Evolutionary Computation for the Job-shop Scheduling Problem

C. Soares

UU-CS-1994-52
December 1994

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Evolutionary Computation for the Job-shop Scheduling Problem

C. Soares

Technical Report UU-CS-1994-52
December 1994

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Evolutionary Computation for the Job-shop Scheduling Problem

Carlos Soares*
email: csoares@cs.ruu.nl

Abstract

Our approach solves a version of the Job-shop Scheduling Problem (JSP) with alternative plans for the jobs and alternative machines for the operations. The search space of this formulation of the JSP is considerably larger than the one of the simple JSP, but our systems is robust, obtaining good solutions with a high probability. It also shows little sensitivity to parameter variation and performs equally well in both the simple and the more complex version of the JSP. We obtain worse results than many other researchers but we use a small population for a small number of generations. The manual for a graphical version of the system is also included in this report.

*The work described in this report was partially supported by a COMETT scholarship.

Contents

Chapter 1

Introduction

This paper describes a *Evolutionary Computation* (EC) -based tool for the *Job-shop Scheduling Problem* (JSP).

In the first chapter, we give a brief introduction to scheduling and Evolutionary Computation. In chapter two, we describe our approach in detail. Finally, the results of the experiments performed are presented and analyzed in chapter three.

1.1 Scheduling

Scheduling problems [2], in general, consist of a set of concurrent and conflicting goals, to be satisfied using a limited set of resources. Often, as in the case of the JSP, all valid combinations of these goals and resources constitute an exponentially growing problem (and search) space. In such cases, it is believed that it is impossible to devise deterministic methods or *optimization algorithms* [1] to obtain the optimal solution in polynomial time and using polynomial memory. This means that they are NP-complete.

So, when an optimal solution cannot be found, a *near-optimal* or *good* solution can be very useful. Much work has been done on developing methods that find good solutions with a comfortable degree of confidence. These methods are called *approximation algorithms.* They are classified according to the range of the problems tackled, from *tailored* to *general* [1]. The former are designed for a specific problem type while the latter are applicable to a broad range of problem types. Some of the general approximation algorithms, e.g. Evolutionary Computation can be considered as *algorithmic templates*, since they are generic methodologies and the problem specific details still need to be filled in to obtain an operational algorithm.

Another reason for the use of approximation algorithms is that the real world problems often have characteristics that imply side constraints on the basic scheduling problem. It is often easier to incorporate these constraints in approximation algorithms than in optimization algorithms.

1.2 The Job-shop Scheduling Problem

The Job-shop Scheduling Problem is known to be one of the hardest combinatorial problems [10]. The considerable amount of attention it has received from researchers both in the fields of Operations Research and applied Artificial Intelligence can be explained by its relation to an important problem in a large number of industries, which is the *Manufacturing Planning and Scheduling* (MPS) problem.

Husbands and Mill [8] divide the MPS into several phases, and we concentrate on the following two:

1. problem definition, and
2. simultaneous planning and scheduling.

In the first phase, the **parts** or **components** to be produced and the respective quantities are determined, according to the **orders** and the available stock, thus defining the **jobs** to be completed. Supposing our case consists of an order of 10 units to a bicycle factory, and the stock contains two finished units and three sets of wheels. The list of jobs to be completed consists of five complete bicycles and the production of three bicycles without wheels and the assembly of the stored set of wheels on those. After the jobs are defined, the **plan space** is generated, which consists of all possible **plans** for each job. Each plan consists of a set of **operations** and these operations can be assigned to one or more alternative **machines**. The **schedule space** is implicitly defined when the plan space is generated. A **schedule** is a valid assignment of all the operations to time slots in machines in order to complete the jobs.

In the second phase, both spaces are searched simultaneously, looking for the best solution (a set of plans and their scheduling), according to a predetermined **cost** or **objective function**. This function can take into account general features, like:

- total **makespan**, i.e. the total length of the schedule,
- average job completion time,
- warehouse costs,
- lateness costs,
- earliness costs,
- underutilization of resources,
- client (job) priorities.

or other problem specific ones.

One of the most important aspects of this definition is the *simultaneous* search in both the plan space and the schedule space. Considering that each of them is usually complex, we find ourselves dealing with a huge search space. This is why, traditionally, planning and scheduling are independent processes, with the first creating optimal individual plans which are then fed to the scheduler. This approach is only acceptable when time is a resource with irrelevant cost. Otherwise, it is obvious that a plan that minimizes the cost of a particular component might not be the best when considering all the components together in a valid schedule.

Some common constraints are:

- a machine can perform at most one operation at a time,
- an operation cannot be interrupted.

- critical due dates (jobs which due date cannot be violated),
- machines' maintenance periods, and
- machines' setup times[1] .

The dimension of the search space has created the necessity of dealing with a smaller version of this problem, referred to as *simple JSP* in the rest of this paper. The simple JSP consists of n **jobs** or **orders**, to be completed using m **machines**. Each job consists of a sequence of **operations**, which must be executed in a given order. That order defines the **plan** for the job and it is unique. Each operation has to be executed on a given machine for a given period of time, which defines its **duration**. Two constraints are enforced:

- a machine can perform at most one operation at a time, and
- an operation cannot be interrupted.

The problem is to find a **schedule** (an assignment of the operations to time intervals), such that the **makespan** is minimal.

Although interesting and useful to study, the simple JSP is not sufficient for real-world application. This happens because the simplifications described are by far to restrictive [8].

1.3 Evolutionary Computation

In this section, we make a brief explanation of an Evolutionary Computation (EC) general algorithm. Two examples of clear and complete explanations are the ones by Michalewicz [9] and Goldberg [6].

In EC, a solution to a problem is searched by evolving a population of genotypes. Each genotype encodes a solution, this way defining the **representation**.

That population goes through an iterative process of **selection** and processing by **search operators** which is expected to simultaneously make an exploration of the search space and an exploitation of the best solutions.

The selection process should choose the fittest genotypes more often (exploitation) without totally removing the possibility for lower fitness ones to reproduce (exploration). The **fitness** is a quality measure of the solution represented by a genotype. This way, before the selection, every genotype in the population is evaluated, which means that it will be decoded and then its fitness will be calculated.

Two common search operators are the crossover and the mutation. In the former, a set of one or more genotypes, called parents, exchange information by creating a new one, called ofspring, which inherits values from them. Mutation makes random changes in a genotype.

[1] A machine needs a setup time when an operation different than the previous one is to be executed.

We present the an informal description of a general EC algorithm, adapted from Heitöker and Beasley [7]:

```
// start with an initial time
$t \leftarrow 0$

// initialize a usually random population of individuals
initpopulation $P(t)$

// evaluate fitness of all initial individuals in population
evaluate $P(t)$

// test for termination criterion (time, fitness, convergence, etc.)
while not done

    // increase the time counter
    $t \leftarrow t + 1$

    // select sub-population for offspring production
    $P'(t) \leftarrow$ selectparents $P(t - 1)$

    // recombine the "genes" of selected parents
    recombine $P'(t)$

    // perturb the mated population stochastically
    mutate $P'(t)$

    // evaluate it's new fitness
    evaluate $P'(t)$

    // new population
    $P(t) \leftarrow P'(t)$

endwhile
```

1.4 Evolutionary Computation for JSP

As already stated, Genetic Algorithms (GA), or in a broader sense Evolutionary Computation (EC), are one of the algorithmic templates for combinatorial optimization problems. Although they are more *survival* (or *adaptation*) than *optimization* oriented [3, 6] they have been adapted for this purpose, with successful or, at least, encouraging results.

Some of the features of EC for optimization are:

- they combine directed search (exploitation of good solutions) with stochastic search (exploration of the search space);
- a population of potential solutions is maintained;
- although they are a general (weak) method, they can easily, by including problem-specific information in the implementation, be converted to a specific (strong) method;
- efficiency and robustness.

Hybrid optimization can include an EC, sampling large search spaces randomly and efficiently and obtaining near-optimal solutions, which will be used by another technique, heuristic or deterministic, to look for the optimum.

Some of the EC approaches to the JSP and the MSP are described in [2, 8, 10].

1.5 Acknowledgements

Part of the work described was supported by a COMETT scholarship. I want to thank Prof. dr. J.W. de Bakker from the CWI and dr. Joost N. Kok from Utrecht University who supervised my work; dr. Pedro Henriques and dr. José Neves from Minho University for their support; dr. Guszti Eiben from Utrecht University for the helpful comments; my colleagues Arno Knobbe and Eric Wanrooij for the discussion of ideas and for doing most of the graphical interface; Kees van Kemenade from CWI for the exchange of points of view; finally, all my other colleagues from Utrecht University; finally I want to thank the people from TecMinho, especially José Augusto for all the help.

Chapter 2

Approach

In this chapter, we present in detail the approach we have chosen for our implementation of the Job-shop Scheduling Problem (JSP). It is based on the work by Bagchi *et al.* [2].

We describe the representation, the evaluation, the selection and the operators that constitute our Evolutionary Computation algorithm for the JSP.

2.1 Problem definition

We consider a problem that is a version of the simple JSP that includes part of the planning task: there is the possibility of having alternative plans for the jobs and alternative pairs of machines and durations for the operations [2]. Throughout this paper we will call it *enhanced JSP*.

It is important to notice that both the schedule space and the plan space are searched simultaneously, thus increasing considerably the total search space.

An enhanced JSP is as follows: it consists of n jobs[1] to be completed using m machines. A job is defined as a quantity of a part. To produce a part, one of a set of alternative plans (orderings of operations) must be completed. For every operation there are one or more alternative pairs {machine, duration} that can be used. The following two constraints are enforced:

- a machine can perform at most one operation at a time, and
- an operation cannot be interrupted.

In our implementation, as opposed to the one described in [2], the setup times for the machines are not taken into account.

The problem is to select sequences of operations that will satisfy the n jobs and the assignment of a start time, an end time and resources (a machine) for each operation, in order to minimize the makespan. The makespan, as already stated, is the total time to execute the n jobs.

[1]In [2] a *job* is called an *order*.

jobs:

number	part	quantity
0	0	1
1	1	1

plans:

part	operations
0	0, 1, 2
0	3, 1, 2
1	1, 3
1	2, 0

operations:

operation	machine	duration
0	0	2
1	0	3
1	1	2
2	0	1
2	1	3
3	1	3

Figure 2.1: A simple problem.

2.2 Representation

Bagchi *et al.* [2] describe two types of representation:

- *direct* and
- *indirect.*

In the first representation, the schedule is directly coded in a genotype. In this representation not every genotype corresponds to a valid schedule. In fact, only a small amount of all the possible genotypes are legal. All the other genotypes require correction which represents a significant computational overhead. Also due to the same problem, complicated operators are required: for example, if an operation on a machine is postponed then care must be taken to avoid that it conflicts with the following operation on the same job.

Instead, we use an indirect representation, which means that a **schedule builder** or a **decoder** is necessary to translate a genotype into a valid schedule. This representation is inspired by the one used for the Traveling Salesman Problem in [9], and a genotype represents the queue of jobs, i.e, the scheduling priority of the jobs. In this representation, with an appropriate schedule builder, all the genotypes correspond to valid schedules and both the structure and the operators are simpler. On the other hand, there is the need to create the schedule builder and the computational overhead must be taken into account because the fitness evaluation is executed once every generation for every genotype.

job: 1	op: 1	op: 3	job: 0	op: 0	op: 1	op: 2
	m: 0	m: 1		m: 0	m: 1	m: 0

Figure 2.2: An example of a genotype (*op* is the operation and *m* is the machine).

Together with each job, information is stored about the selected plan and also, with each operation, the selected machine from the set of possible alternatives.

To illustrate the representation, we define a small enhanced JSP with two jobs, two machines and four operations (see figure 2.1).

A genotype for this problem could be the one presented in figure 2.2. It represents a schedule in which the priority of job 1 is bigger than the one of job 0. For job 1, the first alternative plan, which consists of operations 1 and 3, was chosen. For operation 1, the first alternative was chosen (machine 0, duration 3) and operation 3 can only be executed in machine 1 with a duration of 3 time units.

2.2.1 Schedule Builder

Bagchi *et al.* [2] consider that deterministic search should be kept to a minimum. We consider this to be true if the objective of one's work is to study the isolated behavior of the EC algorithm, as is our case. But, if a system is meant to solve real-world problems then a hybrid solution seems to us as a better choice. One such solution could be, for example, the EC algorithm doing the initial sampling of the search space, obtaining a set of good solutions which would then be processed by a conventional optimization technique for scheduling.

In our system all the search is carried out by the EC. The schedule builder simply creates a valid schedule from the genotype. It does so using the information about the selected plans and machines stored in the genotype. When a *conflict* arises between two jobs, then the ordering of the jobs in the genotype is used to solve it. We define a conflict as the situation when more than one operation from different jobs have the same ***desired start time*** on the same machine.

The genotype in figure 2.2 would result in the schedule represented in figure 2.3.

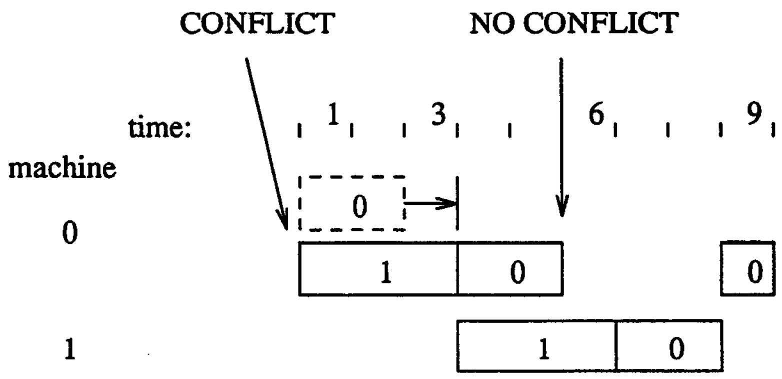


Figure 2.3: The schedule obtained from the example genotype.

An informal description of the schedule building algorithm is as follows. The function `machine(op, j)` returns the machine chosen for operation *op* in job *j* and `duration(op, m)` returns the duration of operation *op* on machine *m*.

```
for all jobs $j$

        $next\_operation$ in $j \leftarrow$ first on the chosen plan
        $desired\_start\_time\_for\_next\_operation$ in $j \leftarrow 1$
end for

$time = 1$
while there are operations to schedule

        for all unoccupied machines $m$ at time $time$

                for all jobs $j$ in the genotype, in a descendent priority, while the slot
                  is not assigned

                        if machine($next\_operation$ in $j$, $j$) $= m$ and
                          the $desired\_start\_time\_for\_next\_operation$ in $j = time$

                                assign slot [$time$, $time$ + duration($next\_operation$ in $j$, $m$) - 1]
                                  to $j$
                                $desired\_start\_time\_for\_next\_operation$ in $j \leftarrow$
                                  $desired\_start\_time\_for\_next\_operation$ + duration($next\_operation$ in $j$, $m$)
                                $next\_operation$ in $j \leftarrow$ next in the chosen plan
                        end if

                end for

        $time \leftarrow time + 1$
        end for

end while
```

When building the schedule represented by the chromosome in figure 2.2, a conflict is detected on *time* = 1 because both jobs want to start at the same time on the same machine. But on *time* = 6 there is no conflict because job 1 is already scheduled at the time when job 0 wants to use machine 1. So, the latter must wait until job 1 finishes the operation on that machine.

2.3 Evaluation and Selection

The fitness function used is a simplified version of the one that Bagchi *et al.* [2] use. It equals the inverse of the makespan of the schedule. This means that the genotypes with higher fitness are the ones with smaller makespan. This is the most common fitness measures used for the JSP.

Our system uses tournament selection [6, 9] with tournament size 2: for every place in the mating pool, two genotypes are randomly picked, and the one with the highest fitness is selected to fill that place.

2.4 Search operators

The operators implemented are similar to those described in [2]. Each of these operators is explained in the next sub-sections. They are:

- two crossover operators:
  - order crossover #1 (OX), and
  - plan crossover (PX);
- and two mutation operators:
  - swap mutation (SM), and
  - plan mutation (PM).

Instead of the partially-mapped crossover (PMX), as Bagchi *et al.* [2], we have implemented the order crossover #1 (OX). A description of these order based crossover operators is given by Starkweather *et al.* [11].

2.4.1 Crossover operators

The OX crossover is widely used in order representations. The description presented here was adapted from [9]:

1. Randomly select two parents.
2. Select a random sub-string from the first parent.
3. Fill the offspring from the beginning to the position where the chosen substring starts, with jobs (and respective plans) from the second parent. The jobs are chosen, starting from the beginning of the second parent, and skipping every job which belongs to the chosen substring.
4. Copy the chosen substring (including the job's plans) from the first parent to the offspring.
5. Fill the rest of the offspring as in step 3.

Figure 2.4 shows an example of how this operator creates offspring from two parents.
The problem-specific crossover (PM) we implemented is slightly different from the one described in [2]. Our version can be described as follows:

1. Randomly select two parents.
2. Select a random sub-string from the first parent.
3. Exchange plans (and selected machines for the operations) in the selected substring with the corresponding element in the second parent.

An example of this kind of crossover is presented in figure 2.5.
In average, the number of crossover operations executed each cycle is:

$$popsize * pC$$

in which, *popsize* is the population size and *pC* is the crossover probability.

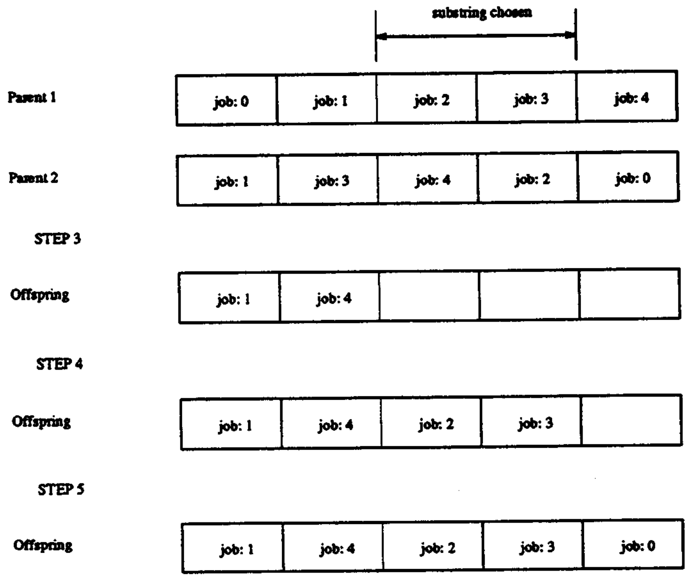


Figure 2.4: An example of the domain independent crossover (OX) operation. The chosen plans and operations were omitted for the sake of clarity.

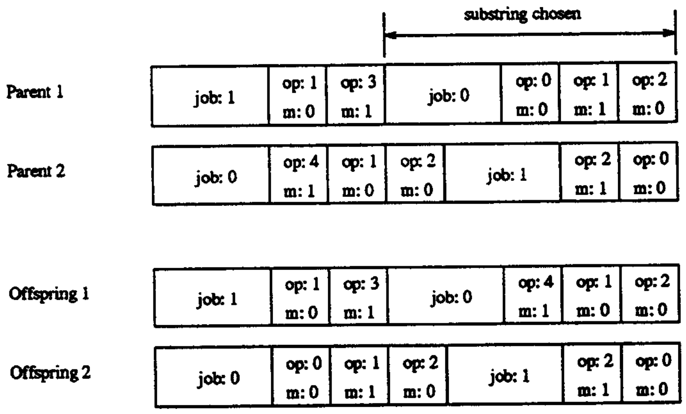


Figure 2.5: An example of the problem specific crossover (PM) operation.

2.4.2 Mutation operators

The swap mutation exchanges the position of two randomly selected jobs in a randomly chosen genotype.

The plan mutation randomly selects a genotype and a position in it, and then randomly chooses a plan and machines for the operations of that plan.

In average, the number of mutation operations executed in each cycle is:

$$popsize * n * pM$$

in which, *popsize* is the population size, n is the number of jobs and *pM* is the mutation probability.

2.4.3 Selection of operators

As pointed out by Bagchi *et al.* [2], a question arises on how to combine the two different operators in each of the operator types. To solve this problem we associate a probability value with the problem independent operator in each of the types (*spC* and *spM*). These values give the probability of selection of the domain independent operator for executing all the operations in a generation. We give an example using the crossover operator type. If the selection probability of the problem independent crossover operator (OX) is $spC = 0.2$, then the selection probability of the plan crossover (PX) will be 1.0 - 0.2 = 0.8. In this case the domain independent operator will be used in 20% of the crossover operations and the problem specific one in the other 80% .

In [2], only one selection probability is used for both the types of operators, i.e. $spC = spM$. We decided to use one for each because we think that a value that gives the best combination between the problem independent and the problem specific operators on one type of operators, might not yield an equally optimal result on the other type.

Every generation only one operator of each type is used, which means that all crossovers will be of the same kind, and also the mutations will be all of the same type.

Chapter 3

Results

Our aim was to test this approach not only in JSPs with simultaneous planning and scheduling (enhanced JSPs), which was already done by Bagchi *et al.* [2], but also in instances of the simple JSP.

The program was tested with three problems (see appendix C). Two of them are well known and often used. They are instances of simple JSP, i.e., there is only one plan for each job and only one machine for each operation. The representation used is thus redundant and the problem specific operators were not used, except in one simple test. The latter was carried out to prove that those operators are not useful for this kind of problem, but nevertheless, the system performs a successful search.

The third problem is used by Bagchi *et al.* [2] to test their implementation. In this problem, each part has two or three alternative plans, and each operation can be executed on one or two alternative machines. In our implementation, the setup times are not considered.

The first problem, to which we will refer as **Problem 1** (see appendix C), is a 6x6 problem, which means that it has six jobs ($n = 6$) and six machines ($m = 6$) with a best solution known to be 55. The second problem (**Problem 2**) is a 10x10 problem with a best solution known to be 930. In these problems the number of operations to schedule is fixed and is 36 for **Problem 1** and 100 for **Problem 2**. We could not find information on the best solution for the third problem (**Problem 3**).

We used these problems to test the effect of parameter variation on the quality of the solutions. The parameters tested are population size (*popsize*), crossover rate (*pC*), mutation rate (*pM*) and the selection probabilities of the domain independent over the problem specific implementations of these operators (*spC* and *spM*).

3.1 Testing conditions

Every test was carried out for 100 generations on **Problem 1** and 200 on the others. Fewer generations were studied on the first problem because it is a simple problem. All results are averaged over five runs, except when otherwise indicated. In the tables, the column named *best* lists the best result in the five runs, and the *average* and *variance* columns respectively list the average of the best over the five runs and the variance on that average.

popsize	best	average	variance
10	58	58.0	0.0
30	58	58.0	0.0
50	58	58.0	0.0
100	58	58.2	0.16

Parameters	
Problem	**Problem 1**
Population	**in test**
# Generations	100
Crossover rate	0.6
spC	1.0
Mutation rate	0.1
spM	1.0
# runs	5

popsize	best	average	variance
10	1013	1034.2	122.96
30	1013	1024.0	48.4
50	1017	1023.6	14.64
100	997	1010.8	158.56

Parameters	
Problem	**Problem 2**
Population	**in test**
# Generations	200
Crossover rate	0.6
spC	1.0
Mutation rate	0.1
spM	1.0
# runs	5

popsize	best	average	variance
10	4385	4565.0	38420.0
30	4125	4141.0	384.0
50	4125	4125.0	0.0
100	4125	4125.0	0.0

Parameters	
Problem	**Problem 3**
Population	**in test**
# Generations	200
Crossover rate	0.6
spC	0.5
Mutation rate	0.1
spM	0.5
# runs	5

Figure 3.1: Effect of population size.

3.2 Population size

The set of values tested for the population size was {10, 30, 50, 100}. Standard probabilities of 0.6 and 0.1 were used, respectively, for crossover rate and mutation rate. The selection probabilities of the domain independent operators over the problem specific ones were 1.0 on the first two problems, which means that only the former were used. On the other problem values of 0.5 were used, which means that each operator is chosen with 50% chance.

The results are presented in figure 3.1.

Problem 1 is a simple problem as the results show. The slight decrease in the average for *popsize* = 100 is due to one run with a best result of 59, which can be explained by the stochastic nature of the process. In the two other problems, the evolution is as expected: the quality of the solutions increases with the population size. In **Problem 2**, the variance increases for *popsize* = 100, but not significantly. In conclusion, a population size of at least 30 should be used.

3.3 Crossover probability

The values tested for the crossover rate were {0.0, 0.1, 0.5, 0.6, 0.7, 0.8, 0.9}. In these experiments, the population size was 30, and the mutation probability was 0.1. Both the operator selection probabilities were set to 1.0 on the first two problems and to 0.5 on **Problem 3**. The results of the first set of experiences were not conclusive, so another set of tests was performed only this time the results were averaged over 10 runs.

The results of the second set of tests are presented in figure 3.2.

In the problems tested the optimal value for the crossover rate appears to be in the range [0.6, 0.8] although the results are not conclusive. Overall the chosen crossover operator does not seem to have a big effect, which is compatible with the work by Eiben and Perck [4] and with the comparison of sequencing operators done by Starkweather *et al.* [11]. In the latter study it is shown that the order crossover #1 does not have a good performance in scheduling problems. For this kind of problems, the referred study concludes that the best crossover operators are the cycle crossover and the order crossover #2.

3.4 Mutation probability

The set of values tested was {0.0, 0.05, 0.1, 0.3, 0.5, 0.7, 0.9}. In these experiments, the population size was 30, and the crossover probability was 0.6. Both the operator selection probabilities were set to 1.0, on the first two problems and to 0.5 on the third.

The results are presented in figure 3.3.

In **Problem 1** and **Problem 2** the results seem to indicate that the value for the mutation rate should be at least 0.1 but it is difficult to define a small optimal range. In **Problem 3** the results are similar with the usual values for the mutation rate, and indicate 0.05 and 0.1 as the best.

pC	best	average	variance
0.0	58	58.3	0.21
0.1	58	58.2	0.16
0.5	58	58.1	0.09
0.6	58	58.0	0.0
0.7	58	58.2	0.16
0.8	58	58.2	0.16
0.9	58	58.3	0.21

Parameters	
Problem	**Problem 1**
Population	30
# Generations	100
Crossover rate	**in test**
spC	1.0
Mutation rate	0.1
spM	1.0
# runs	10

pC	best	average	variance
0.0	997	1028.9	271.49
0.1	997	1031.1	203.29
0.5	1018	1032.8	76.96
0.6	1017	1028.6	93.84
0.7	997	1016.0	182.6
0.8	997	1027.7	205.41
0.9	1013	1027.0	82.8

Parameters	
Problem	**Problem 2**
Population	30
# Generations	200
Crossover rate	**in test**
spC	1.0
Mutation rate	0.1
spM	1.0
# runs	10

pC	best	average	variance
0.0	4125	4385.0	8416.0
0.1	4125	4187.0	8036.0
0.5	4125	4172.5	20306.25
0.6	4125	4153.5	2500.25
0.7	4125	4139.0	804.0
0.8	4125	4125.0	0.0
0.9	4125	4179.0	10244.0

Parameters	
Problem	**Problem 3**
Population	30
# Generations	200
Crossover rate	**in test**
spC	0.5
Mutation rate	0.1
spM	0.5
# runs	10

Figure 3.2: Effect of crossover probability.

pM	best	average	variance
0.0	58	59.8	1.76
0.05	58	58.2	0.16
0.1	58	58.0	0.0
0.3	58	58.0	0.0
0.5	58	58.0	0.0
0.7	58	58.0	0.0
0.9	58	58.0	0.0

Parameters	
Problem	**Problem 1**
Population	30
# Generations	100
Crossover rate	0.6
spC	1.0
Mutation rate	**in test**
spM	1.0
# runs	5

pM	best	average	variance
0.0	1018	1066.6	629.04
0.05	1018	1040.8	189.76
0.1	997	1024.4	208.24
0.3	1016	1027.4	87.84
0.5	1022	1035.0	115.2
0.7	1018	1031.6	75.44
0.9	997	1024.2	230.56

Parameters	
Problem	**Problem 2**
Population	30
# Generations	200
Crossover rate	0.6
spC	1.0
Mutation rate	**in test**
spM	1.0
# runs	5

pM	best	average	variance
0.0	5580	6323.0	363436.0
0.05	4125	4137.0	576.0
0.1	4125	4180.0	6100.0
0.3	4200	4364.0	16064.0
0.5	4360	4553.0	44276.0
0.7	4700	4825.0	6420.0
0.9	4685	4825.0	30340.0

Parameters	
Problem	**Problem 3**
Population	30
# Generations	200
Crossover rate	0.6
spC	0.5
Mutation rate	**in test**
spM	0.5
# runs	5

Figure 3.3: Effect of mutation probability.

spC	*spM*	best	average	variance
1.0	1.0	58	58.0	0.0
0.5	1.0	58	58.2	0.16
1.0	0.5	58	58.4	0.24
0.5	0.5	58	58.2	0.16

Parameters	
Problem	**Problem 1**
Population	30
# Generations	100
Crossover rate	0.6
spC	**in test**
Mutation rate	0.1
spM	**in test**
# runs	5

spC	*spM*	best	average	variance
1.0	1.0	1013	1022.6	30.24
0.5	1.0	1017	1023.4	51.44
1.0	0.5	1019	1033.2	109.36
0.5	0.5	1018	1038.6	543.04

Parameters	
Problem	**Problem 2**
Population	30
# Generations	200
Crossover rate	0.6
spC	**in test**
Mutation rate	0.1
spM	**in test**
# runs	5

Figure 3.4: Effect of domain independent/problem-specific selection probabilities in **Problem 1** and **Problem 2.**

3.5 Operator selection probabilities

The two operator selection probabilities were studied at the same time for the first two problems, which are simple JSPs. Four tests were executed with the following values: $spC = 1.0, spM = 1.0; spC = 0.5, spM = 1.0; spC = 1.0, spM = 0.5; spC = 0.5, spM = 0.5$. In the third problem they were studied one at a time, with the other being set to 0.5. The set of values used for the parameter in test was {0.0, 0.2, 0.4, 0.6, 0.8, 1.0}. For these tests, the population size was 30, the crossover probability was 0.6 and the mutation probability was 0.1.

The results are presented in figures 3.4 and 3.5.

As would be expected, in the two first problems, where there are no alternative plans, the problem specific operators are not useful, and the best results are achieved using only the domain independent ones. Inspite of that, the decrease in performance is not significant.

The first set of tests to study the operator selection probabilities for the crossover operators on **Problem 3** did not yield conclusive results and so another set of tests was performed, this time averaging results over 10 runs. Analyzing the final results for this problem, we conclude that the value for both probabilities should be in the interval [0.2, 0.8]. For the crossover operators the best results are obtained with $spC = 0.8$ and they decrease from that value to 0.2. As for the mutation operators, the best are obtained with 0.2 and decrease until $spM = 0.8$. In both cases the decrease is not very significant. Similar results were obtained by Bagchi *et al.* [2], although they use the same probability for both types of operators ($spC = spM$).

spC	best	average	variance
0.0	4125	4204.5	18382.25
0.2	4125	4159.5	3872.25
0.4	4125	4145.0	1920.0
0.6	4125	4136.5	1190.25
0.8	4125	4131.0	324.0
1.0	4125	4176.5	6295.25

Parameters	
Problem	**Problem 3**
Population	30
# Generations	200
Crossover rate	0.6
spC	**in test**
Mutation rate	0.1
spM	0.5
# runs	10

spM	best	average	variance
0.0	4320	4496.0	9024.0
0.2	4125	4125.0	0.0
0.4	4125	4208.0	17356.0
0.6	4125	4223.0	24016.0
0.8	4125	4303.0	17366.0
1.0	5580	6914.0	1059794.0

Parameters	
Problem	**Problem 3**
Population	30
# Generations	200
Crossover rate	0.6
spC	0.5
Mutation rate	0.1
spM	**in test**
# runs	10

Figure 3.5: Effect of domain independent/problem-specific selection probabilities **Problem 3**.

3.6 Conclusions and future work

Our objective was to study the behavior of a representation designed for enhanced JSPs (JSPs with simultaneous planning and scheduling) not only in this kind of problems but also in simple JSPs.

For the two simple JSPs tested, we have obtained good results (within 10% of the optimum), although in a problem like the 6x6 (**Problem 1**) better results were expected, due to its simplicity: our best was 58, which is 5.4% from the optimum, 55. This can be explained intuitively by the fact that in this approach only one priority list (or ordering) is used to resolve conflicts in every machine, and the best ordering for a set of jobs on one machine may not be the best on another. A different reason could be the conflict definition adopted, which only considers the situation where two jobs have the same desired start time on the same machine. These problems could possibly be avoided with changes in the representation or with a different decoder. In the 10x10 problem (**Problem 2**), which is considered to be hard, the results are quite good: the best result we have obtained was 997, which is within 7.3% of the optimum, 930. We have performed an experiment for each of these two problems with *popsize* = 500 and for 300 generations, and no better solutions were found. We found no information about the best results for the problem described by Bagchi *et al.* [2] (**Problem 3**), although they could only be used as a reference, because we solved an adapted version which did not consider the setup times. Our best was 4125.

The table in figure 3.6 was based on two tables, one from Nakano and Yamada [10] and the other from Yamada and Nakano [12] and also some results from Fang *et al.* [5]. It lists some

Papers	Algorithms	6x6	10x10
Barker85	BAB	55	960
Carlier89	BAB	55	930
Nakano91	EC	55	965
Yamada92	EC	55	930
Fang93	EC	-	949
Soares94	EC	58	997
Optimal		55	930

Figure 3.6: Table of best results of several approaches

Papers	*popsize*	#generations	#evaluations
Nakano91	1000	150	150000
Bagchi91	50	1000	50000
Fang93	500	300	150000
Soares94	30	200	6000

Figure 3.7: Table of best results of several approaches

of the results obtained by other researchers on the two simple JSPs that were also tested in this report. The algorithms used are either Branch and Bound (BAB) or Evolutionary Computation (EC)[1].

Our results, although worse than the ones obtained by all other EC approaches, have some interesting features when compared to them. Either the population size or the number of generations we normally use is smaller than the ones that all the others use (see figure 3.7). Also listed in this table are the values for those parameters used by Bagchi *et al.* [2], although they do not test their implementation in these problems. In their paper, Yamada and Nakano [12] do not describe the values used for the parameters.

We also want to stress the fact that the results listed in table 3.6 are, in some cases, the best obtained in a series of tests. In the conclusions of their paper, Yamada and Nakano [12] state that:

> (...) the rate of obtaining optimal solutions is still small, so further improvements to the algorithm are required (...)

We have obtained a small variance in the average best values over a series of five or ten runs with the same parameters, which is a good indication of the robustness and the efficiency in finding always a good solution.

Another interesting characteristic of our system is the absense of some common features in EC algorithms and GAs for optimization, e.g. elitism. The implementation of such mechanisms could result in a better solution quality.

Tests were made on the parameter sensitivity. Although some conclusions were drawn, the results are not very clear and seem very problem dependent. Anyway, it seems that the

[1]We include the Genetic Algorithm approaches in the broader class of Evolutionary Computation.

system is not very sensitive to changes in the parameters, which increases our believe in its robustness. In general more tests are necessary with this and other problems.

The choice of the order crossover #1 was not the best, as Starkweather *et al.* demonstrate in their comparison of sequencing operators for scheduling [11]. An order crossover #2 or a cycle crossover should be used instead. Inspite of this, we consider that the refinement of the representation or of the decoder is more important to improve the quality of the solutions than the order operators used. This conclusion is compatible with the work of Eiben and Perck [4].

Also the time complexity of the program was tested, but the results are not yet fully analyzed. We stress that the code was not optimized. Comparison with other approaches would be important to measure the computational efficiency of this approach. Some code optimization would be useful before that.

A lot of work remains to be done regarding the Job-shop Scheduling Problem. Some refinement could be done on the representation and new operators could be defined with more problem specific knowledge included. Also some different approaches for the schedule builder can be tested. The most important, however, is the extension of the problem to move it towards the practical problem it has originated from, the Manufacturing Scheduling and Planning.

Chapter 4

User Manual

We describe the use of the program implemented. The graphical version was built using two toolkits developed at Utrecht University[1]:

- Forms Library and
- PlaGeo.

This tool can be retrieved from the anonymous ftpsite **ftp.cs.ruu.nl** in the directory `pub/RUU/CS/NEURO/Planning`. It is compiled for the IRIX 5.2 Operating System on the SGI Indy/Indigo Series. The files needed to run it are listed in the file `README.JSP`.

4.1 Start

The command line can have one of two formats:

- `niceguy` [RETURN]
- `niceguy` *parameters_file* [RETURN]

In the first case the program loads the *default parameters file* (see section 4.2.2 below) and in the other, the parameters are read from the file with file name *parameters_file* (see section 4.2 below).

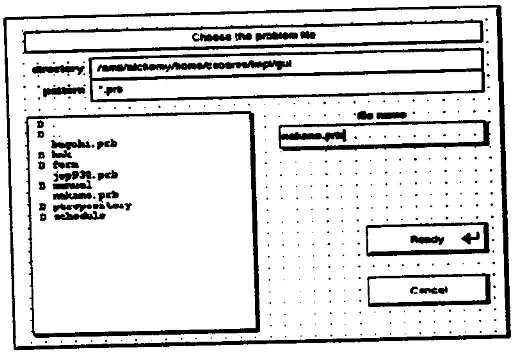
If there is no default parameters file, or the parameters file can not be opened[2], a warning message and then file selector (see figure 4.1) will be presented (see section 4.2.1 below). The default extension for parameters files is '*.par'.

If **Cancel** is pressed, then hard-coded default values will be used and another file selector will be presented, this time to choose the problem file (see section 4.3 below). In case no problem file is selected, then the user will be prompted to confirm the exit from the program.

The interface for the program can be seen in figure 4.2.

[1]These libraries can be retrieved from the ftpsite **ftp.cs.ruu.nl**, in the directories `/pub/SGI/FORMS` and `pub/SGI/GEO`, respectively.

[2]Whether it is the default parameters file or the one given by the user.



directory: /ams/alchemy/home/csoares/impl/gui
pattern: *.prb
file name: nakano.prb

```
D .
D ..
  bogohi.prb
D bak
D form
  jup930.prb
D manual
  nakano.prb
D prevyearstuey
D schedule
```

Figure 4.1: File selector. In this case, to choose the parameters file.

4.2 Parameters File

The format of the parameters file is described in appendix A.

4.2.1 Choose Parameters File

To choose a different parameters file, the **Parameters File** button should be pressed.

If there are unsaved changes in the parameters, then a file selector will be presented to save this parameter setting. If saving is not desired, then press the **Cancel** button.

Then another file selector is presented to choose the new parameters file. Eventual errors in the parameters file will cause a file with the name '*parameters_file_name*`.errors`' to be created. Those parameters that can not be read from the file, will be initialized to the corresponding hard-coded values. If **Cancel** is pressed then the last used parameters file will be reloaded.

The parameters file can be changed only before the GA is started.

When one quits the program (see section 4.7 below) and parameters have been changed, then a file selector is presented, to save the changes.

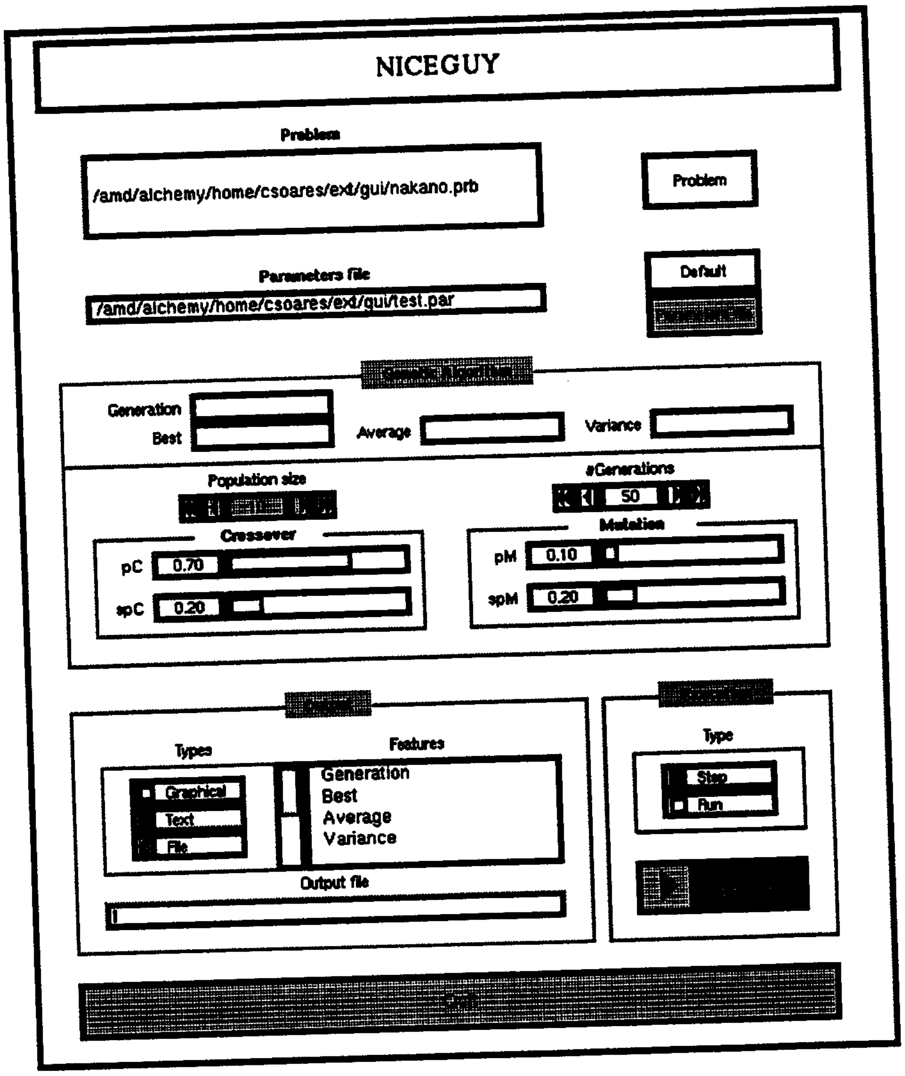
4.2.2 Default Parameters File

To set the default parameters file, click the **Default** button.

4.3 Problem File

The format of the problem file is described in appendix B.

To select a new problem file, the **Problem** button should be pressed.



Problem: /amd/alchemy/home/csoares/ext/gui/nakano.prb

Parameters file: /amd/alchemy/home/csoares/ext/gui/test.par

Generation
Best
Average
Variance

Population size

#Generations: 50

Crossover
pC: 0.70
spC: 0.20

Mutation
pM: 0.10
spM: 0.20

Generation
Best
Average
Variance

Figure 4.2: The interface of the program.

If errors are detected in a problem file, whether when the program starts or the user selects a new one, then a warning will be presented and an errors file is created with the name '*problem_file_name*.`errors`'. Then, the file selector will be shown once again.

If **Cancel** is pressed, then the previous problem, if any, will be reloaded.

4.4 GA Parameters

There are six GA parameters:

Population size This is the number of chromosomes in the population. This parameter can only be changed before the GA is started.

#Generations The value of this parameter determines the number of cycles the GA will run.

pC The probability of the crossover operator be applied in a chromosome.

ratioC This value is the selection probability of the domain independent crossover operator (PMX) being selected over the problem specific one.

pM The probability of the mutation operator be applied to a chromosome.

ratioM This value is the selection probability of the domain independent mutation operator being selected over the problem specific one.

4.4.1 Effect of Changes in the Parameters

All the parameters described, except for the population size, can be changed anytime during the execution of this program. This means that they can be changed when the GA is not started, when it is paused or when it is running. The changes made are introduced in the GA before it starts, or after the completion of the current cycle.

4.5 Output

4.5.1 Output Types

Three types of output are available:

- graphical,
- text, and
- file output.

The second type of output is directed to the terminal where the program was called.

If it is not possible to create the given output file, then a file selector will be presented. Also, if the **File** button is on and the execution is initiated and there is no output file indicated, the file selector will appear.

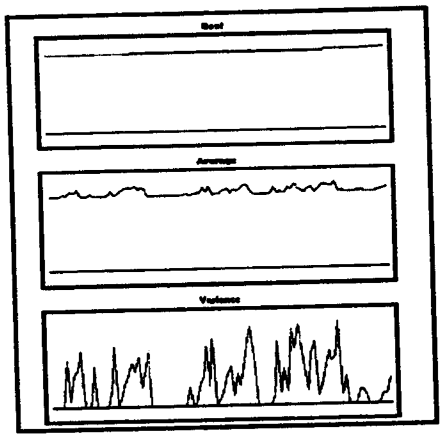


Figure 4.3: Charts window.

4.5.2 Output Features

There are eight output features implemented. To enable/disable a feature, click the output features window on the respective name.

We will explain their meanings in three groups:

Generation Current generation number.

Best The fitness of the best chromosome so far.

Average The average fitness of the current population.

Variance The variance on the average fitness of the current population.

These values are presented in the control panel, above the parameter settings. In the text and file interfaces, when the a new generation is finished, their values are printed and saved, respectively.

Best (Chart)

Average (Chart)

Variance (Chart)

These charts are displayed in a separate window (figure 4.3), that will be shown only when at least one of this features is enabled. It is important to notice that the *time* (x axis) is increases from right to left.

These features do not produce text or file output.

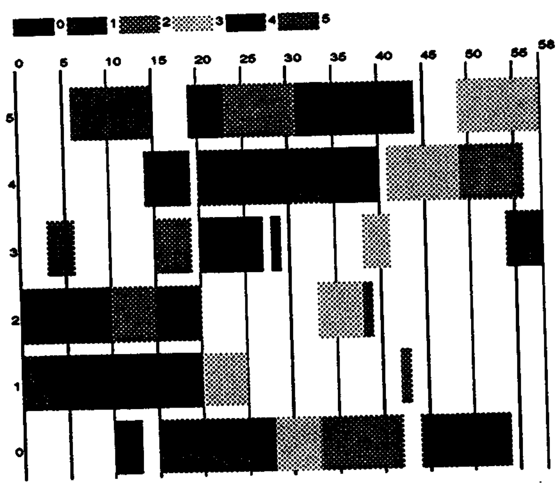


Figure 4.4: Output of a best schedule for a 6x6 problem.

Best Schedule This feature presents a graphical representation of the schedule for the best chromosome so far (figure 4.4). It is possible to print it or save it to a postscript file. In the latter case, a file selector will be presented.

This feature does not also produce text or file output.

4.6 Execution

4.6.1 Execution Types

There are two possible execution types:

Step Pauses after the initialization of the GA and after every cycle,

Run Runs for the given number of generations or until the **Pause** or **Stop** buttons are pressed.

4.6.2 Execution Control

The following are the execution control buttons:

Play (▷) This button starts the execution of the GA. It will remain pressed until the execution is over.

Pause (| |) This button is used to stop the execution. It will have effect only after the current cycle ends. It will automatically be pressed after each cycle in **Step** execution type (see section 4.6.1 above).

Stop (□) When this button is pressed, a window is presented to confirm the interruption of the current execution.

4.6.3 End of Execution

When all the desired number of generations is passed, a confirmation on the end of the execution is required. To continue, change the number of generations and release the **Pause** button by clicking it (see section 4.6.2 above).

4.7 Quit

When the **Quit** button is clicked, a confirmation window is presented.

Bibliography

[1] Aarts E.H.L., van Laarhoven P.J.M., Lenstra J.K., Ulder N.L.J., "A Computational Study of Local Search Algorithms for Job Shop Scheduling", Operations Research Society of America Journal on Computing, Vol. 6, No. 2, 1994

[2] Bagchi S., Uckun S., Miyabe Y., Kawamura K., "Exploring Problem-Specific Recombination Operators for Job Shop Scheduling", *in* Belew R.K. and Booker L.B. (Eds), Proceedings of the 4th International Conference on Genetic Algorithms, Morgan Kaufmann, 1991

[3] De Jong, K.A., "Are genetic algorithms function optimizers", *in* Manner R. and Manderick B. (Eds), Parallel Problem Solving from Nature - 2, Elsevier Science Publishers, 1992

[4] Eiben A.E. and Perk C., "Comparing Decoders and Crossovers for Scheduling by Genetic Algorithms", to be submitted for the ICGA-95, 1994

[5] Fang H., Ross P., Corne D., "A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling and Open-Shop Problems", *in* Forrest S. (Ed), Proceedings of the 5th International Conference on Genetic Algorithms, Morgan Kaufmann, 1993

[6] Goldberg D.E., "Genetic Algorithms in Search, Optimization and Machine Learning", Addison-Wesley, 1986

[7] Heitkötter J. and Beasley D. (eds), "The Hitch-Hiker's Guide to Evolutionary Computation: A list of Frequently Asked Questions (FAQ)", USENET: comp.ai.genetic. Available via anonymous FTP from **rtfm.mit.edu:/pub/usenet/news.answers/ai-faq/genetic/** About 90 pages, 1994

[8] Husbands P., Mill F., "Simulated Co-Evolution as The Mechanism for Emergent Planning and Scheduling", *in* Belew R.K. and Booker L.B. (Eds), Proceedings of the 4th International Conference on Genetic Algorithms, Morgan Kaufmann, 1991

[9] Michalewicz Z., "Genetic Algorithms + Data Structures = Evolution Programs", Springer-Verlag, 1992

[10] Nakano R., Yamada T., "Conventional Genetic Algorithms for Job Shop Problems", *in* Belew R.K. and Booker L.B. (Eds), Proceedings of the 4th International Conference on Genetic Algorithms, Morgan Kaufmann, 1991

[11] Starkweather T., McDaniel S., Mathias K., Whitley D., Whitley C., "A Comparison of Genetic Sequencing Operators" *in* Belew R.K. and Booker L.B. (Eds), Proceedings of the 4th International Conference on Genetic Algorithms, Morgan Kaufmann, 1991

[12] Yamada T. and Nakano R., "A Genetic Algorithm Applicable to Large-Scale Job-Shop Problems", *in* Manner R. and Manderick B. (Eds), Parallel Problem Solving from Nature - 2, Elsevier Science Publishers, 1992

Appendix A

Parameters file format

In this appendix, the format of the parameters file is described.

It consists of two sections, the interface and the GA section. The first is initialized with the command word **INTERFACE**, and the second with **GA**. The interface section is divided in two parts, the output type, with the information about what types of output are desired, and the name of the output file, if any.

The default file extension is '*.par'.

Bold expressions are commands, which must be in uppercase, e.g. **INTERFACE**; Italic expressions are boolean variables, e.g. *graphical_interface*(bool) which take integer values **0** or **1**; integer variables, e.g. *population_size*(int); probability variables which are float variables in the range [0.00, 1.00], e.g. *crossover_probability*(prob) or character strings, e.g. *output_file*(str).

The file name line is only necessary if the file output is on.

INTERFACE	
GRAPHICAL:	*graphical_output*(bool)
TEXT:	*text_output*(bool)
FILE:	*file_output*(bool)
FILENAME:	*output_file*(str)
THISGENERATION:	*this_generation_feature*(bool)
BEST:	*best_fitness_feature*(bool)
AVERAGE:	*pop_average_fitness_feature*(bool)
VARIANCE:	*pop_fitness_variance_feature*(bool)
CHARTBEST:	*best_fitness_chart_feature*(bool)
CHARTAVERAGE:	*pop_average_fitness_chart_feature*(bool)
CHARTVARIANCE:	*population_fitness_variance_chart_feature*(bool)
BESTSCHEDULE:	*best_schedule_feature*(bool)
GA	
PROBLEM:	*problem_file*(str)
POPSIZE:	*population_size*(int)
GENERATIONS:	*no_of_generatinons*(int)
PROBCROSSOVER:	*crossover_probability*(prob)
PROBPMX:	*domain_ind_crossover_sel_probability*(prob)
PROBMUTATION:	*mutation_probability*(prob)
PROBPIMUTATION:	*domain_ind_mutation_sel_probability*(prob)

Appendix B

Problem file format

In this appendix, the file format for problems is described.

The default file extension is '*.prb'.

Bold expressions are commands, which must be in uppercase, e.g., **NRORDERS:**; Italic expressions are integer variables, e.g., *number_of_orders*.

Note that, after every alternative for an operation, a colon is necessary.

```
NRORDERS:          no_of_orders
NRMACHINES:        no_of_machines
NRPLANS:           total_no_of_plans
NROPERATIONS:      no_of_operations
ORDERS:
        part_no         quantity
        part_no         quantity
           .               .
           .               .
           .               .
PLANS:
        part_no         no_of_operations      operation1 operation2 ...
        part_no         no_of_operations      operation1 operation2 ...
           .                 .                    .          .
           .                 .                    .          .
           .                 .                    .          .
OPERATIONS:
        no_of_alternatives    alternative_machine1_no duration1 :
                              alternative_machine2_no duration2 :
                                        .                 .
                                        .                 .

        no_of_alternatives    alternative_machine1_no duration1 :
                              alternative_machine2_no duration2 :
                                        .                 .
                                        .                 .
                 .                              .
                 .                              .
                 .                              .
```

Appendix C

Problems

C.1 Problem 1 : a 6x6 problem

jobs:

part	quantity
0	1
1	1
2	1
3	1
4	1
5	1

plans:

part	operations
0	10, 0, 7, 14, 26, 21
1	4, 9, 16, 24, 3, 12
2	9, 12, 22, 2, 8, 18
3	5, 1, 9, 13, 19, 23
4	11, 6, 17, 25, 0, 15
5	6, 13, 23, 3, 20, 10

operations:

operation	machine	duration
1	0	3
1	0	5
1	0	9
1	0	10
1	1	8
1	1	5
1	1	3
1	1	6
1	1	1
1	2	5
1	2	1
1	2	9
1	3	4
1	3	3
1	3	7
1	3	1
1	4	10
1	4	5
1	4	7
1	4	8
1	4	4
1	4	6
1	5	8
1	5	9
1	5	10

1	5	4
1	5	3

C.2 Problem 2 : a 10x10 problem

jobs:

part	quantity
0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1

plans:

part	operations
0	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
1	10, 11, 12, 13, 14, 15, 16, 17, 18, 19
2	20, 21, 22, 23, 24, 25, 26, 27, 28, 29
3	30, 31, 32, 33, 34, 35, 36, 37, 38, 39
4	40, 41, 42, 43, 44, 45, 46, 47, 48, 49
5	50, 51, 52, 53, 54, 55, 56, 57, 58, 59
6	60, 61, 62, 63, 64, 65, 66, 67, 68, 69
7	70, 71, 72, 73, 74, 75, 76, 77, 78, 79
8	80, 81, 82, 83, 84, 85, 86, 87, 88, 89
9	90, 91, 92, 93, 94, 95, 96, 97, 98, 99

operations:

operation	machine	duration
0	0	29
1	1	78
2	2	9
3	3	36
4	4	49
5	5	11
6	6	62
7	7	56
8	8	44
9	9	21
10	0	43
11	2	90

12	4	75
13	9	11
14	3	69
15	1	28
16	6	46
17	5	46
18	7	72
19	8	30
20	1	91
21	0	85
22	3	39
23	2	74
24	8	90
25	5	10
26	7	12
27	6	89
28	9	45
29	4	33
30	1	81
31	2	95
32	0	71
33	4	99
34	6	9
35	8	52
36	7	85
37	3	98
38	9	22
39	5	43
40	2	14
41	0	6
42	1	22
43	5	61
44	3	26
45	4	69
46	8	21
47	7	49
48	9	72
49	6	53
50	2	84
51	1	2
52	5	52
53	3	95
54	8	48
55	9	72
56	0	47
57	6	65

58	4	6
59	7	25
60	1	46
61	0	37
62	3	61
63	2	13
64	6	32
65	5	21
66	9	32
67	8	89
68	7	30
69	4	55
70	2	31
71	0	86
72	1	46
73	5	74
74	4	32
75	6	88
76	8	19
77	9	48
78	7	36
79	3	79
80	0	76
81	1	69
82	3	76
83	5	51
84	2	85
85	9	11
86	6	40
87	7	89
88	4	26
89	8	74
90	1	85
91	0	13
92	2	61
93	6	7
94	8	64
95	9	76
96	5	47
97	3	52
98	4	90
99	7	45

C.3 Problem 3 : the problem used by Bagchi *et al.*

jobs:

part	quantity
0	40
1	20
0	10
1	20
2	15
1	50
2	40
0	35
1	35
2	80
0	120

plans:

part	operations
0	0, 3, 5
0	0, 4, 5
0	3, 8, 5
1	0, 1, 6
1	6, 8, 9
2	0, 2, 7
2	3, 9

operations:

operation	machine	duration
0	0	40
1	0	7
1	1	10
2	1	20
3	0	12
4	1	30
4	2	5
5	1	16
5	2	8
6	1	15
6	2	12
7	0	5
7	2	8
8	0	4
8	1	6
9	0	8
9	2	3