Basic Research in Computer Science

BRICS DS-01-10 M. T. Jensen: Robust and Flexible Scheduling with Evolutionary Computation

Robust and Flexible Scheduling with Evolutionary Computation

Mikkel T. Jensen

BRICS Dissertation Series **DS-01-10**

ISSN 1396-7002 **November 2001**

See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/01/10/`

Robust and Flexible Scheduling with Evolutionary Computation

Mikkel T. Jensen

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Robust and Flexible Scheduling with Evolutionary Computation

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Mikkel T. Jensen
4th October 2001

Abstract

Over the last ten years, there have been numerous applications of evolutionary algorithms to a variety of scheduling problems. Like most other research on heuristic scheduling, the primary aim of the research has been on deterministic formulations of the problems. This is in contrast to real world scheduling problems which are usually not deterministic. Usually at the time the schedule is made some information about the problem and processing environment is available, but this information is uncertain and likely to change during schedule execution. Changes frequently encountered in scheduling environments include machine breakdowns, uncertain processing times, workers getting sick, materials being delayed and the appearance of new jobs. These possible environmental changes mean that a schedule which was optimal for the information available at the time of scheduling can end up being highly suboptimal when it is implemented and subjected to the uncertainty of the real world. For this reason it is very important to find methods capable of creating *robust* schedules (schedules expected to perform well after a minimal amount of modification when the environment changes) or *flexible* schedules (schedules expected to perform well after some degree of modification when the environment changes).

This thesis presents two fundamentally different approaches for scheduling job shops facing machine breakdowns. The first method is called *neighbourhood based robustness* and is based on an idea of minimising the cost of a neighbourhood of schedules. The scheduling algorithm attempts to find a small set of schedules with an acceptable level of performance. The approach is demonstrated to significantly improve the robustness and flexibility of the schedules while at the same time producing schedules with a low implementation cost if no breakdown occurs. The method is compared to a state of the art method for stochastic scheduling and concluded to have the same level of performance, but a wider area of applicability. The current implementation of the method is based on an evolutionary algorithm, but since the real contribution of the method is a new performance measure, other implementations could be based on tabu search, simulated annealing or other powerful "blind" optimisation heuristics.

The other method for stochastic scheduling uses the idea of coevolution to

create schedules with a guaranteed worst case performance for a known set of scenarios. The method is demonstrated to improve worst case performance of the schedules when compared to ordinary scheduling; it substantially reduces program running times when compared to a more standard approach explicitly considering all scenarios. Schedules based on worst case performance measures often have suboptimal performance if no disruption happens, so the coevolutionary algorithm is combined with a multi-objective algorithm which optimises worst case performance as well as performance without disruptions.

The coevolutionary worst case algorithm is also combined with another algorithm to create schedules with a guaranteed level of *worst deviation performance*. In worst deviation performance the objective is to minimise the highest possible performance difference from the schedule optimal for the scenario that actually takes place. Minimising this kind of performance measure involves solving a large number of related scheduling problems in one run, so a new evolutionary algorithm for this kind of problem is suggested.

Other contributions of the thesis include a new coevolutionary algorithm for minimax problems. The new algorithm is capable of solving problems with an asymmetric property that causes previously published algorithms to fail. Also, a new algorithm to solve the economic lot and delivery scheduling problem is presented. The new algorithm is guaranteed to solve the problem to optimality in polynomial time, something previously published algorithms have not been able to do.

Acknowledgements

I would like to thank Brian Mayoh for being my supervisor. I thank Moutaz Khouja and Tage Kiilsholm Hansen for their cooperation. In no particular order I acknowledge helpful comments from and discussions with Rasmus Ursem, Thiemo Krink, Jakob Vogdrup Hansen, Søren Glud, Zbigniew Michalewicz, Jan Pedersen, an anonymous referee at IEEE Transactions on Evolutionary Computation, Martin Schmidt, Thomas Stidsen, Jürgen Branke, Martin Middendorf and Dirk Mattfeld. Thanks to Morten Peter Jensen, Bente Lynge Hansen and Jakob Vogdrup Hansen for proofreading this work. A special thank you goes to my parents, Morten, Anne and Jakob for always supporting me and believing in me.

Mikkel T. Jensen,
Århus, 4th October 2001

Contents

Chapter 1

Motivation and Introduction

Over the last ten years there has been a surge of research activity in the field of Evolutionary Computation. There have been advances in many areas, and evolutionary computation has been applied to quite a few real world problems. However, most of the research in the field has been focused on various aspects of optimisation and problem solving in static forms. Most of the optimisation problems treated - both artificial benchmarks and real world problems - have been viewed as static and deterministic problem instances. It has been assumed that complete knowledge of the problem was available when it was to be solved. There would be no changes to the problem or in the environment later, which would require the solution to be modified or updated. In short most of the research has largely ignored the fact that often optimisation is a process of sustained pursuit, trying to hold on to an optimum solution which changes over time.

These shortcomings have also been present in the area of scheduling. This may seem surprising, since in scheduling it seems very obvious that as the environment changes the solution may need adaptation. Most scheduling problems in the real world face the possibility of disruptive events such as machines breaking down, workers getting sick or new jobs appearing. When and how these events will occur is not known beforehand, but the possibility that they may happen is known. Since many real world scheduling applications involve huge amounts of money, even a slight improvement of efficiency when handling these events can translate into substantial savings and a competitive edge for the company involved.

In most scheduling problems, the solution found is not implemented all at once, but in a step by step manner. A consequence of the noise and uncertainty present in the real world is that the occurrence of some breakdown or failure part way through the implementation of the schedule can call for a change to the schedule. The process of changing a schedule to accommodate a change in the environment is usually called *rescheduling*. Since the presence of noise and uncertainty is known at the time of scheduling, it seems natural to take possible

future events into consideration before they happen, in order to create schedules that allow adaptation to changes. In this way, stochastic scheduling can be seen as a task in which the objective is to create low cost schedules, but at the same time avoiding painting oneself into a corner by making decisions that do not allow changes to be made to the schedule in the future. Depending on the way the schedules are adapted to changes, schedules prepared for uncertain future events are called *robust* or *flexible*.

The measurement of schedule robustness or flexibility can be done in a variety of ways. The simplest way of measuring schedule performance is by implementing it in the scheduling environment, and noting what the cost of implementing the schedule turned out to be. This ex post evaluation is of course useless when it comes to choosing the best schedule, so a more clever kind of evaluation is needed. More clever ways of measuring schedule performance usually involve some kind of simulation or reasoning about the schedule and the possible future events. Knowledge about the kind of events to be expected is needed, since if no knowledge is available, there is no way of reasoning about the possible future outcomes if the schedule is implemented. The formulation of schedule quality in mathematical terms can be done in a number of different ways. In some situations, a good measure of quality will be the average cost of implementing the schedule (*average performance*). For more critical applications *worst case performance*, in which the costs of the worst possible conditions are to be minimised, are more relevant. Other ways of measuring schedule performance are *worst deviation performance* and *worst relative deviation performance*, both of which try to minimise the highest cost difference between the implemented schedule and the optimal schedule for all scenarios possible.

How to optimise the stochastic performance of a schedule depends on the way schedule quality is measured. If average performance is used, the probabilities of future scenarios need to be known, since without them it is impossible to evaluate schedule performance. If worst case performance is used, knowledge of scenario probabilities is not needed, only the set of possible future scenarios needs to be known. Worst deviation and relative worst deviation performance do not require knowledge of scenario probabilities either, but they require knowledge of the optimal schedules for all possible scenarios. Finding these optimal schedules for all of the scenarios is a very difficult task, since it requires the solution of a huge number of scheduling problems, each of which will usually be NP-complete.

Very few generally applicable methods for creating robust or flexible schedules exist today. The main topic of the present thesis is the development of novel algorithms and methods to create job shop schedules which are flexible or robust when measured using the performance measures listed above. Many of the techniques used have previously been applied successfully to static scheduling problems, and are primarily from the field of evolutionary computation.

A problem arising when working with worst case, worst deviation and relative worst deviation performance measures is the *minimax* problem. Besides scheduling, the minimax problem is relevant to research areas such as mechanical engineering, constrained optimisation, network design and function approximation. The problem is an optimisation problem that can be viewed as an antagonistic game between to players. One player tries to minimise a cost, while the other tries to maximise it. Since the search-spaces of the two players can be huge, a coevolutionary approach in which each player is represented by a population of choices seems a promising approach. Previously published coevolutionary algorithms for minimax problems require the problems to have a certain symmetric property. In this thesis a new algorithm which removes the need for this symmetric property is presented.

The *economic lot delivery and scheduling problem* (ELDSP) is a problem in which a supplier produces a number of products which are stored in an inventory and shipped to a single costumer in batches. This situation often arises in e.g. the automotive industry. The objective of the ELDSP is to minimise the combined costs of production, inventory and shipping. Prior to the development of the algorithm presented in this thesis, no polynomial time algorithm was known to optimally solve this problem.

1.1 List of publications

Most of the results presented in this thesis have been published (or are about to be published) in the following papers:

- M. T. Jensen and T. K. Hansen: Robust Solutions to Job Shop Problems, In P. J. Angeline et al. , editors, *Proceedings of the 1999 Congress on Evolutionary Computation*, volume 2, pages 1138-1144, 1999.

- M. T. Jensen: Neighbourhood based Robustness Applied to Tardiness and Total Flowtime Job Shops, In M. Schoenauer et al. , editors, *Parallel Problem Solving from Nature - PPSN VI proceedings*, pages 283-292, Springer LNCS volume 1917, 2000.

- M. T. Jensen: Improving Robustness and Flexibility of Tardiness and Total Flowtime Job Shops using Robustness Measures, *Journal of Applied Soft Computing*, volume 1, number 1, 2001.

- M. T. Jensen: Finding Worst-Case Flexible Schedules using Coevolution, In L. Spector et al. , editors, *GECCO 2001 - Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1144-1151, 2001.

- M. T. Jensen: A New Look at Solving Minimax Problems with Coevolution, presented at MIC2001, submitted to the conference book.
- M. T. Jensen: Generating Robust and Flexible Job Shop Schedules using Genetic Algorithms. In submission.
- M. T. Jensen and M. Khouja, An Optimal Polynomial Time Algorithm for the Common Cycle Economic Lot and Delivery Scheduling Problem. In submission.

1.2 How to read this thesis

Care has been taken to make this thesis as self-contained as possible. Because of this, there is a fair amount of pages concerned with the introduction of concepts and problems such as evolutionary computation, deterministic and stochastic scheduling. The reader already familiar with these concepts can skip these parts of the thesis and go directly to the more interesting parts. The original work in this thesis is contained in section 2.4 of chapter 2, sections 4.3, 4.5.1 and 4.6 of chapter 4 and all of chapters 5, 6 and 7.

The outline of the thesis is the following:

Evolutionary algorithms are introduced in chapter 2. The chapter introduces the algorithmic ideas used throughout the thesis, most notably genetic algorithms and evolutionary strategies, multi-objective optimisation and coevolution. The most important section of the chapter from a research point of view is section 2.4, since this section describes and tests the new coevolutionary algorithm for minimax optimisation.

Chapter 3 introduces the static formulation of the job shop problem, and introduces the disjunctive graph formulation of job shop problems and schedules. Mattfeld's genetic algorithm for static job shop scheduling (which is used as a starting point for some of the algorithms later in the thesis) is described in detail, along with a brief summary of other static scheduling methods.

Basic concepts and definitions used in stochastic scheduling are presented in chapter 4. The chapter defines the fundamental concepts of robustness and flexibility as well as the stochastic performance measures. The types of scenarios and events usually considered in stochastic scheduling are presented, and the algorithmic complexity of various problems in stochastic scheduling is discussed. The chapter gives a summary of the most interesting approaches for stochastic scheduling published so far.

Chapter 5 presents the idea of neighbourhood based robustness for job shop scheduling. First, the neighbourhood based robustness measures are defined, and a genetic algorithm for minimising the makespan based robustness measure is

presented. The schedules produced by this algorithm are compared to schedules produced by a standard static scheduling algorithm, and found to be superior to these in terms of average performance. The average performance of the schedules is also compared to schedules produced by minimising a previously published slack based robustness measure [72], and found to produce schedules of a comparable level of robustness and flexibility. The neighbourhood based robustness approach is also applied to worst tardiness, summed tardiness and total flowtime problems. The chapter ends with a discussion on how and why the neighbourhood based robustness measures work.

Worst case and worst deviation performance schedules are considered in chapter 6. The same core algorithm is used for the optimisation of both performance measures. This algorithm is based on the coevolutionary minimax algorithm developed in chapter 2. The basic idea in the approach is to use a population of future scenarios (machine breakdowns) to obtain an estimate of the performances of schedules in the schedule population. The performances of the algorithms are compared to the performances of an algorithm optimising the static performance of the schedules and an algorithm using an exact evaluation approach to optimise stochastic performance. The coevolutionary algorithms are shown to outperform the static algorithms in terms of schedule flexibility, while outperforming the exact evaluation approaches in terms of running times. In order to work with worst deviation performance, two algorithms for the simultaneous solution of a large number of related NP-complete problems are developed. One of these algorithms is implemented and tested, and found to outperform a more standard approach. However, even though the new algorithm is found to perform reasonably well, it is deemed inappropriate for use as a "front end" for the worst deviation algorithms, since the solutions found are sometimes too far from optimality.

The economic delivery scheduling problem is treated in chapter 7. The chapter starts out by defining the problem and describing a previously proposed heuristic for solving the problem, Hahm and Yano's heuristic [54]. Following this, a novel polynomial time algorithm for solving the problem is developed. The new algorithm and the heuristic are compared in a computational study.

The thesis ends with chapter 8, in which the main results of the thesis are summarised.

Chapter 2

Evolutionary Computation

This chapter is meant as an introduction to the types of algorithms used later in the thesis.

The first section introduces different evolutionary algorithms (*EAs*). The focus in the first section is on evolutionary algorithms in general. It focuses on genetic algorithms and to a smaller extent evolution strategies, since these are the algorithmic templates used later. The second section describes the use of evolutionary computation for multi-objective optimisation. Sections 2.3 and 2.4 introduce the idea of coevolution for problem solving and demonstrate a new approach to how coevolution can be used to solve minimax problems. This new approach is shown to outperform previously proposed methods on problems that do not have a symmetric property.

The presentation is biased towards the application on optimisation problems, although some of it is more general.

2.1 Evolutionary algorithms

Evolutionary Algorithms are computational models for problem solving, optimisation or simulation inspired by the evolution of species in nature and the Darwinian principles. Common for all evolutionary algorithms is that they work on one or more sets of potential solutions to a problem. Because of the biological inspiration, each solution in an EA is often termed an *individual*, while a set of individuals is called a *population*. The population is manipulated using a cycle of selection of good individuals followed by the generation of new individuals by variational operators such as mutation and recombination.

The basic idea in an evolutionary algorithm is to assign each individual a *fitness*, measuring how well it solves the problem at hand. Individuals which are known to be good or promising (have a high fitness score) are then selected for

```
set t = 0
initialise Pop(t) with random solutions
evaluate Pop(t)
while(not done) do
    set t = t + 1
    select parents from Pop(t − 1)
    generate children from parents using
       variational operators
    evaluate children
    select Pop(t) from children and Pop(t − 1)
od.
```

Figure 2.1: *A simple evolutionary algorithm.*

reproduction, and new individuals which are related to them are generated. The new individuals are inserted in the population if they are acceptable, while inferior individuals are discarded. By constantly creating variations of the best solutions known, the algorithm gradually improves the solution quality. Every iteration of this cycle is called a *generation*.

The way new individuals are generated is usually stochastic. Because of this, evolutionary algorithms are stochastic search algorithms, in which previously sampled points are used to guide the selection of future sampling points. Most evolutionary algorithms today are based on *Genetic Algorithms* (*GAs*), *Evolution Strategies* (*ESs*) and *Evolutionary Programming* (*EP*).

A simple template for an evolutionary algorithm is shown in figure 2.1. This template leaves many details undecided, which is why it can be fitted to all of the arch-typical EAs (ES, GA, and EP).

2.1.1 Evolution strategies

Evolution Strategies were developed by Rechenberg and Schwefel in the sixties. They introduced ES to solve continuous optimisation problems in engineering. A survey of evolution strategies can be found in [103].

The goal in a standard ES is to optimise a function of a number of real variables, $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Every individual is usually represented by a vector of real variables $\mathbf{v} \in V \subseteq \mathbb{R}^n$, where V denotes the search-space. Usually the ES is run for a predefined number of generations, after which the best solution is returned. In evolutionary strategies in their cleanest form, reproduction is done exclusively by mutation, which is done by adding a zero mean Gaussian distributed vector to

the vector representing the individual,

$$\mathbf{v}_{child} = \mathbf{v}_{parent} + \sigma\mathbf{z}, \quad \mathbf{z} \sim N(0, \mathbf{I}_n). \tag{2.1}$$

The severity of the mutation is governed by the parameter σ. Several schemes to adjust the mutation severity while the algorithm is running have been proposed, including adjusting σ based on the success rate of mutation, and auto-adaptive methods making details of the mutation part of the search-space. In a more advanced ES, the mutation vector $\mathbf{z}$ can be taken from distributions in which the mutations of the coordinates are correlated, that is $\mathbf{z} \sim N(0, \mathbf{C})$, where the covariance matrix $\mathbf{C}$ is not necessarily diagonal.

The breeding and replacement strategy of an evolution strategy is usually described in the $(\mu \dot{+} \lambda)$-notation. The population size is μ while the number of offspring per generation is λ. An ES in which the next population is taken as the best individuals among the μ parents and the λ offspring is denoted a $(\mu + \lambda)$-ES. If the μ parents are discarded and the new population is taken from the offspring exclusively, it is termed a (μ, λ)-ES.

A number of theoretical results exist on the properties of evolution strategies as optimisers, a brief overview of these are given in [104].

2.1.2 Genetic algorithms

Genetic algorithms were introduced by Holland [60] in the mid-seventies. Holland was inspired by the principles of genetics, so genetic algorithms place a stronger emphasis on the distinction between the genetic representation of an individual (the *genotype*) and the actual expression of the individual (the *phenotype*) than evolution strategies. Good introductions to genetic algorithms can be found in [80, 81].

The principal variational operator is *recombination*, also termed *crossover*: the combination of two or more parents to produce offspring. This combination is usually performed by taking part of the genotype of each parent, and combining the two parts to obtain a new genotype sharing characteristics of both parents. Mutation operators are also used in genetic algorithms, but mostly to make sure genetic material lost early in the search process can be reintroduced later. This is necessary since usually crossover cannot introduce new genetic material; it merely recombines material already present in the population. Thus, without a mutation operator genetic material not present in the population can never appear.

The heavy use of recombination in genetic algorithms is based on what is known as *the building block hypothesis*. It is assumed that it is possible to take (approximately) half of one genotype, half of another genotype, and produce a

child that shares characteristics of both parents. In this way, if two different individuals are good, crossover can take the good parts from each of them and combine them to form an even better individual. This assumes the two good genotype parts to be compatible. Since there is no way of knowing which part of each genotype is good and which is bad, the combination of parents can only take place in a random fashion, thus the recombination of two good parents can also lead to the combination of two bad parts of the genotype.

Genetic algorithms in their original form use genetic representations based on strings. Suppose binary strings of length L are used, meaning that each individual is a string of length L over the alphabet "`0`" and "`1`". The effect of the crossover operator can be analysed using *schemata*. A schema is a string of length L made from the alphabet "`0`", "`1`" and "`*`". A genetic string is said to *instantiate* or *match* a schema if the schema and the genetic string have the same characters in all string positions in which the schema holds a "`0`" or a "`1`", while the asterisks work as "match all" symbols. For instance, the string "`010101`" instantiates the schema "`01**0*`", as well as many others. Since the search-space of genetic strings can be visualised as an L-dimensional hypercube, a schema can be seen as a hyperplane. All genetic strings matching the schema will be on one side of the hyperplane, while all strings not matching it will be on the other side.

Using the schema formulation, a GA can be said to start with a wide range of different schemata present in the population. A schema is said to be present in the population if it is matched by any of the individuals present in the population. Because of the fitness based selection cycle, schemata representing good building blocks will become more and more predominant in the population as the search progresses. If the recombination operator is *respectful* (meaning that traits shared by the parents will be present in the children, i.e. if the parents both instantiate a schema, so will the children), the building blocks present in the population will gradually combine to form larger and larger building blocks. Another way of seeing this process is to realize that a randomly initialised population of individuals corresponds to a number of schemata much larger than the number of individuals. Evaluating the fitness of the population corresponds to estimating the average fitness of individuals matching these schemata (estimate based on the samples matching the schemata present in the population). High average fitness schemata will tend to have more offspring than those with a low average fitness. As the high average fitness schemata become prevalent in the population, the probability that they are combined by recombination to form even higher average fitness schemata increases.

The role of selection in a genetic algorithm and the implicit estimation of schema fitness have been formally analysed under a number of assumptions. Denote by $n(H, t)$ the number of individuals in the population $P(t)$ matching schema H at generation t. If fitness proportional selection is used, and ignoring the ef-

fects of crossover and mutation, the expected number of individuals matching H at generation $t+1$ should be

$$E(n(H,t+1)) = \sum_{i \in P(t) \cap H} \frac{f(i)}{\overline{f}(t)},$$

where $P(t) \cap H$ denotes the individuals in $P(t)$ matching H, $f(i)$ denotes the fitness of i and $\overline{f}(t)$ denotes the average fitness of the population at time t. This can be rewritten

$$E(n(H,t+1)) = \frac{u(H,t)}{\overline{f}(t)} n(H,t), \quad u(H,t) = \sum_{i \in P(t) \cap H} \frac{f(i)}{n(H,t)},$$

where $u(H,t)$ is the average fitness of individuals matching H at time t.

If we denote by $D_c(H)$ and $D_m(H)$ the probability that an individual matching H at generation t will be disrupted by crossover or mutation and not match H at generation $t+1$, and assume crossover and mutation to work independently of each other, a lower bound on $E(n(H,t+1))$ is

$$E(n(H,t+1)) \geq \frac{u(H,t)}{\overline{f}(t)} n(H,t)(1 - D_c(H))(1 - D_m(H)), \qquad (2.2)$$

where we are ignoring the beneficial effects of crossover and mutation. The disruption probabilities $D_c(H)$ and $D_m(H)$ depend on the details of the operators used, but for the classical choice of one-point crossover, $D_c(H)$ will increase with the *defining length* of H, the longest distance between two non "*" symbols in H. Assuming the mutation to mutate the individual bits with equal probability, $D_m(H)$ will increase with the *order* of H, the number of non "*" symbols in H. Equation (2.2) is known as *the schema theorem*. An often seen interpretation of the theorem is that the number of schemata for which D_c and D_m are low and which will keep on having an above average fitness will increase exponentially in the population increasing their numbers by a factor of $\frac{u(H,t)}{\overline{f}(t)}$ for every generation (up to a certain point). However, this is no guarantee that a GA used for optimisation will find a good solution. First of all there is no guarantee that the optimum solution can be decomposed into a number of schemata which are of low order, have a short defining length and a high average fitness. For a misleading problem the optimum solution could be only decomposable to low order schemata having a low average fitness. Also, even if the optimum solution can be decomposed into short above average fitness schemata, the fitness of these schemata is implicitly estimated based on the individuals in the population, meaning that estimates may deviate significantly from the "real" fitness of the schemata. More in-depth discussions of the schema theorem as well as other theoretical approaches to evolutionary computation can be found in [8, 81].

2.1.3 Epistasis

As stated above, the prime beneficial effect of the use of the recombination operator is the assembly of large above average fitness building blocks from smaller ones. This is only possible if the fitness contribution of building blocks are not affected too much by the presence of other building blocks in the genotype. A problem or problem representation in which the phenotypical effect of the presence of one building block depends on the presence of other building blocks is called *epistatic*. The presence of epistasis is known to be able to cause serious trouble to GAs [12, 85]. The degree of epistasis of a problem is dependent on the choice of genetic representation; an optimisation problem which is highly epistatic for one genetic representation may be low-epistatic in another, [12]. For these reasons, the choice of representation is crucial for the success of a GA.

Another reason why the choice of representation is crucial in the design of an evolutionary algorithm is that the size of the search-space is determined by the representation. In some cases infeasible solutions can be excluded from the search-space by a clever representation, greatly reducing the size of the search-space. A prime example of this is the job shop scheduling problem, see section 3.2.1.

The application of genetic algorithms on e.g. combinatorial optimisation problems has led to the use of genetic representations which are not based on simple strings in which phenotypic traits are coded by independent genes. It has been found that for some problems the use of a recombination operator does not improve the optimisation performance [52]. In many cases this is due to epistatic effects; if the recombination of two individuals produces an individual which has nothing in common with the parents, the use of recombination has degenerated to random search.

2.1.4 Applying evolutionary algorithms

When an optimisation problem is to be solved with an evolutionary algorithm, many choices have to be made. The choice of genetic representation is usually the first and probably most important single decision, but many other decisions also profoundly affect the effectiveness of the algorithm.

When choosing a set of genetic operators (mutation, crossover) the roles of the different operators should be considered. A mutation operator should be strong enough to ensure sufficient genetic diversity in the population, but at the same time weak enough to support local search and small improvements in a hillclimbing like manner. Crossover operators should be respectful [97], that is able to combine building blocks to form larger building blocks which share the phenotypical traits of the smaller building blocks. On the other hand, it has sometimes

been found that crossover operators which are not respectful can perform better than respectful ones, [77].

Once a suitable overall structure for the evolutionary algorithm has been found (ES, GA), and genetic representation and operators have been decided, there are still many open issues. The following is a short list of issues to be decided every time an evolutionary algorithm is designed:

- Selection. How do you select the individuals to be used for reproduction? A number of different selection methods like proportional selection, tournament selection and rank based selection [9] have been proposed.

- Replacement. How are the new individuals incorporated into the population after reproduction? In a simple generational genetic algorithm often the offspring simply replaces the parents, but sometimes it is helpful to keep some of the parents in the population. This is usually done by *elitism*: making sure a certain number of the best individuals never leave the population. Another option is having a *steady state* algorithm, in which only a very small part of the population is replaced in every generation.

- Crossover and mutation rate. When and how much are the different genetic operators to be used? Often fixed values are used for these parameters, but what should these values be? Other choices include making these parameters adaptive or auto-adaptive.

- Population size, number of generations. Generally, the larger these parameters are, the better the algorithm will perform, but at the expense of longer run-times, since more fitness evaluations will be involved. For a fixed allowed number of fitness evaluations, the choice of these parameters becomes a tradeoff. A large population size means a better exploration of the search-space, while a large number of generations allows for better exploitation of the promising solutions found.

Besides the above mentioned parameters, a number of other possibilities are open when applying an evolutionary algorithm. It is often found that a genetic algorithm gets caught in a local optimum, and that all or most of the population concentrates on a small part of the search space located around the local optimum. This is usually termed *premature convergence*. Often this problem is solved using *structured populations*: by dividing the population into smaller subparts that tend to breed more amongst themselves, the dispersion of genetic material is slowed down, giving the algorithm more time to settle on the most promising part of the search-space. The two most prominent kinds of structured populations are *island models* [76] and *diffusion models* [93]. Other ways to accomplish a better

exploration of the search-space are niching methods such as *sharing*, *crowding* [75], *tagging* [38] and *multinational* genetic algorithms [110].

It has been found in many studies that evolutionary algorithms benefit from hybridisation with other methods. Generally, the performance of evolutionary algorithms can be improved considerably by applying problem specific knowledge in the EA. This is often done by using problem specific hillclimbers in decoding procedures or to improve the solutions found by the EA. Another possibility is the use of problem specific knowledge in the design of genetic operators [74].

Since there is no theory adequately describing how genetic algorithms work, some of these decisions are largely up to experience and taste of the person designing the algorithm. Due to the large number of design decisions, the interdependence between these and the lack of mathematical understanding of both the evolutionary algorithms and the search-spaces of the problems they are applied to, it is usually infeasible to find the optimal set of parameters for a given problem. This is particularly so since the optimal set of parameters can also be expected to vary from problem instance to problem instance. Usually the parameters are fixed in a trial and error fashion in which a number of possibilities are tried out, and the one observed to work best is used.

2.1.5 Advantages of evolutionary computation

Evolutionary algorithms have been applied to a large number of academic and real world problems over the last fifteen years. These applications have proven the worth of evolutionary algorithms, particularly for problems which are intractable for more traditional methods (NP-hard problems). Since EC can be used as a blind heuristic search method to most problems, it can almost always be applied. The fact that EAs can at the same time be hybridised with problem specific techniques such as hillclimbers for improved performance makes them very flexible tools, since usually a "quick and dirty" EA can be implemented. Later more effort can be put into this algorithm in the form of problem specific knowledge for improved performance. Other modern search methods have these properties as well; Simulated Annealing and Tabu Search [99] are both generally applicable optimisation methods, and have also been demonstrated to perform really well in many applications.

However, for some types of problems EAs have at least two huge advantages over simulated annealing, tabu search and most other methods.

- There is increasing interest in applying contemporary search heuristics to dynamic problems. Simulated annealing and tabu search both work on a single solution which is constantly improved. In a dynamic environment which is constantly changing, an EA has the advantage of a population of

solutions. If an environment change makes the current best solution unacceptable, chances are there is another solution in the population with a better performance. To cite an old cliché, when using an EA in a dynamic environment, one may avoid "having all of ones eggs in one basket".

- Many optimisation problems in the real world are multi-objective problems. What is sought is not a solution optimising one single objective, but rather a solution which is a compromise between many different objectives. Often it is desirable to find not just one solution, but a number of different non-dominated solutions, from which a human expert can make a choice. In this situation, EAs have the advantage of already working with a set of solutions, while simulated annealing and tabu search both work on one solution.

2.2 Multi-objective evolutionary algorithms

Standard instances of evolutionary algorithms are designed for optimising one performance measure only. For many optimisation problems this is insufficient, since real world problems often involve tradeoffs between different criteria. As an example, consider the problem of designing a communication network. At least three different objectives are relevant to such a problem: 1) Communication capacity. Between any two points in the network the bandwidth should be as high as possible (or above some threshold value). 2) Cost. The investment of building the network should be as low as possible. 3) Robustness. The network should be robust to failures; even if a connection or node fails, the rest of the network should stay connected and functional to some degree.

Given a number of objectives instead of just one, the ordering of two different solutions is not always straightforward. If network s_1 has lower cost than network s_2, but s_2 has better capacity, which one is preferable? Usually multi-objective solutions are ranked based on the Pareto domination criterion: Given an optimisation problem of n objectives $p_1, p_2, \ldots, p_n$ to be minimised, solution s_1 is said to *Pareto dominate* solution s_2 if

$$\forall i \in \{1 \ldots n\} : p_i(s_1) \leq p_i(s_2) \quad \wedge \quad \exists j \in \{1 \ldots n\} : p_j(s_1) < p_j(s_2).$$

The set of solutions sought by a multi-objective optimisation algorithm is known as the *Pareto optimal front* or *Pareto optimal set*. The Pareto optimal front is the set of solutions in the search-space which are not dominated by any other solutions. If the Pareto optimal set is infinite or very large, what is sought is a number of solutions with a reasonable spread, such that a large number of diverse solutions is returned by the algorithm.

A number of different approaches for multi-objective evolutionary algorithms (*MOEA*s) exist of which the most prominent are probably aggregation methods and Pareto based methods. For surveys of MOEA algorithms see [29, 34].

The most straight-forward approach is to use aggregation of the n objectives to one scalar objective by generating a fitness measure as a weighted sum of the objectives $F = w_1 p_1 + \ldots + w_n p_n$. Multiple points on the Pareto front can be found by running the algorithm with different values for the weights w_i. Two fundamental problems arise with this approach. Due to the definition of F, algorithms of this kind are unable to find non-convex parts of the Pareto optimal set. Furthermore, one run of the algorithm will only return one solution on the Pareto front. Examples of this approach can be found in [21, 66, 72].

Contemporary MOEAs use selection and replacement based on the multi-objective domination criterion defined above. Examples of this approach are Fonseca and Fleming's *MOGA* [44], Horn et al.'s *NPGA* [61], Corne, et al. 's *PESA* [31], Zitzler and Thiele's *SPEA* [116] and Deb et al. 's *NSGA-II* [35, 36]. All of these algorithms use niching (usually a variant of sharing or niche counting) to ensure that a diverse Pareto set is found, and all except one (the NPGA) use elite methods or an external storage to keep the best individuals found so far. There has been no comparative study to reveal which of these algorithms generally performs best, but recent studies, [31, 36] indicate that the NSGA-II or PESA algorithms are probably the best overall performers. In the following, a description of the NSGA-II will be given.

2.2.1 The non-dominated sorting GA, NSGA-II

The NSGA-II [35, 36] is an extension of the NSGA algorithm, which was also published by Deb [34]. The algorithm is based on an idea of transforming the n objectives to a single fitness measure by the creation of a number of fronts, sorted according to non-domination. During fitness assignment, the first front is created as the set of solutions not dominated by any solutions in the population. These solutions are given the highest fitness and are temporarily removed from the population. After this, a second non-dominated front consisting of the solutions which are now non-dominated is built, assigned the second-highest fitness etc. This is repeated until all of the solutions have been assigned a fitness. After each front has been created, its members are assigned density estimates (distance to closest neighbours in the front in objective space, as illustrated in figure 2.2) later to be used for niching.

Selection in the algorithm is performed in size two tournaments: The solution with the lowest front number wins. If the solutions come from the same front, the solution with the highest distance to the closest neighbours wins. Breeding occurs in generations. In each generation N new individuals are bred, where N

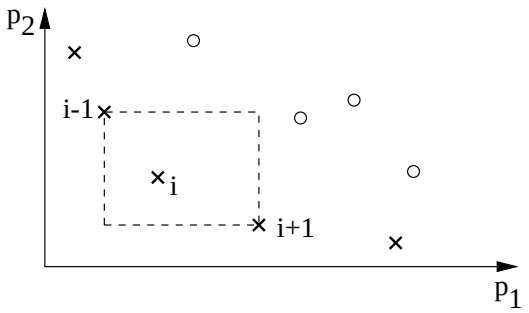

Figure 2.2: *Niching in the NSGA-II is done by assigning each solution a niche measure, $I_{distance}$, which is its distance to the closest neighbours in its front, measured along the objective axis. In this example, solution i is assigned the distance $I[i]_{distance} = |p_1(i-1) - p_1(i+1)| + |p_2(i-1) - p_2(i+1)|$. Solutions with an extremal position in one of the objectives are assigned the niching distance ∞.*

is the population size. Of the $2N$ individuals, the N best individuals are kept for the next generation. In this way a huge elite can be kept from generation to generation.

The overall structure of the NSGA-II algorithm can be seen in figure 2.3. The `Non-dominated-sort` procedure sorts the population into a number of fronts as described above. This can be done in $O(MN^2)$ time, where M is the number of objectives and N is the population size. The overall running time is $O(GMN^2)$, where G is the total number of generations. This is a high running time, since most traditional evolutionary algorithms have running times proportional to N, not N^2. This makes the application of the NSGA-II with large populations slow. Most other contemporary MOEAs including PESA, MOGA and SPEA also have running times proportional to N^2, since they require every member of the population to be compared to every other member at every generation step. For the scheduling problems solved using NSGA-II later in this thesis (see chapter 6) the N^2 running time is not a real concern, since the time spent calculating the objectives is far greater than the time spent sorting the population.

In a recent study, [36], the NSGA-II has been demonstrated to be superior to the SPEA algorithm and another MOEA algorithm. Furthermore, it has been shown to perform well on a number of realistic mechanical design problems [37].

2.3 Coevolutionary algorithms

The evolutionary algorithms described in the previous sections mostly dealt with optimisation on a well-defined static fitness landscape. Ideally, individuals gradu-

```
Generate P_0 at random.
P_0 = (F_1, F_2, ...) =Non-dominated-sort(P_0)
for all F_i ∈ P_0
    Crowding-distance-assign(F_i)
t = 0
while(not done) do
    Generate child population Q_t from P_t
    R_t = P_t ∪ Q_t
    F = (F_1, F_2, ...) =Non-dominated-sort(R_t)
    P_{t+1} = ∅
    i = 1
    while |P_{t+1}| < N do
        crowding-distance-assign(F_i)
        P_{t+1} = P_{t+1} ∪ F_i
        i = i + 1
    od
    sort(P_{t+1})
    P_{t+1} = P_{t+1}[0 : N]
    t = t + 1
od
```

Figure 2.3: *The NSGA-II algorithm.*

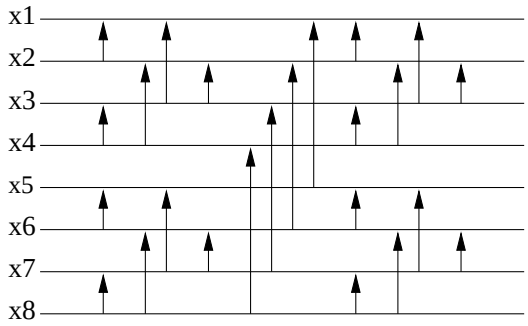

Figure 2.4: *Example sorting network. This network takes eight inputs* $(x1, \ldots, x8)$*, which are fed into the network along the lines on the left. The network is to be read from left to right. Each vertical arrow represents a comparison and possible swap of values; at the position of the arrow, the two values are compared, and if they are in the wrong order they are swapped. If the network is correct, for every possible input the values coming out on the right of the network will be ordered according to magnitude.*

ally approach the fitness optimum, and once the population is concentrated there, evolution comes to a standstill; no further improvement is possible. This is not the case in most biological evolutionary systems, where the fitness of one individual or species is closely coupled to other individuals or species by interactions. If lions evolve sharper claws this directly affects the fitness of the zebra, since this gives the lions an advantage when they are hunting the zebra. The zebra will be pressed to develop thicker skin, better camouflage or another suitable answer to the increased threat of lions. In the same way, when the zebra evolves a new trait to avoid getting eating by lions, the lions are pushed to find a suitable countermeasure. This coupling of fitnesses between individuals is known to lead to coevolutionary arms races in which two or more interacting species try to outdo each other. These arms races may go on forever or gradually approach some stable optimum.

Coevolution has been applied a number of times in optimisation and artificial life contexts. One of the first coevolutionary algorithms published was the pioneering work of Hillis on sorting networks [59], first published in 1990.

Hillis was interested in finding the minimum number of comparisons sorting network for sorting 16 numbers (see figure 2.4). Since most (random) sorting networks are not correct i.e., they will not sort all test cases correctly, the first task is to evolve networks that do the sorting correctly. For this reason the networks were assigned fitness by letting each individual sort a number of random test cases, and measuring how many outputs were correctly sorted. The networks

were not explicitly rewarded for being small, but a diploid (and fairly complex) representation made sure there was a drive towards short networks, see [59, 81] for details. It was found that the GA converged on suboptimal networks of 65 comparisons or more. This is quite far from the smallest known network, which uses 60 comparisons. Hillis found that the GA got caught in local optima because the random test cases were not challenging enough. He conceived the idea of having a population of sets of test cases coevolving with the network population. The test case population would be rewarded if the networks were unable to sort them correctly, which should make it converge on particularly difficult instances. This kind of fitness interaction is often termed *host-parasite* interaction or *predator-prey* interaction, since the success of one species adversely affects the success of the other. In this example the test case instances can be seen as parasites exploiting the weaknesses of the networks, the hosts. After implementing the coevolutionary algorithm, Hillis found that the algorithm was no longer as prone to getting stuck in local optima. As the networks got better and better the test cases would get harder and harder, forcing the network population to keep changing. The best networks found with the coevolutionary approach had 61 comparisons, a substantial improvement over the standard GA approach.

Hillis' sorting networks used a host-parasite fitness interaction, in which the two populations were constantly competing and in which success for one species meant failure for the other. Another possibility observed in nature and also used in artificial coevolutionary systems is a symbiotic or cooperative relationship in which several individuals try to accomplish some task, and are all rewarded if they succeed.

During the 90's, a large number of applications of coevolution were published. Noteworthy areas and applications include artificial life [100, 105], games such as backgammon [95] and the prisoner's dilemma [32], constraint satisfaction [90] and design and training of artificial neural networks [82, 91]. The wide variety of problems coevolution have been applied to, the ability of coevolution to speed convergence [90, 91] and the potential of dividing problems into smaller subproblems in symbiotic coevolution indicate that coevolutionary algorithms are probably among the most promising directions of research in evolutionary algorithms. However, one must also bear in mind that the inner workings of coevolutionary algorithms are more complicated and even less understood from a theoretical viewpoint than standard evolutionary algorithms. Coevolutionary dynamics can lead to unexpected and surprising results of Red Queen effects combined with genetic drift or lack of genetic diversity leading to mediocre solutions taking over a population, see [32, 92] and the next section.

2.4 Minimax problems

A minimax problem is an optimisation problem in which the task is to find the solution $x \in X$ with the minimum worst cost F, where some problem parameter $s \in S$ is chosen by an adversary. The minimax problem is often formulated: Minimise

$$\varphi(x) = \max_{s \in S} F(x, s) \quad \text{subject to } x \in X \tag{2.3}$$

or simply: find

$$\min_{x \in X} \max_{s \in S} F(x, s). \tag{2.4}$$

The minimax problem was originally formulated by game theorists, and can be seen as an antagonist game in which each of two players has a set of options. The player trying to find the *solution*, x, tries to minimise the cost, while the player determining the *scenario*, s, tries to maximise the cost. The minimax problem can be exemplified by a mechanical engineer designing a structure, which is to withstand the possible "attacks" of e.g. the weather while deforming as little as possible. At the same time, the design may have to satisfy a set of constraints (e.g. a budget). Minimax problems are found in many different areas and are known to be relevant to research in scheduling, [58, 68], network design [68], mechanical engineering [6, 94], constrained optimisation [5, 7] and function approximation [40].

Minimax problems seem well suited for coevolutionary algorithms, since the two different search spaces can be searched with two different populations. A predator-prey interaction can take place between the two populations: the solution population (P_X) needs to find solutions which evaluate to low F values with the scenario population (P_S), which needs to find scenarios s which evaluate to high values of F with the solution population.

This approach was used by Herrmann [58] on a simple parallel machine scheduling problem, where the solution space was the assignment of tasks to machines and the scenario space was the processing time of the tasks. The objective of the algorithm was to minimise the worst case makespan of the schedule. The genetic algorithm was demonstrated to converge to the most robust schedule (the schedule with the best worst case performance) and the worst case processing time scenario, which was trivially known beforehand. Barbosa [6] used an equivalent approach to design mechanical structures facing external forces. A system coevolving design and the external forces was used to find structures with the minimal worst case deformation given the possible forces. The structures evolved were found to be similar to the known optimal designs. In another paper [7], constrained optimisation problems were solved using coevolution by transforming them to minimax problems using a Lagrangian formulation.

```
create initial populations P_X(0) and P_S(0)
set t = 0
while(t < t_max) do
   for each x ∈ P_X(t) set h[x] = max_{s∈P_S(t)} F(x,s)      (*)
   for each s ∈ P_S(t) set g[s] = min_{x∈P_X(t)} F(x,s)      (*)
   generate new population P_X(t+1) from P_X(t) basing
     selection on −h[x]
   generate new population P_S(t+1) from P_S(t) basing
     selection on g[s]
   set t = t + 1
od
for each x ∈ P_X(t) set h[x] = max_{s∈P_S(t)} F(x,s)          (*)
for each s ∈ P_S(t) set g[s] = min_{x∈P_X(t)} F(x,s)          (*)
return x_0 ∈ P_X(t) minimising h[x] and
       s_0 ∈ P_s(t) maximising g[s]
```

Figure 2.5: *Herrmann's algorithm for solving minimax problems. The last three lines were not present in Herrmann's formulation [58], since he only published the main loop of the algorithm.*

When solving minimax problems, it is often required of the problem that the solution (x^*, s^*) satisfies

$$F(x^*, s) \leq F(x^*, s^*) \leq F(x, s^*) \quad \forall x \in X, s \in S. \tag{2.5}$$

which can be shown (see appendix A) to be equivalent to

$$\min_{x \in X} \max_{s \in S} F(x, s) = \max_{s \in S} \min_{x \in X} F(x, s). \tag{2.6}$$

The problems treated in [6, 7, 58] satisfy (2.6). However, not all minimax problems satisfy this constraint. In the next sections it will be demonstrated that the algorithms proposed in [6, 7, 58] are likely to fail if (2.6) is not satisfied. The problem in the existing algorithms is demonstrated to be located in the fitness evaluation, and an algorithm with a new kind of fitness evaluation is proposed. The new algorithm is demonstrated to solve the problem.

2.4.1 Previously proposed algorithms

The algorithm proposed by Herrmann [58] is displayed in figure 2.5. The algorithm used by Barbosa [6, 7] is slightly different from the algorithm of figure 2.5, since it has sub-cycles in which the scenario population is fixed and a number of

solution generations are bred and vice versa. This means that in Barbosa's algorithm, each population takes turns, evolving on a "frozen" image of the other population. It also means that the Barbosa algorithm uses twice as many evaluations of the objective function $F(x, s)$ to evaluate the same number of individuals as Herrmann's algorithm. Barbosa's algorithm can be seen in appendix B. The fitness evaluation, which is in focus here, is identical to the one used in Herrmann's algorithm.

In terms of the fitness evaluation, the two populations are treated symmetrically in the algorithm, since both kinds of individuals get their fitness assigned based on the extremal values found when evaluating them against the other population. In the following, algorithms with this kind of fitness evaluation are termed *symmetric evaluation algorithms*. The symmetric evaluation is reasonable if the solution to the problem satisfies (2.6). If this is not the case, the two populations are not symmetrical in the search-space, and treating them symmetrically in the algorithm can cause serious problems. To realize this, consider the very simple function of x and s:

$F(x,s)$	$s=1$	$s=2$	$\max_{s\in S} F(x,s)$
$x=1$	3	2	3
$x=2$	1	4	4
$\min_{x\in X} F(x,s)$	1	2	

The function can take two values for x, represented by two rows, and two values for s, represented by two columns. In addition to the function values $F(x, s)$, the values of $\max_{s\in S} F(x, s)$ for each x and $\min_{x\in X} F(x, s)$ for each s have been printed. The function does not satisfy (2.6). Running a symmetric evaluation algorithm on this function will put an evolutionary pressure on the x population to converge to the solution $x = 1$, since this minimises $\max_{s\in S} F(x, s)$. It will also put pressure on the s population to converge to the scenario $s = 2$, since this maximises $\min_{x\in X} F(x, s)$. However, the solution $F(1, 2) = 2$ does not correspond to $\min_{x\in X} \max_{s\in S} F(x, s)$, which is $F(1, 1) = 3$.

Another way of seeing this, is to realize that when the populations P_X and P_S are diverse enough, the evolutionary pressure on P_X will be in accordance with a game in which the solution x is picked first, and the scenario s is picked afterwards to maximise the cost (this game corresponds to the $\min_{x\in X} \max_{s\in S} F(x, s)$ problem). The evolutionary pressure on P_S will be in accordance with a game in which the scenario s is picked first and the solution x is picked afterwards to minimise the cost (this game corresponds to the $\max_{s\in S} \min_{x\in X} F(x, s)$ problem). This is illustrated in figure 2.6.

The difficulty stems from the fact that for a problem not satisfying (2.6), the fitness of a scenario s cannot be found only by considering the performance of

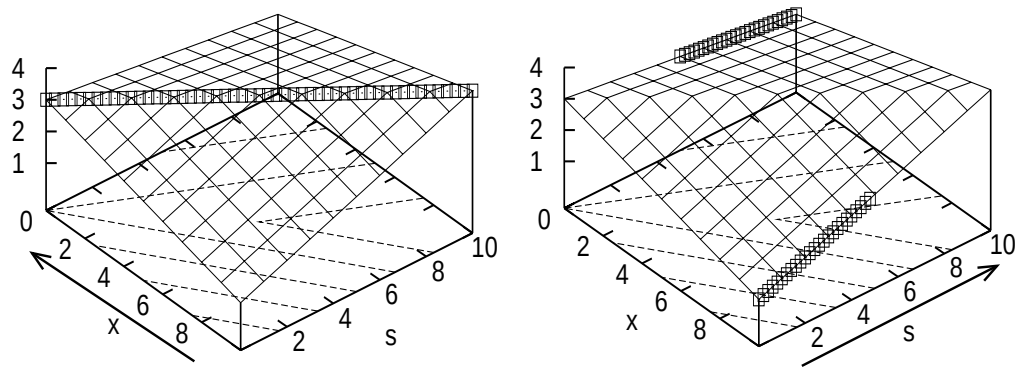

Figure 2.6: *Illustration of the significance of the picking order of x and s. The function is related to the function on page 23, and equal to the two-planes function used in the experiments.* ***Left:*** *x is picked first and s is picked afterwards to maximise the cost. The optimal choices of s have been indicated for a range of x values. Clearly the best choice of x is $x = 0$, as the arrow indicates.* ***Right:*** *s is picked first and x is picked afterwards to minimise the cost. The optimal choices of x have been indicated for a range of s values. The best choice of s is $s = 10$ as indicated by the arrow.*

s on P_X. A scenario which has a low $\min_{x \in X} F(x, s)$ value may deserve a high fitness if it causes a solution to get a high $\max_{s \in S} F(x, s)$ value. For this reason, when assigning fitness to the scenarios, it has to be taken into account how well the other scenarios perform against the solutions. If there is a solution in the population for which the scenario is the worst possible (or if it is close to being the worst possible), then the scenario should be given a high fitness.

More generally, consider a problem for which the solution and scenario corresponding to $\min_{x \in X} \max_{s \in S} F(x, s)$ is found at x_0, s_0, while the solution and scenario corresponding to $\max_{s \in S} \min_{x \in X} F(x, s)$ is located at x_1, s_1. Let us examine the behaviour of a symmetric fitness evaluation algorithm on this kind of problem.

While the scenario population P_S is diverse enough, the solution population P_X will converge to $x = x_0$, attempting to minimise $\max_{s \in P_S} F(x, s)$. When P_X converges to $x = x_0$, P_S will begin to converge to the corresponding worst case scenario $s = s_0$.

However, at the same time provided the solution population P_X is diverse enough, the scenario population P_S will try to maximise $\min_{x \in P_X} F(x, s)$ by converging to $s = s_1$. This will cause the solution population P_X to converge to the corresponding best case solution $x = x_1$.

In other words, the coevolutionary algorithm will have at least two different attractors, x_0, s_0 and x_1, s_1. If the solutions x_0, s_0 and x_1, s_1 are stable, the coevo-

```
for all x ∈ P_X(t) set h[x] = −∞
for all s ∈ P_S(t) set g[s] = −∞
for all x ∈ P_X(t) do
    for all s ∈ P_S(t) do
        set h[x] = max(h[x], F(x, s))
        set k[s] = F(x, s)
    od
    sort P_S(t) on k[s] in ascending order
    for all s ∈ P_S(t)
        if ⌊g[s]⌋ ≠ (index of s in P_S(t)) then
            set g[s] = max(g[s], index of s in P_S(t))
        else
            set g[s] = g[s] + 1/(|P_X(t)|+1)
od
```

Figure 2.7: *The asymmetric fitness evaluation for minimax problems.*

lutionary algorithm may find either one. However, there is no guarantee that this will happen, since coevolutionary dynamics can also cause the algorithm to never converge.

2.4.2 The asymmetric fitness evaluation

A fitness evaluation capable of solving the problems described above is displayed in figure 2.7. It can be inserted in the algorithm of figure 2.5, replacing the lines marked `(*)`.

The use of the $h[x]$ array is unchanged; it simply holds the worst performance found for each solution. The array $g[s]$ still holds the fitness of the scenarios, but the calculation of $g[s]$ has been changed. First, the arrays $h[x]$ and $g[s]$ are initialised.

In each step of the outer loop, a solution $x \in P_X(t)$ is tested against all scenarios in $P_S(t)$. The performance of each s against x is recorded in $k[s]$, while $h[x]$ is updated to always hold the worst performance of x found so far. After this, the scenario population $P_S(t)$ is sorted on $k[s]$ in such a way that the scenario which caused the highest $F(x, s)$ value for the solution x is located at the last position of $P_S(t)$.

At the end of the outer loop, the fitness $g[s]$ of each scenario s is set to the highest index seen so far for s. Every time s is observed to have the highest index observed so far for s in $P_S(t)$, $g[s]$ is increased slightly (by $\frac{1}{|P_X(t)|+1}$) in order to

reward scenarios which several times get the same position in the sorting of $P_S(t)$.

After the evaluation is complete the list $g[s]$ contains the best index observed of each scenario s, modified slightly for multiple occurrences of the same index. The scenario that was found to be the worst for the most solutions will have the highest possible fitness, followed by the other scenarios which were found to be the worst for at least one solution. The scenario found to be the second-worst for the highest number of solutions will be ranked immediately after these, etc.

Consider the use of this fitness evaluation on the simple problem of section 2.4.1. Assume only one of each phenotype is present in the populations. The solution $x = 1$ is tested against both scenarios, and $s = 1$ is found to be the worst, setting $g[1] = 2$ and $g[2] = 1$. $h[1]$ is set to the worst performance found, $F(1, 1) = 3$. After this, the solution $x = 2$ is tested against both scenarios, and $s = 2$ is found to be the worst, updating $g[2] = 2$, while still $g[1] = 2$. $h[2]$ is set to $F(2, 2) = 4$. After this, the solution $x = 1$ will be preferred in the reproductive step of the algorithm since $h[1] = 3 < h[2] = 4$, while neither $s = 1$ nor $s = 2$ are preferred over the other, since $g[1] = g[2] = 1$. The scenarios $s = 1$ and $s = 2$ will be assigned the same fitness as long as both $x = 1$ and $x = 2$ are present in the P_X population. Had more than one of the same kind of solution been present in P_X, the scenario which was worst to the dominant kind of solution would have been assigned the highest fitness, since the `else` part of the `if` statement would be reached.

This fitness evaluation ensures that scenarios which are very bad (relative to the other scenarios) for at least one solution will be kept in the population, while scenarios which may be “quite bad” to all solutions but not “very bad” to at least one will be removed. This is opposed to the symmetric fitness evaluation, which prefers scenarios which are “quite bad” for all solutions, but removes solutions which are “not so bad” to some solutions, even if they are “very bad” to others.

In order to reflect the change of the fitness evaluation, the return step of the algorithm in figure 2.5 should be changed to

`return the` $x_0 \in P_X(t)$ `with minimum` $h[x_0]$`, and the` $s_0 \in P_S(t)$ `which maximises` $F(x_0, s)$.

Because of the use of sorting in step 3, the asymmetric fitness evaluation needs more processing time than the symmetric evaluation used in section 2.4.1; in fact the running time of the fitness evaluation increases from $O(|P_X|\,|P_S|)$ to $O(|P_X|\,|P_S| \log |P_S|)$.

2.4.3 Experiments

A genetic algorithm based on the algorithm of section 2.4.1 was implemented both with the symmetric fitness evaluation and the asymmetric evaluation of section 2.4.2. The implementation was programmed to work on functions of real variables, using a real valued encoding. Details of this genetic algorithm were as follows:

- A linear ranking based selection scheme was used.
- A crossover rate of 0.8 was used, crossover placing the offspring at a uniformly distributed point between the parents. Offspring created by crossover were subject to mutation with probability 0.2.
- Offspring not created by crossover were made using a mutation operator adding a uniformly distributed value in the range $(-1, 1)$ to the genotype.
- A population size of 50 was used for both populations.
- The algorithm was run for 100 generations.

The crossover rate was decided by trying out the values 0.0, 0.6, 0.8 and 1.0 on the problems later in this section and picking the one that worked best. At each generation after reproduction, 10% of each population was set to completely random individuals, except in the last two generations, where this feature was disabled in order to help the populations converge. The random individuals were added since in some experiments it turned out to be necessary to keep the diversity of the populations at a high level. As the populations converged, coevolutionary dynamics would go rampant and drive both populations away from the optimum, see [92] and the experiment on the damped cosine function in this section.

The errors given in the following are *mean square errors* (MSE), defined as

$$MSE(x) = \frac{1}{n}\sum_{i=1}^{n}(x_i - x^*)^2,\ MSE(s) = \frac{1}{n}\sum_{i=1}^{n}(s_i - s^*)^2, \tag{2.7}$$

where (x_i, s_i) is the solution and scenario found in the ith experiment, and (x^*, s^*) are the optimal values.

In the problem with two optima (x_1^*, s_1^*) and (x_2^*, s_2^*), the MSE is calculated as

$$MSE(x) = \frac{1}{n}\sum_{i=1}^{n}(x_i - x_{(i)}^*)^2,\ MSE(s) = \frac{1}{n}\sum_{i=1}^{n}(s_i - s_{(i)}^*)^2, \tag{2.8}$$

where $(x_{(i)}^*, s_{(i))}^*)$ is the optimum ((x_1^*, s_1^*) or (x_2^*, s_2^*)) minimising $(x_i - x_{(i)}^*)^2 + (s_i - s_{(i)}^*)^2$.

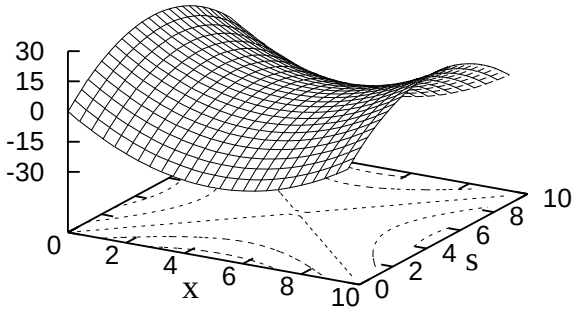

Figure 2.8: *Surface plot of the saddle-point function.*

		Symmetric algorithm		Asymmetric algorithm	
Function	Satisfies (2.6)	MSE(x)	MSE(s)	MSE(x)	MSE(s)
Saddle-point	yes	$2.15\ \mathrm{E}{-12}$	$2.04\ \mathrm{E}{-12}$	$2.05\ \mathrm{E}{-12}$	$2.03\ \mathrm{E}{-12}$
Two-plane	no	0.0000	10.987	0.0036576	0.017004
Sine	no	6.2345	37.724	0.35056	0.78844
Cosine	no	14.386	0.15923	0.037081	0.21164

Table 2.1: *Summarised results of the experiments. For each problem the mean square error (MSE) on solutions (x) and scenarios (s) for the symmetric evaluation and asymmetric evaluation algorithm are reported.*

For every function of the next subsections the $\min_{x \in X} \max_{s \in S} F(x, s)$ was solved using the symmetric evaluation algorithm and the asymmetric evaluation algorithm. For all experiments, the mean square errors were calculated based on 1000 runs of each algorithm.

A saddle-point function

The first experiment was made on a function satisfying (2.6), simply to observe that both algorithms worked on that kind of problem, and to observe any kind of difference in the performance of the two algorithms. The function was

$$F(x, s) = (x - 5)^2 - (s - 5)^2, \quad x \in [0; 10],\ s \in [0; 10] \tag{2.9}$$

A plot of the function can be seen in figure 2.8. The optimal minimax solution is known to be $x = 5$, $s = 5$. As can be seen in table 2.1, the experiments showed that both the symmetric and the asymmetric algorithm solved the problem to a very high average precision. In both cases the mean square error was found to be less than 10^{-11} on both x and s. There did not seem to be a performance difference between the two algorithms for this problem.

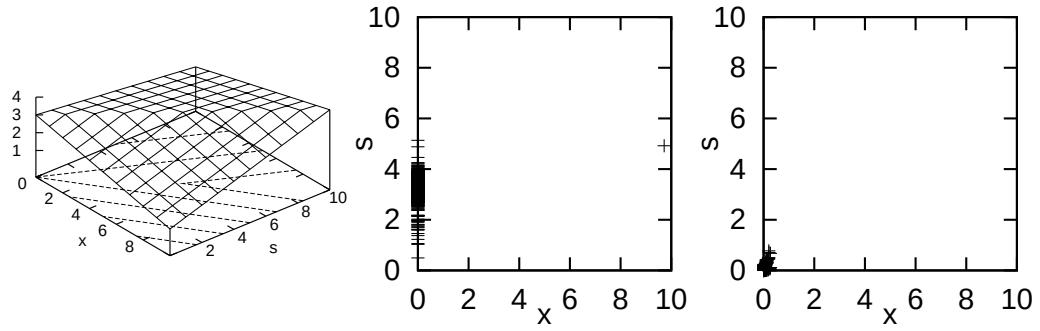

Figure 2.9: ***Left:*** *Surface plot of the two-plane function.* ***Middle:*** *The solutions found by the symmetric evaluation algorithm.* ***Right:*** *The solutions found by the asymmetric evaluation algorithm. The optimum is located at* $(0, 0)$.

A two-plane function

The function

$$F(x, s) = \min(3 - \frac{2}{10}x + \frac{3}{10}s, 3 + \frac{2}{10}x - \frac{1}{10}s),\ x \in [0; 10], s \in [0; 10] \quad (2.10)$$

has been constructed from two planes in such a way that the corners $(0, 0)$, $(10, 0)$, $(0, 10)$, $(10, 10)$ have the same values as the function from the table of section 2.4.1. As argued in section 2.4.1, the function can be expected to cause trouble for the symmetric algorithm, since there is an evolutionary pressure on the scenario population to move away from the minimax optimum at $(0, 0)$ towards $s = 10$. A surface plot of the function can be seen on the left-hand-side of figure 2.9. The first 500 solutions found by each algorithm have been plotted in the middle and right-hand parts of the figure. As can be seen, the asymmetric algorithm comes very close to the optimum every time, while the symmetric algorithm consistently finds the optimum solution but fails to find the optimum scenario. This is also evident from the mean square errors of table 2.1.

A damped sinus function

The function

$$F(x, s) = \frac{\sin(x - s)}{\sqrt{x^2 + s^2}}, x \in (0; 10], s \in (0; 10] \quad (2.11)$$

is antisymmetric around the line $x = s$. It has been designed in such a way that the solution to $\min_{x \in X} \max_{s \in S} F(x, s)$ is located at $x = 10, s = 2.125683$, while the solution to $\max_{s \in S} \min_{x \in X} F(x, s)$ is located at $x = 2.125683, s = 10$. As suggested in section 2.4.1, the problem causes serious trouble for the symmetric

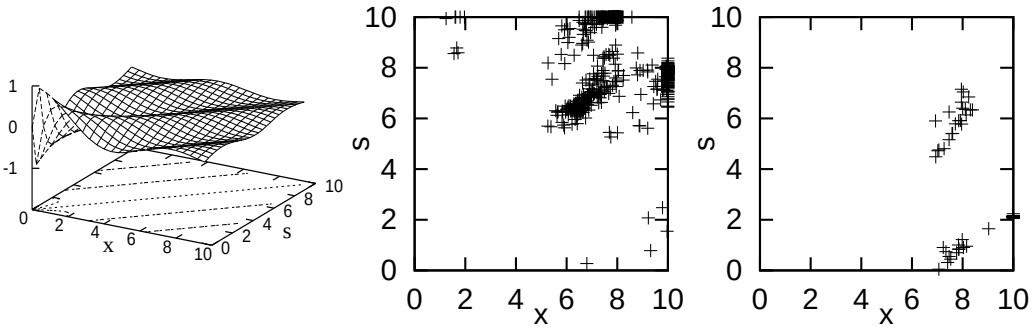

Figure 2.10: ***Left:*** *Surface plot of the antisymmetric sine function.* ***Middle:*** *The solutions found by the symmetric evaluation algorithm.* ***Right:*** *The solutions found by the asymmetric evaluation algorithm. Most of the solutions of the asymmetric algorithm are located at the optimum point* $(10, 2.125683)$.

algorithm. As can be seen from table 2.1 and the middle plot of figure 2.10, the algorithm rarely comes close to the optimum solution. The asymmetric algorithm comes close to the optimum in more than 90% of the cases (all of these runs are located at a single point of the right hand side of figure 2.10), but also fails in some cases.

A damped cosine wave

An experiment was made on the function

$$F(x,s) = \frac{\cos(\sqrt{x^2+s^2})}{\sqrt{x^2+s^2}+10},\ x \in [0;10], s \in [0;10]. \tag{2.12}$$

This function does not satisfy (2.6). A surface plot can be seen on the left hand side of figure 2.11. The function is known to have two optimal minimax solutions, one at $x = 7.04414634, s = 10$ and one at $x = 7.04414634, s = 0$.

The performance of the symmetric evaluation algorithm is displayed in table 2.1 and the middle of figure 2.11. In general the solutions found are far from the optimum, while usually an optimum scenario is found. The asymmetric evaluation algorithm is observed to perform somewhat better. The solutions are usually quite close to the optimum, while the scenarios are sometimes a bit off. The plots show two part-circle shapes, rooted in each of the two optima.

The problem is difficult to solve, since both optima are needed in the scenario population in order to keep coevolutionary dynamics from generating suboptimal solutions. Consider what happens if $x = 7.04$ and $s = 10$ are present in the populations, but $s = 0$ is not. It will be favourable for the solution $x = 7.04$ to decrease, since this minimises the observed worst cost. The observed best solution

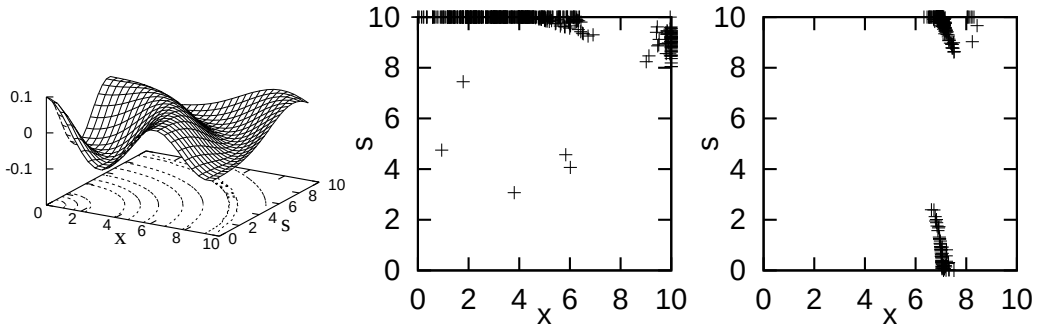

Figure 2.11: ***Left:*** *Surface plot of the cosine-wave function.* ***Middle:*** *The solutions found by the symmetric evaluation algorithm.* ***Right:*** *The solutions found by the asymmetric evaluation algorithm.*

x may continue to decrease for a while, until a suitable low value (e.g. $s = 1.0$) appears in the scenario population. In the same way, if $s = 10$ is not present in the population, the observed optimal value for x may start to increase, until a high value appears in the scenario population. These problems indicate that maintaining population diversity can be crucial for the success of coevolutionary algorithms applied to minimax problems.

Keeping both optima in the population is made difficult by the heavy use of recombination in the algorithm. Consider what happens if the scenario population is divided such that half the population is located on the $s = 0$ peak, and half of it is at $s = 10$. Assume that the solution population is such that both peaks are equally good, so the $s = 0$ and $s = 10$ individuals will all be assigned (roughly) the same fitness. Since almost all of the new individuals are generated using crossover, nearly half the new individuals will be generated with one $s = 0$ parent and one $s = 10$ parent. All of these new individuals will be located uniformly between 0 and 10 because of the crossover operator used. For this reason, there is a strong drive in the scenario population away from the two optima towards the middle of the search-space. This is illustrated in figure 2.12. This problem could probably be avoided using some kind of speciation method, which could increase the performance significantly. Another possibility would be to use an algorithm based on mutation as the predominant operator.

Conclusions drawn from the experiments

The experiments of this section have clearly shown that the coevolutionary approach to minimax problems of Herrmann [58] fails on problems where the two search-spaces do not share the symmetric property (2.6). A similar set of experiments were conducted for the very similar algorithm presented by Barbosa

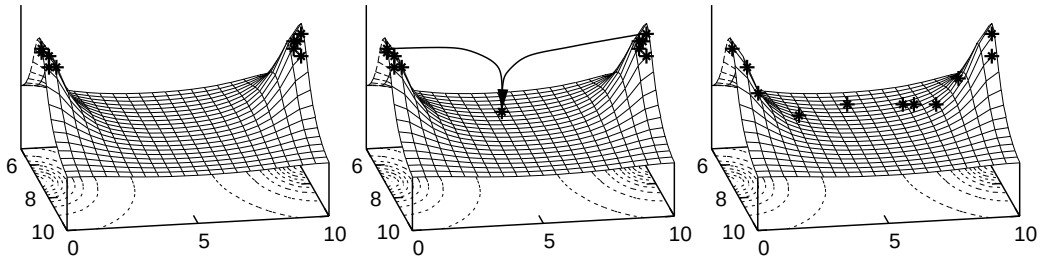

Figure 2.12: *An illustration of crossover breaking moving the scenario population away from the two peaks.* ***Left:*** *Population concentrated on the two peaks (the ideal situation).* ***Middle:*** *Crossover "averaging out" two individuals.* ***Right:*** *The population as it may look one generation later.*

[5, 7] (the MSE errors can be found in appendix B). These experiments indicated that in terms of mean square errors this algorithm performed slightly better than Herrmann's algorithm, but that the fundamental problem remained unsolved; the algorithm had severe difficulties converging to the right solution and scenario.

The experiments have also demonstrated that the new asymmetric fitness evaluation is able to solve the problems of the fitness evaluation used in [5, 6, 7, 58]. The algorithm based on this fitness evaluation clearly outperforms the other algorithms on the three benchmarks not satisfying (2.6).

Another lesson learned from the experiments is that keeping the diversity of the populations high can be crucial for coevolution to work when solving minimax problems. If the diversity of the scenario population gets too low, suboptimal solutions can be able to take over in the solution population, or coevolutionary dynamics can lead the search away from the optimum. For this reason, diversity maintaining measures such as crowding, tagging or structured populations [10, 102] seem like a good idea to combine with the ideas presented here.

Chapter 3

Deterministic Scheduling

Following Błazewicz et al., [18], scheduling problems can be broadly defined as "*the problems of the allocation of resources over time to perform a set of tasks*". The scheduling literature is full of very diverse scheduling problems [23, 45]. In this chapter, a definition of and an introduction to the job shop scheduling problem will be given. For reasons of brevity and in order to make the presentation concrete, production specific terms will be used, but in fact the job shop scheduling problem is relevant also to non production problems, e.g. aircraft queueing up to land, taxi, and getting service at an airport or nurses and doctors taking care of patients at a hospital. The chapter is not meant to be a thorough description of static job shop scheduling (for good introductions to static scheduling see [18, 45]), it is rather meant as a necessary introduction to the concepts used in the chapters on stochastic scheduling, along with a brief overview of static scheduling techniques.

3.1 Introduction to job shop scheduling

In this section, the deterministic job shop problem will be introduced, and a few of its fundamental properties discussed.

3.1.1 Definition of the problem

A *job shop problem* P of size $n \times m$ consists of n *jobs* $\{J_1, J_2, \ldots, J_n\}$ and m *machines* $\{M_1, M_2, \ldots, M_m\}$. For each job J_i, a sequence of k_i *operations* $O_i = (o_{i1}, o_{i2}, \cdots, o_{ik_i})$ describing the *processing order* of the operations of J_i is given. The processing orders of the jobs are sometimes called *the technological constraints*. The operation o_{ij} is the jth operation of job i, and is to be processed on a specific machine $M_{o_{ij}}$ and has a *processing time* $\tau_{o_{ij}}$. The set of all operations in P is denoted T.

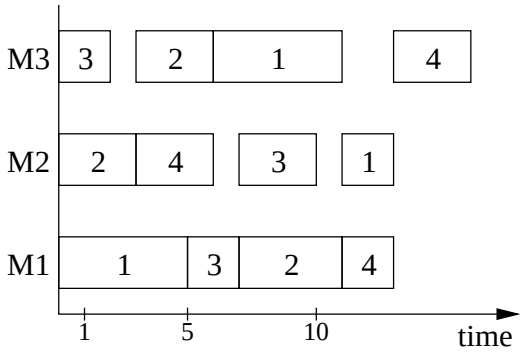

Figure 3.1: *Example Gantt-chart.*

Each job can have an associated *release time* r_i before which no processing of the job can take place, and each machine an associated *initial setup time* u_i before which no processing can be done on the machine. It is customary to work with discrete time in job shop scheduling, meaning that processing times, release times and initial setup times are all integer. When the operations are processed on the machines, each machine can process only one operation at a time, only one operation from each job can be processed at a time and no preemption can take place (that is, once an operation has started it cannot be stopped until it has finished).

A *schedule* is a description of when to process each of the operations satisfying the constraints. A schedule is often visualised using a Gantt-chart, an example of which can be found in figure 3.1. Each row in the Gantt-chart represents a machine and each box represents an operation. The boxes have been labelled with the job number they belong to, and the timetabling information of each operation can be read on the time-axis.

Two other classical scheduling problems are closely related to job shop scheduling. The *flow shop problem* is a special case of the job shop problem in which the machine processing orders O_i are the same for all the jobs. This does not mean the jobs need to be identical, since the operation processing times may vary from job to job. A special flow shop is the *permutation flow shop*, in which the same processing sequence must be used on all the machines. The *open shop problem* is equivalent of the job shop problem except that there are no technological constraints; the operations of a job need not be processed in any particular order.

For the job shop formulation to apply to a given problem, the processing environment must satisfy a number of assumptions:

- No identical machines. Each machine is unique; no choice is allowed as to which machine an operation is to be processed on.

- Schedule independent processing times. The processing times are required to be independent of the schedule. This means that the processing times are independent of the processing orders of the machines. In some problems, machine setup times may be dependent on the sequence. Fixed transportation costs from machine to machine is also required.

- No two operations of the same job can be processed simultaneously. This excludes problems in which a job is composed by a number of independently produced components to be assembled into one product.

- In process inventory is allowed. Jobs are allowed to wait for their next machine in the processing line for as long as they need. This excludes processing environments in which there is limited room to hold the in process inventory as well as environments in which for some reason jobs must keep on running, once processing has been started (e.g. in a steel plant in which you literally have to work while the iron is hot).

- All of the problem is known before scheduling takes place. In particular, no new jobs arrive over time.

- Nothing unforeseen ever happens. Unforeseen events include the breakdown of a machine or the processing of an operation getting delayed.

Furthermore, in classical job shop scheduling it is usually required that for each job J_i the sequence of operations O_i contains exactly one operation to be processed on each of the machines. The problems treated in this chapter will be satisfying all of these constraints, while in the chapters 4, 5, and 6 on stochastic scheduling the assumption of no unforeseen events will be removed.

Many different performance measures exist for job shop problems, but before these can be defined some notation needs to be introduced:

- C_i. The *completion time* of the last operation of job J_i. Often referred to as the completion time of J_i.

- F_i. The flow-time of job J_i, defined as the time elapsed from J_i becomes ready for processing until it has finished. $F_i = C_i - r_i$.

- d_i. Some problems specify a due date d_i for each job. d_i is the deadline for J_i, the time at which the processing must be finished.

- L_i. The *lateness* of J_i, $L_i = C_i - d_i$. The lateness measures how much later than the deadline the job finishes. If the job finished earlier than d_i, it is assigned a negative lateness.

- T_i. The *tardiness* of J_i, $T_i = \max(L_i, 0)$.
- E_i. The *earliness* of J_i, $E_i = \max(-L_i, 0)$.

Frequently used performance measures for job shop problems are:

- *makespan* $C_{max} = \max_{i \in \{1...n\}} C_i$. The maximum completion time. Some authors refer to this measure as the *length* of the schedule.
- *total flow-time* $F_{\sum} = \sum_{i \in \{1...n\}} F_i$. The total time spent on all the jobs.
- *total lateness* $L_{\sum} = \sum_{i \in \{1...n\}} L_i$. The summed lateness of all the jobs.
- *total tardiness* $T_{\sum} = \sum_{i \in \{1...n\}} T_i$. The summed tardiness of all the jobs.
- *total earliness* $E_{\sum} = \sum_{i \in \{1...n\}} E_i$. The summed earliness of all the jobs.
- *maximum lateness* $L_{max} = \max_{i \in \{1...n\}} L_i$. The lateness of the latest job. Sometimes called the *worst lateness*.
- *Maximum Tardiness* $T_{max} = \max_{i \in \{1...n\}} T_i$. The tardiness of the tardiest job. Sometimes called the *worst tardiness*.

All of these performance measures are to be minimised. Usually the total earliness $E_{\sum}$ is not used as a performance measure on its own. It is used in conjunction with another performance measure (e.g. $T_{\sum}$) to form a composite performance measure (e.g. $\alpha T_{\sum} + \beta E_{\sum}$). This is done for problems in which it is important to minimise inventory costs by processing jobs as close to the due date as possible.

3.1.2 Complexity of static job shop problems

Most variants of the job shop problem except a few formulations with the number of machines or jobs limited to 1 or 2 are known to be NP-hard [23, 45]. In particular, job shop problems with the number of machines m fixed ($m \geq 2$) using the C_{max} and $F_{\sum}$ performance criteria are NP-hard in the strong sense [46, 47]. Since C_{max} problems can trivially be reduced to T_{max} and L_{max} problems, and $F_{\sum}$ problems can trivially be reduced to $T_{\sum}$ and $L_{\sum}$ problems, these problems are NP-hard in the strong sense as well.

According to [45] theorem 11.6 and [46] theorem 1, strong NP-hardness of a problem implies that it is impossible to create a search heuristic which guarantees to find a solution for which the relative error is bounded by

$$\frac{\text{performance measure of found solution}}{\text{performance measure of optimum solution}} \leq 1 + \epsilon$$

and which runs in polynomial time both in the problem size and $\frac{1}{\epsilon}$. This result is very important, since it indicates that efficient approximation algorithms with guaranteed performances should not be expected for these problems unless $P = NP$. For this reason, most research focused on finding (near) optimal schedules has been turned towards implicit enumeration algorithms (branch and bound techniques), local improvement methods (shifting bottleneck) and heuristic search methods such as genetic algorithms, tabu search and simulated annealing.

Experience has shown job shop problems not just to be NP-hard, but to be very difficult to solve heuristically even for problems in this complexity class. The 10×10 instance `ft10` of [43] was first published in 1963, and remained open until the optimal solution was published by Adams et al. in 1988 [2]. Recent results, [96], indicate that the open shop problem (which is not as well researched as the job shop problem) is even more difficult than the job shop problem. In [96] this extraordinary problem hardness is attributed in part to the larger search-space of the open shop for problems of the same size, and in part to smaller gaps between the performance of optimal schedules and lower bounds.

3.1.3 Regular measures and classes of schedules

Performance measures for which the performance cannot be worsened by changing a schedule such that an operation finishes earlier are called *regular*. The performance measures C_{max}, $F_{\sum}$, T_{max}, $T_{\sum}$, L_{max} and $L_{\sum}$ are all regular, [45], since C_i, F_i and L_i can never increase by finishing an operation earlier. Measures based on earliness are irregular. Schedules are often categorised in the following classes:

- A schedule in which no operation can be started earlier without changing the processing order or violating the technological constraints is termed *semi-active*.
- A schedule in which no operation can be started earlier without delaying at least one other operation or violating the technological constraints is termed *active*.
- A schedule in which no machine is ever idle if an operation is ready to be processed on it is called *non-delay*.

Examples illustrating semi-active, active and non-delay schedules can be found in figure 3.2. The set of non-delay schedules is a subset of the active schedules, which is a subset of the semi-active schedules. For regular performance measures it can be shown that for any problem an optimal active schedule exists, [45]. However, it has also been demonstrated that for some problems no non-delay schedule

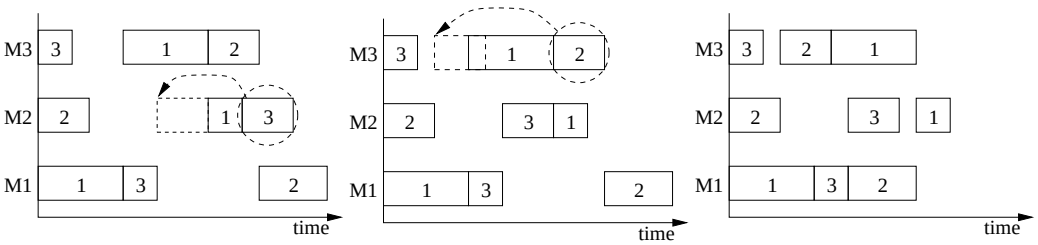

Figure 3.2: ***Left:*** *Semi-active schedule. The schedule is semi-active since no operation can be started earlier by sliding it to the left. The schedule is not active, since the operation marked by the circle can be started earlier by leap-frogging it past the operation in front of it, which does not cause any operation to be delayed.* ***Middle:*** *Active schedule. The schedule is active since no operation can be started earlier without delaying another operation. The operation marked by the circle can be leapfrogged to start earlier, but this will delay the operation of job 1; note the conflict between the leapfrogged operation (dashed) and the operation of job 1. The schedule is not non-delay, since the operation marked by the circle is ready for processing at the time marked by the arrow, yet machine 3 is kept idle.* ***Right:*** *Non-delay schedule. The schedule is non-delay since no machine is ever kept idle if an operation is ready for processing on it.*

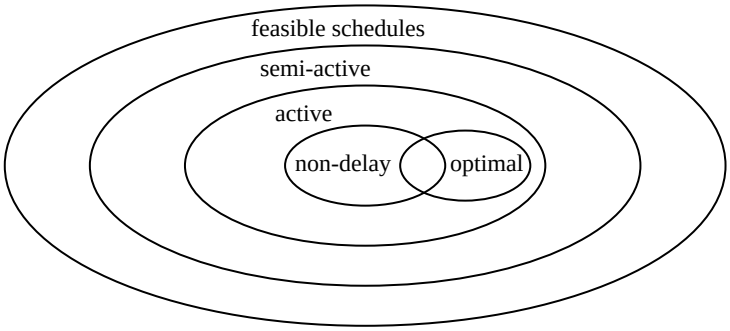

Figure 3.3: *The hierarchy of the schedule classes for regular performance measures. The set of non-delay schedules is a subset of the active schedules, which is a subset of the semi-active schedules. For regular performance measures, the optimal schedules is a subset of the active schedules, but not a subset of the non-delay schedules.*

is optimal. The relationship between these classes of schedules and optimal schedules is illustrated in figure 3.3. For this reason, when solving scheduling problems involving regular performance measures, usually only the set of active (or sometimes semi-active) schedules are searched, since this brings a huge reduction of the size of the search-space while still guaranteeing that an optimal schedule can be found.

For many problems the average performance (when sampling schedules at random) of non-delay schedules is significantly higher than the average performance of active schedules. Many scheduling problems are so hard that with present day techniques it is not realistic to find the globally optimal solution. Experiments have shown that restricting the schedules searched by the scheduler to incorporate only non-delay schedules or a set of schedules somewhere between active and non-delay schedules can sometimes improve scheduling performance, [16]. This effectively guarantees that the optimal solution will never be found for a number of problems, since the optimal solution is not in the search-space of the algorithm, but it has been observed to increase average performance.

Searching only semi-active, active or non-delay schedules has the added advantage that a schedule is completely described by the *the processing sequence*, the processing orders on the machines. When a schedule is represented in a search algorithm, it often suffices to hold a representation of the processing sequence instead of holding the timetabling information. For this reason, sometimes the term *schedule* (a schedule includes timetabling information) is used interchangeably with the term processing sequence (which holds not explicit timetabling information).

3.1.4 The graph representation

A very useful representation of schedules and scheduling problems is *the graph representation*. Much reasoning about scheduling is based on this representation, including hillclimbing and branch and bound techniques. This section begins with a description of the graph representation for schedules, followed by a description of how to represent scheduling problems.

Representing schedules

For regular performance measures it suffices to represent a schedule by the processing sequence. The processing sequence can be thought of as a number of "process before" relations between the operations. These "process before" relations can be represented by a directed graph $G = (V, A \cup E_s)$ in which each node in *the node set* V represents an operation and each arc in *the conjunctive arc set* A and *the disjunctive arc set* E_s represents a relation. In figure 3.4, the Gantt-chart

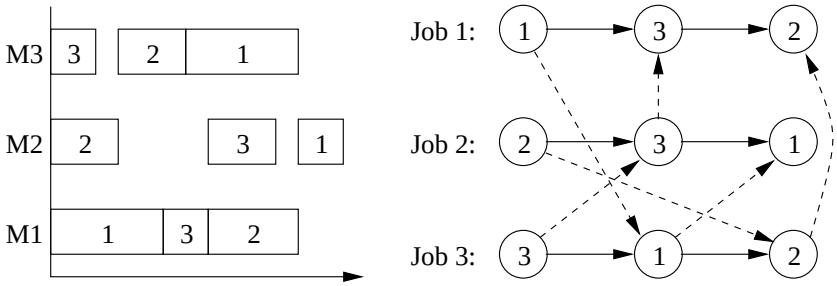

Figure 3.4: *A Gantt-chart and a graph representing the schedule.*

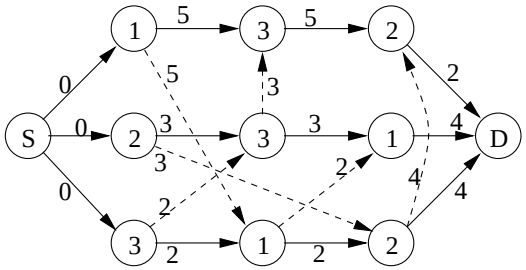

Figure 3.5: *Graph representing the schedule of figure 3.4, augmented with source and drain nodes and weights representing processing times.*

of a schedule has been drawn along with its graph-representation. The nodes in the figure have been organised so that each row of nodes represents a job. Each node has been labelled with its machine number. The solid arcs in the graph are in the conjunctive arc set A. They represent technological constraints and are fixed by the problem instance. The dashed arcs are in the disjunctive arc set E_s. They represent the processing orders of the machines and were decided by the scheduler. Note that there are a number of arcs not explicitly drawn on the graph. For job 1, the first operation is on machine 1, the second on machine 3, and the third on machine 2. Yet no arc has been drawn from the operation on machine 1 to the operation on machine 2. This relation is implicitly there, since there are arcs from the operation on machine 1 to the operation on machine 3 to the operation on machine 2. When drawing the graph in this way, each node o will have out-degree at most two, one arc leading to the *job successor*, denoted $SJ(o)$, and one to the *machine successor*, denoted $SM(o)$. The in-degree will also be at most two, one arc coming from the *job predecessor*, denoted $PJ(o)$, and one coming from the *machine predecessor*, denoted $PM(o)$.

This representation can be further enhanced by adding two dummy nodes, a source node (marked S in figure 3.5) connected to each node representing the first

operation in a job and a drain node (marked D), that all operations last in a job are connected to. The arcs can be augmented with weights equal to the processing times of the operations (nodes) in which they originate. The arcs leading out of the source node are assigned the weight zero (we are assuming that the problem has zero release and initial setup-times, removing this assumption is straight-forward). The makespan of the semi-active schedule can now be calculated as the heaviest path from the source to the drain.

Given the processing order of machines in a semi-active schedule, the graph representing the semi-active schedule can easily be constructed by connecting nodes representing operations on the same machine with arcs reflecting the processing sequence. The timetabling information of the schedule can then be calculated in the following way.

The completion time c of an operation o can be calculated as

$$c(o) = \tau_o + h(o),$$

where $h(o)$ is the starting time of o, often called *the head* of o. The head of an operation can be calculated as

$$h(o) = \max(c(PM(o)), c(PJ(o))),$$

where $c(PM(o))$ is set to zero if $PM(o)$ is undefined, and $c(PJ(o))$ is set to zero if $PJ(o)$ is undefined.

In the same way, the minimum time elapsed from the end of processing operation o until all the operations following it are finished is termed *the tail* of o, $t(o)$. The tail of o can be calculated as

$$t(o) = \max(\tau_{SM(o)} + t(SM(o)), \tau_{SJ(o)} + t(SJ(o))),$$

where again $t(SM(o))$ and $\tau_{SM(o)}$ are set to zero if $SM(o)$ is undefined, and the same is done for $SJ(o)$.

An operation is termed *critical* if it cannot be delayed without worsening the performance of the schedule. For the makespan criterion, an operation is critical if

$$C_{max} = h(o) + \tau_o + t(o). \tag{3.1}$$

The critical operations in a schedule form one or more *critical paths*. A critical path is a sequence of critical operations $o_1, o_2, \ldots, o_l$, such that $SM(o_i) = o_{i+1}$ or $SJ(o_i) = o_{i+1}$ for all $0 \leq i < l$. A *critical block* is a subsequence of a critical path $o_a, o_{a+1}, \ldots, o_b$, such that $SM(o_i) = o_{i+1}$ for all $a \leq i < b$, and such that neither $PM(o_a)$ nor $SM(o_b)$ are critical. Informally, a critical block is a sequence of consecutive critical operations on the same machine.

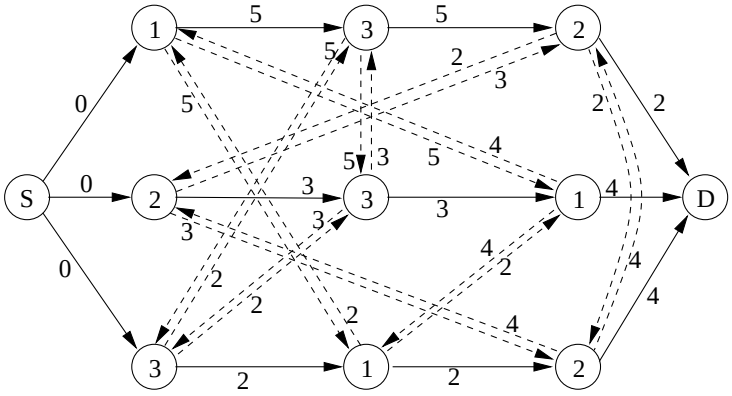

Figure 3.6: *Graph-representation of the problem of figures 3.4 and 3.5.*

Critical paths and critical blocks are important, since they form the basis of many iterative improvement heuristics on scheduling problems. From equation (3.1), it is clear that for makespan problems, a local change cannot improve schedule makespan if it does not change the head $h(o)$ or tail $t(o)$ of all critical operations. It can be shown [39] that a local change involving only small changes cannot improve makespan if it does not include changes at the beginning or end of a critical block. Considerations like these form the basis for tabu search approaches such as [39, 86] and the hillclimber used in [77], described in section 3.2.2.

Representing scheduling problems

A graph representing a job shop problem is created by making a graph $G = (V, A \cup E)$ where the disjunctive arc set E holds all possible machine processing relations, see figure 3.6. In the problem graph, every pair of nodes representing operations on the same machine are connected by an arc pair, one arc going in each direction. The minimum makespan problem can be formulated as a graph problem in which to choose a subset $E_s \subset E$ such that all operations on the same machine are connected by a Hamiltonian path, minimising the heaviest path from source to drain. In order to transform the graph into a graph representing a schedule, for each of the arc pairs in E one arc will have to be chosen. A feasible schedule has been generated when there are no cycles in the graph and when all operations on the same machine are connected by a Hamiltonian path.

Graph representations of problems based on other measures than makespan can be made in similar ways.

The Hamming distance measures on schedules

A distance measure on job shop schedules will now be defined. The distance measure is due to Mattfeld [77], and will later be useful for describing hillclimbers and the neighbourhood based robustness technique presented in chapter 5.

Consider a scheduling problem P represented by the graph $G_P = (V, A \cup E)$ and two schedules solving P, s_1 and s_2 represented by $G_1 = (V, A \cup E_{s_1})$ and $G_2 = (V, A \cup E_{s_2})$. The *absolute Hamming distance* $D(s_1, s_2)$ is defined

$$D(s_1, s_2) = \frac{1}{2} \sum_{(o_1,o_2)\in E} ((o_1, o_2) \in E_{s_1}) \oplus ((o_1, o_2) \in E_{s_2}), \tag{3.2}$$

where $\oplus$ represents logical exclusive-or, such that the term under the summation is 1 if (o_1, o_2) is in E_{s_1} but not in E_{s_2} or vice versa. The $\frac{1}{2}$ factor in front of the sum is there since the pair (o_1, o_2) is present in E in the form (o_2, o_1) as well, meaning that every pair of operations is counted twice in the summation. The absolute Hamming distance $D(s_1, s_2)$ is equal to the number of precedence relations that differ from s_1 to s_2.

Based on the absolute Hamming distance, the $\mathcal{N}_k$-neighbourhoods on schedules can now be defined. The $\mathcal{N}_k$-neighbourhood of the schedule s is the set of feasible schedules that have an absolute Hamming distance to s of k or shorter:

$$\mathcal{N}_k(s) = \{s' \in S \mid D(s, s') \leq k\}. \tag{3.3}$$

The *relative Hamming distance* $d(s_1, s_2)$ between two schedules is defined as their absolute Hamming distance divided by the total number of operation pairs on the same machine:

$$d(s_1, s_2) = \frac{2D(s_1, s_2)}{|E|}. \tag{3.4}$$

For problems where every job is to be processed once on each machine, this is equal to

$$d(s_1, s_2) = \frac{D(s_1, s_2)}{mn(n-1)}. \tag{3.5}$$

3.1.5 Schedule generation

The classical algorithm for active schedule generation is known as *the Giffler-Thompson algorithm*, [48], and was first published in 1960. The original algorithm generates active schedules, but can easily be modified to generate semi-active or non-delay schedules instead. The Giffler-Thompson algorithm can be seen in figure 3.7.

```
set s to the empty schedule
set A = {o_i1 | 1 ≤ i ≤ n}, the set of operations that
   are first in a job
while (A is not empty) do
   find the operation o ∈ A with the earliest
     potential completion time σ (if two or more
     operations are tied, pick one at random)
   set M* to the machine that is to process o
   set Q to the set of operations from A that are
     to be processed on M* and have possible
     starting times earlier than σ
   pick operation o* = Φ(Q) from Q, remove o* from A
   add o* to s with starting time h(o*)
   if a job successor SJ(o*) of o* exists add it to A
od
```

Figure 3.7: *The Giffler-Thompson algorithm*

In the following, an operation o is said to be *schedulable* if it has not been scheduled, but all of the operations of the same job that must precede it have been scheduled.

The set A is used to hold the set of schedulable operations. s is the schedule being constructed. Initially A is set to the operations to be processed first in each job, and s is set to the empty schedule. After the initialisation the schedule s is iteratively built operation by operation, while making sure no "holes" large enough to accommodate an operation are left in the schedule. In the main loop, this is done by locating in A the operation o with the earliest possible completion time σ. The machine on which this operation is to be processed is called M^*. The set of operations of A which are to be processed on M^* and have potential starting times earlier than σ is stored in Q. An operation o^* is then picked from Q according to some scheduling policy Φ. Φ is a procedure determining which of the schedulable operations to schedule first. It can be a simple dispatching rule, a procedure inspecting a genetic string or just a procedure returning a random operation in Q. o^* is added to the schedule and if the job successor $SJ(o^*)$ exists, it is added to A. This cycle is repeated until there are no operations left in A.

The fact that this algorithm can only generate active schedules is easily realised. When scheduling an operation from Q on machine M^*, we are choosing from the set of operations which have potential starting times earlier than the earliest potential completion time of the operations in A. Since all operations later to be added to A will have potential completion times later than σ, there is no way a hole large enough to process an operation from start to finish can be left in the

schedule.

Given a set of processing sequences (one sequence for each machine) describing an active schedule, the algorithm can be used to construct the schedule. This is done simply by letting the choice of o^* from Q be decided by the processing sequence of M^*. The algorithm can be shown to be able to generate any active schedule for a given problem, see [48].

It is instructive to visualise how the Giffler-Thompson algorithm works on the graph representation of a scheduling problem. Consider the problem graph of figure 3.6. Each step in the following is visualised in figure 3.8, where the operations in A have been marked by dashed circles on the graph representing the problem. Operations included in the set Q have been marked by squares. The arcs in the arc pairs have only been drawn after they are resolved to improve readability of the figure. The Gantt-charts representing the schedule as it looks at each step have also been drawn.

After the initial step of the algorithm, the set of schedulable operations A holds the three nodes which the source S is connected to. Inspecting the weights of the arcs we see that the operation with earliest possible completion time $\sigma = 2$ is to be processed on machine 3. Since this is the only operation in A to be processed on machine 3, it is scheduled for processing, removed from the set A and the second operation of job 3 (to be processed on machine 1) is added to A. This resolves the arc pairs involving the operation, the resolved arcs can be seen in the graph. After this similar events set the first operation of job 2 to be processed first on machine 2 (no other choice is possible), which adds the second operation of job 2 (on machine 3) to A. For the third operation to be scheduled machine 1 holds the earliest possible completion time of $\sigma = 4$ (the second operation of job 3). In this case there are two possible choices of which operation to schedule: the first operation of job 1 or the second operation of job 3. In the figure the second operation of job 3 has been chosen. This step is the last in the figure, but the algorithm will keep running until all of the operations have been added to the schedule and A is empty.

The Giffler-Thompson algorithm can be modified to generate non-delay schedules. Figure 3.9 shows a version of the algorithm capable of generating non-delay and active schedules. The algorithm contains the parameter δ, that decides which set of schedules the algorithm is capable of generating. For $\delta = 0$, the algorithm produces non-delay schedules, while $\delta = 1$ indicates that all active schedules can be produced (for suitable Φ). For values $0 < \delta < 1$, the algorithm produces active schedules with a bias towards non-delay schedules; the smaller δ is, the less idle time is allowed on the machines before they have to start processing.

The algorithm works in a way very similar to the original Giffler-Thompson algorithm, but a few steps have been changed. The most important difference is

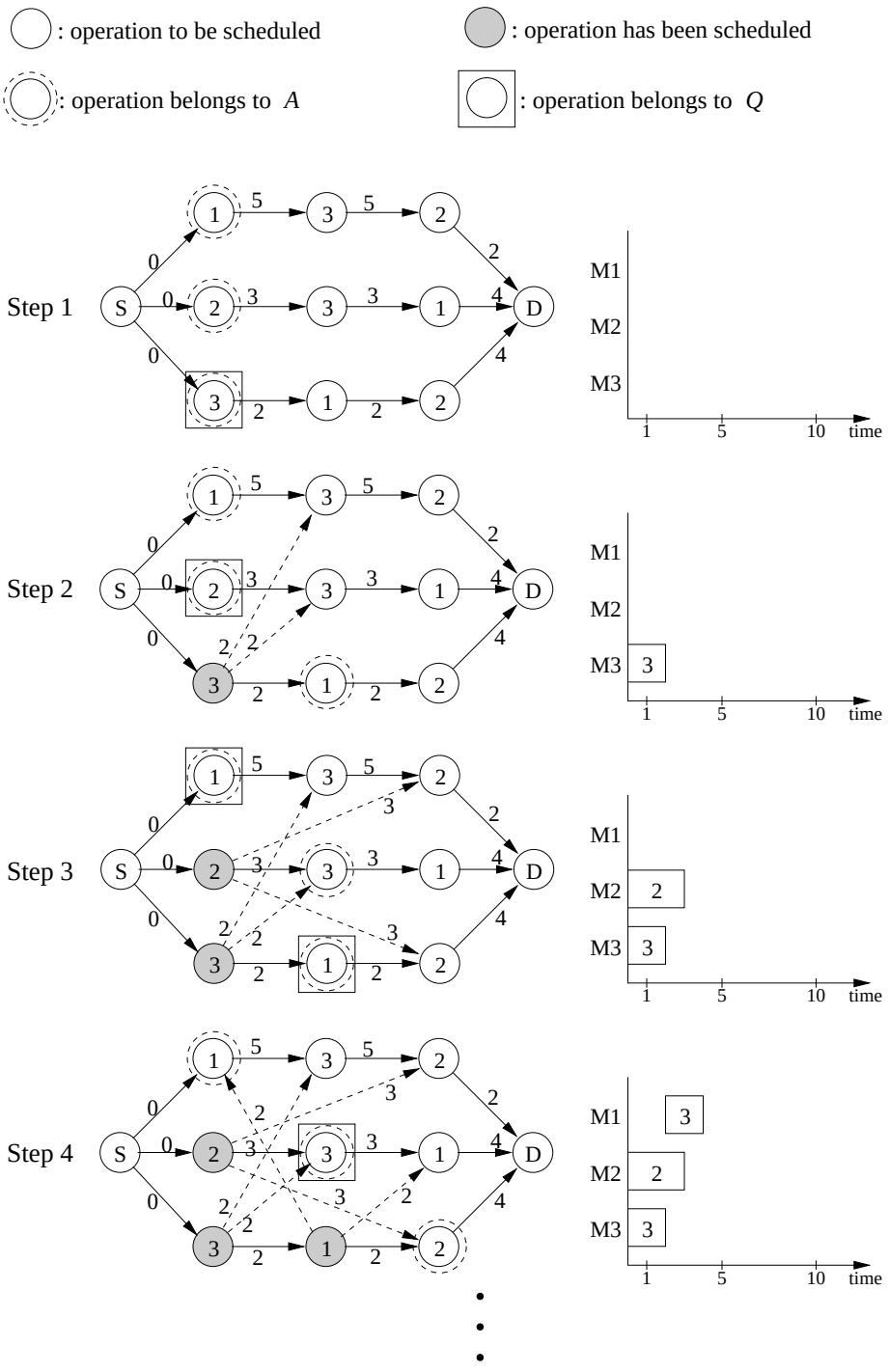

Figure 3.8: *A few steps of the Giffler-Thompson algorithm of figure 3.7 visualised on a problem graph, and with Gantt-charts showing the partially complete schedule. Unresolved arc pairs have not been drawn to improve readability.*

```
set s to the empty schedule
set A = {o_i1 | 1 ≤ i ≤ n}, the set of operations that are
   first in a job
while (A is not empty) do
   find the operation o ∈ A with the earliest
     potential completion time σ (if two or more
     operations are tied, pick one at random)
   set M* to the machine that is to process o
   find the earliest possible starting time h(o') of
     an operation o' ∈ A to be processed on M*
   set Q to the set of operations from A to be
     processed on M* with potential starting times
     min(h(o') + δ(σ − h(o')), σ − 1) or earlier
   pick o* = Φ(Q) from Q and remove it from A
   add operation o* to s with starting time h(o*)
   if a job successor SJ(o*) of o* exists add it to A
od
```

Figure 3.9: *The modified Giffler-Thompson algorithm for active/non-delay schedule generation. The lines that are different from the ordinary Giffler-Thompson algorithm (figure 3.7) have been printed in bold.*

the construction of Q, which is has been changed so that operations which require M^* to wait for too long are excluded.

3.2 Evolutionary scheduling

This section contains a short introduction to evolutionary methods applied to job shop scheduling in general, followed by a detailed description of the genetic algorithm used for some of the experiments in chapter 5. Reviews on evolutionary approaches to scheduling can be found in [24, 30, 55]. The section also has a very brief introduction to the application of other search heuristics to job shop scheduling.

3.2.1 EAs and job shop scheduling

The first attempt at solving the job shop problem using evolutionary methods was published in 1985 by Davis, [33]. Since then a substantial number of papers have appeared on the subject. Finding a suitable genetic representation for the schedules is not straight-forward so many of the papers focus on the development

of an efficient representation.

One of the earlier attempts at solving job shop problems with evolutionary methods was done by Nakano and Yamada, [84]. They used a standard binary string representation with a binary variable for each pair of operations o_1, o_2 to be processed on the same machine. This binary variable would be 1 if o_1 preceeded o_2 in the processing order, and 0 if o_2 preceeded o_1. This representation allows for the use of standard binary genetic operators, but causes several other problems. First of all, the size of the representation (number of bits) for a $n \times m$ problem is given as $\frac{1}{2}mn(n-1)$. The fact that the representation size scales quadratically with the number of jobs makes it very poor for large problem instances. Furthermore, the representation contains infeasible schedules, meaning that every new individual needs to go through a repair algorithm in order to guarantee feasibility. A new concept introduced in [84] is the idea of *forcing*; after a genotype g has gone through the repair step of the algorithm, it is replaced by g', its feasible counterpart. This is a kind of Lamarckian learning, and Nakano and Yamada demonstrate it to improve both the solution quality and the convergence speed of the algorithm.

In a later paper, [115], Yamada and Nakano do experiments with a different representation. In this representation there is a position in the chromosome for every operation, in which the end of processing time of the operation is recorded. This representation has the advantage of being very direct; the gene practically is the schedule, very little decoding is needed. Yamada and Nakano develop a crossover operation called the *GA/GT* crossover, which is based on the Giffler-Thompson algorithm. Their experiments indicate that the proposed algorithm outperforms their previous approach, [84], and that the use of the crossover operation is very helpful; it improves convergence speed and reduces the variability of final solution quality.

The similarity between the Travelling Sales Person (*TSP*) problem and the job shop problem has been exploited by several authors. Permutation based representations used for the TSP have been used as inspiration for JSP representations [28, 53, 113]. An example of this is the work by Fang et al. [41, 42], in which a schedule is represented by a sequence of numbers from $\{1, .., n\}$. During decoding the sequence is read from left to right, and the numbers in the sequence are used as indexes in a circular list of uncompleted jobs. If the number present at the ith position in the gene is a, the ith operation scheduled will be the first unprocessed operation of the job at position $a \bmod l$ in the list of uncompleted jobs, where l is the current length of the list. This representation can only represent feasible schedules. A problem in this representation is that the meaning of genes located late in the genetic string depends on the contents of the genes earlier in the string. This implies that the convergence rates are different for different parts of the chromosome; early in the run of the algorithm, the first part of the genetic

string will begin to converge, while the rest of the string will not converge, since the meaning of the rest of the string is not clear until the first part of the gene has converged. In the later stages of the run, the first part of the string will be converged, and the later parts will begin converging as well. In this representation, the dependency of the later parts of the string depending on the earlier parts can also be seen as a *false competition* or a *competing conventions* problem in the GA. The same building block (or *forma*, [97]) can be represented in several different ways. A problem arises when two genotypes both holding the same building block, but represented in different ways, are recombined to form a new genotype. Chances are that the building block will not be present in the offspring, since if the earlier parts of the parents strings do not agree, the schema that decoded to the building block in a parent will decode to something different in the child.

The most commonly used genetic representations for jobs shop like problems currently are probably permutation with repetition representations (see [14, 77] and the next section) or more direct representations based on start or ending times of operations, used as operation priorities during decoding. These representations cannot represent infeasible schedules, so no repair procedures are necessary. Usually, the representations are redundant, one schedule can be represented in many different ways. Furthermore, the representations are often biased; some schedules have only a small number of representations, while other schedules have a huge number of representations. This situation is not ideal; usually an unbiased representation is preferable to a biased one, but for these problems bias is probably the price to pay for avoiding infeasibility. Very often variants of the Giffler-Thompson algorithm (section 3.1.5) are used for decoding these representations, [55]. The crossover operators used are often designed to work on the schedule level to combine parents in a respectful way (like the THX operator [74], or the PPX operator [17]).

More recent work on GAs and static job shop scheduling focuses more on performance enhancing measures and less on the development of representations. Typically these articles present methods that include hillclimbers (these algorithms are sometimes called *genetic local search* or *memetic algorithms*) [1, 77], the application of problem specific knowledge [16, 25], or they use more advanced evolutionary models such as spatial populations to avoid premature convergence and local optima [22, 74, 77]. An example of the latter is presented by Lin, Goodman and Punch in [74]. The algorithm presented in this paper seems inspired by the one presented in [115]; the genetic representation is done by recording operation starting times, and the genetic operators are akin to those presented in [115]. In [74] it is demonstrated that clever use of population structures can improve the performance of the algorithm, especially when huge population sizes and long running times are allowed. The authors do experiments with island based and diffusion based distributed genetic algorithms, as well as combinations of these.

3.2.2 Mattfeld's GA3

One of the most successful GAs for job shop scheduling is the algorithm presented by Mattfeld in [77]. The performance of Mattfeld's GA is compared to some other heuristic job shop scheduling methods in [111]. It is concluded to be the most efficient genetic algorithm for job shop scheduling, while in some situations it is inferior to the tabu search algorithm due to Nowicki and Smutnicki [86], see section 3.3.2.

In his Ph.D thesis, [77], Mattfeld presents three different GAs. They are labelled GA1, GA2 and GA3, where the algorithms with a higher label number have a better performance and are more complicated than the algorithms with a lower label number. Only GA3 will be presented here.

GA3 uses a permutation-based representation which has the advantage over some other job shop representations that it cannot represent infeasible schedules. The decoding used in the algorithm is a very simple procedure producing semi-active schedules. After the decoding step, the schedule is improved by a hill-climber. A structured population in the form of a diffusion model is used, and the rates of crossover and mutation are adapted using a behavioural model. Mattfeld's GA3 is a fairly complex GA. The main reasons for its success are probably the use of problem specific knowledge in the hillclimber, and the diffusion model combined with the behavioural model, which counter premature convergence and makes the algorithm search a larger part of the search-space before convergence.

Mattfeld's algorithm forms the basis for some of the experiments in chapter 5, so it will be treated in detail in this section.

Representation

A semi-active schedule is uniquely determined by its graph-representation, as discussed in section 3.1.4. Since the graph representation is an acyclic directed graph in which we know which nodes are to be connected, the graph can be determined from a topological sorting of the nodes. Since each node is an operation that can be determined by a job number and the number of the operation in the processing sequence of the job, such a sorting can be written as a sequence $((j_1, b_1), (j_2, b_2), \ldots, (j_k, b_k))$, where j_i and b_i are the job number and number in the processing sequence of operation i respectively. If the graph represents a feasible schedule, clearly (j_i, l) must precede $(j_i, l+1)$ in the sorting, otherwise the technological constraints of the problem are violated. Obviously, the first occurrence of (j_i, l) must be $(j_i, 1)$, the second must be $(j_i, 2)$ and so forth. Thus, the topological sorting of the graph (and the schedule) can be uniquely determined from a sequence of job numbers $g = (j_1, j_2, j_3, \ldots, j_k)$. Since this sequence must contain just as many occurrences of J_i as there are operations in job J_i, the repre-

sentation is called *permutation with repetition*.

A very intuitive way of decoding this kind of representation is by reading the representation from left to right and constructing the Gantt-chart of the schedule operation by operation. The gene $g = (1, 3, 1, 2 \ldots)$ would be decoded by saying "First schedule the first operation of job 1, then the first operation of job 3, then the second operation of job 1, then the first operation of job 2,...". In the following, this kind of decoding is called *semi-active decoding*. This is the kind of decoding used in GA3.

There are other ways of decoding genes of this kind as well. The gene can be used as input to the scheduling policy Φ in a schedule builder such as the algorithms of figures 3.7 and 3.9. When Φ picks an operation from Q, the leftmost operation of g that has not yet been scheduled will be selected.

The permutation with repetition representation has the advantage that it cannot represent infeasible schedules. It is easy to realise that any feasible semi-active schedule can be represented, since such schedules can be described by the topological sorting of their graph representation. Using this representation also has the advantage that genetic operators developed for permutation representations of the TSP can be modified to work on it.

Hillclimbing

It is often found that the use of problem specific knowledge can greatly improve the performance of a GA. Mattfeld's GA3 uses problem specific knowledge in the form of a hillclimber designed to improve the schedules created by semi-active decoding by decreasing the schedule makespan. The GA uses forcing; after the hillclimber has completed, the improved schedule is written back to the gene so that a subsequent semi-active decoding of the gene will yield the improved schedule.

The hillclimber is best described using two neighbourhoods, $\mathcal{N}_{hc,feasible}$ and $\mathcal{N}_{hc}$. The neighbourhood $\mathcal{N}_{hc}$ includes moves that can render the schedule infeasible, so the moves in $\mathcal{N}_{hc}$ are considered candidates for $\mathcal{N}_{hc,feasible}$, which is a subset of $\mathcal{N}_{hc}$ that contains only feasible moves. The $\mathcal{N}_{hc}$ and $\mathcal{N}_{hc,feasible}$ neighbourhoods were originally developed by Dell'Amico and Trubian [39][1] who used them in a tabu search algorithm. Mattfeld chooses them after comparing to three hillclimbers based on other neighbourhoods in a computational study. We shall start by defining the $\mathcal{N}_{hc}$ neighbourhood; how $\mathcal{N}_{hc,feasible}$ is found from $\mathcal{N}_{hc}$ will be made clear later.

[1] Neither Dell'Amico and Trubian [39] nor Mattfeld [77] make an explicit distinction between $\mathcal{N}_{hc}$ and $\mathcal{N}_{hc,feasible}$. However, in our opinion the distinction improves the readability of the method.

block structure	small block	block begin	block end
	(o_1, o_2)	$(o_1, o_2, SM(o_2))$	$(PM(o_1), o_1, o_2)$
permutations	$(o_2, o_1)^{\heartsuit}$	$(o_2, o_1, SM(o_2))^{\heartsuit}$	$(PM(o_1), o_2, o_1)^{\heartsuit}$
		$(o_2, SM(o_2), o_1)^{\dagger}$	$(o_2, PM(o_1), o_1)^{\dagger}$
		$(SM(o_2), o_2, o_1)^{\dagger}$	$(o_2, o_1, PM(o_1))^{\dagger}$

Figure 3.10: *The moves in $\mathcal{N}_{hc}$. For critical blocks of length two only the move labelled "small block" is tried, while for blocks of length three or more all the moves labelled "block begin" and "block end" are tried in the beginning and end of the block. Moves that can lead to infeasible schedules are marked †, while safe moves are marked ♡.*

$\mathcal{N}_{hc}$ is made up of moves at the beginning or end of critical blocks in the schedule. It can be proven, [77], that a hillclimbing move reversing one arc in the schedule graph can only improve makespan if it is made at the beginning or end of a critical block. If o_1 and o_2 are two consecutive operations in a critical block, $\mathcal{N}_{hc}$ includes all permutations of $(PM(o_1), o_1, o_2)$ and $(o_1, o_2, SM(o_2))$ in which o_1 and o_2 are reversed. Since a move can only improve makespan if it takes place at the beginning or end of a critical block, it is safe to disregard moves in which $PM(o_1)$ and $SM(o_2)$ are both critical. This leads to the moves shown in figures 3.10 and 3.11.

Depending on the structure of the critical block, two cases exist. Blocks of length two are too short for the moves labelled "block begin" and "block end", so only the move labelled "short block" is performed on them. For blocks of length three or more, the moves labelled "block begin" are tried at the beginning of the block, while the moves labelled "block end" are tried at the end of it.

The makespan after a move can be estimated using local considerations in the schedule graph by calculating the length of the longest path from S to D going through the nodes affected by the move. In the following, unprimed variables represent values as they are before the move, while primed variables represent values after the move. In order to avoid confusion, the names of the nodes are not changed in the following, so e.g. the node called $SM(o_2)$ before the move is also called $SM(o_2)$ after the move, despite of it no longer being the machine successor of o_2.

The length of the longest path found will be a lower bound on makespan after the move. Consider the "small block" move of figure 3.11, $(o_1, o_2) \rightarrow (o_2, o_1)$. After the move, the head of o_2 can be calculated

$$h'(o_2) = \max(h(PM(o_1)) + \tau_{PM(o_1)}, h(PJ(o_2)) + \tau_{PJ(o_2)}).$$

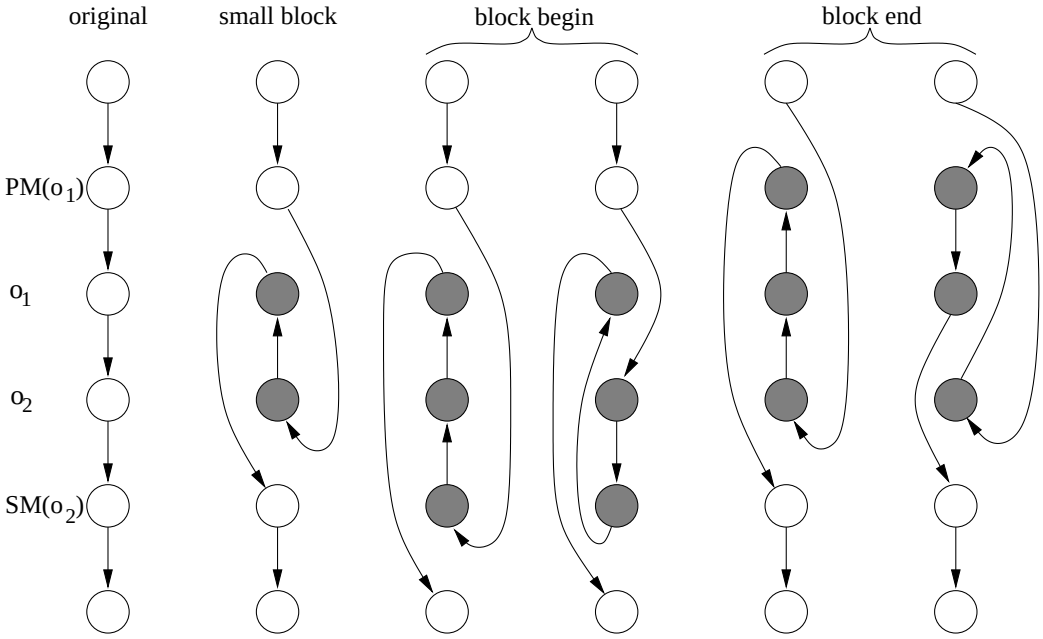

Figure 3.11: *Graph drawings of the moves in $\mathcal{N}_{hc}$. Machine sequences have been drawn, while job sequences have been left out for clarity. The nodes taking part in the move are shaded.*

In the same way, the head of o_1 after the move will be

$$h'(o_1) = \max(h'(o_2) + \tau_{o_2}, h(PJ(o_1)) + \tau_{PJ(o_1)}).$$

The tails of o_1 and o_2 after the move can be calculated

$$t'(o_1) = \max(t(SM(o_2)) + \tau_{SM(o_2)}, t(SJ(o_1) + \tau_{SJ(o_1)}))$$

$$t'(o_2) = \max(t'(o_1)) + \tau_{o_1}, t(SJ(o_2) + \tau_{SJ(o_2)})).$$

After the move, the schedule makespan is bounded from below by

$$C'_{max} \geq C_{max,est} = \max(h'(o_1) + \tau_{o_1} + t'(o_1), h'(o_2) + \tau_{o_2} + t'(o_2)). \quad (3.6)$$

Estimates for the other moves can be made in the same way.

It can be shown [77], that reversing one critical arc in a schedule can never lead to an infeasible schedule. For this reason, the moves marked ♡ in figure 3.10 are safe; they always generate feasible solutions. The moves involving the reversal of more than one arc are not safe in this way. Consider the move $(PM(o_1), o_1, o_2) \rightarrow (o_2, PM(o_1), o_1)$. If a path from $SJ(PM(o_1))$ to $PJ(o_2)$ exists, a cycle will

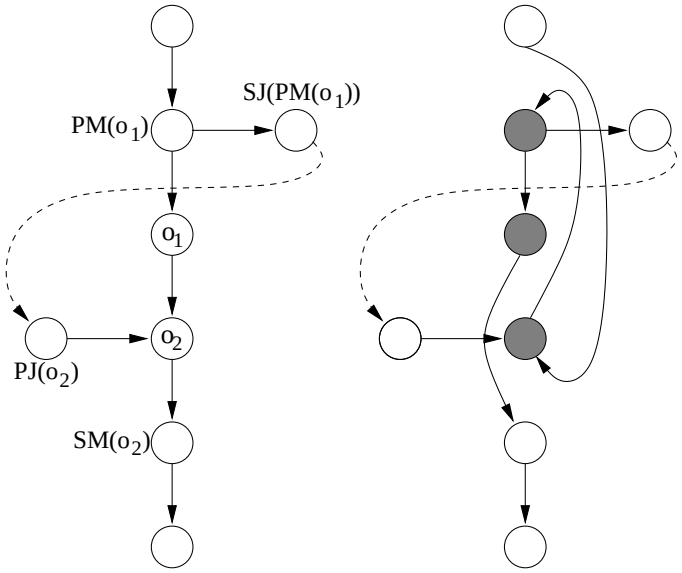

Figure 3.12: *An example of a move including two arc-reversals leading to an infeasible schedule. The left schedule (before the move) is feasible, but after the move* $(PM(o_1), o_1, o_2) \rightarrow (o_2, PM(o_1), o_1)$*, the right schedule contains a cycle because of the path from* $SJ(PM(o_1))$ *to* $PJ(o_2)$*.*

appear in the graph representation, and the schedule will be infeasible, see figure 3.12. Moves that are "unsafe" in this way have been marked with a † in figure 3.10. It has been shown by Dell'Amico and Trubian, [39], that if one of the moves marked † does lead to an infeasible move, the makespan estimate for that move cannot be smaller than the makespan estimate for the corresponding "small block" move. This means that if we always prefer the move with the smallest estimated makespan and prefer the "small block" move when there is tie, we avoid the infeasible moves. In some cases this procedure will also exclude feasible moves from being considered.

The neighbourhood $\mathcal{N}_{hc,feasible}$ guaranteed to be feasible is constructed from $\mathcal{N}_{hc}$ following the procedure outlined above; for every critical block of length two, the small block move of $\mathcal{N}_{hc}$ is included in $\mathcal{N}_{hc,feasible}$. For every critical block longer than two, the $\mathcal{N}_{hc}$ move in the beginning of the block which is estimated to improve makespan the most is included in $\mathcal{N}_{hc,feasible}$, and the same is done for the end of the block.

It has been shown, [39], that the $\mathcal{N}_{hc,feasible}$ neighbourhood does not have the *connectivity* property; there is no guarantee that the optimum schedule can be reached from a given schedule using moves in $\mathcal{N}_{hc,feasible}$. Other neighbourhoods exist that have this property, but these are much larger than $\mathcal{N}_{hc,feasible}$ (implying

```
calculate performance C_max(s) of current schedule s
set continue to true
while (continue) do
   set the priority queue Q to empty
   for s' ∈ N_hc,feasible(s) do
      if C_max,est(s') < C_max(s) then
         insert s' in Q with priority C_max,est(s')
   od
   set continue to false
   while s not updated and Q not empty do
      delete s' in Q with lowest priority
      calculate C_max(s')
      if C_max(s') < C_max(s) then
         update s by setting s = s', set continue to true
   od
od
```

Figure 3.13: *Pseudo-code for the C_{max} hillclimber based on $\mathcal{N}_{hc,feasible}$.*

more computational effort). The $\mathcal{N}_{hc,feasible}$ neighbourhood has been designed to include the most promising moves and exclude moves that may be necessary to guarantee connectivity, but which hold little promise for immediate improvement. Besides, even for a neighbourhood with the connectivity property it is highly unlikely that a random schedule will be taken to the global optimum by a hillclimber, given the complexity of the job shop problem.

The hillclimber used in Mattfeld's GA3 uses a steepest descent strategy, an algorithmic template for it can be seen in figure 3.13.

Operators

The genetic operators used in GA3 are called *Generalised Order Crossover*, GOX, and *Position Based Mutation*, PBM.

The GOX operator was first presented by Bierwirth in [14]. It is inspired by the Order Crossover (OX) operator used on the TSP. The idea in GOX is to transfer the relative order of operations in the parents to the offspring. The operator does not treat the parents symmetrically; one parent is called *the donator*, the other is called *the receiver*. During crossover, a substring is chosen from the donator, randomly located in the string and of length between one third and half of the total length of the genotype. The operations in the receiver that correspond to the substring are located and deleted, and the child is created by making a cut in the

donator	3	1	2	1	2	3	2	3	1
receiver	3	~~2~~	1	~~2~~	~~1~~	1	~~3~~	3	2
child	3	2	1	2	3	1	1	3	2

Figure 3.14: *GOX example, the substring does not wrap around the end-points.*

receiver at the position where the first operation of the substring used to be and inserting the donator substring there. This has been exemplified in figure 3.14 for two genes for a 3×3 problem. The chosen substring has been underlined in the donator and the child, and the operations deleted in the receiver have been marked.

In case the donator substring wraps around the end-points of the donator genetic string, a different procedure is followed. The operations in the receiver that correspond to the operations in the donator substring are still deleted, and the donator substring is inserted in the receiver in the same positions it occupies in the donator (at the ends). The procedure is exemplified in figure 3.15.

donator	3	2	2	1	1	3	2	3	1
receiver	~~2~~	~~3~~	1	3	1	~~1~~	~~3~~	2	2
child	3	2	1	3	1	2	2	3	1

Figure 3.15: *GOX example, the substring wraps around the end-points.*

The GOX operator includes an implicit mutation. Consider the relative positioning of the second "3" and the second "1" in the two parent chromosomes of figure 3.14. In both cases the second "1" precedes the second "3". In the child this is not the case, there the second occurrence "3" precedes the second occurrence of "1". If o_{12} and o_{32} are to be processed on the same machine, this reversal of ordering means that despite of o_{12} getting processed before o_{32} in both parents, o_{32} will be processed before o_{12} in the child. In this way, the GOX operator is disrespectful, since traits present in both parents need not be present in the children.

Mattfeld chooses the GOX operator for crossover after comparing it to two other operators, termed *Generalised Position Crossover*, GPX, and *Precedence Preserving Crossover*, PPX[2]. The PPX operator has been designed to respect the processing order of operations as much as possible. If o_a precedes o_b in both parents this will always be the case in offspring as well. In this sense, PPX is the most respectful crossover operator conceivable for this problem. The GPX operator is very similar to the GOX operator, but it is designed to respect the absolute positions of the genes to a higher degree than GOX does. The operators are compared

[2]In [77], Mattfeld uses the name *Generalised Uniform Crossover*, GUX instead of PPX. Here the name PPX will be used, since this name is better established in literature.

in terms of how well they convey phenotypical traits (precedence relations among operations) in the parents to the offspring, and how well the fitness of offspring correlates to the fitness of parents. In terms of conveying phenotypical traits, the PPX operator is found to be superior to the two other operators, while in terms of fitness correlation the GOX and GPX operators are found to perform equally well, while the PPX operator seems inferior. This is surprising, since fitness is expected to follow the phenotypical traits. In an experiment involving a simple GA and the `ft10` problem, Mattfeld [77] concludes that the GOX and GPX operators find solutions of comparable quality, while the PPX operator is inferior. Since GOX at the same time manages to keep population diversity at a higher level than GPX, GOX is concluded to be the superior operator.

Because theoretical considerations [97] imply that generally a crossover operator should be as respectful as possible, it is quite surprising that GOX is found to be superior to PPX. In a study by Bierwirth, Mattfeld, and Kopfer [17] the PPX operator is also compared to GOX and GPX. In this study it is concluded that PPX is superior to GOX in terms of the achieved fitness if easy problems and simple decoders are used. However, this result does not contradict the finding that for difficult problems GOX performs better than PPX.

The superiority of GOX over PPX on hard problems has also been confirmed by us. This was done by replacing GOX with PPX in GA3. The result holds even if parameters are changed to allow mutation to influence the search more when PPX is used. For easy problems it is the other way around; on these problems the experiments indicated that the algorithm using PPX was able to find the optimum faster than the algorithm using GOX.

The unexpected success of the GOX operator is probably due to the fact that for difficult problems, GAs often have problems with premature convergence on local optima. Apparently the implicit mutation of GOX is capable of avoiding this problem to a higher extent than PPX. Intuitively, the same effect should be achievable by using PPX with a higher mutation-rate, or a more severe mutation, but the experiments done for this thesis have not been able to confirm this.

The mutation operator used in GA3, *Position Based Mutation*, PBM, works by deleting a randomly picked position, and inserting its value at a random position in the gene. This mutation operator is chosen after comparing it to the operators *Order Based Mutation*, OBM, which swaps the values of two random positions, and *Swap Based Mutation*, SBM, which swaps the values of two random adjacent positions. In the comparison it is found that the SBM operator does very little change to the schedules, OBM does quite substantial changes, while PBM is somewhere in between. Mattfeld argues that PBM is probably preferable, since it does a significant amount of change to the schedules, while the mutated schedules are still highly correlated to the original schedules.

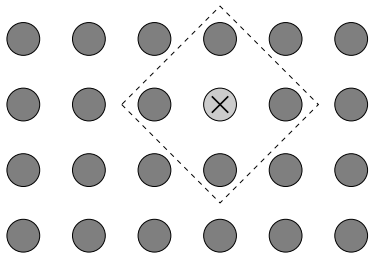

Figure 3.16: *Local neighbourhood in the grid structured population. When the individual marked "x" breeds by crossover, the other parent is selected among the four individuals surrounding it.*

Population structure and selection

In order to avoid premature convergence, GA3 uses the diffusion model, where the population is arranged in a toroidal grid structure. Individuals are only allowed to breed locally, which slows down the spread of genetic material in the population. The grid size used in the experiments is 10×10.

Breeding takes place in generations. Except for offspring that are not accepted and individuals that are "sleeping" (see the next section), the entire population is replaced. When an individual breeds by sexual reproduction, the other parent is found between the four nearest neighbours in the grid, see figure 3.16. The least fit individual among the four neighbours is located, and the other parent is chosen randomly where the probability of each individual is proportional to the objective function value of the worst individual in the neighbourhood minus the objective value of the individual (recall that the objective function is to be minimised). This excludes the worst individual among the neighbours from being a parent, except if all the neighbours have the same objective value, and makes the selection favour the best individual even if all the individuals in the neighbourhood are very close to the same objective value. The individual created in this way is inserted in the next generation, where it is placed in the grid in the centre of the neighbourhood.

A new individual is only accepted into the new population if it is not very bad compared to the individual it replaces. A new individual with makespan $C_{max}(child)$ is allowed to replace its parent with makespan $C_{max}(parent)$ if

$$C_{max}(child) < C_{max}(parent) + 0.1(C_{max}(parent) - LB),$$

where LB is a lower bound on the optimal makespan for the problem. The lower bound is found by calculating the total processing time on each machine and of each job. The largest value found can be used as a lower bound on makespan. This bound is improved by calculating for each machine the earliest starting time and smallest tail possible for the last operation on the machine.

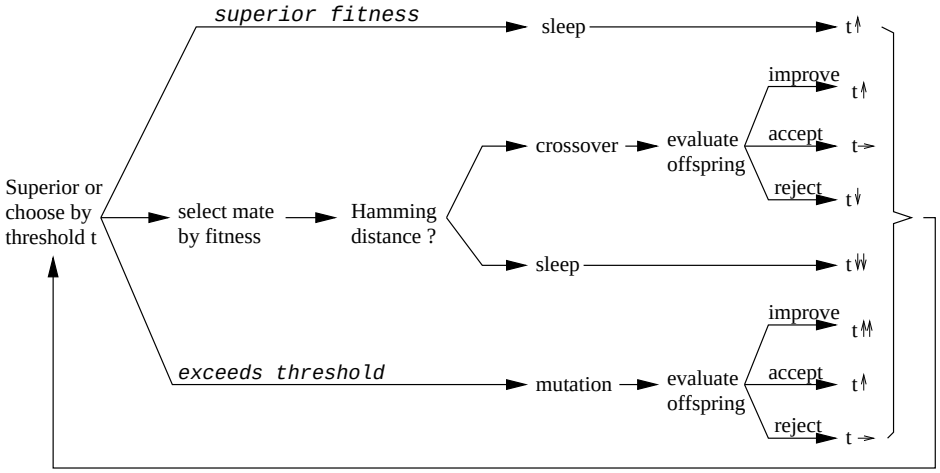

Figure 3.17: *Schematic drawing of the behavioural model of GA3.*

Behavioural model

The grid structured population and local mating scheme counters global premature convergence at the expense of increased inbreeding in the algorithm. In GA3 the local loss of genetic diversity following inbreeding is prevented by adaptive control of crossover and mutation. This is done using a behavioural model in which every position in the grid has a state that reflects the history of that position. If a position has a history of successful crossovers, it will tend to prefer breeding by crossover, if it does not have a history of successful crossover it will tend to prefer breeding by mutation. A schematic drawing of the model is shown in figure 3.17.

The model uses a state in the form of a threshold value in the range $[0; 1]$ for every position in the grid. A high threshold value indicates an individual with a preference for breeding by crossover, while a low value indicates a preference for mutation. Every time an individual needs to breed, it is compared to the four individuals in its neighbourhood. If the individual is better than all of them it is put to sleep in order to preserve its valuable genetic material. It is copied unchanged to the next generation, and its threshold is increased "a little" (indicated by the $\uparrow$ symbol in figure 3.17). If the individual is not superior to all of its neighbours, a random number is drawn uniformly from the range $[0; 1]$. If the number drawn is smaller than the threshold of the individual, it has chosen to breed by crossover. Next, it finds an individual to mate with from the local neighbourhood as described in the previous section. It evaluates the relative Hamming distance to this individual, and if the distance is too small (less than 0.01), the crossover is not performed. The threshold of the individual is lowered "somewhat" (as indi-

cated by the $\downarrow\downarrow$ in the figure), since the small distance between the individuals can be taken as an indication of very low genetic diversity. If the relative Hamming distance is not too small crossover is performed, and if the child does not perform much worse than the parent, the child replaces the parent. If the child is better than the parent, the threshold is increased "somewhat" (indicated by the $\uparrow\uparrow$). If it is not better than the parent but still accepted, the threshold is increased "a little". The threshold is lowered "a little" (indicated by the $\downarrow$) if the child is rejected. If mutation is selected when the random number is drawn, the individual is subjected to the mutation operator and depending on the performance of the child the threshold is unchanged ($\rightarrow$), increased "a little" ($\uparrow$), or increased "somewhat" ($\uparrow\uparrow$).

Every time a threshold is increased "somewhat" ($\uparrow\uparrow$), its value is increased by 0.15. If it is increased "a little" ($\uparrow$), its value is increased by 0.05. If a value is decreased "a little", its value is decreased by 0.05, while decreasing it "somewhat" ($\downarrow\downarrow$) means decreasing it by 0.15.

The thresholds in the model are initialised to 1, meaning that all individuals will breed by crossover in the beginning of the run. Later in the run the genetic diversity falls because of convergence and spread of genetic material. As the unsuccessful crossovers start to happen the rate of crossover falls, and mutations become more common.

3.2.3 Re-implementing GA3

Mattfeld's GA3 was implemented for the experiments presented in chapter 5. The implementation was done in regular `C`. During experiments with the implementation it became clear that the algorithm could not reproduce the results reported in [77]. In most situations, there was a small but significant difference between the results reported by Mattfeld and those achieved with the new implementation, usually such that the results reported by Mattfeld were slightly better than those of the new implementation. This caused much checking of code, but no errors were found.

The original implementation of GA3 was obtained from Mattfeld in order to do a rerun of the experiments and inspect the code directly. Because of changes to the `gcc` compiler and the `LEDA` software library that the original implementation makes heavy use of, it was impossible to make the original implementation run. The difference between the original and the new implementation was never found.

During experiments it was found that the performance of the new implementation of GA3 could be improved by using a hillclimber climbing the $\mathcal{N}_1$ neighbourhood, which was working "on top" of the $\mathcal{N}_{hc}$ hillclimber used after decoding. The $\mathcal{N}_{hc}$-hillclimber was run on every $\mathcal{N}_1$-neighbour of the schedule, until no further improvement could be made. This new hillclimber was only used on the best individual of the last generation of the GA. Static scheduling results for the new

implementation of GA3 can be seen in appendix G, where for each problem the makespan is found in the row labelled “active” and the column labelled “P”.

3.3 Other kinds of scheduling algorithms

In this section other scheduling methods than evolutionary algorithms will be outlined. Since these methods are not the focus of this thesis, only a very brief introduction will be given for each method.

Good introductions to heuristic scheduling algorithms can be found in [23, 62, 83].

3.3.1 Branch and bound algorithms

Branch and bound algorithms are enumerative search procedures based on the construction of a tree of partial solutions. The tree contains the entire set of feasible solutions in the leaves, while nodes inside the tree represent partial solutions. How the algorithm traverses the search tree is governed by a branching scheme and a bounding scheme. The branching scheme of the algorithm decides which operations can follow a given node, while the bounding scheme is used to limit the search by pruning large parts of the tree before they are searched explicitly. The bounding scheme uses a global upper bound on the best solution possible (usually an initial upper bound is constructed using a heuristic before the search begins), and for each node a local lower bound on the quality of solutions in the subtree of the node. All nodes with a lower bound higher than the current upper bound are disregarded in the search, since their subtrees cannot contain a solution better than the current best. Once the search reaches a leaf with a better solution quality than the current upper bound, the upper bound is updated and the search continues elsewhere in the tree. This process continues until all leaves in the tree have been visited or pruned.

The construction of good upper bounds for the nodes is crucial for the efficiency of branch and bound methods, since this is what keeps the algorithm from having to traverse the entire search tree. Usually this is done using graph-theoretic considerations and by calculating upper bounds on one-machine problems derived from the partial solutions. Branching is usually performed on the graph representation of the schedule/problem, in which a branching operation consists of selecting a disjunctive arc pair from E and generating two new partial solutions, one for each possible orientation of the arc in E_s. Often the branching schemes involve complicated reasoning about the partial schedule to identify the best choice for the next branch. For examples of this kind of branch and bound algorithms, see [23, 26].

Branch and bound algorithms have proven very useful for small to medium sized problems, especially since they guarantee finding the global optimum and provide proofs of optimality. According to [62] they are not useful for large problem instances because of the excessive running time.

3.3.2 Tabu search

The tabu search meta-heuristic is an iterative search procedure, usually attributed to Fred Glover [49]. The basic idea in tabu search in combinatorial optimisation is to let a solution move around in the search-space, picking the next solution from a neighbourhood. The algorithm keeps a memory of previously visited solutions, and the search is usually not allowed to revisit solutions already in the memory. Often this is implemented by having a *tabu list* of forbidden moves. Every time a move is made, the inverse move is added to the tabu list, where it stays for a number of moves. When a move is made, usually the best solution in the neighbourhood not forbidden by memory is picked. Since the tabu list stores moves and not solutions, it is usually necessary to allow the algorithm to implement moves in the tabu list, if they satisfy an aspiration criterion. This search strategy often leads to the solution "rolling" from one local optimum to the next in the search-space. Tabu search has proven very efficient for a number of combinatorial optimisation problems.

Tabu search methods have been applied to scheduling problems by many authors, see e.g. [23, 39, 86, 87, 88]. A particularly efficient algorithm for makespan job shop problems was published by Nowicki and Smutnicki in [86]. Their algorithm uses a simple and small neighbourhood, containing the moves reversing two critical operations at the beginning or end of a critical block in the schedule (the moves marked by ♡ in figure 3.10), except that only moves belonging to one critical path are considered. This neighbourhood is not connected, meaning that given a feasible solution, there is no guarantee that an optimal solution can be reached by a sequence of moves from the neighbourhood. It is much smaller than neighbourhoods used in many other tabu search algorithms for the job shop problem, meaning that moves can be applied much faster. The algorithm employs an algorithm working on the graph representation for fast evaluation of makespans after moves. The authors also propose a method termed *back jump tracking*, in which good solutions found during the run are stored along with their tabu lists. The algorithm later returns to these solutions and uses them for starting points in new runs of the search. In experiments, the authors demonstrate the algorithm to be very fast and at the same time produce very good solutions. In a recent study, [111], the Nowicki and Smutnicki algorithm was concluded to still be state-of-the-art.

3.3.3 The shifting bottleneck heuristic

The shifting bottleneck heuristic was first published by Adams et al. in [2]. The heuristic takes advantage of the fact that a one machine scheduling problem with release dates and due dates (despite being NP-hard) can be solved to optimality very quickly using the algorithm published by Carlier in 1982. The shifting bottleneck heuristic works by relaxing the problem into a number of one machine subproblems that are solved one at a time. In its basic form, the shifting bottleneck heuristic uses the cycle

1. Bottleneck identification.
2. Optimising the processing order of the bottleneck machine while keeping processing orders on the other machines fixed.
3. Re-optimising the processing orders of critical machines previously sequenced, one at a time.
4. If there are any unsequenced machines, go to 1.

The bottleneck identification in step one is done by considering each unsequenced machine subproblem and calculating a lower bound on the solution to the problem for each subproblem. The unsequenced machine leading to the highest lower bound is then picked as the most important remaining bottleneck. The cycle keeps un running until all machines have been sequenced and no further re-optimisation of individual machines is possible. In the most basic form, the above procedure is carried out once. In more advanced and time-consuming variants, the procedure outlined above is incorporated into a partial enumeration scheme, in which the choice of the next bottleneck (step 1 above) is incorporated into a beam search-like tree.

The shifting bottleneck heuristic was among the first really efficient approximation algorithms published for the job shop problem, since at the time it was published research was focused on priority dispatch rules and complete enumeration methods. It was also the first algorithm published to optimally solve the `ft10` problem, which had remained unsolved for more than 20 years.

3.3.4 Other heuristics

A multitude of other heuristics have been applied to the JSP with varying degrees of success.

The first approximation algorithms proposed for job shop scheduling were *priority dispatch rules (PDRS)*, in which the basic idea is to assign priorities to the jobs or operations in the problem and to construct the schedule based on a

greedy schedule builder such as the algorithms of figures 3.7 or 3.9. Systems of this kind are probably the most prevalent kind found in real applications today. PDRS are easy to implement and have a very low computational cost, but often produce highly suboptimal solutions.

Other heuristic scheduling algorithms include simulated annealing [18], ant systems [79] and others.

3.4 Benchmark Problems

The benchmarks used in this thesis come from [43], [69], [107] and [108]. They are all well established benchmarks, studied intensely in scheduling research. The problems are the following:

- `ft10,ft20`: Published by Fisher and Thompson in [43]. The `ft10` and `ft20` are difficult, small to medium-sized problems. The `ft10` problem is the most widely used single job shop benchmark problem; virtually all published algorithms have been tested on this problem. Fisher and Thompson also published a problem called `ft06`, but it has not been used in the experiments for this thesis, since the operation processing times come from another interval than all the other problems used.

- `la01-la40`: Published by Lawrence in [69]. The problems come in eight different sizes, ranging from 10×5 to 30×10. Most of the problems are not difficult, while some of the larger instances are quite hard.

- `swv01-10`: Published by Storer et al. in [107]. The original test suite consists of 20 problems, but in the experiments used in this thesis, only the first 10 have been used because the last 10 are quite large leading to large computation times for the algorithms proposed in chapter 5. The problems `swv01-10` are of two different sizes and considered hard.

- `ta01-40`: Published by Taillard in [108]. The original test suite consists of 80 problems, some of which are very large, but again the large problem instances were too time-consuming for the algorithms proposed in chapter 5, so only the first 40 problems were used.

The benchmarks span a wide range of problem difficulties and sizes. The problem sizes are listed in table 3.1. For all the benchmarks the processing times of the operations are integer and fall in the range $\{1, \ldots, 100\}$. The problems are downloadable from the OR-library at `http://www.ms.ic.ac.uk/info.html`.

None of the benchmarks have due dates, so in order to do the experiments involving tardiness of chapter 5, the `ft` and `la` problems were augmented with due

problem	size
ft10	10×10
ft20	20×5
la01-05	10×5
la06-10	15×5
la11-15	20×5
la16-20	10×10
la21-25	15×10
la26-30	20×10

problem	size
la31-35	30×10
la36-40	15×15
swv01-05	20×10
swv06-10	20×15
ta01-10	15×15
ta11-20	20×15
ta21-30	20×20
ta31-40	30×15

Table 3.1: *The sizes (number of jobs×number of machines) of the benchmark problems.*

dates. This was done by for each problem generating a random active schedule, and setting the due date of each job to its completion time minus 5% (loose problems, labelled $\sigma = 0.95$), 10% (medium problems, labelled $\sigma = 0.90$) and 15% (tight problems, labelled $\sigma = 0.85$).

Since no standard benchmarks for stochastic job shop scheduling are available, the static benchmarks discussed here were used as starting points for the experiments on stochastic scheduling presented in chapters 5 and 6.

Chapter 4

Introduction to Stochastic Scheduling

The previous chapter was on deterministic scheduling problems. It was assumed that every aspect of the problem was known when the schedule was made. Real world scheduling problems are not like that. In the real world things go wrong: Machines break down, deliveries get delayed, and workers get sick. Furthermore, real world scheduling usually is a process of sustained pursuit; jobs arrive continuously over time, and from time to time the schedule has to be modified to accommodate new jobs.

It is very important that real world scheduling systems are able to properly handle these stochastic aspects of scheduling. This not only means being able to handle the events once they have happened. A good scheduling system for a stochastic environment should also be able to generate schedules that are prepared for the events; robust or flexible schedules.

The focus of this chapter and chapters 5 and 6 is on generating robust and flexible schedules, mainly for schedules facing breakdowns. The present chapter will introduce terminology, discuss the important question of how to measure stochastic performance, and present previous work.

4.1 Introduction

A stochastic scheduling problem usually holds a set of possible scenarios, of which only one will come true. It is the task of the scheduler to create a schedule which will perform well for the scenario that comes true. Only when it has turned out which scenario came true can the true merit of the schedule be evaluated. The problem is of course, that at the time processing is to begin the scheduler does not know which scenario will come true.

Before processing can begin, the scheduler has to make some sort of decision, despite the uncertainty of the environment. At the very least, he[1] has to decide which operations are to start running at the beginning of processing; he has to create the *initial schedule*. In most cases he will make a more detailed initial schedule than just which operations are to be processed first. Usually (but not always, see section 4.7.3 and [114]), the scheduler starts by creating an initial schedule (or initial processing order), which covers processing of all jobs known to the scheduler. This initial schedule can later be modified if need be.

When the scheduler has created the initial schedule, processing can begin. Usually, processing will keep on following the initial schedule or processing order until an incident that calls for a change of schedule happens. Often the scheduling problem is made up in such a way that there is a "normal" mode of operation in which processing carries on the way it is supposed to. This "normal" mode of operation stretches intervals of time that are punctuated by incidents such as the appearance of new jobs, breakdowns or unknown operation processing times becoming known. These incidents reflect changes to the processing environment the scheduler was considering when he made the schedule. In the following, such changes in the scheduling environment are called *events*. An event occurs at a specific time during the execution of a schedule. A *scenario* consists of a number of events happening while processing a schedule. A special scenario is when no events happen, that is the environment remains unchanged during the entire schedule execution.

Whenever an event changes the environment, the scheduler is faced with a *rescheduling problem*. Solving a rescheduling problem means changing the part of the schedule not yet implemented to incorporate the change, while respecting the part of the schedule already implemented. The schedule as it looked prior to the event is termed the *preschedule*. The preschedule includes the part of the schedule that will not be implemented due to the event; it is the schedule the scheduler was going to implement if the environment did not change. In the following, a solution to a rescheduling problem is called a *new schedule*. The concepts of preschedules, new schedules, events and scenarios are exemplified in figure 4.1.

When solving a rescheduling problem, the scheduler is often trying to find a schedule of the optimal quality (minimal cost), but there may be other goals to consider:

- Sometimes the new schedule should be as similar to the preschedule as possible. This can be the case if *schedule nervousness* is to be kept at a low level. A schedule is called nervous if it is experiencing frequent, large changes. Reasons to keep schedule nervousness at a low level could be to

[1] In this thesis, the word *he* will be used for people of unspecified gender. This is shorter and sounds better than the more politically correct *she or he*.

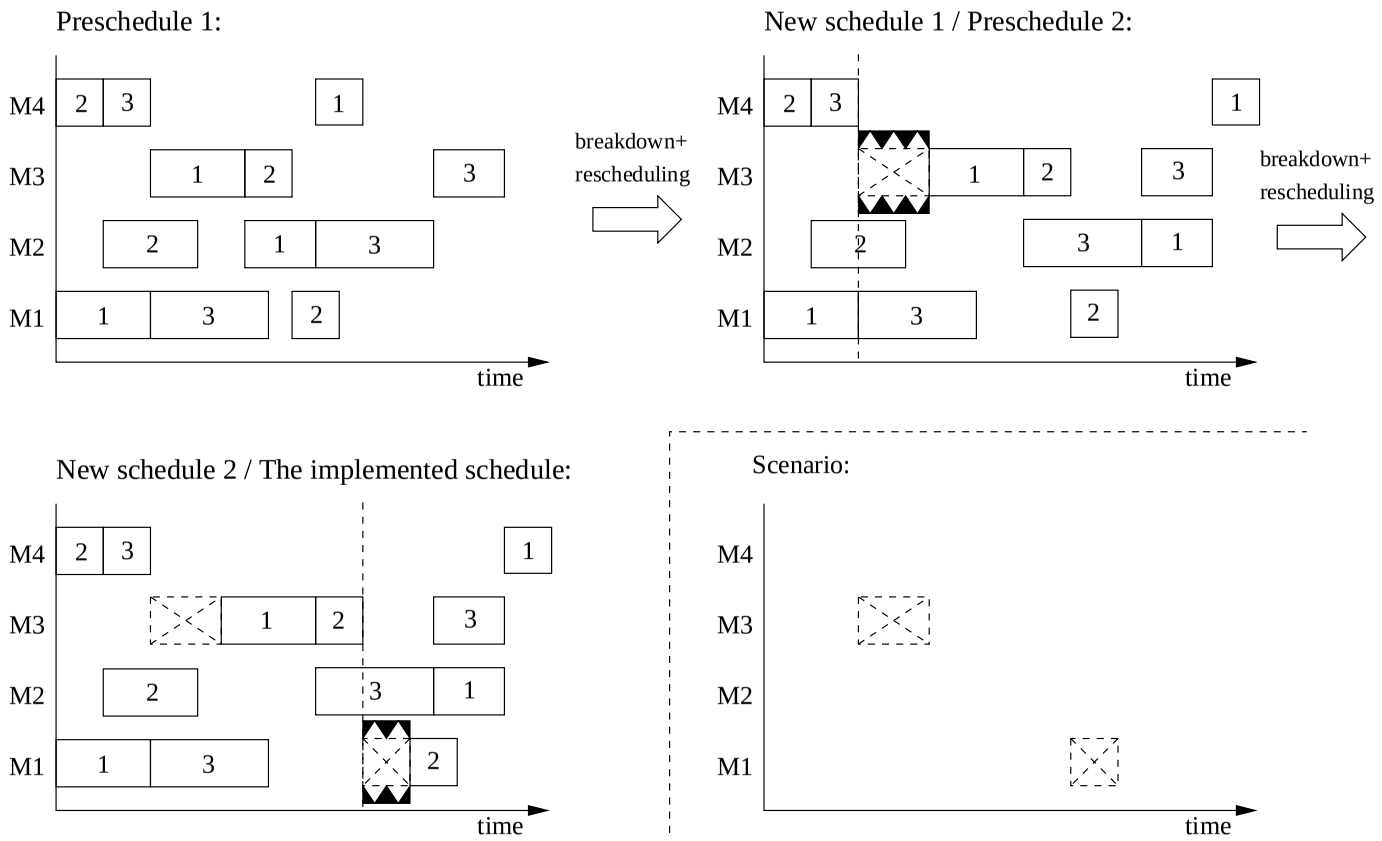

Figure 4.1: ***Top left:*** *The preschedule.* ***Top right:*** *A breakdown occurs (marked by the black triangles). The schedule is changed to a new schedule. This new schedule is also a preschedule for the next event.* ***Bottom left:*** *Another breakdown happens. The schedule is changed to another new schedule, which becomes the implemented schedule since no further events happen.* ***Bottom right:*** *The scenario for this course of events is the two breakdowns indicated on the chart.*

avoid upsetting workers by changing their hours constantly, avoid costs of recalling workers from vacation and avoid costs of ordering new raw materials at short notice.

- In some cases it is important to solve the rescheduling problem quickly. If a machine has suddenly broken down, the whole processing floor may be idle, waiting for the new schedule.

- The robustness or flexibility quality of the new schedule may be important.

Rescheduling can be done in several ways. Since a job shop rescheduling problem is also a job shop problem, it can be done by a standard scheduling algorithm, solving the rescheduling problem from scratch as proposed in [15, 42]. Another approach is to use the preschedule as a guide. In the case of a breakdown, the processing order of the preschedule can be used in the new schedule, generating a new schedule which is a replica of the preschedule in which operations affected by the breakdown have been delayed to accommodate the breakdown. This kind of rescheduling is usually termed *right-shifting*. The preschedule processing order can also be used as a starting point for more sophisticated rescheduling methods, such as hillclimbing. In this thesis rescheduling methods trying to find the best new schedule from a set of schedules are said to do *rescheduling using search*.

Another aspect of rescheduling is how to generate the preschedule. The quality of the best new schedule and the hardness of the rescheduling problem depends on the preschedule as well as on the nature of the unexpected event. For this reason is seems natural to take into consideration the possibility of unexpected events already when the preschedule is generated. One preschedule may generate a rescheduling problem that is easy to solve and has a low cost optimal new schedule, while the same breakdown for another preschedule may generate a hard rescheduling problem with a high cost optimal new schedule. A preschedule that generates easy rescheduling problems with low cost solutions is clearly preferable to a preschedule that leads to hard rescheduling problems with high cost solutions, all other things being equal.

4.2 Measuring stochastic performance

How to measure the stochastic performance $P(s)$ of a schedule s depends on what the scheduler wants to achieve. In the following a number of performance measures will be defined. Every performance measure $P(s)$ reflects some aspect of the cost of implementing schedule s, meaning that a low performance measure indicates a good schedule.

In some cases the scheduler wants to minimise the average cost of implementing the schedule, where the average is taken over all the scenarios. This has the drawback that the probability distribution of scenarios needs to be known. This kind of performance is defined

$$P_{average}(s) = \sum_{b \in B} p(b)C(s,b), \tag{4.1}$$

where B is the set of possible future scenarios, $p(b)$ is the probability of scenario b, and $C(s,b)$ is the cost if scenario b happens when implementing schedule s. This kind of performance is called *average performance* or *expected performance*. Average performance should be used in situations where the scheduler is not very concerned with avoiding the consequences of a very serious breakdown, but wants to minimise the average cost of the processing plant over some period of time.

In some cases, the scheduler wants to minimise the consequences of the worst case scenario. This is the case for very critical applications such as the scheduling of aircraft queueing to land in an airport. The performance measure is defined as

$$P_{worst\ case}(s) = \max_{b \in B} C(s,b). \tag{4.2}$$

This kind of performance is termed *worst case performance*. As remarked in [68], worst case scheduling tends to generate very conservative schedules, since the schedules must guard against every possible contingency.

A way of dealing with the worst case scenario in a less conservative fashion is *worst deviation performance*. The performance measure is defined

$$P_{deviation}(s) = \max_{b \in B}[C(s,b) - C^*(b)], \tag{4.3}$$

where $C^*(b) = \min_{s \in S} C(s,b)$ is the cost of implementing the schedule which is optimal for scenario b. The idea in worst deviation performance is to create a schedule which is close to the optimal performance for every scenario in B. In order to calculate $P_{deviation}(s)$, the best possible performance $C^*(b)$ must be known for all scenarios $b \in B$, which makes this performance measure quite difficult to work with.

A related measure is *relative worst deviation performance*, which measures the maximum percentage distance to the optimum

$$P_{relative}(s) = \max_{b \in B} \left[\frac{C(s,b) - C^*(b)}{C^*(b)} \right]. \tag{4.4}$$

As for worst deviation performance, knowledge of $C^*(b)$ is needed for all $b \in B$. Note that since $\frac{C(s,b)-C^*(b)}{C^*(b)} = \frac{C(s,b)}{C^*(b)} - 1$, minimising $P_{relative}(s)$ is the same as minimising $\max_{b \in B} \frac{C(s,b)}{C^*(b)}$.

4.2.1 Relations between the performance measures

A result due to Kouvelis and Yu, [68] states that if an optimal worst case schedule s_{wc}, worst deviation schedule s_{dev}, or relative worst deviation schedule s_{rel} is known, a limit on how much the expected performance of this schedule deviates from the expected performance of the optimal average case schedule s_{avg} can be found.

For an optimal deviation performance schedule the argument is as follows. Let s_{dev} denote a schedule that minimises $P_{deviation}(s)$ and s_{avg} denote a schedule that minimises $P_{average}(s)$. A bound on $P_{average}(s_{dev})$ can be established as follows:

$$\begin{aligned} P_{average}(s_{dev}) &= \sum_{b \in B} p(b)\, C(s_{dev}, b) \\ &\leq \sum_{b \in B} p(b)(P_{deviation}(s_{dev}) + C^*(b)) \\ &= P_{deviation}(s_{dev}) + \sum_{b \in B} p(b) C^*(b) \\ &\leq P_{deviation}(s_{dev}) + P_{average}(s_{avg}), \end{aligned} \tag{4.5}$$

where the last inequality holds because $\sum_{b \in B} p(b) C^*(b)$ is a trivial lower bound on $P_{average}(s_{avg})$. In the same way, for schedules s_{rel} minimising $P_{relative}(s)$ and s_{wc} minimising $P_{worst\ case}(s)$, the bounds

$$P_{average}(s_{rel}) \leq (1 + P_{relative}(s_{rel}))P_{average}(s_{avg}) \tag{4.6}$$

and

$$\begin{aligned} P_{average}(s_{wc}) &\leq P_{worst\ case}(s_{wc}) \\ &\leq P_{worst\ case}(s_{wc}) - \min_{b \in B} C^*(b) + P_{average}(s_{avg}) \end{aligned} \tag{4.7}$$

can be established.

4.3 Robustness and flexibility

The concepts of robustness and flexibility are central to this thesis. Most of this chapter and the following two chapters are dedicated to algorithms for robust or flexible schedule generation. The definitions used here for these concepts are the following:

- Robustness of a schedule. A schedule expected to perform well (have a low cost) relative to other schedules, when facing some set of scenarios and when right-shifting is used for rescheduling is said to be *robust*.

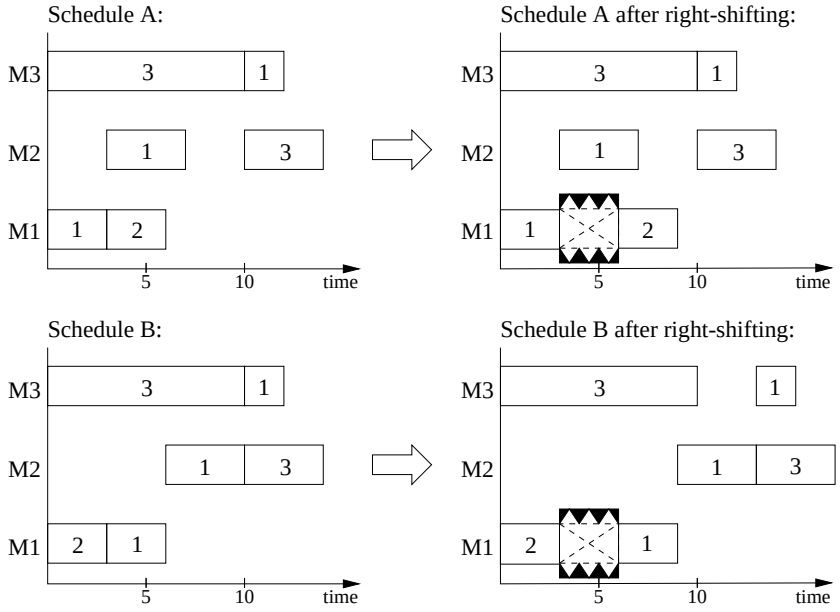

Figure 4.2: *Example of schedule robustness. For the breakdown indicated by the triangles, schedule A is more robust than schedule B.*

- Flexibility of a schedule. A schedule expected to perform well (have a low cost) relative to other schedules, when facing some set of scenarios and when some rescheduling method using search is used is said to be *flexible with regard to that rescheduling method*.

An example of schedule robustness can be found in figure 4.2. In the figure, Gantt-charts of two schedules have been drawn on the left. On the right, they have both been subjected to the same breakdown (indicated by the black triangles), and right-shifting rescheduling has been performed. Since schedule A has a shorter makespan after right-shifting than schedule B, schedule A can be said to be more robust than schedule B when facing the breakdown.

An example of schedule flexibility can be found in figure 4.3. The only difference between schedules A (top left) and B (bottom left) is the ordering of the first operations on machine M1. The schedules have both been subjected to the same breakdown. If rescheduling using complete reordering of the processing sequence is used, schedule A can be rescheduled to a schedule with makespan 20 (top right of the figure). For schedule B, reordering the processing order creates a new schedule of makespan 23 (bottom right of the figure). Thus, when facing the breakdown indicated on the figure, schedule A is more flexible than schedule B with regard to rescheduling using complete reordering of the processing sequence.

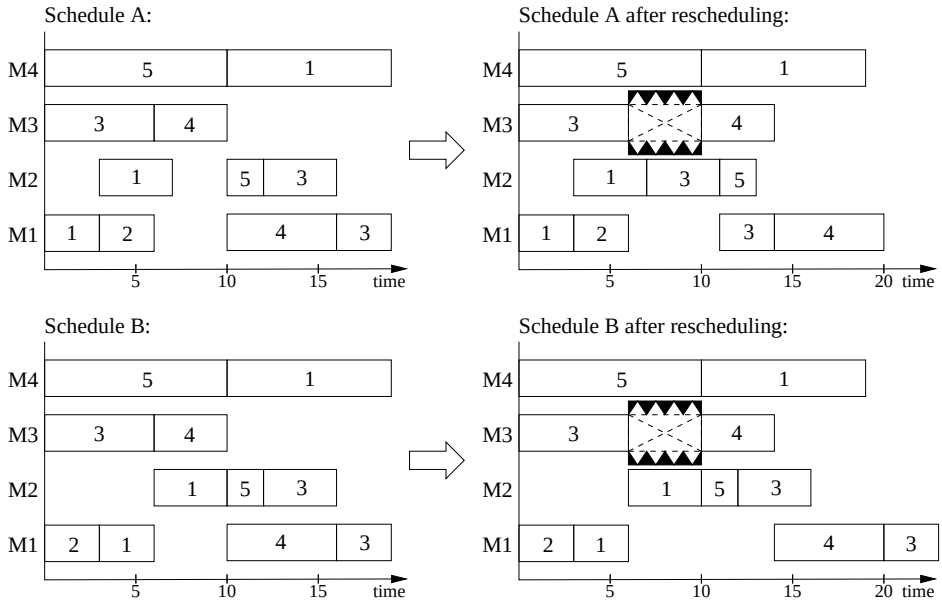

Figure 4.3: *Example of schedule flexibility. For the breakdown indicated by the triangles, schedule A is more flexible than schedule B with regard to a rescheduling method allowing complete reoptimisation of the processing sequence.*

The term ”robust” seems to be well established in literature, while flexibility was introduced in [21, 63]. Despite of many authors using the term ”robust”, the meaning of the word is somewhat diffuse and changes from author to author. In literature focused on stochastic or dynamic problems, the term “robust” usually refers to a solution that is somehow prepared for changes in the environment (robust or flexible in the terms of this thesis). Literature focused on more standard optimisation using meta-heuristics uses the term to denote an algorithm that performs well on a broad range of problems.

Most authors agree that a robust solution is a solution prepared for the uncertainty of unknown future events. In some problems the uncertainty lies in the fact that the problem instance may change, (e.g., [68, 72]), while for other problems the uncertainty may come from an imprecise implementation of the solution found (due to imprecision in e.g. machinery [13, 20, 109]). In this thesis, robustness refers to changes in the problem instance.

Sometimes “robust” has been used to describe a solution which does not change its performance much if a change happens in the environment. This definition of robustness is not very useful, since it is very easy to generate a schedule being “robust” in this way; simply create a schedule with lots of idle-time. The idle-time can act as a buffer to absorb delays in processing. Such a schedule will have a very poor performance, and will in fact be completely useless. For this

reason, the definitions of robustness and flexibility used here are based on the absolute performance of the schedules, not the change in performance.

The distinction used in this thesis between robust and flexible solutions is relatively new. In some literature (e.g. [56, 70]), no distinction is made between robust and flexible schedules; the term "robust" is used to indicate a solution which is prepared for future events using some kind of rescheduling (not necessarily right-shifting). In our opinion, the distinction between robustness and the different kinds of flexibility is important. A schedule that performs well relative to other schedules after an unexpected event and one kind of rescheduling need not perform well relative to other schedules if another kind of rescheduling is used. Consider the schedules of figure 4.3. Schedule A is more flexible than schedule B for complete rescheduling. However, there is no difference between the two schedules in terms of robustness; if right-shifting rescheduling is used, they will both end up with a makespan of 23. Since the notions of a schedule being robust, and flexible with regard to various rescheduling methods may be independent, different concepts are needed in order be able to make an adequate description.

4.4 Events and rescheduling problems

In this chapter, stochastic scheduling problems with three different kinds of events are treated. An event can be the breakdown of a machine, rendering the machine unable to do processing for a certain amount of time (due to the definition of the job shop problem, the machine needs to become operational again, otherwise the problem may be unsolvable), the appearance of new jobs, or the fixation of an uncertain operation processing time. In real world scheduling, a scheduling problem will often be prone to several different kinds of events (e.g. appearance of new jobs and breakdowns), but the scheduling problems here will all be subject to only one type of event.

In a problem with uncertain operation processing times, usually the processing time of an operation o is known to fall in a certain interval $\tau_o \in \{\tau_o^{min} \dots \tau_o^{max}\}$. In this case, an event happens every time the actual value of an operation processing time becomes known (e.g. when the operation finishes processing). In order to make this kind of problem fit the framework presented in this section, we will allow release times and initial setup times in this kind of problem to also vary in time intervals $r_i \in \{r_i^{min} \dots r_i^{max}\}$ and $u_j \in \{u_i^{min} \dots u_i^{max}\}$.

Whenever an event happens, something has occurred in the scheduling environment that calls for the attention of the scheduler. A breakdown has occurred, an uncertain operation processing time has been fixed, or new jobs have appeared. When the event has happened, the scheduler is free to change the part of the schedule not yet implemented to adapt it to the new environment. If the event happens

at time t, the scheduler is free to reschedule any operation o that has not started strictly earlier than t. A rescheduling problem P' is generated from the original problem P and the preschedule s in the following way (primed variables refer to P', unprimed variables refer to P):

1. The rescheduling problem P' has the same number of machines m and jobs n as the original problem P. The number of jobs may later be increased.

2. The time the event happens is denoted t.

3. For each machine M_k, the initial setup time of machine M'_k is set to $u'_k = \max(t, c, u_k)$, where c is the completion time of the latest operation commenced on M_k prior to time t (c is $-\infty$ if no such operation exists). In the case of a problem with uncertain operation times, u'_k is set to a time interval $\{u_k^{min'} \ldots u_k^{max'}\}$ by taking into account that c is not a known value, but a range of values.

4. For each job J_i, a job J'_i is constructed by setting the release time to $r'_i = \max(t, c, r_i)$, where c denotes the completion time of the latest operation of J_i to begin processing prior to t (c is $-\infty$ if no such operation exists). In the case of a problem with uncertain operation times, r'_i is set to a time interval $\{r_i^{min'} \ldots r_i^{max'}\}$ by taking into account that c is not a known value, but a range of values. For each operation o_{ij} in J_i, the same operation exists in J'_i if the starting time of o_{ij} in s is equal to or greater than t. The technological constraints of J'_i are equal to the technological constraints of the corresponding operations in J_i, and the due date of J_i is set to $D'_i = D_i$.

5. If we are dealing with uncertain operation times, we are done. If we are dealing with breakdowns or arriving jobs, P' now corresponds to P, except that part of s has been implemented and cannot be changed. The relevant part of P' can now be changed to reflect the change in the environment.

4.4.1 Machine breakdowns

If the event happening is a machine breakdown, step 5 above consists in updating the initial setup time of the broken down machine M_{broke}. This is done according to the characteristics of the breakdown: the *breakdown time* (the time the breakdown occurs), the *breakdown duration* and the machine affected by the breakdown. If the breakdown happens while the machine is processing an operation o, there are two different interpretations of how the breakdown affects the operation being processed. The two interpretations, in this thesis called *preemptive*

and *non-preemptive* breakdowns, are not equivalent, since they lead to different rescheduling problems.

Some authors have used the assumption that the operation has to restart processing from scratch; the processing done prior to the breakdown is wasted. In this case, the initial setup time of the broken machine is set to $u'_k = t + \tau_{breakdown}$, where $\tau_{breakdown}$ denotes the duration of the breakdown. Since the operation o disrupted by the breakdown needs to restart processing from scratch, the scheduler is free to reschedule it, so an operation o' equal to o is added to the beginning of the job J'_i that corresponds to the job J_i to which o belongs. This kind of breakdown is denoted a *preemptive breakdown*.

Other authors use the assumption that the processing of the operation is delayed by the duration of the breakdown, but that the processing done prior to the breakdown is not wasted. In this case, the initial setup time of the broken machine is set to $u'_k = \tau_{breakdown} + c_o$, where c_o is the end of processing time the operation has in the preschedule. With respect to the calculation of tardiness, flow-times etc., the end of processing time of the operation is considered to be $\tau_{breakdown} + c_o$. Since the operation continues processing on the broken machine immediately after the breakdown, there is no need to add o to P'. The release time of job J'_i, is set to $r'_i = \tau_{breakdown} + c_o$. This kind of breakdown is denoted a *non-preemptive breakdown*.

For preemptive as well as non-preemptive breakdowns, if a machine breaks down while it is not processing any operation, the initial setup time is set to $u'_k = t + \tau_{breakdown}$.

These ways of creating a rescheduling problem following a breakdown are based on an assumption that at the time the breakdown happens, the scheduler knows exactly when the broken machine will become operational again. This is probably not the case in most real world situations; in real problems the scheduler will probably have some idea about when the machine will become operational again, but he will not know the precise time. Also, in real world problems the estimate of when the machine will become operational again will probably be more and more precise as time progresses; right after the breakdown the estimate may be very imprecise (it may be "anywhere from 20 minutes to 2 days"), while at a later stage when the problem has been located it may be more precise ("between 4 and 5 hours"). These aspects of the problem are ignored in the present formulation.

4.4.2 The appearance of new jobs

If the event happening is the appearance of new jobs, these are added to P' in step 5, while the rest of P' remains unchanged.

4.4.3 Calculation of flow-times

The calculation of flow-times after a rescheduling problem has been generated and solved does not follow the formulation of section 3.1.1 completely. Instead of using the release times r_i' of the rescheduling problem P', the release times r_i of the original problem P need to be used, since the release times in P' are not the true release times of the jobs, but artificial release times designed to make the rescheduling problem reflect the breakdown.

4.5 Complexity of stochastic job shops

It was argued in section 3.1.2, that the deterministic job shop problem is NP-hard in the strong sense for the number of machines $m \geq 2$. The robust or flexible stochastic counterpart of the problem should be expected to be at least as hard: on top of the combinatorial explosion involved in solving the deterministic problems, there is the added complexity of handling the possible future scenarios. Except for very small and simple stochastic problem formulations, any hope of finding (provably) optimal robust or flexible schedules seems very optimistic.

It has been shown that some problems which are polynomially solvable for deterministic job shops are NP-hard when viewed as stochastic problems. In [68] it is shown that the single machine robust summed flow-time job shop problem is NP-hard for uncertain processing times, right-shifting rescheduling for optimality criteria $P_{worst\ case}(s)$, $P_{deviation}(s)$ and $P_{relative}(s)$, even if there are only two scenarios. The deterministic counterpart of this problem is polynomially solvable by the shortest processing time rule, [4].

The stochastic two machine permutation flow-shop problem with makespan criterion has also been demonstrated to be NP-hard. The problems are known to be polynomially solvable in the deterministic case [45], while in [68] a variant of the problem with uncertain processing times and right-shifting rescheduling is shown to be NP-hard for the $P_{relative}(s)$ and $P_{worst\ case}(s)$ performance criteria.

4.5.1 Complexity of rescheduling

Since a rescheduling problem is a job shop problem, its complexity could be expected to be in the NP-class, since standard job shop problems are NP-hard. This turns out to be correct, but the result is not as straight-forward as it might seem at first. A rescheduling problem P_{resch} is a slightly perturbed version of another (possibly larger) job shop problem $P_{original}$. If we make the assumption that an optimal preschedule $s_{original}$ is known for $P_{original}$, maybe $s_{original}$ can be used to locate the optimal solution s_{resch} to P_{resch}.

It can be shown that finding the makespan optimal solution to a breakdown rescheduling problem P_{resch} generated from the original problem P is NP-hard, even if the makespan optimal solution to P is known, if the number of machines m is greater than or equal to 3. This statement holds for preemptive as well as non-preemptive breakdowns.

The proof of this statement is by reduction of the 0-1 knapsack problem to a rescheduling problem, and can be found in appendix C. The result is shown to also hold for the maximum tardiness, summed tardiness and number of tardy jobs optimality criteria, and for rescheduling problems created by adding a new job to the problem.

The complexity of rescheduling problems with fewer than three machines is currently unknown. If we extend the job shop problem and allow a job to have several operations on the same machine, the proof of appendix C can easily be modified to cover problems with two machines. If this extension is not made, the complexity of the rescheduling problems is uncertain at this stage.

4.6 Similarity between schedules

In some situations it is desirable for the new schedule after rescheduling to be as similar to the preschedule as possible. This is the case when there is a cost associated with a high schedule nervousness. For instance, it may upset workers if their working hours are often changed at short notice.

In order to be able to compare schedules, several distance measures have been defined. In [57] Hart et al. define a distance measure based on a Hamming distance calculation on the processing sequences of the machines. Each position in the processing sequences contributes a 0 if the two sequences agree, 1 if they disagree. The contributions from all the positions are summed to get the distance between the two schedules. Other distance measures on schedules found in literature are the measure introduced in [22], the absolute and relative Hamming distance measures of section 3.1.4 (originally introduced in [77]), and measures based on differences between start-times of operations [70].

Most of the previously proposed distance measures work on the processing sequence level. They consider the processing sequence of the operations, while ignoring the timetabling information of the schedules. The overlap measure defined below is an attempt to measure the likeness between two schedules as seen on the processing floor. The measure is defined such that two identical schedules will have an overlap measure of one, while two completely different schedules will have an overlap measure of zero.

For two job shop schedules s_1 solving P_1 and s_2 solving P_2, where P_1 and P_2 are identical problems except that the availability of machines need not be the

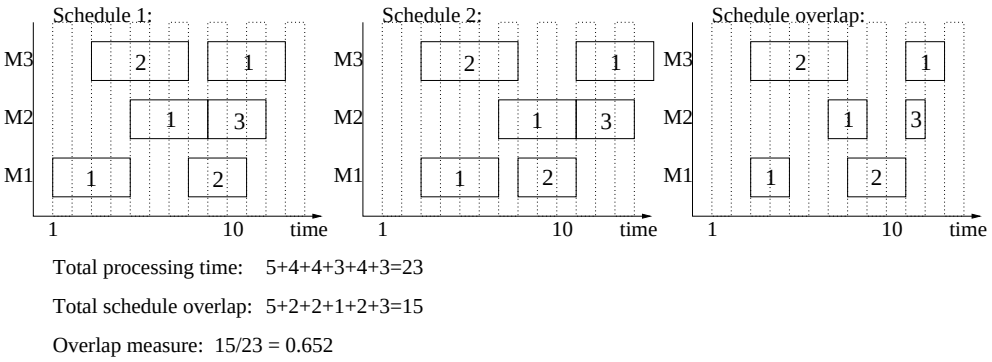

Figure 4.4: *Example of schedule overlap calculation. The different patterns represent different jobs.*

same, the *schedule overlap* $O(s_1, s_2)$ is defined to be

$$O(s_1, s_2) = \frac{\sum_{o \in T} oo(s_1, s_2, o)}{\sum_{o \in T} \tau_o}, \tag{4.8}$$

where T is the set of operations present in the problems. The *operation overlap* $oo(s_1, s_2, o)$ for the operation of job j on machine m between the schedules s_1 and s_2 is

$$\begin{aligned} oo(s_1, s_2, o) = \quad & \max\big(\min(t_{end}(s_1, o), t_{end}(s_2, o)) - \\ & \max\big(t_{start}(s_1, o), t_{start}(s_2, o)\big), 0\big), \end{aligned}$$

and $t_{start}(s, o)$, $t_{end}(s, o)$ and τ_o denote the starting time, ending time and processing time of the operation o in schedule s.

The idea in the schedule overlap measure is to measure the amount of overlap between the processing of operations on the shop floor. If $O(s_1, s_2)$ is 0.5, it means that for half the total processing time of all the operations, the machines on the shop floor are operating on the same operations in s_1 and s_2. An example of the calculation of the overlap measure can be found in figure 4.4.

4.7 Previous work

In this section previous work on robust and flexible job shop scheduling will be described. The first four subsections contain brief descriptions of the most important results, the work by Kouvelis and Yu [68], Leon, Wu and Storer [72], Wu, Byeon and Storer, [114], and Branke and Mattfeld [21]. The last subsection holds approaches by a number of other authors.

4.7.1 Robust discrete optimisation

Kouvelis and Yu [68] describe robust decision making for a number of discrete optimisation problems, including a few scheduling problems.

The book concentrates on worst case, worst deviation and relative worst deviation performance. The scheduling problems treated are all of the uncertain operation processing time type.

Besides the theoretical results presented earlier in section 4.2.1 and in the beginning of section 4.5, Kouvelis and Yu present branch and bound algorithms and a few heuristic approaches for generation worst deviation robust schedules for a single machine problem with total flow-time criterion and a two machine flow shop with makespan criterion.

The authors manage to show that for the worst deviation formulation of robustness, for any schedule there will be a scenario which belongs to the set of *extreme point scenarios*, which will maximise the deviation from optimality for that schedule. An extreme point scenario is a scenario, where the processing time of each operation is equal to the minimum or maximum value attainable for that operation. This drastically reduces the number of scenarios that need to be considered in the algorithms.

For the single machine problem, the foundation of the branch and bound algorithm presented is the fact that for the deterministic variant of the problem (in which the operation processing times are known), the optimal processing sequence can be easily found using the SPT (shortest processing time) rule, [45]. This fact is used along with a simple rule that can determine the optimal processing order of two operations in certain cases to limit the space searched by a branch and bound algorithm. The algorithm also uses an efficient way of calculating the worst deviation from optimality for a given schedule.

The branch and bound algorithm for the two machine flow-shop problem, bases the calculation of bounds on Johnson's algorithm, [45], which solves to optimality the deterministic version of the problem. This algorithm also uses an efficient way of calculating the worst deviation from optimality for a given schedule.

For both problems, a number of heuristic algorithms to generate robust schedules are demonstrated. Experiments on a large number of relatively small randomly generated problems demonstrate that the branch and bound algorithms have reasonable running times, and that some of the heuristics generate solutions which are very close to optimality. Compared to schedules produced by standard scheduling algorithms, the schedules generated by the heuristics and the branch and bound algorithms are clearly superior, both in terms of worst deviation performance and average performance.

The results presented by Kouvelis and Yu are very impressive, but unfortu-

nately they only apply to one and two machine problems. The catch in the algorithms presented, branch and bound algorithms as well as heuristics, is the use of polynomially solvable deterministic versions of the problems (the SPT rule and Johnson's algorithm), which is clearly not an option in more general formulations in which the deterministic versions of the problems will be NP-hard. Also, the efficient calculation of worst deviation performance and the application of simple rules to infer relations between some of the operations will be much harder in a more general setting.

4.7.2 A slack-based approach

In [72], Leon, Wu and Storer develop a robustness measure for a makespan job shop which is experiencing frequent disruptions in the form of breakdowns. The authors are interested in improving average performance when right-shifting rescheduling is used. They also do experiments to investigate whether the method they propose can improve average performance when no breakdown happens but noise is imposed on operation processing times.

Schedules facing one breakdown

Call the makespan of schedule s after a number of breakdowns and rescheduling $C'_{max}(s)$. According to [72], the average makespan $E[C'_{max}(s)]$ of schedule s after a number of disruptions can be calculated

$$E[C'_{max}(s)] = E[\delta(s)] + C_{max}(s), \tag{4.9}$$

where $C_{max}(s)$ is the makespan of s without disruptions, and $\delta(s)$ is the makespan increase caused by the breakdowns. Based on this, Leon et al. state that a robustness measure $R(s)$ can be defined

$$R(s) = r\, E[C'_{max}(s)] + (1 - r)\, E[\delta(s)], \tag{4.10}$$

where $r \in [0; 1]$ is a parameter used to tune the robustness measure.

The authors state an exact expression for $E[\delta(s)]$, which holds under the assumption that (i) only one breakdown happens, (ii) breakdowns happen on a specific machine, (iii) the density functions $g(d)$ and $f(T)$ of breakdown duration d and breakdown arrival time T are known, (iv) right-shifting is used for rescheduling.

The *slack* $w(o)$ of an operation o in a schedule s is defined as the time by which the operation can be delayed without worsening the makespan of the schedule if the processing order is kept. The slack of operation o can be calculated as

$$w(o) = C_{max} - h(o) - \tau(o) - t(o), \tag{4.11}$$

where $h(o), t(o)$ and $\tau(o)$ are the head, tail and processing time of o in the schedule. In [72] preemptive breakdowns are used, which means that the makespan delay resulting from a breakdown at time T with duration d can be calculated

$$\begin{aligned} D(T,d) &= \max(0, \max(0, T+d-h(o)) - w(o)) = \\ &\quad \max(0, T+d-h(o)-w(o)), \end{aligned} \tag{4.12}$$

where o is the operation affected by the breakdown. The average makespan delay can then be calculated

$$E[\delta(s)] = \int_{T=0}^{\infty} \int_{d=0}^{\infty} D(t,d) f(T) g(d) \, \mathrm{d}T \, \mathrm{d}d, \tag{4.13}$$

which is relatively easy to do[2].

Schedules facing a number of breakdowns

Removing the assumption that all breakdowns happen at a specific machine is straightforward. Unfortunately, removing the assumption that only one breakdown happens is very difficult. Instead of doing this, Leon, Wu and Storer state three different expressions expected to correlate to $\delta(s)$ after a number of breakdowns. All of the expressions are based on the slack of operations in the schedules, equation (4.11).

The breakdowns simulated in the experiments reported in [72] are defined to happen with exponentially distributed inter-arrival times with mean 400 time-units, while breakdown duration d is fixed at 50 time-units. After a correlation study, the expression found to correlate best to $\delta(s)$ for ten 20×5 problem instances is the average operation slack

$$RD3(s) = \frac{\sum_{o \in O} w(o)}{|O|}. \tag{4.14}$$

Since $RD3(s)$ is correlated to $\delta(s)$, the authors expect

$$RM3(s) = C_{max}(s) - RD3(s) \tag{4.15}$$

to correlate to $C'_{max}(s)$. The minus in equation (4.15) is due to a negative correlation between $\delta(s)$ and $RD3(s)$.

[2] In [72] equation (4.13) is stated using integrals, since Leon, Wu and Storer are using a definition of breakdowns that can lead to non-integer breakdown times. This formulation has been kept here, but for equation (4.13) to fit the framework of this thesis properly, it should be reformulated using sums.

Based on equation (4.9) the robustness measure is defined:

$$Z_r(s) = r\,RM3(s) + (1-r)\,RD3(s). \tag{4.16}$$

The authors construct a GA capable of minimising $Z_r(s)$ or $C_{max}(s)$. For each problem they do experiments in which the GA minimises $C_{max}(s)$ and experiments in which it minimises $Z_r(s)$ for values $r = 1$, $r = 0.85$ and $r = 0$. The performances after a series of breakdowns $C'_{max}(s)$ are compared.

$r = 1$: The experiments show a significant decrease in average makespan after a series of breakdowns $C'_{max}(s)$ if $Z_{r=1}(s)$ is minimised rather than $C_{max}(s)$.

$r = 0.85$: Minimising $Z_{r=0.85}(s)$ leads to schedules with a lower $C'_{max}(s)$ than if C_{max} is minimised, but a higher $C'_{max}(s)$ than if $Z_{r=1}(s)$ is minimised. The performance variability $\delta(s)$ is lower than when $Z_{r=1}(s)$ is minimised. This is to be expected from the definition of $Z_r(s)$, since decreasing r puts more weight on $RD3(s)$, which is known to correlate to $\delta(s)$.

$r = 0$: Schedules found by minimising $Z_{r=0}(s)$ show very poor performance after a series of breakdowns. This is to be expected since setting $r = 0$ places no significance on $C_{max}(s)$ or $C'_{max}(s)$ in the objective. The performance variability $\delta(s)$ is lower than in the $Z_{r=1}(s)$ and $Z_{r=0.85}(s)$ experiments.

The experiments show that using the $Z_r(s)$ robustness measure can improve performance significantly when the schedules are facing breakdowns. They also indicate that if the minimisation of makespan variability is very important, values of r less than 1 can be helpful, but this comes at the cost of increasing makespan $C'_{max}(s)$.

Comparing the raw makespan performance $C_{max}(s)$ (without breakdowns) of $Z_{r=1}(s)$ optimised schedules and $C_{max}(s)$ schedules the authors find that the increased robustness of the $Z_{r=1}(s)$ schedules comes at a cost of a slightly increased raw makespan.

Imposing noise on processing times

The authors also do experiments where no breakdowns happen, but in which noise is added to operation processing times according to a uniform distribution. After creating the schedule the processing time of each operation is set to

$$\tau'_o = \tau_o + 2\,\tau_o\,d\,(u_o - 0.5),$$

where the u_o are drawn from a uniform $(0, 1)$ distribution and d is a parameter determining the amplitude of the noise. The makespan of the schedule with

the new processing times but the processing order found for the unperturbed processing times is calculated. Leon, Wu and Storer do experiments for $d = 0, 0.1, 0.25, 0.5, 1.0$. They find that for $d \geq 0.25$, $Z_{r=1}(s)$ optimised schedules are significantly better than $C_{max}(s)$ schedules, while for very low d values the opposite is true, due to the superior initial makespan of the $C_{max}(s)$ optimised schedules.

4.7.3 Preprocess first schedule later

A fundamentally different approach for stochastic scheduling is developed by Wu, Byeon and Storer in [114]. In this paper, it is argued that since the exact conditions on the shop floor are not known a priori, it makes sense to postpone the actual scheduling until the conditions are known. In other words, the position of an operation in the processing sequence is not decided until the operation starts processing. The authors work on a weighted tardiness job shop problem and develop a scheduling heuristic they term *preprocess first schedule later* (PFSL), in which a small critical set of decisions are made in a preprocessing step, and the actual scheduling is done later using a dispatching heuristic.

The preprocessing is done on the graph representation of the scheduling problem, which is decomposed into a series of subproblems to be solved during schedule execution. The decomposition is done by dividing the schedule operations into sequentially ordered subsets. During processing the operations belonging to subset i subset must precede operations belonging to subset $i + 1$, so the preprocessing step fixes a number of disjunctive arcs in the graph representation of the problem. This process is exemplified in figure 4.5.

The problem decomposition is done using a branch and bound approach, in which a tree of partial decompositions is searched. In the search, lower and upper bounds on the best performance achievable with the partial decompositions are calculated. The upper bounds are calculated using a dispatching heuristic to generate processing orders for the schedule subproblems, while lower bounds are made using graph-theoretic considerations. When working on deterministic problems, the algorithm returns the schedule with the lowest cost found during the run. In the stochastic experiments, the algorithm returns the problem decomposition with the lowest robustness measure, where the robustness measure is defined as a weighted average of lower and upper bounds.

In experiments the authors demonstrate that for a number of small deterministic problems, the proposed approach using decomposition to two subproblems generates better schedules than a well-performing iterative improvement heuristic.

For stochastic shop conditions in which the processing times of operations are perturbed by small to moderate levels of noise, the authors demonstrate that when considering average schedule cost, the PFSL heuristic performs much better

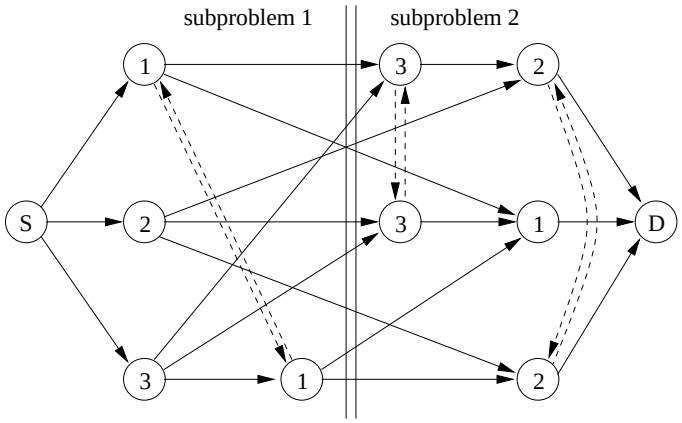

Figure 4.5: *Example of decomposing a job shop problem into two subproblems. The problem of figure 3.6 has been decomposed into two subproblems. This has resolved a large number of disjunctive arcs. For a larger problem instance more disjunctive arcs will be left undecided by the decomposition.*

than generating a fixed schedule using the iterative improvement heuristic and somewhat better than using a dynamic dispatch scheduler to decide the processing sequences during schedule execution. When a very high level of noise is added to the processing times, the PFSL heuristic seems inferior to the dynamic dispatch scheduler. The fact that the dynamic dispatching scheduler beats the PFSL in very chaotic conditions is not surprising, since if the noise gets large enough, the actual operation processing times have very little correlation to the processing times on which the decomposition was based.

The ideas and algorithms presented in [114] are very interesting and seem to work really well on small problems. However, they may not be applicable on large scale problems, since they are based on a branch and bound algorithm. Furthermore, the branch and bound algorithm working on the problem decomposition may be even more susceptible to problem sizes than ordinary branch and bound algorithms working on schedules, since in the latter great care is often taken to create branching schemes that lead to tight bounds on schedule performance. This idea is probably harder to implement on an algorithm working at the decomposition level, since the set of allowable branches is limited.

4.7.4 A flexibility measure for new arriving jobs

In [21] Branke and Mattfeld study a job shop problem with new random jobs arriving continuously over time. The authors consider the known part of the problem fixed and deterministic and solve it using a GA. When new jobs arrive, the part of the schedule already implemented is removed and the new jobs are added

to the problem, which is solved again using the GA. In this way the problem is decomposed into a series of deterministic subproblems. This strategy was first proposed by Raman and Talbot [98]. It has been used by a number of authors, [15, 16, 27, 73, 101, 112], and is usually called a *rolling time horizon* approach.

In [21] it is argued that when a rolling time horizon approach is used it is important to avoid idle machines early in the schedules. This is because idle-time placed early in a schedule is likely to be implemented as idle-time (meaning that more work will have to be done later), while idle-time placed late in the schedule is an asset that can be utilised later for accommodating new jobs.

The authors work on mean tardiness job shops, and define a flexibility measure which is a linear combination of the tardiness and a penalising term for idle-time early in the schedule. Since early idle times can also be avoided by using non-delay schedules, a tunable decoder similar to the modified Giffler-Thompson algorithm of figure 3.9 is used. Thus, the idle-time in the schedules is controlled by two parameters: the weighting of the idle-time punishment in the fitness evaluation, and the idle-time allowed by the schedule builder.

The interdependency between the two parameters is studied in experiments. In the study both parameters are observed to influence the mean tardiness of the schedules, and the level of interdependency between them is found to be low. The experiments demonstrate that the flexibility of the schedules can be improved substantially by explicitly taking early idle-times into account in the fitness evaluation.

The approach in [21] is remarkable because it is almost directly transferable to real world scheduling, and because it shows that in some cases substantial improvements can be reached using very simple ideas. It also emphasises the relevance of "blind" meta-heuristics such as genetic algorithms and tabu search in the scheduling domain. The objective function used is not complicated, but because it is composed of several independent terms it will be very difficult to construct a more problem specific algorithm (i.e. a bottleneck heuristic or a branch and bound algorithm) for it.

4.7.5 Other approaches

Other approaches on robust and flexible scheduling include the work by Al'Harkan [3], in which it is found that for total tardiness job shops facing uncertain processing times with known distributions, average performance can be improved considerably by estimating the performance of each schedule using simulation. The scheduling is done in a genetic algorithm in which fitness evaluation is done by sampling the processing times repeatedly until the average performance is known with a predefined certainty. The method is compared to a method called *probability Gantt charting*, in which fitness evaluation is done in a deterministic

way by replacing the processing time of each operation by a pessimistic estimate. The latter method is found not to work well. A related approach is followed by Ventouris in [112], in which fitness evaluation by sampling operation times is compared to deterministic scheduling and found to decrease the tardiness under stochastic conditions.

An approach to generate robust schedules is given by Hart and Ross in [56], where an artificial immune system (*AIS*) is evolved to solve tardiness job shop problems. The solutions produced by the AIS are observed to be robust in the sense that schedules produced by the AIS to solve a set of similar problems are more similar to each other than schedules produced by a GA scheduler.

In [106], Sotskov et al. consider the problem of calculating *stability radii* for makespan and total flow-time job shops. The stability radius of an optimal schedule is the largest perturbation of operation processing times for which the schedule is guaranteed to remain optimal. The problem studied is interesting because it can reveal the conditions under which an optimal schedule can be expected to be optimal not just for the theoretical problem considered when making the schedule, but also in its practical realization. There are several problems associated with the method. First of all, a globally optimal solution to the scheduling problem needs to be found. This is known to be NP-hard, and very difficult for large problems. Secondly, a very large number of schedules need to be considered when calculating the stability radius of the schedule, making this calculation very time-consuming. Thirdly, the method only specifies how to calculate the stability radius of a given schedule, not how to find a schedule with a high stability radius. This means that in order to use the method in practise, a number of different optimal schedules need to be found. The stability radii of these schedules can then be calculated, and the most stable schedule selected.

Based on studies of real world scheduling systems, McKay et al. [78] argue that when considering disruptions such as breakdowns or preventive maintenance in production systems, the actual disruption (called the *primary event* in [78]) is not the only thing to consider. Since there is a high risk of a secondary event happening at the same machine shortly after the first one, scheduling on the machine should give high priority to operations which are not very sensitive to a secondary event. The secondary event could be another breakdown, or it could be inefficient processing for some time after the disruption. The idea of avoiding scheduling sensitive jobs on the machine immediately after a breakdown is termed *aversion dynamics*. The authors model a one machine production system and compare the performance of a priority dispatch rule using the idea of aversion dynamics to a number of standard priority dispatch rules.

In [70, 71] Leon, Wu and Storer use a game-like approach to the problem of rescheduling after a machine breakdown. This is done in order to create a rescheduling method creating flexible new schedules. A game tree is constructed to eval-

uate the consequences of various choices of new schedules on future performance. The game tree holds two kinds of nodes: nodes representing possible future disruptions on the shop floor (called "*chance nodes*") and nodes representing possibilities for rescheduling (called "*decision nodes*"). The disruptions considered are machine breakdowns, while the rescheduling method used in the decision nodes is a branch and bound algorithm changing the processing order of the disrupted machine. The tree constructed does not represent all possible contingencies (this would be intractable), but rather a sample of contingencies. The rescheduling decisions in the tree are evaluated to identify the decision leading to the lowest expected makespan, while also considering the degree of schedule disruption. In [71] the approach is compared to a system using right-shifting rescheduling and one using complete rescheduling (generation of a new schedule from scratch after the disruption). The game-like approach is found to compare favourably to both of these in terms of schedule quality (makespan) after a number of disruptions and also to the complete rescheduling approach in terms of the amount of schedule disruption done in rescheduling.

A different kind of rescheduling is suggested by Bean et al. in [11], in which breakdowns are considered for a problem taken from the car manufacturing industry. It is proposed after a breakdown to reconstruct part of the schedule to later match up with the preschedule. This match up is done by gradually considering a larger and larger subproblem of the rescheduling problem until a match up is found.

Chapter 5

Neighbourhood Based Robustness for Scheduling

In this chapter, the neighbourhood based robustness approach for job shop scheduling will be presented. First, a few results on robust solutions for continuous function optimisation will be presented, since they form the inspiration for the work presented in this chapter. In section 5.2, the neighbourhood based robustness measures for job shop scheduling will be defined. In section 5.3 it is investigated if the neighbourhood based robustness measure improves robustness and flexibility for job shop schedules under the makespan criterion. Schedules produced by minimising the robustness measure are compared to schedules produced by two standard schedulers, and to schedules produced by minimising the robustness measure defined in [72]. The usefulness of neighbourhood based robustness measures on maximum tardiness, summed tardiness and total flow-time job shops is investigated in sections 5.4-5.6.

The chapter ends in section 5.7, in which it is discussed how and why the neighbourhood based robustness approach works.

5.1 Inspiration from continuous function optimisation

The neighbourhood based robustness approach is inspired by recent work on continuous function optimisation, [13, 20, 109]. In engineering applications there is an imprecision when going from the "ideal" solution calculated by the engineer to the actual implementation of the solution. The imprecision comes from the uncertainty inherent when doing things in the real world. For instance, a steel rod supposed to be 1 meter long may only be 99.9cm in reality, a resistor supposed to have a 1000Ω resistance could be 1023Ω etc. Because of this imprecision, often

the solution actually implemented will not be the solution found by the engineer, but a solution close to it. For this reason, it is desirable for a solution to a problem with this kind of uncertainty to be located on a broad peak in the objective landscape. If this is the case, the solution actually implemented in the real world will still be acceptable, even if it is not exactly equal to the solution the engineer was "aiming for". When deciding which solution to "aim for", a tradeoff arises between the height and the broadness of the peaks. The measurement of peak broadness can be done in a number of ways. For some problems, it is known how much the actual solution can deviate from the ideal one, or the distribution of noise present in the implementation is known. Usually the broadness of the peak is not explicitly calculated, instead the performance is evaluated in a worst case or average case manner, see section 4.2. For worst case performance a method termed *robust convex optimisation* is developed in [13] for certain kinds of convex programs.

Evolutionary algorithms for finding broad peaks are developed independently by Tsutsui and Ghosh in [109] and Branke in [20]. The basic idea in both algorithms is to perturb solutions slightly according to some distribution of noise prior to fitness evaluation. Tsutsui and Ghosh use a genetic algorithm in which after decoding, each phenotype x is perturbed once to $x' = x + \delta$ where δ is sampled from the noise distribution. The fitness of x is calculated from $f(x')$, where f is the objective function. This way of evaluating fitness means that a solution located on a narrow high peak will often be inferior to a solution located on a lower broad peak, since the noise imposed on the phenotype will take the solution from the narrow high peak to a low fitness area, while the solution located on the broad low peak will still be acceptable. An illustration of this kind of fitness evaluation is shown in figure 5.1. Using the schema theorem (see section 2.1.2), Tsutsui and Ghosh are able to show that such an algorithm evolves as if the objective function was

$$F(x) = \int_{-\infty}^{\infty} f(x+\delta)\, q(\delta)\, \mathrm{d}\delta, \tag{5.1}$$

where $f(x)$ is objective function used to evaluate the perturbed phenotypes and $q(\delta)$ is the density function of the noise. $F(x)$ is termed *the effective objective function* in such an algorithm. The fact that the genetic algorithm behaves as if $F(x)$ was the objective function is also demonstrated in experiments. In [20], Branke demonstrates that if the fitness of an individual is calculated as an average of a number of independent perturbations of the phenotype, the average robustness quality of the solutions $F(x)$ is improved.

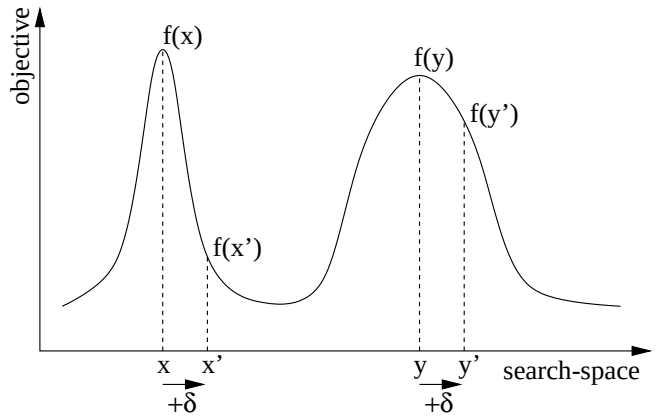

Figure 5.1: *The idea in neighbourhood based robustness. If for some reason the solution is changed, the broad peak may do much better than the narrow peak, since the solutions close to the broad peak are still reasonable solutions (compare* $f(x')$ *to* $f(y')$*, and note that* x' *and* y' *were produced using the same perturbation:* $x' = x + \delta,\ y' = y + \delta$*).*

5.2 Robustness measures for scheduling

Finding robust or flexible solutions as broad peaks in the objective landscape can also be applied to scheduling. The idea in doing this is that if a small set of schedules close to the preschedule is known to be good before an unexpected event, then perhaps one of the schedules can work (partly) around an event that makes the preschedule unacceptable.

The most natural way to proceed is to define a robustness measure which has its minimum on broad peaks in the search-space. Since the search-space of job shop scheduling problems is very different from the continuous functions dealt with in the previous section, equation (5.1) cannot be used right away. Because the focus is on job shops with regular performance measures, the schedules can be considered sequences of operations. Since the space of sequences is discrete, the integral of equation (5.1) must be replaced with a sum. Furthermore, a proximity relation for schedules is needed. The $\mathcal{N}_k$-neighbourhood of section 3.1.4 is a natural choice, so

$$R(s) = \sum_{s' \in \mathcal{N}_k(s)} \phi(s, s')\, C(s') \tag{5.2}$$

is a candidate for a robustness measure for job shop scheduling. In equation (5.2) $C(s')$ is the cost of schedule s' and $\phi(s, s')$ is the weight or probability of schedule s' given schedule s.

The definition of R above contains a few free parameters. First of all the cost function C needs to be decided, but also the size of the neighbourhood k and the

weighting $\phi(s, s')$ must be determined. Since the size of $\mathcal{N}_k$ increases sharply with k, and since evaluating R takes more time the larger $\mathcal{N}_k$ is, so far only robustness measures with $k = 1$ have been tested. Regarding the weighting $\phi(s, s')$, the simple choice of $\phi(s, s') = \frac{1}{|\mathcal{N}_1(s)|}$ has been used.

5.3 Experiments on makespan

For makespan problems, the neighbourhood based robustness measure

$$R_{C_{max}}(s) = \sum_{s' \in \mathcal{N}_1(s)} \frac{1}{\mathcal{N}_1(s)} C_{max}(s') \tag{5.3}$$

is used. A number of questions regarding the measure arise:

1. Is there any difference between optimising this robustness measure and simply optimising the makespan? Maybe the best way of optimising the robustness measure would be to optimise the makespan. This could be the case for a function of local minima in which all of the minima had the same shape.

2. If the robustness measure is optimised, how does the makespan of the solution compare to the makespan of a solution found by optimising makespan? Can solutions that are good in the ordinary sense (they have a low makespan) and have a low robustness measure be found?

3. When faced with machine breakdowns, are schedules optimised for the robustness measure more robust or flexible than schedules optimised for makespan?

The first question is relevant since if there is no difference between minimising C_{max} and $R_{C_{max}}$, there is no need to do any further work on $R_{C_{max}}$-minimal schedules. In that case minimising $R_{C_{max}}$ will be equivalent to ordinary scheduling.

Question two is relevant because if something can be gained from minimising $R_{C_{max}}$ instead of C_{max}, it is relevant to ask if something else is lost. If $R_{C_{max}}$ turns out to improve the robustness or flexibility of the schedules in case of a breakdown, we also need to know if this improvement comes at a cost of decreased performance in case no breakdown is encountered. If question two can be answered with a yes for all problems, it means that we can minimise $R_{C_{max}}$ instead of C_{max} without any cost in the static performance of the schedules.

The last question is probably the most interesting one; the whole idea in minimising $R_{C_{max}}$ is to achieve robust or flexible schedules.

It is hard to answer these questions in an analytical fashion. Doing that requires profound knowledge and understanding of the shape of the makespan landscape, knowledge which is currently not available. Instead, we shall resort to experiments in an attempt to answer the questions. Questions one and two are answered in section 5.3.3, while question three is answered in sections 5.3.6-5.3.12.

5.3.1 Minimising $R_{C_{max}}$

In order to answer the questions in the previous section, algorithms for minimising C_{max} and $R_{C_{max}}$ are needed. For minimising C_{max}, the re-implementation of Mattfeld's genetic algorithm was used. The algorithm was described in details in section 3.2.2. In what follows, this algorithm is called *the C_{max}-GA* or *the makespan GA*.

The same genetic algorithm was modified to minimise $R_{C_{max}}$. This algorithm is called *the $R_{C_{max}}$-GA* or *the robustness GA*. Minimising $R_{C_{max}}$ should be expected to be more time consuming than minimising C_{max}. When evaluating a schedule for C_{max} just a single schedule needs to be examined. When evaluating a schedule for $R_{C_{max}}$, the makespan C_{max} of a set of schedules needs to be calculated.

In order to be able to compare the results produced by the robustness GA and the makespan GA, the two algorithms need to be of comparable efficiency. If the results of a very good makespan GA were to be compared to the results of a very poor robustness GA, the most likely conclusion would be that minimising $R_{C_{max}}$ produces inferior results when compared to minimising C_{max}. The same would probably be true the other way around. In order to make sure the two algorithms are of comparable efficiency, for every C_{max}-specific part of the makespan GA, a comparable $R_{C_{max}}$-specific part needs to be created for the robustness GA. The C_{max}-specific parts of the makespan algorithm are the following:

- The fitness evaluation.
- The calculation of a lower bound. Recall that lower bounds are used in the acceptance criterion for new individuals.
- The C_{max} hillclimber.

For the fitness evaluation, the calculation of C_{max} was simply replaced by a calculation of $R_{C_{max}}$. The lower bound calculation of the C_{max} algorithm was used unmodified in the $R_{C_{max}}$-GA. This works because the lower bound on the minimal C_{max} value is also a lower bound on the minimal $R_{C_{max}}$ value. It is difficult to come up with a simple method for calculating a better bound on $R_{C_{max}}$. The approach is acceptable since the exact parameters for the acceptance criterion were

```
calculate robustness measure R_{C_max}(s)
and makespan C_max(s) of current schedule s
set continue to true
while (continue) do
   set the priority queue Q to empty
   for s' ∈ N_{hc,feasible}(s) do
      if C_{max,est}(s') < C_max(s) then
         insert s' in Q with priority C_{max,est}(s')
   od
   set continue to false
   while s not updated and Q not empty do
      delete s' in Q with lowest priority
      calculate R_{C_max,est}(s') ≈ R_{C_max}(s')
      if R_{C_max,est}(s') < R_{C_max}(s) then
         update s by setting s = s'
         calculate R_{C_max}(s) and C_max(s)
         set continue to true
   od
od
```

Figure 5.2: *Pseudo-code for the $R_{C_{max}}$ hillclimber.*

set somewhat arbitrarily, indicating that a small imprecision in the lower bound may not change the behaviour of the algorithm profoundly. The C_{max} hillclimber was replaced by a $R_{C_{max}}$ hillclimber. The details of this hillclimber are in the next section.

5.3.2 A hillclimber for $R_{C_{max}}$

The $R_{C_{max}}$ hillclimber is a modified version of the C_{max} hillclimber. It searches the same neighbourhood and picks the moves it makes in a way very similar to the C_{max} hillclimber. While considering the moves in $\mathcal{N}_{hc}$, it makes the same estimates on C_{max} as the C_{max} hillclimber, and constructs $\mathcal{N}_{hc,feasible}$ in exactly the same way. Adapting the use of $\mathcal{N}_{hc}$ and $\mathcal{N}_{hc,feasible}$ from the C_{max} hillclimber makes sense, since the minima of $R_{C_{max}}$ can be expected to coincide with minima of C_{max}, which the C_{max} hillclimber has previously demonstrated to find efficiently.

Pseudo-code for the $R_{C_{max}}$ hillclimber can be seen in figure 5.2. The pseudo-code closely resembles the code of the C_{max} hillclimber of figure 3.13. The major difference between the C_{max} hillclimber and the $R_{C_{max}}$ hillclimber is in the `if`

statement at the end of the loop. In the C_{max} hillclimber the makespan of the new schedule s' is evaluated, and the schedule is kept if it improves makespan. In the $R_{C_{max}}$ hillclimber, an estimate $R_{C_{max},est}(s')$ of $R_{C_{max}}(s')$ is made, and if the estimate is seen to improve the robustness measure the new schedule is kept. The estimate $R_{C_{max},est}(s')$ is an upper bound on $R_{C_{max}}(s')$, so every move made is known to improve the robustness measure of the schedule.

Estimating $R_{C_{max}}$

Given a schedule s, what is needed in order to evaluate $R_{C_{max}}(s)$ is the makespan of all schedules in $\mathcal{N}_1(s)$. Constructing all these schedules is very time-consuming, especially if it needs to be done many times inside a hillclimber. Fortunately, given the graph representation of s, estimating the makespan of all schedules in $\mathcal{N}_1(s)$ is straight-forward.

An estimate of the makespan after a $\mathcal{N}_1$-move was made in equation (3.6). Note that although the calculation of $C_{max,est}$ does not hold for infeasible schedules, it is safe to use on $\mathcal{N}_1$-moves, since $\mathcal{N}_1$ only includes feasible moves. This estimate is used for the $R_{C_{max}}$ estimate.

After the $\mathcal{N}_1$-move, the critical path in the schedule s' is either going through one of the nodes that were interchanged in the move, or it is elsewhere in the schedule. In the first case $C_{max,est}(s')$ of (3.6) is equal to $C_{max}(s')$. In the second case the makespan of the original schedule is an upper bound on the makespan of the schedule after the move, since the critical path in the schedule after the move is also in the schedule before the move (it need not be critical in the original schedule). For these reasons

$$C_{max,ub}(s') = \max(C_{max,est}(s'), C_{max}(s)) \tag{5.4}$$

is an upper bound on the makespan $C_{max}(s')$ after the move.

The estimate of the robustness measure can the be calculated

$$R_{C_{max},est}(s) = \frac{C_{max}(s) + \sum_{s' \in \mathcal{N}_1(s) \setminus \{s\}} C_{max,ub}(s')}{|\mathcal{N}_1(s)|}. \tag{5.5}$$

Since $C_{max,ub}(s')$ is an upper bound on $C_{max}(s')$, clearly $R_{C_{max},est}(s)$ is an upper bound on $R_{C_{max}}(s)$. Furthermore, if s is a locally makespan optimal schedule, all of the $\mathcal{N}_1$ neighbours will have a makespan of $C_{max}(s)$ or higher, in which case $R_{C_{max},est}(s)$ becomes equal to $R_{C_{max}}(s)$. Thus, if $R_{C_{max}}$-optimal schedules are coincidental with locally C_{max}-optimal schedules located on broad peaks in the makespan landscape, for $R_{C_{max}}$-optimal schedules the $R_{C_{max},est}$ estimate can be expected to be exact.

prob-lem	semiactive run		active run		robust run	
	best	mean	best	mean	best	mean
la01	688.7	696.6	678.4	695.3	674.4	674.5
la02	684.7	697.1	685.5	696.3	682.7	684.5
la06	932.9	943.5	927.2	938.5	926.0	926.0
la07	899.6	911.2	896.5	908.3	890.9	892.6
la26	1244.4	1256.8	1243.9	1256.0	1231.7	1240.4
la27	1299.2	1313.1	1297.1	1311.6	1282.6	1296.2
la31	1793.2	1802.9	1790.1	1797.4	1784.1	1784.7
la36	1336.3	1358.0	1339.1	1356.3	1327.8	1342.9
ft10	987.7	1005.0	987.3	1004.4	987.2	998.6
ft20	1197.7	1225.6	1197.1	1223.8	1184.7	1203.4

Table 5.1: *The robustness measures of the solutions found in the semiactive, active and robust runs. The averages have been taken over 400 runs.*

5.3.3 Preliminary experiments on $R_{C_{max}}$

The genetic algorithm for minimising C_{max} described in section 3.2.2 produces semi-active schedules. Since the robustness of a schedule may be improved by producing active schedules rather than semi-active ones, a modified version of the algorithm that produces active schedules was made. It works by putting the final schedule through a procedure that leap-frogs operations to make the schedule active. The procedure is the same as the one used for schedule generation in [14]. In what follows, experiments with this algorithm are labelled "active", while experiments with the unmodified algorithm are labelled "semi-active".

In order to answer the first two questions posed in section 5.3, a number of selected problems were solved using the $R_{C_{max}}$-GA and both variants of the C_{max}-GA. For each problem, the algorithms were run 400 times and the values of C_{max} and $R_{C_{max}}$ recorded. The problems used were la01, la02, la06, la07, la26, la27, la31, la36, ft10, and ft20 (see section 3.4). These problems were used since they span a wide range of problem sizes and difficulties. The average robustness measures found in the experiments can be seen in table 5.1.

From table 5.1 it is evident that for all the problems on average the algorithm minimising $R_{C_{max}}$ produces schedules with a lower robustness measure than the semi-active and active algorithms minimising C_{max}.

Since the distribution of the observations is unknown, it is difficult to do statistical testing of this. The average standard deviation of the observations in the experiments minimising $R_{C_{max}}$ is 3.3, while for the active and semi-active experiments it is slightly more than 6. The distributions of $R_{C_{max}}$ were investigated by

problem	size	optimum	semiactive	active	robust
la01	10×5	666	666.0	666.0	666.0
la02	10×5	655	655.1	655.1	655.6
la06	15×5	926	926.0	926.0	926.0
la07	15×5	890	890.0	890.0	890.0
la26	20×10	1218	1221.2	1221.2	1219.5
la27	20×10	1235	1276.9	1276.6	1272.4
la31	30×10	1784	1784.0	1784.0	1784.0
la36	15×15	1268	1297.0	1297.0	1299.7
ft10	10×10	930	947.8	947.7	950.6
ft20	20×5	1165	1198.6	1197.7	1189.8

Table 5.2: *The mean makespan of the solutions found in the experiment.*

plotting empirical distribution functions, and it was found that in approximately 75% of the experiments the distribution functions were quite close to a Gaussian distribution. In the rest of the experiments the distributions were not Gaussian. Assuming that the robustness measures are Gaussian distributed the observations can be compared using a t-test. This has been done and for all problems the observed robustness measure averages for the robust runs turned out to be significantly different from the active and semiactive results at a significance level of 95%. Because of the non-Gaussian distribution for some of the experiments, this conclusion should be taken with a grain of salt. However, given the relatively large difference between the numbers and the consistency from problem to problem, there is little doubt that the result holds. From this can be concluded that the answer to the first question in the beginning of section 5.3 is yes; there is difference between minimising C_{max} and minimising $R_{C_{max}}$, and the lowest $R_{C_{max}}$ values are obtained by minimising $R_{C_{max}}$.

Comparing the results of the semi-active and active algorithms, there is not much difference, but in all cases the robustness measures are slightly lower for the active schedules than for the semi-active schedules. The difference was tested using a t-test, assuming Gaussian distributions. The difference turned out to be significant for all the problems except ft10 and ft20.

Turning to the makespan performance of the schedules, it seems clear from table 5.2, that in most cases schedules found by minimising the robustness measure have a makespan comparable to schedules found by minimising makespan. In three cases (la02, la36 and ft10) the average makespan of schedules found in the robust experiments are higher than the average makespan in the active or semi-active experiments. This indicates that for some problems there may be a tradeoff between the robustness measure of a schedule and the raw makespan.

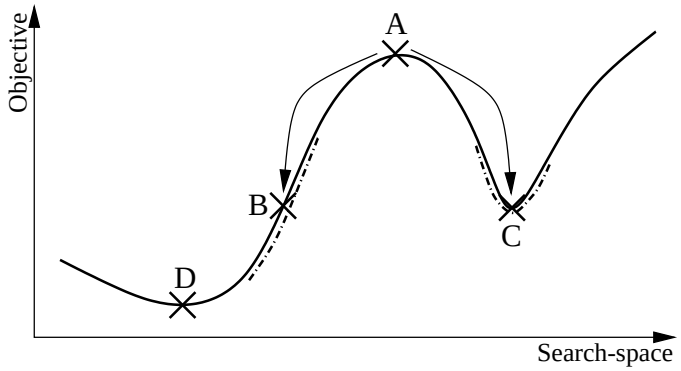

Figure 5.3: *The neighbourhood evaluation seen by the $R_{C_{max}}$ hillclimber working as a lookahead for C_{max} minimisation.*

This question can be further explored by considering the makespan and robustness measure of every schedule produced in the experiments. For each of the ten problems, the Pareto non-dominated solutions in the set of solutions found in the experiments were identified. For the problems `la01`, `la06`, `la07`, `la26`, `la27`, `la31` and `ft10` it was found that there was no tradeoff; among the schedules found in the experiments, the $R_{C_{max}}$-minimal schedule was also C_{max}-minimal. For the problems `la02`, `la36` and `ft20` a tradeoff was identified; these problems all had two or more Pareto-optimal schedules. Note that this experiment provides just an indication of whether a tradeoff exists or not; since the schedules have been found by using a heuristic, we have no guarantee that the solutions we are considering are the real Pareto-front.

In conclusion, question two of section 5.3 cannot be answered with an unconditional yes. For some of the problems we have identified a tradeoff between the optimal C_{max} and $R_{C_{max}}$ values, and the average performances of the algorithms support this. We can, however, answer question three with a conditional yes; for the majority of the problems no tradeoff was identified, neither in the optimal values found nor the averages reported in tables 5.1 and 5.2. For most problems it is possible to find schedules with a low robustness measure as well as a low makespan.

For three of the problems (`la26`, `la27` and `ft20`), the average makespan found for the $R_{C_{max}}$ minimised schedules is lower than for the C_{max} minimised schedules. This is probably because the minimisation of the $R_{C_{max}}$-estimate in the hillclimber used after decoding can in some cases work as a kind of "look-ahead" for the minimisation of C_{max}. Consider a hillclimber when it has the choice of two different steps, both leading to the same objective function value, in a situation exemplified in figure 5.3. The initial position of the hillclimber is A, and it has the choice of moving to the points B or C. A hillclimber evaluating the raw objective

function (C_{max}) will consider only the values at the points labelled B and C. Since the values are identical, the hillclimber will make an arbitrary choice, meaning that it may well get trapped in the local optimum at C. A hillclimber minimising a neighbourhood based robustness measure ($R_{C_{max}}$) will consider all the points in a neighbourhood around B and C (as indicated by the dashed lines[1]), and prefer B over C. In the next move, this hillclimber will take one more step from B to the global optimum at D, while a hillclimber preferring C over B will be trapped at C forever. However, when comparing the makespan performance of the C_{max}- and $R_{C_{max}}$-hillclimbers, it should be remembered that the $R_{C_{max}}$-hillclimber uses more processing time. Thus, if makespan is to be minimised, it is probably better to use the C_{max}-hillclimber, since more solutions can be tested in the same time.

5.3.4 Breakdown generation

The breakdowns used in the experiments of this chapter are machine breakdowns. They are generated as described in section 4.4 in such a way that all operations have an equal probability of being affected by a breakdown. A breakdown is generated by uniformly choosing an operation o_X in the schedule and letting the breakdown occur at the time the operation was supposed to start processing. This time is called the *breakdown time*. Since o_X has not yet started processing at the breakdown time, the scheduler is free to reschedule o_X, as well as all other operations supposed to start processing at the breakdown time or later. Once the breakdown has happened and rescheduling has been performed, the schedule is allowed to run until all operations have finished processing; each experiment simulates exactly one breakdown.

The processing time of operations in all the problems used are in the interval $\{1, \ldots, 100\}$, so it is natural to use a breakdown duration in that range. Unless otherwise indicated, the breakdown duration used in the experiments is 80.

5.3.5 Rescheduling

There has not been much previous work addressing the rescheduling problem, although several people have remarked that it can be solved in the same way as a standard job shop problem, since a job shop rescheduling problem is also a job shop problem [16, 42]. For most authors actually doing experiments with robust scheduling and rescheduling, right-shifting rescheduling is the method of choice [3, 68, 72].

[1]Since the $R_{C_{max}}$-hillclimber minimises the estimate $R_{C_{max},est}$ instead of $R_{C_{max}}$, what is actually seen by the $R_{C_{max}}$-hillclimber in the neighbourhood around B is never lower than the objective level at the point B, but this does not change the situation significantly.

When solving a rescheduling problem it is possible to make use of the preschedule. Due to the breakdown, it is not possible to implement the preschedule anymore, but as the preschedule is known to be an acceptable solution to a problem closely related to the rescheduling problem it makes sense to use it as a starting point for the rescheduling procedure. This may decrease computational costs considerably, and it may also increase the similarity between the preschedule and the new schedule.

In the experiments, a rescheduling problem in the form of a non-preemptive machine breakdown is made, and rescheduling done in five different ways:

1. Right-shifting. Simply wait for the breakdown to be repaired and use the scheduling order of the preschedule. This is expected to yield low quality results compared to other methods, but at a very low computational cost.

2. $\mathcal{N}_1$-based rescheduling. All $\mathcal{N}_1$-neighbours of the right-shifted preschedule are generated, and the one best solving the rescheduling problem is used. This is the most simple kind of search based rescheduling used, and is expected to yield better results than right-shifting, still at low computational cost.

3. Hillclimbing rescheduling. The right-shifted preschedule is used as a starting point for a hillclimber, which finds a locally optimal solution to the rescheduling problem. In experiments minimising C_{max}, the C_{max} hillclimber of section 3.2.2 was used. In experiments minimising $R_{C_{max}}$, the $R_{C_{max}}$ hillclimber of section 5.3.2 was used.

4. Reduced rescheduling. Generate a reduced rescheduling problem as described in [41] by removing all operations not affected by the breakdown from the problem. An operation is affected by a breakdown if it succeeds the operation hit by the breakdown in the graph representation of the preschedule, or equivalently, if an increase in the breakdown duration can lead to a delay of the operation if a right-shifting rescheduling method is used. The reduced problem is solved from scratch using the GA. Generation of a reduced rescheduling problem is illustrated in figure 5.4.

5. Complete rescheduling. Solve the rescheduling problem described above from scratch using the GA. This is expected to give a high schedule quality at a high computational cost.

In the experiments involving robustness measures rescheduling using methods 3-5 minimised the robustness measures, not the "raw" performance measures.

The search based methods may be augmented by turning the problem into a multi-objective optimisation problem, optimising schedule quality and similarity

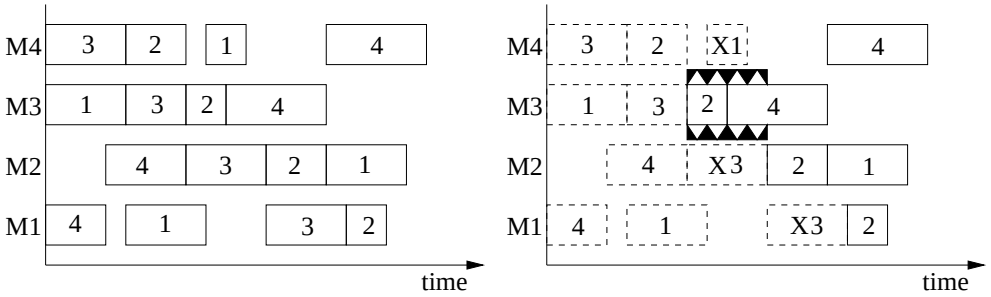

Figure 5.4: *Gantt-chart example of the generation of a reduced rescheduling problem. The operations have been labelled with their job numbers.* ***Left:*** *Preschedule.* ***Right:*** *Schedule at breakdown. The breakdown happens at machine M3 and has been marked by the black triangles (the schedule is infeasible, since it processes operations during the breakdown; it has not been rescheduled). The dashed operations will not be in the reduced rescheduling problem, while the solid operations will be. The operations marked by 'X' commence processing after the breakdown but will not be in the reduced rescheduling problem since they are not affected by the breakdown. After rescheduling these operations will be scheduled in the same way as they were in the preschedule.*

to the preschedule. This changes the problem completely, since there will not be a single best solution but rather a Pareto front of non-dominated solutions. Treating rescheduling as a multi-objective optimisation problem will not be treated further here.

The performance of methods 4 and 5 may be improved (both in terms of computational cost and in terms of similarity to the preschedule) by seeding the initial population of a GA with individuals produced from the population that solved the original problem. This was proposed in [15] and will be investigated in section 5.3.8.

The search-spaces of the five rescheduling methods have been visualised in figure 5.5. Since for this problem a large search-space can be expected to mean a higher solution quality (and a longer running time), complete rescheduling can be expected to outperform all the other rescheduling methods. In the figure, neither the search-space of hillclimbing nor $\mathcal{N}_1$ rescheduling are shown to contain the other. In reality, the part of the $\mathcal{N}_1$ search-space not contained in the hillclimbing search-space is very small, while the part of the hillclimbing search-space not contained in the $\mathcal{N}_1$ search-space is large. Furthermore, because of the construction of hillclimbing moves, for every schedule s in the $\mathcal{N}_1$ search-space not in the hillclimbing search-space, there will be a schedule in the hillclimbing search-space estimated to do better than s. For these reasons hillclimbing can be expected to outperform $\mathcal{N}_1$ rescheduling in most cases. Since the search-space

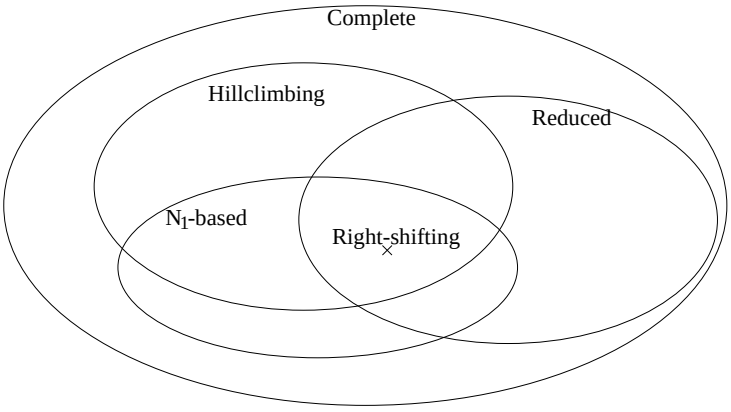

Figure 5.5: *The relationship between the search-spaces of the five rescheduling methods.*

of right-shifting is just a single point contained in all the other search-spaces, right-shifting can be expected to be the poorest (and fastest) of the rescheduling procedures. Reduced rescheduling can be expected to perform worse than complete rescheduling. Its relationship to the other rescheduling methods cannot be determined from the figure, since the reduced search-space is not contained in and does not contain the other search-spaces. The sizes of the sets are a bit misleading in the figure; for many rescheduling problems the search-spaces of complete and reduced rescheduling will be much larger than the other search-spaces.

5.3.6 The first rescheduling experiments

A set of experiments was conducted on the 10 problems of section 5.3.3. Each experiment consisted of a run of the scheduling GA to create a preschedule, the simulation of a breakdown and rescheduling using each of the five rescheduling methods. For every experiment the makespan of the preschedule and the makespan of the new schedules were recorded, as was the overlap between the new schedule and the preschedule. Scheduling was done using either the semi-active or active version of the C_{max}-GA (labelled "semi-active" and "active" experiments) or the $R_{C_{max}}$-GA (labelled "robust" experiments). In the following, the averages were taken over 400 runs of each algorithm.

Makespan results

The average makespans for each combination of problem, scheduling algorithm and rescheduling method can be seen in table 5.3. There is a subtable for every problem instance, and each subtable has three rows, reporting results for the

la01						
method	P	1	2	3	4	5
semiactive	**666.0**	719.7	712.2	709.2	704.2	702.6
active	**666.0**	721.6	714.7	711.1	706.0	704.4
robust	**666.0**	**707.3**	**701.7**	**700.9**	**699.8**	**699.1**
la02						
method	P	1	2	3	4	5
semiactive	**655.1**	716.8	711.3	**708.6**	**700.6**	**698.0**
active	655.2	717.5	713.1	710.9	702.2	699.5
robust	656.0	**713.9**	**710.3**	710.1	701.8	699.7
la06						
method	P	1	2	3	4	5
semiactive	**926.0**	978.5	971.3	966.3	962.5	962.3
active	**926.0**	966.8	959.3	954.5	951.5	951.3
robust	**926.0**	**946.1**	**945.3**	**945.3**	**945.1**	**945.1**
la07						
method	P	1	2	3	4	5
semiactive	**890.0**	946.3	941.2	936.3	931.3	930.0
active	**890.0**	940.0	933.4	928.5	923.0	921.8
robust	**890.0**	**920.5**	**918.0**	**917.2**	**916.3**	**916.1**
la26						
method	P	1	2	3	4	5
semiactive	1221.2	1282.7	1277.0	1272.1	1258.5	1255.8
active	1221.4	1280.4	1275.2	1270.4	1257.5	1254.9
robust	**1219.2**	**1271.1**	**1266.8**	**1264.0**	**1255.1**	**1252.2**
la27						
method	P	1	2	3	4	5
semiactive	1276.8	1338.8	1333.3	1329.5	1311.7	1308.8
active	1276.5	1337.3	1331.8	1326.9	1309.6	1307.5
robust	**1272.2**	**1323.0**	**1318.5**	**1314.9**	**1301.5**	**1298.3**
la31						
method	P	1	2	3	4	5
semiactive	**1784.0**	1833.3	1824.6	1815.7	1805.5	1804.8
active	**1784.0**	1823.7	1815.3	1806.6	1799.2	1798.8
robust	**1784.0**	**1795.6**	**1793.9**	**1793.3**	**1792.8**	**1792.7**

Table 5.3: Part 1: *Makespan results from the first rescheduling experiments. Typically, the standard deviations of the observations (an observation is the makespan after a single run of the algorithm) are in the range 25-35, while standard deviations on the averages are between 1 and 2.*

la36						
method	P	1	2	3	4	5
semiactive	1297.3	1347.8	1340.0	1336.4	1328.7	1325.4
active	**1296.6**	1346.0	1338.6	1334.9	**1326.4**	**1323.4**
robust	1299.6	**1340.1**	**1335.1**	**1333.5**	1328.6	1325.9
ft10						
method	P	1	2	3	4	5
semiactive	948.5	1013.1	1007.5	1004.3	992.9	991.2
active	**947.5**	1011.8	1006.1	1002.7	**991.8**	**989.4**
robust	950.6	**1007.8**	**1002.2**	**1001.1**	992.8	989.9
ft20						
method	P	1	2	3	4	5
semiactive	1196.9	1265.6	1261.8	1257.2	1236.3	1234.2
active	1198.0	1267.1	1262.8	1258.3	1239.1	1236.5
robust	**1188.7**	**1249.1**	**1243.9**	**1239.5**	**1228.0**	**1226.4**

Table 5.3: Part 2: *Makespan results from the first rescheduling experiments.Typically, the standard deviations of the observations (an observation is the makespan after a single run of the algorithm) are in the range 25-35, while standard deviations on the averages are between 1 and 2.*

semi-active, active, and robust runs respectively. Every subtable has six columns. The column labelled "P" reports the average preschedule makespan in the experiments. The columns labelled "1"-"5" report the average makespan of the schedules after a breakdown and rescheduling using the rescheduling method indicated by the column, see section 5.3.5. Recall that the methods are labelled according to their computational effort; rescheduling method "1" (right-shifting) has the lowest computational effort, method "5" (complete rescheduling) has the highest. In every column the best performance (lowest number) has been printed in bold.

As can be seen, in many cases the schedules labelled "robust" clearly outperform the other schedules. For seven out of the ten problems, the best performance for all of the rescheduling methods as well as the preschedule makespan is found by the $R_{C_{max}}$-GA. In some of these cases the performance difference is quite large. For the problems la06, la07 and la31 the average makespan is decreased by 19 or more for right-shifting rescheduling when using the $R_{C_{max}}$-minimised schedules instead of the ordinary schedules. For the same problems the rescheduling method expected to have the worst performance (right-shifting) for $R_{C_{max}}$-minimised schedules outperforms the rescheduling method expected to have the best performance (complete rescheduling) for C_{max}-minimised schedules.

Going through the table column by column, we see that for right-shifting rescheduling (1) and $\mathcal{N}_1$-based rescheduling (2), the $R_{C_{max}}$-minimised schedules are superior to the other schedules in all cases. This means that for all the ten problems the robustness and flexibility with respect to $\mathcal{N}_1$-rescheduling is improved by using these schedules. Considering hillclimbing rescheduling (3), reduced rescheduling (4), and complete rescheduling (5), the $R_{C_{max}}$-minimised schedules are outperformed by active or semi-active scheduling for the problems `la02`, `la36`, and `ft10`. In all the cases the performance difference is small and could be due to random variations. For the other problems, the $R_{C_{max}}$-minimised schedules outperform the ordinary schedules. It seems that the more sophisticated the rescheduling method, the smaller the difference between the performance of $R_{C_{max}}$-minimised schedules and the ordinary schedules.

Comparing the performance of semi-active and active schedules, it is evident that for some problems (`la06`, `la07` and `la31`) active schedules perform much better than semi-active schedules. For the other problems, the performance seems to be on the same level; in some cases active schedules are slightly better than semi-active ones, in other cases it is the other way around. Judging from the experiments of table 5.3, active schedules are preferable to semi-active schedules.

Comparing the rescheduling methods to each other, the expectations from section 5.3.5 are confirmed. Generally right-shifting is the poorest rescheduling method, followed by $\mathcal{N}_1$-based rescheduling. Hillclimbing rescheduling performs better than these two methods, while being outperformed by reduced and complete rescheduling. Complete rescheduling can be seen to have the best performance in all cases.

Since the distributions of makespans are not known, it is difficult to do statistical comparisons of the numbers in table 5.3. Empirical distribution functions for the makespans after rescheduling were plotted for the ten problems. A minority of these were found to resemble Gaussian distributions, but most of them do not look like any known distributions, so no classical statistical tests were performed.

Instead of doing statistical testing, the performances of the algorithms can be compared by comparing the distribution functions for each of the experiments. In figures 5.6 and 5.7, empirical distribution functions for makespan after rescheduling have been plotted for the `la06` and `la26` problems for each of the rescheduling methods. Each plot has three distribution functions, one for semi-active, one for active, and one for robust schedules. The number of experiments needed to reach an acceptable level of uncertainty was much higher than the experiments used to construct table 5.3; the plots are the result of 2800 runs of each algorithm.

Since the plots are distribution functions, they can be used to read the probability of ending up with a makespan lower than a given value for a rescheduling method and schedule type. For example for the `la06`, right-shifting rescheduling and active schedules, the probability of ending up with a makespan of 970 or

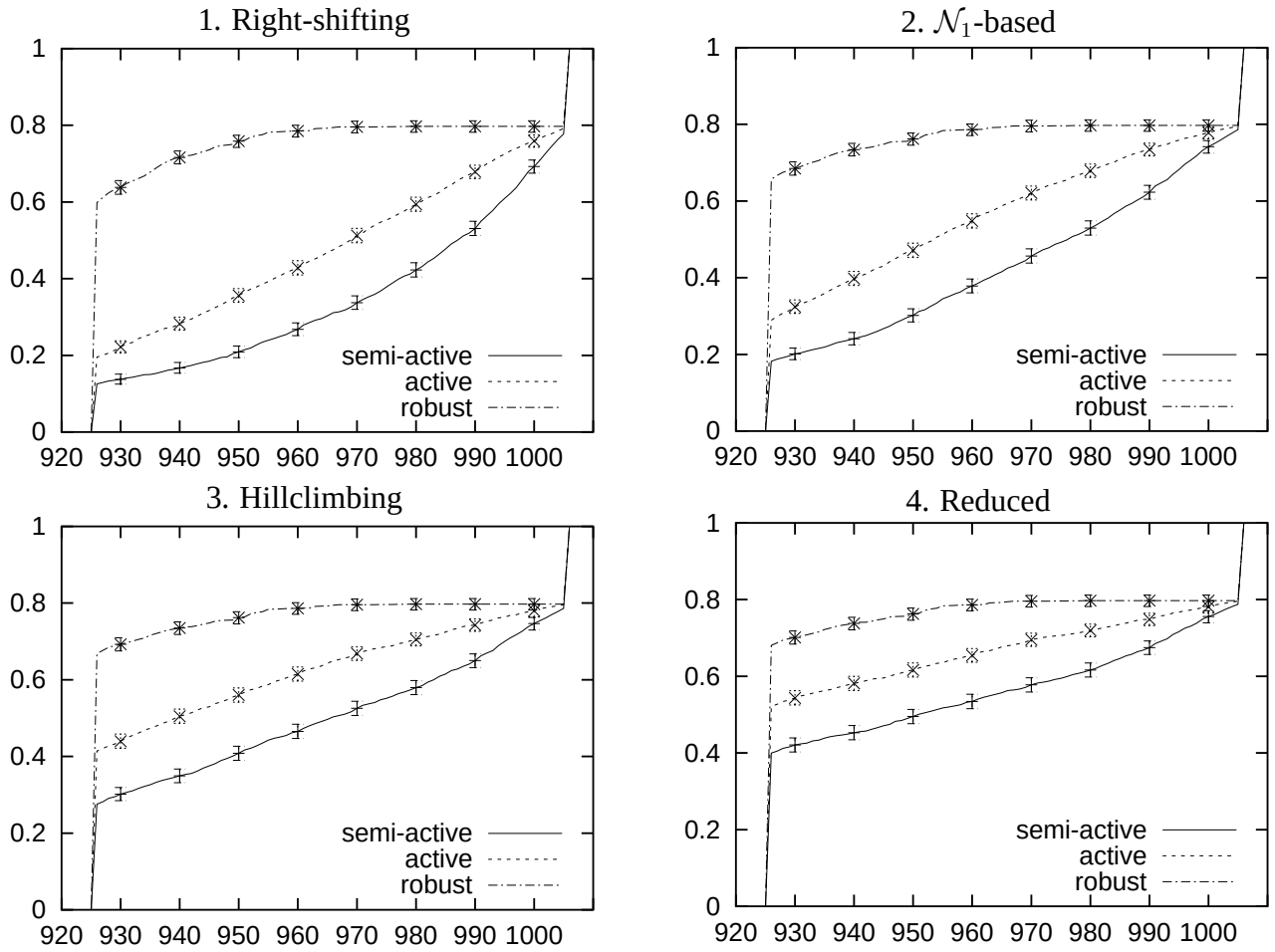

Figure 5.6: Part 1: *Observed distribution functions for makespan after rescheduling for the* `la06` *problem. The error bars on the plots are 95% confidence intervals on the percentiles.*

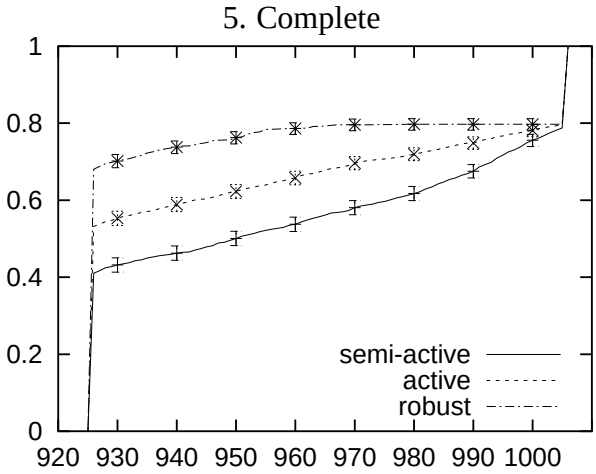

Figure 5.6: Part 2: *Observed distribution functions for makespan after rescheduling for the* `la06` *problem. The error bars on the plots are 95% confidence intervals on the percentiles.*

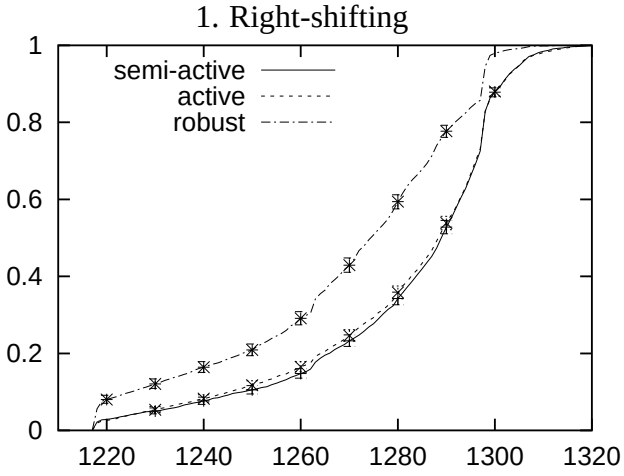

Figure 5.7: Part 1: *Observed distribution functions for makespan after rescheduling for the* `la26` *problem. The error bars on the plots are 95% confidence intervals on the percentiles.*

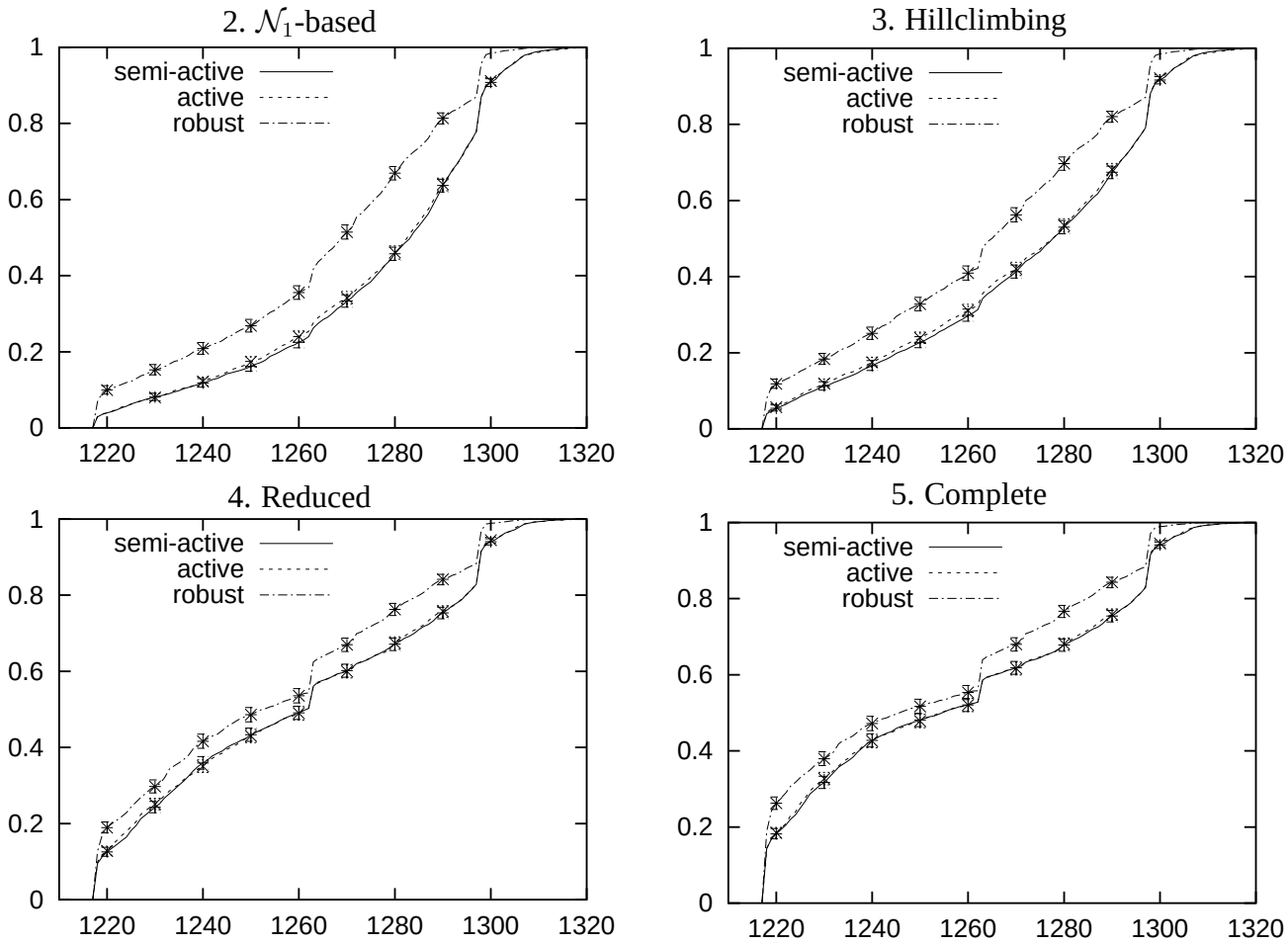

Figure 5.7: Part 2: *Observed distribution functions for makespan after rescheduling for the* `la26` *problem. The error bars on the plots are 95% confidence intervals on the percentiles.*

less can be seen to be 50%. On all the plots, the graphs for robust schedules are located above and to the left of the graphs for semi-active and active schedules, indicating that for these schedules, there is a higher probability of ending up with a low makespan schedule after rescheduling. The error bars on the plots indicate 95% confidence intervals on the percentiles (how these are calculated is explained in appendix E), so for both problems, they indicate that the difference between $R_{C_{max}}$-minimised schedules and C_{max}-minimised schedules is statistically significant for all rescheduling methods. From the plots it is also evident that in some cases the use of $R_{C_{max}}$-minimised schedules decreases the probability of ending up with a very high makespan schedule considerably. Consider the `la06` problem with hillclimbing rescheduling. For $R_{C_{max}}$-minimised schedules, approximately 75% of the schedules end up with a makespan of 945 or less. For active schedules, 75% of the schedules end up with a makespan of 991 or less.

All the plots for the `la06` problem have a sharp increase at the 80% percentile, because breakdowns happening on one of the machines (one fifth of the breakdowns) cannot be countered in any way. The machine is critical during the entire period of processing so breakdowns happening at this machine will always increase the makespan by the breakdown duration.

Plots similar to the ones of figures 5.6 and 5.7 were also made for the problems `la01`, `la02`, `la07`, `la27`, `la31`, `la36`, `ft10` and `ft20`. They were qualitatively equivalent to the plots for the `la06` and `la26` problems, except `la02`, `la36` and `ft10` for hillclimbing, reduced and complete rescheduling, where no significant differences between the three scheduling methods were found.

From these experiments can be concluded that for this kind of breakdown, if makespan is to be kept minimal after rescheduling (regardless of the rescheduling method) $R_{C_{max}}$-minimised schedules are preferable to C_{max}-minimised schedules. In a majority of the experiments of this section, $R_{C_{max}}$-minimised schedules outperformed the other schedules, and while a few cases were found in which C_{max}-minimised schedules were slightly better than $R_{C_{max}}$-minimised schedules, these differences did not turn out to be statistically significant.

Overlap results

As argued in section 4.6, it is sometimes desirable for new schedules to be as similar to the preschedule as possible. The similarity between the preschedule and the new schedule can be measured using the overlap measure defined in section 4.6. The average schedule overlaps between the part of the preschedule not implemented and the new schedule can be seen in table 5.4. The table has been arranged in the same way as table 5.3. The reader is reminded that the rescheduling algorithms do not consider the overlap in any explicit way; they simply minimise makespan (or the robustness measure, depending on the algorithm). Thus,

la01					
method	1	2	3	4	5
semiactive	0.593	0.642	**0.632**	0.617	0.465
active	0.562	0.611	0.600	0.594	0.464
robust	**0.623**	**0.649**	0.625	**0.630**	**0.562**
la02					
method	1	2	3	4	5
semiactive	0.519	**0.568**	**0.560**	0.590	0.479
active	0.490	0.538	0.523	0.572	0.477
robust	**0.525**	0.565	0.547	**0.609**	**0.536**
la06					
method	1	2	3	4	5
semiactive	0.602	0.661	**0.654**	0.547	0.328
active	0.609	0.667	0.653	0.553	0.331
robust	**0.640**	**0.691**	0.646	**0.600**	**0.385**
la07					
method	1	2	3	4	5
semiactive	0.551	0.606	0.597	0.551	0.320
active	0.567	0.619	0.614	0.569	0.358
robust	**0.612**	**0.652**	**0.631**	**0.597**	**0.427**
la26					
method	1	2	3	4	5
semiactive	0.588	0.634	0.635	0.606	0.419
active	0.607	0.657	0.650	0.624	0.423
robust	**0.636**	**0.677**	**0.670**	**0.657**	**0.511**
la27					
method	1	2	3	4	5
semiactive	0.590	0.637	0.635	0.614	0.439
active	0.586	0.637	0.652	0.615	0.435
robust	**0.624**	**0.666**	**0.666**	**0.641**	**0.491**
la31					
method	1	2	3	4	5
semiactive	0.658	0.718	**0.747**	0.517	0.278
active	0.640	0.693	0.721	0.510	0.278
robust	**0.715**	**0.762**	0.728	**0.565**	**0.346**

Table 5.4: Part 1: *Overlap results from the first rescheduling experiments.*

la36					
method	1	2	3	4	5
semiactive	0.632	0.691	0.700	0.680	0.571
active	0.624	0.677	0.688	0.687	0.593
robust	**0.666**	**0.713**	**0.711**	**0.696**	**0.621**
ft10					
method	1	2	3	4	5
semiactive	0.477	0.518	0.516	0.587	0.504
active	0.478	0.525	0.523	0.592	0.513
robust	**0.500**	**0.543**	**0.538**	**0.602**	**0.546**
ft20					
method	1	2	3	4	5
semiactive	0.445	0.493	0.481	0.476	0.268
active	0.439	0.489	0.474	0.474	0.271
robust	**0.499**	**0.548**	**0.535**	**0.511**	**0.326**

Table 5.4: Part 2: *Overlap results from the first rescheduling experiments.*

if one kind of schedule on average ends up having a higher preschedule overlap than another kind of schedule, the reason is to be found in the search-spaces of the rescheduling problems.

The results in the table indicate that for most problems the overlap is larger for the $R_{C_{max}}$-minimised schedules than for the ordinary schedules. For all ten problems in the table this is always the case for right-shifting, reduced and complete rescheduling, while it does not hold in one case for $\mathcal{N}_1$-based rescheduling and in four cases for hillclimbing rescheduling. In some cases the differences between the overlaps are very small and may be insignificant, while in other cases they are substantial. On average the difference is largest for complete rescheduling, where the average overlap for the $R_{C_{max}}$-minimised schedules is 0.478, while for semi-active and active schedules it is 0.406 and 0.412 respectively.

Since the distribution of overlaps after rescheduling is unknown, no statistical testing has been performed to compare the overlap measures. Instead, empirical distribution functions have been drawn in the same ways as they were for the makespan results. The plots for the la06 and la26 problems are displayed in figures 5.8 and 5.9. For the la06 problem the confidence intervals (see appendix E for an explanation of these) on the graph clearly demonstrate that the difference between C_{max}- and $R_{C_{max}}$-minimised schedules is statistically significant for right-shifting, $\mathcal{N}_1$-based, reduced and complete rescheduling. For these plots, the graphs for $R_{C_{max}}$-minimised schedules are located to the right and below of the other graphs, indicating a higher probability for a high overlap. No significant

difference can be observed for hillclimbing rescheduling for `la06`. The graphs of `la26` show that for this problem there is a statistically significant difference for all rescheduling methods.

Plots similar to the ones of figures 5.8 and 5.9 were drawn for the other eight problems. They showed that for all of the problems, the differences in overlap were significant for right-shifting and complete rescheduling. In all these cases, the difference was in favour of the $R_{C_{max}}$-minimised schedules. For $\mathcal{N}_1$-based rescheduling, the plots showed a significant difference for seven of the problems, while for the remaining three there was no difference. For hillclimbing rescheduling for three problems, there was a significant difference, and for reduced rescheduling for seven problems. In all cases where a significant difference was found, the difference was in favour of the $R_{C_{max}}$-minimised schedules, showing them to have a higher overlap to the preschedule than the C_{max}-minimised schedules.

All in all the overlap results on these ten problems indicate, that if overlap is to be high after rescheduling, the $R_{C_{max}}$-minimised schedules are preferable to C_{max}-minimised schedules. In all the cases where a significant difference was found in the plots (37 cases all in all), it was in favour of the $R_{C_{max}}$-minimised schedules, while some cases were also identified (13, all found for $\mathcal{N}_1$-based, hillclimbing and reduced rescheduling) in which there was no significant difference between the overlaps for $R_{C_{max}}$ and C_{max}-minimised schedules. This indicates that in terms of overlap, for some problems there is something to gain by using $R_{C_{max}}$-minimised schedules, while nothing ever seems to be lost.

5.3.7 Correlation study

The correlation between the robustness measure ($R_{C_{max}}$) and makespan after rescheduling was investigated for the problems `la06`, `la26` and `ft10`. This was done in order to further investigate the connection between low robustness measures and low after-rescheduling-makespan found in previous sections. The correlations study was done by creating 150 schedules for each problem, 50 minimising $R_{C_{max}}$, 50 minimising C_{max} using the active GA, and 50 minimising C_{max} using the semi-active GA. For every schedule, all the breakdowns possible allowed by the definition of section 5.3.4 were generated (i.e. one breakdown per operation), and rescheduling was carried out using the five rescheduling methods. After rescheduling, the makespan and preschedule overlap of the new schedule were recorded, and averages calculated. For the deterministic rescheduling methods (right-shifting, $\mathcal{N}_1$-based and hillclimbing) the averages found in this way are accurate, while for the stochastic rescheduling methods (reduced and complete) some uncertainty is present.

The results for right-shifting rescheduling and the `la26` and `ft10` problems have been plotted in figure 5.10. In the top left diagram, the average after re-

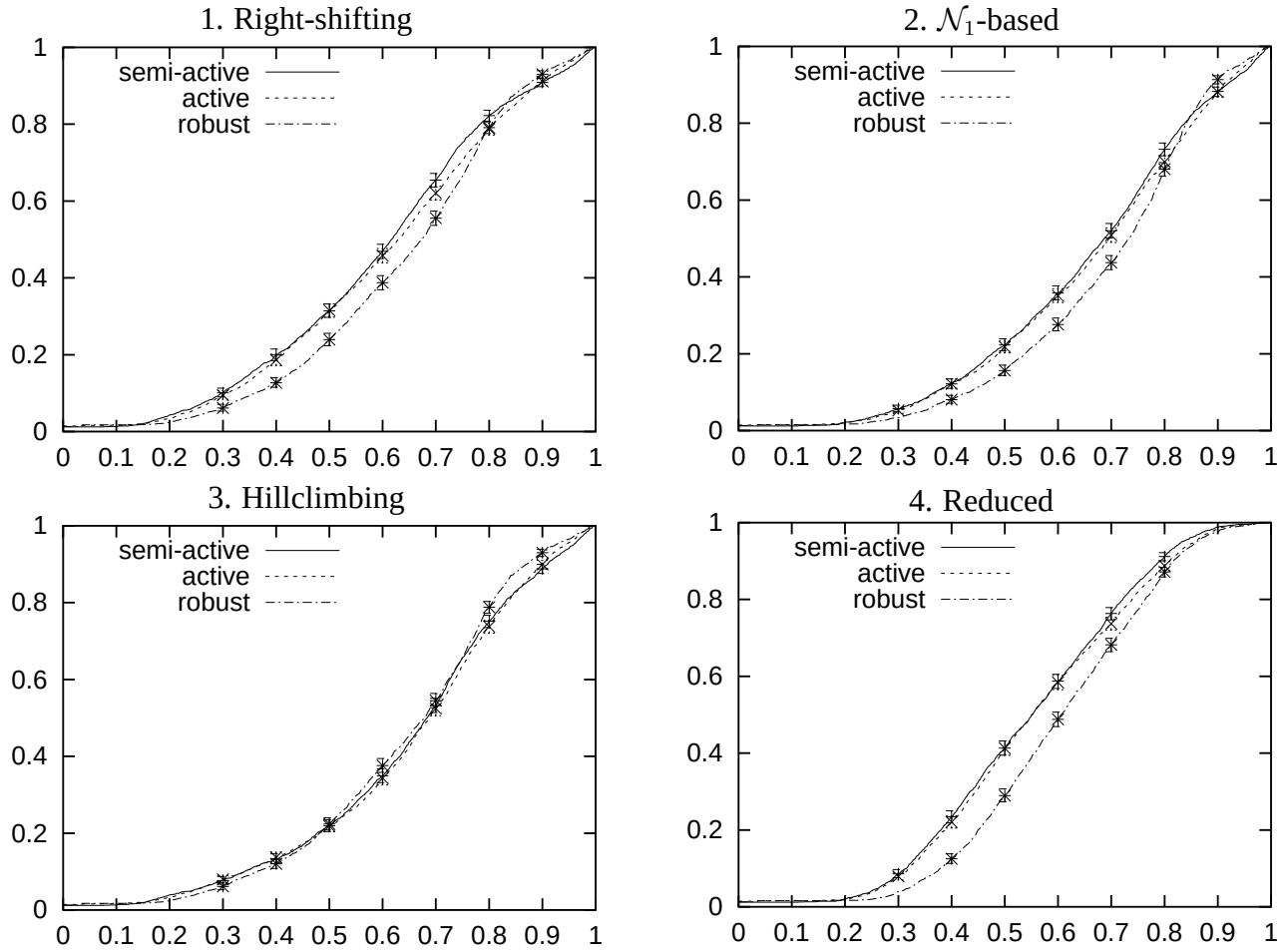

Figure 5.8: *Observed distribution functions for preschedule overlap after rescheduling for the* `la06` *problem. The error bars on the plots are 95% confidence intervals on the percentiles.*

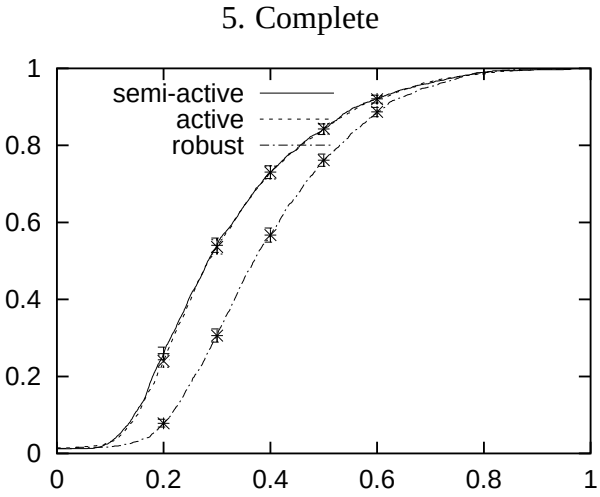

Figure 5.8: Part 2: *Observed distribution functions for preschedule overlap after rescheduling for the* `la06` *problem. The error bars on the plots are 95% confidence intervals on the percentiles.*

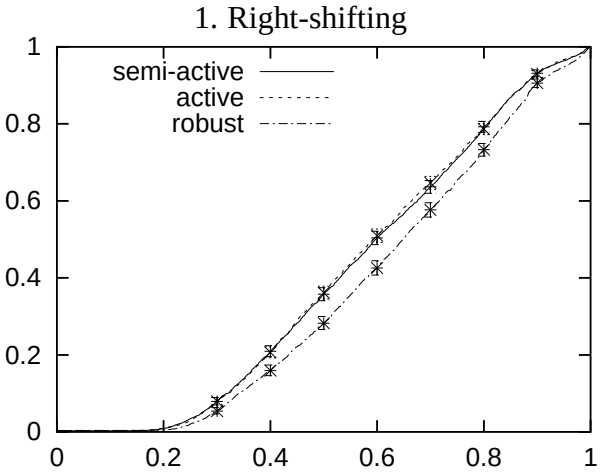

Figure 5.9: Part 1: *Observed distribution functions for preschedule overlap after rescheduling for the* `la26` *problem. The error bars on the plots are 95% confidence intervals on the percentiles.*

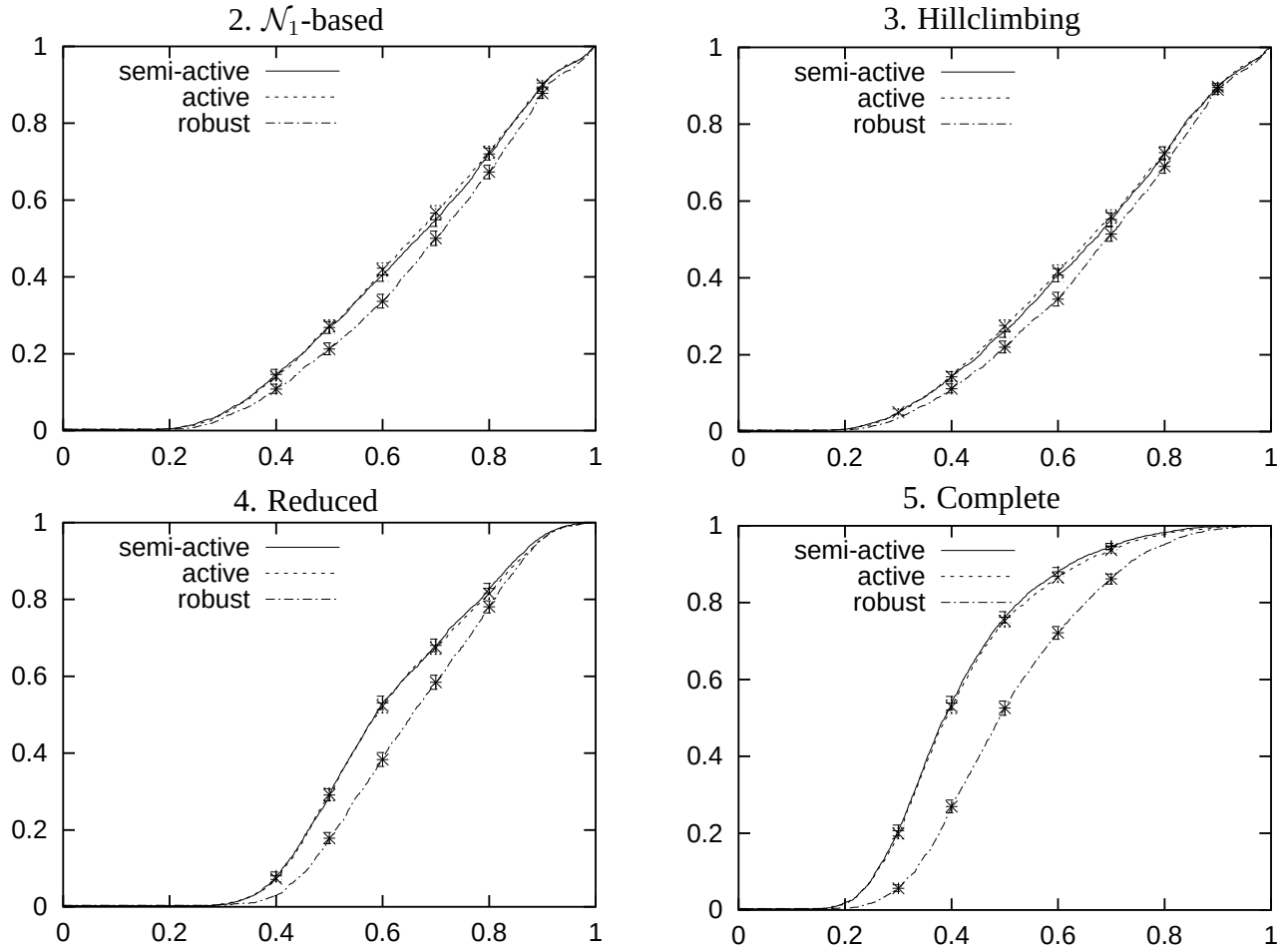

Figure 5.9: Part 2: *Observed distribution functions for preschedule overlap after rescheduling for the* `la26` *problem. The error bars on the plots are 95% confidence intervals on the percentiles.*

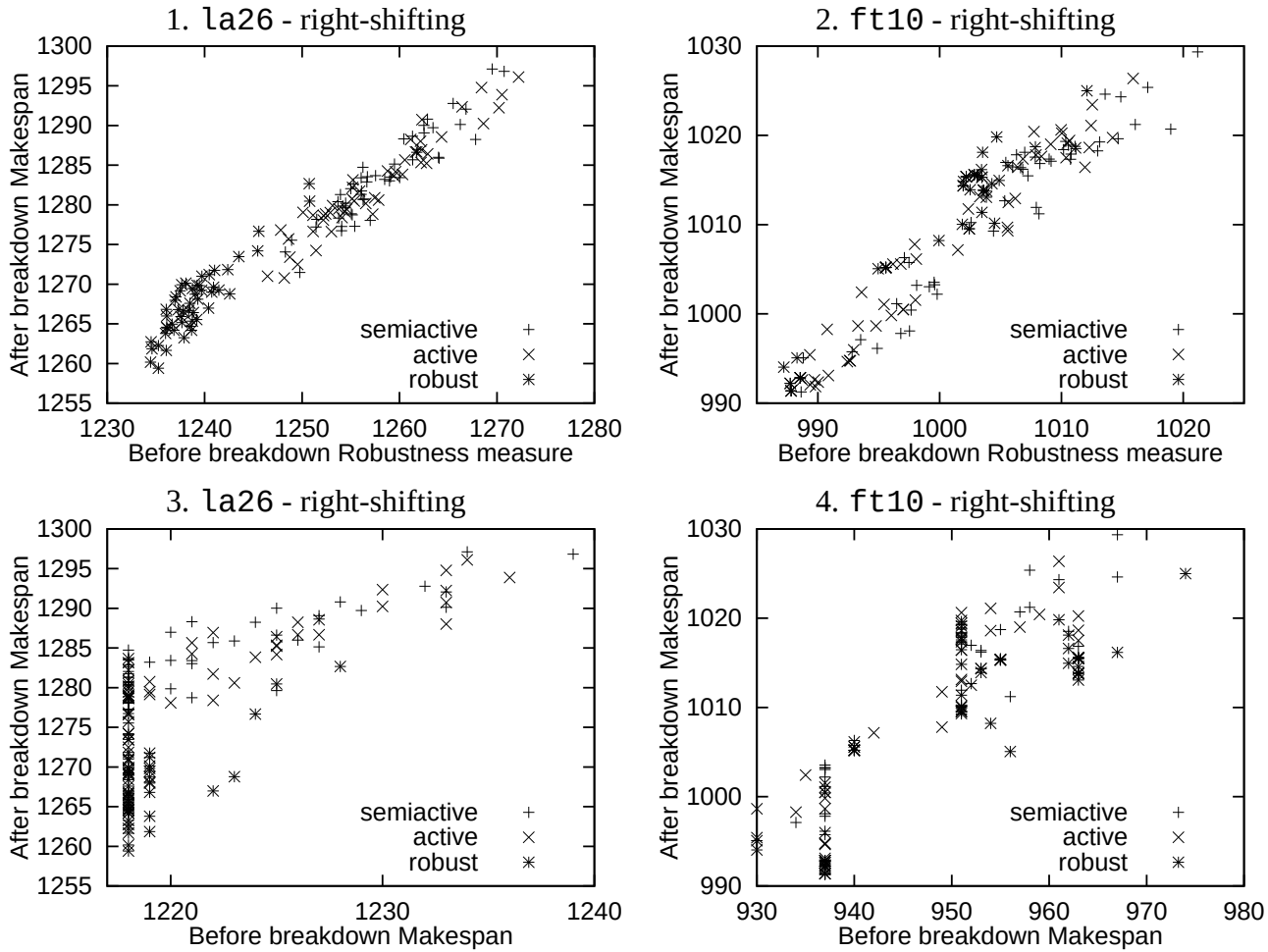

Figure 5.10: ***1*** *and* ***2****: Observed average makespan after breakdown and rescheduling as a function of preschedule* $R_{C_{max}}$ *for* `la26` *(1) and* `ft10` *(2) and right-shifting rescheduling.* ***3*** *and* ***4****: Same as a function of preschedule makespan.*

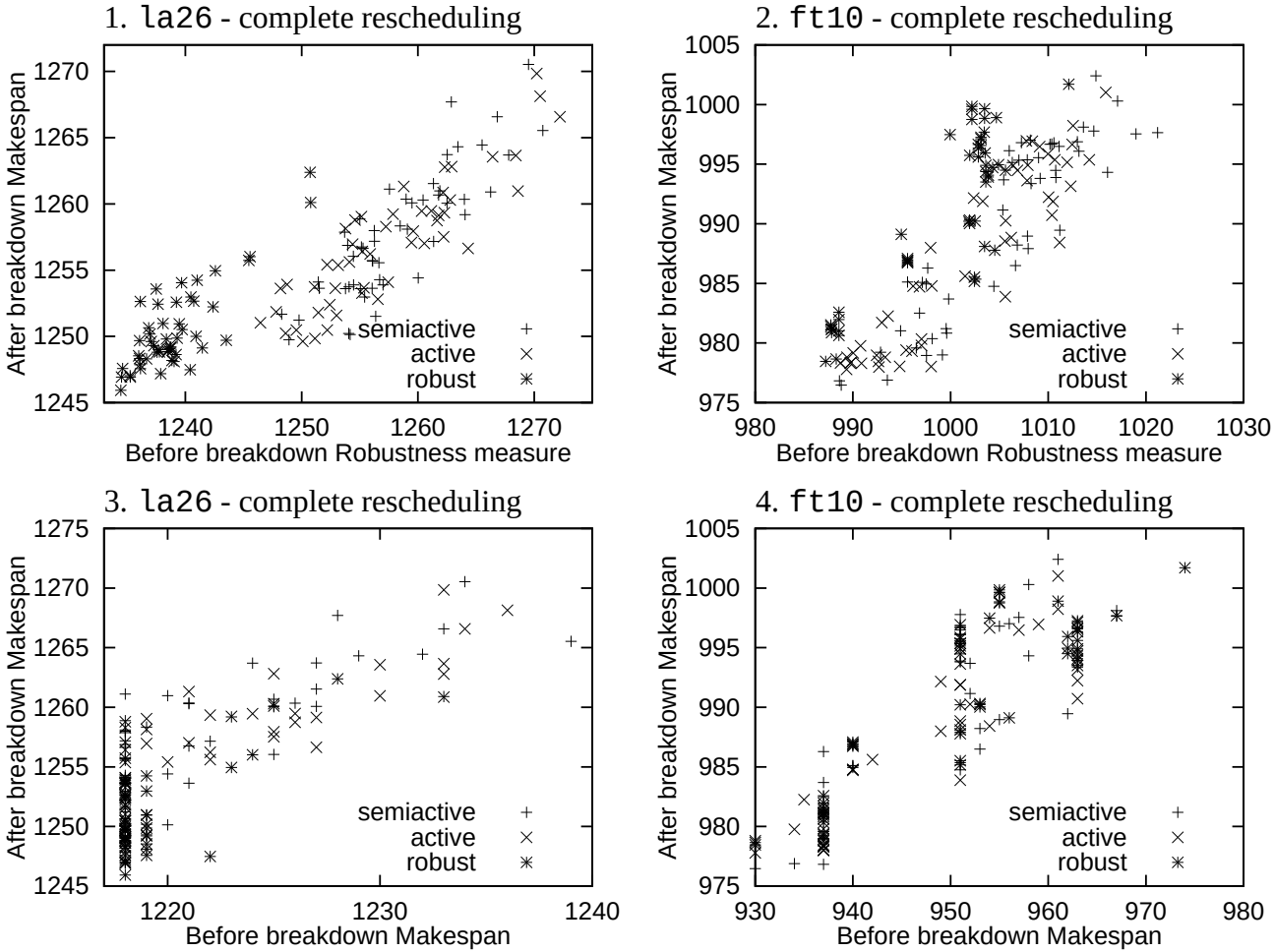

Figure 5.11: ***1*** *and* ***2****: Observed average makespan after breakdown and rescheduling as a function of preschedule* $R_{C_{max}}$ *for* `la26` *(1) and* `ft10` *(2) and complete rescheduling.* ***3*** *and* ***4****: Same as a function of preschedule makespan.*

scheduling performance has been plotted as a function of preschedule robustness measure for the `la26` problem. The points indicate an almost linear relationship; a low robustness measure on average means a low after rescheduling makespan, while a high robustness measure means a high after rescheduling makespan. The corresponding plot for the `ft10` problem (top right) is qualitatively equivalent.

The two plots in the top row of the figure can be compared to the plots in the bottom row, in which the after rescheduling performance has been plotted as a function of preschedule makespan, for the same sets of experiments on `la26` and `ft10`. These plots also show some correlation, but it is evident that while low robustness measure is almost a guarantee of a low after rescheduling makespan, the same can not be said about low makespan schedules; in many cases a low preschedule makespan can be seen to lead to a high after rescheduling makespan.

The plots for $\mathcal{N}_1$-based and hillclimbing rescheduling were qualitatively the same as the plots for right-shifting, although for hillclimbing rescheduling the linear relationships between the robustness measure and after rescheduling makespan was not as clear cut as for right-shifting.

The results for complete rescheduling and the `la26` and `ft10` problems can be seen in figure 5.11. For the `la26`, problem there is a clear correlation between low robustness measures and low after rescheduling makespans. The relationship seems to be more or less linear for schedules found in the semiactive and active C_{max}-GAs, while the points found by the $R_{C_{max}}$-GA form a cluster of their own in the plot, most of them being far to the left, without being any lower (better after rescheduling performance) than the best schedules found by the C_{max}-GAs. This indicates that for this kind of rescheduling using only the robustness measure to predict after rescheduling performance is too simplistic. This statement is even more true for the plot for `ft10` at the top right of figure 5.11. Again, a correlation between the robustness measure and the after rescheduling performance is evident, but the points corresponding to schedules found by minimising $R_{C_{max}}$ are generally located above the points found by minimising C_{max}. For this particular combination of problem and rescheduling method, the schedules found by minimising $R_{C_{max}}$ perform worse than the C_{max}-minimised schedules, see table 5.3. The plots at the bottom row of figure 5.11 show after rescheduling makespan as a function of before breakdown makespan for the same experiments. The plots suggest that for this kind of rescheduling, the after rescheduling performance correlates almost as strongly to the preschedule performance as to the robustness measure. However, from the results of table 5.3, the after rescheduling performance should still be expected to correlate more to the robustness measure than to the preschedule makespan.

The plots for reduced rescheduling were found to be equivalent to the plots for complete rescheduling.

5.3.8 Reusing the population in rescheduling

Since the problem we are solving when facing a rescheduling problem is often quite similar to the original scheduling problem, a straight-forward idea is to reuse the population from the last generation of scheduling (in the following termed *the reused population*) as the initial population for rescheduling when using complete or reduced rescheduling. The idea behind this would be to decrease the number of generations necessary to achieve a given solution quality, thus decreasing the time required for rescheduling, and to increase the similarity between the preschedule and the rescheduled schedule. Similar approaches were used recently in [16, 21, 73] for stochastic dynamic job shop problems solved on a rolling time horizon basis.

Since the population in the late stages of a GA can sometimes degenerate (very low genetic diversity), it may be a good idea to mutate some of the individuals in the reused population before starting the rescheduling GA. In order to find a suitable level of mutation, a set of experiments were conducted in which each individual was mutated using position based mutation with a probability of 0.0, 0.2, 0.4, 0.6, 0.8 and 1.0. The experiment was done on the `la26` problem, using 100 generations for scheduling and 100 generations for rescheduling. The robust searching scheme was used. The mutation rate was found only to have a small impact on the makespan after rescheduling, and almost no impact on similarity. Since makespan performance was slightly superior for values 0.4 and 0.6, the value 0.4 was used for the rest of the experiments.

The effect of reusing the population in the $R_{C_{max}}$-GA can be studied for the `la26` and `ft10` problems in figure 5.12. In the two plots at the top of the figure, the average makespan after rescheduling has been plotted as a function of the number of generations in the complete rescheduling algorithm with reuse of the population and without reuse. The two plots at the bottom of the figure are equivalent to the plots at the top, except the bottom plots show the average preschedule overlap after rescheduling. The averages in the plots have been calculated based on 2500 runs of each algorithm. The error-bars on the plots are 95% confidence intervals on the averages[2]. Considering makespan performance it is clear from the plot that if only little time (fewer than approximately 3500 fitness evaluations for these problems) is available to do the rescheduling, makespan performance can be improved by reusing the population. If many fitness evaluations are available for rescheduling, no performance gain is present. In fact, the plots of figure 5.12 suggest that for a large number of generations, the makespan performance can be slightly worse when reusing the population. However, comparing the empirical

[2]The intervals have been created assuming the individual observations to be normally distributed. In fact the distribution functions are unknown, so the confidence intervals should be taken as no more than guidelines on the uncertainty of the averages.

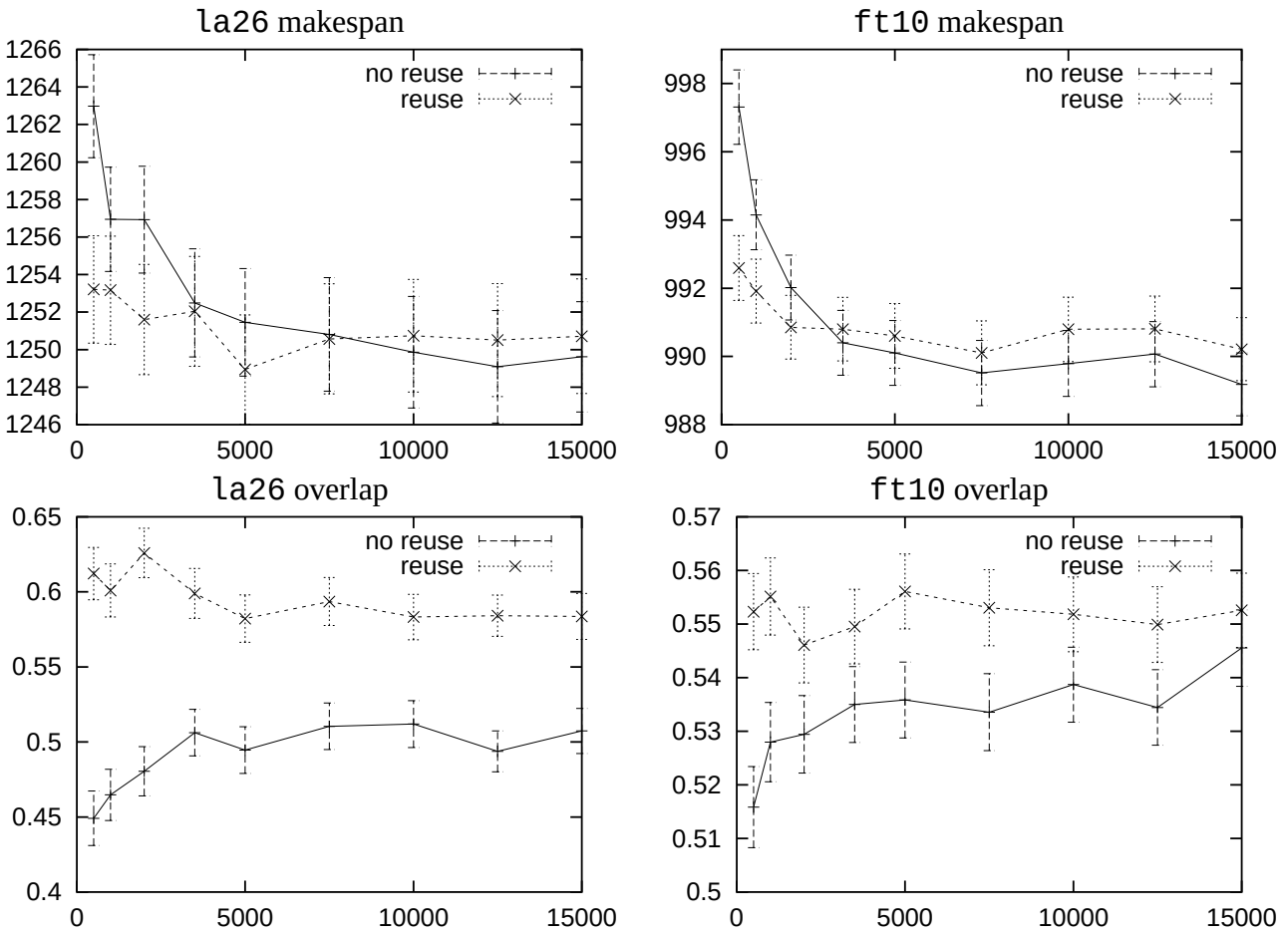

Figure 5.12: *Observed averages for makespan after rescheduling (top row), and for preschedule overlap with and without reuse of the population (bottom row). The plots were made for complete rescheduling of the* `la26` *and* `ft10` *problems.*

distribution functions for makespans after 10000 generations using the same kind of plots used in figure 5.6 revealed no statistically significant difference for any of the problems.

Considering overlap performance it is clear from the plots that reusing the population significantly increases overlap in all the experiments. Using empirical distribution function plots, the preschedule overlaps after 10000 generations of the reusing and non-reusing algorithms were compared for both problems. In both cases the difference turned out to be statistically significant. It is interesting that when not reusing the population, the overlap performance starts off very low and then increases, while when reusing the population the overlap starts off quite high and stays at more or less the same level, regardless of the number of fitness evaluations. This difference in behaviour probably corresponds to the solution of the non-reusing GA in the early stages of the run being more or less random, not having much similarity with the preschedule, later the overlap increases to a certain point, indicating that the GA settles on a peak having some similarity with the preschedule. On the other hand, the reusing GA starts with a solution very close to the preschedule. The constant level of preschedule overlap indicates that the algorithm stays close to this solution, even for a large number of generations.

The overlap to preschedule increase caused by reusing the population was also observed in experiments on the problems `la01`, `la02`, `la06`, `la07`, `la26`, `la27`, `la31`, `la36` and `ft20`.

Since the experiments clearly showed that reusing the population increases the similarity to the preschedule, while no significant drop in makespan performance was found, it was decided to reuse the population in rescheduling for the rest of the experiments.

5.3.9 The effect of the breakdown duration

The effect of the breakdown duration has been investigated for the `la06`, `la26`, `ft10` and `ft20` problems by varying the duration over nine values in the range 0 to 180 time units, and for each combination of scheduling and rescheduling method noting the makespan average after rescheduling. The results for the `la06` problem have been visualised in figure 5.13. The plots are qualitatively equivalent for all the rescheduling methods. For breakdown durations longer than 40 the $R_{C_{max}}$-minimised schedules clearly outperform the C_{max}-minimised schedules. The difference in makespan average between the two kinds of schedules increases with the duration until a duration of around 140, from where the difference appears to be constant. For breakdown durations of 0 or 20, there does not seem to be any difference between the two kinds of schedules.

The results for the `ft20` and `la26` were of a similar nature, except that for these problems there was also a clear performance difference in favour of the

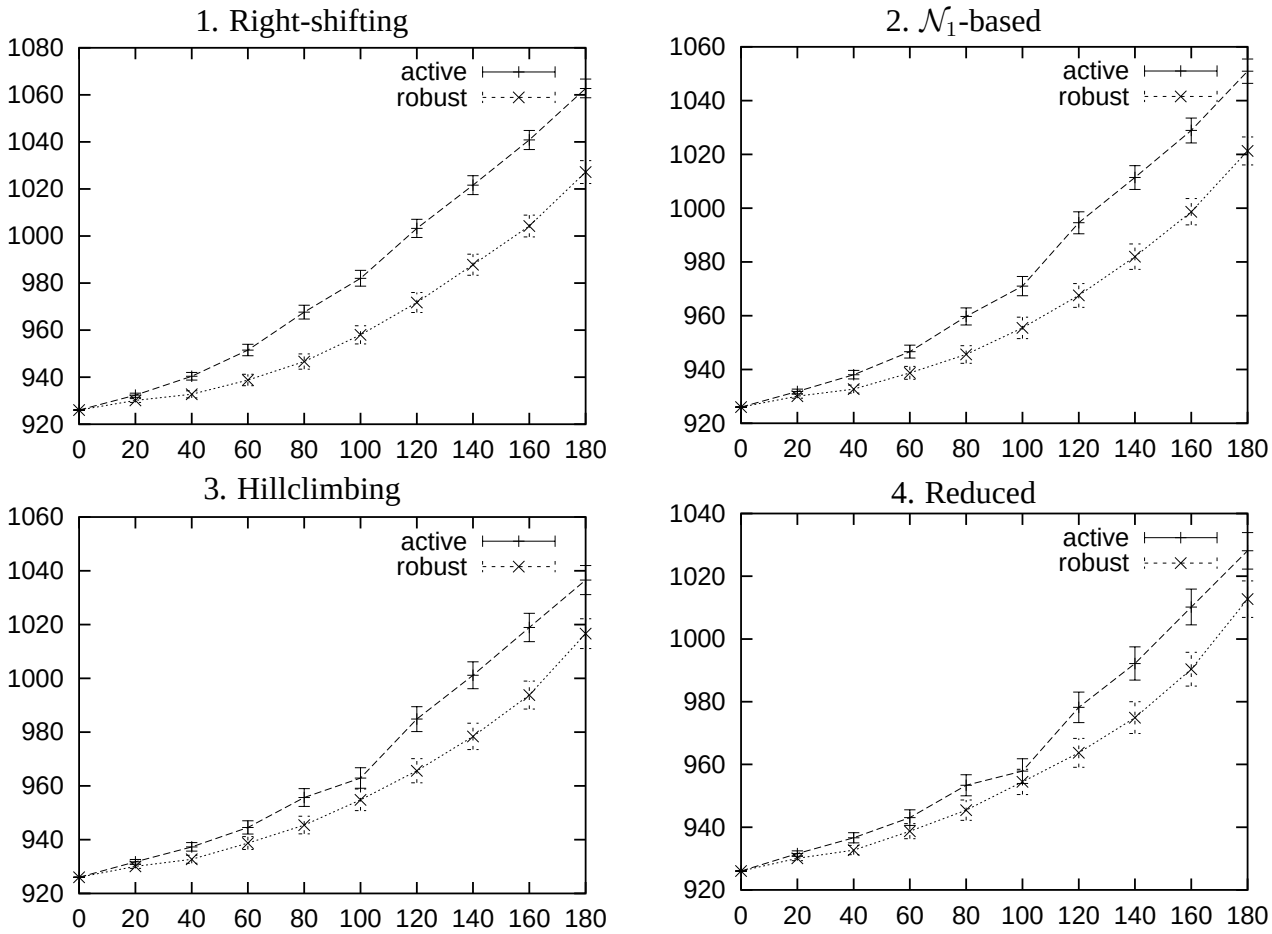

Figure 5.13: Part 1: *The effect of the breakdown duration on average makespan after rescheduling for the* `la06` *problem.*

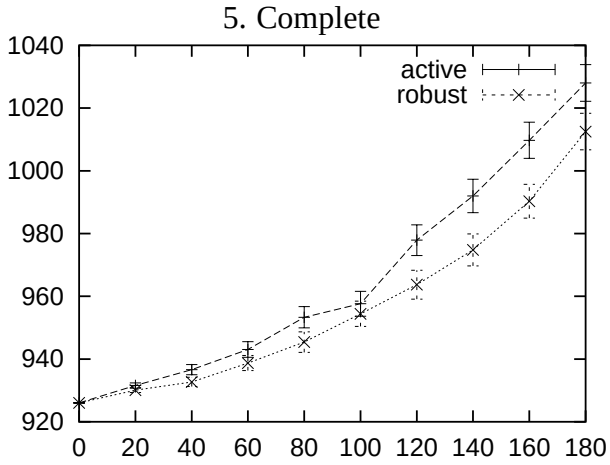

Figure 5.13: Part 2:*The effect of the breakdown duration on average makespan after rescheduling for the* `la06` *problem.*

$R_{C_{max}}$-minimised schedules for breakdown durations of 20. This was the case for all the rescheduling methods. For `ft10`, there was a small improvement from using the $R_{C_{max}}$-minimised schedules for right-shifting and $\mathcal{N}_1$-based rescheduling for all breakdown durations. For hillclimbing rescheduling, the $R_{C_{max}}$- and C_{max}-minimised schedules performed equally well. For reduced and complete rescheduling, the two kinds of schedules seem to perform on the same level for breakdown durations in the range 20-80, while for longer durations the C_{max}-minimised schedules were slightly superior.

For three of the four problems studied, there was a performance improvement from using the $R_{C_{max}}$-minimised schedules regardless of the breakdown duration and rescheduling method (except for the `la06` problem breakdown duration 20, where the two methods performed equally well). For the fourth problem there was a small loss in rescheduling performance for some combinations of breakdown duration and rescheduling method. The performance of the $R_{C_{max}}$-minimised schedules when compared to C_{max}-minimised schedules is highly dependent on the problem instance, the breakdown duration and the rescheduling methods. However, since the performance gains found are substantial, and since for the problems studied they are much more frequent than the performance losses, this study indicates that in many cases there is a lot to gain from using the $R_{C_{max}}$-minimised schedules, while in a few cases there is a little to lose.

5.3.10 Comparing to slack based robustness

Since the neighbourhood based robustness approach deals with a problem closely related to the problem studied by Leon et al. in [72] (see section 4.7.2), it is natural

Problem	size	C_{max}-GA	$R_{C_{max}}$-GA	$R_{C_{max},est}$-GA	Z_r-GA
ft10	10×10	3.2	19.1	6.6	3.6
ft20	20×5	5.2	25.1	10.7	5.9
la01	10×5	1.4	5.7	1.6	1.4
la02	10×5	1.6	6.3	2.3	1.6
la06	15×5	2.3	10.8	3.2	2.4
la07	15×5	2.5	12.0	4.1	2.7
la26	20×10	11.9	91.3	27.2	13.8
la27	20×10	11.7	92.6	27.4	14.5
la31	30×10	22.8	212.8	49.5	34.7
la36	15×15	10.2	79.7	22.7	11.3

Table 5.5: *CPU-time used in seconds on running the C_{max}-GA or the $R_{C_{max}}$-GA on some of the problems.*

to compare the achievements of the two methods. Recall that [72] describes a robustness measure $Z_r(s)$ based on the slack of the schedules to improve average makespan after a number of breakdowns. The two methods were not developed to cope with exactly the same problem, since Leon et al. studies schedules facing a series of preemptive breakdowns, while the experiments described in this chapter are on schedules facing one non-preemptive breakdown. Furthermore, Leon et al. use only right-shifting rescheduling, for which the $Z_r(s)$ robustness measure is specifically designed. However, as it is not inconceivable that increasing the slack in schedules can also be beneficial for more sophisticated rescheduling techniques, $Z_r(s)$-minimised schedules are subject to all five kinds of rescheduling in the experiments presented here.

In the next section, the methods will be compared using the same kind of breakdowns used earlier in this chapter, with a breakdown duration of 80.

The genetic algorithm described in section 3.2.2 was modified to minimise the $Z_{r=1}(s)$ robustness measure by simply changing the objective function to $Z_{r=1}(s)$. No other part of the algorithm was modified. In the following this algorithm will be called the Z_r-GA.

5.3.11 Using the robustness estimate in fitness evaluation

The exact evaluation of $R_{C_{max}}$ in the fitness evaluation of the $R_{C_{max}}$-GA is very time consuming. This is because all the $\mathcal{N}_1$-neighbours of every schedule need to be created. For this reason, runs of the $R_{C_{max}}$-GA are much slower than runs of the C_{max}-GA. The running times of the two algorithms can be seen in the first two columns of table 5.5 for the ten problems. Instead of evaluating the exact value of $R_{C_{max}}$, it is possible to use $R_{C_{max},est}$ in the fitness evaluation. $R_{C_{max},est}$

is calculated by the $R_{C_{max}}$-hillclimber used after decoding, and for the reasons explained in section 5.3.2, we expect $R_{C_{max},est}$ to be a good estimate of $R_{C_{max}}$. The running time of a variant of the GA using $R_{C_{max},est}$ in the fitness evaluation can be seen in the third column of table 5.5. It is evident that the $R_{C_{max},est}$-GA is much faster than the $R_{C_{max}}$-GA; for the larger problems the running time is typically decreased by 50% to 75%. The $R_{C_{max},est}$-GA is still significantly slower than the C_{max}-GA. The reason for the long running times of the $R_{C_{max}}$- and $R_{C_{max},est}$-GAs is the evaluation of makespan of a large number of schedules in $\mathcal{N}_1(s)$ for every fitness evaluation. The exact number of neighbours of a schedule depends on the schedule as well as the problem, but as a rule of thumb, the number of neighbours of a schedule is approximately 9 times the number of operations. This means that for a 10×10 problem, approximately 90 schedules need to be evaluated in every fitness evaluation.

Comparing the running time of the Z_r-GA to the other GAs, it turns out to be quite fast. It is almost as fast as the C_{max}-GA on most problems, and sometimes requires only half the CPU-time of the $R_{C_{max},est}$-GA on a given problem. The running time difference between the Z_r-GA and the $R_{C_{max},est}$-GA comes from the use of the $R_{C_{max}}$-hillclimber in the $R_{C_{max},est}$-GA; it is significantly slower than the C_{max}-hillclimber. The algorithms were implemented in `C` and ran on a 250 Mhz SGI O2 computer.

Turning to the quality of the solutions found by the $R_{C_{max},est}$-GA compared to the solutions found by the $R_{C_{max}}$-GA, experiments on the ten problems indicated that the $R_{C_{max}}$ values found by the $R_{C_{max},est}$-GA were slightly higher than the values found by the $R_{C_{max}}$-GA, see table 5.6.

The experiments with the $R_{C_{max},est}$-GA indicate that substantial amounts of CPU-time can be saved by using $R_{C_{max},est}$ in the fitness evaluation of the GA instead of calculating the exact value of $R_{C_{max}}$. This comes at a small cost in solution quality. It may be possible to create an algorithm combining the speed of the $R_{C_{max},est}$-GA with the precision of the $R_{C_{max}}$-GA by using $R_{C_{max},est}$ for fitness evaluation in the early stages of the algorithm and calculating the exact value of $R_{C_{max}}$ in the later stages. This idea is akin to the algorithms presented in [20, 65] and will not be pursued further in this thesis.

In order to further quantify the difference between schedules produced by minimising $R_{C_{max}}$ and $R_{C_{max},est}$, both algorithms will be used in the experiments later in the thesis.

5.3.12 Experiments on more problems

For each of the problems `la01-40`, `swv01-10` and `ta01-40`, a series of experiments were made. For each problem 400 schedules were created using each of the following algorithms:

problem	$R_{C_{max}}$-GA		$R_{C_{max},est}$-GA	
	best	mean	best	mean
la01	674.4	674.5	675.6	676.0
la02	682.7	684.5	683.3	687.8
la06	926.0	926.0	926.0	926.1
la07	890.9	892.6	891.2	893.3
la26	1231.7	1240.4	1231.9	1241.3
la27	1282.6	1296.2	1281.3	1297.1
la31	1784.1	1784.7	1784.2	1785.2
la36	1327.8	1342.9	1331.1	1349.4
ft10	987.2	998.6	988.0	999.8
ft20	1184.7	1203.4	1181.3	1203.2

Table 5.6: *The robustness measures of the solutions found in runs of the $R_{C_{max}}$ and $R_{C_{max},est}$-GAs. Averages of 400 runs.*

- The active C_{max}-GA - ordinary scheduling. In the following schedules produced by this algorithm will be called 'active' or C_{max}-minimised.
- The $R_{C_{max}}$-GA - scheduling based on the neighbourhood based robustness measure. Schedules produced by this algorithm will be called 'robust' or $R_{C_{max}}$-minimised.
- The $R_{C_{max},est}$-GA - scheduling based on the estimate of the neighbourhood based robustness measure. Schedules produced by this algorithm will be called 'fast robust' or $R_{C_{max},est}$-minimised.
- The $Z_r(s)$-GA - scheduling with the slack-based robustness measure. Schedules produced by this algorithm will be called 'slack based' or Z_r-minimised.

Each schedule was subjected to a random breakdown of the kind described in section 5.3.4 with duration 80, and rescheduled using each of the five rescheduling methods. For each problem, each of the above algorithms were run 400 times and average makespan performances and average overlaps to the preschedule were calculated. The makespan results are reported for each problem in appendix G.

In the following tables, average performances are presented for groups of problems. The problem groups have been formed from problems from the same test-suite having the same size. Thus, the problems `la01-40` have been split into eight groups of five problems, the `swv01-10` problems have been split into two groups of five problems, and the `ta01-40` problems have been split into four

groups of ten problems. The problems `ft10` and `ft20` have not been used in this study, since they would each constitute a group of just one problem.

In the table, there is a subtable for every problem group. Every subtable has four rows of entries, one for each scheduling algorithm. The row has been labelled "active" for the C_{max}-GA, "slack" for the Z_r-GA, "fast robust" for the $R_{C_{max},est}$-GA and "robust" for the $R_{C_{max}}$-GA. The columns labelled by a number represents a rescheduling method, the numbers are equal to the numbers they were given in section 5.3.5. The column labelled 'P' represents preschedule makespan before the breakdown.

In the tables, for every combination of problem group and rescheduling method (i.e. in every column), the lowest average has been printed in bold.

`la01-la05`, size: 10×5						
method	P	1	2	3	4	5
active	**620.41**	677.76	672.80	670.11	665.22	663.94
slack	624.81	**667.64**	665.74	664.77	662.15	661.12
fast robust	621.47	669.18	**665.42**	**664.56**	**662.14**	**660.99**
robust	621.37	669.43	666.36	665.67	663.22	661.98

`la06-la10`, size: 15×5						
method	P	1	2	3	4	5
active	**917.60**	964.50	958.11	954.03	949.78	949.36
slack	**917.60**	**941.38**	940.68	940.25	939.77	939.70
fast robust	**917.60**	941.45	**940.03**	**939.62**	**938.94**	**938.82**
robust	**917.60**	942.14	940.62	940.09	939.45	939.36

`la11-la15`, size: 20×5						
method	P	1	2	3	4	5
active	**1182.00**	1225.95	1220.25	1215.48	1212.28	1211.97
slack	**1182.00**	**1206.45**	1205.92	1205.77	1205.42	1205.35
fast robust	**1182.00**	1206.64	**1205.55**	**1205.16**	**1204.90**	**1204.89**
robust	**1182.00**	1208.77	1207.54	1207.08	1206.83	1206.81

`la16-la20`, size: 10×10						
method	P	1	2	3	4	5
active	**867.49**	922.83	916.04	913.28	**902.99**	**900.28**
slack	876.83	**917.61**	**913.63**	**912.16**	907.25	906.37
fast robust	873.44	919.40	914.76	912.91	906.23	904.25
robust	874.13	919.43	915.36	914.17	907.41	905.40

la21-la25, size: 15×10						
method	P	1	2	3	4	5
active	**996.08**	1052.83	1046.52	1042.26	1029.83	1026.73
slack	998.33	**1044.06**	1039.73	1037.57	1031.13	1028.84
fast robust	996.43	1044.54	**1039.44**	**1037.00**	**1028.67**	**1026.24**
robust	996.69	1044.63	1039.76	1037.58	1029.27	1027.15

la26-la30, size: 20×10						
method	P	1	2	3	4	5
active	1265.55	1322.17	1315.49	1310.37	1295.72	1292.41
slack	1268.24	1311.41	1307.84	1305.33	1295.97	1293.94
fast robust	**1261.70**	**1308.23**	**1303.18**	**1299.85**	**1289.97**	**1287.62**
robust	1261.88	1309.40	1304.80	1301.71	1291.42	1289.15

la31-la35, size: 30×10						
method	P	1	2	3	4	5
active	**1792.40**	1834.81	1826.68	1819.23	1811.94	1811.56
slack	1792.65	1811.19	1809.37	1808.05	1806.65	1806.49
fast robust	**1792.40**	**1809.43**	**1807.05**	**1805.60**	**1804.75**	**1804.69**
robust	**1792.40**	1810.13	1807.85	1806.61	1805.80	1805.73

la36-la40, size: 15×15						
method	P	1	2	3	4	5
active	**1300.49**	1351.63	1343.74	1339.19	1325.61	1322.09
slack	1305.94	1345.40	1340.33	1337.59	1329.86	1326.43
fast robust	1300.51	1342.67	**1335.71**	**1332.56**	**1323.16**	**1320.50**
robust	1301.50	**1342.18**	1336.75	1334.51	1325.57	1322.53

swv01-swv05, size 20×10						
method	P	1	2	3	4	5
active	1655.73	1712.67	1705.27	1698.56	1671.96	1667.69
slack	1646.56	1695.76	1691.23	1687.61	1665.30	1661.60
fast robust	**1628.66**	**1680.07**	**1673.58**	**1668.44**	**1646.54**	**1642.41**
robust	1632.12	1685.25	1680.04	1675.97	1652.85	1649.25

swv06-swv10, size 20×15						
method	P	1	2	3	4	5
active	1959.61	2014.51	2007.57	2001.02	1968.21	1963.11
slack	1949.19	1997.02	1992.16	1988.26	1961.58	1956.99
fast robust	**1929.36**	**1980.78**	**1974.27**	**1969.82**	**1943.61**	**1938.99**
robust	1932.83	1984.32	1978.22	1973.95	1947.15	1941.96

`ta01-ta10`, size: 15×15						
method	P	1	2	3	4	5
active	1268.59	1322.54	1314.81	1310.39	1295.86	1291.74
slack	1272.49	1315.31	1309.88	1306.94	1299.19	1295.90
fast robust	**1267.20**	**1314.07**	**1307.46**	**1304.47**	**1293.99**	**1290.38**
robust	1268.40	1314.41	1308.51	1305.96	1295.77	1292.27

`ta11-ta20`, size: 20×15						
method	P	1	2	3	4	5
active	1456.74	1508.10	1500.34	1494.61	1475.55	1471.20
slack	1457.31	1497.65	1492.61	1489.02	1477.07	1473.13
fast robust	**1443.36**	**1487.91**	**1481.54**	**1477.33**	**1463.67**	**1460.14**
robust	1445.80	1489.39	1483.55	1480.10	1466.54	1462.88

`ta21-ta30`, size: 20×20						
method	P	1	2	3	4	5
active	1725.12	1774.07	1764.99	1759.09	1737.38	1732.98
slack	1722.26	1762.43	1756.33	1752.34	1738.16	1733.96
fastrob	**1708.36**	**1751.31**	**1745.23**	**1741.55**	**1726.68**	**1722.86**
robust	1711.56	1754.13	1747.84	1744.03	1728.56	1724.68

`ta31-ta40`, size: 30×15						
method	P	1	2	3	4	5
active	1970.04	2014.72	2006.51	1998.87	1974.86	1969.55
slack	1962.49	1997.10	1992.53	1988.46	1971.12	1966.39
fastrob	**1926.80**	**1967.05**	**1961.60**	**1956.99**	**1939.54**	**1935.22**
robust	1930.09	1970.41	1965.00	1960.38	1942.80	1938.15

When inspecting the tables, it becomes clear at once that when considering averages over several problems, the slack-based, robust, and fast robust scheduling methods generally outperform the 'ordinary' active schedules when breakdowns happen. The only exception from this is the 10×10 group of the Lawrence test-suite, where active schedules outperform the other schedules for reduced and complete rescheduling. There is a tendency that for problems with a high job to machine ratio (15×5, 20×5, 20×10 and 30×10), the difference between active schedules and the other schedules is largest, while for problems with a moderate job to machine ratio (10×10, 15×10, 15×15) the difference is not so large. This observation is interesting because in static job shop scheduling it has been found that problems with a high job to machine ratio are generally easier to solve than other problems, [62]. The same effect seems to be present in stochastic job shop scheduling; it is easier to prepare problems with a high job to machine ratio for future events.

Inspecting the preschedule performance without breakdowns it is clear that for some of the problem groups the C_{max}-minimised schedules are superior to the others. This particularly seems to be the case for problems with a low job to machine ratio. For other problem groups (mostly smallish problems from the Lawrence suite), all the algorithms manage to find a minimal makespan schedule every time. In some of the groups (`swv01-05`, `swv06-10`, `ta11-20`, `ta21-30` and `ta31-ta40`), the preschedule makespans produced by the robust and fast robust algorithms are significantly lower than for the slack and active algorithms. This is probably because of the "look-ahead" effect of the robustness measure as discussed in section 5.3.3.

Comparing the slack-based, robust, and fast robust schedules, generally the fast robust schedules have the best performance for rescheduling methods 2-5 (the search based methods). For method 1 (right-shifting) the slack-based schedules have the best performance for smaller sized problems (up to 150 operations), while fast robust or robust schedules are a little better for the larger problems. In most cases in these experiments, the performance difference between slack-based, fast robust and robust schedules is quite small, and inspecting the results for individual problems in appendix G it becomes clear that what algorithm performs best often depends on the problem instance; the small differences in averages reported in the tables in this section cover a wide range of variation from problem to problem. Within each problem group, each of the slack-based, fast robust and robust scheduling methods are often found to be superior to the others on some of the problems. Thus, despite of the differences in averages reported above it seems premature to conclude that either of the slack-based, robust or fast robust scheduling methods are superior to the other in terms of schedule quality after rescheduling.

The only cases in which any of the three algorithms substantially outperform each other are the `swv01-05`, `swv06-10`, `ta11-20`, `ta21-30` and `ta31-40` groups, where the slack-based method performs significantly worse than the other two algorithms. However, the cause for this difference may be the better preschedule makespan of the robust and fast robust schedules. Also, for a number of the other problems and problem groups the robust and fast robust schedules outperform the slack-based schedules in terms of preschedule makespan.

Turning to the overlap performance after rescheduling, averages for the problem groups can be seen in the tables at the end of appendix G. The tables indicate that for right-shifting rescheduling, the slack-based, robust and fast robust schedules produce higher overlaps than the active schedules. It also seems the slack-based schedules may produce a marginally higher overlap than the robust and fast robust schedules. The same statements may hold for $\mathcal{N}_1$-based and hillclimbing rescheduling, although the differences from method to method are very small. For reduced and complete rescheduling the results found in section 5.3.6 do not al-

ways hold. In section 5.3.6 the robust schedules were found to never produce a lower overlap than active schedules in reduced and complete rescheduling. This is contradicted by the averages in section G, in which for several problem groups the highest overlap results for reduced and complete rescheduling are produced by active schedules. Inspecting the individual problems (not listed in the appendix), it becomes clear that also for the problem groups in which on average the robust schedules produce the highest overlap there are problem instances for which the opposite is true. The difference between the experiments used here and in section 5.3.6 is that in section 5.3.6 the population was not reused in reduced and complete rescheduling, while here it is. In section 5.3.8 reusing the population was found to increase the overlap after rescheduling, and the experimental results here indicate that the overlap is increased even more for active schedules than for robust schedules.

The conclusion on the experiments for overlap must be that for right-shifting, $\mathcal{N}_1$-based, and hillclimbing rescheduling the slack-based, robust and fast robust schedules are better than the active schedules. As for the makespan, it is premature to make any conclusion when comparing the slack-based, robust and fast-robust schedules, since the variations from problem to problem are large. Concerning reduced and complete rescheduling it seems difficult to conclude anything based on the experiments; there is much variation in the averages from problem group to problem group, and even more from problem instance to problem instance.

Comparing the five rescheduling methods to each other the expectations from section 5.3.5 turn out to be right. In terms of schedule makespan after rescheduling, the methods with higher indexes (larger search-spaces) generally produce lower makespans. In terms of preschedule overlap, it seems that generally speaking the best rescheduling methods are $\mathcal{N}_1$-based rescheduling (2), hillclimbing rescheduling (3), and reduced rescheduling (4), while right-shifting (1) and complete rescheduling (5) seem inferior in this respect. However, in applications where similarity to the preschedule is very important, it may make more sense to use rescheduling methods that explicitly consider preschedule similarity as well as schedule cost.

5.4 Experiments on maximum tardiness

In this section it is investigated if the neighbourhood based robustness approach can be used on maximum tardiness problems. The tightness of the scheduling problems may influence the usefulness of the robustness measures, so this influence of problem tightness is also investigated.

Section 5.4.1 discusses how to achieve robust and flexible schedules for tardiness problems and describes the algorithms used. The results are presented in

sections 5.4.2-5.4.4.

5.4.1 Algorithms for maximum tardiness

When creating schedules for tardiness problems facing breakdowns, it may be worthwhile to minimise the lateness of the schedules instead of the tardiness (recall that $T_i = \max(L_i, 0)$, so minimising lateness will always minimise tardiness too). If T_{max} is minimised, the minimisation process will stop once $T_{max} = 0$ is reached. If L_{max} is minimised, L_{max} will be minimised even if $L_{max} < 0$. This is likely to improve dynamic performance of the schedule, since minimising L_{max} beyond $L_{max} < 0$ will add more slack to the schedule. This slack can be thought of as a "buffer time"; if the schedule has e.g. $L_{max} < -20$, then every part of the schedule can be delayed by 20 time-units while still achieving the best possible tardiness performance, $T_{max} = 0$. Note that this approach can only be expected to improve dynamic performance for loose problems; if no solution exists for which $L_{max} < 0$, minimising L_{max} is equivalent to minimising T_{max}.

For this reason three kinds of schedules will be compared in this section: schedules produced by minimising T_{max}, L_{max} and $R_{L_{max}}(s)$, where

$$R_{L_{max}}(s) = \sum_{s' \in \mathcal{N}_1(s)} \frac{1}{\mathcal{N}_1(s)} L_{max}(s'). \tag{5.6}$$

Note that for the same reasons stated above the robustness measure is based on L_{max} instead of T_{max}. It should be relatively straight-forward to define a slack-based robustness measure like the one defined for makespan in [72] (see section 4.7.2) for maximum tardiness problems, but so far this has not been done.

The program used for makespan problems were modified to minimise the three performance measures. The implementation was unchanged except for the following details:

- Instead of the grid-like population structure (diffusion model), a simple unstructured population was used. Thus, the behavioural model was abandoned as well. The algorithm used a simple unstructured population with tournament selection, tournament size two. Generational replacement of the entire population was used. This change was made because in preliminary experiments it was found to outperform the diffusion based GA for some problems. It was later found that for other problems the diffusion GA was superior to the simple GA. For hard problems the diffusion GA produces the better performance, while for easier problems the basic GA more consistently finds the optimum.

- All new individuals were generated using GOX crossover and mutated using PBM with a probability of 0.1.

- Elitism was not used, but the all time best individual was recorded and returned at the end of the run.

- The C_{max}- and $R_{C_{max}}$-hillclimbers were modified to work on L_{max} and $R_{L_{max}}$ as described in appendix F.

As with the algorithms working on makespan, the result returned by the tardiness algorithms were improved by a $\mathcal{N}_1$-hillclimber working "on top of" the decoding hillclimber. The schedules returned by the algorithms minimising T_{max} and L_{max} were transformed into active schedules in the way described in section 5.3.3.

5.4.2 Running the experiments

The experiments were carried out on the Lawrence test-suite (`la01-40`). This test suite was preferred because it was found to give the most valuable results for the makespan experiments, since the results were not distorted by superior makespan performance of the robust schedules, as was the case for some of the other test-suites. Since the Lawrence test-suite does not have due-dates, these were added as described in section 3.4.

The experiments were carried out on two versions of the generated problems: the loose problems ($\sigma = 0.95$) and the tight problems ($\sigma = 0.85$). For each problem, 400 preschedules were made using each of the three algorithms, and rescheduling was done using the five rescheduling methods. For each problem average performance after rescheduling was calculated for each combination of scheduling algorithm and rescheduling method. The averages for each problem are reported in appendix H.

5.4.3 The loose problems $\sigma = 0.95$

As in the makespan experiments, the problems were grouped according to size. The following table is organised in the same way as the table in section 5.3.12. The algorithm minimising T_{max} has been labelled "naive", the algorithm minimising L_{max} "active" and the algorithm minimising $R_{L_{max}}$ "robust".

`la01-la05`, size: 10×5						
method	P	1	2	3	4	5
naive	**6.00**	56.50	51.40	48.49	41.51	40.39
active	**6.00**	47.48	42.87	40.34	**36.00**	**35.49**
robust	**6.00**	**44.52**	**40.49**	**39.76**	37.21	35.91

la06-la10, size: 15×5						
method	P	1	2	3	4	5
naive	**18.20**	66.80	60.45	55.27	44.42	43.96
active	**18.20**	52.80	48.82	45.96	40.38	40.08
robust	18.28	**41.48**	**39.60**	**39.48**	**37.83**	**37.16**
la11-la15, size: 20×5						
method	P	1	2	3	4	5
naive	**21.20**	76.63	71.38	67.01	53.35	52.95
active	**21.20**	67.83	63.88	60.08	50.69	50.46
robust	21.30	**50.95**	**47.90**	**46.63**	**43.99**	**43.81**
la16-la20, size: 10×10						
method	P	1	2	3	4	5
naive	**0.03**	44.93	38.65	35.72	27.45	26.26
active	**0.03**	32.61	28.28	26.07	**20.62**	**19.59**
robust	0.04	**29.58**	**25.81**	**24.73**	22.93	21.19
la21-la25, size: 15×10						
method	P	1	2	3	4	5
naive	**0.00**	36.83	29.71	25.07	13.03	12.47
active	**0.00**	6.50	5.21	4.52	2.59	2.52
robust	**0.00**	**5.17**	**4.05**	**3.71**	**2.47**	**2.25**
la26-la30, size: 20×10						
method	P	1	2	3	4	5
naive	**0.00**	35.66	28.17	22.51	9.35	8.98
active	**0.00**	0.47	0.37	0.33	**0.19**	**0.18**
robust	**0.00**	**0.29**	**0.21**	**0.21**	0.21	0.19
la31-la35, size 30×10						
method	P	1	2	3	4	5
naive	**0.00**	33.35	26.47	20.72	8.21	8.15
active	**0.00**	6.05	4.67	3.94	2.42	2.55
robust	**0.00**	**2.17**	**1.94**	**1.84**	**1.38**	**1.38**
la36-la40, size 15×15						
method	P	1	2	3	4	5
naive	0.10	38.16	30.09	25.06	12.27	11.80
active	**0.08**	10.46	8.15	7.03	**4.29**	**4.13**
robust	0.23	**9.32**	**7.51**	**6.93**	5.24	4.87

When inspecting the tables it becomes clear that for all of the loose problems ($\sigma = 0.95$), the standard scheduling method optimising T_{max} is outperformed by the other two methods for all rescheduling methods. From the single problem averages of appendix H this can be seen to hold not only for the averages over

problem groups, but for every single problem instance.

Comparing the performance of the neighbourhood based robustness measure $R_{L_{max}}$ to the performance of the simple robustness measure L_{max}, it seems that for the simple rescheduling methods (labelled 1-3), the neighbourhood based robustness measure generally performs best, although there are a few exceptions, especially for the problem sizes 10×5 and 15×15. The performance difference is largest for the problems with a high job to machine ratio; for problem sizes $15 \times 5, 20 \times 5, 20 \times 10$, and 30×10 there are substantial differences between the two methods. Considering the complex rescheduling methods 4 and 5, the neighbourhood based robustness measure slightly outperforms the L_{max} measure for problem sizes 15×5 and 20×5, while for the other problems, the L_{max} and $R_{L_{max}}$ methods do equally well. With respect to preschedule performance, the T_{max} and L_{max} methods seem to do equally well, while the $R_{L_{max}}$ method performs slightly worse than these.

5.4.4 The tight problems $\sigma = 0.85$

The following table holds the results for the tight problems.

`la01-la05`, size 10×5						
method	P	1	2	3	4	5
naive	50.83	111.15	106.54	103.98	**99.03**	**97.94**
active	**50.79**	112.75	108.32	105.61	99.74	98.68
robust	53.63	**106.74**	**102.80**	**102.02**	99.58	98.22

`la06-la10`, size 15×5						
method	P	1	2	3	4	5
naive	84.84	138.88	133.21	128.88	118.71	118.05
active	**84.83**	133.12	127.71	124.22	115.91	115.34
robust	85.81	**114.73**	**112.49**	**112.15**	**110.56**	**109.64**

`la11-la15`, size 20×5						
method	P	1	2	3	4	5
naive	129.86	187.55	182.33	177.93	166.30	165.92
active	**129.68**	185.97	180.87	176.72	165.15	164.77
robust	130.44	**169.53**	**165.80**	**163.86**	**160.28**	**159.80**

`la16-la20`, size 10×10						
method	P	1	2	3	4	5
naive	**59.40**	113.43	107.11	**104.43**	**97.60**	**95.72**
active	**59.40**	113.83	108.20	105.38	98.79	97.02
robust	64.78	**111.47**	**106.85**	105.02	102.08	99.72

`la21-la25`, size 15×10						
method	P	1	2	3	4	5
naive	26.27	80.67	74.25	70.52	59.15	56.90
active	**26.18**	76.34	70.78	67.14	57.35	55.60
robust	27.43	**69.18**	**65.17**	**63.41**	**57.24**	**55.09**

`la26-la30`, size 20×10						
method	P	1	2	3	4	5
naive	28.77	84.54	77.97	73.19	61.17	59.21
active	**28.68**	83.69	77.19	72.29	60.17	57.86
robust	28.72	**72.64**	**67.63**	**65.41**	**59.39**	**57.17**

`la31-la35`, size 30×10						
method	P	1	2	3	4	5
naive	103.99	154.09	147.11	140.63	127.07	126.04
active	104.11	153.82	146.32	139.97	126.10	126.05
robust	**97.27**	**129.08**	**125.80**	**123.90**	**116.30**	**115.82**

`la36-la40`, size 15×15						
method	P	1	2	3	4	5
naive	74.86	126.81	119.39	114.72	**103.71**	**101.92**
active	**74.79**	127.11	119.33	114.55	103.80	102.00
robust	80.49	**120.32**	**115.10**	**113.16**	107.84	105.59

When comparing the T_{max} and L_{max} methods on the tight problems, there is still a small performance advantage of the L_{max} method in a few cases, but by and large the two methods perform equally well. The $R_{L_{max}}$ method can be seen to outperform the other methods in most cases for the simple rescheduling methods 1-3. With regard to rescheduling methods 4 and 5, for the problems with a high job to machine ratio it usually outperforms the other methods, demonstrating an improved flexibility of the schedules. For problems with a low job to machine ratio the $R_{L_{max}}$ method is outperformed by the other methods in a few cases.

5.5 Experiments on total tardiness

For the total tardiness experiments, the performance measures $T_{\sum}$, $L_{\sum}$ and $R_{L_{\sum}}$ were used. The summed tardiness $T_{\sum}$ is what would be minimised by a standard algorithm working on a problem like this. Minimising the total lateness $L_{\sum}$ does not necessarily mean minimising $T_{\sum}$, but like in the maximum tardiness experiments it may make sense to minimise lateness rather than tardiness, since it may increase the schedule slack. The $R_{L_{\sum}}$ measure is the neighbourhood based

robustness measure defined for summed lateness,

$$R_{L_\Sigma}(s) = \sum_{s' \in \mathcal{N}_1(s)} \frac{1}{\mathcal{N}_1(s)} T_\Sigma(s'). \tag{5.7}$$

The algorithms used in the total tardiness experiments were identical to the ones used on the maximum tardiness experiments, except for the objective functions. Since no suitable hillclimber for rescheduling total tardiness problems was available, no hillclimbing rescheduling was performed in the experiments.

Experiments were done with algorithms using the active/non-delay schedule builder of figure 3.9, since this seems more reasonable than using the hillclimbing decoder minimising maximum lateness used in the maximum tardiness experiments. However, it was found that the algorithm using the hillclimbing decoder produced better schedules than the algorithm using the active/non-delay decoder, so the hillclimbing decoder was used in the experiments. This is remarkable, since the hillclimbing decoder minimises a different (but related) performance measure.

5.5.1 Loose problems $\sigma = 0.95$

The results of the total tardiness experiments for individual problems can be found in appendix I. The total tardiness averages for problem groups are in the following table. It has been organised like the table of section 5.3.12. The algorithms have been labelled "naive" for the T_Σ-GA, "active" for the L_Σ-GA and "robust" for the R_{L_Σ}-GA.

`la01-la05`, size 10×5					
method	P	1	2	4	5
naive	16.46	200.27	155.62	102.98	98.63
active	**16.45**	161.92	127.23	**88.43**	**85.15**
robust	17.13	**148.12**	**119.48**	91.44	87.93
`la06-la10`, size 15×5					
method	P	1	2	4	5
naive	55.57	250.32	202.63	126.77	124.70
active	**55.05**	204.79	173.47	122.03	120.23
robust	59.42	**160.52**	**142.69**	**117.00**	**115.43**
`la11-la15`, size 20×5					
method	P	1	2	4	5
naive	**61.67**	340.16	279.36	160.21	157.79
active	61.72	284.55	238.67	151.33	149.25
robust	62.48	**217.68**	**188.45**	**140.52**	**140.01**

`la16-la20`, size 10×10					
method	P	1	2	4	5
naive	**0.01**	167.32	121.22	54.66	50.38
active	**0.01**	93.51	71.92	**36.03**	**33.32**
robust	1.04	**89.94**	**69.77**	44.18	42.10

`la21-la25`, size 15×10					
method	P	1	2	4	5
naive	**0.00**	152.94	105.52	23.60	21.35
active	0.01	26.41	17.46	**4.36**	**4.03**
robust	**0.00**	**18.69**	**12.84**	5.36	5.22

`la26-la30`, size 20×10					
method	P	1	2	4	5
naive	**0.00**	167.59	114.47	16.97	16.02
active	**0.00**	6.10	4.12	0.58	0.58
robust	**0.00**	**2.04**	**1.37**	**0.39**	**0.32**

`la31-la35`, size 30×10					
method	P	1	2	4	5
naive	0.01	189.21	126.54	14.67	14.23
active	0.01	20.34	13.20	4.10	4.26
robust	**0.00**	**8.39**	**6.66**	**3.24**	**3.56**

`la36-la40`, size 15×15					
method	P	1	2	4	5
naive	0.43	156.45	103.74	22.45	18.89
active	**0.23**	48.85	32.72	**8.84**	**7.77**
robust	1.10	**39.71**	**28.44**	13.44	11.39

For the loose problems, the results of the summed tardiness problems are resemblant of the maximum tardiness problems. The $R_{L_\sum}$ robustness measure seems to improve the performance of rescheduling methods 1 and 2 more than the $L_\sum$ measure on all problems, and for problems with a high job/machine ratio, $R_{L_\sum}$ often outperforms $L_\sum$ on the performance of rescheduling methods 4 and 5 as well. In some cases this comes at a cost of a slight increase in preschedule cost. The $L_\sum$ measure outperforms the standard $T_\sum$ measure as well, but not as much as $R_{L_\sum}$.

The similarity in behaviour between loose summed tardiness problems and maximum tardiness problems does not come as a surprise; in a loose summed tardiness problem often only one or a few jobs will be tardy. In this case minimising maximum tardiness and summed tardiness are almost the same.

5.5.2 Tight problems $\sigma = 0.85$

The following table holds the results of the summed tardiness experiments on tight problems.

`la01-la05`, size 10×5					
method	P	1	2	4	5
naive	201.67	493.76	431.79	356.45	346.60
active	**201.28**	497.38	434.39	**354.54**	**346.08**
robust	218.36	**472.83**	**420.95**	364.32	356.43

`la06-la10`, size 15×5					
method	P	1	2	4	5
naive	372.76	674.76	614.95	504.12	493.16
active	**371.92**	649.19	596.55	**499.86**	**490.04**
robust	380.50	**605.49**	**567.11**	501.38	493.88

`la11-la15`, size 20×5					
method	P	1	2	4	5
naive	679.62	1127.66	1051.29	862.61	848.10
active	679.30	1109.58	1036.06	**861.15**	**843.78**
robust	**679.20**	**1059.72**	**1000.89**	863.39	852.24

`la16-la20`, size 10×10					
method	P	1	2	4	5
naive	273.61	555.18	**504.39**	412.51	**396.08**
active	**269.91**	565.29	509.84	**412.24**	398.68
robust	302.30	**548.48**	505.65	439.16	426.47

`la21-la25`, size 15×10					
method	P	1	2	4	5
naive	138.26	499.22	422.97	268.51	256.31
active	**132.50**	457.60	390.96	**259.38**	**245.35**
robust	144.01	**410.93**	**363.26**	265.80	253.96

`la26-la30`, size 20×10					
method	P	1	2	4	5
naive	168.61	658.64	554.98	**307.46**	**288.49**
active	167.30	663.01	563.55	313.80	293.42
robust	**166.93**	**560.41**	**489.56**	310.42	293.67

`la31-la35`, size 30×10					
method	P	1	2	4	5
naive	946.47	1648.58	1483.74	1048.71	1015.21
active	945.33	1647.68	1489.25	1046.61	1021.07
robust	**744.32**	**1209.72**	**1112.03**	**862.58**	**846.74**

`la36-la40`, size 15×15					
method	P	1	2	4	5
naive	**448.17**	876.32	785.94	**600.21**	**576.84**
active	449.21	869.33	779.75	606.21	581.87
robust	490.73	**824.72**	**758.34**	633.21	605.42

The results for the tight problems differ qualitatively from the results on the loose problems. For rescheduling methods 1 and 2, the schedules produced by using $R_{L_\sum}$ still seem superior to those produced using $L_\sum$. For rescheduling methods 4 and 5 there is only little difference between the performances of the $L_\sum$ schedules and the $T_\sum$ schedules, while in several instances the $R_{L_\sum}$ schedules perform substantially worse than the other schedules, indicating that for tight summed tardiness problems, flexibility can be worsened by using this robustness measure.

In the experiments, the use of the $L_\sum$ performance measure did not degrade the preschedule cost when compared to the $T_\sum$ experiments. For the tight problems, the use of the $R_{L_\sum}$ measure increased preschedule cost in many cases, although there were a few intriguing exceptions in which the preschedule costs found by using $R_{L_\sum}$ were much lower than the preschedule costs found using the other performance measures.

5.6 Experiments on total flow-time

For the experiments on total flow-time schedules produced by minimising the total flow-time F were compared to schedules produced by minimising the neighbourhood based robustness measure

$$R_{F_\sum}(s) = \sum_{s' \in \mathcal{N}_1(s)} \frac{1}{\mathcal{N}_1(s)} F(s'). \tag{5.8}$$

The algorithms used were similar to those used for the maximum tardiness experiments, except that the active/non-delay schedule builder of figure 3.9 was used for decoding with the parameter $\delta = 0.5$, since this value was found to give good results in [16]. Because no suitable hillclimber for rescheduling total flow-time problems was available, no hillclimbing rescheduling was performed in the experiments.

The following tables report averages over problem groups for the Lawrence test-suite. The table has been constructed in the same way as the table in section 5.3.12. The results for individual problems are reported in appendix J. The label "active" has been used for the algorithm minimising F, while "robust" has been used for the algorithm minimising $R_{F_\sum}$.

`la01-05`, size 10×5					
method	P	1	2	4	5
active	**4572.5**	4947.6	4896.0	**4783.0**	**4766.7**
robust	4591.2	**4933.0**	**4886.4**	4797.8	4784.9

`la06-10`, size 15×5					
method	P	1	2	4	5
active	**9453.3**	10012.3	9931.7	**9702.8**	**9674.7**
robust	9481.1	**9991.2**	**9916.7**	9728.7	9705.6

`la11-15`, size 20×5					
method	P	1	2	4	5
active	15590.4	16344.8	16231.0	15818.1	15758.1
robust	**15538.5**	**16196.7**	**16099.1**	**15782.1**	**15744.6**

`la16-20`, size 10×10					
method	P	1	2	4	5
active	**7451.5**	7842.4	7774.4	**7638.9**	**7616.0**
robust	7461.0	**7826.3**	**7764.4**	7650.5	7631.7

`la21-25`, size 15×10					
method	P	1	2	4	5
active	**12925.0**	13513.1	13414.3	**13170.9**	**13127.9**
robust	12945.6	**13455.7**	**13370.7**	13180.3	13145.8

`la26-la30`, size 20×10					
method	P	1	2	4	5
active	21243.8	22027.2	21885.6	21476.6	21382.5
robust	**21230.9**	**21885.9**	**21772.9**	**21448.5**	**21376.5**

`la31-35`, size 30×10					
method	P	1	2	4	5
active	42390.9	43501.1	43300.3	42493.3	42366.6
robust	**42353.1**	**43097.5**	**42938.2**	**42320.0**	**42200.9**

`la36-40`, size 15×15					
method	P	1	2	4	5
active	**17067.9**	17657.9	17545.3	**17280.8**	**17238.7**
robust	17096.2	**17617.1**	**17522.4**	17308.4	17265.5

Qualitatively, the results of the total flow-time experiments are similar to the results for the tight summed tardiness problems. For rescheduling using methods 1 and 2, the R_{F_Σ} runs are slightly superior, while in many cases they are inferior when methods 4 and 5 are used. With respect to preschedule performance, the two methods had comparable performances. For some problem sizes F seems to be better, while on others R_{F_Σ} seems to do best.

5.7 Discussion

In section 5.2 it was stated that the idea behind the neighbourhood based robustness measure was that when a set of schedules close to the preschedule are known to have a good performance, maybe when a breakdown happens one of them can work around the breakdown, facilitating rescheduling. This explanation may account for the relatively good performance of the robust schedules when $\mathcal{N}_1$-based rescheduling is used, since a move in $\mathcal{N}_1$ made "downstream" from the breakdown may in some cases alleviate the effects of the breakdown by rushing a critical part of the schedule while delaying a non-critical part. This explanation may also hold in part for hillclimbing rescheduling. Since the $\mathcal{N}_1$-neighbourhood is a part of the neighbourhood used by the hillclimber, a set of moves in $\mathcal{N}_1$ known not to be too bad before a disruption may have a higher chance of turning into good starting moves for the hillclimber after a disruption than a set of very poor $\mathcal{N}_1$ moves.

However, this "probably one good move in $\mathcal{N}_1$" argument does not explain the good performance achieved for right-shifting, reduced, and complete rescheduling on makespan and loose tardiness problems.

The slack hypothesis

When comparing a schedule obtained by minimising a neighbourhood based robustness measure to an "ordinary" schedule, the robust schedule can be expected to have more slack than the ordinary schedule, since the operation swaps considered in the robustness measure lead to small delays in the schedule. In order to have a low robustness measure, the schedule is required to still have a good performance despite these delays, which translates into saying that the schedule must have slack.

When turning to the good performance of reduced and complete rescheduling, the increased slack in the schedules may account for some of the improvement over ordinary schedules (since the slack-based schedules also perform well for these rescheduling methods), but probably not all of it. It is hard to decide how much of a factor the increased slack in the schedule is for the complex rescheduling methods, since the location of slack in the schedules probably means just as much as the amount. However, by inspecting the results of individual problem instances in the appendixes it can be seen that very often if a scheduling method is superior to the other scheduling methods for right-shifting, the same is the case for the other rescheduling methods. Since the explanation for the good performance of the schedule in right-shifting can only be slack, these schedules must have a high amount of well-placed slack. These observations support a hypothesis that the improved flexibility of the schedules could be due to slack. In the following this hypothesis will be termed *the slack hypothesis*.

The conflicting arcs hypothesis

Another possible explanation for the good performance for reduced and complete rescheduling could be that the neighbourhood based robustness approach reduces the number of *potentially conflicting arcs* in the solutions. A set of *conflicting arcs* is a choice of orientations for some of the disjunctive arcs in the graph representation that will always lead to a poor schedule, regardless of the orientation of the other disjunctive arcs. A simple example of conflicting arcs can be found in figure 5.14. The four schedules in the figure are all solutions to the same problem. The four schedules are the only active schedules possible for this problem. The low makespan schedules A and B have no conflicting arcs, while the schedules C and D have a high makespan, because the arcs in these schedules are in conflict. It is interesting to note that the disjunctive arcs of the bad schedules are also present in the good schedules (e.g. the disjunctive arc pointing up in schedule D is present in schedule A). For this problem the quality of the schedule is not determined by the presence of individual arcs alone, but by the combination of several arcs.

In figure 5.15, a more complex example of conflicting arcs is presented. The figure has two graph representations and four schedules. Each of the graph representations has an undecided disjunctive arc (in what follows, this arc will be referred to as *arc v*), and thus represents both the schedules in its row. The three first schedules (A, B and C) have no conflicting arcs, and they all have the optimal makespan for this problem. Schedule D has a conflict between the arcs of machines 3 and 4, and has a suboptimal makespan. From a makespan point of view, schedules A, B and C are equally good. However, if we consider how flexible the schedules are with respect to changing the orientation of the disjunctive arc left undecided in the graphs, it seems clear that A and B are preferable to C. If the orientation of arc v is changed partway through the processing of schedule A, the result will be schedule B, which is still optimal. In the same way, schedule B can easily be changed into schedule A partway through processing. However, if for some reason arc v needs to be reversed partway through processing schedule C, the result will be the suboptimal schedule D. Thus, schedule C can be said to have a potential conflict between arc v and the disjunctive arc oriented upwards of machine 4. This indicates that schedule C is in some sense less flexible than schedules A and B. For this simple example, the ability to reverse arc v without increasing makespan does not lead to improved rescheduling performance. Larger examples in which this is the case can be constructed.

Since the robustness measure reverses the orientation of every disjunctive arc in the graph representation of the solution, the number of potentially conflicting arcs will be minimised when the robustness measure is minimised. Because the presence of all but one of the arcs of a conflicting arc set will lead to a set of conflicting arcs (and thus a bad schedule) when the robustness measure is evaluated

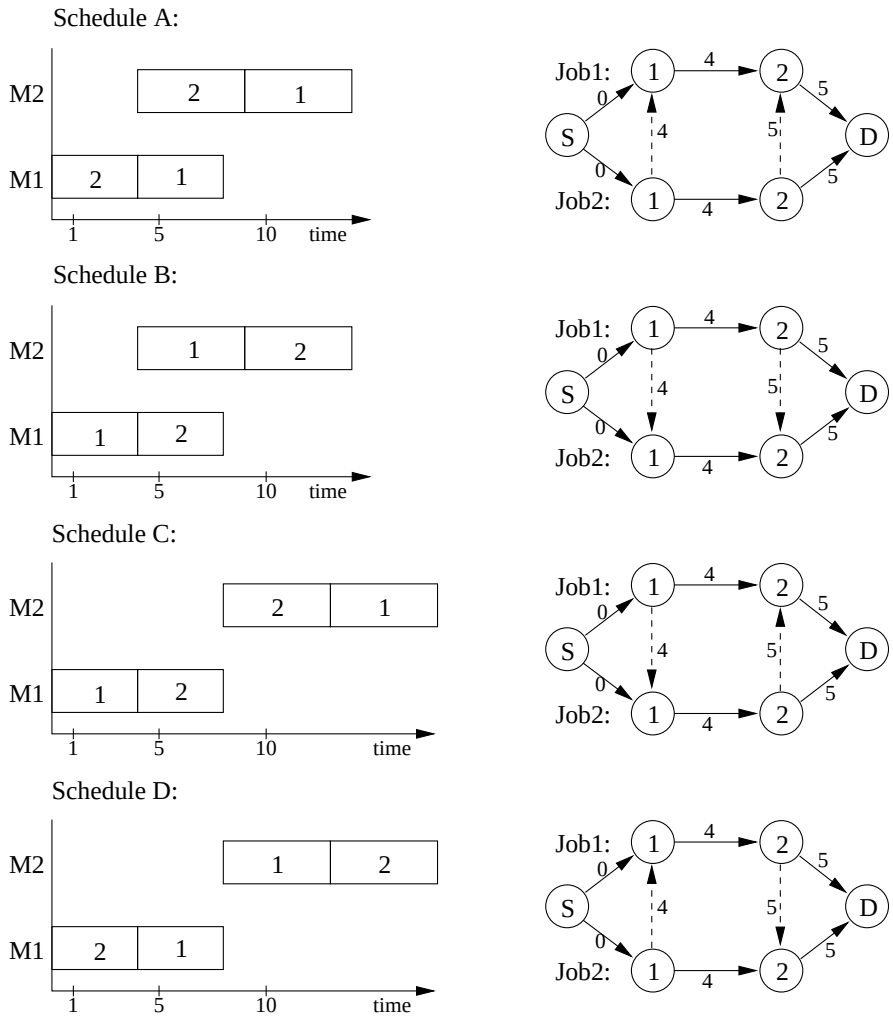

Figure 5.14: *Simple example of conflicting arcs. Schedules A and B have no conflicting arcs and have a low makespan. Schedules C and D have conflicting arcs and a high makespan.*

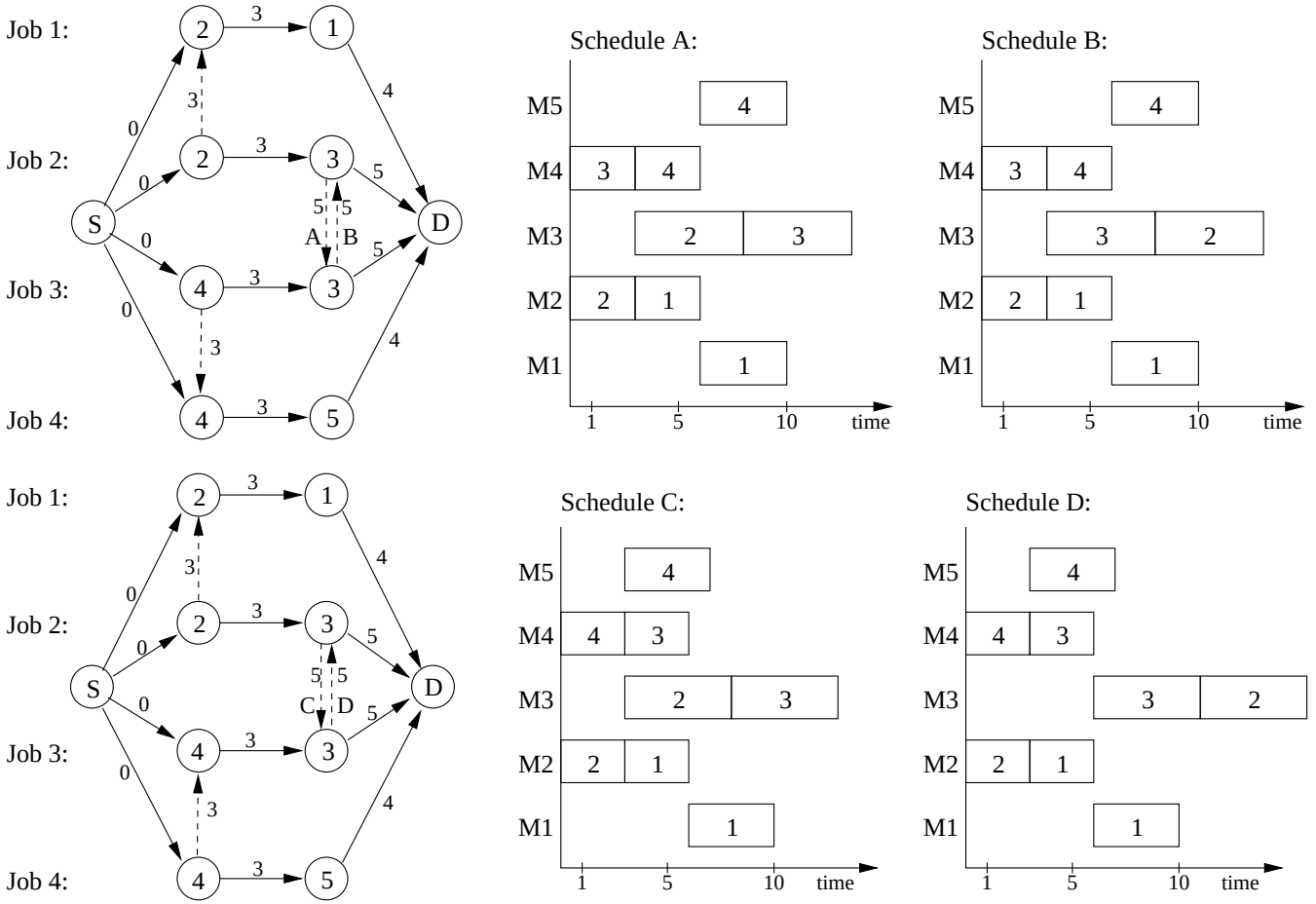

Figure 5.15: *More complex example of conflicting arcs. Schedules A, B and C have no conflicting arcs. Schedule D has a conflicting arc set and a lower performance than the other schedules.*

the presence of sets of potentially conflicting arcs will be minimised. Since the rescheduling problem is derived from the original problem and the preschedule, this probably means the rescheduling problems end up having fewer potentially conflicting arc sets, meaning they are easier to solve. Below this hypothesis will be termed *the conflicting arcs hypothesis*[3]. The hypothesis is supported by the fact that when the population is not reused, the new schedules found after reduced or complete rescheduling end up being more similar to the preschedule when $R_{C_{max}}$ is minimised than when C_{max} is minimised. A good solution close to the preschedule processing order probably has a larger basin of attraction, if the preschedule processing order has few conflicting arcs, than if it has many.

The conflicting arcs hypothesis is related to the PFSL approach presented in [114] (see section 4.7.3). In this approach a few critical decisions are made in a preprocessing step, and detailed scheduling is done by dispatching rules while the schedule is being executed. The basis for this approach is a conjecture that the flexibility of a schedule depends on a few critical decisions (arcs) while the rest of the schedule is less important in this respect. The notion of trying to avoid potentially conflicting arcs in the schedules in some sense is exactly the same; avoiding potentially conflicting arcs means trying not to make the decisions that impose unnecessary constraints on the rest of the solution.

Why does neighbourhood based robustness work?

For tight summed tardiness and total flow-time problems, the neighbourhood based approach has been found not to improve the performance of reduced and complete rescheduling. Starting from the conflicting arcs hypothesis, the reason for this may be that problems of this kind have more conflicting arcs, making it impossible to find a solution with few conflicting arcs. Problems of this kind probably contain more conflicting arcs than makespan and maximum tardiness problems, since it is very hard to change anything in a total flow-time or tight summed tardiness schedule without changing the performance.

Regarding the slack hypothesis, it can be argued that since it is very hard to change anything in a tight summed tardiness or total flow-time problem, it should be nearly impossible to create schedules with slack (indeed, even how to make a reasonable definition of slack for these problems is not straight-forward). This statement is disproved by the superior performance of the R_{L_Σ}- and R_{F_Σ}-minimised schedules when right-shifting rescheduling is used. The superior performance clearly indicates that some kind of slack is present in these schedules. Despite this the R_{L_Σ}- and R_{F_Σ}-minimised schedules often turn out to be inferior

[3]The conflicting arcs hypothesis was originally suggested in a different form by an anonymous reviewer at *IEEE Transactions on Evolutionary Computation*.

to ordinary schedules when reduced or complete rescheduling is used, indicating that the slack is not helpful for rescheduling in this case.

Based on the above considerations, we feel that the most probable explanations for the good performance of the neighbourhood based approach are

- For right-shifting: increased slack in the schedules.
- For $\mathcal{N}_1$-based and hillclimbing rescheduling: The increased probability of good moves in $\mathcal{N}_1$.
- For reduced and complete rescheduling: The conflicting arcs hypothesis.

It is also worthwhile to keep in mind that the different explanations need not exclude each other. The conflicting arcs hypothesis and the slack hypothesis can both be true, and probably the increased slack has some effect for all of the complex rescheduling methods.

When comparing the neighbourhood based approach to the slack-based approach from [72], the experiments indicate that they have more or less equal levels of performance on makespan problems for all the rescheduling methods, while the neighbourhood based approach was found to often outperform the slack-based approach in terms of preschedule makespan. The slack-based approach is extendable to maximum tardiness problems, and could be expected to have more or less the same level of performance on these problems as the neighbourhood based approach. The running time of the slack-based algorithm is substantially lower than the neighbourhood based algorithm, so in the choice between these two approaches the choice basically comes down to preferring lower running time or better preschedule performance[4]. An argument in favour of the slack-based algorithm is the fact that it is better understood than the neighbourhood based approach.

When considering problems for which the notion of slack is difficult to define and work with (e.g. summed tardiness and total flow-time problems), the neighbourhood based approach is highly relevant, since it has been demonstrated to improve robustness and to some extent flexibility for this kind of problem over standard scheduling, and since to my best knowledge no other approach has been published.

Based on the experiments of this chapter and the above considerations, the neighbourhood based robustness approach can be expected to generally improve the robustness quality of schedules, while flexibility with respect to search-based rescheduling methods is not always improved. The flexibility can be conjectured to improve for problems in which the solutions contain few *critical parts*, parts of

[4]If the slack-based algorithm is allowed to run for more generations, it may reach the level of preschedule performance of the neighbourhood based algorithm.

the solution that are essential for the quality. In problems with few critical parts it will often be possible to make small changes to non-critical parts of the solution without ruining the solution. This property is probably necessary if potentially conflicting arcs are to be removed from the solutions.

Possible future work

If the conflicting arcs hypothesis is the reason for the improved rescheduling performance observed in this chapter, it may be an indication that more clever ways of creating flexible schedules can be conceived. The neighbourhood based robustness approach does not consider the temporal relationships between the potentially conflicting arcs. If a disjunctive arc v will conflict with a set of other disjunctive arcs W if v is reversed, the implications for schedule flexibility depends on the time at which v is implemented in the schedule relative to the implementation times of the arcs in W. If v is implemented earlier than or at the same time as the arcs in W, the implications on schedule flexibility may not be as severe as they will be if the arcs in W are implemented before v. The size of the conflicting arc set W may also be important; if W is large, there are many arcs of which maybe just one needs to be reversed in order to allow a reversal of v. If W is small, fewer possibilities exist for allowing v to be reversed.

A scheduling method taking into account temporal relationships and sizes of conflict sets will probably have to consider the conflict sets in an explicit way, rather than in the implicit way of the neighbourhood based robustness approach. If conflicting arcs are to be considered explicitly, probably a completely different kind of algorithm is needed. A branch and bound algorithm could be useful, since traditional branch and bound algorithms for scheduling are working with partial schedules and establish lower bounds on these, something which is closely related to finding conflicting sets of arcs.

Chapter 6

Worst Case Performance Scheduling with Coevolution

In chapter 5, job shop scheduling facing disruptions was considered from an average performance point of view. In this chapter the same problem will be treated from a worst case and worst deviation point of view. Recall that in worst case performance the scheduler is interested in minimising the consequences of the worst possible conditions, while worst deviation performance minimises the maximum distance to optimal performance.

The outline of this chapter is as follows. In sections 6.1 and 6.2 a scheduling system based on the coevolutionary minimax algorithm developed in section 2.4 is presented. The algorithm is compared to two other approaches. The worst case scheduling approach is found to improve the worst case performance but at the same time to degrade the ordinary preschedule performance. Section 6.3 considers an extension of the algorithm of section 6.1 in which worst case performance and ordinary preschedule performance are simultaneously optimised in a multiobjective approach. Section 6.4 describes how an evolutionary algorithm can be used to simultaneously solve many closely related problems in one program run. This is necessary for the worst deviation performance algorithm presented in section 6.5.

6.1 Worst case performance

Minimising the worst case cost of a schedule facing a number of scenarios can be formulated as a minimax problem. The problem is to minimise

$$\varphi(s) = \max_{b \in B} C(s, b) \quad \text{subject to } s \in S, \tag{6.1}$$

where S is the set of feasible schedules and B is the set of likely future scenarios. $C(s, b)$ is the cost of scenario b happening when implementing schedule s. In the following, B will be a set of possible future machine breakdowns, at most one breakdown happening in every scenario. More generally each scenario could also be a sequence of breakdowns, new jobs arriving etc.

Since equation (6.1) matches the problem formulation of section 2.4, the considerations from that section apply here. A schedule minimising (6.1) can be found using a standard optimisation algorithm if $\max_{b \in B} C(s, b)$ is evaluated for every schedule considered. If the set of scenarios B is large, this can be expected to be quite time-consuming because of the large number of evaluations. A coevolutionary approach to limit the number of evaluations by limiting the number of scenarios considered seems like a good idea.

It can be shown (see appendix D), that the job shop scheduling problem facing breakdowns does not satisfy the symmetric property of equation (2.6). This means that the coevolutionary minimax algorithms of [7] and [58] should not be expected to work, while the new minimax algorithm of section 2.4 could work.

6.1.1 Rescheduling and breakdown sets

A very important decision in scheduling systems like these is how to do rescheduling. When a real world scheduling system encounters a breakdown, it makes sense to run the entire scheduling algorithm again, the way it was done in chapter 5 with reduced and complete rescheduling. However, this is not possible when the preschedule has to be found, since rescheduling has to be performed a huge number of times during a single run of the algorithm. Rescheduling must be fast. One choice is to use right-shifting, but this is not a good choice when worst case performance is considered; whenever a breakdown strikes at a critical operation the makespan of the schedule will always be increased by the breakdown duration. A possible solution to these problems is to use a local search technique for rescheduling: it is reasonably fast, and it can be able to improve on schedules that are broken in critical places. In all experiments in this chapter the rescheduling is done by the hillclimber used for decoding in Mattfeld's GA3, see section 3.2.2. Other reasonable choices for rescheduling in a system like this could be running a tabu search for a very limited number of iterations, or fixing the processing sequences for all machines but one and calculating an optimal processing sequence for this one machine, as proposed in [71].

When working with worst case performance and job shop problems, there are certain breakdowns that cannot be countered in a clever way. For this reason, limited *breakdown sets* will be used in the experiments. These breakdowns will be designed to hold only breakdowns that can be countered by clever scheduling, while breakdowns which are impossible to hedge against will be left out.

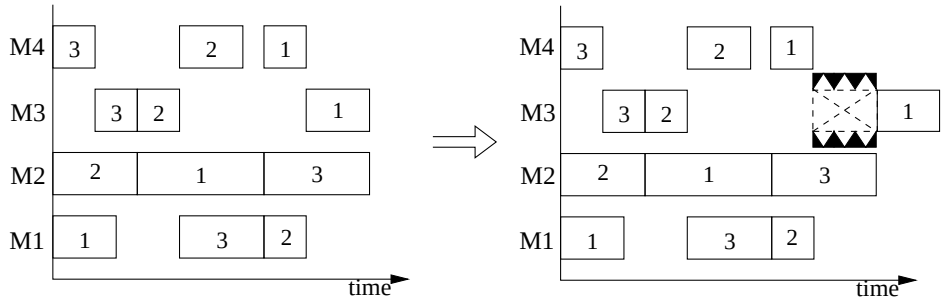

Figure 6.1: *A breakdown very late in the schedule can be countered optimally by right-shifting.*

If a breakdown strikes an operation which is very late in the schedule no rescheduling will be able to improve the schedule. To realize this, consider the schedule in figure 6.1. To the left is a schedule for a 3×4 problem. To the right, the schedule has met with a breakdown (indicated by the dashed block and the triangles). Since the breakdown occurs very late, most of the schedule has already been executed, and there are no choices for the rescheduler. No matter what kind of rescheduling method is employed, the result can never be better than the right-shifted schedule (as shown on the figure), and since the operation hit by the breakdown is critical the makespan is increased by the breakdown duration. Considering this kind of breakdown when working with worst case performance will lead to minimal makespan schedules being optimal, meaning that the search will find the same schedules as an ordinary static scheduling algorithm. For this reason breakdowns happening late in the schedules will not be included in the breakdown sets defined later. Not considering breakdowns happening very late in the schedules may also make sense in real scheduling environments. In systems with new jobs appearing over time handled on a rolling time horizon basis, it can be reasonable only to consider breakdowns falling within the part of the schedule expected to be implemented before the next batch of jobs are included in the schedule.

There is another kind of breakdown which also needs to be excluded from the breakdown sets. Consider figure 6.2. The schedule is the same as in figure 6.1, but this time the breakdown happens almost at the beginning of processing. This breakdown is handled optimally by right-shifting too, since the part of the schedule hit by the breakdown is critical, and will be critical in any acceptable schedule. Breakdowns of this kind will be disregarded in the breakdown sets, since again they will lead to minimal makespan schedule being optimal. Excluding certain machines from consideration with regard to breakdowns may also in some cases make sense in real scheduling systems, since some machines may be known to often break down while others almost never do.

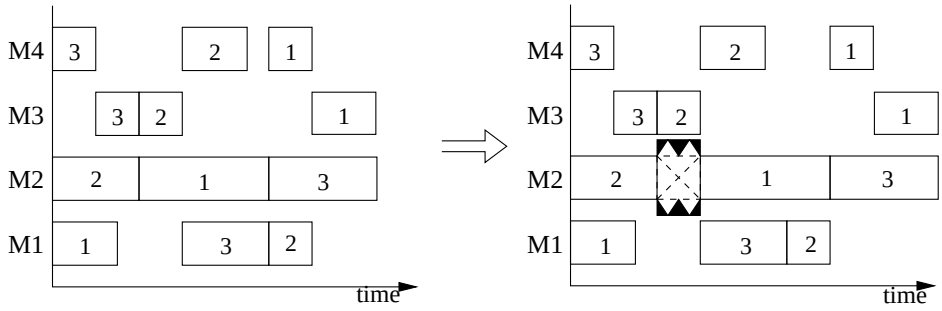

Figure 6.2: *A breakdown striking at a place which is critical in all reasonable schedules can be countered optimally by right-shifting.*

In theory a breakdown set for a scheduling problem can hold any subset of the breakdowns possible for the problem. The breakdown sets used in the experiments were all generated in an ad hoc way, and are all representable by a subset B_M of the machines in the problem and a time interval $B_T = \{t_{min} \ldots t_{max}\}$ stretching from the beginning of processing ($t_{min} = 0$) until some specified time t_{max}. The breakdowns in the set are allowed to take place on the machines in the machine subset and start within the time interval. They all have the same duration τ_B.

6.1.2 The scheduling algorithms

Three scheduling algorithms were used in the experiments. All of the systems use a variant of the same genetic algorithm.

The first GA simply minimises the preschedule cost (makespan). This GA is referred to as the *preschedule performance GA*. It is used mostly to verify that the worst case performance after rescheduling is improved when using the other two algorithms. The second GA optimises the after rescheduling performance of the schedules. The fitness evaluation is done in an exact way, making sure the worst case breakdown is tested by trying a large number of breakdowns. This algorithm, termed the *exact evaluation GA*, is quite slow. The third GA also optimises after breakdown and rescheduling performance, but this is done by letting the schedule population coevolve with a population of breakdowns. In this way time can be saved compared to the exact evaluation GA, at the expense of having some degree of noise in the fitness evaluation. The breakdown population size μ and number of progeny λ are important parameters in these algorithms, so they are termed *coevolutionary* $(\mu + \lambda)$ *algorithms*.

The following details hold for the scheduling part of all the genetic algorithms.

- Evolution occurs as in a standard generational GA with complete replacement in every generation, except for an elite of one.

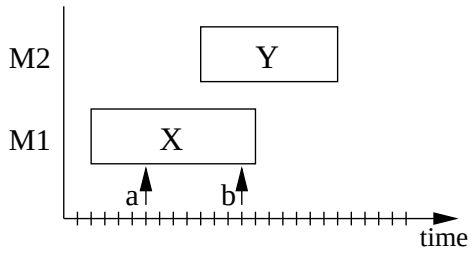

Figure 6.3: *Rounding up a breakdown time.*

- The schedule representation and decoding used is the permutation with repetition representation and the hillclimbing decoder of section 3.2.2.
- All new schedule individuals were created using crossover. The crossover used is the Generalised Order Crossover (GOX). Each new individual was mutated with a probability of 0.1. The mutation operator is Position Based Crossover (PBM).
- Tournament selection with a tournament size of two is used.
- A population size of 100 schedules is used, and the algorithms run for 100 generations.

In the preschedule performance GA, the objective function is the makespan of the preschedule, $C_{max}(s)$. In the exact evaluation GA it is $\max_{b \in B} C_{max}(s, b)$. In the coevolutionary algorithm a schedule population P_S and a breakdown population P_B coevolve. For the schedules, the objective function is $\max_{b \in P_B} C_{max}(s, b)$. For the breakdowns, the fitness evaluation is the asymmetric fitness evaluation of section 2.4.2.

The exact evaluation algorithm

Due to the nature of job shop schedules, it is not necessary to consider all breakdowns in B in order to calculate the exact worst case performance of a schedule. Consider the breakdown during processing of the operation marked "X" in figure 6.3. Any solution to the rescheduling problem for the breakdown time marked by "b" is bound to also by a solution to the breakdown with the breakdown time marked "a", since the breakdown marked "b" constrains new schedules more than the breakdown marked "a", while the processing of operation "X" will finish at the same time for both breakdown times (the preschedule finishing time plus the duration, since non-preemptive breakdowns are used). On the other hand, there exists solutions to the rescheduling problem of "a" that are not solutions to "b" (since the operation "Y" can be rescheduled for "a", but not for "b"). For these

reasons the best possible solution to the rescheduling problem "b" will never be better than the best possible solution to the rescheduling problem "a". Generally, a breakdown can never be worsened (meaning that $C_{max}(s, b)$ can never be decreased) by rounding up the breakdown time to the time just prior to the finishing time of an operation being processed at the breakdown time.

Therefore, when evaluating the worst case performance for a given schedule in the exact evaluation GA, only breakdown times that are immediately prior to the end of processing of an operation need to be considered, along with breakdowns at time t_{max}.

The coevolutionary algorithm

In the coevolutionary algorithm, the breakdown population evolves in a $(\mu + \lambda)$-evolutionary strategy. The population size is μ, and in each generation λ new individuals are generated. The new individuals compete with their parents and in every generation the λ worst individuals are discarded.

Each breakdown is represented by a breakdown time $t \in B_T$ and the machine breaking down $w \in B_M$. Remember that the downtime τ_B is fixed by the breakdown set. No crossover is used on the breakdowns; they only breed by mutation. For the breakdowns, selection for breeding is done on a linear ranking basis.

When a breakdown is mutated, in 50% of the cases only the breakdown time is changed. The breakdown time is perturbed by adding a Gaussian distributed value with zero mean and standard deviation $\frac{1}{4}(t_{max} - t_{min})$. In 25% of the cases only the machine is changed and set to a random machine in B_M. In the last 25% of the cases, the individual is a completely random individual drawn uniformly from B.

In the coevolutionary GA, when evaluating $C_{max}(s, b)$ for a given schedule $s \in P_S$ and breakdown $b \in P_B$, the breakdown time of b is rounded up to the time just before the end of processing of the current operation, or to t_{max}. Because of the additional constraints introduced by rounding, rounding makes the fitness evaluation focus on the hardest breakdowns, which means that the reliability of the worst case performance estimate $\max_{b \in B} C_{max}(s, b)$ is increased. The rounded breakdown time is not transferred to the breakdown gene; it is only used in the evaluation of b against the schedule s.

6.2 Experiments

For the experiments, the ten problems `la01`, `la02`, `la06`, `la07`, `la26`, `la27`, `la31`, `la36`, `ft10` and `ft20` were selected.

Problem	Opt. presch. makespan	Breakdown times B_T	Machines B_M	Best w.c. makespan
la01	666	0-299	1,2,3,4	691
la02	655	0-299	1,2,3,5	726
la06	926	0-299	2,3,4,5	926
la07	890	0-299	2,3,4,5	890
la26	1218	0-599	2,3,4,6,7,8,9	1261
la27	1235	0-599	1,2,3,5,6,8,9,10	1305
la31	1784	0-599	2,3,4,5,6,8,9	1784
la36	1268	0-599	1,2,3,4,6,7,8,9,10	1318
ft10	930	0-299	3,4,5,6,7,8,9,10	994
ft20	1165	0-599	1,2	1227

Table 6.1: *The breakdown sets used in the experiments. The last column lists the lowest worst case makespan found in the runs of section 6.2.1.*

For each scheduling problem a breakdown set was created. This was done by inspecting a number of near-optimal schedules and selecting machines and times for each problem in such a way that late breakdowns and parts of the schedules that would always be (near) critical were not included in the breakdown sets. The breakdown duration of $\tau_B = 80$ was used in all the experiments, as it was felt that the breakdown duration should be comparable to the operation processing times, which are in the range $\{1, \ldots, 100\}$. The details of the breakdown sets can be seen in table 6.1.

For each scheduling problem, six different sets of runs were performed: Four runs with the coevolutionary $(\mu + \lambda)$-algorithm with $(\mu + \lambda)$ taking the values $(4 + 2), (8 + 4), (12 + 6)$ and $(16 + 8)$ to determine the effect of the breakdown population size, one run of the preschedule performance algorithm to determine the worst case breakdown performance improvement of the coevolutionary algorithms, and one run of the exact evaluation GA to determine the effect of the noise present in the fitness evaluation of the coevolutionary GA.

6.2.1 Results relevant to scheduling

The average worst case makespan after a breakdown and rescheduling can be seen in table 6.2. The averages have been calculated over 400 runs. The averages in table 6.2 and the rest of the tables in section 6.2 differ slightly from the numbers previously published in [64]. This is because after the publication of [64], a programming error in the implementation of the algorithms was found, and the experiments had to be redone.

Problem	(4+2)	(8+4)	(12+6)	(16+8)	Presch. perf.	Exact eval.
la01	707.9	701.1	699.4	698.6	731.2	696.9
la02	737.6	736.2	735.1	734.6	737.7	734.4
la06	927.4	926.5	926.2	926.2	949.7	926.0
la07	896.8	893.2	892.2	892.1	946.9	891.9
la26	1287.9	1281.8	1279.3	1278.9	1293.4	1278.2
la27	1343.5	1335.5	1332.5	1331.9	1344.0	1330.2
la31	1800.6	1794.0	1790.8	1788.8	1830.0	1784.0
la36	1357.9	1346.9	1343.2	1342.2	1371.3	1340.8
ft10	1022.5	1020.8	1020.8	1018.6	1037.6	1018.9
ft20	1267.8	1264.9	1267.0	1265.3	1271.2	1264.5
Average	1134.6	1130.1	1128.7	1127.7	1151.3	1126.6

Table 6.2: *Average worst case performances.*

In all of the experiments, there is an improvement in the worst case performance when using the coevolutionary algorithm instead of the preschedule performance algorithm. In some cases the improvement is substantial (problems la01, la07, la31 and la36), while in other cases it is modest (la02). Generally, the coevolutionary algorithm performs better for high values of $(\mu + \lambda)$.

Considering the makespan performance of the preschedules without breakdown and rescheduling (table 6.3), it is evident that for some of the problems the increased flexibility observed in table 6.2 comes at a cost in preschedule performance. For la27, la36 and ft10, the preschedule makespan is increased by 10 or more by using the $(16 + 8)$ coevolutionary GA instead of the preschedule performance GA. In other cases, la06, la07 and la31 there is no increase in preschedule makespan at all.

The variation in after breakdown performance and preschedule performance from problem to problem indicates that for some problems and breakdown sets optimising worst case performance after breakdowns using a coevolutionary GA seems to perform very well. Consider la07 and la31, where a substantial improvement in worst case performance comes at no cost in preschedule performance. For other problems the performance is quite poor. For la02 and la27 a small or modest performance increase after rescheduling comes at a high price in preschedule performance. These observations indicate that if a scheduling system like the coevolutionary GA is to be used in the real world, great care will have to be taken. A way of circumventing this problem is to create a multi-objective version of the algorithm that optimises preschedule performance as well as worst case performance. This kind of system is considered in section 6.3.

Problem	(4+2)	(8+4)	(12+6)	(16+8)	Presch. perf.	Exact eval.
la01	666.5	666.6	666.9	666.5	666.0	667.0
la02	663.7	663.5	662.7	662.3	658.0	662.8
la06	926.1	926.0	926.0	926.0	926.0	926.0
la07	890.1	890.0	890.0	890.0	890.0	890.0
la26	1224.7	1223.7	1222.4	1223.0	1218.4	1222.7
la27	1284.9	1282.8	1281.2	1281.0	1267.5	1281.5
la31	1784.1	1784.0	1784.0	1784.0	1784.0	1784.0
la36	1315.7	1315.1	1314.8	1314.7	1297.5	1314.8
ft10	984.4	986.4	987.0	985.8	959.7	986.4
ft20	1199.4	1198.2	1199.4	1197.4	1192.2	1197.9
Average	1093.9	1093.7	1093.4	1093.1	1085.9	1093.3

Table 6.3: *Average preschedule performance without breakdown.*

6.2.2 Results relevant to evolutionary computation

From table 6.2 it is evident that the noise present in the fitness evaluation of the coevolutionary GA can have a negative effect on performance. For the small values of $(\mu + \lambda)$, in most cases the performance is a bit worse than the performance of the exact evaluation GA. For the higher values of $(\mu + \lambda)$, the fitness evaluation noise is smaller due to better sampling of the breakdown search-space, and the performance seems to be almost as good as that of the exact evaluation GA.

The effect of the population size μ on the noise in the fitness evaluation of the final individual has been investigated for the la07 problem in the top plot of figure 6.4. In the plot it is evident, that there is a significant drop in noise when increasing μ from 4 to 8, while smaller drops arise when increasing μ to 12 and 16. The effect of the population size on worst case performance has been plotted in the middle plot of figure 6.4. In the plot it is evident that as the breakdown population size increases, the worst case cost drops. In the bottom part of the figure the CPU-time spent has been plotted as a function of μ. The CPU-time seems to increase linearly with μ. The computation time is really $O((\mu + \lambda)\log(\mu + \lambda))$ because of the asymmetric fitness evaluation, but since the time spent in the hillclimber is far greater than the other contributions to processing time, it increases linearly for reasonable values of $(\mu + \lambda)$. Similar plots were made for the other problems as well, and they were all qualitatively equivalent to the ones of figure 6.4.

The average processing times for one run of each algorithm can be seen in table 6.4. The experiments were run on a 250MHz SGI O2 computer. It is evident that even for small values of $(\mu + \lambda)$, the coevolutionary GAs are much slower than the preschedule performance GA. This is due to the processing time spent

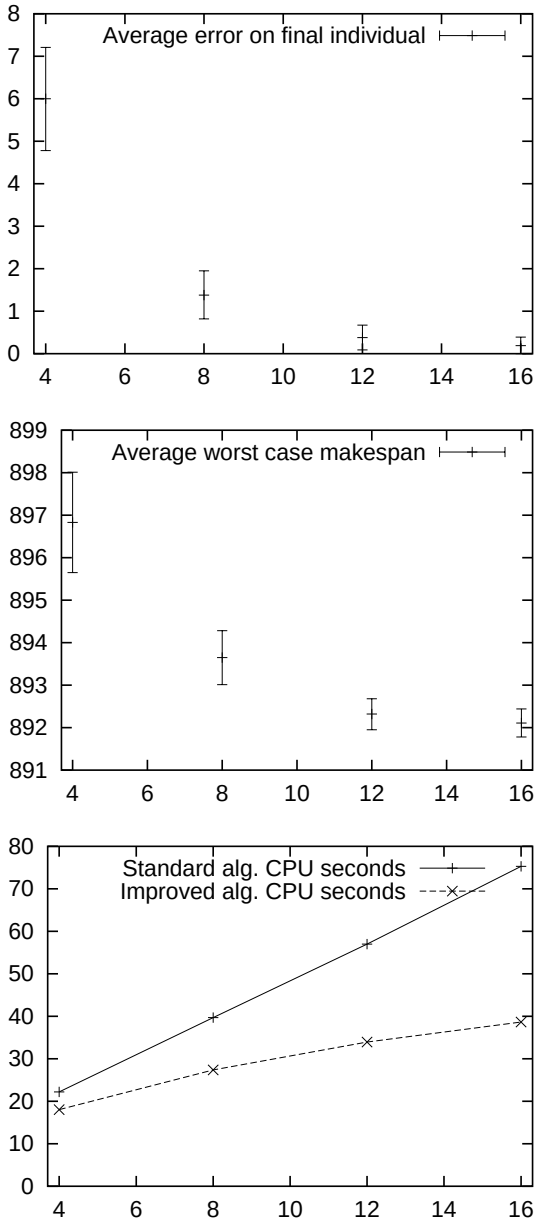

Figure 6.4: *Plots for the coevolutionary runs on the* `la07` *problem.* ***Top:*** *The average error on the fitness evaluation of the final individual for various values of* μ *(*$\lambda = \frac{1}{2}\mu$*). The error bars indicate 95% confidence intervals on the average.* ***Middle:*** *Average worst case makespan.* ***Bottom:*** *Average CPU-time used in seconds. The "standard algorithm" is the algorithm of section 6.1.2, while the "improved algorithm" is the algorithm of section 6.2.3.*

Problem	(4+2)	(8+4)	(12+6)	(16+8)	Presch perf.	Exact eval.
la01	13.9	27.6	40.5	54.1	1.5	47.7
la02	13.9	25.5	37.3	48.4	1.3	49.2
la06	22.1	38.1	55.6	73.2	3.1	68.7
la07	22.2	39.7	57.0	75.3	3.2	75.1
la26	72.0	128.2	185.3	244.3	13.6	686.2
la27	86.1	141.0	199.7	262.0	13.9	755.0
la31	159.0	278.9	395.5	521.2	30.9	1427.9
la36	74.6	136.1	194.1	258.2	10.9	627.4
ft10	31.9	63.6	89.9	123.8	3.5	70.1
ft20	42.5	70.3	100.6	128.7	5.6	139.7
Average	53.8	94.9	135.6	178.9	8.8	394.7

Table 6.4: *Average processing time in CPU-seconds.*

doing rescheduling and evolving the breakdown population.

Comparing the processing times of the exact evaluation GA and the coevolutionary GA, it seems that for the smaller problems (la01, la02, la06, la07, ft10 and ft20) only a modest amount of processing time is saved, and only if a small breakdown population size μ is used. In some cases with breakdown population sizes $\mu = 16$ the coevolutionary GA is slower than the exact evaluation GA. For the small values of μ some processing time is saved (typically 50%-70% for $\mu = 4$ and 7%-50% for $\mu = 8$). Given the slightly superior quality of the solutions found by the exact evaluation GA it seems that for small problems the exact evaluation GA should be preferred unless time is very critical.

For the larger problems la26, la27, la31 and la36, more time can be saved. For the smallest breakdown population size $\mu = 4$ around 90% of the processing time is saved. For the largest breakdown population size $\mu = 16$, approximately 65% of the processing time is saved. These are substantial savings, since the processing time for a run of the exact evaluation GA is more than 10 CPU-minutes for all of these problems. For larger problems more time is expected to be saved. Which population size to use is a tradeoff, since the quality of schedules improves with larger μ.

6.2.3 Further investigations

Comparing coevolution to a random sample GA

In order to further investigate the benefit of using a coevolutionary approach to estimate schedule worst case performance, the performance of the coevolutionary

algorithm was compared to a genetic algorithm in which the worst case performance of every schedule is estimated by testing it on a set of randomly sampled breakdowns (called *the random sample GA*). In this algorithm the objective function of the schedule is set to $\max_{b \in B_R} C_{max}(s, b)$, where B_R is a set of random breakdowns, sampled uniformly and independently for every schedule. Since the time used to evolve the breakdown population in the coevolutionary GA is negligible compared to the time spent in the makespan hillclimber, the two algorithms have the same running times for the same number of schedule $\times$ scenario evaluations. The average worst case makespans of this algorithm can be compared to the performance of the coevolutionary GA in table 6.5. The averages have been taken over 400 runs. There is a subtable for each problem, and each of the algorithms corresponds to a row in the table. Each column refers to a $(\mu + \lambda)$-setting for the coevolutionary GA. The random sample GA used $\mu + \lambda$ scenario-evaluations for each schedule.

From the table, it is evident that the coevolutionary approach outperforms the random sample approach on all the problems when the same number of schedule $\times$ scenario evaluations are used. In some cases the coevolutionary approach outperforms the random sample approach by far. For the problems `la02`, `la06`, `ft10` and `ft20`, the coevolutionary approach with the lowest number of evaluations $(4 + 2)$ outperforms the random sample approach with the highest number of evaluations $(16 + 8)$, meaning that better solutions are found while saving 75% of the CPU-time. In other cases the random sample approach produces almost the same level of performance (average difference less than 1, `la01`, $(\mu + \lambda) = (4 + 2)$). The difference in performance between the two approaches probably depends on the difficulty of the breakdown search-space. If only few breakdowns produce the worst possible performance, or if several breakdowns are needed at the same time to guarantee the worst case, the coevolutionary approach should be expected to have an edge when compared to the random sample approach.

Improving runtime

The algorithm used for the experiments of section 6.2 performed rescheduling every time a schedule encountered a breakdown. This is not always necessary. Because of the rounding up of breakdown times (see section 6.1.2), it will sometimes happen that two or more breakdowns in the breakdown population decode to the same machine and breakdown time for a given schedule. If the results of previous evaluations are kept for each schedule and reused if the schedule meets the same breakdown again, computational effort can be saved. A variant of the algorithm using this trick was implemented. The rescheduling results are stored in a datastructure associated with each individual in the schedule population. Since

	Algorithm	(4+2)	(8+4)	(12+6)	(16+8)
`la01`:	$(\mu+\lambda)$-coev. GA	707.9	701.1	699.4	698.6
	random sample GA	708.6	704.2	704.0	702.8

	Algorithm	(4+2)	(8+4)	(12+6)	(16+8)
`la02`:	$(\mu+\lambda)$-coev. GA	737.6	736.2	735.1	734.6
	random sample GA	743.2	740.6	738.7	737.9

	Algorithm	(4+2)	(8+4)	(12+6)	(16+8)
`la06`:	$(\mu+\lambda)$-coev. GA	927.4	926.5	926.2	926.2
	random sample GA	932.0	930.2	929.3	927.6

	Algorithm	(4+2)	(8+4)	(12+6)	(16+8)
`la07`:	$(\mu+\lambda)$-coev. GA	896.8	893.2	892.2	892.1
	random sample GA	907.3	899.8	896.1	895.7

	Algorithm	(4+2)	(8+4)	(12+6)	(16+8)
`la26`:	$(\mu+\lambda)$-coev. GA	1287.9	1281.8	1279.3	1278.9
	random sample GA	1298.1	1292.6	1288.5	1286.4

	Algorithm	(4+2)	(8+4)	(12+6)	(16+8)
`la27`:	$(\mu+\lambda)$-coev. GA	1343.5	1335.5	1332.5	1331.9
	random sample GA	1352.2	1347.8	1343.2	1342.0

	Algorithm	(4+2)	(8+4)	(12+6)	(16+8)
`la31`:	$(\mu+\lambda)$-coev. GA	1800.6	1794.0	1790.8	1788.8
	random sample GA	1806.1	1798.2	1796.1	1793.1

	Algorithm	(4+2)	(8+4)	(12+6)	(16+8)
`la36`:	$(\mu+\lambda)$-coev. GA	1357.9	1346.9	1343.2	1342.2
	random sample GA	1368.5	1363.1	1359.2	1356.6

	Algorithm	(4+2)	(8+4)	(12+6)	(16+8)
`ft10`:	$(\mu+\lambda)$-coev. GA	1022.5	1020.8	1020.8	1018.6
	random sample GA	1045.6	1036.9	1032.4	1028.7

	Algorithm	(4+2)	(8+4)	(12+6)	(16+8)
`ft20`:	$(\mu+\lambda)$-coev. GA	1267.8	1264.9	1267.0	1265.3
	random sample GA	1280.7	1278.5	1276.0	1277.1

Table 6.5: *Average worst case performances of the coevolutionary GA and the random sample GA using the same amount of schedule $\times$ scenario evaluations.*

Problem	(4+2)	(8+4)	(12+6)	(16+8)	Exact eval.
la01	11.3	16.9	20.6	25.8	47.7
la02	11.2	16.5	20.7	24.3	49.2
la06	16.8	25.3	30.1	33.3	68.7
la07	18.0	27.4	33.9	38.6	75.1
la26	65.9	104.9	140.5	171.3	686.2
la27	71.9	113.3	148.7	184.5	755.0
la31	140.9	235.7	313.0	389.4	1427.9
la36	66.0	104.5	135.9	166.3	627.4
ft10	23.8	33.5	40.4	42.7	70.1
ft20	31.0	46.1	58.1	67.1	139.7
Average	45.7	72.4	94.2	114.3	394.7

Table 6.6: *Average processing time in CPU-seconds for the improved algorithm. The processing times of the exact evaluation algorithm have also been included.*

the entire schedule population (except for an elite of one) are replaced in every generation, these datastructures are cleared in every generation. The average running times of the improved algorithm for different problems and $(\mu + \lambda)$-settings can be seen in table 6.6.

Comparing the running times of table 6.6 to the running times of the standard algorithm (table 6.4), it is evident that in every single experiment, the average running time is lower than the standard algorithm. For small $(\mu + \lambda)$ values, only a little processing time is saved, but for large $(\mu + \lambda)$-values the savings are substantial, especially for small and medium sized problems. For problem ft10 and the $(16 + 8)$-algorithm, the running time is reduced to one third of that of the standard algorithm. For the $(16 + 8)$-algorithm, the average relative reduction over all the problems is 47%.

Because of the decreased computational demand of the improved algorithm, it was used as the starting point for the algorithms in the next subsection and in section 6.3.

Time needed to reach a certain level of performance

The performance improvement of the coevolutionary approach over the exact evaluation GA is in its running time. Another way of comparing the running times of the algorithms is by measuring how much time is needed to reach a specific level of performance. To this end, new formulations of the exact evaluation GA and the coevolutionary GA were created.

The exact evaluation GA was changed to accept as a parameter a desired level

Problem	$(4+2)$-coev.	$(8+4)$-coev.	Exact eval.
la01	39	28	26
la02	9	8	2
la06	100	100	100
la07	100	100	100
la26	13	7	0
la27	15	10	0
la31	100	100	99
la36	59	44	2
ft10	6	6	1
ft20	8	4	2

Table 6.7: *Percentage of successful runs.*

of worst case performance and stop running when a schedule meeting the requirement was found. The coevolutionary GA was changed in the same way, except that since worst case performance is not evaluated in an exact way in the algorithm, the performance of schedules estimated to meet the requirement was evaluated in an exact way, and the algorithm was allowed to keep running if the schedule did not meet the requirement.

Both algorithms are prone to get stuck in local minima, so they were programmed to reinitialise the schedule population at random if there was no improvement in the best individuals' worst case performance for 100 generations. The best observed worst case performances of table 6.1 were used as the required performance levels. Since for some of the problems it turned out to be really difficult to find the best worst case performance, each algorithm run was terminated after 10 CPU-minutes. Some of the runs ran for a few seconds more than this, since the time-consumption of the algorithms was checked between generations, meaning that it was possible for the algorithms to exceed the 10 CPU-minutes by one generation. Runs that succeeded to meet the requirement within this allowed processing time will be called *successful* in the following, while runs failing to do so will be called *unsuccessful*.

The experiments involved the exact evaluation algorithm and the coevolutionary algorithm with the parameter settings $(\mu + \lambda) = (4 + 2)$ and $(8 + 4)$. Each algorithm was run 100 times on 10 problems, and for each algorithm and problem combination the success-rate (percentage of successful runs) and the average number of CPU-seconds used were recorded. The results are displayed in tables 6.7 and 6.8.

Inspecting the success-rates of table 6.7, it becomes clear that there is a big variation in difficulty for these problems. The problems `la06`, `la07` and `la31`

Problem	$(4+2)$-coev.	$(8+4)$-coev.	Exact eval.
la01	473.1	509.0	512.3
la02	572.5	571.7	594.3
la06	0.6	0.8	1.9
la07	20.4	32.1	79.3
la26	558.5	582.2	603.7
la27	544.6	576.3	603.3
la31	52.2	81.4	445.4
la36	402.8	462.6	600.2
ft10	584.8	590.0	594.8
ft20	581.1	588.9	591.3

Table 6.8: *Average number of CPU-seconds in the experiments.*

are relatively easy; for these problems the runs are always successful. Some of the other problems must be quite hard; for the problems la02, ft10 and ft20 the success-rates are close to zero for all three algorithms. It is quite surprising that la02 turns out to be so difficult. The problem is very little (50 operations), and finding the optimal preschedule makespan is easy.

Comparing the success-rates of the three algorithms, it becomes clear that the coevolutionary approach is superior to the exact evaluation approach. On the problems not always solved successfully, the (4+2)-coevolutionary approach always (except for ft10 where the success-rates are equal) outperforms the (8+4)-coevolutionary approach, which again outperforms the exact approach. For some of the problems (la26, la27, la36, ft10 and la20) the difference is quite high; consider the la36 problem which is solved successfully in 59% of the runs of the (4+2)-coevolutionary algorithm, and only 2% for the exact evaluation approach.

Inspecting table 6.8 of average numbers of CPU-seconds, it is difficult to compare the average running times for the hard problems, since the processing times reflect a mixture of successful and unsuccessful runs. For the easy problems (la06, la07 and la31) it is clear that there is a significant performance improvement when using the coevolutionary approach. For la06 the running times are very small, but the (4+2)-coevolutionary approach saves approximately two thirds of the running when comparing to the exact evaluation algorithm. For la07 and la31 both the running times and the relative improvement are larger. For la31, the (4+2)-coevolutionary approach saves 88% of the average running time.

In conclusion, these experiments indicate that the coevolutionary approach is never inferior to the exact evaluation approach, while in some cases it is clearly su-

perior to it. For the difficult problems, the coevolutionary approach has a markedly higher success-rate than the exact evaluation approach, while for easier problems the average time needed to reach a specific level of performance is substantially smaller.

6.3 Multi-Objective algorithm

In section 6.2.1 it was found that for some problems the improved worst case makespan produced by the coevolutionary system comes at a cost of increased preschedule makespan. This increase in preschedule makespan indicates, that for some of the problems there may be a tradeoff between the worst case makespan and the preschedule makespan; minimising the worst case makespan and the preschedule makespan may be conflicting objectives. On the other hand, these objectives need not be conflicting. The increased preschedule makespan observed in section 6.2.1 could also be caused by a lack of selection pressure to minimise preschedule makespan.

If preschedule cost and worst case cost are conflicting objectives, in many real world scheduling systems it will not suffice to use an algorithm minimising only worst case performance. If the guaranteed worst case performance comes at a price of increased cost if no breakdown happens, the tradeoff between worst case performance and preschedule cost may need to be considered every time a new schedule is created. In these cases it will be beneficial if the scheduling system produces a number of non-dominated solutions, from which one solution can be selected by a human expert.

6.3.1 The algorithm

In this section, the coevolutionary scheduling system presented in section 6.1 and 6.2 will be combined with a multi-objective genetic algorithm to minimise preschedule makespan and worst case makespan at the same time. The multi-objective algorithm used is the Pareto-based NSGA-II introduced in section 2.2.1, developed by Deb et al. [35, 36]. Although the NSGA-II and the coevolutionary system from the last sections are both fairly advanced systems, combining them into one algorithm is not difficult. It can be done by simply letting the NSGA-II take over the evolution of schedules in the coevolutionary system, while keeping the part of the algorithm controlling the breakdowns unchanged. The overall structure of the combined system has been visualised in figure 6.5. At the top of the figure two boxes representing the schedule population and the breakdown population have been drawn. The dashed arrows on the figure represent fitness evaluations, while the solid arrows represent transfer of genetic material.

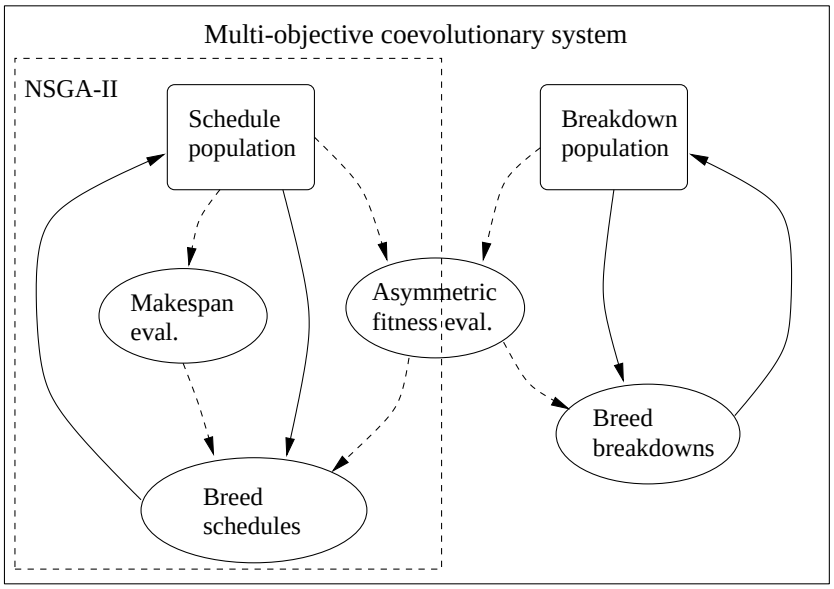

Figure 6.5: *Schematic drawing of the multi-objective coevolutionary scheduling system.*

Since the coevolutionary system and the NSGA-II are both generational, the combined algorithm works in a generational manner; first all individuals are assigned a fitness, then new individuals are bred, and these new individuals are then incorporated into the old populations. For the breakdown population, the breeding happens using the same evolutionary strategy described in section 6.1.2. For the schedule part, the breeding happens as in the NSGA-II; for a population size of N, N new individuals are bred in every generation, and of the $2N$ individuals the N best are kept. This replacement scheme effectively means that an elite size of N is used.

Because of the replacement scheme used in the schedule part of the algorithm, potentially a lot of individuals are kept from generation to generation. Since the most time-consuming part of the algorithm is the evaluation of the worst case makespan, it makes sense to remember the result of previous evaluations from generation to generation. However, since the breakdown population changes from generation to generation, it would be unfair to simply keep the worst case performance of old individuals and reuse it in the next generation. Doing this would mean that a "lucky" schedule-individual evaluated against a weak population of breakdowns could have an artificially high fitness for the rest of the program run. Instead, a different approach is taken. Every time a schedule is evaluated against a breakdown, a datastructure associated with the schedule records the result. If the schedule ever encounters the same breakdown, the result is found in the datastruc-

ture and there is no need to do the computation again. Because of the rounding up of breakdown times described in section 6.1.2 a substantial number of evaluations are saved in this way (preliminary experiments indicated that often more than 50% of the evaluations are saved). The worst case makespan used in fitness calculations is the worst performance ever produced by the schedule in the evaluation of the current generation, as well as previous ones. This means that good schedules that have stayed in the population for several generations have a better worst case performance estimate (and thus a more accurate fitness evaluation) than new individuals, since their worst case performance is based on evaluations over several generations rather than just the current one.

These ideas are related to the idea of *life time fitness evaluation* (LTFE) [91], in which good individuals are tested more often than bad individuals, and in which a history of past evaluations is kept. The main difference between LTFE as described in [91] and the approach used in the scheduling system is that in LTFE the evaluations are remembered on a limited time-scale; when the evaluations become too old they are thrown out of the memory. In the scheduling system the evaluations are remembered on an unlimited time-scale and never forgotten. In LTFE good solutions are tested more frequently than bad ones, which is not the case in the scheduling system. However, almost the same effect is probably achieved by storing previous evaluations on an infinite time-scale; old (meaning good) individuals have been tested against a larger number of breakdowns than new individuals.

Since it was felt that an accurate fitness estimation would be necessary for the results of the experiments to be trustworthy, the breakdown part of the algorithm was run using a $(\mu + \lambda) = (16 + 8)$ setting. The breakdown part of the algorithm was exactly the same as described in section 6.1.2. For the schedule part of the system, a population size of $N = 100$ was used. The schedules were bred as described in section 6.1.2, except a mutation rate of 0.2 was used after crossover, since in preliminary experiments it seemed to perform slightly better than the setting used in section 6.1.2.

6.3.2 Experiments

The experiments had two purposes: to investigate if the objectives of minimising preschedule makespan and worst case makespan are really conflicting, and to see how well the system outlined in the previous section works. Since there is no guarantee that the algorithm actually finds the globally optimal solutions, the first question cannot really be answered based on the experiments of this section; the experiments can only be taken as a hint to what the answer may be.

The multi-objective algorithm was run 400 times for each of the ten problems. For every problem, the average number of solutions returned by the algorithm

Problem	Non-dominated solutions
la01	(689,666)
la02	(726,655)
la06	(926,926)
la07	(890,890)
la26	(1260,1218)
la27	(1303,1270), (1308,1269), (1314,1256), (1335,1255)
la31	(1784,1784)
la36	(1318,1315), (1324,1297), (1335,1296), (1339,1291), (1341,1281), (1348,1278)
ft10	(999,960), (1004,930)
ft20	(1224,1178), (1253,1173)

Table 6.9: *The best known non-dominated solutions (worst case makespan, preschedule makespan).*

was calculated, and all the solutions returned were recorded and compared in an attempt to identify the true Pareto-optimal set.

For the problems la01, la02, la06, la07, la26 and la31 the algorithm almost always returned only one solution, and the best solution returned during all the runs dominated all other solutions. This indicates that for these problems minimising preschedule cost and worst case cost are probably not conflicting objectives. The best known non-dominated solutions found in all the multi-objective runs can be found in table 6.9. For the problems la06, la07 and la31, the Pareto optimal solution is known, since for these problems, the best observed worst case performance is equal to the minimum preschedule makespan.

The average smallest preschedule makespan and the average smallest worst case makespan of the returned solutions, the success-rate (percentage of the solutions returned identical to the solutions of table 6.9) and the average number of non-dominated solutions returned can be found in table 6.10.

For the problems la27, la36, ft10 and ft20, the program often returned more than one solution, and none of the solutions returned were able to dominate all the other solutions. Figure 6.6 shows plots of the preschedule makespan and worst case makespan of all the solutions returned during the program runs. The solutions not dominated by any other solutions found in the multi-objective runs have been plotted larger than the others. As the plots indicate, most of the solutions returned by the multi-objective system are inferior to the best solutions. This is not surprising, since the experiments of section 6.2.3 indicated that these problems are all hard.

For la27, la31, ft10 and ft20, the success-rates (in table 6.10) are very low. For these problems most of the best known solutions were found only once.

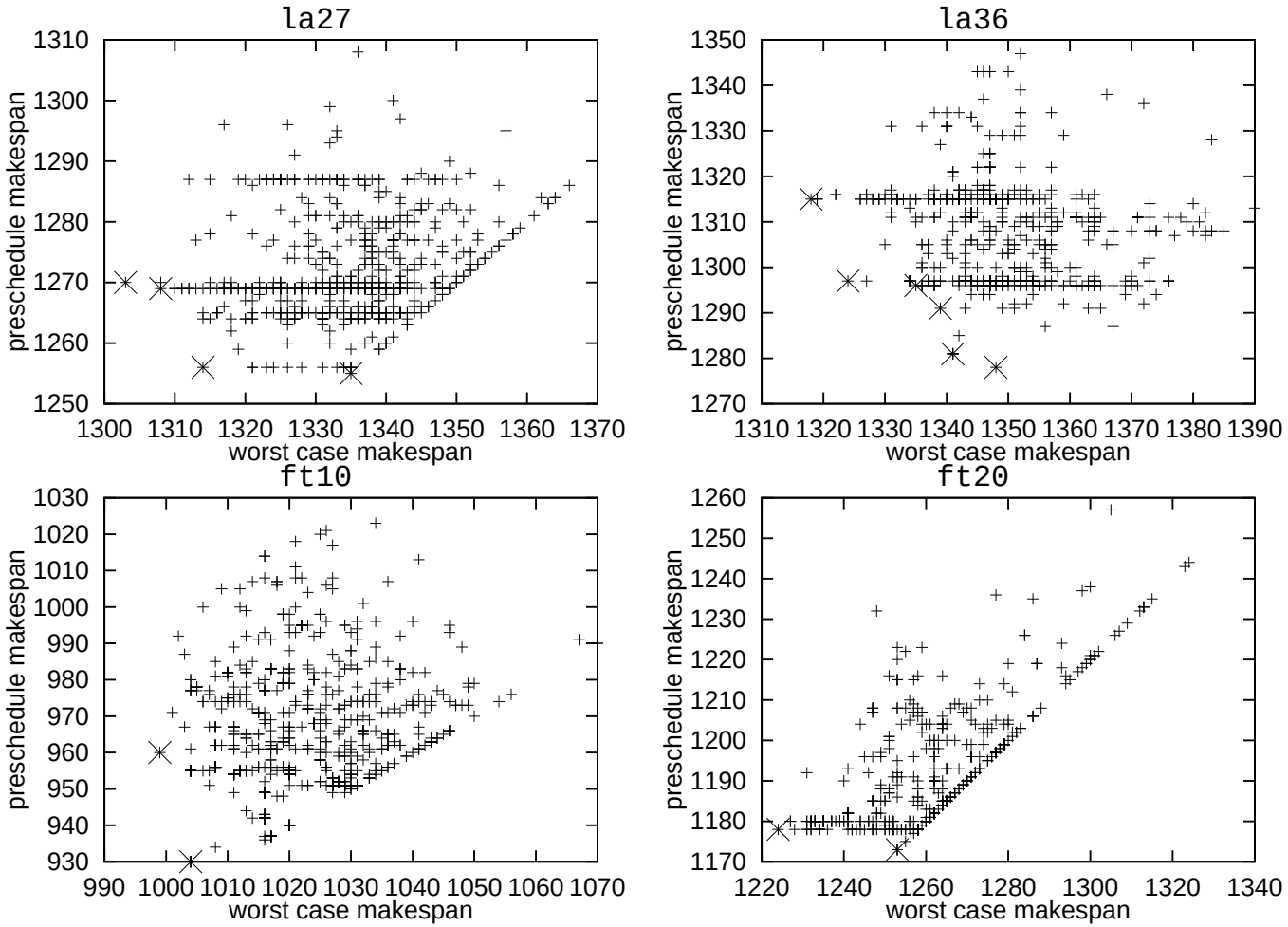

Figure 6.6: *The performances of the solutions returned by the multi-objective algorithm. The solutions not dominated by any other solutions have been plotted larger.*

Problem	Avg. C_{max}	Avg. wc. perf	Success-rate	#sol. avg
la01	666.0	696.3	0.5	1.08
la02	655.4	731.0	10.7	1.08
la06	926.0	926.0	100.0	1.00
la07	890.0	891.5	68.3	1.00
la26	1219.6	1282.1	0.3	1.32
la27	1268.9	1331.5	0.6	1.58
la31	1784.0	1784.1	100.0	1.00
la36	1298.5	1344.4	1.3	1.98
ft10	953.4	1015.6	0.9	2.33
ft20	1188.1	1259.7	0.7	1.39
Average	1085.0	1126.2		

Table 6.10: *Summarised results of the multi-objective experiments.*

This means that the non-dominated solutions listed in table 6.9 should not be trusted too much. There may be other solutions dominating the solutions found in the runs, and the Pareto sets listed are probably incomplete. For the problems la27, la36 and ft20, the minimal preschedule makespans were not found in any of the runs (the minimal makespans are listed in table 6.1). The same was the case for ft10 and the best observed worst case makespan.

For some of the problems, the results indicated that finding a schedule guaranteeing worst case performance can be significantly harder than finding a minimal makespan preschedule. In the ft10 experiment, the minimal makespan preschedule was found seven times, while the best known worst case performance (table 6.1) was not found once. For the la01 problem, the minimum preschedule makespan was found in every single run, while the best known worst case cost was found twice in 400 runs.

The average performance of the multi-objective algorithm is listed in columns two and three of table 6.10. The average performances can be compared to the averages for the preschedule performance GA and the coevolutionary (16+8) GA in tables 6.2 and 6.3. Comparing the numbers it seems that on average the multi-objective algorithm is just as good at minimising preschedule makespan as the preschedule performance GA, and just as good at minimising worst case cost as the coevolutionary (16+8) GA. The averages indicate that the multi-objective algorithms may actually be slightly better than both of the other algorithms. Comparing the multi-objective algorithm to the preschedule GA and the coevolutionary (16+8) GA is fair, since the algorithms breed the same number of schedules, and since the multi-objective algorithm uses roughly the same number of calls to the schedule hillclimber (the most time-consuming part of the algorithm) as the coevolutionary (16+8) GA.

When comparing the average performances of table 6.10 to the averages in tables 6.2 and 6.3, the average performance of the multi-objective system is sometimes substantially better than the performances of the best of the other algorithms, while the opposite is rarely the case. This may be an indication that for solutions not in the true Pareto-optimal set (most of the solutions returned for these problems are not optimal), minimising preschedule makespan and worst case cost are not conflicting objectives. On the contrary, working on two objectives may help the algorithm escape local optima, since it needs to get stuck in both objectives in order to really be trapped.

6.3.3 Discussion

The average performances achieved by the multi-objective algorithm compare favourably to the average performances of the algorithms presented previously in this chapter, so in this respect the algorithm performs well.

For the easy problems (`la01`, `la02`, `la06`, `la07`, `la26` and `la31`) no tradeoff between worst case cost and preschedule cost was identified.

For the hard problems (`la27`, `la36`, `ft10` and `ft20`), the number of best known non-dominated solutions indicate that for some problems, there may be a tradeoff between preschedule cost and worst case cost. For these problems the algorithm was unable to find solutions from the best known set of solutions in most of the runs. Usually, only around two solutions were returned by the algorithm, despite the best known non-dominated solution set for some of the problems being larger. Although the algorithm can be said to outperform the algorithms from the previous sections of this chapter, there is clearly room for improvement, both in terms of the quality of the solutions found and the ability of the algorithm to cover a wide part of the front of Pareto-optimal solutions.

6.4 Solving many problems at once

In order to work with the worst deviation performance of equation (4.3), knowledge of $C^*(b)$ is needed for all possible breakdowns $b \in B$. This requirement makes a very high computational demand, since B can be large, and since finding $C^*(b)$ involves finding the optimal preschedule for scenario b. For a fixed breakdown duration, working with one breakdown occurrence and integer breakdown times requires the generation of approximately $mC_{max}(P)$ schedules, where $C_{max}(P)$ is the minimal makespan for the problem and m is the number of machines. For the 10×10 sized problem `ft10`, this requires the solution of some 10000 job shop problems.

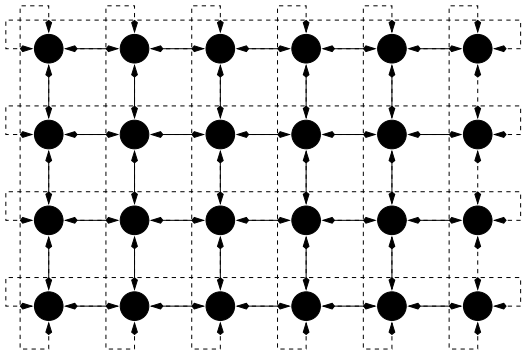

Figure 6.7: *A standard diffusion GA. The fitness evaluation is the same everywhere in the grid. Two positions are neighbours (and can mate, using the crossover operator) if they are connected with an arrow.*

Solving several thousand NP-complete problems of even a moderate size to guaranteed optimality is intractable, and will not be attempted here. Instead, two approaches for approximating $C^*(b)$ for all b will be given. The idea of section 6.4.1 is based on a diffusion GA, and has not been implemented or tested. In section 6.4.2 an idea of reusing populations in a GA from problem to problem is presented. This approach has been implemented and compared to a more standard approach.

In section 6.5, the algorithm presented in section 6.4.2 is used to create worst deviation performance schedules.

In the following sections, the notation $C^*(w,t)$ is sometimes used to denote $C^*(b)$, where b is a breakdown happening at machine w, and time t. In these cases, the breakdown duration τ_B is fixed, and does not need to be explicitly stated.

6.4.1 A diffusion genetic algorithm approach

It seems reasonable that the optimal preschedule for a breakdown happening at machine w at time t is closely related to the optimal preschedule for a breakdown happening at w at time $t+1$. If the solution to one of the problems is known, maybe it can be exploited in the search for the other solution. This can be used in a diffusion based GA, searching for the optimal preschedules for all possible breakdowns happening at a specific machine at the same time.

A standard diffusion model has a grid-structured population, in which mating and selection take place in small neighbourhoods of the grid (illustrated in figure 6.7). The idea behind this structuring of the population is to slow down the dissemination of genetic material, and allow different parts of the population to focus on different parts of the search-space. Usually, the grid is constructed in a toroidal

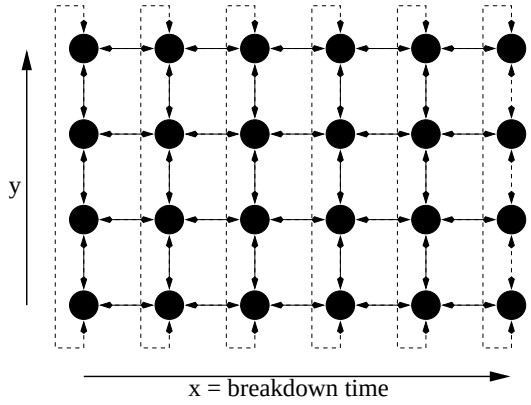

Figure 6.8: *The proposed diffusion GA in which the fitness evaluation depends on the x-coordinate in the grid. The grid connectivity no longer "wraps around" in the x direction.*

fashion in order to avoid having a number of individuals on the edge of the grid. In the standard diffusion models, the same fitness evaluation is used all over the grid.

Since the preschedules optimal for breakdown times very close to each other can be expected to be related to each other, it may make sense to vary the breakdown time used in fitness evaluation as a function of one of the grid coordinates, e.g. x. A straightforward way of doing this could be simply to set the breakdown time to x, which will be assumed in the rest of this section. The grid is illustrated in figure 6.8. The idea in using this kind of population is to have a large population in which different parts of the population focus on different breakdown times. The individuals all interact and compete with each other, and hopefully $C^*(b)$ can be found for all breakdowns happening on a specific machine in a single program run.

The fitness evaluation is independent of the y-coordinate, so it makes sense to "wrap around" the connectivity of the individuals in the y direction. It does not make sense to wrap around in the x direction, since this would mean that schedules supposed to be optimal for the lowest breakdown time would be connected to schedules supposed to be optimal for the highest breakdown time. Since it is probably a good idea to avoid individuals positioned on the edge as much as possible, it may make sense to add a few extra columns of individuals as "padding" at the ends of the grid. The fitness evaluation of this padding should use the minimal breakdown time at one end of the grid, and the maximal breakdown time at the other end. Doing this will not avoid having individuals at the edge of the grid, but it will avoid having all of the individuals of certain breakdown times at the edge.

The size needed for the grid depends on the problem being solved. Clearly

there is no need to make the grid longer than the minimal makespan of the problem, since a breakdown happening after schedule completion can never affect the schedule. In order to get a proper size of the grid, a good makespan schedule is needed before the diffusion GA can be started. Since the range of possible breakdown times is large, a grid of this kind will have to be much larger than the grids used in more standard applications of diffusion models. If a grid height of 10 is used, for a problem with minimum makespan 1000, the population will consist of 10000 individuals.

The selection scheme used in an algorithm like this can be a crucial choice. In diffusion models selection is done locally. In Mattfeld's GA3 explained in section 3.2.2, an individual I chooses its mate from the four individuals surrounding it (see figure 3.16). Assume the same is done in the proposed algorithm. In this algorithm, the individuals above and below individual I will have a fitness based on the same breakdown time as I, call this breakdown time x, while the individuals to the left and right will have a fitness based on $x - 1$ and $x + 1$. This means that individuals are not compared on an equal basis, since $C^*(w, x-1)$, $C^*(w, x)$ and $C^*(w, x + 1)$ (and thus the best possible objective function value) may have different values. This is potentially a serious problem. Consider what can happen if $C^*(w, x - 1)$ has a low value, while $C^*(w, x)$ and $C^*(w, x + 1)$ have high values. Assume that the individuals located at x-coordinates $x - 1$ and $x + 1$ have all converged to their optimal solutions, while the individuals at x-coordinate x are all suboptimal. When an individual with x-coordinate x chooses at mate from its neighbours, it may be favourable to pick one with x-coordinate x or $x + 1$. However, since the objective function value at $x - 1$ is better, the individuals with this x-coordinate will always be favoured in selection. If the optimal schedule for breakdown time $x - 1$ is very different from the optimal schedule for x, the individuals at coordinate x may experience a constant influx of unsuitable genetic material via crossovers with neighbours at $x - 1$.

This effect of distorted selection due to the slight differences in fitness evaluation may mean that individuals located at positions in the grid where $C^*(w, t)$ and the optimal schedule change may not converge to the right schedule. A way of solving this problem would be to reevaluate the objective function values of individuals during selection, but this could significantly increase the running times of the algorithms. Another way of solving the selection distortion problem could be to use a behavioural scheme like the one used in GA3 (page 59). Each individual (or alternatively, each x-coordinate) could have a threshold setting the probability of making a crossover with selection from a neighbourhood subset including the neighbour to the left, and another threshold setting the probability of making a crossover with selection from a neighbourhood subset including the neighbour to the right. These thresholds should be adapted based on the history of previous crossovers, such that e.g. an individual with a history of many previous unsuc-

cessful crossovers with an individual on the left would tend not to make that kind of crossovers. This kind of adaptiveness could lead to the formation of many almost non-interbreeding subpopulations positioned in bands over the length of the grid.

The idea of letting the breakdown time depend on the position in the population grid is related to the Terrain-Based Genetic Algorithm investigated by Gordon et al. in [50], in which parameter values of a diffusion GA (mutation rates and details of the crossover operator) are allowed to depend on the position in the grid. The objective of Gordon et al.'s approach is to let the algorithm itself find good parameter values.

Assume that the proposed algorithm could be implemented without reevaluation of objective functions to avoid distorted selection. A quick estimate of the number of fitness evaluations needed to solve a 1000 makespan 10×10 problem would be the following. In the proposed approach, 10 runs (one for each machine) of the algorithm are run. Assume a grid height of 10 is used (as in GA3), meaning that the population size becomes 10000. Assume the algorithm is run for 100 generations, this means 10^6 fitness evaluations per run, a total of 10^7 fitness evaluations. In addition to this, a conventional GA will need to be run to find the makespan of the problem, but assuming this GA will need 10^4 fitness evaluations, the running time contribution is negligible compared to the contribution of the other part of the algorithm.

Assume the same problem is solved using a standard GA finding each $C^*(b)$ separately. Using a population size 100 GA, running 100 generations, since there are approximately 10000 possible breakdowns, this will require 10^8 fitness evaluations. The proposed approach promises a running time improvement of approximately 90% in this example. An interesting question is, how the quality of the solutions found compares in the two different algorithms, but this will not be investigated in this thesis.

Alternatively, the standard GAs can each be run for 10 generations, letting the standard GA approach use the same number of fitness evaluations as the diffusion-based approach. But since 10 generations are usually not enough for a GA to converge on a good optimum, the diffusion-based approach could be expected to return better solutions.

6.4.2 A "population reuse" approach

In section 5.3.8, it was found that reusing the population when a GA was used for rescheduling after a breakdown could decrease the number of fitness evaluations needed to reach a certain level of performance. The idea of reusing the population is applicable to the problem of finding $C^*(b)$ for all $b \in B$ as well. For breakdowns happening on the same machine we expect the optimal schedule for breakdown

```
Initialise(pop)
set breakdowntime = 0
EvaluateFitness(pop)

for(gen = 1;gen <PRE_GENERATIONS;gen = gen + 1) do
   Evolve(pop)
   EvaluateFitness(pop)
od
set C(breakdowntime)=BestObjectiveValue(pop)

set breakdowntime = breakdowntime + 1
while(breakdowntime < minimal makespan) do
   EvaluateFitness(pop)
   for(gen = 0;gen <CYCLE_GENERATIONS;gen = gen + 1) do
      Evolve(pop)
      EvaluateFitness(pop)
   od
   set C(breakdowntime)=BestObjectiveValue(pop)
   set breakdowntime = breakdowntime + 1
od

set C(minimal makespan)=minimal makespan
```

Figure 6.9: *The algorithm for finding $C^*(b)$ for all $b \in B$ on a specific machine by reusing the population in an evolutionary algorithm.*

time x to be related to the optimal schedule for $x + 1$. Therefore, it may be beneficial to use a GA to find the schedule for x, and then reuse the population to find a schedule for $x + 1$. After finding the schedule for $x + 1$, the population can be used as a starting point to find the schedule for $x + 2$ etc. This process should keep on running until x is higher than the minimal makespan of the schedule.

The algorithm has been stated in more detail in figure 6.9. The algorithm is intended to find $C^*(b)$ for all breakdowns in B happening at a specific machine. The algorithm of the figure starts by using a pre-specified number of generations (PRE_GENERATIONS) for finding the schedule for breakdown time 0. After the `for`-loop, the value found for $C^*(b)$ is stored in $C(breakdowntime)$, the breakdown time is increased and the main loop entered. In the main loop the fitness of the population is reevaluated (since we are now working on a different breakdown), and the population is evolved for a number of generations (CYCLE_GENERATIONS). At the end of the main loop, $C(breakdowntime)$ is set and $breakdowntime$ increased. The main loop keeps running until *breakdown-*

time reaches the minimal known makespan for the problem.

The minimal makespan for the problem can be pre-calculated by another algorithm, but it can also be found during the run of the algorithm. During fitness evaluation, the makespan of every schedule is calculated during schedule construction. If the variable $minimal\ makespan$ is updated during fitness evaluation to always hold the lowest makespan found so far, at the end of the algorithm run it should hold the minimum makespan for the current problem. This is so since when the breakdown-time has increased beyond the completion of the final operation on the machine affected by the breakdown, the selection pressure in the algorithm will favour low makespan schedules.

The above algorithm has been implemented with the addition that before the algorithm starts, a lower bound on after rescheduling performance is calculated for all breakdown times. This is done using a simple bottleneck consideration for each machine as done in GA3, but also considering the breakdown. The two `for`-loops have been changed to also stop if the best performing individual in pop has reached the lower bound. This should save some computational effort.

In what follows, this algorithm will be referred to as *the reusing algorithm*. The details of the evolutionary part of the algorithm were as follows. The representation was permutation with repetition. A crossover rate of 0.8 was used, using GOX as the crossover operator. Individuals generated by crossover were subject to mutation with probability 0.2. The mutation operator used was PBM, and individuals not generated by crossover were generated by mutation. Every generation the entire population was replaced except for an elite of one. Selection was tournament based selection with a tournament size of two. Decoding was the hillclimbing decoder of GA3. For rescheduling after a breakdown the same hillclimber was used.

For comparison reasons, a GA finding each $C^*(b)$ from scratch was also made. This algorithm is termed *the reinitialising algorithm*. The details of the reinitialising algorithm were exactly the same as described for the reusing algorithm, except that the population was initialised at random every time a new breakdown was considered.

Experiments

The algorithms were run on the problems `la01`, `la02`, `la06`, `la07`, `la11`, `la12`, `la16`, `la17`, `ft10` and `ft20`. The problems `la26`, `la27`, `la31` and `la36` were not used in the experiments, since already for the smaller problems the running times were excessive. The set of breakdowns B included breakdowns on all machines with breakdown times in the range $\{0 \dots C_{max}\}$, where C_{max} is the minimal makespan of the problem, and duration $\tau_B = 80$.

Both algorithms were run 100 times on each problem. Each algorithm run

was allowed to run for the same number of generations; 100 to find the very first schedule for each machine, and 10 generations for each breakdown. In an attempt to estimate the true values of $C^*(w,t)$, the lowest estimates found for $C^*(w,t)$ in all runs were identified as

$$G(w,t) = \min\big(A_1^{reuse}(w,t)\ldots A_n^{reuse}(w,t), A_1^{reinit}(w,t)\ldots A_n^{reinit}(w,t)\big). \quad (6.2)$$

$A_i^{reuse}(w,t)$ denotes the estimate of $C^*(w,t)$ found in the ith run of the reusing algorithm, and $A_i^{reinit}(w,t)$ denotes the corresponding estimate found by the reinitialising algorithm. The notation $G(b)$ will sometimes be used instead of $G(w,t)$ to denote the best known makespan for breakdown b.

The behaviour of the two algorithms can be compared for four runs of each algorithm for the `ft10` and `ft20` problems in figure 6.10. In each plot, the makespans $A_1^{reuse}(w,t)$ and $A_1^{reinit}(w,t)$ found in one run of each of the algorithms have been plotted as a function of the breakdown time t. The best known values $G(w,t)$ have also been plotted. From these plots it is clear, that the reinitialising algorithm has a quite erratic behaviour. In many cases the plot moves up and down in a unpredictable fashion, often being quite far from the best known makespan. This is probably because the reinitialising algorithm does not have enough time to converge (recall that it is only allowed to run for 10 generations), and because it starts converging on different optima in the different runs. The performance of the reusing algorithm seems much more stable. The makespans found for breakdown times close to each other seem highly correlated; usually there is little or no difference between two adjacent breakdown times. Generally, the reusing algorithm performs better than the reinitialising algorithm, since it usually produces lower makespans. However, it is clear from the plots that the reusing GA is prone to get trapped in local minima, as it often finds makespans that are somewhat higher than the best known performance. For the easier problems (`la01`, `la06`, `la07`, `la11`, `la12` and `la07`), the reusing algorithm more consistently finds the best known solution. For these problems, the reinitialising algorithm also generally comes closer to the best known solutions, but in many cases it still performs worse than the reusing algorithm.

In order to do a better comparison of the two algorithms, the average estimates of $C^*(w,t)$ found by the two algorithms can be compared. The average estimate of $C^*(w,t)$ of algorithm x (where x is *reuse* or *reinit*) for a breakdown at machine w and time t is

$$\overline{A^x}(w,t) = \frac{1}{n}\sum_{i=0}^{n} A_i^x(w,t).$$

Plots of $\overline{A^x}(w,t)$ as a function of t for the problems and machines displayed in figure 6.10 can be seen in figure 6.11. Plots for a few other problems are in fig-

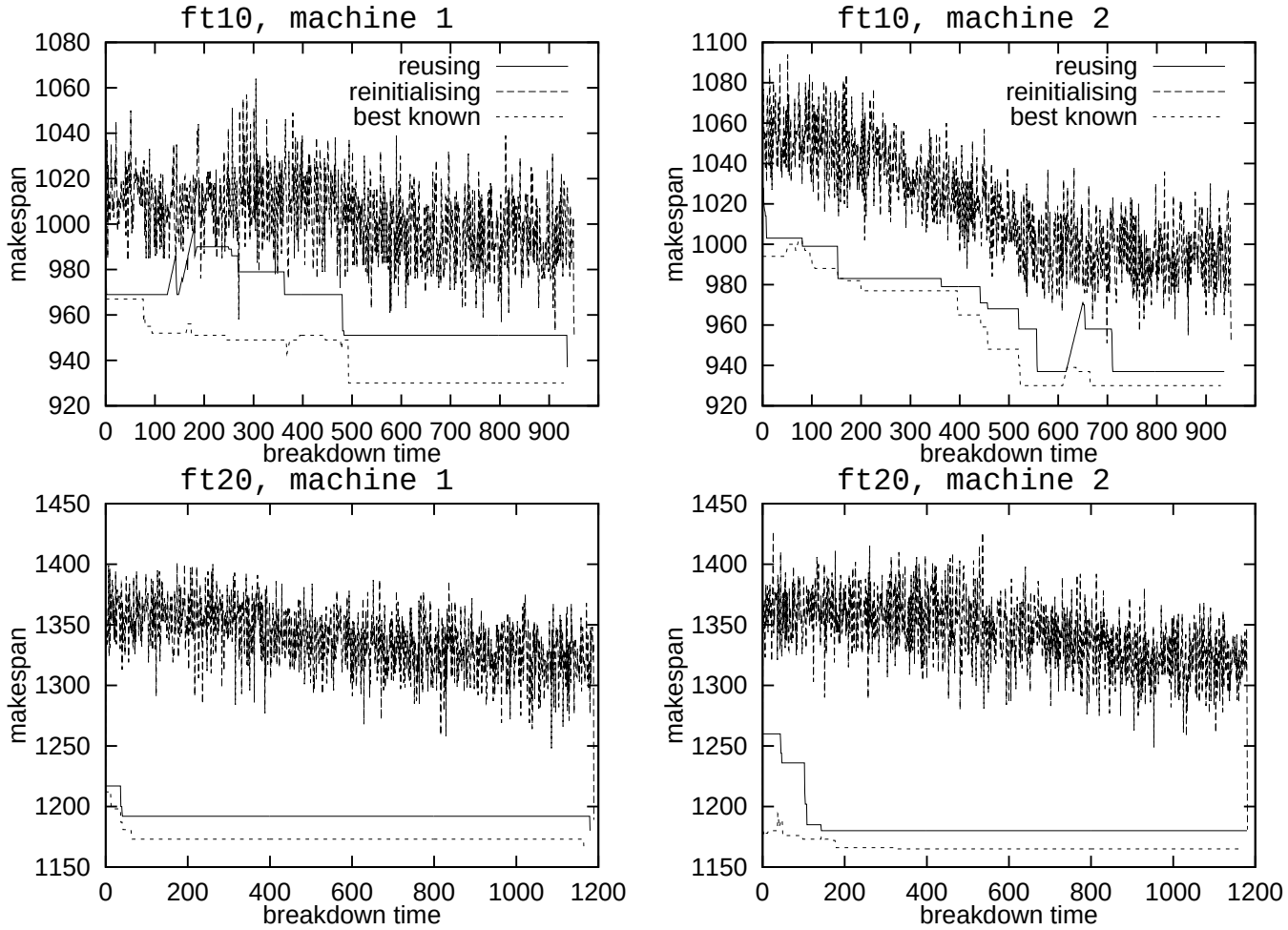

Figure 6.10: *Typical plots of A_i^{reuse} and A_i^{reinit}, the results of individual runs on the* `ft10` *and* `ft20` *problems. The best known makespans $G(w,t)$ have also been plotted. The legend on the plots for* `ft20` *is the same as on the* `ft10` *plots.*

ure 6.12. The plots of figure 6.11 confirm the impression from figure 6.10, that generally the reusing algorithm performs better than the reinitialising algorithm. On the plots of figure 6.11, the average for the reusing algorithm is always substantially lower than the reinitialising algorithm. This also holds for the plots for `ft10` and `la07` of figure 6.12, but it does not hold for the two plots for the `la02` problem. In parts of these plots, the reinitialising algorithm has lower makespan averages than the reusing algorithm, while in other parts of the same plots it is the other way around. These plots are very atypical, but they can be taken as an indication that sometimes it is better for the algorithm to start from scratch with a new population, instead of having to find the way out of a local optimum from an already converged population.

Inspecting the plots of $G(w,t)$ in figures 6.11 and 6.12, it seems that in some cases $G(w,t)$ is relatively constant over t. This is the case for the plots for `ft20` and `la07`. For other problems, $G(w,t)$ exhibits a lot of variation over t, even though usually it remains relatively constant within small ranges. From the plots for all the problems and all the machines (not shown), it seems that for problems with a job to machine ratio of three or higher (`la06`, `la07`, `la11`, `la12` and `ft20`), the $G(w,t)$ functions are relatively constant over t, while for problems with a job to machine ratio of one (`la16`, `la17` and `ft10`) the $G(w,t)$ plots are more complex. For the `la01` and `la02` problems with a job to machine ratio of two, one problem (`la02`) has complex $G(w,t)$ plots, while the other problem (`la01`) has relatively constant $G(w,t)$ plots. This observation is probably related to an observation often made in static scheduling: Generally problems with a high job to machine machine ratio are much easier to solve than problems with a low job to machine ratio [62].

It is hard to judge how different the true plots of $C^*(b)$ are from the $G(w,t)$ estimates. The extremely rugged shape of $G(w,t)$ within small intervals on the `la02` plots can be taken as an indication that in these intervals, $G(w,t)$ is probably not equal to $C^*(b)$. For some of the problems and some values of t and w, the $G(w,t)$ values are equal to the lower bounds calculated by the algorithm, or to the optimal makespans for the problems without any breakdown. In these cases, the $G(w,t)$ estimates are known to be equal to $C^*(b)$, but for other values of t and w, the accuracy is unknown.

In order to allow a systematic comparison of the two algorithms for all the problems and machines, the average makespans taken over the range of possible breakdown times have been calculated for every machine and problem. These averages are calculated as

$$\overline{A^x}(w) = \frac{1}{C_{max}} \sum_{t=0}^{C_{max}-1} \overline{A^x}(w,t),$$

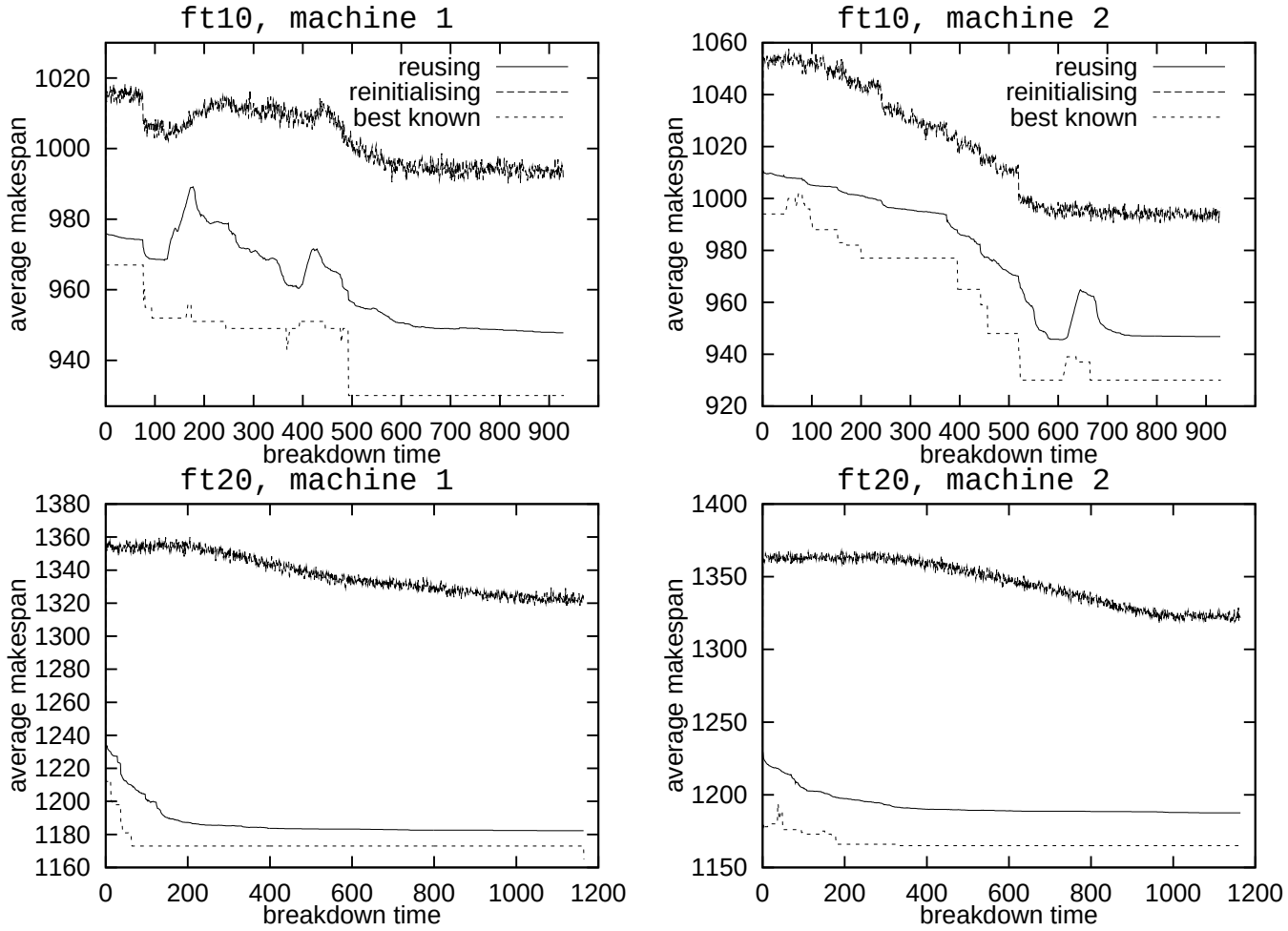

Figure 6.11: *Plots of $\overline{A^x}(w,t)$, the average estimates of $C^*(b)$ from the two algorithms for two machines of the* `ft10` *and* `ft20` *problems (same problems as in figure 6.10). The legend on the plots for* `ft20` *is the same as on the* `ft10` *plots.*

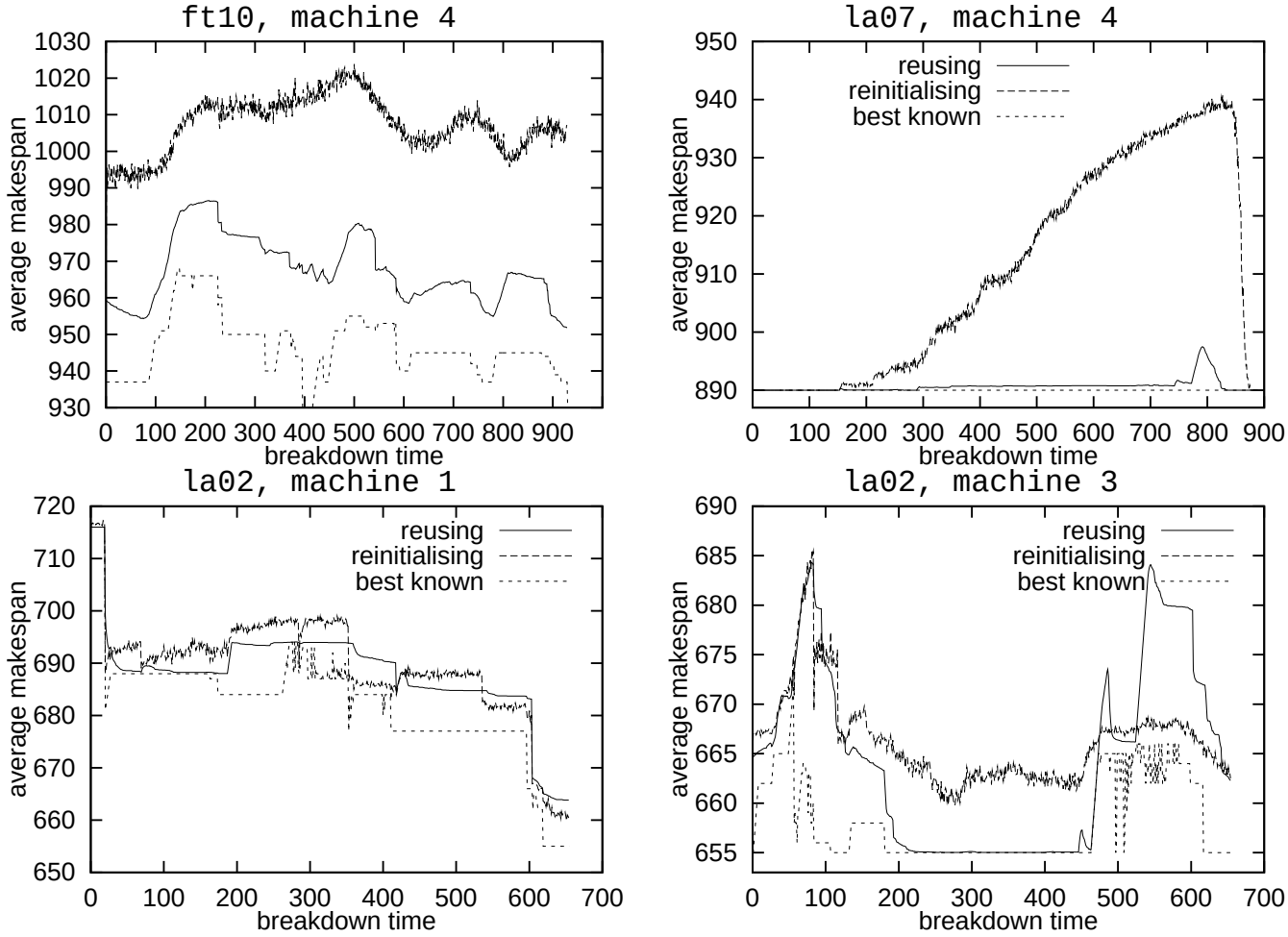

Figure 6.12: *Plots of $\overline{A^x}(w,t)$, the average estimates of $C^*(b)$ from the two algorithms for a few problem/machine combinations. The legend on the plot for* `ft10` *is the same as on the other figures.*

where again x stands for *reuse* or *reinit*. In a similar way, the average best found makespan can be calculated

$$\overline{G}(w) = \frac{1}{C_{max}} \sum_{t=0}^{C_{max}-1} G(w,t).$$

The relative error on a specific problem and machine over all the runs of algorithm x can now be defined

$$r^x(w) = \frac{\overline{A^x}(w) - \overline{G}(w)}{\overline{G}(w)}$$

In the following table, the r values for the reusing and the reinitialising algorithms can be compared. There is a subtable for every problem (the ten-machine problems `la16`, `la17` and `ft10` each have two subtables). The subtables have a column for every machine. The bottom row of each subtable reports the average best makespan $\overline{G}(w)$, while the middle rows report the $r^x(w)$ values for the two algorithms.

`la01`	M_1	M_2	M_3	M_4	M_5
reusing	0.00%	0.27%	0.03%	0.21%	0.00%
reinitialising	0.00%	0.28%	0.51%	0.24%	0.00%
avg.best.makespan	687.03	666.00	667.57	666.19	746.00

`la02`	M_1	M_2	M_3	M_4	M_5
reusing	0.86%	0.35%	0.92%	0.15%	0.96%
reinitialising	1.05%	1.21%	1.23%	0.18%	1.63%
avg.best.makespan	682.28	665.30	658.23	734.68	659.33

`la06`	M_1	M_2	M_3	M_4	M_5
reusing	0.00%	0.00%	0.00%	0.00%	0.00%
reinitialising	0.00%	0.00%	0.00%	0.01%	0.00%
avg.best.makespan	1006.00	926.00	926.00	926.00	926.00

`la07`	M_1	M_2	M_3	M_4	M_5
reusing	0.00%	0.01%	0.00%	0.08%	0.01%
reinitialising	0.00%	0.07%	0.10%	2.32%	0.01%
avg.best.makespan	968.11	890.44	890.00	890.00	890.18

`la11`	M_1	M_2	M_3	M_4	M_5
reusing	0.00%	0.00%	0.00%	0.00%	0.00%
reinitialising	0.00%	0.00%	0.00%	0.26%	0.00%
avg.best.makespan	1302.00	1222.00	1222.00	1222.00	1222.00

`la12`	M_1	M_2	M_3	M_4	M_5
reusing	0.00%	0.00%	0.00%	0.00%	0.00%
reinitialising	0.00%	0.00%	0.00%	0.00%	0.00%
avg.best.makespan	1106.21	1119.00	1039.00	1039.92	1039.00

la16	M_1	M_2	M_3	M_4	M_5
reusing	1.18%	1.56%	1.13%	2.38%	1.62%
reinitialising	3.00%	2.93%	3.17%	2.88%	2.89%
avg.best.makespan	959.94	948.73	979.74	950.82	950.17
la16	M_6	M_7	M_8	M_9	M_{10}
reusing	1.37%	1.46%	1.26%	1.39%	1.23%
reinitialising	3.04%	3.24%	2.91%	2.75%	2.92%
avg.best.makespan	962.55	957.12	951.44	950.72	952.09
la17	M_1	M_2	M_3	M_4	M_5
reusing	0.60%	0.42%	0.50%	0.76%	0.33%
reinitialising	0.95%	1.11%	0.88%	0.98%	0.82%
avg.best.makespan	791.28	792.95	786.46	827.33	788.36
la17	M_6	M_7	M_8	M_9	M_{10}
reusing	0.92%	0.92%	0.41%	0.46%	0.72%
reinitialising	1.07%	1.43%	0.64%	0.75%	1.54%
avg.best.makespan	789.08	798.60	790.12	786.14	792.42
ft10	M_1	M_2	M_3	M_4	M_5
reusing	2.06%	1.95%	2.45%	2.19%	2.21%
reinitialising	6.45%	6.33%	6.70%	6.38%	6.30%
avg.best.makespan	942.21	956.90	948.81	946.95	946.92
ft10	M_6	M_7	M_8	M_9	M_{10}
reusing	1.87%	2.38%	1.71%	1.73%	2.20%
reinitialising	6.47%	6.53%	5.29%	6.18%	6.47%
avg.best.makespan	941.04	951.93	964.19	949.34	941.91
ft20	M_1	M_2	M_3	M_4	M_5
reusing	1.10%	2.23%	0.17%	0.99%	2.95%
reinitialising	13.91%	15.34%	11.08%	9.50%	16.28%
avg.best.makespan	1174.16	1166.84	1197.97	1243.38	1174.71

Inspecting the numbers in the table, it becomes clear that there is a large variation from problem to problem. For the easiest problems, la01, la02, la06, la07, la11 and la12, the relative errors are very low for both algorithms. For the somewhat harder problems la16 and la17 (these problems are not really hard, just a little harder than the previous problems), the relative errors are a little higher than for the easiest problems, and there is a clear difference in the performance of the two algorithms. For la16 and la17, the relative error of the reinitialising algorithm is usually around twice as large as for the reusing algorithm. Turning to the hard problems of ft10 and ft20, the relative errors are much larger than they are for the other problems, and the performance difference between the two algorithms is substantial. For ft10, the relative error of the

reinitialising algorithm is typically 6%, while the error of the reusing algorithm is always less than one third of this. For `ft20`, the relative error of the reinitialising algorithm is in the range 9%-17%. The error of the reusing algorithm is often ten times smaller than this.

For these experiments, the relative error of the reusing algorithm is never higher than the relative error of the reinitialising algorithm, while the opposite is often true. For easy problems there is only a small difference between the average performances of the algorithms, while for hard problems, the difference can be large.

From the point of view of optimising worst deviation performance, the maximum difference between the $C^*(w,t)$ estimates $A(w,t)$ returned by the algorithm and the true $C^*(w,t)$ values may be of more interest than the average difference. This is so, since in a worst deviation algorithm any error in $A(w,t)$ will result in lack of selection pressure on the schedules to minimise schedule cost for the breakdown of machine w at time t. An error in the estimation of $C^*(w,t)$ by Δ can potentially lead to errors in the objective function evaluation $P_{deviation}(s)$ of Δ. Thus, it seems reasonable to expect a worst deviation algorithm working with a $C^*(b)$ estimate faulty by a value of Δ to return solutions suboptimal by at least Δ.

For one run of either the reusing or the reinitialising algorithm on all the machines of a problem, the maximum difference between the estimated values $A(w,t)$ and the best known values $G(w,t)$ can be calculated

$$\Delta_i^x = \max_{w\in\{1\ldots m\}} \max_{t\in\{0\ldots C_{max}-1\}} A_i^x(w,t) - G(w,t),$$

where x is *reuse* or *reinit*. Note that as was done for relative errors, we are using $G(w,t)$ instead of $C^*(w,t)$, since the latter is unknown. The average maximum error over the experiments can be calculated

$$\overline{\Delta}^x = \frac{1}{n}\sum_{i=1}^{n} \Delta_i^x.$$

$\overline{\Delta}^x$ has been calculated for every combination of problem and algorithm, the results are in table 6.11 The table confirms the results found for relative errors, that in some cases the reusing algorithm performs much better than the reinitialising algorithm. However, it also seems clear that even though the average relative errors reported earlier in this section seemed modest for the reusing algorithm, the maximum errors are substantial for most of the problems.

Even though the reusing algorithm clearly performs better than the reinitialising algorithm, there is room for improvement. The plots of figures 6.10-6.12 show that often the reusing algorithm gets stuck in local optima when t changes,

	la01	la02	la06	la07	la11
reusing	21.0	45.2	0.0	15.0	0.0
reinitialising	48.0	45.2	11.3	64.5	0.0

	la12	la16	la17	ft10	ft20
reusing	0.0	75.3	51.0	84.7	71.9
reinitialising	37.8	105.3	76.2	140.3	283.1

Table 6.11: *The $\overline{\Delta}^x$ values for the eight problems and the two algorithms.*

which is probably the reason for the high $\overline{\Delta}$ values found. Overcoming this problem seems an interesting direction of research. When t changes, there must be a tradeoff between reusing the knowledge present in the population and at the same time allowing new solutions to be found by exploring other parts of the search-space. Maybe adaptive control of mutation rates or speciation mechanisms can help solve this problem.

6.5 Worst deviation performance

In order to minimise the worst deviation performance

$$P_{deviation}(s) = \max_{b \in B}[C(s,b) - C^*(b)], \quad C^*(b) = \min_{s \in S} C(s,b),$$

knowledge is needed of $C^*(b)$ for every $b \in B$. This knowledge can be approximated using the methods of the last section. Once the $C^*(b)$ values have been found, $P_{deviation}(s)$ can be minimised in the same way $P_{worst\ case}(s)$ was minimised in section 6.1. In this section, the estimates of $C^*(b)$ from the last section will be used in coevolutionary and exact evaluation approaches similar to the ones of section 6.1 to minimise worst deviation performance $P_{deviation}(s)$.

6.5.1 Performance of C_{max}-minimised schedules

There is an upper limit on the worst deviation performance of a schedule produced by minimising $C_{max}(s)$. Since the hillclimber used for rescheduling is guaranteed to find a schedule which is at least as good as the schedule found using right-shifting, we must have

$$C_{max}(s,b) \leq C_{max}(s) + \tau_B,$$

where τ_B is the breakdown duration. Assuming s to be minimal with respect to $C_{max}(s)$, this gives us

$$\begin{aligned} P_{deviation}(s) &= \max_{b \in B}(C_{max}(s,b) - C^*(b)) \\ &\leq \max_{b \in B}(C_{max}(s) + \tau_B - C^*(b)) \\ &\leq \tau_B \qquad \text{(since } C^*(b) \geq C_{max}(s)\text{)}. \end{aligned}$$

Inspecting table 6.11 of average highest estimation errors on $C^*(b)$ for the reusing algorithm, it becomes clear that an algorithm minimising $P_{deviation}(s)$, basing the evaluation of $P_{deviation}(s)$ on the estimates returned by the reusing algorithm, should not be expected to perform much better than an algorithm simply minimising C_{max}, since in many cases the average highest estimation errors are of a size comparable to the highest expected error of a C_{max}-minimal schedule, $\tau_B = 80$.

6.5.2 The algorithm

An algorithm for minimising $P_{deviation}(s)$ needs two steps:

Step 1: Identify (or approximate) $C^*(b)$ for all $b \in B$.

Step 2: Locate a schedule minimising $P_{deviation}(s)$, using the $C^*(b)$ values found in step 1 to evaluate $P_{deviation}(s)$.

The two steps are independent, so any algorithm finding $C^*(b)$ can be used in step 1, and any algorithm minimising $P_{deviation}(s)$ using knowledge of $C^*(b)$ can be used in step 2. Note that an algorithm for minimising $P_{relative}(s)$ can be constructed in exactly the same way; the only difference would be the minimisation of $P_{relative}(s)$ instead of $P_{deviation}(s)$ in step 2.

For step 1, the reusing algorithm of section 6.4.2 can be used, but the $\overline{\Delta}$ values of table 6.11 indicate that if this is done, for many of the problems the schedule returned by step 2 of the algorithm should not be expected to have a worst deviation better than can be expected from an algorithm minimising C_{max}, since $\overline{\Delta}$ and τ_B are of comparable magnitudes. Instead of having a "real" step 1, the algorithm of this section will use as input for step 2 the $G(w,t)$ values identified in section 6.4.2. This estimate is probably reasonably accurate, and even if some of the $G(w,t)$ values are not accurate, the approach will still be useful for evaluating the performance of the algorithm proposed for step 2.

The worst deviation performance of the schedules found by the algorithm needs to be evaluated. Since exact knowledge of $C^*(b)$ is not available, the performance of the final schedule will be estimated as

$$P_{deviation}(s) \doteq \max_{b \in B}(C_{max}(s,b) - G(b)). \tag{6.3}$$

In this way, the same $G(b)$ values used for minimisation are used for evaluating the final solution. Clearly, a true evaluation of $P_{deviation}(s)$ would be preferable to this, but unfortunately it is not possible at this stage. An advantage of evaluating $P_{deviation}(s)$ as in (6.3) is that the performance of step 2 of the algorithm can be more accurately evaluated, since the error introduced in step 1 is ignored.

Two algorithms will be tested for step 2. The coevolutionary and exact evaluation approaches of section 6.1.2 will be changed to work with worst deviation performance. The performance of these algorithms will be compared to the worst deviation performance of a standard algorithm minimising makespan.

The coevolutionary algorithm used in step 2 is similar to the algorithm of section 6.1.2, except for three small changes. Firstly, the objective function for schedules is changed to $\max_{b \in P_B}[C_{max}(s,b) - G(b)]$ instead of $\max_{b \in P_B} C_{max}(s,b)$. Secondly, the asymmetric fitness evaluation is still used for the breakdowns, but the assignment is based on the values of $C_{max}(s,b) - G(b)$ instead of $C_{max}(s,b)$. Thirdly, the rounding of breakdown times is changed. For two breakdowns b_1 happening at machine w at time t and b_2 happening at the same machine at time $t+1$ and affecting the same operation, we know that $C_{max}(s,b_2) \geq C_{max}(s,b_1)$ because of the argument on page 155. Thus provided that $G(b_2) \leq G(b_1)$, we can increase the breakdown time from t to $t+1$, since we must have

$$C_{max}(s,b_2) - G(b_2) \geq C_{max}(s,b_1) - G(b_1).$$

In the algorithm this rounding is repeated iteratively until no further rounding is possible. The rounding of breakdown times increases the precision of the objective function evaluation in the algorithm, since it makes the evaluations focus on harder breakdowns. The rounded value for t is only used in the evaluation against the schedule s; it is not written back in the breakdown gene.

The exact evaluation algorithm used in step 2 is similar to exact evaluation algorithm of section 6.1.2, except that the objective function $\max_{b \in B}[C_{max}(s,b) - G(b)]$ is used instead of $\max_{b \in B} C_{max}(s,b)$. The selection of breakdowns to be tested against a specific schedule is changed such that only breakdowns that cannot be rounded up (by increasing t to $t+1$) are used.

All other details of the algorithms, such as rescheduling, representation, decoding, genetic operators and parameter values were exactly as described in section 6.1.2. The algorithm minimising C_{max} was identical to the preschedule performance GA used in section 6.1.2.

6.5.3 Experiments

Since the limitations of worst case performance stated in section 6.1 do not apply to worst deviation performance, the experiments were conducted allowing the

Problem	(12+6)	(16+8)	(20+10)	(24+12)	Presch. perf.	Exact eval.
la01	55.3	54.5	53.7	52.5	71.0	48.7
la02	74.9	73.9	72.4	72.0	75.1	66.3
la06	27.9	23.5	15.5	13.0	68.4	0.2
la07	77.2	77.1	77.1	76.8	76.8	76.0
la11	35.9	27.9	22.3	15.4	70.0	0.1
la12	24.7	16.9	12.1	13.3	66.3	0.0
la16	105.7	103.8	101.5	101.5	93.0	92.9
la17	89.0	88.2	87.5	87.7	83.3	82.9
ft10	117.2	115.7	115.6	115.2	107.6	106.0
ft20	105.3	104.0	102.8	103.4	104.4	97.9
Average	71.3	68.5	66.1	65.1	81.8	57.1

Table 6.12: *Average worst deviation performances.*

breakdowns to happen on all machines and at all times prior to the minimal makespan for the given problem.

The coevolutionary algorithm was run using the parameter values $(\mu + \lambda) = (12 + 6), (16 + 8), (20 + 10)$ and $(24 + 12)$. The maximum values of μ and λ were increased when comparing to the experiments on worst case performance, since the breakdown sets used in the experiments on worst deviation performance are larger, meaning that a larger search-space has to be covered by the breakdown population.

Each of the coevolutionary, exact evaluation and preschedule performance GAs were run 400 times on the problems, and the average worst deviation performance was calculated for every combination of algorithm and problem.

The average worst deviation performances are in table 6.12. Inspecting the table it becomes clear, that there is a huge variation from problem to problem. For some of the problems, the best worst deviations are very low; the average worst deviations of la06, la11 and la12 using the exact evaluation algorithm are all very close to zero. For other problems the average worst deviations are quite high, consider la16, la17, ft10 and ft20, in which the average worst case deviations for all the algorithms are higher than the theoretical upper bound for a makespan minimal schedule ($\tau_B = 80$). Comparing the average performances of the preschedule performance and exact evaluation GAs, it seems that for the problems in which the exact evaluation GA finds a high worst deviation value (la02, la07, la16, la17, ft10 and ft20), the preschedule performance GA finds a value which is a slightly higher. For the problems in which the exact evaluation algorithm finds low values (la01, la06, la11 and la12), the exact evaluation GA clearly outperforms the preschedule performance GA, which finds

Problem	la01	la02	la06	la07	la11
Best performance	45	60	0	76	0
Problem	la12	la16	la17	ft10	ft20
Best performance	0	75	80	65	62

Table 6.13: *The lowest worst deviation performances found for the ten problems.*

substantially higher worst deviation performances. The high worst deviation performances returned by all of the algorithms for some of the problems may be an indication that for these problems it is not possible to find schedules with a worst deviation performance much better than what is always attainable by minimising C_{max}.

Comparing the performance of the coevolutionary GA to the exact evaluation GA, the coevolutionary GA can be seen to yield substantially higher worst deviation performances than the exact evaluation algorithm, even for the highest settings of $(\mu + \lambda)$. The performance difference between the two algorithms is much larger than it was for the worst case performance experiments of section 6.2.1. The reason for this is probably two-fold. The sizes of breakdown sets B are much larger in the worst deviation performance experiments than they were in the worst case performance experiments. Even though the breakdown population sizes have been increased in the worst deviation experiments, they have not been scaled up as much as the B search-spaces. Besides of this, the rounding up of breakdown times possible for worst deviation performance does not always allow as much rounding up as worst case performance does, meaning that a better sampling of the breakdown times may be necessary. Comparing the worst deviation performance of the coevolutionary GA to the performance of the preschedule performance GA, the experiments indicate that for the problems for which the exact evaluation GA finds low worst deviation schedules the coevolutionary algorithm is also capable of outperforming the preschedule performance GA. For the problems for which the exact evaluation algorithm and the preschedule performance algorithm perform equally well, in some cases the coevolutionary algorithms return schedules that perform a little worse than the preschedule performance algorithm.

Considering the best found worst deviations for the ten problems (table 6.13), there is again a huge variation from problem to problem. For some of the problems, the lowest worst deviations are remarkably low. For the problems `la06`, `la11` and `12` the best found worst deviations are 0, which means that for these problems a schedule which is optimal for all possible breakdowns exists. For some of the other problems, the lowest worst deviation is equal or very close to the upper limit of $\tau_B = 80$.

The average processing times of the algorithms are displayed in table 6.14.

Problem	(12+6)	(16+8)	(20+10)	(24+12)	Presch. perf.	Exact eval.
la01	27.0	32.7	38.1	43.1	1.5	144.7
la02	37.6	48.3	58.6	47.3	1.3	422.0
la06	40.9	50.4	59.9	68.0	3.2	183.6
la07	42.3	52.7	63.2	75.3	3.3	211.5
la11	57.0	71.4	87.9	98.8	4.7	332.9
la12	55.5	68.4	82.5	94.5	4.3	316.7
la16	57.2	71.8	90.8	104.8	3.4	2394.2
la17	57.3	78.2	97.2	115.5	3.4	2398.9
ft10	62.2	82.4	98.5	118.6	4.0	3060.7
ft20	68.8	87.4	99.8	117.5	5.8	769.1
Average	50.6	64.4	77.7	88.3	3.5	1023.4

Table 6.14: *Average processing times (CPU-seconds) in the worst deviation experiments.*

The processing times include only the time used in "step 2" of the algorithm, so the time needed to estimate $C^*(b)$ is not included. Inspecting the running times, it becomes clear why the performance differences between the exact evaluation GA and the coevolutionary algorithm are so large; for some of the problems the (la16, la17 and ft10), the CPU-time consumption of the exact evaluation algorithm is more than twenty times that of the coevolutionary algorithm with the highest $(\mu + \lambda)$-setting. Considering this, it may be possible to improve performance of the coevolutionary algorithm to the same level as the exact evaluation algorithm, while still maintaining a lower processing time. By inspecting some of the individual runs of the coevolutionary GA it was found that the main reason for failure of the algorithm was when it returned a schedule evaluated to have a low worst deviation on the breakdown population. Using an exact evaluation (in order to evaluate the performance of the algorithm) the individual would subsequently be found to have a higher worst deviation performance. Thus, the performance of the coevolutionary GA could be improved by improving the precision of the objective evaluation of the schedules. This could be done by gradually increasing $(\mu + \lambda)$ during the run, or by simply using exact worst deviation evaluations very late in the run, or on schedules evaluated to have a low worst deviation performance on the breakdown population.

However, it is debatable how much effort it is worth putting into improving the performance of the coevolutionary algorithm at this stage; the time needed to create a proper estimate of $C^*(b)$ ("step 1" of the algorithm) will probably be larger than the time needed to run the exact evaluation algorithm.

6.6 Conclusion

A minimax formulation of job shop scheduling to minimise worst case and worst deviation cost has been presented. Two coevolutionary algorithms estimating the performance of the schedules by evaluating them against an evolving population of breakdowns has been developed. These algorithm have been compared to algorithms minimising preschedule cost, and algorithms evaluating the exact worst case or worst deviation cost of the schedules.

For worst case performance, the coevolutionary algorithm was found to create schedules with a substantially lower worst case cost than the preschedule performance algorithm. The coevolutionary approach was also found to be more efficient than the exact evaluation approach when considering the time needed to reach a certain level of performance, or the probability of reaching a certain level of performance with a given amount of processing time.

For worst deviation performance, the coevolutionary algorithm was not able to reach the same level of performance as an exact evaluation approach, but it performed better than the preschedule performance algorithm in most cases. The coevolutionary algorithm was much faster than the exact evaluation algorithm, and it can probably be improved to return schedules of the same quality as the exact evaluation approach, while still using less computational resources.

The tradeoff between worst case cost and preschedule cost has been investigated using a multi-objective approach, which incorporated parts of the coevolutionary algorithm. The experiments indicated that for some problems, there is a tradeoff between worst case cost and preschedule cost, while for other problems no tradeoff was found. In terms of average performance, the multi-objective algorithm was found to be able to compete with the preschedule performance algorithm as well as the coevolutionary algorithm using approximately the same number of calls to the rescheduler.

Two approaches to estimate the best achievable performance $C^*(b)$ for all breakdowns b possible for a problem have been presented. One of the approaches, a genetic algorithm reusing the schedule population from breakdown to breakdown, was implemented and tested. The algorithm was found to clearly outperform a more standard approach using the same number of fitness evaluations. However, the results returned by the population reuse approach were found to be too imprecise to be useful for minimising the worst deviation performance of schedules, so instead the best results found in all experiments were used in the experiments. The approaches for estimating $C^*(b)$ seem an interesting line of new research, especially since the method may also be useful in other fields than scheduling. Performance measures such as worst deviation or relative worst deviation may also be used when designing e.g. mechanical structures, in which case knowledge of a huge number of solutions to closely related problems will also be

needed. The approaches presented here will probably be extendable to cover that kind of problem.

Chapter 7

The Economic Lot and Delivery Scheduling Problem

The economic lot delivery and scheduling problem, *ELDSP*, was first defined by Hahm and Yano in [54]. Consider a production plant supplying components to an assembly facility. The plant produces several different components that are used at a constant rate at the assembly facility. The plant has one machine that produces all of the components one at a time. The components are accumulated and shipped to the assembly in deliveries. The plant incurs holding costs for the components waiting to be sent to the assembly, sequence independent setup costs when changing the production of the machine, production costs when making the components and delivery costs when sending a batch to the assembly. The task in the ELDSP is to find a production sequence and a cycle time that minimise the total costs at the plant.

This problem is relevant in supply chains in industries, such as the car industry. The problem formulation is based on the idea of minimising holding and transportation costs, as emphasised in just-in-time production. In the present formulation, the supplier plant is *captive* of the assembly; it only produces components to one assembly. Such suppliers exist in the car industry as well as other industries [54].

In this chapter a new polynomial time algorithm guaranteed to find optimal solutions to the ELDSP will be presented. The algorithm is a significant improvement over the previously known best algorithm for the problem, a heuristic published by Hahm and Yano in [54] that does not guarantee optimal solutions. The outline of the chapter is as follows. In the first section the problem will be defined. In the following sections a few basic properties of it will be discussed, and the heuristic proposed in [54] described. The new algorithm is presented in section 7.4. In section 7.5 problem properties that make the heuristic of [54] find suboptimal solutions are discussed, and an experimental comparison between the

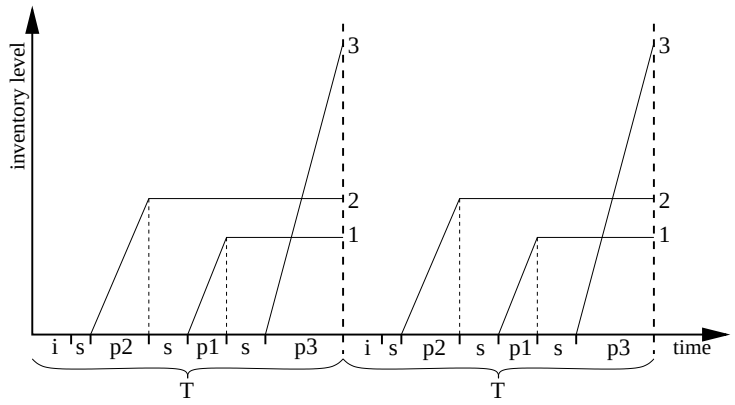

Figure 7.1: *Inventory levels for two production cycles. The problem has three components, labelled* $1, 2$ *and* 3*. The actions of the machine are indicated on the time axis.* `i` *indicates idle,* `s` *indicates setup,* `px` *indicates production of component* `x`*.*

two algorithms performed.

7.1 Problem formulation

Since the demand for the different components is constant, it seems natural to solve the problem by using a cyclic production of the components. Every time the cycle completes, there will be a delivery to the assembly. Within each cycle, each component needs to be produced in the amount specified by the demand. Because of the setup costs and setup times when changing the production it can never be advantageous to produce the same component twice during the same cycle. Because of the holding costs, it will always be optimal to start production at the latest possible time within each cycle. Thus, what is needed to solve a problem instance is the cycle time and the production sequence within each cycle. Since the objective is to minimise the total cost, there is a tradeoff between the setup and transportation costs on one side and the holding costs on the other. Minimising setup and transportation costs requires production of large batches meaning long cycle times, while minimising holding costs means keeping inventory levels low, i.e. short cycle times. The inventory levels of a few components in a schedule have been visualised in figure 7.1.

The following notation will be used:

J: Number of components.

A: Transportation cost per delivery.

D_j: Demand for component j per unit time.

S_j: Setup cost for component j.

s_j: Setup time for component j.

p_j: Production time per unit of component j.

h_j: Inventory holding cost of component j per unit per unit time.

T: Time cycle length. A decision variable.

$q = (q[1], q[2], \ldots, q[J])$: Production sequence vector. q contains a permutation of the numbers $1, \ldots, J$. A decision variable.

The ELDSP optimisation problem can be stated: Minimise the total cost per time TC:

$$TC = \frac{S + A}{T} + (\alpha + \beta)T + Z_1(q) + Z_2(q)T, \tag{7.1}$$

where

$$S = \sum_{j=1}^{J} S_j \tag{7.2}$$

$$\alpha = \frac{1}{2}\sum_{j=1}^{J} D_j h_j (1 - p_j D_j) \tag{7.3}$$

$$\beta = \sum_{j=1}^{J} D_j^2 p_j h_j \tag{7.4}$$

$$Z_1(q) = \sum_{i=1}^{J} D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} s_{q[j]} \tag{7.5}$$

$$Z_2(q) = \sum_{i=1}^{J} D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} D_{q[j]} p_{q[j]} \tag{7.6}$$

subject to

$$T \geq \tau_{min} = \frac{\sum_{j=1}^{J} s_j}{1 - \sum_{j=1}^{J} p_j D_j}.$$

The last inequality simply ensures that the cycle time is long enough to meet the demand. The first term in equation (7.1) reflects the total transportation and setup costs at the supplier, while the second term ($(\alpha + \beta)T$) reflects the holding cost at the assembly and the holding costs at the supplier during production. The remaining terms in the equation reflect the holding costs after production at the supplier.

7.2 Basic results

According to [4] and [54], given a production sequence the corresponding optimal production cycle time T can be found in time $O(J)$, using equation (5) of [54]. This can be shown by taking the derivative of TC with respect to T:

$$\frac{\partial TC}{\partial T} = -\frac{S+A}{T^2} + \alpha + \beta + Z_2(q).$$

Since $\frac{\partial^2 TC}{\partial T^2}$ can be seen to be positive and T is positive, the minimum of TC with respect to T for a fixed production sequence q can be found at:

$$\frac{\partial TC}{\partial T} = 0 \Rightarrow T = T^*(q) = \sqrt{\frac{S+A}{\alpha+\beta+Z_2(q)}}. \tag{7.7}$$

In the following, $T^*(q)$ will be used to denote the optimal cycle time for processing sequence q. The equation also gives a trivial upper bound on the optimal T value. Since $Z_2 \geq 0$, any optimal cycle time will satisfy

$$T \leq \tau_{max} = \sqrt{\frac{S+A}{\alpha+\beta}}.$$

Given a cycle time T the corresponding optimal production sequence q can be calculated in time $O(J \log J)$. The only sequence-dependent parts of the TC definition is $Z_1(q) + Z_2(q)T$, so in order to minimise TC, we must minimise this expression:

$$\begin{aligned} Z_1(q) + Z_2(q)T &= \sum_{i=1}^{J} \left(D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} T D_{q[j]} p_{q[j]} + s_{q[j]} \right) \\ &= \sum_{i=1}^{J} \left(-D_{q[i]} h_{q[i]} (F_i - T) \right), \end{aligned}$$

where F_i is the flow time of component i. From the last part of the equation it is evident, that this problem is equivalent to minimising a weighted flow time problem with weights $w_i = -D_i h_i$ and processing times $t_i = T p_i D_i + s_i$. According to theorem 2.4 of [4], this problem can be solved to optimality using a processing sequence satisfying

$$\frac{t_{q[1]}}{w_{q[1]}} \leq \frac{t_{q[2]}}{w_{q[2]}} \leq \ldots \leq \frac{t_{q_n}}{w_{q_n}} \Rightarrow$$

$$\frac{T p_{q[1]} D_{q[1]} + s_{q[1]}}{h_{q[1]} D_{q[1]}} \geq \frac{T p_{q[2]} D_{q[2]} + s_{q[2]}}{h_{q[2]} D_{q[2]}} \geq \ldots \geq \frac{T p_{q[J]} D_{q[J]} + s_{q_n}}{h_{q[J]} D_{q[J]}}. \tag{7.8}$$

```
sort sequence q^(0) in non-increasing order of p_j/h_j
set n = 0
do
   set n = n + 1
   set T_(n) = max(τ_min, sqrt((S+A)/(α+β+Z_2(q^(n-1)))))
   make sequence q^(n) from T_(n) satisfying eq.(7.8)
while (q^(n) ≠ q^(n-1) ∧ T^(n) ≠ τ_min)
return T_(n) and q^(n)
```

Figure 7.2: *Hahm and Yano's heuristic algorithm for the ELDSP.*

Given a cycle time T, this sequence can easily be constructed in time $O(J \log J)$, since it requires the sorting of J elements. The optimal processing sequence for cycle time T found in this way will be denoted $q^*(T)$ in the following.

7.3 Previous work

Two algorithms have previously been proposed to solve the ELDSP. An iterative improvement heuristic was developed in [54], while an evolutionary algorithm was suggested in [67]. Hahm and Yano's heuristic will be described in the next section. The evolutionary algorithm will not be dealt with in detail, since for the formulation of the ELDSP considered here it will be outperformed in terms of solution quality (and possibly speed as well) by the algorithm presented in section 7.4 and in terms of speed by the heuristic of [54].

7.3.1 Hahm and Yano's heuristic

Hahm and Yano's heuristic [54] is displayed in figure 7.2. The heuristic is an iterative improvement algorithm in which the production sequence and cycle time are changed alternately until no further improvement is made. The first sequence $q^{(0)}$ is chosen such that the first cycle time $T_{(1)}$ is known to be higher than the optimal cycle time. It is shown in [54] that the sequence $T_{(1)}, T_{(2)}, \ldots$ will be monotonically decreasing, until the algorithm stops in a global or local optimum. Furthermore, the final value of $T_{(n)}$ cannot be smaller than the optimum T value.

Hahm and Yano also create a modified version of the algorithm, which is similar to the one of figure 7.2, except that in the first step the sequence $q^{(0)}$ is ordered in reverse order; non-decreasing in $\frac{p_j}{h_j}$. This algorithm works in exactly the same way as the above algorithm, except that it can be shown to approach the optimal

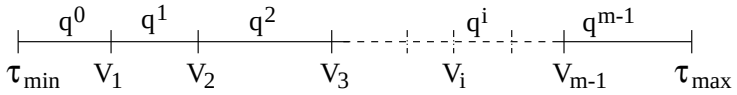

Figure 7.3: *The search-space of T is divided into a number of intervals. Inside each interval a specific processing sequence is optimal.*

cycle time T from below. Hahm and Yano show that if both algorithms return the same T the optimal solution has been found. If they return different T values, a bound on the error of the solution returned by the algorithm of figure 7.2 can be calculated.

Hahm and Yano did experiments for 72 randomly generated problem instances with three, six or nine components. They reported that for all these instances, the algorithm of figure 7.2 was able to locate the optimal solution.

7.4 The new algorithm

Assume a production cycle time T is given. The corresponding production sequence q is easily found using equation (7.8). Suppose the cycle time is changed to $T' = T + \delta$. The optimal production sequence q' corresponding to T' will be equal to q if equation (7.8) holds for T' and q. Due to the continuous nature of equation (7.8), this means that for certain intervals $[T_{min,i}, T_{max,i}]$ of T the optimal production sequence q_{opt} is fixed. This has been illustrated in figure 7.3.

Due to the structure of equation (7.8), only a small number of intervals on T can exist. Each interval endpoint T_{end} corresponds to two lines of equation (7.8) crossing each other. These interval endpoints can be found by solving a number of equations of the form

$$\frac{Tp_iD_i + s_i}{h_iD_i} = \frac{Tp_jD_j + s_j}{h_jD_j}.$$

Since at most $\frac{1}{2}J(J-1)$ different values of T can be found in this way, at most $\frac{1}{2}J(J-1)+1$ different intervals can exist. Furthermore, these interval endpoints can easily be found in time $O(J^2)$. Once the endpoints have been found, the intervals on T can be found by sorting the endpoints. Once the intervals have been constructed, the optimal production sequence q for each interval $[T_{min,i}, T_{max,i}]$ can be found using equation (7.8) with any value $t \in (T_{min,i}, T_{max,i})$. When the optimal sequence has been found for the interval $[T_{min,i}, T_{max,i}]$, the optimal value of T within the interval will either be $T_{min,i}$, $T_{max,i}$ or be given by equation (7.7). Since the intervals are known to cover the entire search-space of T, we are assured the global optimum will be found in this way. The algorithm has been written in

a more structured way in figure 7.4. A similar approach was used by Goyal, [51], to find an efficient algorithm for the Joint Replenishment Problem.

Time used in steps 4 and 6 is $O(1)$. Time used in step 1 is $O(J)$. Time used in step 3 is $O(J^2 log J)$. Time used inside the loop of step 5 is $O(J \log J)$. Time used in step 2 is $O(J^2)$. Time used in step 5 is $O(J^3 \log J)$. The loop in step 5 runs for $O(J^2)$ iterations, each iteration taking $O(J \log J)$ time. The time usage in step 5 is dominant, and the overall time used by the algorithm is $O(J^3 \log J)$.

7.4.1 Improving the algorithm runtime

The dominant contribution to the algorithm time usage is the calculation of the optimal processing sequence inside the main loop (step 5). This calculation can be improved by a bookkeeping scheme, making it possible to construct the new optimal processing sequence for the new interval from the processing sequence of the old interval. In the following, we assume for simplicity that only two components are interchanged when going from one interval to another (removing this assumption is not difficult).

Keep a datastructure E with changes in the processing sequence for each interval endpoint. $E_i = (c_{i,1}, c_{i,2})$ indicates that when going from interval i to $i+1$ the processing order of components $c_{i,1}$ and $c_{i,2}$ must be exchanged. The datastructure E can be initialised in steps 2 and 3 in the algorithm.

Keep a datastructure P with information about the current location of each component in the processing sequence. If $q[i] = c$ then $P[c] = i$. With this information available the optimal processing sequence in interval $i + 1$ can be constructed from the optimal processing sequence q of interval i by interchanging the values of $q_{P[c_{i,1}]}$ and $q_{P[c_{i,2}]}$. Since at most $\frac{1}{2}J(J-1)$ such interchanges can be done during the entire run of the algorithm, and since the new datastructures can be initialised in time $O(J^2)$, the $\log J$ factor vanishes.

After this improvement, the time complexity of the algorithm is $O(J^3)$, and the dominant contributions to the runtime are the calculations of $Z_1(q)$ and $Z_2(q)$ during calculation of optimal cycle time and total cost in the main loop (step 5). The cost of these calculations is $O(J)$, since each component has to be considered once in equations (7.5) and (7.6). This can be improved to $O(1)$ in a manner similar to the one used to improve the calculation of the optimal processing sequences. If we denote by q the optimal processing sequence at interval i and by q' the optimal processing sequence at interval $i+1$, the values $Z_1(q')$ and $Z_2(q')$ can be calculated from $Z_1(q)$ and $Z_2(q)$. Since we assume that only two consecutive components at positions $P[c_{i,1}] = a$ and $P[c_{i,2}] = a + 1$ are interchanged when

1. Calculate minimal allowable cycle time τ_{min} and maximum optimal cycle time τ_{max}:

$$\begin{aligned} \tau_{min} &= \frac{\sum_{i=1}^{J} s_i}{1 - \sum_{j=1}^{J} p_j D_j} \\ \tau_{max} &= \sqrt{\frac{S + A}{\alpha + \beta}}. \end{aligned}$$

2. For each pair of components i, j solve equation

$$\frac{T p_i D_i + s_i}{h_i D_i} = \frac{T p_j D_j + s_j}{h_j D_j}$$

   to find T. Store the T values larger than τ_{min} and smaller than τ_{max} in V.

3. Insert τ_{min} and τ_{max} in V and sort it in increasing order, removing duplicates. Set $n = \text{size}(V)$. The values in V correspond to $T_{min,i}$ and $T_{max,i}$.

4. Set $TC_{best} = \infty$.

5. `for` $(i = 1; i < n - 1; i = i + 1)$ {

   - set $T = \frac{1}{2}(V_i + V_{i+1})$.
   - Calculate optimal sequence $q^*(T)$ corresponding to T, using equation (7.8). q is guaranteed to be the optimal production sequence for all cycle times $T \in [V_i, V_{i+1}]$.
   - Calculate optimal cycle time $T^*(q)$ corresponding to q, using equation (7.7).
   - Calculate total costs $TC_T = TC(q, T)$, $TC_{V_i} = TC(q, V_i)$ and $TC_{V_{i+1}} = TC(q, V_{i+1})$ using equation (7.1).
   - `if` $TC_T < TC_{best} \wedge V_i < T < V_{i+1}$ `then`
     `set` $TC_{best} = TC_T,\ T_{best} = T,\ q_{best} = q$.
   - `if` $TC_{V_i} < TC_{best}$ `then`
     `set` $TC_{best} = TC_{V_i},\ T_{best} = V_i,\ q_{best} = q$.
   - `if` $TC_{V_{i+1}} < TC_{best}$ `then`
     `set` $TC_{best} = TC_{V_{i+1}},\ T_{best} = V_{i+1},\ q_{best} = q$.

   }

6. `return` T_{best} `and` q_{best}.

Figure 7.4: *The polynomial time algorithm to solve the ELDSP.*

going from i to $i+1$, the only entries changed in q are q_a and q_{a+1}:

$$\begin{aligned}
Z_1(q') &= \sum_{i=1}^{J} D_{q'_i} h_{q'_i} \sum_{j=i+1}^{J} s_{q'_j} \\
&= Z_1(q) - D_{q_a} h_{q_a} \sum_{j=a+1}^{J} s_{q[j]} - D_{q_{a+1}} h_{q_{a+1}} \sum_{j=a+2}^{J} s_{q[j]} \\
&\quad + D_{q_{a+1}} h_{q_{a+1}} \sum_{j=a+1}^{J} s_{q[j]} + D_{q_a} h_{q_a} \sum_{j=a+2}^{J} s_{q[j]} \\
&= Z_1(q) + D_{q_{a+1}} h_{q_{a+1}} s_{q_{a+1}} - D_{q_a} h_{q_a} s_{q_{a+1}} \\
Z_2(q') &= \sum_{i=1}^{J} D_{q'_i} h_{q'_i} \sum_{j=i+1}^{J} D_{q'_j} p_{q'_j} \\
&= Z_1(q) - D_{q_a} h_{q_a} \sum_{j=a+1}^{J} D_{q[j]} p_{q[j]} - D_{q_{a+1}} h_{q_{a+1}} \sum_{j=a+2}^{J} D_{q[j]} p_{q[j]} \\
&\quad + D_{q_{a+1}} h_{q_{a+1}} \sum_{j=a+1}^{J} D_{q[j]} p_{q[j]} + D_{q_a} h_{q_a} \sum_{j=a+2}^{J} D_{q[j]} p_{q[j]} \\
&= Z_1(q) + D_{q_{a+1}} h_{q_{a+1}} D_{q_{a+1}} p_{q_{a+1}} - D_{q_a} h_{q_a} D_{q_{a+1}} p_{q_{a+1}}
\end{aligned}$$

Since each such interchange can be done in time $O(1)$ and since there are at most $O(J^2)$ interchanges, the dominant contribution to time complexity now becomes step 3, the sorting of the interval endpoints. The time complexity of the final algorithm is $O(J^2 \log J)$.

7.5 Comparison to Hahm and Yano's heuristic

The heuristic presented by Hahm and Yano in [54] is able to solve the majority of problems to optimality. It does not guarantee optimal solutions, but it was able to find the optima of all 72 problems investigated by Hahm and Yano. In this section, insight will be given into how and why the heuristic fails in some cases.

With respect to the cycle time T, the heuristic approaches the optimum from above. It works by alternately calculating new values for T and q. This process is illustrated in figure 7.5. The total cost curves of three sequences q_1, q_2 and q_3 have been indicated on the figure, as functions of the cycle time T. The solutions considered by the algorithm have been marked by crosses, and the moves made

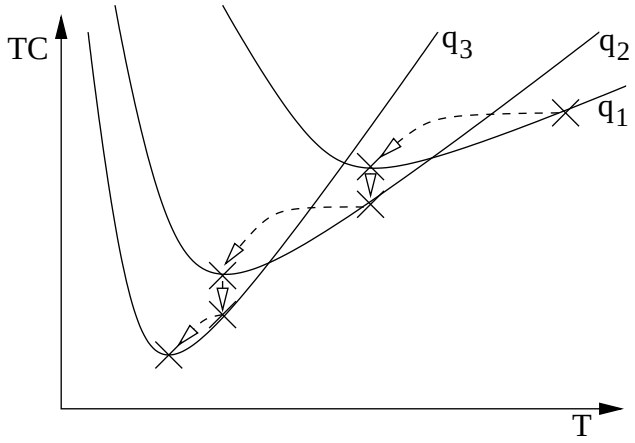

Figure 7.5: *The iterative improvement done by Hahm and Yano's algorithm. For this example, the algorithm finds the global optimum.*

visualised with dashed arrows. In the example of figure 7.5 the algorithm manages to find the optimal solution.

Hahm and Yano's algorithm keeps running until it finds a processing sequence q' for which

$$q' = q^*(T^*(q')). \tag{7.9}$$

If a suboptimal solution satisfies this, the algorithm may get stuck in it. So, the algorithm fails if it finds a processing sequence q which is suboptimal, but for which the optimal choice of cycle time does not lead to a new processing sequence in the next step of the algorithm. This kind of local optimum is illustrated on figure 7.6, along with the behaviour of the algorithm.

We introduce the following notation

- $(T^*(q_g), q_g)$ denotes the global optimum.
- $(T^*(q_l), q_l)$ denotes a local optimum if one exists.
- $O = \{q \mid \exists T : q = q^*(T)\}$ the set of processing sequences optimal for some cycle time.
- $C_1 = \{q \mid \exists T > T^*(q_g) : q = q^*(T)\}$ the set of processing sequences in O with optimal cycle times greater than the globally optimal cycle time.

In the following, a number of characteristics that make the heuristic fail will be discussed. Problems that make the heuristic fail but do not satisfy the following may exist.

The total cost curve of a production sequence $q \in C_1$ can basically be positioned in two different ways relative to the globally optimal production sequence.

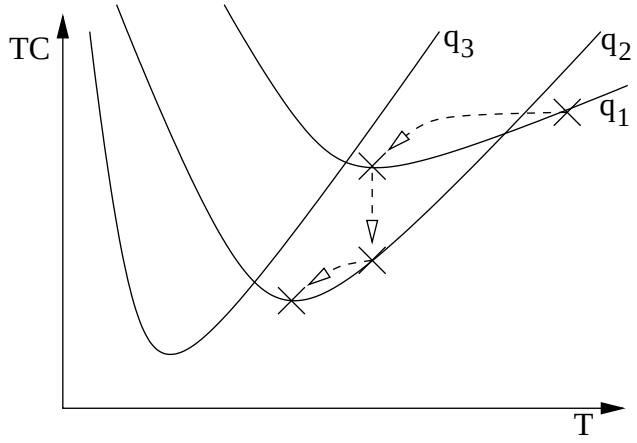

Figure 7.6: *The iterative improvement done by Hahm and Yano's algorithm. In this example, the algorithm gets trapped in a local optimum.*

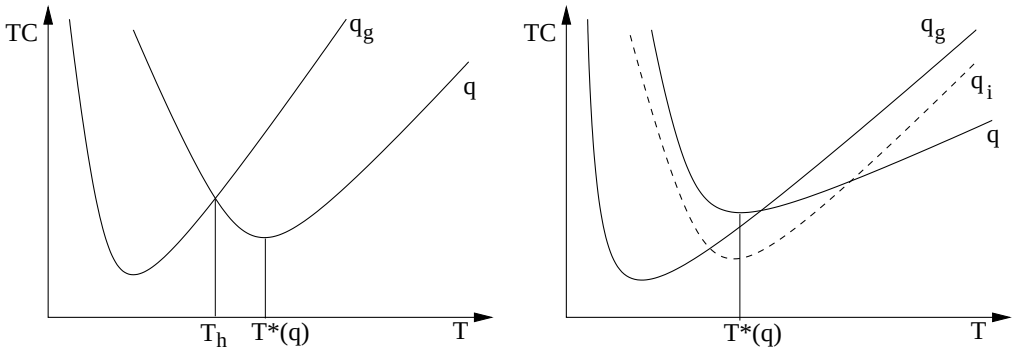

Figure 7.7: *Illustration of the two ways the total cost curve of a production sequence $q \in C_1$ can be placed relative to the total cost curve of the globally optimal q_g.*

These have been illustrated on figure 7.7. In the left plot, the optimal choice of T for sequence q places the total cost below the total cost curve for the optimal sequence q_g. It will not be possible for Hahm and Yano's heuristic to move from q to q_g directly; if the algorithm cannot move to any other production sequence either, q may act as a local minimum, trapping the algorithm. Production sequences placed in this way satisfy

$$\exists T_h : T^*(q_g) < T_h < T^*(q_l) \text{ and } TC(T_h, q_g) = TC(T_h, q), \tag{7.10}$$

that is a local maximum separating the two minima exists at T_h, between the cycle times optimal for the two production sequences.

In the right plot of figure 7.7 the total cost curves of q and q_g have been plotted in solid. The optimal choice of T for sequence q places the total cost above the total cost curve for the optimal sequence q_g. Thus, it will be possible for the heuristic to move directly from q to q_g unless there is another production sequence q_i with a total cost curve located below the total cost curve of q at $T^*(q)$. Such a total cost curve has been plotted dashed on the plot. In order for a production sequence q placed in this way not to allow a direct move to the optimum production sequence, the following must be satisfied.

$$\exists q_i \in C_1 : TC(T^*(q), q_i) < TC(T^*(q), q_g) < TC(T^*(q), q). \tag{7.11}$$

Since Hahm and Yano's algorithm cannot move directly from any production sequence in C_1 satisfying (7.10) or (7.11) to the optimal production sequence, the algorithm is guaranteed to fail on a problem that satisfies (7.10) or (7.11) for all $q \in C_1$, provided that the heuristic does not initialise $q^{(0)}$ to the optimal sequence. Note that the heuristic may also fail for problems not satisfying (7.10) or (7.11) for all sequences in C_1; the heuristic may also fail if some of the sequences do not satisfy (7.10) or (7.11), as long as the sequences actually considered during the algorithm run do.

In order to generate problems for which the heuristic fails we will now focus on problems for which all the production sequences in C_1 are likely to satisfy (7.10). Requiring (7.10) to hold for all $q \in C_1$, solving for T_h and using (7.7) leads to

$$\forall\, q \in C_1 : \sqrt{\frac{A+S}{\alpha+\beta+Z_2(q_g)}} < \frac{Z_1(q_g)-Z_1(q)}{Z_2(q)-Z_2(q_g)} < \sqrt{\frac{A+S}{\alpha+\beta+Z_2(q)}}. \tag{7.12}$$

This requires $Z_1(q_g) < Z_1(q)$ and $Z_2(q_g) > Z_2(q)$ for all $q \in C_1$, which is the same as

$$\forall q \in C_1 : \sum_{i=1}^{J} D_{q_g[i]} h_{q_g[i]} \sum_{j=i+1}^{J} s_{q_g[j]} < \sum_{i=1}^{J} D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} s_{q[j]} \tag{7.13}$$

Number of components	Number of problems generated	Value of F	Number of problems satisfying (7.13) (7.14)	Maximum relative TC difference	Average relative TC difference
2	10000000	10	26525	1.018%	0.226%
2	10000000	60	7295	1.235%	0.289%
3	10000000	2	12194	0.519%	0.061%
3	10000000	30	660	1.040%	0.202%
5	10000000	1	1617	0.191%	0.014%
5	10000000	6	72	0.359%	0.085%
7	10000000	1	99	0.062%	0.009%
7	10000000	4	4	0.113%	0.058%

Table 7.1: *Results of the experiments on randomly generated problems.*

$$\forall q \in C_1 : \sum_{i=1}^{J} D_{q_g[i]} h_{q_g[i]} \sum_{j=i+1}^{J} D_{q_g[j]} p_{q_g[j]} > \sum_{i=1}^{J} D_{q[i]} h_{q[i]} \sum_{j=i+1}^{J} D_{q[j]} p_{q[j]}. \quad (7.14)$$

Inequalities (7.13) and (7.14) are likely to hold if there is a component r for which s_r is much larger than s_i and p_r is much smaller than p_i for all $i \in \{1..J\} \setminus \{r\}$. Component r may end up being scheduled late in the sequence because of the small p_r, while it should be scheduled early because of the large s_r.

Problems likely to satisfy (7.13) and (7.14) were generated at random by drawing A from $U(0,5)$, and for each component but one drawing S_i from $U(0,1)$, s_i from $U(0,0.25)$ and h_i from $U(0,1)$. p_i and D_i were generated from uniform distributions with values satisfying $0 < p_i D_i < \frac{1}{J}$ in order to insure feasibility. For one component r, s_r and p_r were generated using $s_r = F \sum_{j \neq r} s_j$ and $p_r = \frac{1}{F} \min_{j \neq r} p_j$, where F is a parameter used to increase the probability of generating problems satisfying (7.13) and (7.14). The problems generated were then solved using Hahm and Yano's heuristic and the new algorithm. In the experiments, a number of values in the range $\{1, .., 100\}$ were tried out for the value of F. The properties of the problems were found to depend on F, so in the following results are reported for a few values of F for every problem size.

The experiments revealed that it is quite hard to generate problems for which the Hahm and Yano heuristic fails, and when it fails the total cost of the solution found by the heuristic is only slightly higher than the cost of the globally optimal solution. A summary of the experiments can be found in table 7.1. The experiments indicated that the probability of generating a problem for which the heuristic fails decreases as the number of components increase. This is probably

because the size of the set C_1 increases with the number of components, leading to a more smooth total cost landscape, which is easier searchable for the heuristic. Even for a very small number of components the probability of generating a problem on which the heuristic fails is very small; for a two component problem it is less than one percent regardless of the setting of F. The experiments also indicate that for the problems solved sub-optimally by the heuristic, the distance to the optimum in terms of total cost is very small. The averages reported in the last column of table 7.1 were taken over the problems satisfying (7.13) and (7.14) and hence solved sub-optimally by the heuristic, so the average cost-difference over all the problems generated is much smaller.

None of the experiments succeeded in generating a problem instance in which the solution produced by the Hahm and Yano heuristic was suboptimal by more than 1.3%. This despite a lot of effort was put into trying out different ways of generating problem parameters. Some of the problems were even used as a starting point for a simple $(1 + 1)$-evolutionary strategy to increase the relative error produced by the heuristic. Based on these experiments it is tempting to conjecture that something in the structure of the ELDSP makes it impossible to generate problems with very bad locally optimal solutions.

7.5.1 Which algorithm is preferable?

Comparing the new algorithm to the heuristic, it has been found that for the vast majority of problems the heuristic manages to find the global optimum. However, it has also been demonstrated that problems do exist for which the heuristic finds suboptimal solutions, while the new algorithm is guaranteed to always find the optimum. In terms of running time there is no reason to prefer the heuristic over the exact algorithm, since both algorithms are very fast[1], and since for real world problems once a solution to a problem has been found, it can be expected not to change for some time, meaning that the need to solve problems of this kind will not be frequent.

[1]The current implementation of the new algorithm (the $O(J^3 \log J)$ version of figure 7.4) solves around 30000 seven component problems or 3000 thirty component problems in one second on a 250MHz SGI O2 computer. The heuristic is even faster.

Chapter 8

Conclusion

This thesis has presented work and progress relevant to stochastic scheduling, evolutionary computation, and static scheduling. The main contributions of this work are in the area of stochastic job shop scheduling, for which we have presented a number of new ideas.

The neighbourhood based robustness approach has been demonstrated to significantly improve the robustness and flexibility of job shop schedules facing machine breakdowns when average performance is considered. We have shown that the method works for a range of problem types. It has been demonstrated to increase schedule robustness and flexibility with regard to four different rescheduling methods for makespan, maximum tardiness and loose summed tardiness problems. For tight summed tardiness and total flowtime problems the method has been shown to improve schedule robustness and to a smaller extent flexibility. The neighbourhood based robustness approach produces schedules of a quality equal to the schedules produced by minimising the slack based robustness measure, a state-of-the-art method for makespan job shops facing breakdowns.

A coevolutionary approach for creating schedules guaranteeing a certain level of worst case makespan performance for a set of scenarios has been developed. For medium or large sized problems, the approach is able to reach a specific level of performance faster and more reliably than a more standard approach. The possibility of a tradeoff between worst case performance and static schedule performance has been investigated using a multi-objective algorithm. The existence of the tradeoff has been found to be problem dependent; for some of the problems the worst case optimal schedule was also optimal in terms of static schedule performance, while for others no single solution simultaneously optimised both measures.

The coevolutionary algorithm used for worst case performance scheduling is a special case of a new algorithm generally applicable to minimax optimisation problems. The algorithm is also applicable on problems from outside the schedu-

ling domain, and has been demonstrated to be capable of solving problems without a certain symmetric property, something previously published coevolutionary algorithms for minimax problems have been unable to do.

In order to be able to work with worst deviation performance, we have presented a genetic algorithm for solving a large number of closely related scheduling problems. To our best knowledge, no algorithms for doing this have previously been published. The algorithm has been demonstrated to solve easy problems well, while it does not perform well on hard problems. The algorithm used for solving many related scheduling problems is quite general; it can be used for other applications in which a set of closely related problems need to be solved. The experiments indicate that the method is prone to get stuck in local minima, so an interesting line of research is the development of techniques for preventing this.

The results from the algorithm for solving many related problems simultaneously have been used to produce schedules with a good worst deviation performance. The worst deviation was based on the makespan criterion, and a hill-climber was used for rescheduling. This was done using two algorithms, one using an exact evaluation approach, and one using the coevolutionary minimax algorithm. The experiments showed that in many cases the exact evaluation algorithm was able to produce schedules with a much better worst deviation than a standard scheduling approach, while never being inferior to it. The experiments also suggested that there is a huge variation in the best achievable worst deviation performance from problem to problem. For some problems the best worst deviation is only slightly better than what can be achieved using a standard scheduling algorithm considering only preschedule cost, while for other problems a much better worst deviation can be achieved. When comparing the performance of the coevolutionary algorithm to the exact evaluation algorithm, the coevolutionary algorithm was found to be much faster than the exact evaluation algorithm, while being slightly inferior in terms of schedule quality.

A new algorithm for solving the economic lot and delivery scheduling problem has been presented. The algorithm runs in polynomial time, and guarantees that the global optimum of the problem will be found. No previously published algorithm has been able to do this. The algorithm has been compared to a previously published heuristic in a computational study, and has been found to outperform the heuristic in some cases.

Bibliography

[1] Aarts, van Laarhoven, Lenstra, and Ulder. A computational study of local search algorithms for job shop scheduling. *ORSA Journal on Computing*, 6(2), 1994.

[2] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401, March 1988.

[3] I. Al-Harkan. *On merging sequencing and scheduling theory with genetic algorithms to solve stochastic job shops*. PhD thesis, University of Oklahoma, 1997.

[4] K. R. Baker. *Introduction to sequencing and scheduling*. John Wiley and Sons, 1974.

[5] H. J. C. Barbosa. A genetic algorithm for min-max problems. In V. Uskov, B. Punch, and E. D. Goodman, editors, *Proceedings of the First International Conference on Evolutionary Computation and Its Applications*, pages 99–109, 1996.

[6] H. J. C. Barbosa. A coevolutionary genetic algorithm for a game approach to structural optimization. In *Proceedings of the seventh International Conference on Genetic Algorithms*, pages 545–552, 1997.

[7] H. J. C. Barbosa. A coevolutionary genetic algorithm for constrained optimization. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proceedings of the 1999 Congress of Evolutionary Computation*, volume 3, pages 1605–1611. IEEE Press, 1999.

[8] T. Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*, chapter B2 - Theoretical Foundations and Properties of Evolutionary Computation. IOP Publishing and Oxford University Press, 1997.

[9] T. Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*, chapter C2 - Selection. IOP Publishing and Oxford University Press, 1997.

[10] T. Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*, chapter C6 - population structures. IOP Publishing and Oxford University Press, 1997.

[11] J. C. Bean, J. R. Birge, J. Mittenthal, and C. E. Noon. Matchup schedules with multiple resources, release dates and disruptions. *Operations Research*, 39:470–483, 1991.

[12] D. Beasley, D. R. Bull, and R. R. Martin. Reducing epistasis in combinatorial problems by expansive coding. In S. Forrest, editor, *The Proceeding of the Fifth International Conference on Genetic Algorithms*, pages 400–407. Morgan Kaufmann Publishers, 1993.

[13] A. Ben-Tal and A. Nemirovski. Robust convex optimization. *Mathematics of Operations Research*, 23(4):769–805, November 1998.

[14] C. Bierwirth. A generalized permutation approach to job shop scheduling with genetic algorithms. *OR Spektrum*, 17:87–92, 1995.

[15] C. Bierwirth, H. Kopfer, D. C. Mattfeld, and Ivo Rixen. Genetic algorithm based scheduling in a dynamic manufacturing environment. In *Proceedings of IEEE Conference on Evolutionary Computation*, pages 439–443, 1995.

[16] C. Bierwirth and D. C. Mattfeld. Production scheduling and rescheduling with genetic algorithms. *Evolutionary Computation*, 7(1):1–17, 1999.

[17] C. Bierwirth, D. C. Mattfeld, and H. Kopfer. Permutation representations for scheduling problems. In *Proceedings of the 4th Conference on Parallel Problem Solving from Nature*. Springer Verlag, 1996.

[18] J. Błazewicz, K. H. Ecker, G. Schmidt, and J. Węglarz. *Scheduling in Computer and Manufacturing Systems*. Springer, 1994.

[19] P. Blæsild and J. Granfeldt. *Statistik for biologer og geologer*. Department of Mathematics, University of Aarhus, 1995.

[20] J. Branke. Creating robust solutions by means of evolutionary algorithms. In *Parallel Problem Solving from Nature V*, LNCS vol. 1498, pages 119–128. Springer Verlag, 1998.

[21] J. Branke and D. C. Mattfeld. Anticipation in dynamic optimization: The scheduling case. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VI proceedings*, LNCS vol. 1917, pages 253–262. Springer, 2000.

[22] C. A. Brizuela and N. Sannomiya. A diversity study in genetic algorithms for job shop scheduling problems. In *GECCO 99, Proceedings of the Genetic and Evolutionary Computation Conference*, pages 75–82. Morgan Kaufmann publishers, 1999.

[23] Peter Brucker. *Scheduling Algorithms*. Springer, 1998.

[24] R. Bruns. Scheduling. In T. Bäck, D. B. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, chapter F1 - Evolutionary Computation Applications. IOP Publishing and Oxford University Press, 1997.

[25] Ralf Bruns. Direct chromosome representation and advanced genetic operators for production scheduling. In Stephanie Forrest, editor, *Proceedings of the fifth international conference on Genetic Algorithms*, 1993.

[26] J. Carlier and E. Pinson. An algorithm for solving the job-shop scheduling problem. *Management Science*, 35(2):164–176, 1989.

[27] H. M. Cartwright and A. L. Tuson. Genetic algorithms an flowshop scheduling: towards the development of a real-time process control system. In T. C. Fogarty, editor, *Proceedings of the AISB Workshop on Evolutionary Computing*, LNCS vol. 865, pages 277–290. Springer-Verlag, 1994.

[28] G. A. Cleveland and S. F. Smith. Using genetic algorithms to schedule flow shop releases. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 160–169, 1989.

[29] C. A. C. Coello. A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information Systems*, 1(3):269–308, 1999.

[30] D. Corne and P. Ross. Practical issues and recent advances in job- and open-shop scheduling. In Z. Michalewicz and D. Dasgupta, editors, *Evolutionary Algorithms in Engineering Applications*, pages 531–546. Springer, 1997.

[31] D. W. Corne, J. D. Knowles, and M. J. Oates. The pareto envelope-based selection algorithm for multiobjective optimization. In M. Schoenauer,

K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VI proceedings*, LNCS vol. 1917, pages 839–848. Springer, 2000.

[32] P. J. Darwen and X. Yao. Evolving robust strategies for iterated prisoner's dilemma. In X. Yao, editor, *Progress in Evolutionary Computation*, volume 956 of *Lecture Notes in Artificial Intelligence*, pages 276–292. Springer, 1994.

[33] Lawrence Davis. Job shop scheduling with genetic algorithms. In John J. Grefenstette, editor, *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 136–140, 1985.

[34] K. Deb. Evolutionary Algorithms for Multi-Criterion Optimization in Engineering Design. In Kaisa Miettinen, Marko M. Mäkelä, Pekka Neittaanmäki, and Jacques Periaux, editors, *Evolutionary Algorithms in Engineering and Computer Science*, chapter 8, pages 135–161. John Wiley & Sons, Ltd, Chichester, UK, 1999.

[35] K. Deb, S. Agrawal, and A. T. Meyarivan. A fast and elitist multi-objective genetic algorithm: NSGA-II. Technical report, Kanpur Genetic Algorithms Laboratory (KanGAL), Indian Institute of Technology Kanpur, 2000. KanGAL Report No. 200001.

[36] K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VI proceedings*, LNCS vol. 1917, pages 849–858. Springer, 2000.

[37] K. Deb, A. Pratab, and S. Moitra. Mechanical component design for multiple objectives using elitist non-dominated sorting GA. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VI proceedings*, LNCS vol. 1917, pages 859–868. Springer, 2000.

[38] K. Deb and W. M. Spears. Speciation methods. In T. Bäck, D. B. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, chapter C6.2. IOP Publishing and Oxford University Press, 1997.

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

[40] V. F. Dem'yanov and V. N. Malozemov. *Introduction to minimax*. John Wiley & sons, 1974.

[41] Hsiao-Lan Fang. *Genetic Algorithms in Timetabling and Scheduling*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994.

[42] Hsiao-Lan Fang, Peter Ross, and Dave Corne. A promising genetic algorithm approach to job-shop scheduling, rescheduling, and open-shop scheduling problems. In Stephanie Forrest, editor, *Proceedings of the fifth International Conference on Genetic Algorithms*, 1993.

[43] H. Fisher and G.L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In J.F. Muth and G.L. Thompson, editors, *Industrial Scheduling*, pages 225–251. Prentice Hall, 1963.

[44] C. M. Fonseca and P. J. Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In S. Forrest, editor, *Proceedings of the fifth International Conference on Genetic Algorithms*, 1993.

[45] S. French. *Sequencing and Scheduling*. Mathematics and its applications. Ellis Horwood Limited, 1982.

[46] M. R. Garey and D. S. Johnson. Strong NP-Completeness Results: Motivation, Examples and Implications. *Journal of the ACM*, 25:499–508, 1978.

[47] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[48] B. Giffler and G. L. Thompson. Algorithms for solving production scheduling problems. *Operations Research*, 8:487–503, 1960.

[49] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[50] V. S. Gordon, R. Pirie, A. Wachter, and S. Sharp. Terrain-based genetic algorithm (TBGA): Modeling parameter space as terrain. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 229–235, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[51] S. K. Goyal. Determination of optimum packaging frequency of items jointly replenished. *Management Science*, 21(4):436–443, 1974.

[52] B. Greene. Crossover and diploid dominance with deceptive fitness. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proceedings of the 1999 Congress on Evolutionary Computation*, volume 2, pages 1369–1376, Mayflower Hotel, Washington D.C., USA, 1999. IEEE Press.

[53] J. Grefenstette, R. Gopal, B. Rosmaita, and D. Van Gucht. Genetic algorithms for the travelling salesman problem. In John J. Grefenstette, editor, *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 160–165, 1985.

[54] J. Hahm and C. A. Yano. The economic lot and delivery scheduling problem the common cycle case. *IIE Transactions*, 27:113–125, 1995.

[55] E. Hart and D. Corne. The state of the art in evolutionary approaches to timetabling and scheduling. Technical report, EvoStim working group on Scheduling and Timetabling, 1997.

[56] E. Hart and P. Ross. An immune system approach to scheduling in changing environments. In *GECCO 99, Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1559–1566. Morgan Kaufmann publishers, 1999.

[57] E. Hart, P. Ross, and J. Nelson. Producing robust schedules via an artificial immune system. In *Proceedings of the IEEE World Congress on Computational Intelligence*, 1997.

[58] J. W. Herrmann. A genetic algorithm for minimax optimization problems. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proceedings of the 1999 Congress on Evolutionary Computation*, volume 2, pages 1099–1103. IEEE Press, 1999.

[59] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In C. G. Langton, C. T. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Proceeding from Artificial Life II*, pages 313–324. Addison-Wesley, 1992.

[60] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.

[61] J. Horn, N. Nafpliotis, and D. E. Goldberg. A niched pareto genetic algorithm for multiobjective optimization. In *In Proceedings of the First IEEE Conference on Evolutionary Computation*, volume 1, pages 82–87, 1994.

[62] A. S. Jain and S. Meeran. Deterministic job-shop scheduling: Past, present and future. *European Journal of Operational Research*, 113:390–434, 1999.

[63] M. T. Jensen. Neighbourhood based robustness applied to tardiness and total flowtime job shops. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VI proceedings*, LNCS vol. 1917, pages 283–292. Springer, 2000.

[64] M. T. Jensen. Finding worst-case flexible schedules using coevolution. In *GECCO-2001 Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1144–1151, 2001.

[65] M. T. Jensen and T. K. Hansen. Robust solutions to job shop problems. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proceedings of the 1999 Congress on Evolutionary Computation*, volume 2, pages 1138–1144, Mayflower Hotel, Washington D.C., USA, 6-9 July 1999. IEEE Press.

[66] G. Jones, R. D. Brown, D. E. Clark, P. Willett, and R. C. Glen. Searching databases of two-dimensional and three-dimensional chemical structures using genetic algorithms. In Stephanie Forrest, editor, *Proceedings of the fifth International Conference on Genetic Algorithms*, pages 597–602, 1993.

[67] M. Khouja, Z. Michalewicz, and P. Vijayaragavan. Evolutionary algorithm for economic lot and delivery scheduling problem. *Fundamenta Informaticae*, 35:113–123, 1998.

[68] P. Kouvelis and G. Yu. *Robust Discrete Optimization and Its Applications*. Kluwer Academic Publishers, 1997.

[69] S. Lawrence. *Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (Supplement)*. Graduate School of Industrial Administration, Carnegie-Mellon University, 1984.

[70] V. J. Leon, S. D. Wu, and R. H. Storer. Robust scheduling and game-theoretic control for short-term scheduling of job-shops. In G. Fandel,

T. Gulledge, and A. Jones, editors, *Operations Research in Production Planning and Control*, pages 321–335. Sprinter-Verlag, 1993.

[71] V. J. Leon, S. D. Wu, and R. H. Storer. A game-theoretic control approach for job shops in the presence of disruptions. *International Journal of Production Research*, 32(6):1451–1476, 1994.

[72] V. J. Leon, S. D. Wu, and R. H. Storer. Robustness measures and robust scheduling for job shops. *IIE Transactions*, 26(5):32–43, September 1994.

[73] S. Lin, E. D. Goodman, and W. F. Punch. A genetic algorithm approach to dynamic job shop scheduling problems. In Thomas Bäck, editor, *Proceedings of the seventh International Conference on Genetic Algorithms*, pages 481–488. Morgan Kaufmann, 1997.

[74] S. Lin, E. D. Goodman, and W. F. Punch. Investigating parallel genetic algorithms on job shop scheduling problems. In *Proceedings of the sixth International Conference on Evolutionary Programming*, pages 383–394. Springer Verlag, 1997.

[75] S. W. Mahfoud. Niching methods. In T. Bäck, D. B. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, chapter C6.1. IOP Publishing and Oxford University Press, 1997.

[76] W. N. Martin, J. Lienig, and J. P. Cohoon. Island(migration) models: evolutionary algorithms based on punctuated equilibria. In T. Bäck, D. B. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, chapter C6.3. IOP Publishing and Oxford University Press, 1997.

[77] D. C. Mattfeld. *Evolutionary Search and the Job Shop*. Production and Logistics. Physica-Verlag, 1996.

[78] K. N. McKay, T. E. Morton, P. Ramnath, and J. Wang. 'aversion dynamics' scheduling when the system changes. *Journal of Scheduling*, 3:71–88, 2000.

[79] D. Merkle and M. Middendorf. Prospects for dynamic algorithm control: Lessons from the phase structure of ant scheduling algorithms. In *GECCO 2001 - Genetic and Evolutionary Computation Conference Workshop Program*, pages 121–126, 2001.

[80] Z. Michalewicz. *Genetic algorithms + Data structures = evolution programs*. Springer, 1999.

[81] M. Mitchell. *An introduction to genetic algorithms*. The MIT Press, 1996.

[82] D. E. Moriarty and R. Miikkulainen. Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation*, 5(4):373–399, 1997.

[83] T. E. Morton and D. W. Pentico. *Heuristic Scheduling Systems*. John Wiley & Sons, 1993.

[84] R. Nakano and T. Yamada. Conventional genetic algorithm for job shop problems. In *Proceedings of the fourth International Conference on Genetic Algorithms*, pages 474–479, 1991.

[85] B. Naudts, D. Suys, and A. Verschoren. Epistasis as a basic concecpt in formal landscape analysis. In T. Bäck, editor, *Proceedings of the seventh International Conference on Genetic Algorithms*, pages 65–72. Morgan Kaufmann Publishers, 1997.

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

[87] C. Oğuz, A. Janiak, and M. Lichtenstein. Metaheuristic algorithms for hybrid flow-shop scheduling problems with multiprocessor tasks. In *4th Metaheuristics International Conference - book of abstracts*, volume 2, pages 477–481, 2001.

[88] D. Pacciarelli and M. Pranzo. A tabu search algorithm for the railway scheduling problem. In *4th Metaheuristics International Conference - book of abstracts*, volume 1, pages 159–164, 2001.

[89] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[90] J. Paredis. Co-evolutionary constraint satisfaction. In Y. Davidor, H-P. Schwefel, and R. Männer, editors, *Proceedings of the Third Conference on Parallet Problem Solving from Nature*, volume 866 of *Lecture Notes in Computer Science*. Springer, 1994.

[91] J. Paredis. Steps towards co-evolutionary classification neural networks. In *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 102–108, 1994.

[92] J. Paredis. Coevolving cellular automata: Be aware of the red queen! In Thomas Bäck, editor, *Proceedings of the seventh International Conference on Genetic Algorithms*, pages 393–400. Morgan Kaufmann, 1997.

[93] C. C. Pettey. Diffusion(cellular) models. In T. Bäck, D. B. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, chapter C6.4. IOP Publishing and Oxford University Press, 1997.

[94] E. Polak. On the mathematical foundations of nondifferentiable optimization in engineering design. *SIAM review*, 29:21–89, 1987.

[95] J. B. Pollack, A. D. Blair, and M. Land. Coevolution of a backgammon player. In C. Langton and T. Shimohara, editors, *Proceedings of the Fifth Artificial Life Conference*. MIT Press, 1996.

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

[97] N. J. Radcliffe. Genetic set recombination and its application to neural network topology optimisation. *Neural Computing and Applications*, 1(1):67–90, 1993.

[98] N. Raman and F. B. Talbot. The job shop tardiness problem: A decomposition approach. *European Journal of Operational Research*, 69:187–199, 1993.

[99] C. R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. McGraw-Hill, 1995.

[100] C. W. Reynolds. Competition, coevolution and the game of tag. In R. Brooks and P. Maes, editors, *Proceedings of Artificial Life IV*, pages 59–69. MIT Press, 1994.

[101] I. Rixen, C. Bierwirth, and H. Kopfer. A case study of operational just-in-time scheduling using genetic algorithms. In *Evolutionary Algorithms in Management Applications*, pages 112–123. Springer, 1995.

[102] C. D. Rosin and R. K. Belew. Methods for competitive co-evolution: Finding opponents worth beating. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 373–380, 1995.

[103] G. Rudolph. Evolution strategies. In T. Bäck, D. B. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, chapter B1.3. IOP Publishing and Oxford University Press, 1997.

[104] H. P. Schwefel and G. Rudolph. Contemporary evolution strategies. In F. Morana et al., editor, *Advances in Artificial Life, Third European Conference on Artificial Life*, pages 893–907. Springer, Berlin, 1995.

[105] K. Sims. Evolving 3d morphology and behavior by competition. In R. Brooks and P. Maes, editors, *Proceedings of Artificial Life IV*, pages 28–39. MIT Press, 1994.

[106] Y. N. Sotskov, V. S. Tanaev, and F. Werner. Stability radius of an optimal schedules: A survey and recent developments. In G. Yu, editor, *Industrial Applications of Combinatorial Optimization*, pages 72–108. Kluwer Academic Publishers, 1998.

[107] R. H. Storer, S. D. Wu, and R. Vaccari. New search spaces for sequencing problems with applications to job shop scheduling. *Management Science*, 38(10):1495–1509, 1992.

[108] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64:278–285, 1993.

[109] Shigeyoshi Tsutsui and Ashish Ghosh. Genetic algorithms with a robust solution searching scheme. *IEEE Transactions on Evolutionary Computation*, 1(3):201–208, September 1997.

[110] R. K. Ursem. Multinational evolutionary algorithms. In P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proceedings of the 1999 Congress on Evolutionary Computation*, volume 3, pages 1633–1640, Mayflower Hotel, Washington D.C., USA, 1999. IEEE Press.

[111] M. Vazguez and D. Whitley. A comparison of genetic algorithms for the static job shop scheduling problem. In M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VI proceedings*, LNCS vol. 1917, pages 303–312. Springer, 2000.

[112] C. Ventouris. Gestaltung robuster maschinenbelegungspläne unter verwendung evolutionärer algorithmen. Master's thesis, Institut AIFB, Universität Karlsruhe, 1998.

[113] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesmen: The genetic edge recombination operator. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 133–140, 1989.

[114] S. D. Wu, E. Byeon, and R. H. Storer. A graph-theoretic decomposition of the job shop scheduling problem to achieve scheduling robustness. *Operations Research*, 47(1):113–124, January-February 1999.

[115] T. Yamada and R. Nakano. A genetic algorithm applicable to large-scale job-shop problems. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 281–290. Elsevier Science Publishers, 1992.

[116] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, November 1999.

Index

Appendix A

Proof of the minimax equivalence

In this appendix the equivalence of the equations

$$\exists x^* \in X, s^* \in S \ : \ F(x^*, s) \leq F(x^*, s^*) \leq F(x, s^*) \quad \forall x \in X, s \in S \quad \text{(A.1)}$$

and

$$\min_{x \in X} \max_{s \in S} F(x, s) = \max_{s \in S} \min_{x \in X} F(x, s) \quad \text{(A.2)}$$

will be proved.

Assume (A.1) prove (A.2), (A.1) $\Rightarrow$ (A.2).

Assume (A.1). Consider $\max_{s \in S} F(x_1, s)$. We must have

$$\max_{s \in S} F(x_1, s) \geq F(x_1, s^*) \geq F(x^*, s^*).$$

Since this holds for all $x_1 \in X$, and since (A.1) tells us that $\max_{s \in S} F(x^*, s) = F(x^*, s^*)$, we must have

$$\min_{x \in X} \max_{s \in S} F(x, s) = F(x^*, s^*).$$

In the same way, it can be shown that

$$\max_{s \in S} \min_{x \in X} F(x, s) = F(x^*, s^*).$$

Assume (A.2) prove (A.1), (A.2) $\Rightarrow$ (A.1)

Assume (A.2). This means that the minimax and maximin problems are solvable, hence we can assume without loss of generality that

- (x_1, s_1) is the solution to $\min_{x \in X} \max_{s \in S} F(x, s)$.

- (x_2, s_2) is the solution to $\max_{s \in S} \min_{x \in X} F(x, s)$.

Since (x_1, s_1) solves $\min_{x \in X} \max_{s \in S} F(x, s)$, we must have

$$\forall s \in S : F(x_1, s) \leq F(x_1, s_1). \tag{A.3}$$

Since (x_2, s_2) solves $\max_{s \in S} \min_{x \in X} F(x, s)$, we must have

$$\forall x \in X : F(x, s_2) \geq F(x_2, s_2). \tag{A.4}$$

Because we are assuming (A.2), we must have $F(x_1, s_1) = F(x_2, s_2)$, which with equations (A.3) and (A.4) leads to

$$F(x_1, s_1) = F(x_2, s_2) = F(x_1, s_2). \tag{A.5}$$

Coupling this with (A.3) and (A.4), we see that (x_1, s_2) also solves both $\max_{s \in S} \min_{x \in X} F(x, s)$ and $\min_{x \in X} \max_{s \in S} F(x, s)$, and that (A.1) must hold for (x_1, s_2).

Appendix B

Barbosa's minimax algorithm and experiments

The coevolutionary minimax algorithm used by Barbosa in [5, 6, 7] is the following:

---

create initial populations $P_X(0)$ and $P_S(0)$
for each individual $x \in P_X$ set $h[x] = \max_{s \in P_S} F(x, s)$
for each individual $s \in P_S$ set $g[s] = \min_{x \in P_X} F(x, s)$
for $k = 1, 2, \ldots, max\_cycles$ do
  for $i = 1, 2, \ldots, max\_gen\_X$ do
    generate P'_X from P_X based on fitness $h[x]$
    set $P_X = P'_X$
    for each individual $x \in P_X$ set $h[x] = \max_{s \in P_S} F(x, s)$
  od
  for $i = 1, 2, \ldots, max\_gen\_S$ do
    generate P'_S from P_S based on fitness $g[s]$
    set $P_S = P'_S$
    for each individual $s \in P_S$ set $g[s] = \min_{x \in P_X} F(x, s)$
  od
od
return $x_0 \in P_X$ with minimal $h[x_0]$
  and $s_0 \in P_S$ with maximal $g[s_0]$

For the experiments the algorithm was implemented with the same details as the algorithms presented in section 2.4.3. The most notable difference from Barbosa's implementation is the use of a real-valued encoding; Barbosa used a gray encoding (thus, the genetic operators are also different). However, this difference should not be expected to change the behaviour of the algorithm in a profound way.

The algorithm has been tested on the problems of section 2.4.3 with different settings for max_cycles, max_gen_X and max_gen_S. The parameters were always so that the total number of individuals tested was equal to the number of individuals tested in the experiments of section 2.4.3. This means that in the experiments the Barbosa algorithm used twice as many calls to the objective function $F(x, s)$ as the Herrmann and the asymmetric algorithms. In the following the notation $max\_cycles \times (max\_gen\_X + max\_gen\_S)$ is used to describe the parameter settings of each run. Averages are over 1000 runs. The results are in the following tables, along with the average errors of Herrmann's algorithm and the asymmetric evaluation algorithm, which have been included to ease comparison.

The saddlepoint function		
Parameters	MSE(x)	MSE(s)
$20 \times (2 + 8)$	68.2416 E-12	0.4696 E-12
$20 \times (3 + 7)$	16.9736 E-12	0.6477 E-12
$20 \times (4 + 6)$	4.8903 E-12	1.3026 E-12
$20 \times (5 + 5)$	2.4300 E-12	2.2293 E-12
$50 \times (2 + 2)$	1.9158 E-12	2.1266 E-12
$20 \times (6 + 4)$	1.2138 E-12	4.9054 E-12
$20 \times (7 + 3)$	0.8869 E-12	17.4306 E-12
$20 \times (8 + 2)$	0.5024 E-12	102.2954 E-12
Herrmann	2.1545 E-12	2.0414 E-12
Asymmetric	2.0540 E-12	2.0262 E-12

The Twoplanes function		
Parameters	MSE(x)	MSE(s)
$20 \times (2 + 8)$	0.1420	3.8116
$20 \times (3 + 7)$	0.2035	2.0667
$20 \times (4 + 6)$	0.1059	1.9008
$20 \times (5 + 5)$	0.0103	2.2645
$50 \times (2 + 2)$	0.2091	11.5237
$20 \times (6 + 4)$	0.0144	1.7771
$20 \times (7 + 3)$	0.0374	2.5138
$20 \times (8 + 2)$	0.3593	3.4261
Herrmann	0.0000	10.9874
Asymmetric	0.0037	0.0170

The damped sinus function		
Parameters	MSE(x)	MSE(s)
$20 \times (2 + 8)$	0.8725	32.4230
$20 \times (3 + 7)$	1.7462	23.1327
$20 \times (4 + 6)$	1.6672	20.5224
$20 \times (5 + 5)$	2.1443	17.6714
$50 \times (2 + 2)$	2.5375	34.9545
$20 \times (6 + 4)$	2.3374	15.8595
$20 \times (7 + 3)$	3.6256	16.5533
$20 \times (8 + 2)$	5.0788	18.5681
Herrmann	6.2345	37.7239
Asymmetric	0.3506	0.7884

The damped cosine function		
Parameters	MSE(x)	MSE(s)
$20 \times (2 + 8)$	1.7520	0.1107
$20 \times (3 + 7)$	3.0075	0.2942
$20 \times (4 + 6)$	2.2827	0.8826
$20 \times (5 + 5)$	1.9143	2.0342
$50 \times (2 + 2)$	4.8721	0.4976
$20 \times (6 + 4)$	2.1082	2.9088
$20 \times (7 + 3)$	5.3575	4.8862
$20 \times (8 + 2)$	5.3842	4.7691
Herrmann	14.3860	0.1592
Asymmetric	0.0371	0.2116

From the experiments on the simple saddlepoint function it is evident that in some cases the precision of the return values x and s can be controlled by max_gen_X and max_gen_S; for this function it is clear that the error on x (s) decreases as max_gen_X (max_gen_S) increases. However, for the other problems this does not seem to be the case. This is probably because in most cases the Barbosa algorithm fails to converge to the correct solution.

Comparing the performance of the Barbosa algorithm to the other algorithms it is evident that in many cases the the Barbosa algorithm obtains errors significantly lower than Herrmann's algorithm. This indicates that in some cases the algorithm performance is increased by keeping one population frozen while the other population evolves.

Comparing the Barbosa algorithm to the asymmetric evaluation algorithm it is clear that the asymmetric evaluation algorithm outperforms the Barbosa algorithm by far for all problem but the naive saddlepoint function, in which no performance difference was expected.

Appendix C

NP-completeness of rescheduling problems

In this appendix it will be proven that finding the makespan optimal solution to a breakdown rescheduling problem P_{resch} generated from the original problem P is NP-hard, even if the makespan optimal solution to P is known, if the number of machines m is greater than or equal to 3.

It is outlined that the proof can easily be extended to cover maximum tardiness or summed tardiness problems, and that it also holds for rescheduling problems generated by appearing jobs.

C.1 NP-completeness of the makespan rescheduling problem

The proof is by reduction of the 0-1 knapsack problem (which is known to be NP-complete, [89]) to a rescheduling problem. The 0-1 knapsack problem is the following: Given a set of natural numbers $A = \{a_1, a_2, \ldots, a_k\}$ and a natural number c determine whether there is a subset $Z \subseteq A$ such that $c = \sum_{a_i \in Z} a_i$. If this is the case, the 0-1 knapsack problems is said to be solvable.

To make the reduction, we first construct an ordinary makespan job shop problem based on the knapsack problem:

- The problem has three machines, M_1, M_2 and M_3.
- The problem has $k + 1$ jobs, $J_1, J_2, \ldots, J_k, J_{k+1}$, where k is the size of set A.
- Each of the first k jobs consists of one operation to be processed on machine M_1, with processing time $\tau_{i1} = a_i$.

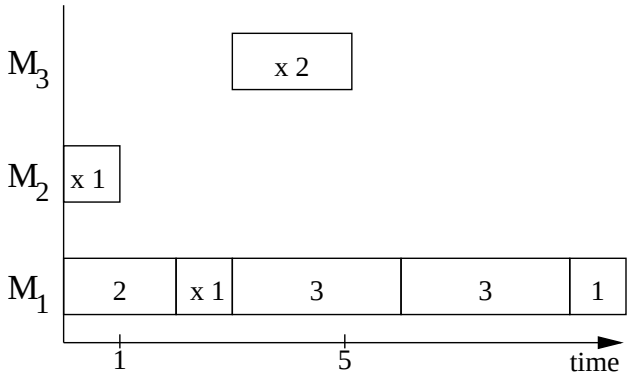

Figure C.1: *Gantt chart of example optimal schedule for a constructed makespan problem. The problem corresponds to the 0-1 knapsack problem* $A = \{1, 2, 3, 3\}, c = 7$*. The numbers represent processing times of the operations. Operations belonging to job* J_{k+1} *have been marked with an* x*.*

- Job J_{k+1} consists of three operations, the first $o_{k+1,1}$ to be processed on M_2 with processing time $\tau_{k+1,1} = 1$, the second to be processed on M_1 with processing time $\tau_{k+1,2} = 1$, the third to be processed on machine M_3 with processing time $\tau_{k+1,3} = \sum_{a_i \in A} a_i - c$.

An optimal schedule to this problem can be easily constructed by processing the first operation of job J_{k+1} on machine M_2 at time 0, processing any operation of jobs $J_1 - J_k$ with processing time less than $c - 1$ at M_1 at time 0 (we assume without loss of generality that such an operation exists. If this is not the case the 0-1 knapsack problem is trivial), processing $o_{k+1,2}$ of job J_{k+1} and the operations of the remaining jobs $J_1 - J_k$ after this operation has finished. Operation $o_{k+1,3}$ is processed on M_3 when $o_{i+1,2}$ has finished. This gives a makespan of $\sum_{a_i \in A} a_i + 1$, which is clearly optimal, since there is no idle time on M_1. A schedule of this kind is illustrated on figure C.1.

A rescheduling problem to solve the 0-1 knapsack problem can now be constructed by making a breakdown of machine M_2 at time $t = 0$. The rescheduling problem is constructed in such a way that it has a solution with makespan $\sum_{a_i \in A} a_i + 1$ if and only if the 0-1 knapsack problem can be solved. The breakdown downtime is chosen to be $\tau_{breakdown} = c - 1$. Note that the rescheduling problem can be constructed using both preemptive and non-preemptive breakdowns, since no operation is being processed at the time of the breakdown. Because of the construction of the problem the rescheduling problem can still be solved to makespan $\sum_{a_i \in A} a_i + 1$ if and only if the 0-1 knapsack problem can be solved.

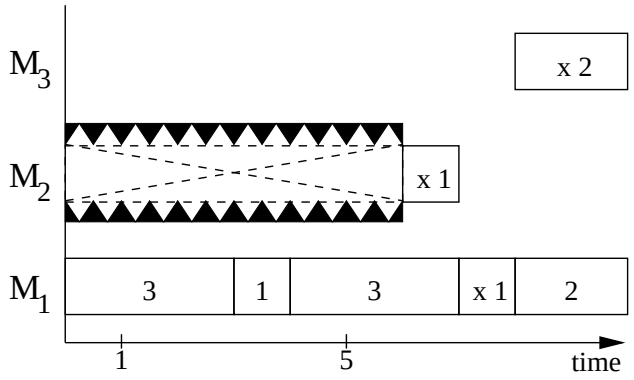

Figure C.2: *Gantt chart of example optimal schedule for the makespan rescheduling problem corresponding to the problem of figure C.1. The numbers represent processing times of the operations. Operations belonging to job J_{k+1} have been marked with an x. The black triangles and the dashed box box of M_2 represent the breakdown. Since the makespan of this schedule is the same as the makespan of the schedule on figure C.1, the 0-1 knapsack problem is solvable.*

Suppose the knapsack problem can be solved. This means that a set $Z \subseteq A$ exists, such that $\sum_{a_i \in Z} = c$. This means that a set of operations can be found to be processed on M_1 such that there is no idle time prior to the processing of $o_{k+1,2}$, which is processed on M_1 at time c. After this all the other operations of jobs $J_1 - J_k$ are processed on machine M_1, and processing of operation $o_{k+1,3}$ of job J_{k+1} is started on machine M_3 at time $c + 1$. Machines M_1 and M_3 complete at time $\sum_{a_i \in A} a_i + 1$, while M_2 completes somewhat earlier. The makespan of the schedule is $\sum_{a_i \in A} a_i + 1$. A schedule of this kind is illustrated on figure C.2.

Suppose there is no solution to the 0-1 knapsack problem. This means no subset Z can be found such that $\sum_{a_i \in Z} a_i = c$. This means it is impossible to avoid idle time on M_1 if operation $o_{k+1,2}$ is to be processed on M_1 at time c. If M_1 is idle for any length of time the makespan of the schedule will be more than $\sum_{a_i \in A} a_i + 1$. If idle time is avoided by starting $o_{k+1,2}$ later than time c processing of $o_{k+1,3}$ on machine M_3 will complete later than $\sum_{a_i \in A} a_i + 1$, and again the makespan will be higher than $\sum_{a_i \in A} a_i + 1$.

C.2 Complexity of other rescheduling problems

The makespan rescheduling problem can easily be reduced to a worst tardiness rescheduling problem. The problem constructed in the previous section is augmented with due dates of $D = \sum_{a_i \in A} a_i + 1$ for all jobs. The preschedule con-

structed above is not tardy, and thus optimal. The rescheduling problem can be solved without tardiness if and only if the 0-1 knapsack problem can be solved, since a non-tardy schedule is equivalent to a schedule of makespan $\sum_{a_i \in A} a_i + 1$. Note also that when tardiness problems are considered, the result can be extended to cover problems with two machines, since machine 3 and the third operation of J_{k+1} are not needed in the NP-completeness proof for a tardiness problem if the due date of job J_{k+1} is set to $c + 2$. The same argument can be used for summed tardiness and number of tardy jobs rescheduling problems. Thus, all these different kinds of tardiness rescheduling problems are NP-complete, even if the optimal schedule is known prior to the breakdown.

The result can also be extended to cover rescheduling problems with new jobs. Instead of a breakdown, at new job J_{k+2} can be added. The job should have two operations: $o_{k+2,1}$ to be processed on M_2 with $\tau_{k+2,1} = c - 1$ and $o_{k+2,2}$ to be processed on M_3 with $\tau_{k+2,2} = 2$. Since $o_{k+2,1}$ will play the part of the breakdown in the previous proof, it is easy to realize that the rescheduling problem can only be solved to makespan $\sum_{a_i \in A} a_i + 1$ (or without tardiness) if the 0-1 knapsack problem is solvable.

Appendix D

Proof that the job shop problem facing breakdowns does not satisfy the symmetric property

In this appendix it is demonstrated that the job shop scheduling problem facing preemptive breakdowns does not necessarily satisfy the symmetric property (2.6). That is

$$\min_{s\in S}\max_{b\in B} C(s,b) = \max_{b\in B}\min_{s\in S} C(s,b)$$

does not necessarily hold. The proof is by counterexample.

Consider a three machines, two jobs job shop scheduling problem under the makespan criterion. The details of the problem can be deducted from the schedules of figure D.1. The scenario set B is defined to hold the two scenarios

b_1: Machine 1 breaks at time 5 for a duration of 4 time-units.

b_2: Machine 3 breaks at time 5 for a duration of 4 time-units.

For schedules facing these two breakdowns, the optimal schedules using right-shifting rescheduling (as well as any other reasonable kind of rescheduling, since right-shifting provides optimal new schedules for b_1 and b_2) are the schedules on figure D.1. Schedule A is optimal for b_2 (makespan 11), while schedule B is optimal for b_1 (makespan 10). The makespans for the different combinations of schedules and breakdowns are listed in the table:

$C_{max}(s,b)$	b_1	b_2	$\max_{b\in B} C_{max}(s,b)$
$s =$schedule A	12	11	12
$s =$schedule B	10	13	13
$\min_{s\in S} C_{max}(x,s)$	10	11	

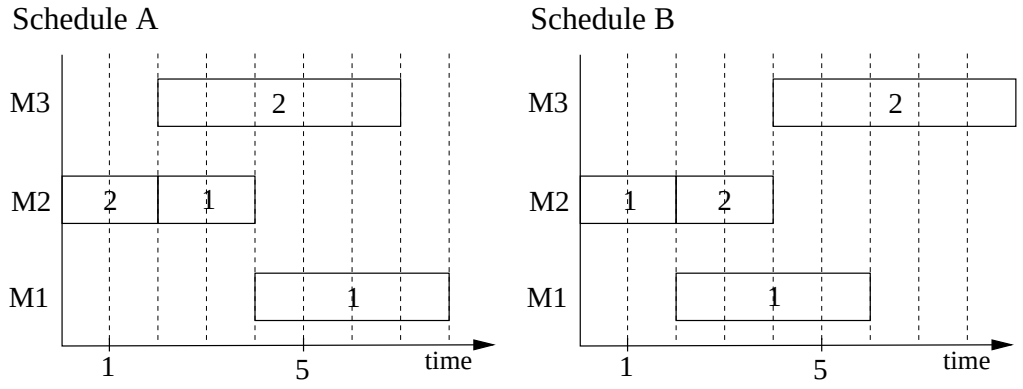

Figure D.1: ***Left**: Schedule A, optimal for scenario b_2 (a breakdown on machine M3). **Right**: Schedule B, optimal for scenario b_1 (a breakdown on machine M1).*

The table is equivalent to the table in section 2.4.1 except for an additive constant. As can be observed from the table, $\min_{s \in S} \max_{b \in B} C_{max}(s, b) = 12$, while $\max_{b \in B} \min_{s \in S} C_{max}(s, b) = 11$, so in this example clearly equation (2.6) does not hold.

Appendix E

Calculating errors bars on observed distribution functions

The calculation of error bars on the plots of chapter 5 is used to ensure the statistical significance of some of the results of that chapter. The error bars are calculated in the following way.

Consider an experiment that gives a random number as a result. This experiment is repeated N times, each experiment being independent, and the results are recorded. The number of observations n_x being less or equal to some number x is binomially distributed, call the probability parameter p_x, so $n_x \sim b(N, p_x)$. Clearly, p_x is equal to the x-percentile of the distribution function of the original experiment. Thus, if we need to calculate a confidence interval on the x-percentile, all we need to do is calculate a confidence interval on p_x.

Calculating a confidence interval $[p_{x,min}; p_{x,max}]$ on the probability parameter of a binomial distribution is straightforward. According to [19] a good approximation of the confidence interval is:

$$p_{x,min} = \frac{1}{n + u^2_{1-\alpha/2}} \left[(n_x - \tfrac{1}{2}) + \frac{u^2_{1-\alpha/2}}{2} - u_{1-\alpha/2} \sqrt{\frac{(n_x - \frac{1}{2})(N - n_x + \frac{1}{2})}{N} + \frac{u^2_{1-\alpha/2}}{4}} \right],$$

$$p_{x,max} = \frac{1}{n + u^2_{1-\alpha/2}} \left[(n_x - \tfrac{1}{2}) + \frac{u^2_{1-\alpha/2}}{2} + u_{1-\alpha/2} \sqrt{\frac{(n_x - \frac{1}{2})(N - n_x + \frac{1}{2})}{N} + \frac{u^2_{1-\alpha/2}}{4}} \right],$$

where $u_{1-\alpha/2}$ denotes the $(1-\alpha/2)$-percentile of the standard Gaussian distribution and α is the confidence level of the interval. The approximation is acceptable if $N\,p_{x,min}(1-p_{x,min}) > 9$ and $N\,p_{x,max}(1-p_{x,max}) > 9$. This is the case for all the error bars calculated in chapter 5.

Appendix F

The maximum lateness hillclimbers

The L_{max}- and $R_{L_{max}}$-hillclimbers (in the following called *the lateness hillclimbers*) work in almost the same way as the C_{max} and $R_{C_{max}}$-hillclimbers (*the makespan hillclimbers*). As a first step a schedule is produced from the gene using semi-active decoding. This semi-active schedule is then improved by a hillclimber using a neighbourhood based on the same ideas as the neighbourhood used by the makespan hillclimbers.

In order to properly describe the hillclimbers, we need to introduce some notation. Denote the last operation of job J_j as E_j, the maximum lateness and maximum tardiness of a schedule can be calculated

$$L_{max} = \max_{j=1...n}(h(E_j) + \tau_{E_j} - d_j) \tag{F.1}$$

$$T_{max} = \max(L_{max}, 0).$$

In this way, the L_{max} (or T_{max}) job shop problem can be formulated as a graph problem in which the task is to choose the Hamiltonian paths (the machine processing sequences) such that (F.1) is minimised.

In order to estimate the effect of hillclimbing moves on schedule performance, the *lateness tail* $l(o)$ of an operation in a semiactive schedule s will be defined. $l(o)$ satisfies

$$L_{max}(s') \geq l(o) + \sigma$$

where s' is the schedule which has the same processing orders as s, and which is semiactive except that $h(o)$ has been increased by σ (the beginning of processing of o is delayed by σ). For σ large enough, $L_{max}(s') = l(o) + \sigma$ holds. The lateness tail will later be used to estimate the effect of hillclimbing moves in the schedule. When an operation o is delayed ($h(o)$ is increased), it can affect the schedule performance in these ways:

1. Operation o is the last operation of a job J_j (i.e. $o = E_j$). In this case we define
$$l_J(o) = h(o) - \tau_o - d_j,$$
where τ_o and d_j are the processing time of o and the due date of J_j.

2. It is not the last operation of a job. The delay of o may delay the processing of its job successor $SJ(o)$. In this case define
$$l_J(o) = l(SJ(o)) + h(SJ(o)) - h(o) - \tau_o.$$

3. It has a machine successor $SM(o)$. The delay of o may cause $SM(o)$ to be delayed. Define
$$l_S(o) = l(SM(o)) + h(SM(o)) - h(o) - \tau_o.$$

4. If the machine successor $SM(o)$ is undefined, $l_S(o)$ is set to $-\infty$.

The lateness tail is defined to be
$$l(o) = \max(l_J(o), l_S(o)).$$

As for makespan schedules, an operation is termed *critical* in a schedule if it cannot be delayed without worsening schedule performance. Operation o is critical if $l(o) = L_{max}$. The set of critical operations in a schedule is called *the critical path*. A number of consecutive critical operations on the same machine are called *a critical block*.

Based on these critical blocks, the neighbourhood described in section 3.2.2 can be used on maximum lateness problems. Using the lateness tails, the estimates on makespan after a hillclimbing move can easily be changed to cover maximum lateness. The effect of a "small block" move (figure 3.11) is estimated as follows. The heads of o_1 and o_2 after the move can be calculated
$$h'(o_2) = \max(h(PM(o_1)) + \tau_{PM(o_1)}, h(PJ(o_2)) + \tau_{PJ(o_2)}),$$
$$h'(o_1) = \max(h'(o_2) + \tau_{o_2}, h(PJ(o_1)) + \tau_{PJ(o_1)}).$$
The lateness tail of o_1 and o_2 after the move can be calculated
$$l'(o_1) = \max(l_J(o_1), l(SM(o_2)) + h(SM(o_2)) - h'(o_1) - \tau_{o_1}),$$
$$l'(o_2) = \max(l_J(o_2), l'(o_1) + h'(o_1) - h'(o_2) - \tau_{o_2}).$$
The maximum lateness of the schedule after the move $L_{max}(s')$ is bounded by
$$L_{max}(s') \geq L_{max,bound}(s') = \max(l'(o_1), l'(o_2))$$

If o_1 or o_2 is still critical after the move, $L_{max}(s') = \max(l'(o_1), l'(o_2))$ will be the case. Bounds on schedule performance for the other moves can be made in similar ways.

It is easy to realize that the properties guaranteeing schedule feasibility after moves made by the makespan hillclimber must also hold for a hillclimber using the same principle on a maximum lateness problem. In order to realize this, consider that for every critical block in a maximum lateness schedule an equivalent makespan schedule and problem can be constructed holding the same moves, with the estimates on makespan after the move being equal to the estimates on maximum lateness in the lateness problem. Since the chosen move is guaranteed to be feasible in the makespan schedule, it must be feasible in the maximum lateness schedule also.

The construction of the $R_{C_{max}}$-hillclimber was made based on the C_{max}-hillclimber, and the same approach can be followed for constructing a $R_{L_{max}}$-hillclimber based on the L_{max}-hillclimber. In the same way the neighbourhood used in the C_{max}-hillclimber is also used in the $R_{C_{max}}$-hillclimber, the neighbourhood used in the L_{max}-hillclimber is also used in the $R_{L_{max}}$- hillclimber. The estimate on $R_{L_{max}}$ after a move is made using the small block move in the same way described for the $R_{C_{max}}$- hillclimber in section 5.3.2.

Appendix G

Results of makespan experiments

G.1 Makespans

Average makespans found in the experiments on neighbourhood based robustness and flexibility of section 5.3.12. There is a subtable for each problem instance, and four rows in each subtable representing four different scheduling algorithms. The row labelled "P" represents the preschedule makespan without any breakdowns, while the columns labelled 1-5 represent the different rescheduling methods after a breakdown, see section 5.3.5. The averages were calculated after 400 runs of each algorithm. The breakdown duration was 80.

The best (lowest) makespan achieved for each problem and rescheduling method has been printed bold.

ft10						
method	P	1	2	3	4	5
active	**948.2**	1010.1	1004.2	1000.9	989.9	987.9
slack	949.1	**1005.8**	**1000.4**	**998.2**	**989.7**	**987.3**
fastrob	950.4	1008.7	1003.8	1003.1	993.1	990.8
robust	949.6	1007.8	1002.0	1000.3	991.3	989.8

ft20						
method	P	1	2	3	4	5
active	1196.6	1265.5	1261.2	1256.0	1234.8	1232.1
slack	1196.2	1251.3	1248.6	1246.1	1235.0	1233.3
fastrob	**1189.7**	**1248.6**	**1243.8**	**1240.2**	**1225.7**	**1224.1**
robust	1189.8	1249.6	1244.8	1240.8	1227.6	1225.8

la01						
method	P	1	2	3	4	5
active	**666.0**	720.6	714.0	710.6	705.1	704.0
slack	666.1	**702.5**	**700.2**	**699.0**	**696.8**	**695.7**
fastrob	**666.0**	707.6	701.0	700.0	699.9	698.5
robust	**666.0**	706.0	700.8	700.3	699.7	698.6

la02						
method	P	1	2	3	4	5
active	**655.1**	717.7	712.5	710.6	**700.8**	**698.7**
slack	660.9	**712.9**	709.8	708.3	703.0	701.1
fastrob	655.9	713.9	**708.6**	**708.1**	701.2	699.0
robust	655.7	713.2	709.8	709.1	704.0	701.5

la03						
method	P	1	2	3	4	5
active	**597.1**	664.1	661.1	658.9	654.9	653.8
slack	609.1	661.2	659.8	659.0	656.4	655.2
fastrob	598.3	661.5	659.3	658.1	655.6	654.9
robust	598.3	**658.2**	**656.0**	**654.8**	**652.0**	**651.5**

la04						
method	P	1	2	3	4	5
active	**590.8**	650.3	648.1	644.5	639.7	638.0
slack	595.0	**645.2**	643.5	642.4	639.4	638.6
fastrob	594.2	645.7	**642.5**	**641.0**	**638.5**	**637.1**
robust	593.7	649.8	647.0	646.0	642.8	641.1

la05						
method	P	1	2	3	4	5
active	**593.0**	635.8	628.1	625.1	622.7	622.5
slack	**593.0**	616.5	615.3	615.2	615.1	615.1
fastrob	**593.0**	617.3	615.8	615.5	615.5	615.5
robust	**593.0**	**615.5**	**614.3**	**614.2**	**614.2**	**614.2**

la06						
method	P	1	2	3	4	5
active	**926.0**	967.2	959.7	954.1	950.3	949.9
slack	**926.0**	943.8	943.8	943.8	943.8	943.8
fastrob	**926.0**	**940.6**	**939.9**	**940.0**	**939.8**	**939.8**
robust	**926.0**	944.3	943.6	943.5	943.4	943.4

la07						
method	P	1	2	3	4	5
active	**890.0**	944.5	937.5	932.7	928.3	927.6
slack	**890.0**	919.0	917.6	917.0	916.0	915.8
fastrob	**890.0**	**918.2**	**915.5**	**914.7**	**913.5**	**913.2**
robust	**890.0**	922.0	919.6	918.7	917.7	917.1

la08						
method	P	1	2	3	4	5
active	**863.0**	920.5	915.4	910.0	898.9	898.1
slack	**863.0**	**886.0**	**884.9**	**883.8**	**882.3**	**882.2**
fastrob	**863.0**	890.4	887.6	886.5	884.5	884.2
robust	**863.0**	890.1	887.2	885.7	883.6	883.4

la09						
method	P	1	2	3	4	5
active	**951.0**	999.3	993.6	990.2	989.0	989.0
slack	**951.0**	**982.9**	**982.1**	**981.7**	**981.7**	**981.6**
fastrob	**951.0**	984.4	983.8	983.8	983.7	983.7
robust	**951.0**	983.0	982.2	982.2	982.2	982.2

la10						
method	P	1	2	3	4	5
active	**958.0**	995.0	987.8	984.0	982.5	982.4
slack	**958.0**	975.2	975.0	975.0	975.0	975.1
fastrob	**958.0**	**973.7**	**973.3**	**973.2**	**973.2**	**973.2**
robust	**958.0**	975.2	974.9	974.8	974.8	974.8

la11						
method	P	1	2	3	4	5
active	**1222.0**	1263.6	1257.4	1251.5	1249.2	1249.1
slack	**1222.0**	**1238.7**	**1238.6**	**1238.6**	**1238.6**	**1238.6**
fastrob	**1222.0**	1241.5	1240.3	1240.2	1240.2	1240.2
robust	**1222.0**	1244.6	1243.3	1242.9	1242.9	1242.9

la12						
method	P	1	2	3	4	5
active	**1039.0**	1086.3	1080.9	1076.6	1074.7	1074.4
slack	**1039.0**	1070.4	1070.0	1069.9	1069.8	1069.8
fastrob	**1039.0**	**1070.1**	**1069.5**	**1069.4**	**1069.4**	**1069.4**
robust	**1039.0**	1070.4	1069.7	1069.6	1069.6	1069.6

la13						
method	P	1	2	3	4	5
active	**1150.0**	1199.8	1193.2	1187.4	1183.4	1183.2
slack	**1150.0**	**1176.9**	**1176.3**	1176.1	1176.0	1176.0
fastrob	**1150.0**	1178.8	1177.7	1177.2	1177.0	1177.0
robust	**1150.0**	1178.3	1176.4	**1175.6**	**1175.5**	**1175.5**

la14						
method	P	1	2	3	4	5
active	**1292.0**	1317.2	1311.9	1308.6	1307.6	1307.6
slack	**1292.0**	1308.5	1308.4	1308.4	1308.4	1308.4
fastrob	**1292.0**	**1305.4**	**1305.1**	**1305.1**	**1305.1**	**1305.1**
robust	**1292.0**	1311.8	1311.4	1311.3	1311.4	1311.3

la15						
method	P	1	2	3	4	5
active	**1207.0**	1262.9	1257.9	1253.2	1246.5	1245.5
slack	**1207.0**	1237.7	1236.3	1235.9	1234.3	1233.9
fastrob	**1207.0**	**1237.4**	**1235.1**	**1234.0**	**1232.9**	**1232.8**
robust	**1207.0**	1238.7	1236.9	1236.1	1234.8	1234.8

la16						
method	P	1	2	3	4	5
active	**949.6**	998.8	993.2	991.0	**983.6**	**982.3**
slack	966.1	1000.3	997.7	996.8	996.6	995.6
fastrob	958.6	998.9	993.4	**990.8**	987.9	986.4
robust	957.5	**995.0**	**992.6**	992.1	989.6	989.0

la17						
method	P	1	2	3	4	5
active	**785.2**	839.5	829.1	825.5	**816.7**	**814.3**
slack	793.1	**828.7**	**824.0**	**822.2**	818.8	817.9
fastrob	791.1	830.8	826.2	824.5	817.9	817.0
robust	793.7	833.9	828.9	827.0	821.8	820.4

la18						
method	P	1	2	3	4	5
active	**849.4**	908.8	901.7	899.2	**886.2**	**882.4**
slack	861.2	**902.2**	**898.7**	**897.7**	890.7	889.2
fastrob	860.1	905.2	900.8	899.5	889.3	887.4
robust	859.8	904.0	899.6	898.4	888.8	886.3

la19						
method	P	1	2	3	4	5
active	**846.4**	909.2	903.0	899.8	**889.1**	**885.4**
slack	852.6	**903.2**	**898.3**	**896.4**	890.1	888.2
fastrob	850.4	908.4	902.5	899.6	893.9	890.3
robust	852.5	906.4	901.0	899.1	892.9	889.1

la20						
method	P	1	2	3	4	5
active	**906.9**	957.9	953.1	950.8	**939.4**	**937.0**
slack	911.2	**953.6**	**949.5**	**947.7**	939.9	940.9
fastrob	907.1	953.7	950.9	950.2	942.1	940.2
robust	907.1	957.8	954.7	954.2	944.1	942.2

la21						
method	P	1	2	3	4	5
active	1070.0	1128.2	1121.5	1117.2	1102.1	1097.8
slack	1071.4	1121.2	1116.0	1112.4	1104.7	1100.4
fastrob	**1068.3**	**1121.0**	1115.2	**1111.8**	**1100.9**	**1097.4**
robust	1069.7	1121.1	**1115.0**	1112.2	1101.0	1097.7

la22						
method	P	1	2	3	4	5
active	**939.4**	997.1	990.4	985.9	974.3	**971.3**
slack	946.5	990.7	985.9	984.1	976.7	974.7
fastrob	943.5	991.6	986.1	983.0	974.3	972.2
robust	942.8	**990.5**	**985.2**	**982.4**	**973.8**	971.5

la23						
method	P	1	2	3	4	5
active	**1032.0**	1083.9	1076.8	1071.2	1058.9	1056.0
slack	1032.6	1063.2	1058.6	1056.7	1052.6	1051.5
fastrob	1032.2	1063.2	**1057.7**	**1056.1**	1051.6	1050.2
robust	1032.2	**1062.8**	1058.2	1056.6	**1050.8**	**1049.7**

la24						
method	P	1	2	3	4	5
active	947.1	1006.8	1001.2	998.0	**986.6**	**983.9**
slack	948.0	**1001.7**	**998.0**	**996.1**	990.0	987.8
fastrob	**946.8**	1003.9	1000.3	998.7	989.1	986.3
robust	947.2	1005.1	1000.7	998.8	989.8	987.6

la25						
method	P	1	2	3	4	5
active	991.8	1048.1	1042.6	1039.0	**1027.3**	**1024.6**
slack	993.2	1043.5	1040.2	1038.6	1031.7	1029.7
fastrob	**991.4**	**1043.1**	**1038.0**	**1035.5**	1027.5	1025.1
robust	991.5	1043.8	1039.6	1037.9	1030.9	1029.3

la26						
method	P	1	2	3	4	5
active	1221.4	1281.2	1275.9	1272.2	1262.4	1259.3
slack	1222.5	1271.9	1268.6	1266.1	1258.3	1255.8
fastrob	1219.2	1269.8	1265.1	1262.2	1254.8	1252.5
robust	**1219.1**	**1268.8**	**1264.4**	**1261.9**	**1254.2**	**1251.5**

la27						
method	P	1	2	3	4	5
active	1276.5	1333.1	1326.7	1321.8	1304.7	1301.5
slack	1279.7	1326.1	1322.2	1318.7	1305.8	1304.1
fastrob	1272.8	**1322.1**	**1316.6**	**1313.5**	**1300.4**	**1298.1**
robust	**1272.0**	1324.1	1319.1	1315.7	1301.2	1299.0

la28						
method	P	1	2	3	4	5
active	1243.1	1299.0	1292.9	1287.3	1274.2	1270.8
slack	1245.8	1284.9	1281.0	1278.5	1269.4	1267.4
fastrob	1240.2	**1283.5**	**1278.9**	**1275.3**	**1267.4**	**1264.9**
robust	**1240.1**	1284.4	1279.2	1276.4	**1267.4**	1265.3

la29						
method	P	1	2	3	4	5
active	1229.3	1283.1	1276.2	1270.2	1254.8	1250.9
slack	1234.0	1276.2	1272.3	1270.3	1260.3	1258.0
fastrob	**1220.9**	**1269.5**	**1264.4**	**1260.8**	**1248.2**	**1245.2**
robust	1221.9	1270.8	1265.0	1261.0	1249.7	1246.6

la30						
method	P	1	2	3	4	5
active	1356.7	1410.1	1402.6	1398.1	1386.7	1384.2
slack	1359.2	1397.9	1395.1	1393.0	1386.1	1384.4
fastrob	**1355.4**	**1396.2**	**1390.9**	**1387.4**	**1379.2**	**1377.4**
robust	1355.5	1396.6	1393.2	1391.1	1382.1	1380.4

la31						
method	P	1	2	3	4	5
active	**1784.0**	1825.5	1816.6	1808.4	1802.4	1802.2
slack	**1784.0**	**1795.7**	**1793.8**	**1793.0**	**1792.6**	**1792.5**
fastrob	**1784.0**	1797.1	1795.3	1794.3	1794.1	1794.0
robust	**1784.0**	1800.0	1798.1	1797.3	1796.8	1796.8

la32						
method	P	1	2	3	4	5
active	**1850.0**	1895.8	1887.9	1880.7	1871.3	1870.8
slack	**1850.0**	1864.7	1862.9	1862.1	1861.0	1861.0
fastrob	**1850.0**	**1862.8**	**1861.3**	**1860.4**	**1860.1**	**1860.1**
robust	**1850.0**	1864.6	1862.7	1861.8	1861.6	1861.5

la33						
method	P	1	2	3	4	5
active	**1719.0**	1759.6	1750.1	1742.5	1736.1	1735.9
slack	1719.1	1733.7	1732.7	1731.9	1731.3	1731.3
fastrob	**1719.0**	**1731.3**	**1729.8**	**1729.1**	**1728.7**	**1728.7**
robust	**1719.0**	1732.8	1731.1	1730.0	1729.4	1729.4

la34						
method	P	1	2	3	4	5
active	**1721.0**	1773.9	1766.3	1758.1	1748.5	1747.8
slack	1721.2	1753.9	1751.4	1749.0	1745.5	1745.3
fastrob	**1721.0**	**1749.0**	**1745.1**	**1742.1**	**1739.8**	**1739.6**
robust	**1721.0**	1749.8	1746.4	1744.3	1742.3	1742.2

la35						
method	P	1	2	3	4	5
active	**1888.0**	1922.9	1915.5	1909.5	1904.4	1904.0
slack	1888.9	1907.8	1906.0	1904.2	1902.9	1902.5
fastrob	**1888.0**	1906.9	1903.7	1902.1	1901.0	1901.0
robust	**1888.0**	**1903.8**	**1901.3**	**1899.9**	**1899.1**	**1898.9**

la36						
method	P	1	2	3	4	5
active	**1297.2**	1345.7	1338.1	1334.5	**1324.0**	**1321.0**
slack	1303.9	1342.3	1338.4	1336.6	1332.7	1330.3
fastrob	1299.8	1341.2	1334.3	1331.6	1326.3	1324.8
robust	1299.8	**1336.6**	**1331.8**	**1330.6**	1325.3	1323.5

la37						
method	P	1	2	3	4	5
active	**1440.4**	1490.3	1481.1	1476.7	1462.9	1459.7
slack	1447.8	1483.2	1477.6	1474.9	1468.6	1465.4
fastrob	1442.0	**1480.5**	**1472.7**	**1469.2**	**1460.7**	**1458.3**
robust	1443.3	1480.8	1475.1	1473.1	1465.3	1462.6

la38						
method	P	1	2	3	4	5
active	1253.7	1305.3	1298.0	1292.7	1276.0	1270.7
slack	1254.3	1300.9	1295.3	1291.8	1278.8	1273.2
fastrob	**1244.5**	**1293.2**	**1286.3**	**1283.4**	**1267.6**	**1263.5**
robust	1248.2	1293.7	1287.3	1284.1	1270.3	1266.0

la39						
method	P	1	2	3	4	5
active	**1254.1**	1307.6	1300.9	1297.0	1285.1	1282.0
slack	1260.2	1300.8	1296.0	1293.3	1286.1	1283.4
fastrob	1256.9	1301.3	1294.1	**1290.9**	**1283.2**	**1280.6**
robust	1256.9	**1299.5**	**1294.0**	1291.9	1284.3	1281.5

la40						
method	P	1	2	3	4	5
active	**1257.1**	1309.2	1300.6	1295.1	1280.2	1277.1
slack	1263.4	1299.9	1294.4	1291.4	1283.1	1279.8
fastrob	1259.3	**1297.1**	**1291.1**	**1287.8**	**1278.0**	**1275.2**
robust	1259.3	1300.3	1295.5	1292.7	1282.6	1279.1

swv01						
method	P	1	2	3	4	5
active	1645.2	1700.8	1693.6	1686.5	1662.6	1657.7
slack	1637.5	1683.2	1678.7	1675.1	1654.2	1650.4
fastrob	**1615.6**	**1666.2**	**1659.1**	**1653.8**	**1631.1**	**1626.9**
robust	1622.3	1673.2	1667.1	1662.9	1640.6	1636.6

swv02						
method	P	1	2	3	4	5
active	1675.9	1729.0	1721.0	1714.1	1691.0	1686.5
slack	1665.2	1713.3	1708.7	1703.9	1682.4	1677.9
fastrob	**1643.2**	**1692.0**	**1684.6**	**1677.6**	**1658.8**	**1654.5**
robust	1646.0	1698.5	1692.9	1689.1	1667.6	1664.8

swv03						
method	P	1	2	3	4	5
active	1615.6	1676.0	1667.6	1660.6	1627.6	1623.4
slack	1605.7	1657.4	1652.8	1649.8	1624.8	1621.1
fastrob	**1587.4**	**1639.1**	**1632.8**	**1628.4**	**1605.9**	**1601.1**
robust	1589.1	1643.2	1638.3	1634.2	1606.3	1602.8

swv04						
method	P	1	2	3	4	5
active	1694.7	1749.4	1742.0	1735.1	1708.6	1704.5
slack	1682.6	1733.0	1728.5	1724.6	1701.3	1697.0
fastrob	**1668.4**	**1719.9**	**1713.6**	**1708.8**	**1683.1**	**1679.1**
robust	1671.0	1721.3	1716.2	1712.3	1690.6	1686.5

swv05						
method	P	1	2	3	4	5
active	1647.2	1708.2	1702.1	1696.6	1669.9	1666.4
slack	1641.8	1691.9	1687.5	1684.7	1663.9	1661.5
fastrob	**1628.6**	**1683.3**	**1677.8**	**1673.6**	**1653.8**	**1650.4**
robust	1632.1	1690.1	1685.7	1681.4	1659.1	1655.6

swv06						
method	P	1	2	3	4	5
active	1949.1	2004.7	1997.6	1992.5	1958.8	1955.1
slack	1938.7	1984.8	1979.8	1975.2	1948.2	1944.5
fastrob	**1918.1**	**1968.5**	**1961.1**	**1956.4**	**1932.7**	**1928.4**
robust	1923.6	1973.6	1967.0	1962.7	1936.2	1931.3

swv07						
method	P	1	2	3	4	5
active	1850.8	1906.7	1900.5	1894.7	1866.1	1861.5
slack	1844.6	1895.6	1890.7	1886.8	1862.6	1859.4
fastrob	**1827.8**	1882.0	1875.4	1871.1	**1847.5**	**1842.5**
robust	1828.9	**1880.1**	**1874.6**	**1870.8**	1848.4	1843.1

swv08						
method	P	1	2	3	4	5
active	2054.8	2109.4	2102.7	2095.7	2062.5	2056.0
slack	2042.4	2090.6	2085.1	2081.4	2054.1	2047.7
fastrob	**2019.5**	**2069.4**	**2062.9**	**2058.2**	**2030.4**	**2025.2**
robust	2027.9	2080.3	2074.6	2070.2	2039.7	2034.1

swv09						
method	P	1	2	3	4	5
active	1938.8	1992.3	1985.1	1977.8	1942.6	1937.2
slack	1925.9	1974.1	1969.7	1965.6	1936.6	1931.8
fastrob	**1903.1**	**1954.4**	**1948.1**	**1943.8**	**1916.5**	1911.4
robust	1904.0	1957.3	1951.0	1946.3	1917.2	**1911.2**

swv10						
method	P	1	2	3	4	5
active	2004.6	2059.5	2052.0	2044.4	2011.2	2005.8
slack	1994.4	2040.0	2035.5	2032.3	2006.4	2001.4
fastrob	**1976.6**	**2029.0**	**2022.8**	**2018.8**	**1992.4**	**1987.6**
robust	1979.7	2030.3	2023.9	2019.7	1994.3	1990.1

ta01						
method	P	1	2	3	4	5
active	**1271.0**	1323.4	1315.1	1310.5	**1296.2**	1292.9
slack	1279.6	1316.5	1310.7	1307.9	1301.9	1297.9
fastrob	1273.6	**1312.9**	**1306.8**	**1304.2**	1296.4	**1292.5**
robust	1274.1	1316.5	1311.0	1308.1	1299.0	1295.9

ta02						
method	P	1	2	3	4	5
active	**1275.1**	1326.7	1318.3	1313.7	1300.5	1296.5
slack	1278.0	1317.2	1310.8	1307.6	1299.5	1297.2
fastrob	1276.2	**1315.2**	**1309.2**	**1306.6**	**1295.5**	**1291.8**
robust	1276.2	1317.6	1310.3	1307.6	1297.0	1294.3

ta03						
method	P	1	2	3	4	5
active	1255.0	1311.5	1304.1	1300.2	1282.0	1278.3
slack	1257.3	1304.7	1298.6	1295.5	1285.1	1282.1
fastrob	**1252.4**	**1301.9**	**1295.4**	**1292.1**	**1279.5**	**1276.3**
robust	1253.5	1305.3	1299.2	1296.3	1284.8	1281.7

ta04						
method	P	1	2	3	4	5
active	1214.8	1272.4	1264.6	1259.9	1243.0	1238.7
slack	1215.4	1263.9	1257.5	1254.0	1244.3	1240.5
fastrob	**1205.7**	1260.6	1253.0	1249.1	1238.2	1235.1
robust	1208.4	**1258.0**	**1251.6**	**1248.4**	**1235.7**	**1232.3**

ta05						
method	P	1	2	3	4	5
active	1257.9	1311.2	1303.4	1297.6	1286.4	1282.0
slack	1260.5	1302.5	1297.6	1293.9	1286.6	1284.0
fastrob	**1253.7**	**1301.2**	**1293.6**	**1290.4**	**1281.8**	**1278.3**
robust	1256.4	1303.2	1296.2	1293.3	1285.2	1282.1

ta06						
method	P	1	2	3	4	5
active	**1271.9**	1327.0	1319.3	1314.6	**1298.0**	**1293.6**
slack	1274.5	1320.1	1314.8	1311.4	1304.5	1300.2
fastrob	1272.0	1322.5	1314.7	1311.8	1299.6	1294.9
robust	1272.9	**1319.9**	**1313.4**	**1310.7**	1299.5	1294.9

ta07						
method	P	1	2	3	4	5
active	1264.6	1317.2	1309.6	1305.0	1291.3	1287.3
slack	1267.1	1309.8	1305.4	1302.2	1294.7	1291.4
fastrob	**1263.7**	**1307.5**	**1302.2**	**1299.7**	**1290.0**	**1286.7**
robust	1264.1	1308.3	1302.9	1300.6	1290.6	1287.7

ta08						
method	P	1	2	3	4	5
active	**1256.4**	1312.7	1305.4	1301.6	1286.6	1282.8
slack	1261.1	1306.5	1300.8	1298.0	1290.6	1287.4
fastrob	1256.6	1307.3	1300.3	**1296.8**	**1284.2**	**1280.8**
robust	1258.4	**1305.7**	**1299.9**	1297.7	1287.1	1282.5

ta09						
method	P	1	2	3	4	5
active	1330.5	1382.5	1376.1	1372.5	1357.7	**1353.2**
slack	1336.4	1379.6	1375.2	1373.8	1364.2	1360.9
fastrob	**1328.6**	**1376.4**	**1370.6**	**1367.7**	**1356.8**	**1353.2**
robust	1329.4	1376.9	1372.8	1371.1	1361.4	1357.4

ta10						
method	P	1	2	3	4	5
active	**1288.6**	1340.8	1332.2	1328.4	**1316.9**	**1312.1**
slack	1294.9	**1332.1**	**1327.4**	**1325.2**	1320.5	1317.5
fastrob	1289.5	1335.1	1328.7	1326.4	1317.9	1314.3
robust	1290.6	1332.7	1327.8	1325.7	1317.4	1313.8

ta11						
method	P	1	2	3	4	5
active	1463.8	1514.6	1505.5	1498.9	1479.7	1474.9
slack	1460.6	1504.4	1498.6	1494.8	1479.2	1475.6
fastrob	**1444.6**	**1493.4**	**1487.4**	**1482.9**	**1465.9**	**1461.9**
robust	1448.0	1493.7	1488.6	1484.5	1469.2	1465.2

ta12						
method	P	1	2	3	4	5
active	1442.6	1495.2	1488.2	1481.6	1463.3	1458.8
slack	1440.6	1481.6	1477.2	1474.0	1461.5	1457.0
fastrob	**1429.9**	**1475.3**	1468.9	**1463.8**	**1449.9**	**1446.8**
robust	1431.7	1476.0	**1468.4**	1464.9	1452.2	1449.5

ta13						
method	P	1	2	3	4	5
active	1437.8	1490.5	1482.3	1476.5	1454.2	1449.0
slack	1436.0	1480.0	1474.2	1469.9	1455.9	1450.5
fastrob	**1420.0**	**1468.5**	**1461.2**	**1456.2**	**1440.4**	**1436.6**
robust	1423.9	1472.2	1466.2	1462.0	1444.9	1440.5

ta14						
method	P	1	2	3	4	5
active	1392.5	1445.3	1437.7	1432.8	1412.3	1408.4
slack	1392.5	1430.5	1425.6	1422.0	1412.2	1409.9
fastrob	**1383.1**	**1426.4**	**1420.3**	**1416.2**	**1401.4**	**1398.5**
robust	1384.6	1427.4	1421.9	1418.8	1405.9	1403.7

ta15						
method	P	1	2	3	4	5
active	1459.1	1510.1	1502.6	1496.6	1474.3	1470.3
slack	1458.4	1499.7	1495.1	1491.8	1479.0	1475.1
fastrob	**1445.2**	**1491.5**	**1485.4**	**1481.0**	**1466.3**	**1461.9**
robust	1448.6	1493.0	1487.0	1483.7	1467.9	1464.2

ta16						
method	P	1	2	3	4	5
active	1436.1	1490.2	1482.8	1478.4	1460.9	1456.8
slack	1435.6	1480.6	1476.2	1473.3	1462.3	1458.4
fastrob	**1426.0**	**1473.5**	**1468.1**	**1465.5**	**1452.3**	**1449.2**
robust	1428.4	1475.4	1470.7	1468.3	1455.0	1451.1

ta17						
method	P	1	2	3	4	5
active	1543.8	1589.3	1582.3	1577.2	1562.7	1559.6
slack	1550.3	1583.9	1579.2	1575.9	1566.2	1562.8
fastrob	**1535.6**	**1571.5**	**1564.8**	**1561.4**	**1552.0**	**1549.7**
robust	1537.3	1573.2	1567.4	1564.4	1554.4	1551.7

ta18						
method	P	1	2	3	4	5
active	1515.5	1561.3	1553.4	1546.6	1527.9	1523.3
slack	1518.6	1553.5	1547.7	1543.8	1533.4	1529.6
fastrob	**1502.0**	1543.5	**1536.2**	1532.6	**1518.7**	**1514.8**
robust	1502.6	**1542.8**	1536.5	**1532.4**	1519.3	1515.1

ta19						
method	P	1	2	3	4	5
active	1451.7	1504.0	1495.2	1489.3	1473.0	1468.2
slack	1457.3	1491.9	1487.6	1484.9	1474.9	1470.9
fastrob	**1436.9**	**1477.7**	**1471.3**	**1466.8**	**1455.8**	**1451.9**
robust	1440.6	1481.6	1475.7	1472.4	1462.1	1458.0

ta20						
method	P	1	2	3	4	5
active	1424.5	1480.4	1473.5	1468.3	1447.1	1442.6
slack	1423.2	1470.4	1464.6	1459.7	1446.2	1441.3
fastrob	**1410.3**	**1457.7**	**1451.8**	**1447.0**	**1434.0**	1430.1
robust	1412.2	1458.7	1453.2	1449.6	1434.4	**1429.6**

ta21						
method	P	1	2	3	4	5
active	1788.5	1836.4	1826.9	1820.5	1796.4	1792.2
slack	1788.2	1827.9	1820.9	1816.3	1802.5	1797.7
fastrob	**1772.6**	**1813.0**	**1806.2**	**1802.2**	**1787.0**	**1782.7**
robust	1774.7	1813.6	1808.0	1804.5	1788.7	1784.8

ta22						
method	P	1	2	3	4	5
active	1703.0	1754.1	1745.2	1740.5	1717.2	1712.0
slack	1696.3	1734.5	1728.1	1723.5	1709.3	1703.6
fastrob	**1680.0**	**1726.1**	**1720.6**	**1715.7**	**1697.3**	**1692.8**
robust	1684.0	1728.1	1721.7	1717.8	1701.9	1697.2

ta23						
method	P	1	2	3	4	5
active	1656.8	1710.4	1702.1	1694.7	1672.3	1668.2
slack	1653.7	1696.3	1690.2	1685.9	1673.0	1669.1
fastrob	**1640.2**	**1683.2**	**1676.6**	**1673.2**	**1661.5**	**1657.5**
robust	1643.7	1688.7	1682.7	1678.5	1663.1	1659.7

ta24						
method	P	1	2	3	4	5
active	1745.7	1790.3	1780.5	1773.4	1756.2	1752.5
slack	1746.4	1780.5	1774.8	1771.1	1760.6	1757.2
fastrob	**1736.3**	**1774.2**	**1769.0**	1766.0	1755.5	**1751.1**
robust	1738.8	1776.3	1769.1	**1765.8**	**1754.8**	1751.2

ta25						
method	P	1	2	3	4	5
active	1700.6	1749.4	1740.0	1734.4	1714.7	1709.8
slack	1699.5	1740.6	1735.0	1731.0	1717.9	1714.7
fastrob	**1687.3**	**1729.2**	**1723.4**	**1719.9**	**1706.4**	**1703.3**
robust	1690.2	1735.8	1729.9	1726.3	1711.7	1707.6

ta26						
method	P	1	2	3	4	5
active	1752.5	1796.6	1788.0	1782.8	1763.0	1758.9
slack	1751.2	1792.4	1786.6	1783.2	1767.2	1763.3
fastrob	**1736.1**	**1779.2**	**1773.7**	**1771.1**	**1756.8**	**1753.0**
robust	1740.2	1782.2	1776.1	1772.9	1757.9	1754.4

ta27						
method	P	1	2	3	4	5
active	1820.8	1867.4	1857.7	1851.3	1829.4	1823.7
slack	1817.1	1854.3	1847.9	1844.1	1830.1	1825.2
fastrob	**1798.1**	**1839.5**	**1832.8**	**1829.4**	**1813.4**	**1809.0**
robust	1800.9	1842.1	1835.4	1831.2	1813.7	1809.0

ta28						
method	P	1	2	3	4	5
active	1713.7	1763.0	1754.6	1749.2	1725.7	1721.4
slack	1709.9	1749.7	1744.1	1740.6	1724.4	1720.0
fastrob	**1699.9**	**1742.4**	**1736.4**	**1732.2**	**1714.3**	**1711.2**
robust	1702.7	1744.1	1738.3	1734.4	1716.9	1712.9

ta29						
method	P	1	2	3	4	5
active	1678.6	1731.4	1721.7	1716.5	1695.9	1691.8
slack	1674.1	1715.9	1709.9	1706.8	1695.1	1691.2
fastrob	**1660.0**	**1708.3**	**1702.1**	**1698.4**	**1685.6**	**1682.3**
robust	1663.5	1709.9	1702.8	1699.3	1686.3	1683.9

ta30						
method	P	1	2	3	4	5
active	1691.1	1741.8	1733.2	1727.6	1702.9	1699.1
slack	1686.1	1732.2	1725.8	1721.0	1701.5	1697.6
fastrob	**1673.1**	**1718.0**	**1711.6**	**1707.5**	**1689.1**	**1685.8**
robust	1676.9	1720.5	1714.4	1709.6	1690.5	1686.0

ta31						
method	P	1	2	3	4	5
active	1912.9	1959.4	1950.9	1941.8	1921.9	1916.1
slack	1900.4	1939.1	1934.3	1929.5	1915.6	1910.9
fastrob	**1864.2**	**1905.4**	**1900.5**	**1895.6**	**1880.9**	**1877.1**
robust	1866.6	1910.2	1904.5	1899.5	1883.6	1879.4

ta32						
method	P	1	2	3	4	5
active	2004.5	2049.8	2041.8	2032.8	2013.0	2007.5
slack	1995.9	2028.9	2023.6	2019.3	2000.6	1996.5
fastrob	**1959.7**	**2000.7**	**1994.3**	**1989.3**	**1972.0**	**1967.2**
robust	1962.1	2005.1	1998.3	1993.3	1977.4	1972.2

ta33						
method	P	1	2	3	4	5
active	1988.3	2036.7	2028.5	2020.1	1995.0	1989.8
slack	1976.7	2016.0	2010.4	2005.0	1991.0	1985.5
fastrob	**1941.4**	1987.3	**1980.9**	**1975.2**	**1956.3**	**1952.1**
robust	1944.3	**1986.4**	**1980.9**	1976.0	1958.9	1954.8

ta34						
method	P	1	2	3	4	5
active	2023.4	2065.7	2057.5	2051.4	2027.1	2021.1
slack	2020.6	2051.1	2047.5	2044.4	2025.7	2021.4
fastrob	**1983.6**	**2021.3**	**2015.7**	**2011.7**	**1993.4**	**1988.2**
robust	1987.3	2023.8	2019.4	2015.2	1995.8	1990.1

ta35						
method	P	1	2	3	4	5
active	2112.3	2146.1	2138.8	2132.5	2107.5	2102.9
slack	2117.9	2137.0	2134.7	2133.0	2113.5	2110.9
fastrob	**2081.7**	**2105.6**	**2101.7**	**2099.3**	**2083.0**	**2079.1**
robust	2084.0	2110.3	2105.6	2102.8	2085.6	2081.9

ta36						
method	P	1	2	3	4	5
active	1997.7	2044.1	2035.2	2027.1	1998.1	1992.0
slack	1985.4	2025.6	2019.8	2014.8	1993.0	1988.8
fastrob	**1948.6**	**1994.3**	**1988.8**	**1984.2**	**1963.4**	**1959.4**
robust	1952.5	1997.9	1992.7	1988.4	1964.5	1960.0

ta37						
method	P	1	2	3	4	5
active	1958.2	2002.6	1993.4	1986.1	1962.0	1956.7
slack	1949.7	1982.6	1977.5	1973.1	1956.9	1950.6
fastrob	**1911.9**	**1954.3**	**1948.4**	**1942.8**	**1925.1**	**1919.8**
robust	1915.0	1955.9	1949.6	1944.4	1927.2	1921.8

ta38						
method	P	1	2	3	4	5
active	1853.1	1904.0	1896.7	1888.4	1863.5	1858.7
slack	1840.4	1882.1	1876.8	1871.5	1853.7	1848.8
fastrob	**1805.4**	**1851.2**	**1845.5**	**1840.3**	**1821.8**	**1817.7**
robust	1809.1	1853.6	1847.5	1842.1	1826.0	1821.9

ta39						
method	P	1	2	3	4	5
active	1987.4	2025.8	2017.8	2011.9	1990.2	1985.5
slack	1983.4	2014.3	2010.3	2007.7	1991.8	1986.0
fastrob	**1944.2**	**1981.4**	**1976.4**	**1972.5**	**1957.9**	**1954.5**
robust	1949.5	1987.0	1981.9	1977.7	1962.2	1956.6

ta40						
method	P	1	2	3	4	5
active	1862.6	1912.9	1904.5	1896.5	1870.1	1865.3
slack	1854.6	1894.3	1890.3	1886.3	1869.4	1864.6
fastrob	**1827.2**	**1869.0**	**1863.8**	**1859.2**	**1841.6**	**1837.2**
robust	1830.8	1875.1	1869.9	1865.0	1846.9	1842.3

G.2 Overlaps

Average overlaps to the preschedule after rescheduling. The highest overlaps have been printed bold.

la01-la05					
method	1	2	3	4	5
active	0.537	0.582	0.568	0.589	0.541
slack	**0.579**	**0.616**	**0.591**	**0.632**	0.549
fastrob	0.571	0.609	0.586	0.617	**0.554**
robust	0.577	0.613	0.590	0.618	0.547

la06-la10					
method	1	2	3	4	5
active	0.589	0.647	0.631	0.634	**0.575**
slack	0.637	0.677	0.642	**0.644**	0.518
fastrob	**0.647**	**0.694**	**0.654**	0.636	0.487
robust	**0.647**	0.690	0.653	0.626	0.474

la11-la15					
method	1	2	3	4	5
active	0.598	0.653	0.638	**0.647**	**0.617**
slack	0.655	0.697	0.657	0.635	0.503
fastrob	**0.664**	**0.711**	**0.669**	0.616	0.433
robust	0.660	0.707	0.666	0.594	0.424

la16-la20					
method	1	2	3	4	5
active	0.578	0.623	0.621	0.660	0.589
slack	**0.635**	**0.668**	**0.660**	**0.693**	**0.648**
fastrob	0.613	0.649	0.648	0.684	0.630
robust	0.617	0.650	0.643	0.674	0.616

la21-la25					
method	1	2	3	4	5
active	0.602	0.647	0.648	0.681	0.585
slack	**0.629**	**0.666**	0.659	**0.710**	**0.646**
fastrob	0.614	0.652	0.650	0.696	0.636
robust	0.628	0.665	**0.661**	0.697	0.625

la26-la30					
method	1	2	3	4	5
active	0.603	0.654	0.661	0.644	0.493
slack	**0.659**	**0.696**	**0.687**	**0.703**	**0.600**
fastrob	0.646	0.688	0.686	0.698	0.587
robust	0.651	0.691	**0.687**	0.694	0.582

la31-la35					
method	1	2	3	4	5
active	0.662	0.717	0.735	**0.702**	**0.625**
slack	**0.723**	0.761	**0.736**	**0.702**	0.623
fastrob	0.711	0.758	0.728	0.679	0.556
robust	0.718	**0.763**	0.734	0.670	0.550

la36-la40					
method	1	2	3	4	5
active	0.601	0.657	0.668	0.693	0.604
slack	0.637	0.680	0.680	0.721	0.662
fastrob	0.630	0.677	0.685	**0.725**	**0.665**
robust	**0.642**	**0.687**	**0.686**	0.715	0.647

swv01-swv05					
method	1	2	3	4	5
active	0.501	0.562	0.574	0.532	0.382
slack	**0.548**	**0.597**	0.591	0.567	0.436
fastrob	0.528	0.583	0.590	**0.570**	**0.444**
robust	0.544	0.596	**0.596**	**0.570**	0.436

swv06-swv10					
method	1	2	3	4	5
active	0.520	0.578	0.599	0.564	0.441
slack	**0.561**	**0.607**	**0.613**	**0.607**	**0.504**
fastrob	0.549	0.603	0.612	0.605	0.497
robust	0.547	0.600	0.611	0.598	0.496

ta01-ta10					
method	1	2	3	4	5
active	0.595	0.648	0.656	0.689	0.607
slack	**0.639**	**0.679**	**0.680**	**0.726**	**0.671**
fastrob	0.617	0.665	0.670	0.720	0.665
robust	0.624	0.668	0.670	0.714	0.654

ta11-ta20					
method	1	2	3	4	5
active	0.605	0.657	0.675	0.667	0.543
slack	**0.645**	**0.687**	0.690	**0.714**	**0.625**
fastrob	0.626	0.675	0.687	0.713	0.623
robust	0.642	0.686	**0.693**	0.708	0.612

ta21-ta30					
method	1	2	3	4	5
active	0.616	0.677	0.699	0.691	0.582
slack	**0.654**	**0.700**	**0.711**	**0.725**	0.646
fastrob	0.643	0.689	0.702	0.724	**0.649**
robust	0.648	0.695	0.708	0.722	0.641

ta31-ta40					
method	1	2	3	4	5
active	0.647	0.706	0.733	0.614	0.444
slack	**0.711**	**0.750**	**0.755**	0.679	**0.556**
fastrob	0.689	0.735	0.747	0**.682**	0.552
robust	0.691	0.734	0.745	0.671	0.540

Appendix H

Results of maximum tardiness experiments

H.1 Loose problems, $\sigma = 0.95$

Results for individual problems for the experiments of section 5.4.3.

ft10						
method	P	1	2	3	4	5
naive	**0.0**	47.5	40.4	35.8	17.9	16.8
active	**0.0**	16.7	**13.0**	**12.2**	**6.6**	**6.2**
robust	**0.0**	**15.6**	13.1	12.3	8.0	7.3

ft20						
method	P	1	2	3	4	5
naive	**0.0**	45.2	39.3	33.4	12.8	12.3
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la01						
method	P	1	2	3	4	5
naive	**0.0**	55.0	49.7	46.5	38.5	37.1
active	**0.0**	46.7	41.8	**37.8**	**33.1**	**32.0**
robust	**0.0**	**42.5**	**39.2**	38.9	35.5	34.9

la02						
method	P	1	2	3	4	5
naive	**0.0**	49.2	44.8	41.9	34.0	32.2
active	**0.0**	**37.1**	**33.9**	**32.6**	29.5	28.5
robust	**0.0**	37.5	34.5	32.7	**28.5**	**24.8**

la03						
method	P	1	2	3	4	5
naive	**0.0**	46.5	42.8	41.6	37.2	36.5
active	**0.0**	39.1	**33.0**	**32.3**	**31.2**	**31.1**
robust	**0.0**	**38.9**	36.6	36.4	35.9	35.5

la04						
method	P	1	2	3	4	5
naive	**0.0**	45.3	38.8	35.1	28.3	27.5
active	**0.0**	**29.1**	**23.8**	**21.2**	**16.8**	**16.7**
robust	**0.0**	32.6	24.6	23.5	22.2	21.8

la05						
method	P	1	2	3	4	5
naive	**30.0**	86.4	80.9	77.3	69.6	68.7
active	**30.0**	88.0	82.7	79.5	71.2	70.4
robust	**30.0**	**70.8**	**67.6**	**66.6**	**61.4**	**60.6**

la06						
method	P	1	2	3	4	5
naive	**43.0**	98.4	92.7	86.9	75.9	75.2
active	**43.0**	97.5	91.9	87.2	75.2	74.4
robust	**43.0**	**77.0**	**74.2**	**73.3**	**67.3**	**66.7**

la07						
method	P	1	2	3	4	5
naive	**0.0**	39.1	32.6	27.2	15.6	15.4
active	**0.0**	7.7	6.4	6.0	5.5	5.5
robust	**0.0**	**4.4**	**4.2**	**4.3**	**4.8**	**4.4**

la08						
method	P	1	2	3	4	5
naive	**0.0**	37.7	31.8	26.6	11.7	11.4
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la09						
method	P	1	2	3	4	5
naive	**16.0**	74.9	68.0	63.5	51.3	50.4
active	**16.0**	75.2	68.1	64.0	52.7	51.7
robust	16.4	**57.1**	**53.3**	**53.3**	**48.1**	**47.4**

la10						
method	P	1	2	3	4	5
naive	**32.0**	83.8	77.2	72.2	67.5	67.4
active	**32.0**	84.5	76.9	71.8	68.7	68.5
robust	**32.0**	**69.3**	**66.3**	**66.4**	**66.9**	**66.2**

la11						
method	P	1	2	3	4	5
naive	**36.0**	97.1	93.2	88.9	76.6	76.0
active	**36.0**	96.2	91.2	86.3	74.4	74.2
robust	**36.0**	**68.7**	**65.2**	**63.3**	**61.1**	**60.9**

la12						
method	P	1	2	3	4	5
naive	**14.0**	75.2	69.1	65.6	51.6	51.5
active	**14.0**	76.7	69.9	64.5	51.5	51.4
robust	14.2	**63.4**	**57.8**	**54.5**	**49.3**	**48.8**

la13						
method	P	1	2	3	4	5
naive	**0.0**	60.8	57.0	53.5	38.8	38.1
active	**0.0**	56.7	53.3	49.8	38.9	38.3
robust	0.2	**42.1**	**37.7**	**36.8**	**32.0**	**31.3**

la14						
method	P	1	2	3	4	5
naive	**56.0**	109.9	102.9	97.5	86.5	86.4
active	**56.0**	108.2	101.3	96.0	83.8	83.5
robust	**56.0**	**80.3**	**78.0**	**76.8**	**76.5**	**76.5**

la15						
method	P	1	2	3	4	5
naive	**0.0**	40.2	34.6	29.6	13.3	12.8
active	**0.0**	2.0	1.7	**1.7**	**0.7**	**0.7**
robust	**0.0**	**1.8**	**1.6**	**1.7**	1.2	1.7

la16						
method	P	1	2	3	4	5
naive	**0.0**	41.1	34.4	31.6	23.8	22.9
active	**0.0**	**19.9**	**16.4**	**14.3**	**10.2**	**9.6**
robust	**0.0**	**19.9**	17.0	16.5	12.8	12.0

la17						
method	P	1	2	3	4	5
naive	0.2	48.8	43.2	41.2	**36.4**	34.4
active	0.2	47.1	43.1	41.1	**36.4**	34.7
robust	**0.1**	**43.2**	**39.3**	**38.4**	39.0	**34.3**

la18						
method	P	1	2	3	4	5
naive	**0.0**	48.3	41.7	39.2	27.1	25.4
active	**0.0**	27.7	23.6	21.5	**14.4**	**13.2**
robust	**0.0**	**25.7**	**21.7**	**20.4**	15.7	14.9

la19						
method	P	1	2	3	4	5
naive	**0.0**	48.9	43.1	39.2	28.0	26.8
active	**0.0**	42.1	36.9	33.7	**27.1**	**26.1**
robust	0.1	**35.6**	**31.5**	**29.9**	30.0	27.6

la20						
method	P	1	2	3	4	5
naive	**0.0**	37.6	30.8	27.4	21.9	21.8
active	**0.0**	26.2	21.4	19.7	**14.9**	**14.5**
robust	**0.0**	**23.5**	**19.5**	**18.4**	17.2	17.1

la21						
method	P	1	2	3	4	5
naive	**0.0**	32.1	25.5	19.9	10.3	9.8
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la22						
method	P	1	2	3	4	5
naive	**0.0**	31.5	24.5	19.7	9.4	9.4
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la23						
method	P	1	2	3	4	5
naive	**0.0**	31.9	24.7	20.2	9.4	9.1
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la24						
method	P	1	2	3	4	5
naive	**0.0**	44.9	38.3	34.1	16.6	15.4
active	**0.0**	**5.8**	**4.4**	**3.8**	**2.3**	**2.3**
robust	**0.0**	6.5	4.9	4.7	3.0	2.5

la25						
method	P	1	2	3	4	5
naive	**0.0**	43.7	35.7	31.3	19.4	18.6
active	**0.0**	26.7	21.6	18.8	10.6	10.3
robust	**0.0**	**19.4**	**15.3**	**13.8**	**9.4**	**8.8**

la26						
method	P	1	2	3	4	5
naive	**0.0**	31.0	22.5	16.1	6.2	6.0
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la27						
method	P	1	2	3	4	5
naive	**0.0**	35.1	27.9	22.0	7.6	7.3
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la28						
method	P	1	2	3	4	5
naive	**0.0**	44.8	36.1	28.7	13.3	12.7
active	**0.0**	**0.2**	0.2	0.2	**0.0**	**0.0**
robust	**0.0**	**0.2**	**0.1**	**0.1**	0.3	0.3

la29						
method	P	1	2	3	4	5
naive	**0.0**	40.4	34.2	29.8	14.0	13.5
active	**0.0**	2.1	1.7	1.5	0.9	0.9
robust	**0.0**	**1.2**	**1.0**	**0.9**	**0.8**	**0.7**

la30						
method	P	1	2	3	4	5
naive	**0.0**	26.9	20.2	16.0	5.7	5.5
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la31						
method	P	1	2	3	4	5
naive	**0.0**	30.6	24.5	19.3	6.4	6.2
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la32						
method	P	1	2	3	4	5
naive	**0.0**	32.1	25.6	18.3	4.9	4.7
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la33						
method	P	1	2	3	4	5
naive	**0.0**	39.6	31.8	25.3	8.3	7.8
active	**0.0**	3.7	2.5	2.0	0.7	0.8
robust	**0.0**	**0.6**	**0.5**	**0.5**	**0.2**	**0.1**

la34						
method	P	1	2	3	4	5
naive	**0.0**	38.4	30.6	24.4	11.0	11.6
active	**0.0**	14.0	10.8	9.0	5.4	5.9
robust	**0.0**	**3.9**	**3.6**	**3.4**	**2.4**	**2.6**

la35						
method	P	1	2	3	4	5
naive	**0.0**	26.0	19.8	16.3	10.5	10.4
active	**0.0**	12.6	10.0	8.7	6.0	6.1
robust	**0.0**	**6.4**	**5.6**	**5.3**	**4.2**	**4.2**

la36						
method	P	1	2	3	4	5
naive	**0.0**	34.9	26.9	22.4	12.7	12.6
active	**0.0**	11.7	**8.8**	**7.6**	**4.8**	**4.7**
robust	**0.0**	**11.0**	9.1	8.3	7.0	6.9

la37						
method	P	1	2	3	4	5
naive	**0.2**	39.2	30.3	25.2	10.7	10.4
active	0.3	**6.4**	**4.8**	**4.2**	**3.1**	**2.7**
robust	0.4	6.9	5.3	5.2	4.3	3.2

la38						
method	P	1	2	3	4	5
naive	0.3	35.8	29.5	25.1	12.6	12.2
active	**0.1**	4.3	3.6	**3.2**	**1.9**	**2.0**
robust	0.7	**3.8**	**3.5**	3.4	2.9	2.5

la39						
method	P	1	2	3	4	5
naive	**0.0**	33.3	25.7	20.2	8.0	7.5
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la40						
method	P	1	2	3	4	5
naive	**0.0**	47.3	38.9	33.1	18.2	17.1
active	**0.0**	29.1	22.8	19.5	**11.7**	**11.0**
robust	**0.0**	**24.9**	**19.6**	**17.8**	12.1	11.7

H.2 The tight problems, $\sigma = 0.85$

Results for individual problems for the experiments of section 5.4.4

ft10						
method	P	1	2	3	4	5
naive	**47.6**	108.6	103.5	100.6	86.5	84.1
active	47.8	**104.6**	**99.4**	**96.0**	**81.9**	**79.6**
robust	52.1	105.8	99.7	97.5	88.1	86.2

ft20						
method	P	1	2	3	4	5
naive	**1.9**	62.2	56.2	51.7	34.9	33.2
active	2.6	50.7	46.0	43.1	29.3	28.1
robust	2.6	**45.5**	**41.0**	**39.1**	**27.0**	**25.7**

la01						
method	P	1	2	3	4	5
naive	**57.0**	118.0	112.7	108.8	104.4	101.8
active	**57.0**	113.9	109.1	**105.1**	**100.8**	**98.4**
robust	**57.0**	**111.6**	**107.3**	106.4	103.5	101.3

la02						
method	P	1	2	3	4	5
naive	**31.0**	95.0	92.8	91.9	**86.3**	85.0
active	**31.0**	96.1	93.3	92.4	86.5	**84.9**
robust	**31.0**	**92.5**	**90.7**	**89.9**	86.5	85.5

la03						
method	P	1	2	3	4	5
naive	44.7	**101.8**	**97.8**	**96.4**	**93.9**	**93.0**
active	**44.6**	105.2	101.0	98.8	94.8	93.7
robust	50.2	107.1	103.6	103.7	102.0	101.3

la04						
method	P	1	2	3	4	5
naive	**32.4**	94.4	88.9	86.0	80.4	80.4
active	**32.4**	**93.4**	**88.1**	**85.1**	**79.1**	**79.2**
robust	41.0	95.8	88.7	87.0	85.0	84.3

la05						
method	P	1	2	3	4	5
naive	**89.0**	146.6	140.5	136.8	130.2	129.5
active	**89.0**	146.9	140.8	136.7	127.2	126.4
robust	**89.0**	**122.7**	**119.5**	**119.1**	**118.7**	**117.8**

la06						
method	P	1	2	3	4	5
naive	**136.0**	190.0	184.0	179.9	170.1	169.6
active	**136.0**	188.0	180.4	176.1	164.9	164.4
robust	**136.0**	**166.8**	**164.4**	**163.9**	**160.3**	**159.9**

la07						
method	P	1	2	3	4	5
naive	**47.1**	102.6	96.3	91.3	83.5	82.8
active	**47.1**	101.4	94.9	90.1	82.1	81.5
robust	47.2	**76.0**	**73.7**	**73.3**	**73.3**	**73.0**

la08						
method	P	1	2	3	4	5
naive	**0.0**	51.1	45.8	41.9	29.6	28.6
active	**0.0**	22.3	20.1	19.2	**15.5**	**14.8**
robust	**0.0**	**20.6**	**19.0**	**19.0**	18.6	16.4

la09						
method	P	1	2	3	4	5
naive	**111.1**	168.9	164.5	159.6	142.4	141.5
active	**111.1**	170.9	165.5	159.8	**142.2**	**141.3**
robust	115.9	**153.4**	**149.8**	**148.8**	144.6	143.3

la10						
method	P	1	2	3	4	5
naive	**130.0**	181.7	175.5	171.6	167.9	167.7
active	**130.0**	178.9	172.3	168.2	164.8	164.7
robust	**130.0**	**165.0**	**163.4**	**163.7**	**163.4**	**163.0**

la11						
method	P	1	2	3	4	5
naive	**161.0**	222.7	218.6	213.2	198.3	197.8
active	**161.0**	224.4	220.1	215.3	198.2	197.7
robust	**161.0**	**193.1**	**190.0**	**188.3**	**185.0**	**184.5**

la12						
method	P	1	2	3	4	5
naive	**121.0**	182.3	177.0	173.2	161.8	161.6
active	**121.0**	182.1	177.1	174.0	164.0	163.9
robust	**121.0**	**174.8**	**167.7**	**162.3**	**158.9**	**158.2**

la13						
method	P	1	2	3	4	5
naive	**118.0**	178.9	175.3	172.4	157.9	157.3
active	118.1	179.5	175.9	172.8	158.5	157.6
robust	120.0	**164.2**	**160.9**	**160.2**	**152.5**	**152.0**

la14						
method	P	1	2	3	4	5
naive	**186.0**	236.1	229.0	223.1	211.7	211.4
active	**186.0**	233.7	227.1	222.2	211.7	211.6
robust	**186.0**	**209.0**	**206.4**	**205.5**	**205.5**	**205.4**

la15						
method	P	1	2	3	4	5
naive	63.2	117.8	111.7	107.7	101.8	101.4
active	**62.3**	113.6	108.4	105.5	**99.3**	**99.0**
robust	64.2	**104.4**	**102.3**	**101.8**	99.6	**99.0**

la16						
method	P	1	2	3	4	5
naive	52.5	**105.6**	**99.6**	**97.5**	**89.9**	**87.6**
active	**51.8**	108.0	102.6	100.2	93.6	91.3
robust	62.2	111.7	107.1	106.2	103.5	100.0

la17						
method	P	1	2	3	4	5
naive	**64.3**	**122.8**	**117.0**	**114.0**	**110.2**	**107.3**
active	65.1	123.4	117.9	115.9	111.9	110.0
robust	67.0	123.2	119.6	118.4	115.2	113.5

la18						
method	P	1	2	3	4	5
naive	69.6	123.9	117.8	113.7	103.9	101.4
active	**69.4**	126.2	121.2	117.1	105.9	103.2
robust	73.1	**115.0**	**110.9**	**107.5**	**103.3**	**101.2**

la19						
method	P	1	2	3	4	5
naive	**64.2**	121.9	**114.1**	112.0	**104.4**	**103.2**
active	**64.2**	**120.2**	114.3	**111.7**	105.5	104.2
robust	70.8	120.9	114.3	112.2	109.6	107.7

la20						
method	P	1	2	3	4	5
naive	**46.4**	93.0	86.9	84.9	79.6	79.1
active	**46.4**	91.4	84.9	82.0	**77.0**	76.4
robust	50.8	**85.3**	**81.3**	**80.5**	77.5	**76.0**

la21						
method	P	1	2	3	4	5
naive	0.5	56.4	51.0	48.4	38.5	35.4
active	**0.4**	49.8	44.0	41.3	33.3	31.3
robust	0.9	**44.6**	**40.2**	**38.3**	**31.6**	**29.0**

la22						
method	P	1	2	3	4	5
naive	**0.0**	50.8	43.4	38.8	27.1	25.4
active	**0.0**	33.9	29.2	26.3	**19.0**	**18.1**
robust	**0.0**	**30.9**	**27.7**	**26.0**	19.9	19.2

la23						
method	P	1	2	3	4	5
naive	17.0	67.9	60.9	56.8	47.7	45.8
active	**16.9**	70.9	64.8	59.9	51.4	49.5
robust	17.1	**54.8**	**51.2**	**49.7**	**45.6**	**43.5**

la24						
method	P	1	2	3	4	5
naive	**46.3**	104.8	99.5	96.7	**83.9**	**81.3**
active	46.9	106.7	101.8	98.4	85.1	82.6
robust	50.0	**102.3**	**97.9**	**95.9**	87.4	84.8

la25						
method	P	1	2	3	4	5
naive	67.5	123.4	116.4	111.8	98.5	96.6
active	**66.7**	120.3	114.0	109.8	**98.0**	**96.5**
robust	69.2	**113.2**	**108.8**	**107.1**	101.7	99.0

la26						
method	P	1	2	3	4	5
naive	**18.5**	73.9	66.8	61.2	47.1	44.9
active	18.6	72.2	65.2	59.5	45.0	43.2
robust	20.2	**62.3**	**56.5**	**53.7**	**44.7**	**43.1**

la27						
method	P	1	2	3	4	5
naive	27.1	87.6	80.7	75.8	61.8	59.8
active	**26.5**	83.1	76.9	71.8	**57.8**	**55.7**
robust	29.2	**76.2**	**71.3**	**68.3**	60.1	57.3

la28						
method	P	1	2	3	4	5
naive	34.4	92.6	86.5	81.0	68.0	66.0
active	**33.7**	91.1	84.9	80.1	**67.7**	**65.7**
robust	34.2	**80.7**	**75.8**	**73.8**	68.0	65.9

la29						
method	P	1	2	3	4	5
naive	49.7	104.0	97.7	93.3	82.7	80.6
active	50.3	107.0	100.7	97.0	85.5	83.6
robust	**47.7**	**92.8**	**88.1**	**85.9**	**81.2**	**80.2**

la30						
method	P	1	2	3	4	5
naive	14.2	64.7	58.1	54.6	46.2	44.7
active	14.3	65.1	58.2	53.0	44.8	41.1
robust	**12.4**	**51.2**	**46.6**	**45.3**	**42.9**	**39.4**

la31						
method	P	1	2	3	4	5
naive	**78.1**	129.1	121.5	114.5	104.3	103.7
active	**78.1**	127.9	121.4	114.7	106.1	105.6
robust	78.2	**105.1**	**101.6**	**99.9**	**96.8**	**96.6**

la32						
method	P	1	2	3	4	5
naive	32.1	94.1	88.5	82.9	56.0	55.2
active	33.6	94.3	87.8	82.8	55.3	55.5
robust	**20.3**	**70.5**	**65.8**	**62.1**	**40.9**	**40.1**

la33						
method	P	1	2	3	4	5
naive	103.3	157.0	150.8	145.1	131.2	128.3
active	102.7	154.5	148.0	142.8	125.8	125.4
robust	**86.1**	**127.9**	**124.6**	**122.7**	**114.9**	**113.5**

la34						
method	P	1	2	3	4	5
naive	145.0	193.4	185.0	176.6	166.5	166.0
active	144.2	193.2	184.4	175.7	167.1	167.9
robust	**140.7**	**166.5**	**163.6**	**162.2**	**158.7**	**158.7**

la35						
method	P	1	2	3	4	5
naive	161.5	196.9	189.7	184.1	177.4	176.9
active	162.0	199.1	190.0	183.8	176.2	175.9
robust	**161.2**	**175.4**	**173.3**	**172.5**	**170.2**	**170.2**

la36						
method	P	1	2	3	4	5
naive	86.8	136.7	130.1	125.4	117.1	114.6
active	**85.9**	135.2	127.6	123.8	**114.5**	**113.8**
robust	94.1	**128.6**	**124.0**	**122.2**	119.8	117.6

la37						
method	P	1	2	3	4	5
naive	**95.2**	141.7	134.1	**130.1**	**120.2**	**119.0**
active	96.9	142.0	135.0	130.5	122.4	120.7
robust	101.4	**137.9**	**132.9**	131.7	129.7	128.8

la38						
method	P	1	2	3	4	5
naive	**56.3**	111.0	104.2	98.7	86.4	85.2
active	56.7	110.9	103.2	98.3	**85.6**	84.5
robust	59.6	**99.7**	**94.2**	**92.0**	86.6	**84.1**

la39						
method	P	1	2	3	4	5
naive	36.5	89.8	83.2	78.5	**67.7**	**65.2**
active	**35.4**	90.4	83.2	78.0	68.7	65.7
robust	41.8	**83.5**	**78.2**	**76.4**	70.4	68.1

la40						
method	P	1	2	3	4	5
naive	99.5	157.6	148.8	143.4	129.4	126.9
active	**99.1**	156.9	147.7	**142.1**	**127.9**	**125.2**
robust	105.5	**151.9**	**146.3**	143.4	132.7	129.4

Appendix I

Results of total tardiness experiments

I.1 Loose problems $\sigma = 0.95$

Results for individual problems for the experiments of section 5.5.1.

ft10					
method	P	1	2	4	5
naive	**0.0**	162.1	116.7	32.7	29.0
active	**0.0**	63.5	47.3	**11.1**	**10.1**
robust	**0.0**	**48.3**	**33.4**	13.4	11.8

ft20					
method	P	1	2	4	5
naive	**0.0**	191.9	144.6	16.8	15.2
active	**0.0**	0.1	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la01					
method	P	1	2	4	5
naive	**0.0**	206.8	161.9	100.0	97.0
active	**0.0**	153.4	114.7	**79.0**	**76.7**
robust	**0.0**	**145.1**	**110.1**	81.7	80.0

la02					
method	P	1	2	4	5
naive	**0.0**	191.0	141.8	72.7	65.0
active	**0.0**	124.0	93.3	53.3	47.3
robust	**0.0**	**101.2**	**74.5**	**49.4**	**43.5**

la03					
method	P	1	2	4	5
naive	**0.0**	170.7	128.2	79.7	74.5
active	**0.0**	129.6	**102.4**	**64.9**	**61.4**
robust	**0.0**	**128.0**	108.9	82.0	76.7

la04					
method	P	1	2	4	5
naive	**0.0**	128.1	95.1	55.5	53.9
active	**0.0**	91.2	74.2	**49.0**	**47.3**
robust	**0.0**	**88.9**	**70.3**	**49.0**	47.9

la05					
method	P	1	2	4	5
naive	**82.3**	304.8	251.1	206.9	202.8
active	**82.3**	306.2	250.7	**199.0**	**195.5**
robust	85.6	**285.2**	**239.6**	207.3	206.6

la06					
method	P	1	2	4	5
naive	133.6	398.6	348.5	**251.3**	247.0
active	**133.4**	431.1	368.2	253.7	249.8
robust	139.6	**342.5**	**308.8**	252.4	**246.4**

la07					
method	P	1	2	4	5
naive	**0.0**	140.6	98.6	26.7	25.1
active	**0.0**	22.6	16.7	7.2	7.2
robust	**0.0**	**6.6**	**5.8**	**5.5**	**5.4**

la08					
method	P	1	2	4	5
naive	**0.0**	145.1	104.2	23.5	22.1
active	**0.0**	0.1	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la09					
method	P	1	2	4	5
naive	26.4	222.2	176.4	113.6	114.1
active	**26.1**	238.4	189.1	128.4	126.2
robust	29.4	**178.9**	**149.0**	**106.8**	**107.6**

la10					
method	P	1	2	4	5
naive	117.8	345.1	285.5	218.8	215.2
active	**115.7**	356.8	304.2	228.9	227.0
robust	128.1	**279.6**	**248.9**	**217.3**	**215.1**

la11					
method	P	1	2	4	5
naive	**65.7**	454.0	384.2	225.5	222.3
active	66.0	439.2	374.7	230.2	223.2
robust	67.3	**323.8**	**274.1**	**197.4**	**192.7**

la12					
method	P	1	2	4	5
naive	27.7	307.3	247.9	143.2	141.0
active	**27.5**	314.8	255.5	143.7	142.4
robust	29.6	**243.8**	**204.6**	**135.4**	**135.1**

la13					
method	P	1	2	4	5
naive	**0.0**	275.9	222.3	89.0	85.2
active	**0.0**	168.6	134.4	63.3	61.4
robust	**0.0**	**127.8**	**101.7**	**55.7**	**56.4**

la14					
method	P	1	2	4	5
naive	**215.0**	489.4	419.1	322.4	321.0
active	**215.0**	491.4	422.9	317.7	317.5
robust	215.4	**388.4**	**358.4**	**312.9**	**314.4**

la15					
method	P	1	2	4	5
naive	**0.0**	174.2	123.2	21.0	19.5
active	**0.0**	8.8	5.8	1.7	1.7
robust	**0.0**	**4.6**	**3.5**	**1.2**	**1.5**

la16					
method	P	1	2	4	5
naive	**0.0**	130.7	95.2	40.1	35.7
active	**0.0**	**47.8**	**32.6**	**17.8**	**15.0**
robust	0.2	51.7	38.2	26.8	24.8

la17					
method	P	1	2	4	5
naive	**0.0**	196.2	144.9	76.4	70.7
active	**0.0**	**143.3**	**115.1**	**65.0**	**58.2**
robust	0.2	149.8	121.6	76.8	72.7

la18					
method	P	1	2	4	5
naive	**0.0**	213.9	157.4	60.2	55.1
active	**0.0**	**85.0**	**63.2**	**25.7**	**24.4**
robust	4.6	94.0	70.7	42.3	40.9

la19					
method	P	1	2	4	5
naive	**0.0**	209.5	152.4	66.4	61.1
active	**0.0**	134.7	110.5	**51.3**	**49.0**
robust	0.1	**110.2**	**88.8**	54.9	51.7

la20					
method	P	1	2	4	5
naive	**0.0**	86.3	56.2	30.2	29.2
active	**0.0**	56.9	38.2	20.4	**20.1**
robust	0.1	**44.0**	**29.5**	**20.0**	20.4

la21					
method	P	1	2	4	5
naive	**0.0**	116.1	76.4	14.4	13.5
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la22					
method	P	1	2	4	5
naive	**0.0**	95.3	63.6	9.9	10.4
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la23					
method	P	1	2	4	5
naive	**0.0**	112.2	71.5	17.7	16.4
active	**0.0**	3.4	1.6	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la24					
method	P	1	2	4	5
naive	**0.0**	229.3	165.3	31.4	29.1
active	**0.0**	48.4	32.0	**6.4**	**5.0**
robust	**0.0**	**29.4**	**18.5**	6.8	5.1

la25					
method	P	1	2	4	5
naive	**0.0**	211.8	150.9	44.6	37.3
active	**0.0**	80.2	53.7	**15.3**	**15.1**
robust	**0.0**	**64.0**	**45.7**	20.0	21.0

la26					
method	P	1	2	4	5
naive	**0.0**	133.5	93.0	10.2	9.6
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la27					
method	P	1	2	4	5
naive	**0.0**	144.5	97.2	12.0	11.6
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la28					
method	P	1	2	4	5
naive	**0.0**	215.4	146.8	22.7	20.5
active	**0.0**	8.6	5.3	1.0	1.1
robust	**0.0**	**2.0**	**1.3**	**0.8**	**0.6**

la29					
method	P	1	2	4	5
naive	**0.0**	252.9	175.3	29.2	28.7
active	**0.0**	21.8	15.2	1.8	1.8
robust	**0.0**	**8.2**	**5.5**	**1.2**	**1.0**

la30					
method	P	1	2	4	5
naive	**0.0**	91.6	60.1	10.7	9.7
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la31					
method	P	1	2	4	5
naive	**0.0**	148.6	97.5	7.5	7.3
active	**0.0**	0.1	0.1	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la32					
method	P	1	2	4	5
naive	**0.0**	175.7	111.6	6.7	6.2
active	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
robust	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**

la33					
method	P	1	2	4	5
naive	**0.0**	217.9	154.9	12.3	11.1
active	**0.0**	10.5	6.6	0.5	0.6
robust	**0.0**	**1.5**	**1.1**	**0.3**	**0.3**

la34					
method	P	1	2	4	5
naive	**0.0**	251.2	168.4	27.2	26.1
active	**0.0**	50.5	30.9	7.3	7.4
robust	**0.0**	**17.2**	**13.2**	**4.3**	**4.5**

la35					
method	P	1	2	4	5
naive	0.1	147.5	100.4	18.8	19.8
active	0.1	40.5	28.4	12.7	13.4
robust	**0.0**	**23.2**	**19.0**	**11.7**	**13.0**

la36					
method	P	1	2	4	5
naive	**0.0**	133.0	85.8	24.9	20.3
active	**0.0**	43.8	27.9	**8.5**	**7.1**
robust	0.6	**37.3**	**26.9**	15.1	13.6

la37					
method	P	1	2	4	5
naive	0.4	136.0	88.2	17.6	13.7
active	**0.2**	45.8	30.2	**6.7**	**6.0**
robust	1.1	**24.2**	**18.7**	9.8	7.9

la38					
method	P	1	2	4	5
naive	1.4	152.2	104.3	20.3	18.6
active	**0.8**	23.1	15.1	**7.1**	7.6
robust	2.0	**16.1**	**10.7**	7.5	**7.1**

la39					
method	P	1	2	4	5
naive	**0.0**	110.7	68.4	11.6	10.8
active	**0.0**	2.1	1.2	0.5	0.5
robust	**0.0**	**0.4**	**0.2**	**0.2**	**0.1**

la40					
method	P	1	2	4	5
naive	0.3	250.4	172.1	37.8	31.0
active	**0.2**	129.4	89.1	**21.3**	**17.7**
robust	1.1	**98.8**	**70.1**	27.3	22.0

I.2 The tighter problems $\sigma = 0.85$

Results for individual problems for the experiments of section 5.5.2.

ft10					
method	P	1	2	4	5
naive	184.4	524.5	470.1	**319.4**	**304.9**
active	**183.2**	526.4	**462.7**	323.8	310.5
robust	215.5	**509.7**	466.6	356.5	340.9

ft20					
method	P	1	2	4	5
naive	**4.2**	296.0	235.5	85.0	78.6
active	5.1	238.1	191.6	76.0	71.0
robust	6.9	**177.9**	**144.6**	**65.9**	**65.4**

la01					
method	P	1	2	4	5
naive	**239.3**	505.7	**448.7**	**399.4**	**391.2**
active	**239.3**	525.9	459.0	405.9	399.2
robust	265.3	**504.7**	457.3	411.1	403.7

la02					
method	P	1	2	4	5
naive	**151.9**	530.4	434.3	302.9	292.1
active	152.0	517.7	430.6	**297.0**	**286.3**
robust	161.3	**463.1**	**391.0**	299.1	288.2

la03					
method	P	1	2	4	5
naive	177.0	459.6	409.6	334.7	321.2
active	**175.9**	443.4	398.6	329.7	**317.5**
robust	180.2	**409.2**	**370.4**	**328.5**	318.5

la04					
method	P	1	2	4	5
naive	**57.2**	**320.4**	**268.5**	**218.1**	**211.6**
active	57.8	340.2	286.3	224.0	216.4
robust	85.9	340.3	299.0	247.2	241.0

la05					
method	P	1	2	4	5
naive	383.0	652.6	597.9	**527.2**	**516.9**
active	**381.4**	664.8	606.8	541.9	530.3
robust	399.0	**640.4**	**593.7**	544.5	535.1

la06					
method	P	1	2	4	5
naive	691.6	1038.4	**968.8**	**829.7**	**812.4**
active	**686.8**	1043.9	976.9	831.1	816.3
robust	702.8	**1037.2**	979.5	863.9	843.7

la07					
method	P	1	2	4	5
naive	**48.1**	298.4	240.2	154.7	153.8
active	48.9	293.4	236.9	155.8	154.4
robust	50.1	**224.7**	**191.7**	**148.2**	**148.4**

la08					
method	P	1	2	4	5
naive	**0.0**	231.6	185.0	81.7	76.9
active	**0.0**	93.2	73.2	**43.5**	41.6
robust	**0.0**	**70.1**	**55.7**	**43.5**	**39.6**

la09					
method	P	1	2	4	5
naive	**473.8**	850.2	791.5	**672.4**	**658.2**
active	474.8	866.3	806.5	686.9	675.8
robust	495.5	**818.5**	**770.9**	679.4	666.4

la10					
method	P	1	2	4	5
naive	650.3	955.1	889.2	782.2	**764.5**
active	**649.1**	939.2	879.6	**779.6**	766.0
robust	654.1	**885.3**	**839.3**	784.0	776.0

la11					
method	P	1	2	4	5
naive	884.0	1368.4	1288.9	1105.5	1085.3
active	886.2	1348.5	1276.9	1107.8	**1083.8**
robust	**876.0**	**1301.6**	**1241.9**	**1105.3**	1089.7

la12					
method	P	1	2	4	5
naive	**635.5**	1107.9	1036.4	843.5	826.0
active	638.9	1082.4	1010.1	**831.6**	**811.2**
robust	646.6	**1055.7**	**999.0**	840.6	822.4

la13					
method	P	1	2	4	5
naive	633.3	1197.8	1125.1	**854.8**	839.5
active	636.0	1168.1	1094.5	859.4	**836.5**
robust	**627.6**	**1119.8**	**1056.1**	856.7	845.4

la14					
method	P	1	2	4	5
naive	1046.3	1423.9	1334.0	**1156.2**	**1147.2**
active	1042.2	1406.0	1322.4	1159.2	1151.2
robust	**1039.6**	**1350.1**	**1284.9**	1165.2	1159.3

la15					
method	P	1	2	4	5
naive	199.0	540.4	472.0	353.0	342.6
active	**193.2**	542.8	476.4	**347.8**	**336.2**
robust	206.2	**471.4**	**422.6**	349.2	344.4

la16					
method	P	1	2	4	5
naive	237.4	490.5	443.5	370.2	357.2
active	**232.2**	494.8	441.4	**366.2**	**355.2**
robust	282.2	**479.6**	**440.2**	404.2	396.9

la17					
method	P	1	2	4	5
naive	383.3	734.1	683.0	556.4	**527.3**
active	**380.1**	741.7	685.7	**553.5**	532.4
robust	420.3	**710.9**	**658.6**	577.5	553.8

la18					
method	P	1	2	4	5
naive	293.4	**587.0**	538.4	430.8	409.8
active	**284.8**	591.6	**536.9**	**421.4**	**404.4**
robust	323.4	605.1	556.8	461.2	444.3

la19					
method	P	1	2	4	5
naive	**323.4**	**636.5**	**580.9**	**480.3**	**462.4**
active	327.5	649.8	592.2	492.6	475.6
robust	341.1	640.3	594.5	512.4	500.4

la20					
method	P	1	2	4	5
naive	130.6	327.8	**276.1**	**224.9**	**223.6**
active	**124.9**	348.6	293.0	227.5	225.8
robust	144.5	**306.5**	278.2	240.5	237.0

la21					
method	P	1	2	4	5
naive	1.5	274.4	206.9	91.1	76.2
active	**1.2**	206.2	161.1	**72.2**	**62.2**
robust	5.0	**180.2**	**146.3**	86.8	76.8

la22					
method	P	1	2	4	5
naive	0.3	267.9	203.3	65.3	62.9
active	**0.1**	165.9	128.7	43.3	41.8
robust	1.6	**126.4**	**92.8**	**40.6**	**37.8**

la23					
method	P	1	2	4	5
naive	54.7	377.0	303.4	**177.1**	**163.7**
active	**54.5**	391.3	315.7	181.3	165.5
robust	67.7	**313.3**	**271.0**	191.3	179.3

la24					
method	P	1	2	4	5
naive	323.0	843.0	752.5	**518.9**	507.8
active	**310.2**	823.0	733.1	521.2	**493.6**
robust	338.4	**768.6**	**695.6**	520.9	499.8

la25					
method	P	1	2	4	5
naive	311.9	733.9	648.8	490.2	471.0
active	**296.5**	701.7	616.2	**479.0**	**463.7**
robust	307.3	**666.2**	**610.6**	489.5	476.1

la26					
method	P	1	2	4	5
naive	**48.3**	481.7	386.2	153.1	139.8
active	50.7	462.1	376.1	**146.3**	**134.5**
robust	61.6	**356.0**	**299.3**	159.2	143.9

la27					
method	P	1	2	4	5
naive	**88.5**	607.6	499.6	**241.5**	228.3
active	89.6	628.9	525.6	242.5	**226.0**
robust	98.8	**555.4**	**481.9**	257.3	244.2

la28					
method	P	1	2	4	5
naive	158.0	670.0	574.4	**307.0**	**292.0**
active	156.2	666.5	569.3	323.8	300.6
robust	**155.1**	**558.8**	**484.9**	308.2	297.1

la29					
method	P	1	2	4	5
naive	475.5	1071.4	949.0	**635.5**	**594.9**
active	465.5	1079.9	957.1	649.5	614.3
robust	**455.5**	**1008.1**	**906.5**	647.3	608.5

la30					
method	P	1	2	4	5
naive	72.7	462.5	365.7	200.2	187.5
active	74.5	477.7	389.7	206.9	191.8
robust	**63.6**	**323.7**	**275.2**	**180.1**	**174.7**

la31					
method	P	1	2	4	5
naive	541.2	1213.9	1053.2	706.7	687.6
active	541.1	1263.3	1105.4	709.3	709.8
robust	**433.6**	**847.5**	**760.2**	**588.0**	**576.0**

la32					
method	P	1	2	4	5
naive	116.0	763.4	630.0	206.5	193.3
active	115.9	759.5	626.9	208.9	190.8
robust	**48.6**	**421.0**	**345.2**	**115.5**	**112.9**

la33					
method	P	1	2	4	5
naive	1135.0	1993.8	1828.8	1327.6	1260.6
active	1140.2	1948.9	1784.7	1322.5	1252.0
robust	**805.8**	**1472.4**	**1352.3**	**1012.1**	**1002.0**

la34					
method	P	1	2	4	5
naive	1597.5	2295.0	2113.2	1626.4	1589.2
active	1575.4	2307.1	2131.6	1622.7	1598.8
robust	**1365.7**	**1838.1**	**1733.9**	**1470.9**	**1443.2**

la35					
method	P	1	2	4	5
naive	1342.7	1976.8	1793.4	1376.3	1345.3
active	1354.0	1959.7	1797.7	1369.7	1353.9
robust	**1067.9**	**1469.6**	**1368.5**	**1126.5**	**1099.6**

la36					
method	P	1	2	4	5
naive	**470.2**	**834.5**	**752.6**	**625.3**	**604.9**
active	483.0	851.5	766.7	645.1	622.3
robust	546.3	838.1	775.0	683.2	657.8

la37					
method	P	1	2	4	5
naive	**597.5**	949.8	858.1	**712.7**	683.5
active	602.4	948.0	863.5	717.0	**681.3**
robust	664.7	**915.8**	**856.9**	751.6	718.4

la38					
method	P	1	2	4	5
naive	**349.1**	801.4	706.8	**508.3**	483.9
active	350.4	774.1	680.4	512.0	**482.0**
robust	372.1	**705.5**	**642.9**	523.8	492.9

la39					
method	P	1	2	4	5
naive	142.3	527.2	453.2	279.8	**258.9**
active	**130.1**	523.8	449.4	**276.6**	262.3
robust	167.6	**477.2**	**418.1**	314.6	288.5

la40					
method	P	1	2	4	5
naive	681.7	1268.8	1158.9	**874.8**	**852.9**
active	**680.2**	1249.3	1138.7	880.3	861.5
robust	702.9	**1192.4**	**1099.3**	903.3	878.2

Appendix J

Results of total flow-time experiments

Results for individual problems for the experiments of section 5.6.

ft10					
method	P	1	2	4	5
active	7866.3	8274.0	8207.5	8061.4	8049.4
robust	**7844.8**	**8241.9**	**8176.6**	**8050.6**	**8033.4**

ft20					
method	P	1	2	4	5
active	15204.0	15968.1	15869.3	15480.1	15427.2
robust	**15155.9**	**15852.5**	**15755.3**	**15428.0**	**15393.2**

la01					
method	P	1	2	4	5
active	**5056.5**	5415.9	5361.8	**5260.3**	**5256.5**
robust	5096.7	**5392.2**	**5345.8**	5283.6	5277.9

la02					
method	P	1	2	4	5
active	**4667.7**	**5053.4**	**5004.1**	**4886.1**	**4864.7**
robust	4683.9	5069.1	5016.1	4902.6	4894.4

la03					
method	P	1	2	4	5
active	**4257.6**	4621.4	4577.8	**4479.3**	**4462.8**
robust	4277.0	**4619.2**	**4575.5**	4502.4	4486.6

la04					
method	P	1	2	4	5
active	4575.6	4950.1	4901.0	4799.5	4772.9
robust	**4574.1**	**4910.1**	**4868.6**	**4787.0**	**4756.7**

la05					
method	P	1	2	4	5
active	**4305.1**	4704.5	4643.0	**4497.6**	**4484.2**
robust	4324.5	**4682.1**	**4633.5**	4525.5	4516.1

la06					
method	P	1	2	4	5
active	**9591.3**	10119.8	10041.3	**9827.6**	**9801.1**
robust	9617.0	**10087.9**	**10012.3**	9834.6	9822.8

la07					
method	P	1	2	4	5
active	**8731.3**	9273.0	**9197.6**	**9004.2**	**8979.6**
robust	8756.5	**9256.7**	9197.9	9027.4	9001.9

la08					
method	P	1	2	4	5
active	**8944.9**	9503.3	9425.7	**9208.3**	9184.4
robust	8953.0	**9483.1**	**9409.3**	9216.3	**9183.6**

la09					
method	P	1	2	4	5
active	**10086.4**	10679.8	10591.4	**10332.9**	**10294.1**
robust	10113.2	**10660.9**	**10576.4**	10377.0	10345.1

la10					
method	P	1	2	4	5
active	**9912.7**	10493.0	10409.9	**10148.4**	**10121.6**
robust	9966.0	**10474.9**	**10395.1**	10195.9	10181.9

la11					
method	P	1	2	4	5
active	16062.1	16792.4	16688.5	16256.4	16216.5
robust	**16017.9**	**16702.7**	**16597.3**	**16210.8**	**16179.1**

la12					
method	P	1	2	4	5
active	13864.4	14624.3	14507.4	14103.6	14070.5
robust	**13785.1**	**14481.6**	**14384.6**	**14068.1**	**14049.6**

la13					
method	P	1	2	4	5
active	15006.2	15783.7	15678.9	15308.2	15233.8
robust	**14927.2**	**15575.2**	**15486.9**	**15234.2**	**15204.6**

la14					
method	P	1	2	4	5
active	16872.6	17649.3	17517.7	**17032.8**	**16963.0**
robust	**16836.7**	**17419.2**	**17313.2**	17041.0	16982.8

la15					
method	P	1	2	4	5
active	16533.6	17259.8	17133.2	16736.3	**16646.7**
robust	**16508.1**	**17158.2**	**17066.5**	**16719.7**	16666.8

la16					
method	P	1	2	4	5
active	**7680.4**	8031.5	7968.6	**7867.7**	**7847.2**
robust	7699.0	**8012.4**	**7964.3**	7889.9	7869.1

la17					
method	P	1	2	4	5
active	**6784.9**	7192.1	7116.0	**6961.4**	**6930.6**
robust	6804.1	**7134.0**	**7063.1**	6964.3	6938.6

la18					
method	P	1	2	4	5
active	**7247.0**	**7632.9**	**7572.5**	**7438.5**	**7412.7**
robust	7261.5	7652.6	7591.9	7464.5	7446.2

la19					
method	P	1	2	4	5
active	**7469.9**	7900.5	7823.7	**7680.6**	**7649.7**
robust	7474.8	**7882.6**	**7812.0**	7693.9	7673.1

la20					
method	P	1	2	4	5
active	**7660.5**	**8032.6**	**7966.9**	**7833.1**	**7815.7**
robust	7682.1	8043.7	7987.4	7849.2	7839.2

la21					
method	P	1	2	4	5
active	**13466.4**	14073.3	13969.3	**13723.9**	**13682.0**
robust	13492.3	**14016.5**	**13933.0**	13724.9	13702.4

la22					
method	P	1	2	4	5
active	**12445.0**	13024.4	12920.7	**12678.3**	12651.6
robust	12453.6	**12949.8**	**12859.6**	12700.2	**12650.4**

la23					
method	P	1	2	4	5
active	**13331.5**	13914.0	13819.2	**13588.8**	**13539.9**
robust	13377.4	**13863.8**	**13780.1**	13605.4	13580.4

la24					
method	P	1	2	4	5
active	**12806.8**	13392.9	13307.9	13032.0	12985.5
robust	12830.3	**13337.8**	**13255.2**	**13018.5**	**12983.7**

la25					
method	P	1	2	4	5
active	12575.5	13168.6	13061.8	12838.9	**12788.0**
robust	**12574.2**	**13077.1**	**12998.0**	**12828.2**	12800.0

la26					
method	P	1	2	4	5
active	**21128.7**	21911.8	21781.6	**21373.9**	**21290.2**
robust	21187.1	**21820.0**	**21712.9**	21428.5	21361.9

la27					
method	P	1	2	4	5
active	21580.2	22419.1	22270.9	21801.0	**21696.7**
robust	**21564.1**	**22231.8**	**22116.9**	**21777.4**	21710.7

la28					
method	P	1	2	4	5
active	21460.8	22220.4	22081.2	21682.7	**21606.8**
robust	**21446.2**	**22121.6**	**22009.7**	**21663.9**	21607.5

la29					
method	P	1	2	4	5
active	20283.8	21056.9	20909.2	20537.2	20434.2
robust	**20250.9**	**20889.3**	**20769.8**	**20452.6**	**20376.9**

la30					
method	P	1	2	4	5
active	21765.6	22535.2	22392.5	21995.4	21891.9
robust	**21706.4**	**22374.0**	**22262.7**	**21927.6**	**21832.9**

la31					
method	P	1	2	4	5
active	41242.2	42379.4	42174.6	41392.2	41255.2
robust	**41082.1**	**42021.9**	**41853.3**	**41257.2**	**41139.5**

la32					
method	P	1	2	4	5
active	44815.5	45910.6	45711.4	44933.0	44822.4
robust	**44550.5**	**45503.2**	**45340.5**	**44766.8**	**44628.3**

la33					
method	P	1	2	4	5
active	40823.4	41973.1	41767.0	40977.0	40880.6
robust	**40641.3**	**41532.8**	**41380.1**	**40795.6**	**40714.8**

la34					
method	P	1	2	4	5
active	42366.6	43487.7	43280.8	42484.3	42373.6
robust	**42142.9**	**43097.7**	**42933.1**	**42295.2**	**42216.6**

la35					
method	P	1	2	4	5
active	42706.5	43762.0	43575.0	42687.4	42508.7
robust	**42413.2**	**43332.9**	**43184.0**	**42484.1**	**42305.1**

la36					
method	P	1	2	4	5
active	**17379.7**	17931.1	17826.0	**17588.8**	**17541.8**
robust	17433.6	**17869.1**	**17791.0**	17613.3	17579.2

la37					
method	P	1	2	4	5
active	**18391.5**	18970.0	**18842.5**	**18599.8**	**18551.0**
robust	18424.9	**18947.8**	18847.8	18639.5	18593.9

la38					
method	P	1	2	4	5
active	**16423.2**	**16985.8**	**16884.2**	**16656.1**	**16627.4**
robust	16474.0	17012.3	16912.3	16694.6	16639.1

la39					
method	P	1	2	4	5
active	**16466.9**	17083.3	16966.9	**16681.0**	**16646.0**
robust	16474.0	**17011.7**	**16922.5**	16709.1	16675.6

la40					
method	P	1	2	4	5
active	16678.4	17326.7	17214.2	**16886.1**	**16834.5**
robust	**16674.4**	**17252.1**	**17145.9**	16893.1	16847.1

Recent BRICS Dissertation Series Publications

DS-01-10 **Mikkel T. Jensen.** ***Robust and Flexible Scheduling with Evolutionary Computation*****. November 2001. PhD thesis. xii+299 pp.**

DS-01-9 **Flemming Friche Rodler.** ***Compression with Fast Random Access*****. November 2001. PhD thesis. xiv+123 pp.**

DS-01-8 **Niels Damgaard.** ***Using Theory to Make Better Tools*****. October 2001. PhD thesis.**

DS-01-7 **Lasse R. Nielsen.** ***A Study of Defunctionalization and Continuation-Passing Style*****. August 2001. PhD thesis. iv+280 pp.**

DS-01-6 **Bernd Grobauer.** ***Topics in Semantics-based Program Manipulation*****. August 2001. PhD thesis. ii+x+186 pp.**

DS-01-5 **Daniel Damian.** ***On Static and Dynamic Control-Flow Information in Program Analysis and Transformation*****. August 2001. PhD thesis. xii+111 pp.**

DS-01-4 **Morten Rhiger.** ***Higher-Order Program Generation.*** **August 2001. PhD thesis. xiv+146 pp.**

DS-01-3 **Thomas S. Hune.** ***Analyzing Real-Time Systems: Theory and Tools*****. March 2001. PhD thesis. xii+265 pp.**

DS-01-2 **Jakob Pagter.** ***Time-Space Trade-Offs*****. March 2001. PhD thesis. xii+83 pp.**

DS-01-1 **Stefan Dziembowski.** ***Multiparty Computations — Information-Theoretically Secure Against an Adaptive Adversary*****. January 2001. PhD thesis. 109 pp.**

DS-00-7 **Marcin Jurdziński.** ***Games for Verification: Algorithmic Issues*****. December 2000. PhD thesis. ii+112 pp.**

DS-00-6 **Jesper G. Henriksen.** ***Logics and Automata for Verification: Expressiveness and Decidability Issues*****. May 2000. PhD thesis. xiv+229 pp.**

DS-00-5 **Rune B. Lyngsø.** ***Computational Biology*****. March 2000. PhD thesis. xii+173 pp.**

DS-00-4 **Christian N. S. Pedersen.** ***Algorithms in Computational Biology*****. March 2000. PhD thesis. xii+210 pp.**