the dots depict best result over ten runs. The figure shows that uniform crossover, 𝕀ℂ𝔼 and permutation 𝕀ℂ𝔼 seems quite evenly matched, with uniform crossover a little bit better. One-point crossover scores worse. It is also interesting to see that one-point crossover needs a larger population size for optimal performance.

The conclusion of the experiments is that the random keys evolutionary algorithm can successfully attack flow shop scheduling problems. The use of our kind of linkage information is not very important in this problem, so uniform crossover works very good for this problem. 𝕀ℂ𝔼 and permutation 𝕀ℂ𝔼 score almost equally well because their behaviour is very similar to uniform crossover if no strong correlations between positions can be found. One-point crossover does not score as well as the other three which indicates that there is no interesting interaction between consecutive positions.

The experiments with unlimited evaluations and with a fixed bound show the same results. The two possible effects mentioned are not very large or cancel each other out. So our method of detecting convergence works well and the evolutionary algorithm does not suffer from differences in convergence speed. In practice either method can be used. The result does show that good performance depends on the right choice for the population size, a fact that is often neglected in the literature. If a fixed population size and number of rounds when assessing for instance crossover operators the results can be misleading. Suppose for instance that we would have chosen a fixed population size of 1000. In that case one-point would have shown results very similar to the rest of the operators. A close inspection would reveal that 𝕀ℂ𝔼 is slightly superior to uniform crossover for this population size. An explanation is that 𝕀ℂ𝔼 needs a slightly larger population size to reach its best performance so 𝕀ℂ𝔼 is tested closer to its optimal setting. So in this case we would have drawn false conclusions.

We can compare our results to the results of Taillard using tabu search. We have not come close to the best known solution generated by this method, which is 3037. Apparently more efficient Tabu search methods exist. Although more work using the same benchmark problems has been done[38] we did not find specific data we could compare to.

Altough we have only done a limited study, we believe that taking more evaluations would not change our conclusions. The reason we believe so is that the search space of flow shop scheduling is very small compared to the search space for job shop or flow shop scheduling. So 10.000 iterations can be considered a very high number for this kind of problem even if it is not for other scheduling problems with the same number of jobs and machines.

## 4.3 Job Shop

### 4.3.1 Previous work

The Job Shop Scheduling Problem (JSSP) is a very well researched area. Mikkel T Jensen [22] mentions several publications and tells that Davis[11] was in 1985 the first to test EA on the JSSP. Nakano and Yamada[30] have used an indirect binary representation and used the idea of forcing (see page 44). Later Nakano and Yamada came with an approach using the Giffler Thompson scheduler. Their results have encouraged other people to do the same, so many GT-based algorithms are now in existence.

Vázquez and Whitley [37] have developed the Order Based Giffler-Thompson (`OBGT`) algorithm and provide useful performance data on several benchmark problems. This algorithm uses a direct representation of the schedule, order based crossovers developed by Davis, and the GT algorithm to repair illegal offspring. Another important element of the algorithm is the use of hillclimbing to find local optimal schedules.

Another algorithm tested is abbreviated THX[25]. The time horizon exchange(`THX`) crossover operator works directly on schedules. A crossover moment in the schedule is chosen at random, like in one-point crossover. The first part of the first schedule (the operations scheduled to happen before the selected moment) is copied exactly. The second part of the schedule is created by the

GT algorithm using the ordering of the second schedule.

The THX algorithm uses a mutation operator based on critical blocks. It swaps two randomly chosen elements of the same critical block. The GT algorithm is then used to build a new active schedule based on the ordering of the mutated schedule. The THX operator is a good example of a complex, problem specific crossover operator for a direct representation.

The use of `parallel evolutionary algorithms` has been tested in combination with THX[25]. In standard evolutionary algorithms, and thus in our set-up, the population is an unstructured set: crossover partners are chosen at random. An alternative is to structure the population so that subpopulations with a high probability of inner mating exist and that is what happens in a parallel EA. The article describes two ways of doing this.

In a `fine-grained` parallel EA the population is placed in a rectangular grid. Each individual gets a location described by two number $(x, y)$ and can mate only with $(x+1, y), (x-1, y), (x, y+1), (x, y-1)$. The effect of this is that premature convergence is prevented. A very good building block travels with limited speed through the population. This gives other building blocks the chance to come into existence.

The opposite of fine-grained parallel EAs are of course the `coarse-grained` EAs. In these EAs the population is divided in a number of subsets, called islands. Only individuals living on the same island are combined using crossover. After a fixed number of rounds has gone by a limited number of individuals is exchanged between the islands. This again prevents premature convergence. Good results on Job Shop scheduling are achieved. Apparently many EAs on the JSSP suffer from too fast convergence on schedules discovered early.

Another method is the Heuristically guided Genetic Algorithm (`HGA`) described in [20]and tested in [37]. It uses an implicit representation that encodes for each step in the schedule building a pair $(m, h)$ where $m$ is a method, either Giffler-Thompson or Non Delay, and $h$ a heuristic to be used to select the operation.

The algorithm GA3 is described in the next paragraph.

Not directly relevant but also interesting are results on dynamic job shop scheduling[24]. The performance of GAs is shown to be better than most heuristics or priority rules.

The algorithms mentioned above are compared in the table on page 50

## GA3

One of the most cited algorithms is GA3 designed by Dirk Mattfeld[26]. A good summary of the algorithm can be found in [22]. In his thesis Mattfeld presents three algorithms: GA1, GA2 and GA3. The last one is the most complex. It includes a structured population and hillclimbing.

GA3 includes a crossover operator GOX and a mutation operator PBM. GOX, the Generalised Order based crossover, needs two parents that are permutations in string form (not in random key notation): the donator and the receiver. A substring $S$ of the donator, length one third to one half of the string, is chosen. The included elements are deleted from the receiver and the string is inserted in the receiver at the position of the first deleted element. This procedure is illustrated in the next table:

| | |
|---|---|
| donator | 58**3274**1096 |
| receiver | 9371540682 |
| after deletion | 915068 |
| child | 9**3274**15068 |

When choosing the substring the donator is considered a circular list. If the substring wraps around, so it contains the end and start of the string, the substring is inserted at the exact position it came from. This is illustrated in the next table.

| | |
|---|---|
| donator | **58**32741**096** |
| receiver | 9371540682 |
| after deletion | 371402 |
| child | **58**371402**96** |

Mattfeld has compared this operator with two other operators: GPX, which respects absolute position in a string to a higher degree, and an operator called PPX or GUX, which respects ordering very well. The results show that GOX is superior for difficult scheduling problems. In theory PPX should have been superior because it is more respectful to ordering information, which is important. The explanation offered is that the use of an operator that introduces more noise prevents premature convergence to local optima.

Three mutation operators have been tested for GA3. Position based mutation (PBM), deletes a randomly chosen element from the string and inserts it elsewhere. On the random keys representation this can be done by assigning a randomly chosen key a random new value. In order based mutation (OBM) two values at random positions are swapped. On random keys this can be done by swapping two randomly chosen keys. Swap based mutation (SBM) swaps the values of two adjacent keys. On the random keys representation this is done by converting to the string presentation, making the swap and converting back. The test indicated that SBM makes too little changes to the schedules, while the changes made by OBM are too drastic. So PBM is the preferred way of mutation in GA3.

GA3 is a fine-grained parallel EA. The population is located in a twodimensional grid, so each string has four neighbours. For crossover the worst neighbour is not selected (unless all neighbours are equally fit) and the other neighbours are chosen proportionally to their fitness. The new string replaces the old string if the new string is not too bad (see [22] for the exact formulas). Each individual either is crossed over or is mutated each generation. Which exactly happens depends on the history of that position: if the last crossover operations have been successful crossover is preferred, otherwise mutation. This history prevents 'inbreeding' (crossing over too similar parents) that is caused by the fine-grained structure.

The hillclimbing procedure uses the neighbourhood $N_{hc,feasible}$ defined on page 48.

### 4.3.2 Scheduler choice

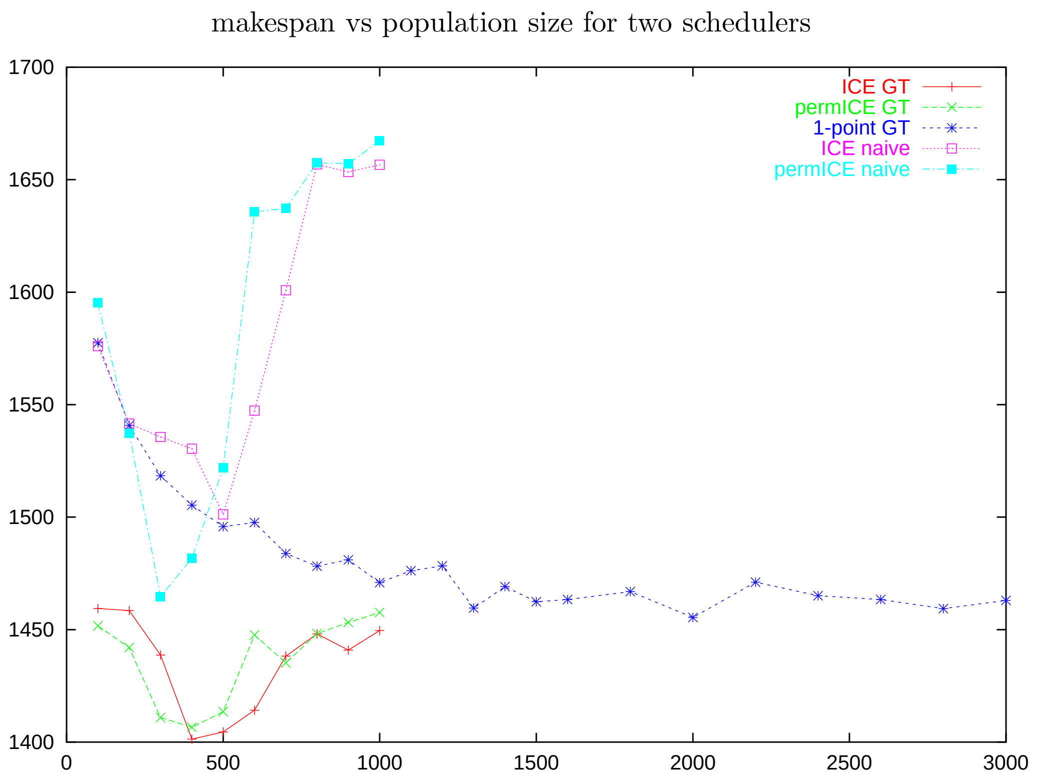
We have described two algorithms for job shop scheduling, and based on the fact that the second produces active schedules, we predict that the second algorithm should produce better schedules. Before turning to evolutionary algorithms we have tested this hypothesis by generating random permutations and building schedules. The results on a benchmark problem were this:

| | |
|---|---|
| optimal makespan | 1231 |
| average value naive algorithm | 2400 |
| average value GT algorithm | 1925 |

This benchmark problem has been used throughout this section. It has 15 jobs and 15 machines.

We see that the GT algorithm indeed searches in a much better subspace. This hints that GT scheduling is better. Complications can arise if the space of active schedules is less 'regular' or 'smooth' so that our algorithms cannot discover any building blocks or pattern. It could also be the case that the algorithms are capable of optimising schedules such that they become active even when using the naive algorithm.

The first results obtained by running the algorithm show that GT is indeed much better. This figure shows results reached at 50.000 evaluations.



It is remarkable that one-point crossover seems to act rather strangely. It is effective even at very large population sizes and it is not inferior to the other two algorithms.

### 4.3.3 Linkage and multipermutations

A `multipermutation` of multiplicity $k$ and size $kn$ is a function $f : \{0, 1, \ldots, kn-1\} \to \{0, 1, \ldots, n-1\}$ in which each value of the range appears $k$ times. So the list 012102 is a multipermutation of multiplicity 2 and size 6. The naive algorithm reduces each permutation to a multipermutation using the line

1. $j = \lfloor P(i)/M \rfloor$

This is one way to do the reduction. One can do it also differently by using the modulus operation. Both methods are common practice in computer programming.

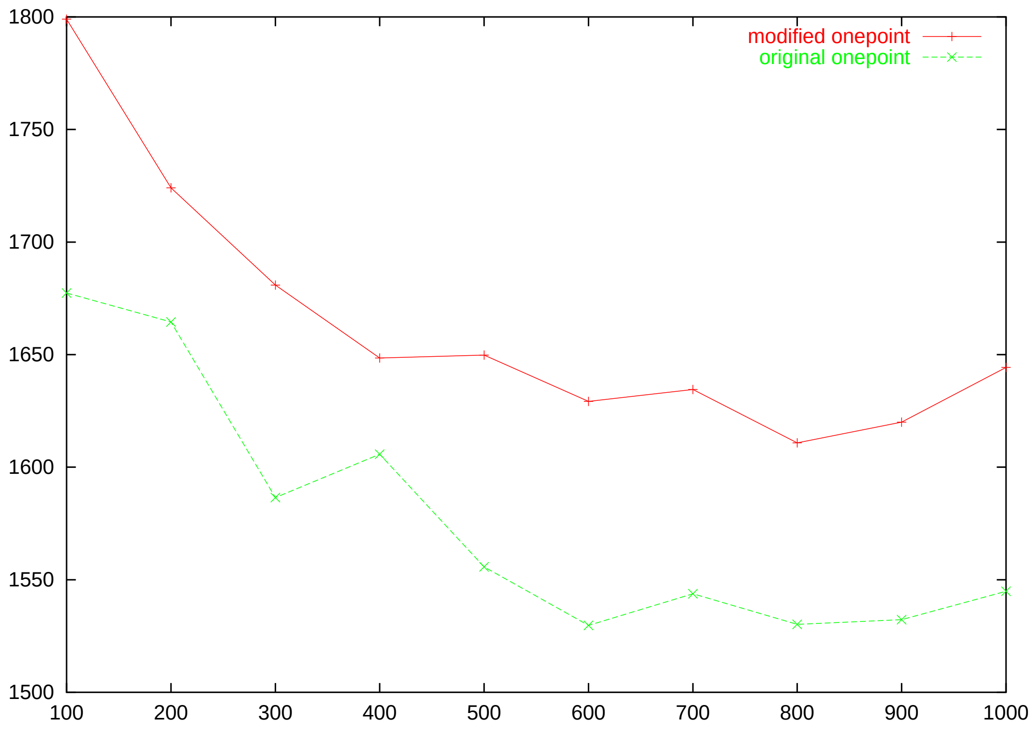
1. $j = P(i) \bmod J$

The difference is that in the original line the random keys that denote the same element in the multipermutation are close together. In the modified line of code the keys are interleaved. For a 3 by 3 example the job each key belongs to is indicated here:

1. original: 0 0 0 1 1 1 2 2 2
2. modified: 0 1 2 0 1 2 0 1 2

The two methods of reduction are equivalent under uniform and (permutation) $\mathbb{ICE}$ crossover. In these operators crossover chances do not depend on the place in the string. In one-point crossover the place is important and we see that the results are different. The best results are obtained by using the original line. This suggests that it is beneficial to keep elements belonging to the same job together.

Job shop with different crossover operators



An explanation for this effect can be given in the terms of reduced `false competition` caused by the redundant encoding. False competition happens if the same building block is present in different encodings. The different encodings will compete against each other instead of competing with other building blocks. If they meet during crossover (both parents have the same building block in different encoding) the building block can be destroyed. This effect can happen in the algorithm 1 representation of multipermutations. The table illustrates this. The building block is in this case the group of 4 keys for job 1.

| jobs of each key | 1 | 1 | 1 | 1 | 2 | 2 | ... |
|---|---|---|---|---|---|---|---|
| parent 1 | *.1* | *.2* | .8 | .7 | .3 | .4 | ... |
| parent 2 | .7 | .8 | *.2* | *.1* | *.5* | *.3* | ... |
| child | *.1* | *.2* | *.2* | *.1* | *.5* | *.3* | ... |

The child will represent a completely different schedule because it will schedule job 1 operations sooner than its parents.

The use of one-point crossover can enforce that keys belonging to the same job are inherited together, thus minimizing the disrupting effect described above. This works best if those keys are close together. This is the case in our first formulation of algorithm 1, and not in the alternative.

### 4.3.4 Jobwise crossover

The notion of an active schedule gives one explanation for the difference in performance. Another explanation could be offered by the disruption explained in the previous subsection. We have seen in the experiment with two different variants of algorithm 1 that it is beneficial to keep the keys belonging to the same job together in crossover. Based on this insight we have developed a special crossover operator. It performs uniform crossover on groups of random keys belonging to the same job.

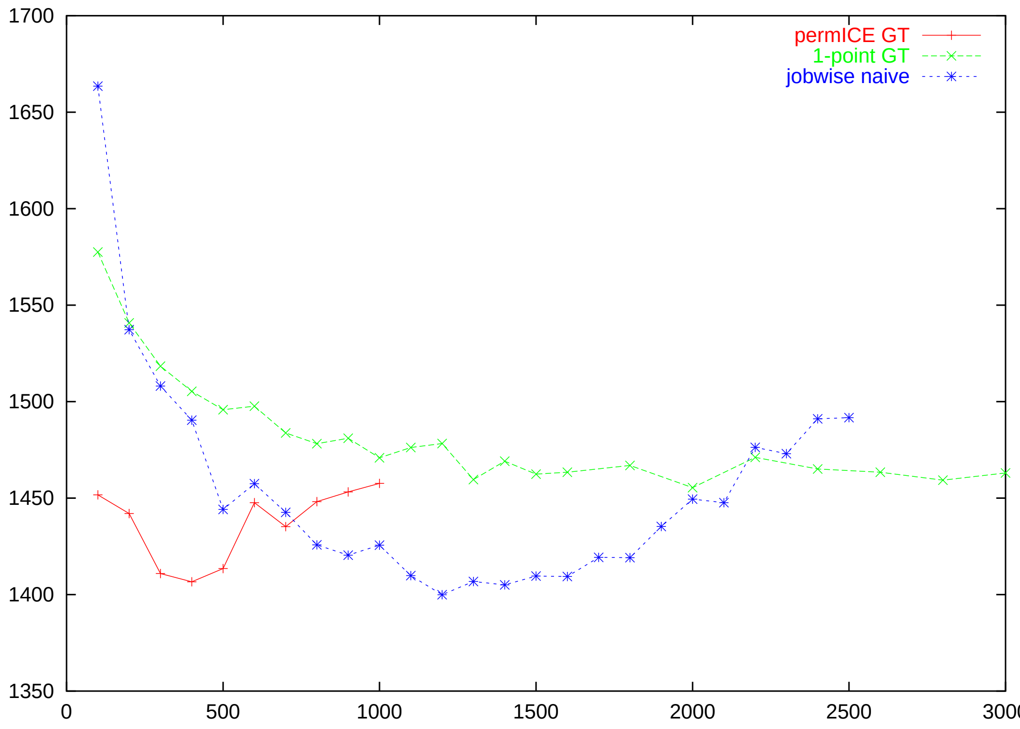
```
1. for $i = 0$ to $J$
   (a) $a = \text{random}(2)$
   (b) for $x = jM$ to $(j+1)M$
       i. if $a = 0$ $t(i) = p(i)$ else $t(i) = q(i)$
```

This jobwise crossover operator is a less general kind of M-point crossover. Not only the number of cuts but also their position is fixed. It never disrupts a block of keys belonging to the same job. According to schema theory it should be less effective because the number of schemata that can be combined is much smaller: It works on strings of length $J$ instead of strings of length $MJ$.

This crossover operator can also be seen as a variant of $\mathbb{ICE}$ with a fixed factorization. One might even say that this is a crossover operator incorporating knowledge of the problem domain. It is not a very difficult operator, but it does need to know the number of jobs, which is information the other operators do not have.

The results achieved by this operator are quite good compared to the other operators.

Jobwise crossover



This operator, using the naive algorithm, outperforms the Giffler-Thompson scheduler. We have shown it in combination with permutation $\mathbb{ICE}$, the best result from the previous figure. Because this operator introduces less entropy(it draws less numbers from the random number generator) it needs a larger population size than $\mathbb{ICE}$ or uniform crossover, but still less than one-point.

Our conclusion is that the naive algorithm does not perform well because our crossover operators (with an exception for jobwise crossover) are not capable of handling multipermutations. The GT-algorithm does not exploit the property of active schedules to its maximal value. An explanation could be that the work done by the scheduler is not fed back to the random keys notation, so the EA cannot learn anything from it.

### 4.3.5 Forcing

When solving a problem using an Evolutionary Algorithm it is possible that a nice and pleasant mapping between strings and candidate solutions can be found, but that this mapping maps some strings to infeasible solutions. In the literature two solutions are proposed:

- Define a new fitness function that penalizes incorrect solutions. The penalty must be small enough so the penalty does not completely overshout the original fitness function, but it must assure that the algorithm does not output a penalized point.

- Change incorrect strings during evaluation to the closest correct solution. In the context of scheduling this means the scheduler not only creates a schedule, but it also constructs a new string that is a repaired version of the string. This is called `forcing`: The string is forced into shape to match all constraints on the search space.

Forcing is often used when people want to apply classical GAs, working on binary strings because a binary representation maximizes the number of schemata, on problems where binary presentations do not work very well. In [30] a matrix is used to represent an ordering on operations, and forcing is used to ensure that the ordering relation is transitive.

The problem of the GT algorithm is that many strings generate the same schedule. The scheduler is so smart that it does not have to use the information in the string for many decisions. We have seen that this results into good schedules, but it makes it also hard for EA to find the optimal schedule: each string contains lots of 'false information'.

We have tried to solve these problems of the GT scheduler by applying forcing. After a schedule has been created (it can be done afterwards because GT does not need forcing) a new string is generated based on the schedule and this is used to replace the input string. So forcing is not applied to replace infeasible permutations, but it is used to give feedback of the decisions made by the scheduler to the EA. It is a convenient way of incorporating problem specific knowledge used in the scheduler into the EA.

To use forcing for each operation a random key must be determined. Since the scheduler puts operations with a smaller key first if they are fighting for a resource, it is reasonable to preserve the starting order in the key order. If not, the outcome of the forcing might not correspond to the schedule that it should represent. We have tried two methods of forcing:

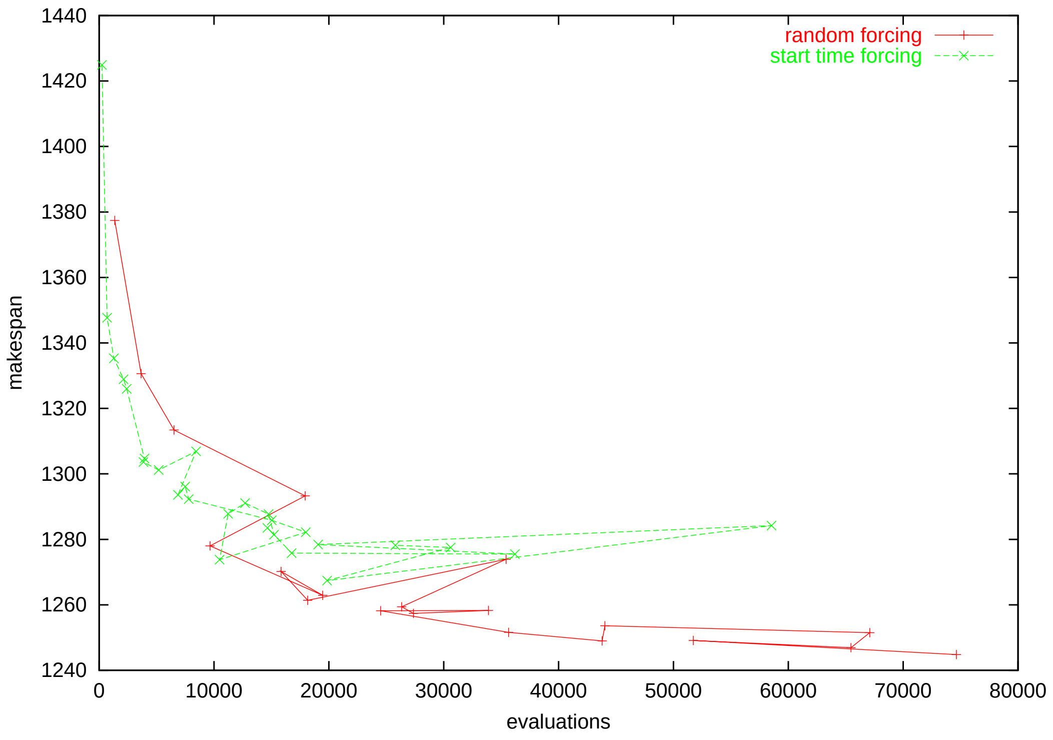
- Random forcing is done by sorting a fresh, randomly created set of keys in the order you want.

- Start time forcing sets the key equal to the start time. The formula for a key $k$ is $k_{iM+m} = C * S_{i,m}$, where $C$ is an appropriate constant so that the keys fit in the interval $[0, 1]$. In our implementation we have taken the lower bound of each problem as $C$. This means the keys almost, but not entirely, fit in $[0, 1]$ but that does not matter for the crossover operator: It can handle that without any change of definitions.

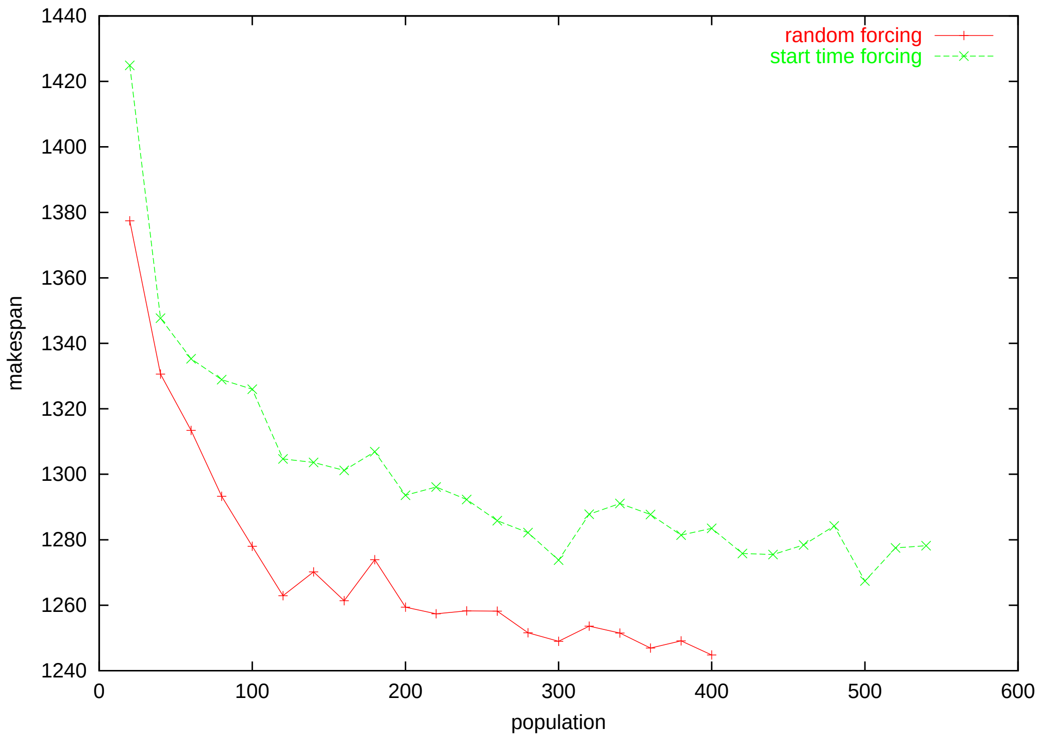
A third method called ordered forcing, which assigns a key based directly on the ordering of the start times, has also been tried but it proved to be very similar to start time forcing so we have restricted ourselves to these two alternatives. Each method has some sort of motivation. Start time forcing is the most informative. The entire schedule is represented and can be constructed from the string. Random forcing introduces additional randomness, which could prevent premature convergence, or one can say it fits better within the random keys paradigm.

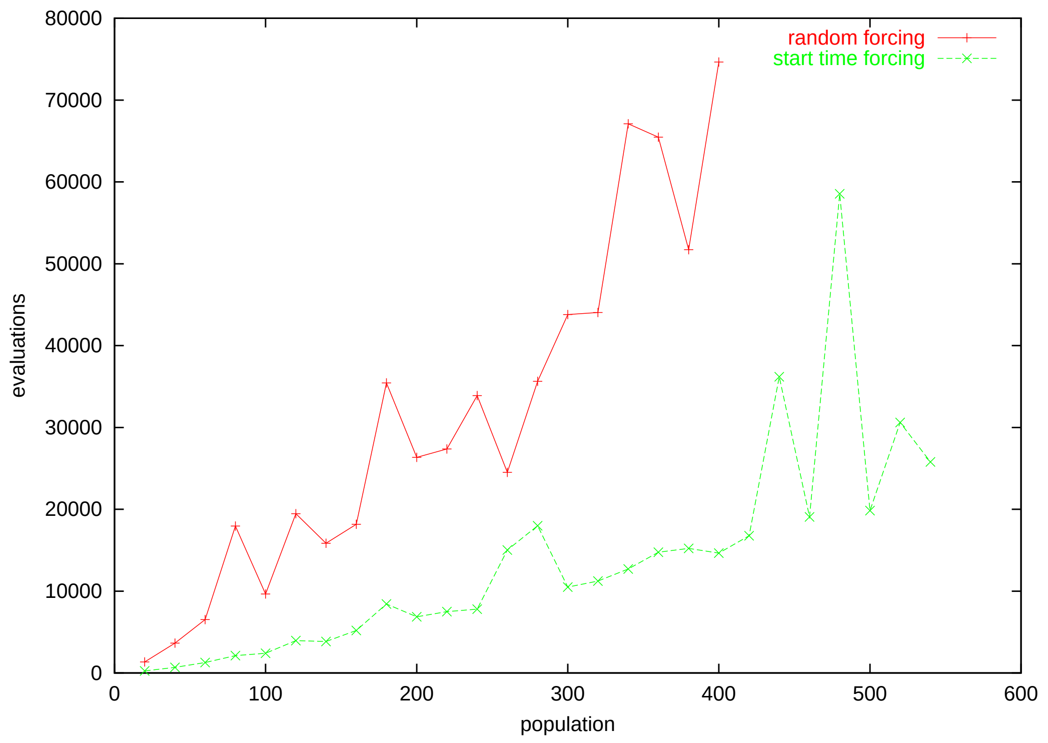
We wanted to display the result of the comparison between the two kinds of forcing using a plot of the average makespan versus the number of evaluations.

makespan versus population size using forcing



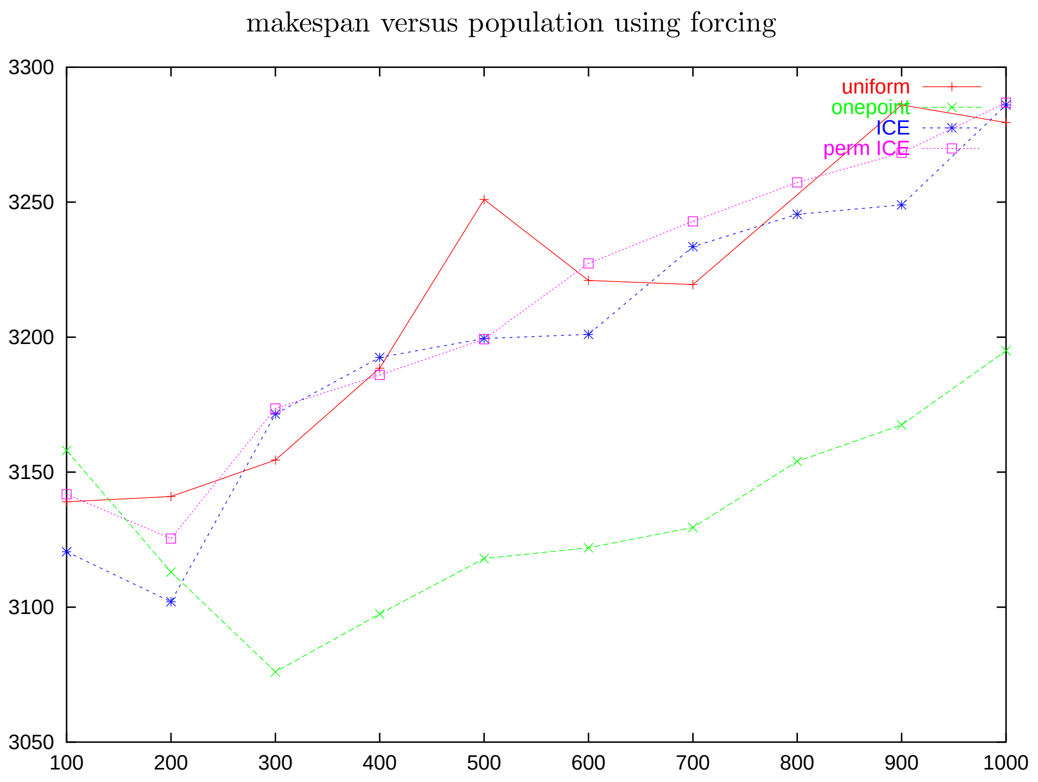
This plot is a little chaotic. In order to explain what is happening we give the easier plots of makespan and number of evaluations versus the population size.





The data has been generated by doing each run ten times using uniform crossover, the GT algorithm and the indicated kind of forcing. What we see is that for the same population size the number of evaluations is much higher using random forcing than in the case of start forcing. This is explained by the randomness introduced by the fresh values that were used in random forcing. The higher number of evaluations results in much better schedules. In the overlapping range of the first figure it seems that start forcing is better.

In order to study the effects of forcing combined with different kinds of crossover we have used the GT algorithm and start forcing tested using different crossover operators. The maximal number of evaluations is 10.000.



The image shows that one-point crossover outperforms the other kinds of crossover. This is important because the use of forcing apparently changes the search space such that preserving linkage between the keys becomes important. 𝕀Cℂ and permutation 𝕀Cℂ do better than uniform crossover because they can detect useful factorizations. Unfortunately they do not outperform one-point crossover.

### 4.3.6 Local search

It is possible to combine the EA approach with a local search strategy. In local search new schedules are derived from one parent. The new schedule is expected to be very similar to the old one and can be said to be very close, hence the name local search.

This type of algorithm is called a `hybrid` algorithm, because it combines the blind search methods of evolutionary algorithms with the specific knowledge often incorporated in local search methods.

What is needed to do local search is the concept of a neighbourhood. For each schedule $S$ a neighbourhood $N(S)$ is the set of close schedules that can be reached from that schedule. Using such neighbourhood either hillclimbing or steepest descent can be used as search strategy. A clear distinction between these strategies is explained below [22] but either term is sometimes used to describe the more general term local search.

In `hill climbing` a schedule is replaced by the first schedule from its neighbourhood that has a better fitness. To do this an arbitrary ordering $N(S)_0, N(S)_1, \ldots$ on the neighbourhood $N(S)$ is needed.

$$S_{i+1} = N(S_i)_{\min x} \, \text{such that} \, f(N(S_i)_x) < f(S_i)$$

$f$ is the function to be minimized. The above formula defines a list of schedules $S_i$. At a certain point the formula can no longer be applied. The point found is a local optimum. This is defined by:

$$\text{localoptimum}(S) \Leftrightarrow f(S) = \min_{T \in N(S)} (f(T))$$

In `steepest descent` the best element of each neighbourhood is selected, not just the first one.

$$S_{i+1} = \text{argmin}_{T \in N(S) \cup S}\, f(T)$$

Steepest descent makes better progress each step but it also evaluates more schedules, so it is more expensive. It also finds a local optimum, but it can be different optimum or their relative speed could differ.

The costs and effectiveness of local search depends on the neighbourhoods used. Ideally it must incorporate as much information of the problem as possible. If local search is the only means of searching such as in tabu search and simulated annealing connectivity is an important property: The neighbourhoods must be large enough so that a good local optimum is reachable from each starting point. If local search is used in addition to an EA the neighbourhood should be small enough to keep the search time reasonably low yet contain enough promising moves.

Note that each mutation operator in fact defines a neighbourhood. Let $M(S,r)$ be a mutation operator taking a schedule (or more general a point in the search space) and a random number $r$ as input.

$$N_M(S) = \{T | \exists r\ T = M(S,r)\}$$

The two ways of defining, as an operator or as a set of neighbours, are equivalent (as long as each member of the neighbourhood has the same chance of being generated). An operational definition can be preferred if the neighbourhood is so large that you never would want to generate it but only to take samples from it.

For job shop scheduling several neighbourhoods have been defined. They all use the notion of critical paths since this notion can be used to predict which changes will actually change the makespan. It can be proven [22] that only swaps involving either the end or the beginning of a critical block can alter the makespan. Based on this property several neighbourhoods can be defined:

The neighbourhood $N_{hc}$ contains swaps at the beginning and end of critical blocks. Which changes are allowed is summarized in the table below, where $(o_1, o_2)$ is a pattern for a critical block of length two, and $(o_1, o_2, o_3, \ldots)$ and $(\ldots, o_1, o_2, o_3)$ patterns for critical blocks of at least length three. $N_{hc}$ consists of all semi-active schedules derivable from the current schedule by making one change matching one of the parents. Certain moves described by the patterns can in fact violate the constraints of the problem so a test is needed before doing a move.

| block | $(o_1, o_2)$ | $(o_1, o_2, o_3, \ldots)$ | $(\ldots, o_1, o_2, o_3)$ |
|---|---|---|---|
| moves | $(o_2, o_1)$ | $(o_2, o_1, o_3, \ldots)$ | $(\ldots, o_1, o_3, o_2)$ |
| | | $(o_2, o_3, o_1 \ldots)$ | $(\ldots, o_3, o_1, o_2)$ |
| | | $(o_3, o_2, o_1, \ldots)$ | $(\ldots, o_3, o_2, o_1)$ |

The neighbourhood $N_{hc,feasible}$ is a subset of $N_{hc}$. It is also defined by the set of schedules obtained by certain moves. The move $(o_1, o_2)$ is allowed, one of the moves at the beginning, namely the one which is estimated to improve the makespan the most, and one move at the end of the block, namely the one the is estimated to give the best improvement. The exact estimates are given in [22] page 53. In order to efficiently calculate this neighbourhood a graph representation of the schedule is needed.

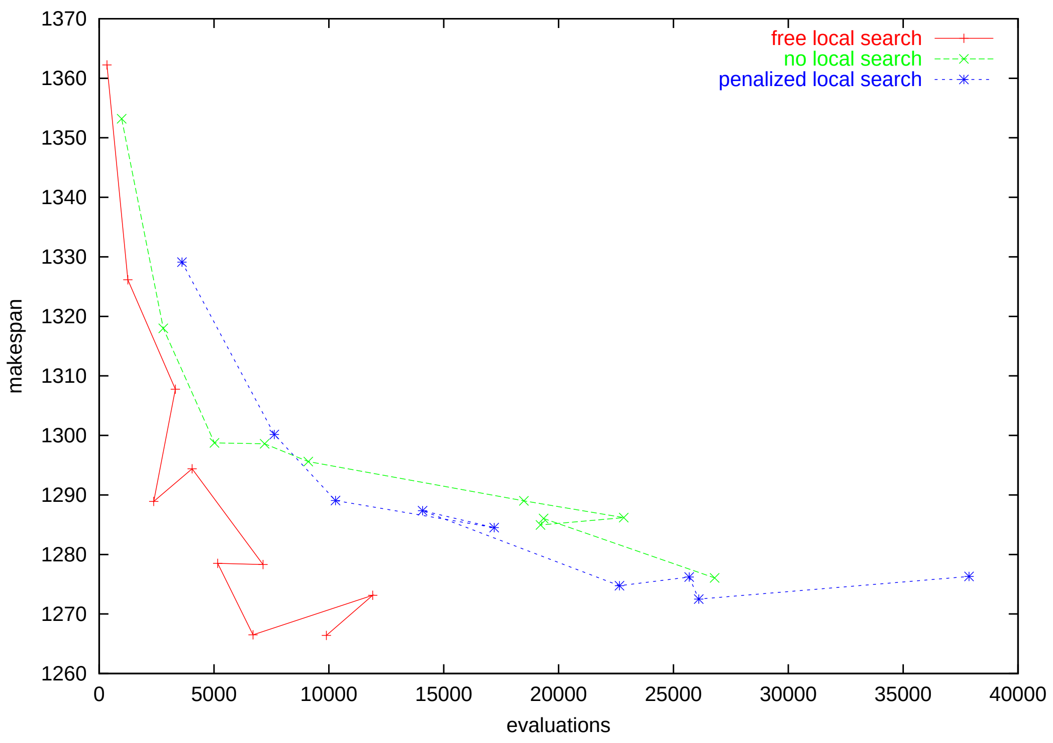
Another, smaller neighbourhood is given in [31]. It has been introduced for use in tabu search. A table of allowed swaps in critical blocks can again describe it.

| block | $(o_1, o_2)$ | $(o_1, o_2, o_3, \ldots)$ | $(\ldots, o_1, o_2, o_3)$ |
|---|---|---|---|
| moves | $(o_2, o_1)$ | $(o_2, o_1, o_3, \ldots)$ | $(\ldots, o_1, o_3, o_2)$ |

A fast way of evaluating the neighbours of a schedule is given in the article, again based on a graph presentation.

We have incorporated a steepest descent search technique based on the last neighbourhood in our algorithms. The results are shown in the next figure. It is based on the GT scheduler combined with start time forcing.

Local search effectiveness



The figure is a plot of makespan versus evaluations, so lower-left lines are better. The line named 'no local search' shows the results without forcing. The 'free local search' line shows what happens if forcing is applied: a better makespan is achieved. This is not very surprising, since a lot more work happens in the same evaluation. The algorithm takes a lot more time. In our implementation the mutations are done on the random keys level and a new schedule is built, so each local steps is as expensive as an evaluation. If we count these evaluations we arrive at the line labelled penalized local search. We see that for a small number of evaluations it is not useful to employ local search, but it has a positive effect if a large number of evaluations can be used.

Local search can be implemented more efficiently, making re-evaluations faster than evaluations of fresh strings. In order to do this one needs to rewrite the scheduler so that more information on the schedule is present. This could make evaluation a little bit slower. Once we do these changes we can use real computing time in a performance measure instead of the number of evaluations. This is possible, but not what we want, because we view the JSSP as a nice example of a hard problem. In practice problems evaluation cost could be much higher. This thesis is not dedicated to the JSSP, but uses it to gain knowledge on EA. Another problem is that we would need computing time data on the other algorithms. These times would be very implementation dependent.

What in fact happens is that when does not uses completely blind methods but hybrid methods, the number of evaluations is no longer a good measure. So we just note that even without a fast implementation local search can be useful, but need not be, and that it is probably useful with a fast implementation. Note also that 𝕀ℂ𝔼 and permutation 𝕀ℂ𝔼 take more time than uniform and one-point crossover so that the comparison between crossover operators would be influenced by the use of a different measure.

The real question is whether results obtained by a combination of EA and local search techniques can achieve better results for fewer costs (evaluation costs or computing costs) than each technique separately. The articles describing hybrid EAs have not answered this question, they only answer whether hybrid EAs are better than pure EAs. The answer is yes, but that is not surprising. Pure EAs have the advantage that they are easier to implement than local search strategies. Hybrid EAs are as hard to implement as tabu search or simulated annealing. So there is a trade-off between the effort an implementer wishes to make and the result he will achieve.

### 4.3.7 Comparison to previous work

The table below shows performance on one of Taillards test problems of different methods. The column named upperbound refers to the best known solution so far. NS refers to a Tabu Search method that has been developed by Nowicki and Smutnicki with a maximum of 100,000 iterations[31]. The other columns list the average result of a population of 100 individuals and a maximum of 100 generations, except our additions uniform and one-point, in which we have used various population sizes with a maximal number of 10.000 evaluations. In brackets the population size is given. We have varied the population in steps of 100.

| upperbound | NS | THX | HGA | GA3 | OBGT | uniform | one-point |
|---|---|---|---|---|---|---|---|
| 2760 | 2775.3 | 3119.6 | 2908.8 | 2927.5 | 3117.5 | 3157.0(200) | 3075.0(300) |
| 2756 | 2767.3 | 3127.0 | 2889.0 | 2932.9 | 2992.5 | 3097.0(100) | 3078.5(300) |
| 2717 | 2743.6 | 2933.6 | 2822.3 | 2780.1 | 2924.4 | 2998.2(200) | 2976.7(300) |

The table shows that Tabu Search is a very good method for solving these problems. Assuming that an iteration in Tabu search is cheaper than in the EAs above, because the schedule can be adapted instead of rebuilt, we can say that NS is still state of the art in scheduling.

The HGA and the GA3 are both hybrid and parallel EA and they do very well on these problems. The OBGT comes in third, although it is beaten on the first problem by one-point crossover. The reasons for the differences between the problems, which are certainly significant because they are much greater than the variance in results for each method, are not clear. One-point seems a little bit better than THX, which is better than uniform.

Much more can be said if one is interested in choosing an algorithm for use in practice. In the benchmark set seven more instances of the same size exist and other test sets exist and give different results[37]. We have not further sorted out this matter because we cannot make fair comparison between hybrid and blind methods and because the explanation will probably involve all new aspects we cannot investigate, like the way the problem has been generated (with uniform numbers or correlations between jobs and machines), the selection method used, parallelism. Best advice is probably to generate a test set based on instances as close as possible to the practical situation and evaluate all methods not completely dominated by another to find the one that works best on that problem.

## 4.4 Open shop

It seems that studying open shop is nowadays more popular than job shop because it is less well researched than the job shop problem.

### 4.4.1 Previous work

In a recent article [23] results obtained by evolutionary algorithms on the open shop testproblems from Taillard [35] can be found. In this article three different combinations of algorithms and schedulers are compared.

The permutation algorithm uses the algorithm A1 scheduler. It is a very ordinary evolutionary algorithm using roulette wheel selection, uniform crossover and swap mutation.

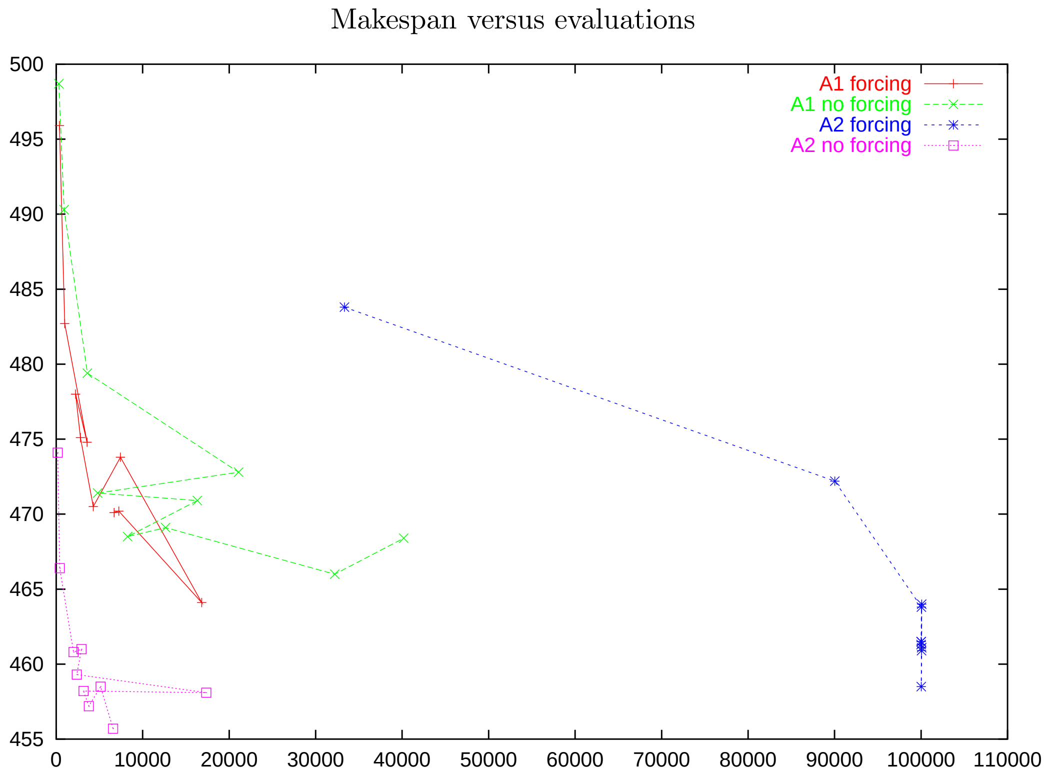
Their second approach is classified as a hybrid genetic algorithm. It is called `hybrid` because it uses the heuristic to begin scheduling the largest jobs. This deviates from the use of hybrid in Job shop, where it denotes the use of local search. From this algorithm we have isolated the A2 scheduler for open shop. The underlying genetic algorithm is again simple, using roulette wheel selection, uniform crossover and swap mutation.

The next candidate uses the Selfish Gene Algorithm (FGA) described on page 16. In order to test this algorithm in its purest form a very primitive representation is given. It does not use permutations directly. The method of scheduling is most similar to algorithm A1. The input of the scheduler is a string of twice the length of the number of operations. Each even (we start at 0) position encodes a job and the next position is interpreted as an operation from that job. A value $v$ refers to the $v$th unfinished job or the $v$th unfinished operation. It is not explicitly stated but assumingly the search for an unscheduled operation is wrapped around if $v$ is larger than the number of unscheduled operations.

The above method is an alternative way to encode permutations for the use in evolutionary algorithms. The drawback of the method above is that the interpretation of the latter numbers depends on the values that precede it. If a change is made in the beginning of the string the interpretation of almost all numbers can change.

### 4.4.2 Forcing

The results on job shop scheduling have tempted us to apply forcing also on this problem. The next figure plots the result with and without forcing for both scheduling algorithms using uniform crossover.



We admit this is a horrible figure with all the lines crossing. This figure has been made with population sizes of 20 to 200, with the maximal number of evaluations of 100000. The experiment has been done with uniform crossover. The kind of forcing applied is start time forcing. Algorithm 2 with forcing reaches the maximal number of evaluations very quickly, and performs very poorly. The reason is that start time forcing in combination with this scheduler is not correct: The forced random keys will not build the same schedule as the original keys. Because algorithm 2 searches for open places in the schedule, a small operation $(j, m)$ for a job can be placed earlier in the schedule than a larger operation from the same job $(j, n)$. Because of start forcing the key of $(j, m)$ will receive a smaller value than the key of $(j, n)$. If this string is interpreted the next time the key of $(j, m)$ will be used to schedule $(j, n)$. This is possible because the algorithm treats the permutation as a multipermutation. The larger job $(j, n)$ will no be placed earlier in the schedule. The conclusion is that you cannot naively add start time forcing to algorithm 2.

The figure shows average results taken over ten independent runs. Nevertheless the image looks very scrambled. This is caused by the fact that we show the dependence of two aspects each of which is subject to statistical variations. This means the dots of the line have variation in all directions. On the basis of this noisy data we can conclude that algorithm 1 performs better with forcing than without forcing, but the result is not as striking as in the job shop example.

### 4.4.3 Results with different crossover

To compare the performance of different crossover operators we have tested them with a fixed number of evaluations and different population sizes. The results are given below. The number of evaluations has been chosen to scale with the number of jobs so that the number of evaluations is not too high for the smaller problems.

| size | evals. | current best | A1 one-point | A1 uniform | A1 𝕀ℂ𝔼 | A1 p𝕀ℂ𝔼 | A1 jobwise |
|---|---|---|---|---|---|---|---|
| 4x4 | 400 | 193 | 196.6 | 195.7 | 197.3 | 196.2 | 195.9 |
| 5x5 | 500 | 300 | 322.3 | 319.9 | 323.1 | 320.1 | 325.2 |
| 7x7 | 700 | 435 | 490.7 | 483.2 | 491.2 | 486.2 | 486.7 |
| 10x10 | 1000 | 641 | 739.4 | 742.4 | 743.8 | 745.2 | 752 |
| 15x15 | 1500 | 937 | 1042.6 | 1045.0 | 1065.3 | 1048.1 | 1060.1 |

| size | evals. | current best | A2 one-point | A2 uniform | A2 𝕀ℂ𝔼 | A2 p𝕀ℂ𝔼 | A2 jobwise |
|---|---|---|---|---|---|---|---|
| 4x4 | 400 | 193 | 213.0 | 213.0 | 213.0 | 213.0 | 213.0 |
| 5x5 | 500 | 300 | 325.0 | 325.0 | 325.0 | 325.0 | 325.0 |
| 7x7 | 700 | 435 | 463.2 | 460.9 | 462.7 | 460.1 | 461.2 |
| 10x10 | 1000 | 641 | 682.7 | 679.7 | 681.4 | 677.8 | 681.2 |
| 15x15 | 1500 | 937 | 952.1 | 949.5 | 950.0 | 948.1 | 950.6 |

We see that algorithm 2 performs better on the larger problems. On the small problems all runs reach the same value. We believe this is the optimal value in the search space of this scheduler. It cannot generate all schedules, and apparently it cannot generate the optimal schedule. From these figures we can conclude that the heuristic is a good one except for very small problems. Permutation 𝕀ℂ𝔼 outperforms the other algorithms when using algorithm 2.

We realise that the chosen number of evaluations has been very small and this means the conclusions must be interpreted carefully. For a realistic number of evaluations (suppose for instance that one computer gets five minutes to come up with a schedule) the results on a 10x10 problem might look like the 4x4 results of our test.

### 4.4.4 Comparison to previous work

In the article [23] results are given on several problem sizes. They use a fixed number of 10000 evaluations on each problem. Their hybrid GA uses algorithm 2 and is listed in the table below.

| method | result | population |
|---|---|---|
| A2 𝕀ℂ𝔼 | 449.75 | 500 |
| A2 uniform | 450.0 | 500 |
| A2 one-point | 451.5 | 400 |
| Hybrid GA | 455 | 200 |
| Permutation GA | 462 | 200 |
| FGA | 462 | 200 |

We see that 𝕀ℂ𝔼 outperforms uniform crossover, and that our algorithms are slightly superior to the hybrid GA. An explanation is that due to the elitist approach we take we can have more generations for the same number of evaluations and because we have tried different population sizes. The hybrid GA has been tested with a population size of 200. Our algorithms are better even at that population size, but the results listed are reached at the indicated sizes. So our set-up is slightly more efficient than the methods used there, but the difference is not very significant. It is nice to see that 𝕀ℂ𝔼 is a little bit better than uniform crossover. This shows that 𝕀ℂ𝔼 is a good choice if evaluating is expensive.

# Chapter 5

# Conclusions

Solving scheduling problems using evolutionary algorithms can be quite a challenge if your goal is to create the best algorithm in the world.

## 5.1 permutation flow shop

Permutation flow shop scheduling is a nice benchmark problem for permutation evolutionary algorithms because there is only one scheduler. Unfortunately the kind of linkage exploitable by $\mathbb{ICE}$ does not seem important in this problem because one-point crossover, $\mathbb{ICE}$ and permutation $\mathbb{ICE}$ cannot outperform uniform crossover. The behaviour of the $\mathbb{ICE}$ variants is very similar to that of uniform crossover so they can be tried on a 'If it does not help it does not hurt' basis on similar problems with expensive evaluation.

## 5.2 Job shop

The choice of the right scheduler is important in Job Shop scheduling. The naive algorithm performs very badly, unless a special design crossover operator is used. None of our original crossover operators is suited for dealing with multipermutations in this case. The use of the GT algorithm can be very useful. It is important that the decisions of the scheduling algorithm are fed back to the EA by means of forcing.

Job shop scheduling is a problem with a relatively structured first phase in which EAs perform well, and a very hard second phase. This means that algorithms capable of finding acceptable schedules in few evaluations need not be good in finding nearly optimal schedules when given more evaluations. This is shown in the difference between start time forcing(acceptable) and random forcing(nearly optimal), and the difference between no local search(acceptable) and penalized local search(nearly optimal).

The use of local search is certainly recommendable if one needs nearly optimal schedules. If you are willing to spend time on an efficient implementation one can create a state of the art scheduler by combining the right scheduler, local search and an EA.

We have not proven the effectiveness of $\mathbb{ICE}$ and permutation $\mathbb{ICE}$ on these problems. Their behaviour has been too similar to uniform crossover. One explanation can be that both algorithms can only detect factorizations with small groups, whereas the jobwise operator and the success of one-point crossover show that one might need algorithms with a small number of groups. We can see (permutation) 𝕀ℂ𝔼 as a linkage-sensitive form of uniform crossover. Having a linkage-sensitive form of one-point crossover would be very interesting.

## 5.3 Open shop

For open shop scheduling the scheduler is again important but the situation is not as black-and-white as in the job shop case. Our algorithm A2 that uses a heuristic performs better in large problems and should be preferred. Only on small problems (again relative to the allowed number of evaluations) the less clever algorithm A1 yields better results because A2 is not always capable of calculating the optimal schedule.

The use of forcing is not as effective on algorithm A1 as it was in the GT algorithm for job shop. Nevertheless it does yield a small advantage.

𝕀ℂ𝔼 and permutation 𝕀ℂ𝔼 perform better than uniform and one-point crossover on open shop problems using A2 (uniform is better than one-point). The difference is not huge but encouraging.

## 5.4 General results

In the literature it is common to compare the performance of several evolutionary algorithms by running them with a fixed population size for a fixed number of generations. The settings are often (implicitly) chosen to suit uniform crossover. In our experiments we have seen that the convergence speed of an EA depends on use of forcing and local search, and that the optimal population size depends on the crossover operator: Crossover operators introducing less entropy need a larger population for reaching best results. So uniform crossover needs the smallest population size, linkage-discovering operators like 𝕀ℂ𝔼 a larger. The jobwise operator and one-point crossover need the largest population size.

Simple EAs are good in the first phase of optimization, but once more evaluations are available and better results are required their performance drops. From a certain moment it can become useful to apply local search, disrupting operators such as GOX (page 39) and random forcing instead of the more direct start time forcing. They slow down the EA convergence and amount to less systematic and more 'random' search which is beneficial in these circumstances. So the best settings in an EA depend on the number of evaluations available.

For this reasons we advocate the use of fitness versus evaluations plots as used in the experiments chapter. They can be hard to read in the presence of noisy data so they need a large number of runs, but they provide a better view of the performance of the algorithm and show the performance with the optimal setting for each number of available evaluations. These plots have often given meaningful data. Only if the convergence speeds show too much variation they cannot be applied and experiments with a maximal number of evaluations can be used. Even then it pays to test more than one population size.

The results show that EAs can be used very well on the three scheduling problems we have selected (which can be characterized as problems with easily solvable constraints).

# Bibliography

[1] Thomas Bäck and Hans-Paul Schwefel. Evolutionary computation: an overview. *Proceedings of the Third IEEE Conference on Evolutionary Computation 1996*, 1996.

[2] S. Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. *Carnegie Mellon University Tech report*, CMU-CS-94-163, 1994.

[3] James C Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA journal on computing*, 6:154–159, 1994.

[4] Christian Bierwirth and Dirk Mattfeld. Production scheduling and rescheduling with genetic algorithms. *Evolutionary computation*, 7:1–17, 1999.

[5] Jacek Blazewicz, Klaus H. Ecker, Guenter Schmidt, and Jan Weglarz. *Scheduling in Computer and Manufacturing Systems*. Springer-Verlag, 1993.

[6] Peter Bosman and Dirk Thierens. Mixed IDEAs. *technical report UU-CS-2000-45*, 2000.

[7] Peter Bosman and Dirk Thierens. Crossing the road to efficient IDEAs for permutation problems. *Proceedings of the Genetic and Evolutionary Computation Conference - GECCO-2001*, pages 219–226, 2001.

[8] Peter Bosman and Dirk Thierens. New IDEAs and more ICE by learning and using unconditional permutation factorizations. *Late-Breaking Papers of the Genetic and Evolutionary Computation Conference - GECCO-2001*, pages 16–23, 2001.

[9] Stephan Cook. The P versus NP problem. *http://www.claymath.org/prizeproblems/pvsnp.htm.*

[10] Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. The selfish gene algorithm: a new evolutionary optimization strategy. *SAC98:Annual ACM symposium on applied computing*, 1998.

[11] L. Davis. Job shop scheduling with genetic algorithms. *International conference Genetic Algorithms (ICGA-85)*, 1985.

[12] Jeremy S. De Bonet, Charles L. Isbell, and Paul Viola. MIMIC: Finding optima by estimating probability densities. *Advances in Neural Information Processing Systems*, 1997.

[13] M. Dell'Amico and M. Trubian. Applying tabu search to te job shop scheduling problem. *Annals of Operations Research*, 41:231–252, 1993.

[14] B Giffler and G Thompson. Algorithms for solving production scheduling problems. *Operations Research*, 8:487–503.

[15] F. Glover and M. Laguna. *Tabu search.* Kluwer academic publishers, 1997.

[16] David E Goldberg. *Genetic Algorithms in search, optimization, and machine learning.* Addison Wesley, 1989.

[17] G. Harik, E. Cantú-Paz, D. E. Goldberg, and B. L. Miller. The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation*, 7(3):231–253, 1999.

[18] G. Harik, F. Lobo, and D. Goldberg. The compact genetic algorithm. *IlliGal report*, 97006, 1997.

[19] Georges Harik. Linkage learning via probabilistic modelling in the ECGA. *IlliGAL Tech Report*, 99010, 1999.

[20] E. Hart and P. Ross. A heuristic combination method for solving job-shop scheduling problems. *Parallel problem solving from nature V*, 1998.

[21] John H. Holland. *Adaptation in natural and artificial systems.* MIT press, 1992.

[22] Mikkel T. Jensen. Robust and flexible scheduling using evolutionary computation. 2001.

[23] Sami Khuri and Sowmya Rao Miryala. Genetic algorithms for solving open shop scheduling problems. *Lecture Notes in Computer Science*, 1070:10–18, 2001.

[24] Shyh-Chang Lin, Erik D. Goodman, and William F. Punch. A genetic algorithm approach to dynamic job shop scheduling problems. *Proceedings of the sixth annual conference on evolutionary programming*, 1997.

[25] Shyh-Chang Lin, Erik D. Goodman, and William F. Punch. Investigating parallel genetic algorithms on job shop scheduling problems. *Proceedings of the seventh international conference on genetic algorithms*, 1997.

[26] Dirk Mattfeld. Evolutionary search and the job shop. 1996.

[27] Melanie Mitchell. *An introduction to genetic algorithms.* MIT press, 1993.

[28] H. Mühlenbein and G Paass. From recombination of genes to the estimation of distributions. *Parallel Problem Solving from Nature*, 7:178–187, 1996.

[29] Heinz Mühlenbein and Thilo Mahnig. FDA – a scalable evolutionary algorithm for the optimization of additively decomposed functions. *Evolutionary computation*, 7:45–68, 1999.

[30] R. Nakano and T. Yamada. Conventional genetic algorithm for job shop scheduling problems. *Proceedings of the fourth International conference on Genetic Algorithms (ICGA-85)*, pages 474–479, 1991.

[31] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *management science*, 42(6):797–813, 1996.

[32] M. Pelikan and G Mühlenbein. The bivariate marginal distribution algorithm. *Advances in soft computing - Engineering design and manufacturing*, pages 521–535, 1999.

[33] Martin Pelikan, David E. Goldberg, and Erick Cantú-Paz. BOA: The bayesian optimization algorithm. *Proceedings of the GECCO-1999 genetic and evolutionary computation conference*, pages 525–532, 1999.

[34] C Prins. Competitive genetic algorithms for the open-shop scheduling problem. *Mathemetical methods of Operations Research*, 52:389–411, 2000.

[35] Eric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.

[36] D. Thierens. Scalability problems of simple genetic algorithms. *Evolutionary Computation*, 7(4):331–352, 1999.

[37] Manuel Vàzquez and L. Darell Whitley. A comparision of genetic algorithms for the static job shop scheduling problem. *PPSN VI*, 2001.

[38] T. Yamada and C. Reeves. Permutation flow shop scheduling by genetic local search. 1997.

# Index