Universiteit Utrecht

Master's thesis (ICA-3220583)

Robustness in parallel machine scheduling

Author:
D.J. van Roermund

Supervisors:
dr. J.A. Hoogeveen
dr. ir. J.M. van den Akker

October 28, 2013

Abstract

In parallel machine scheduling with precedence relations between jobs, the delay of a single task could hinder several other machines. The traditional method for avoiding this situation involves adding idle time between jobs that are dependent on each other, to avoid having to postpone the scheduled execution of successor jobs when the predecessor is delayed. In this thesis we look at a previously introduced alternative definition of robustness, which measures the chances of the propagation of these delays. By minimising this metric, we minimise the possibilities of machines having to wait for others because of a breakdown. Note that a machine that has to wait for itself to complete a task after a breakdown is inevitable.

To explore the effectiveness of this approach, we introduce the concept of fixations, defined such that the jobs involved in a fixated relation are to be scheduled on the same machine. This ensures that these fixated relation can not attribute to the propagation of delays, as their jobs are not on different machines. By maximising the number of fixated relations then, we are minimising the chances of delay propagations, and thereby maximising the robustness of our schedules.

To simplify the implementation of our approach, we start by relaxing the precedence relations to mere correlations, dropping the partial ordering of the jobs. These correlations still signify a connection between two jobs, which we would like to have executed by the same machine, although the order in which this is done is irrelevant. Later we also add proper precedence relations, and include additional correlations to improve the robustness measure.

For finding solutions using our objective of maximising the number of fixated relations, we introduce two local search algorithms: one based on a genetic algorithm and one using simulated annealing. As the additional constraints that demand pairs of jobs are scheduled on the same machine might make it impossible to actually construct a schedule, we have to check every solution that is generated. For quickly evaluating this feasibility of a solution, we define a linear programming formulation that is incrementally solved using column generation. Furthermore we present several extensions to the problem category, to deal with release dates, deadlines and weighted relations. Finally we present another search method that combines the columns generated for the evaluation linear program into new solutions.

After finalising the approach and discussing the experiments used to test its effectiveness, we conclude that the presented heuristic is a very good method for solving larger problems with many machines. It is therefore an excellent complement to the integer linear programming formulation that solves instances to optimality, which is more efficient in solving problems with only a few machines, but has difficulties finding solutions for those with more machines, because of the symmetry in its definition.

Contents

Chapter 1

Introduction

Scheduling problems have a wide range of applications, from designing personnel rosters to generating efficient routes for deliveries. However, a lot of the practical and useful scheduling problems are very hard to solve in a short amount of time, as the number of possible solutions is often enormeous. This is where the field of linear optimisation, or linear programming, comes in. By formulating the problem as an objective function and a set of linear constraints, given some variables, we can solve problems for which listing all permutations of values is clearly intractable.

When we have generated a schedule or roster for such a problem, i.e. an allocation of tasks to machines at specific times, its *robustness* refers to the ability of handling failures or disturbances within the execution of that schedule, for example when a task is delayed. Further delays caused by this are traditionally prevented by adding idle time between tasks that are dependent on each other, which has the downside of increasing the total length of the resulting schedule, even if nothing goes wrong. The trade-off is therefore obvious — we would need to find the right balance between the risks of delays and the maximum duration of the schedule.

In this work we use a different method to avoid delays, by minimising the chance of their propagation. This introductory chapter starts by describing the problems we are looking at, followed by a guide through the previous work this thesis builds on. The last section sketches the outline of the rest of this report.

1.1 Problem definition

We work with problems that have multiple identical machines and a predefined set of jobs. These jobs are characterised by their processing times, release dates and deadlines, and there are precedence relations between them. The objective is to maximise the robustness after minimising the total makespan of the schedules. Using the three-field notation introduced by Graham et al. (1979), this problem category is denoted as:

$$\mathrm{P}|r_j, \bar{d}_j, \mathrm{prec}|\epsilon(C_{\max}, \sum -f_r).$$

Starting from the beginning, this means that a problem instance has m machines, which are:

- working in parallel, completely independent of each other;
- continuously available from time zero onwards — although we are maximising the robustness of the resulting schedules, we do not take the failure of specific machines into account while scheduling;
- equally able to carry out all of the available jobs with the same speed, i.e. jobs of equal length will take the same amount of time regardless of which machine executes them.

The collection of jobs is denoted as $J_1, \ldots, J_n$, with each job j having its own processing time p_j, signifying the amount of time it has to be executed uninterruptedly by one arbitrary machine. For the sake of simplicity, we assume all the given processing times are integral, although extending the formulations that are presented to allow for any positive real number is straightforward. In later chapters we also include the availability of jobs — stating that a job is not to be started before its release date r_j, and barring any disruptions due to breakdowns should be completed by its deadline $\bar{d}_j$ — although initially these are not taken into account.

The jobs are accompanied by a collection of precedence relations, each stating that there should be a given amount of time between the starting times of a pair of jobs, thus imposing a partial ordering on them. The included relations are of the form $S_j - S_i \geq q_{ij}$, with $q_{ij} \geq 0$, forcing job j to start at least q_{ij} time units after job i is started. Within the context of such a relation, job i is called the *predecessor* of j, as it has to be executed before j, just like j is the *successor* of i.

The composition of the objective function and the measurement of the robustness is explained in the following sections. Note that even when this problem is simplified to $\mathrm{P}|p_j = 1, \mathrm{prec}|C_{\max}$, i.e. ignoring the robustness, using unit processing times and leaving out the release dates and deadlines, deciding whether a schedule with $C_{\max} \leq 3$ exists is NP-complete, as proven by Lenstra et al. (1978). This furthermore means that no polynomial approximation algorithms with a worst case bound ρ smaller than $\frac{4}{3}$ can exist, unless $P = NP$. If there would be one, we could apply that heuristic to our instance; if a solution with $C_{\max} = 3$ would exist, the algorithm should return a value lower than four, as $3\rho < 4$. That would allow us to solve the decision variant of this NP-complete problem in polynomial time, meaning we could prove $P = NP$, which is generally believed not to hold.

1.2 Measuring robustness

Hoppenbrouwer (2011) introduced a notion of a-priori robustness of schedules, that is defined in terms of the chances of the propagation of delays. Delay propagation occurs whenever two jobs involved in a precedence relation are scheduled on different machines, and there is not enough idle time to avoid delaying the successor when the execution of the predecessor is impeded. To make matters worse, those delayed successors might be predecessors on their own, leading to a cascade of delayed jobs. With more relations spanning machines like this, the chances of having to postpone jobs in order to keep fulfilling the constraints increases. It is therefore worthwhile to minimise the number of precedence relations of which the jobs are on different machines.

However, simply counting the number of machine spanning relations, i.e. relations of which the jobs are scheduled on different machines, will not do. As he argues in his thesis, such an objective — resembling a graph partitioning problem — does not fully capture the essence of delay propagation. Consider the small example consisting of three jobs and two relations — $(J_1 \to J_2)$ and $(J_1 \to J_3)$ — for which figure 1.1 shows two possible schedules.

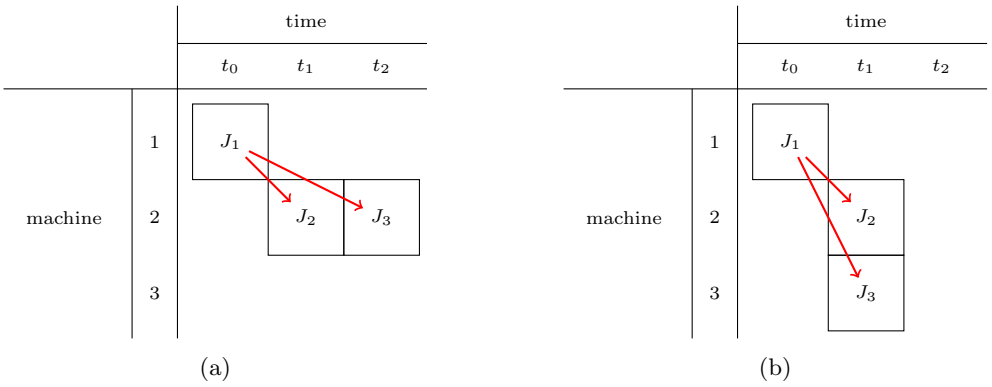


Figure 1.1: Two example schedules with three jobs and two relations.

In both examples, whenever the first machine breaks down and J_1 is delayed, both J_2 and J_3 will have to be postponed — the number of relations with jobs on different machines is 2 in either case. However, the assignment of schedule 1.1a implies a relation from J_2 to J_3. And since these precedence relations are transitive, we can replace $(J_1 \to J_3)$ with $(J_2 \to J_3)$, setting the value of q_{23} to $q_{13} - q_{12}$, without losing any information. This transformation reduces the number of machine spanning relations to only 1 in the first example, as the implied relation is entirely on the second machine. This makes sense from the viewpoint of delay propagation too, as the delay of J_1 can only hinder the jobs on one other machine in that example, as opposed to the two machines that are affected in the second example schedule.

1.2.1 Defining the metric

Hoppenbrouwer therefore proposed a measure that counts, for each job, the number of machines that its successors are scheduled on. For any machine $k \in \{1, \ldots, m\}$ and any job $j \in \{1, \ldots, n\}$, γ_{jk} measures the possibility of a delay propagation towards machine k, in case job j is delayed. Note that for every job j that is not a predecessor in any relation, γ_{jk} will be 0 for all values of k:

$$\gamma_{jk} = \begin{cases} 1 & \text{if job } j \text{ itself is not scheduled on machine } k, \text{ but has at least one successor that is;} \\ 0 & \text{otherwise.} \end{cases}$$

k \ j	1	2	3
1	0	0	0
2	1	0	0
3	0	0	0

(a) Values of γ_{jk} for schedule 1.1a.

k \ j	1	2	3
1	0	0	0
2	1	0	0
3	1	0	0

(b) Values of γ_{jk} for schedule 1.1b.

Table 1.1: Robustness measure of the example schedules of figure 1.1.

Table 1.1 demonstrates the values of γ for the example schedules of figure 1.1, which show that example 1.1a has only one possibility of delay propagation, whereas the other has two — precisely what we concluded earlier. The objective of maximising robustness in this way can thus be formulated as to minimise the sum of all these values, written as $\sum \gamma$ for short:

$$\min \sum_{j=1}^{n} \sum_{k=1}^{m} \gamma_{jk}.$$

1.2.2 Summary of approaches

The problem that Hoppenbrouwer tackled is denoted as $\mathrm{P}|\mathrm{prec}|\mathrm{Lex}(C_{\max}, \sum \gamma)$, where the objective is to first minimise the total makespan and then to minimise $\sum \gamma$ — see the work of Hoogeveen (2005) for more information on this lexicographical order and other compound objectives functions. Note that release dates and deadlines of jobs were not included.

He implemented three methods for solving this problem — an integer linear programming formulation, a constraint programming approach, and a linear programming problem combined with column generation. The first approach was an adaptation of the time indexed ILP presented by Van den Akker et al. (2012), with an added machine index. This formulation suffered from a lot of symmetry, as the solver could swap the machine indices of all tasks assigned to any two machines, attaining a seemingly different solution that essentially represented the exact same schedule. Solving this to optimality was therefore infeasible for larger problem instances.

The second technique made use of the constraint programming possibilities of the CPLEX suite by IBM (2009). This formulation ran much faster than the ILP, although it proved difficult

to come close to optimal solutions. The results were good enough to be used as a starting point for the other two approaches, speeding up their search, using this method as a heuristic.

To counter the symmetry of the ILP formulation, his third approach was to solve the LP relaxation using column generation. The problem was rewritten to make use of machine schedules, which represent collections of jobs that are executed by one machine. The challenge was then to find a suitable combination of these machine schedules, such that every job was executed exactly once. Unfortunately, it proved hard to find good solutions, requiring a lot of columns to be generated, making this approach still very time-consuming. His conclusions were therefore that while $\sum \gamma$ provided a useful measure for the robustness of schedules, directly minimising it with the presented methods was too inefficient — a different approach had to be found.

1.3 Alternative approach: fixations

Because the approaches for directly minimising $\sum \gamma$ did not achieve the results hoped for, we conceived another technique for maximising the robustness of schedules. By forcing the jobs of a relation to be executed on the same machine, we prevent that relation from impairing the objective.

In the previous section we introduced $\sum \gamma$ as a shorthand for $\sum_{j=1}^{n} \sum_{k=1}^{m} \gamma_{jk}$, which is by definition no larger than the number of machine spanning relations. Forcing the jobs of some relation to be scheduled on the same machine ensures that this relation is not spanning multiple machines. Thus such a fixation of relations provides an upper bound on the number of relations with jobs on different machines, i.e. the total number of relations minus the number of fixated ones. With an upper bound on the number of machine spanning relations we also have an upper bound on the number of possible delay propagations — $\sum \gamma$ is no larger than the number of relations that were not fixated. To minimise $\sum \gamma$, we maximise the number of fixated relations, under the condition that we can still create a schedule with the added constraints.

This approach of fixating relations was investigated by Van Roermund (2013), albeit on the simplified problem category in which the precedence relations were reduced to correlations between jobs, i.e. in which the constraints on partial ordering were dropped. These correlations indicate that it can be advantageous to schedule the involved jobs on the same machine, but the ordering of the jobs involved in these relations is not important. An example of such a problem would be the scheduling of calculations over several computers, which could benefit from cached sub-results of other tasks. It would be best to assign tasks that share parts of their input to the same machine, as that would improve the quality of the schedule, although it is in no way required.

The simplification greatly reduced the complexity of the implementation, as the ordering of tasks was no longer important — i.e. a schedule for a single machine could simply be represented by a set of tasks, without keeping track of their starting times — and allowed us to confirm the effectiveness of this approach. A precise definition of these fixations and the objective term $\sum -f_r$ follows in the next chapter.

1.4 Outline of this work

In this thesis we expand on the approach of maximising the robustness by fixating relations in four parts. The first part describes the method in full detail — still using the simplification of the correlations that impose no partial ordering on the jobs — starting with an overview and all necessary definitions in chapter 2. The methods used to evaluate the feasibility of these fixations are explained in chapter 3, and the top-level search for fixations in chapter 4. In the second part we introduce several extensions to the approach. Additions to the job characteristics of the problems are presented in chapter 5, the objective function is augmented in chapter 6, and chapter 7 shows an additional method to search for solutions. In the third part we present the results of this thesis. The values of the various parameters are set in chapter 8, the outcome of the experiments are found in chapter 9, and the conclusion, including pointers for further research, is in chapter 10. Lastly, the appendices in the fourth part contain the tables with the data from the experiments.

Part I

Basic approach

Chapter 2

Overview of the approach

This chapter explains how we employ fixations to tackle the simplified problem category, in which the precedence relations are relaxed to correlations, dropping the constraints on partial ordering. In the first section we describe how to tackle the composed objective function, followed by the formal definition of the fixations. The third section details how we can solve the problem to optimality, and lastly we give an overview of the heuristic approach that we present in this work.

2.1 Minimal maximum makespan

The objective function of the problem category we are solving — $\epsilon(C_{\max}, \sum -f_r)$ — defines a Pareto frontier. To find a Pareto optimal solution, we first apply the first criterion while completely ignoring the second, and then use the found objective value to limit solutions while using solely the second criterion. We thus first solve $\mathrm{P}||C_{\max}$ to find a schedule with minimal total makespan while completely ignoring its robustness. We do this using an adaptation of the constraint programming implementation written by Hoppenbrouwer (2011). This makespan is then used to limit the length of schedules in all other formulations, i.e. while subsequently looking for fixations with greatest robustness, we only allow rosters that are no longer than the found makespan.

To investigate how the robustness increases when allowing the makespan to grow, we will repeat the searches for fixations with several increased values of this upper bound on the makespan, thereby exploring the Pareto frontier. Throughout the rest of this work, we refer to the value of the current upper bound as C, and use it as a given parameter along with the problem instance. In this first part of the thesis we are thus solving $\mathrm{P}|C_{\max} \leq C|\sum -f_r$.

2.2 Defining fixations

To ensure that a specific relation does not contribute to the number of machine spanning relations, we fixate it, i.e. we demand that both jobs involved in that relation are scheduled on the same machine. This way, as far as that relation is concerned, the predecessor job can not propagate a delay to another machine. For every relation $r \in R$ — we use R to represent the set of relations defined by the problem instance — we introduce the binary parameter f_r, indicating whether the two jobs involved in that relation must be scheduled on the same machine:

$$f_r = \begin{cases} 1 & \text{the jobs involved in relation } r \text{ are to be scheduled on the same machine;} \\ 0 & \text{otherwise.} \end{cases}$$

Note that $f_r = 0$ does not mean that the jobs in that relation should be on separate machines, it simply means we allow them to be rostered freely. This parameter is then used as a variable during the search for the best fixation — from here on we use *fixation* to denote an assignment of f_r for all $r \in R$. A fixation is considered to be better than another if it has more relations fixated,

as that gives a tighter bound on $\sum \gamma$. During our search for the best fixation, the objective is thus to maximise the number of fixated relations, i.e. $\max \sum_{r \in R} f_r$. Since the three-field notation by Graham et al. (1979) assumes a minimisation objective, we multiply it by -1 and write $\sum -f_r$ for short. Although our objective approximates $\sum \gamma$, there are situations in which these metrics assess schedules differently — more on this in section 6.2.

2.3 Finding optimal solutions

Given the variables and objective from the previous section, we can define a straightforward integer linear programming formulation that finds an optimal fixation, i.e. that searches for a schedule which has the most relations fixated as possible, while remaining feasible. Its formulation has the binary variables x_{jk} to indicate whether job j is to be executed by machine k, and the variables y_{ij} — mimicking the parameter f_r — indicating whether the jobs i and j are to be executed on the same machine. Since relations consist of two jobs, we denote them as $r : (i, j)$, indicating relation r between predecessor i and successor j. We thus define:

$$x_{jk} = \begin{cases} 1 & \text{job } j \text{ is scheduled on machine } k; \\ 0 & \text{otherwise;} \end{cases}$$

$$y_{ij} = \begin{cases} 1 & \text{jobs } i \text{ and } j \text{ must be scheduled on the same machine;} \\ 0 & \text{otherwise.} \end{cases}$$

The objective, as explained before, is to maximise the number of fixated relations, and the constraints ensure that the resulting schedule is feasible, i.e. can actually be executed. In order, they state that every job has to be scheduled exactly once (2.1), that the combined processing times of the jobs scheduled on one machine can not exceed the given maximum makespan (2.2), and that the jobs of fixated relations have to be scheduled on the same machine (2.3 & 2.4).

$$\begin{aligned}
\max \quad & \sum_{r:(i,j) \in R} y_{ij} \quad \text{s.t.} \\
& \sum_{k=1}^{m} x_{jk} = 1 && \forall j \in \{1, \ldots, n\} && (2.1) \\
& \sum_{j=1}^{n} x_{jk} p_j \leq C && \forall k \in \{1, \ldots, m\} && (2.2) \\
& x_{ik} - x_{jk} \leq 1 - y_{ij} && \forall k \in \{1, \ldots, m\} \ \forall r : (i, j) \in R && (2.3) \\
& x_{jk} - x_{ik} \leq 1 - y_{ij} && \forall k \in \{1, \ldots, m\} \ \forall r : (i, j) \in R && (2.4) \\
& x_{jk} \in \{0, 1\} && \forall j \in \{1, \ldots, n\} \ \forall k \in \{1, \ldots, m\} \\
& y_{ij} \in \{0, 1\} && \forall r : (i, j) \in R
\end{aligned}$$

2.4 Decomposition approach

Because of the machine indices, the ILP formulation presented above suffers from a lot of symmetry, especially with problem instances containing many machines. Using it to solve a problem to optimality can therefore take a lot of time, making this formulation not efficient enough. To come to a better approach, we decompose it into two levels. On the top level, we use local search algorithms to find the best fixations, and on a supporting level we define a linear programming formulation to determine the feasibility of fixations found by the search algorithms.

The complete approach for tackling a problem thus looks as follows. First, we determine the upper bound on the makespan, as described above. We then invoke the search algorithms that

are detailed in chapter 4. Every time a new fixation is generated, we evaluate its feasibility using the LP explained in the next chapter. After the search for the best fixation is done, we verify its feasibility with an ILP — more on that also in chapter 3. Lastly, we increase the value of C and rerun the search a couple of times, to get a view of the interplay between the upper bound on the makespan and the robustness of the resulting schedules.

Chapter 3

Evaluating feasibility

When we have generated a fixation — by means of the search algorithms explained in the next chapter — we need to know whether it is feasible, i.e. whether a schedule can be made while having the jobs of each fixated relation scheduled on the same machine. If we are able to construct such a schedule, we have proof that the fixation is feasible — otherwise we know that no schedules can be created for the fixation that is being evaluated, and thus that it is infeasible.

The integer linear programming formulation we describe in section 3.1 checks just that — it tries to construct a schedule with all fixated relations enforced. However, since we will be evaluating many fixations, solving this ILP for every one of them will take a lot of time. Therefore we devised an approach that gives a quick and good indication of the feasibility of the evaluated fixation, detailed in the second part of this chapter. Whenever we talk of *the* fixation in this chapter, we mean the fixation that is currently being evaluated for its feasibility.

3.1 The verifier ILP

To determine the feasibility of a fixation, we solve the following machine-indexed integer linear program. The variables in this ILP are x_{jk}, indicating whether job j is to be executed by machine k, similar to those in section 2.3. Because the ordering of jobs in a schedule is of no importance, we can aggregate the jobs of fixated relations into blocks that are scheduled at once — this way the search space is reduced. The set of jobs used in this step is thus a modified version of the set defined by the problem instance, here ranging from 1 to n'.

Since we only have to check whether a feasible solution exists, the objective is trivial. The constraints state that every job is to be executed exactly once (3.1) and that the combined processing times of the jobs scheduled on one machine can not exceed the maximum makespan (3.2).

$$\begin{aligned}
&\min \quad 1 \quad \text{s.t.} \\
&\sum_{k=1}^{m} x_{jk} = 1 && \forall j \in \{1, \ldots, n'\} && (3.1) \\
&\sum_{j=1}^{n'} x_{jk} p_j \leq C && \forall k \in \{1, \ldots, m\} && (3.2) \\
&\quad x_{jk} \in \{0, 1\} && \forall j \in \{1, \ldots, n'\}\ \forall k \in \{1, \ldots, m\}
\end{aligned}$$

Since solving this formulation for many fixations will take a lot of time, we only use it to verify the best fixations returned by the search algorithms, and use the indicator LP described next to evaluate all intermediate solutions. Once the search algorithms have found a set of maximum fixations for the current makespan, we first check the best known fixation — should this check fail, the next best solution is checked, etcetera, until a feasible fixation of relations is found.

3.2 The indicator LP

To quickly determine the feasibility of a given fixation, i.e. finding out whether a schedule can be constructed for it, we use the technique introduced by Van den Akker et al. (2012) — we determine the number of machines needed to schedule all tasks within the given maximum makespan. This is done with a linear programming problem that contains all jobs and relations of the problem instance, as well as the additional constraints introduced by the fixated relations. By looking for the minimum number of machines needed to schedule all tasks, and comparing this to the number of available machines as defined by the problem instance, we can determine the feasibility of the fixation. The objective of this LP is thus to minimise the number of machines that are used to execute all of the jobs within the maximum makespan.

If we find a solution to this formulation that uses at most the number of available machines, we assume the fixation is feasible — of course, this does not give a guarantee on the feasibility of that fixation, as we would have to fully run the ILP outlined in the previous section to verify we can actually compose a schedule in that case. However, previous research like that by Van den Akker et al. (2012) has consistently found this to be a very good indicator, i.e. that whenever this LP indicates that we need at most the number of available machines, we can assume it is possible to construct a feasible schedule. In practice, the verifier ILP always succeeds with the first fixation that is checked, proving the effectiveness of this indicator LP. We therefore only verify the feasibility of the best solutions found, after the search algorithms are done.

The goal of minimising the number of occupied machines lends itself excellently for column generation. This well known technique, described by for example Bazaraa et al. (2005), works by dividing a problem into a master problem and a subproblem — the master problem is to find the smallest subset of columns for which all constraints are met, and the subproblem is tasked with generating new columns such that the solution to the master problem can be improved.

3.2.1 Linear programming formulation

The columns we are working with are machine schedules, each consisting of a complete schedule for a single machine, that can be chosen to be executed at the expense of occupying one machine for the full length of the resulting schedule. These machine schedules are represented by the binary parameters a_{js}, where j is the index of the task and s of the column, indicating whether that task is included in that machine schedule:

$$a_{js} = \begin{cases} 1 & \text{job } j \text{ is included in machine schedule } s; \\ 0 & \text{otherwise.} \end{cases}$$

Each of the machine schedules has a binary variable x_s to indicate whether it is included:

$$x_s = \begin{cases} 1 & \text{machine schedule } s \text{ is used;} \\ 0 & \text{otherwise.} \end{cases}$$

Let us assume for the moment that S is the set of all feasible machine schedules, i.e. those that honour the fixation of relations and adhere to the maximum makespan, with which we can solve the following ILP. The objective is to minimise the number of used columns, and the constraints enforce that all jobs are executed exactly once (3.3).

$$\begin{aligned} \min \quad & \sum_{s \in S} x_s \quad \text{s.t.} \\ & \sum_{s \in S} a_{js} x_s = 1 && \forall j \in \{1, \ldots, n\} && (3.3) \\ & x_s \in \{0, 1\} && \forall s \in S && (3.4) \end{aligned}$$

Of course, iterating over the entire set of feasible machine schedules is intractable, as there are far too many to even list them all, let alone use them in our formulation. Therefore, we relax the formulation by substituting constraint 3.4 with the condition $x_s \geq 0 \quad \forall s \in S$ — we can leave out $x_s \leq 1$, as this is already implied by 3.3 — and solve this LP relaxation using column generation. We start out with S as a small set of columns — based on the schedule that was found when determining the minimal maximum makespan, as described in section 2.1 — and we keep expanding this set until we have solved the master problem. To help us find an allowed initial solution, we allow for partially invalid columns, i.e. those that do not honour the fixation, by giving them a penalty. By setting their cost to a large number, we make it impossible to conclude a fixation is feasible based on a solution that uses such a column, and we give the solver a high incentive to move away from using any of these invalid columns. To ensure only feasible columns are added to the LP later on, we enforce the validity of single machine schedules during their generation.

3.2.2 Generating new columns

To introduce new machine schedules, we have to solve the subproblem — also known as the pricing problem — that makes a selection of the jobs that will be included in the new schedule. As a column takes up one machine when used, its *direct gain* $c_s = 1$, whereas its *direct cost* depends on the included jobs. Its *reduced cost* c'_s is therefore equal to the following, where λ_j represents the shadow prices of constraint 3.3, indicating how much we would like to include job j:

$$c'_s = c_s - \sum_{j=1}^{n} \lambda_j a_{js} = 1 - \sum_{j=1}^{n} \lambda_j a_{js}.$$

To find the column with minimal reduced cost, we turn this into an optimisation problem using the following objective function, with the added constraint that the sum over the processing times of the included jobs is at most equal to the maximum makespan (3.5). We denote the outcome of this problem with $\hat{c}$, and define the minimum reduced cost as $c^* = 1 - \hat{c}$.

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} \lambda_j a_{js} \quad \text{s.t.} \\ & \sum_{j=1}^{n} p_j a_{js} \leq C \\ & a_{js} \in \{0,1\} \qquad\qquad \forall j \in \{1,\ldots,n\} \end{aligned} \tag{3.5}$$

Using the processing times of the jobs as weights and the shadow prices λ_j as values, we can solve this problem as a knapsack instance using a dynamic programming routine, with C as the capacity. To ensure compliance with the fixations of relations, we aggregate the jobs of fixated relations into blocks, which have to be included or excluded entirely.

If a new machine schedule with negative reduced cost is found, it is returned to the LP, which we can then solve again, leading to a new set of shadow prices, for which we solve the pricing problem again, etcetera. When $c^* \geq 0$, no more improving columns can be generated for the LP, and it has been solved to optimality. We can then pass the verdict on the feasibility of the fixation back to the search algorithms. Of course, if the objective value of the LP gets below m before we have solved it to optimality, we can stop then and there — adding more columns can only lower the number of needed machines, and we already have our answer.

3.2.3 Heuristic for the pricing problem

The dynamic programming procedure mentioned above can be slow to complete, as it solves the pricing problem to optimality and has a runtime of $O(nC)$. Therefore a faster heuristic

was conceived, based on the greedy approach to continuous knapsack problems, giving a good approximation of the optimal solution. We implemented it as follows.

First, we aggregate all jobs into blocks, such that all jobs that have a fixated relation between them are grouped together. The aggregated processing time of each block is calculated, and the cost of each block is set by summing up the λ_j values of all included jobs, divided by the aggregated processing time of the block, resulting in a ratio that is used to sort the blocks. The sorted list of blocks is then iterated over, starting from the best ratio. If all the jobs of the current block can be added to the machine schedule we are generating, without making it exceed the maximum makespan, they are included — otherwise this block is ignored. All blocks are checked in this way, until one with negative profit is encountered, after which the process is stopped. Since the ordering of jobs in the machine schedule is of no importance, the resulting column can be returned immediately after completing this process.

Because this method does not always give the optimal answer, the slower implementation based on dynamic programming is also required. That is called only when the greedy heuristic fails to find a new column, to minimise the impact on the running time. If the dynamic programming routine is also unable to find a new set of jobs with a combined negative reduced cost, we have solved the LP to optimality.

3.2.4 An intermediate lower bound

An intermediate check on the objective value was introduced by Van den Akker et al. (2012), that can help us determine the infeasibility of a fixation without completely solving the LP. As presented, the linear programming formulation is used to find the minimal number of machine schedules that are needed to schedule all jobs. But for judging the feasibility of a fixation, we have no interest in the resulting objective value — all we need to know is whether this value is at most equal to the number of available machines. If we can prove it is impossible to achieve an objective value this low, i.e. we know for sure that the solution needs more machines, we would be better off aborting the search.

When we have solved the pricing problem to optimality using the dynamic programming routine, and generated a new machine schedule with reduced cost c^*, we know that the following holds for all $s \in S$:

$$1 = c_s = c'_s + \sum_{j=1}^{n} \lambda_j a_{js} \geq c^* + \sum_{j=1}^{n} \lambda_j a_{js}.$$

This expression can be used to find a lower bound of the LP objective as follows.

$$\sum_{s \in S} x_s \geq \sum_{s \in S} x_s \left(c^* + \sum_{j=1}^{n} \lambda_j a_{js} \right)$$

$$\sum_{s \in S} x_s \geq c^* \sum_{s \in S} x_s + \sum_{s \in S} x_s \left(\sum_{j=1}^{n} \lambda_j a_{js} \right)$$

$$\sum_{s \in S} x_s \geq c^* \sum_{s \in S} x_s + \sum_{j=1}^{n} \lambda_j \left(\sum_{s \in S} x_s a_{js} \right)$$

Recall that constraint 3.3 ensured that $\sum_{s \in S} x_s a_{js} = 1 \quad \forall j \in \{1, \ldots, n\}$.

$$\sum_{s \in S} x_s \geq c^* \sum_{s \in S} x_s + \sum_{j=1}^{n} \lambda_j$$

$$(1 - c^*) \sum_{s \in S} x_s \geq \sum_{j=1}^{n} \lambda_j$$

Note that we defined $\hat{c}$ as $1 - c^*$, and since $c^* < 0$, we know that $\hat{c}$ is always greater than 1.

$$\sum_{s \in S} x_s \geq \sum_{j=1}^{n} \lambda_j / \hat{c}$$

That gives us our lower bound — should the value of $\sum_{j=1}^{n} \lambda_j / \hat{c}$ be larger than m, we know that we need more than the available number of machines to schedule all jobs, and can thus conclude that the fixation we are evaluating is infeasible.

3.2.5 Notes on the implementation

The indicator LP has been implemented with the Java API of Gurobi (2009) — as opposed to the other linear programming formulations in this work, which make use of the CPLEX framework by IBM (2009). Instead of constructing the problem every time we are evaluating a fixation, we continuously re-use one instance of the solver. This means we create the model containing the problem data only once, allowing for a significant improvement of the running time of this approach. It does however also mean that when tasked to evaluate another fixation, all columns that are in use by the solver have to be checked, making sure they are valid for the fixation we are evaluating next. Any invalid columns present are given a penalty, to make the solver look for alternatives as it tries to minimise the number of needed machines.

After we have come to a verdict on the feasibility of the fixation, all columns that are not part of the solution base are removed from the solver, keeping the number of variables in use by the solver to a minimum. Additionally, all columns that are generated are stored separately. On the next run, this collection is checked for any columns that are useful for the current fixation, thereby avoiding the need to generate columns that have been constructed before.

Chapter 4

Searching for fixations

In the previous chapter we have seen how we can determine the feasibility of a fixation. This chapter details the devised approaches for finding the best of these fixations. First we describe an approach using genetic local search, followed by one based on simulated annealing.

4.1 Genetic local search

The main method we use to search for fixations is genetic local search — a well known technique, described amongst others by Goldberg (1989), that keeps track of a population of solutions and creates new entries by combining known good ones. A solution, i.e. a fixation, consists of an assignment of the previously mentioned parameter f_r for every $r \in R$, each indicating whether it is to be fixated, i.e. whether the associated jobs are forced to be scheduled on the same machine. In the context of these search algorithms, f_r thus is the variable over which we optimise.

Every iteration of the genetic algorithm, a set of parent solutions is selected from the population and combined into new fixations, as explained in the remainder of this section. These child solutions are then evaluated, i.e. checked for feasibility, using the LP previously described in chapter 3, and added to the population. This process of generating new solutions, evaluating them and updating the population, is repeated until no more improvements can be found, i.e. until no solution that was better than the best known solution has been added to the population for a specific number of iterations, after which the search is stopped and the resulting population is returned. The choice for the value of this parameter, as well as those of others, will be detailed in chapter 8.

To avoid getting stuck in local optima we use a multi-start approach — the algorithm is run three times, after which the resulting populations are merged and the search is run once more based on that population. The best encountered solution is then used as a starting point for the simulated annealing algorithm.

4.1.1 Population of solutions

During the execution of the genetic algorithm, we keep track of a population of 10 solutions. Initially, it only contains one solution, in which no relations are fixated — for this fixation we can certainly generate a schedule, namely the one found when the value of C was determined. Because random solutions have a high chance of not even being feasible, adding any additional fixations to the initial population gives no advantage to the search. Incrementing the size of the population can increase the diversity of the solutions in it, but also increases the running time of the algorithm, as it will then take longer to generate the solutions of the next generation.

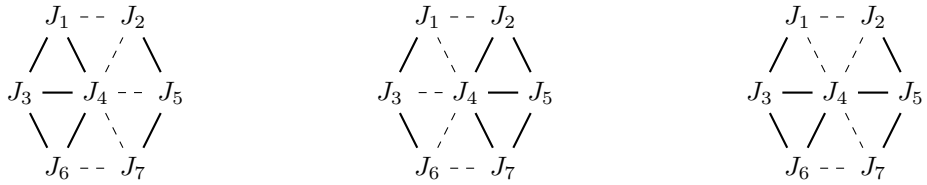
The population is stored in a weighted binary tree, to which every saved solution is added with the number of fixated relations as its weight. This weight denotes the importance of a solution, and is used to make the probability of selecting some solution for reproduction directly proportionate

to its fitness score. This biased random selection is then used when determining which solutions are used as the parents for the next generation — a technique known as roulette wheel or fitness proportionate selection, originally introduced by De Jong (1975). For example, if the population would contain three solutions, with weights 10, 15 and 25 respectively, they would have a 20%, 30% and 50% chance of getting selected as a parent.

To ensure the best solutions encountered are not forgotten, we use an elitist selection scheme — meaning we simply copy the best solutions over into the next generation. For each of the remaining places in the new generation, we select two parents from the original population, so there might be solutions that are employed as parents more than once.

4.1.2 Combining solutions

The conclusions of the experiments conducted by Van Roermund (2013) were that the approach has great potential for finding good fixations, provided that the method used for combining fixations into new ones was improved. The initial method for generating a new fixation did not take any information about the structure of the relations into account — it simply copied the common parts of the parent solutions, and for every relation the parents did not agree on it chose randomly whether to fixate it or not. Take for example the fixations in figure 4.1, in which the nodes represent jobs, bold edges indicate relations that are fixated, and dashed edges those that are not.



(a) Example fixation A.

(b) Example fixation B.

(c) An example result of the naive combination of fixations A and B.

Figure 4.1: Example fixations for a problem instance with seven jobs and twelve relations.

In this example, the relations not involving J_4 are fixated in the exact same way in both fixations A and B, therefore their resulting combination shares these fixations as well. The relations that do involve J_4 are fixated in a completely opposite way in the parent fixations, and therefore the child solution has these relations fixated at random. Unfortunately, this can lead to a fixation like the one in figure 4.1c, in which all jobs are connected by fixated relations, i.e. all jobs are forced to be scheduled on the same machine.

This is unlikely to be desirable, as chances are the combined processing time of these jobs exceeds the value of C — which would make it impossible to create a schedule for it. We therefore needed to improve this method of combining fixations by taking more advantage of the problem structure, which takes us back to the previously mentioned graph partitioning problem.

4.1.3 Improved recombination of fixations

To provide a better way of combining parent solutions into new ones, we devised the following method, consisting of two phases. During this procedure, all jobs are kept in a disjoint-set forest — also known as a union-find data structure, originally introduced by Galler et al. (1964) — to which each task is initially added as a singleton set. The sets in this data structure represent groups of jobs that are forced to be scheduled on a single machine, meaning that all relations that are solely between jobs within one such set have been fixated. By continuously merging these sets, we thus fixate more and more relations, leading to better fixations.

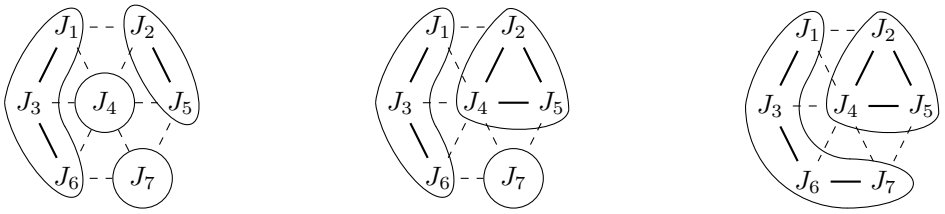
The first phase identifies the common elements of the parent solutions. Every relation that is fixated in both of the parents, is copied over into the new fixation with a certain chance, by merging the sets containing the jobs involved in that relation. The reason for not copying the

fixated relations unconditionally is to allow for some mutations in the recombination — otherwise we would have no way of not fixating some relation once all chosen parent solutions have it fixated. The exact setting of this chance is left to chapter 8.

During the second phase we iterate over the remaining, non-fixated relations in random order, and try to fixate each one by merging the involved sets. Should we be able to construct a feasible single machine schedule that includes all jobs from both sets, we continue with this relation fixated — otherwise we leave those sets as they were and continue on, ignoring that relation. The reason for the random order is to avoid making sub-optimal decisions, even when they are locally optimal — always merging the sets that lead to the best improvement of $\sum -f_r$ does not necessarily return the best possible fixation, and even worse, if the returned fixation turned out to be infeasible, such a greedy approach would regenerate it over and over, never finding a valid solution.

Every time a relation is fixated and sets are merged, we add the fixation as represented at that time by the disjoint-set forest to a list. When we have iterated over all relations, this list is returned to the genetic algorithm. A nice property of this list is that each fixation is a slightly less restricted version of the one after it, i.e. when we are able to prove the feasibility of a fixation in the list, we immediately know that all fixations before it are also feasible. Knowing this, we can use binary search to find the best fixation in the returned list: the fixation in the middle of the list is evaluated using the indicator LP, and should it be feasible we continue searching through the second half of the list; otherwise we only consider the first half of the list as candidate solutions, etcetera. The best fixation found is then added to the population.

Let us illustrate the working of this method using the previous example, with seven jobs and twelve relations, assuming the given maximum makespan ensures that no more than four jobs can be scheduled on a single machine. Taking fixations 4.1a and 4.1b as parents, the first phase copies the relations they both have fixated with a $\frac{3}{4}$ chance, unifying the sets of those jobs — possibly leading to the partitioning depicted in figure 4.2a, in which the sets of the jobs involved in three of the relations are merged, out of the four that the parents have fixated in common.



(a) An example partitioning after phase one.

(b) An example partitioning during the first iteration of phase two.

(c) An example partitioning after phase two.

Figure 4.2: Example combination of fixations 4.1a and 4.1b.

The second phase then iterates over the non-fixated relations, i.e. the dashed lines in the example, and selects one that will be fixated. Note that the relation between J_1 and J_2 is the only one not viable for fixation, as merging those sets would result in a processing time greater than the given maximum makespan. Let us first select the relation $(J_4 \leftrightarrow J_5)$ and unify the involved sets, leading to the situation of figure 4.2b; then we fixate $(J_6 \leftrightarrow J_7)$, resulting in the partitioning shown in figure 4.2c. The list of fixations that would be returned to the genetic algorithm for evaluation would thus be the following.

$$[\{J_1, J_3, J_6\}, \{J_2, J_5\}, \{J_4\}, \{J_7\}]$$
$$[\{J_1, J_3, J_6\}, \{J_2, J_4, J_5\}, \{J_7\}]$$
$$[\{J_1, J_3, J_6, J_7\}, \{J_2, J_4, J_5\}]$$

This example highlights what is both a strength and a weakness of this heuristic, namely its random determination of which sets to merge. The advantage is that more fixations that differ from the parents are tried, while a big part of the genetic information is still honoured. On the

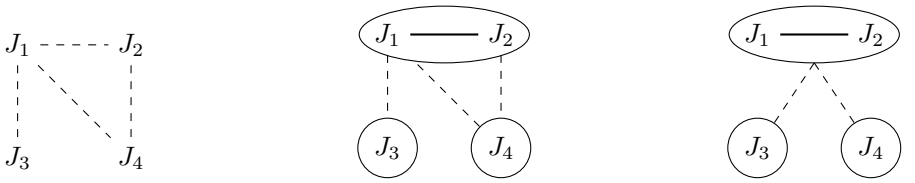
other hand, the disadvantage is that sub-optimal choices can be made — like in the figures above, in which the last partitioning has six relations fixated, whereas both parents had seven. As argued before however, merging the sets such that the most relations are fixated at each step can get stuck, generating infeasible solutions. Another upside is that random selection is much quicker than optimal selection, allowing the heuristic to produce more candidate fixations in less time.

4.1.4 Alternative combining strategy

A characteristic of the technique described above is that cliques — or other tightly connected subsets of jobs — have a higher chance to end up in the same set than jobs with only a single relation between them. This is because there can be multiple edges between jobs inside and outside a set, as illustrated by the following example.

Consider the small problem instance with four jobs and the relations $(J_1 \leftrightarrow J_2)$, $(J_1 \leftrightarrow J_3)$, $(J_1 \leftrightarrow J_4)$ and $(J_2 \leftrightarrow J_4)$, as shown in figure 4.3a. Let us assume that the value of C is chosen such that at most three jobs can be scheduled on a single machine. Let us also assume that the first relation, between jobs J_1 and J_2, has already been fixated, and we are now deciding which relation to fixate next. By iterating over the non-fixated relations in random order, the fourth job now has a $\frac{2}{3}$ chance of getting merged with J_1 and J_2, since two of the three remaining edges involve J_4, as in figure 4.3b.

If we were to use a graph instead of a disjoint-set forest during this process, we could also unite the edges of sets when merging them. This would then lead to the situation as depicted in figure 4.3c, where the original edges $(J_1 \leftrightarrow J_4)$ and $(J_2 \leftrightarrow J_4)$ are replaced by one edge. Now both J_3 and J_4 have an equal chance of getting selected for inclusion.



(a) Problem instance. (b) Using disjoint sets. (c) Using a graph.

Figure 4.3: Example job partitioning for a problem with four jobs and four relations.

Although this strategy would seem to select the next relation to fixate in way that is fairer, giving more chance to relations that would otherwise be outnumbered, its implementation actually performs worse than the one based on the disjoint sets. This can be explained by the fact that it is better to merge cliques together rather than loosely connected jobs, as the former make a larger contribution to the number of fixated relations — as is evident from the previous example, where the set $\{J_1, J_2, J_4\}$ has three relations fixated, whereas $\{J_1, J_2, J_3\}$ would have only two. It is therefore a good thing not to unify the edges between sets, to give the sets which contribute to a greater increase in the number of fixated relations a larger chance of getting merged. We therefore stick with the previously described implementation using disjoint sets.

4.2 Simulated annealing

In addition to the genetic algorithm, we implemented another well known local search heuristic to look for the best fixations — simulated annealing, as introduced by Kirkpatrick et al. (1983). It works by maintaining a single solution and repeatedly looking at its neighbourhood, accepting any change that leads to an improvement, and permitting a worsening with a certain chance. This chance depends on both the magnitude of the negative impact on the objective value and the current temperature. During the execution of the algorithm, this temperature is gradually lowered according to a cooling schedule, to ensure that the chance of accepting declines diminishes.

4.2.1 Changing the solution

For this search algorithm we also use the job-sets representation of solutions. During every iteration, a small modification is applied to the current fixation. This can either be to merge two randomly selected sets of jobs, or to split up some set, i.e. re-adding each of its jobs as singleton sets. Since the unification of two sets can never lead to less fixated relations, a merge always results in a solution that is at least as good as the previous one, provided that it stays feasible. The scattering of a set on the other hand can never lead to more fixated relations, and can therefore never result in an improvement. However, splitting up sets is still needed in order to get away from local optima, in which no more merges can be done. The choice for either change is made randomly, at a ratio that will be determined in chapter 8. If we choose to merge sets, but are unable to find appropriate candidates, we fall back to the scattering. The choice of which sets to operate on is also made randomly, although a check is enforced to see whether the merging of a set does not make it exceed the maximum makespan.

During the run of the algorithm, a disjoint-set forest is maintained for the current solution, with the jobs divided into sets as dictated by the fixation. Every time a change is made, the sets are updated — either by merging two or by scattering one — and the resulting fixation is offered to the indicator LP for evaluation. Should the new solution be deemed feasible, the change in the number of fixated relations is computed. Based on this number and — in case of a lessening — the temperature, a decision is made whether to keep this solution. Even though determining the increase in the number of fixated relations is easier to compute than evaluating a solution, the used order is not an issue given our two possible changes: if a merge has taken place, the number of fixated relations will have increased, so we will want to evaluate the new solution; if a set was scattered, proving its feasibility is trivial given the evaluation of the old solution.

The chance of accepting a worsening is calculated using the formula $e^{\frac{\delta}{t}}$, where δ is the difference in the number of fixated relations, and t represents the current temperature. Should the change be accepted, the algorithm continues with the new fixation; otherwise, the original solution is restored. After a predefined number of iterations in which no improvement has been made, the search is suspended and the changes done since we encountered the best fixation are reverted. We then continue the search, keeping the temperature as it was. This reverting to the best found solution and resuming the search is done up to three times, to avoid getting stuck in local optima.

4.2.2 Cooling schedule

The cooling schedule of the simulated annealing algorithm dictates the course of the temperature during the run. It is controlled by a starting value, to which the temperature is initialised, and a value between 0 and 1 that it is multiplied by every so many iterations.

For example, if the starting temperature is set to 9.5, a decrease of one fixated relation has a $e^{\frac{\delta}{t}} = e^{\frac{-1}{9.5}} = 0.90$ chance of getting accepted, whereas a change leading to ten less fixations has a $e^{\frac{-10}{9.5}} = 0.35$ chance of not being reverted. Then, every so many iterations, the current temperature is multiplied by for example 0.9, thus lowering the chance of moving to a worse solution. See chapter 8 for the determination of the actual values of these parameters.

4.3 Chaining the search algorithms

Initial experimental results showed that the genetic search algorithm finds good solutions faster than the simulated annealing approach, especially when starting from a bare solution with no relations fixated. Therefore, we execute a simulated annealing search starting from the best solution in the returned population of every run of the genetic algorithm during the multi-start approach. Once all three runs of the search algorithms have been completed, we check the best fixations with the feasibility ILP described in section 3.1, to see whether a schedule can actually be formed for this fixation. We then continue by increasing the value of the maximum makespan and redo the searches.

This concludes the description of our basic decomposition approach — using the search algorithms to generate fixations and employing the indicator LP to quickly evaluate their feasibility. In the next part of this thesis we present several extensions to its formulation, to enhance the problem category it is applicable for.

Part II

Extensions

Chapter 5

Job characteristics

In this chapter we present two extensions to the job characteristics of the problem category we are solving — we redefine the correlations as proper precedence relations and include the availability of jobs. The first two sections describe the rationale for their inclusion; the other sections update the formulations of the evaluation methods. The adaptation of the ILP that solves a problem to optimality as presented in section 2.3 is left for the next chapter.

5.1 Including precedence relations

Section 1.3 described the simplification of precedence relations to mere correlations, thereby excluding the partial ordering of jobs. Since these precedence relations are present in a lot of real world problems, we would like to include them in our approach. The precedence relations mentioned in section 1.1 were of the form $S_j - S_i \geq q_{ij}$ for all $r : (i, j) \in R_p$, forcing job j to start at least q_{ij} time units later than job i is started. Note that we use R_p instead of R to indicate the set of precedence relations, as correlations make a comeback in section 6.2. Other types of precedence relations, obtained by changing the $\geq$ to a $\leq$ or $=$, are not included in this work — see section 10.3.1 for thoughts on how those could be incorporated.

We define $pred(j)$ to be the set of predecessors of j, i.e. containing each job i for which there exists a relation $r : (i, j) \in R_p$, and similarly, $succ(j)$ as the set of successors of j, i.e. containing each job i for which there exists a relation $r : (j, i) \in R_p$. We use these definitions in reformulating the LP:

$$pred(j) = \{i \mid (i, j) \in R_p\};$$
$$succ(j) = \{i \mid (j, i) \in R_p\}.$$

5.2 Adding job availabilities

We also include release dates and deadlines of jobs, which indicate the time window in which a job should be executed — job j can not be started before its release date r_j, and it must be finished by its deadline $\bar{d}_j$. Both values should be non-negative, and to allow for the execution of the job, there should be at least p_j time units between them, i.e. $\bar{d}_j - r_j \geq p_j$ must always hold.

Up to now, we have acted as if all release dates were set to time zero, and the deadlines were equal to the value of C, the maximum makespan — but from now on the availability is set for each job individually, and assumed to be given by the problem instance. Since the deadline of a job could be greater than the upper bound on the makespan, we have to take extra care to ensure neither is violated by a schedule.

To find the minimal maximum makespan with these precedence relations and availability restrictions taken into account, we update the contraint programming implementation used to find the shortest possible schedule, making it solve $\mathrm{P}|r_j, \bar{d}_j, prec|C_{\max}$.

5.3 Adapting the verifier ILP

Incorporating starting times into the formulation of the verifier ILP means we can no longer aggregate the jobs of fixated relations, as that would enforce an absolute ordering on those jobs. Instead, we will have to use the full set of jobs and add a time index to the variables, making them indicate both by which machine a job is executed, and at what time that is to be done. This changes the variables used in this formulation to the following:

$$x_{jkt} = \begin{cases} 1 & \text{job } j \text{ is started by machine } k \text{ at time } t; \\ 0 & \text{otherwise.} \end{cases}$$

The objective of the updated ILP remains trivial, as we only need to verify whether a schedule can be made, making the formulation as below. The first two constraints ensure that every job is scheduled exactly once (5.1) and at every single moment each machine is executing at most one job (5.2). The precedence relations are included with constraint 5.3, where the starting time of job j is expressed as $\sum_k \sum_t x_{jkt}t$, and the last constraint makes sure the fixations are honoured (5.4). The availability of each job, together with the maximum makespan, is taken into account by the definition of the time index in each constraint, i.e. ranging from r_j to $\min\left(C, \bar{d}_j\right) - p_j$.

$$\min \quad 1 \quad \text{s.t.}$$

$$\sum_{k=1}^{m} \sum_{t=r_j}^{\min\left(C,\bar{d}_j\right)-p_j} x_{jkt} = 1 \qquad \forall j \in \{1,\dots,n\} \tag{5.1}$$

$$\sum_{j=1}^{n} \sum_{t'=\max(r_j,t+1-p_j)}^{\min\left(t,\min\left(C,\bar{d}_j\right)-p_j\right)} x_{jkt'} \leq 1 \qquad \forall k \in \{0,\dots,m\} \ \forall t \in \{0,\dots,C\} \tag{5.2}$$

$$\sum_{k=1}^{m} \sum_{t=r_j}^{\min\left(C,\bar{d}_j\right)-p_j} x_{jkt}t - \sum_{k=1}^{m} \sum_{t=r_i}^{\min\left(C,\bar{d}_i\right)-p_i} x_{ikt}t \geq q_{ij} \qquad \forall r : (i,j) \in R_p \tag{5.3}$$

$$\sum_{t=r_i}^{\min\left(C,\bar{d}_i\right)-p_i} x_{ikt} = \sum_{t=r_j}^{\min\left(C,\bar{d}_j\right)-p_j} x_{jkt} \qquad \forall k \in \{1,\dots,m\} \ \forall r : (i,j) \in \{r \mid r \in R_p \ \wedge \ f_r = 1\} \tag{5.4}$$

$$x_{jkt} \in \{0,1\} \qquad \forall j \in \{1,\dots,n\} \ \forall k \in \{1,\dots,m\} \ \forall t \in \{r_j,\dots,\min\left(C,\bar{d}_j\right) - p_j\}$$

5.4 Updating the indicator LP

To include these extensions in the indicator linear programming formulation, we need to augment the representation of columns with the starting times of the jobs they contain. To this end we introduce a new parameter of machine schedules, S_{js}, indicating the starting time of job j in column s. Of course, its value only makes sense for the jobs that are included in that machine schedule, i.e. for which a_{js} is set to 1; otherwise, we assume S_{js} is set to zero. We thus include:

$$S_{js} \in \{0, r_j, \dots, \min(C, \bar{d}_j) - p_j\}.$$

5.4.1 Renewed formulation

Since the jobs involved in a relation can be scheduled on different machines, the precedence constraints have to be enforced outside the generation of new columns, i.e. in the formulation of the master problem, which is changed to the following. Note that this is an integer linear

programming formulation again, of which we take the relaxation by changing the variable x_s into a non-negative real number. Honouring the availability of individual jobs does not depend on multiple machine schedules, and can thus be delegated to the subproblem. The demand that all jobs are executed exactly once is unchanged (5.5), and the constraint on partial order has been added (5.6).

$$\begin{aligned}
\min \quad & \sum_{s \in S} x_s \quad \text{s.t.} \\
& \sum_{s \in S} x_s a_{js} = 1 && \forall j \in \{1, \ldots, n\} && (5.5) \\
\sum_{s \in S} x_s a_{js} S_{js} - & \sum_{s \in S} x_s a_{is} S_{is} \geq q_{ij} && \forall r : (i,j) \in R_p && (5.6) \\
& x_s \in \{0,1\} && \forall s \in S
\end{aligned}$$

5.4.2 Enhancing the pricing problem

With the starting times of jobs added to the machine schedules, the pricing problem becomes harder to solve, as we now also need to decide in what order to put the jobs we are including. Since the formulation of the LP gained an additional constraint, we also have an extra set of shadow prices — let λ_j represent the shadow prices of the first constraint like before, and δ_{ij} those of constraint 5.6. As given by Van den Akker et al. (2012), we can aggregate these δ_{ij} values into a dual for individual jobs, which is called Q_j and indicates how much we would like to move job j to the back of the schedule:

$$Q_j = \sum_{i \in pred(j)} \delta_{ij} - \sum_{i \in succ(j)} \delta_{ji}.$$

Using these, we can express the reduced cost of a new column s as:

$$c'_s = 1 - \sum_{j=1}^{n} \lambda_j a_{js} - \sum_{j=1}^{n} Q_j S_{js}.$$

We reformulate the pricing problem using the variable b_{jt}, indicating job j is to be started in the new schedule at time t. We then have that the starting time of job j is represented by the sum $\sum_t b_{jt} t$, provided that $\sum_t b_{jt}$ equals one — the index t ranges from r_j to $\min\left(C, \bar{d}_j\right) - p_j$, although we use this abbreviated form in the text. We thus define:

$$b_{jt} = \begin{cases} 1 & \text{job } j \text{ is started at time } t; \\ 0 & \text{otherwise.} \end{cases}$$

Even though the precedence relations are already enforced in the formulation of the LP, it would be unwise not to include them here as well — generating a new column that does not honour the constraints of the master problem will never provide an improvement to it. We therefore need to ensure that whenever both jobs i and j involved in some relation $r : (i,j)$ are included, the difference between their starting times is at least q_{ij}. However, if only one or neither of the jobs is included, this constraint should evaluate to something trivial, so it has no effects on the solution:

$$\sum_t b_{jt} t - \sum_t b_{it} t \geq q_{ij} \quad \text{if and only if} \quad \sum_t b_{it} = \sum_t b_{jt} = 1.$$

Let us examine the left hand side of this constraint. Since both sums evaluate to zero or larger, we can conclude that the minimal value of the left hand side is minus the maximum starting time of job i. We therefore modify the right hand side by subtracting a value M_{ij}, such that it becomes

no larger than $-\left(\bar{d}_i - p_i\right)$ whenever at most one of the jobs involved in the relation is included, but it is left as it was whenever both jobs are selected. To this end we define M_{ij} as follows:

$$M_{ij} = \left(2 - \sum_{t=r_i}^{\min\left(C,\bar{d}_i\right)-p_i} b_{it} - \sum_{t=r_j}^{\min\left(C,\bar{d}_j\right)-p_j} b_{jt}\right)\left(q_{ij} + \bar{d}_i - p_i\right).$$

We extend the pricing problem using this constraint, leading to the following formulation. The objective maximises the rewards for choosing jobs at certain times, by means of the duals. The constraints enforce that each job is scheduled at most once (5.7), that at any given moment at most one job is scheduled (5.8), that the precedence relations are adhered (5.9), and that the fixation is honoured (5.10). Note that, just like with the verifier ILP, the maximum makespan and the availability of jobs are honoured by the definition of the time indices, and that they therefore do not need a separate constraint.

$$\max \quad \sum_{j=1}^{n} \sum_{t=r_j}^{\min\left(C,\bar{d}_j\right)-p_j} b_{jt}\left(\lambda_j + tQ_j\right) \quad \text{s.t.}$$

$$\sum_{t=r_j}^{\min\left(C,\bar{d}_j\right)-p_j} b_{jt} \leq 1 \qquad \forall j \in \{1,\ldots,n\} \tag{5.7}$$

$$\sum_{j=1}^{n} \sum_{t'=\max(r_j,t+1-p_j)}^{\min\left(t,\min\left(C,\bar{d}_j\right)-p_j\right)} b_{jt'} \leq 1 \qquad \forall t \in \{0,\ldots,C\} \tag{5.8}$$

$$\sum_{t=r_j}^{\min\left(C,\bar{d}_j\right)-p_j} b_{jt}t - \sum_{t=r_i}^{\min\left(C,\bar{d}_i\right)-p_i} b_{it}t \geq q_{ij} - M_{ij} \qquad \forall r : (i,j) \in R \tag{5.9}$$

$$\sum_{t=r_i}^{\min\left(C,\bar{d}_i\right)-p_i} b_{it} = \sum_{t=r_j}^{\min\left(C,\bar{d}_j\right)-p_j} b_{jt} \quad \forall r : (i,j) \in \{r \mid r \in R \ \wedge \ f_r = 1\} \tag{5.10}$$

$$b_{jt} \in \{0,1\} \qquad \forall j \in \{1,\ldots,n\} \ \forall t \in \{r_j,\ldots,\min\left(C,\bar{d}_j\right) - p_j\} \tag{5.11}$$

5.4.3 A new pricing heuristic

The ILP formulation of the pricing problem outlined above can be solved to optimality using a CPLEX implementation, though this can take a lot of time, so a faster heuristic is employed as well — only when that is unable to find any new columns, the ILP is invoked. Because of the time indices for each job, we are unable to aggregate jobs that are fixated together, meaning we are not able to diminish the search space that way. Instead, we divide the pricing heuristic into two phases — during the first we determine which jobs are included in the new column, and during the second we set the starting time of each of the chosen jobs.

To determine which jobs get selected for the new column, we solve the LP relaxation of the formulation presented above. By relaxing constraint 5.11 to $b_{jt} \geq 0$ — an upper limit of 1 is already implied by 5.7 — the problem becomes easier to solve. We use the resulting set of values, consisting of a number between 0 and 1 for each task, as the chances to include the corresponding jobs. When the solver has successfully solved the relaxation, we iterate over the sets of jobs that are grouped together based on the current fixation, and sum up their values of b_{jt}. That sum, divided by the number of jobs in the set, represents its chance of inclusion. We then randomly select some of those sets, and try to create a feasible assignment of starting times for them using the second phase. To increase our chances of finding an improving column and possibly generating more than one, we repeat the random selection of sets of jobs a number of times.

The second phase of this heuristic is tasked to return the starting times for a given set of jobs, such that the reduced cost of the resulting column is as low as possible. For this phase we make use of a constraint programming problem, that tries to construct a valid machine roster that includes all the jobs that were selected during the first phase. The constraints given to the CP solver ensure that each of the given tasks is scheduled within its availability and the current maximum makespan, and that at all times at most one job is included, as we are generating a single machine schedule. The value of each task starting at a specific time is calculated from the shadow prices, just like in the objective of the ILP above. If we are able to find a feasible assignment for the starting times of the indicated jobs, i.e. one that honours these constraints and makes the reduced costs come out negative, we return the generated roster — otherwise we return nothing.

After having called the second phase for each of the generated lists of jobs during the first phase, we possibly have multiple new columns that can be returned to the indicator LP, which can then be solved again. If we found no feasible columns, the integer linear programming implementation of the pricing problem as described in the previous section is invoked.

Chapter 6

Objective terms

This chapter presents two extensions to the terms used in the objective function of the problem category — we detail the addition of weights to relations, and discuss how to better approximate the original robustness measure $\sum \gamma$. As before, the first two sections describe the rationale for their inclusion, the others give the updated formulations of the search methods.

6.1 Introducing weighted relations

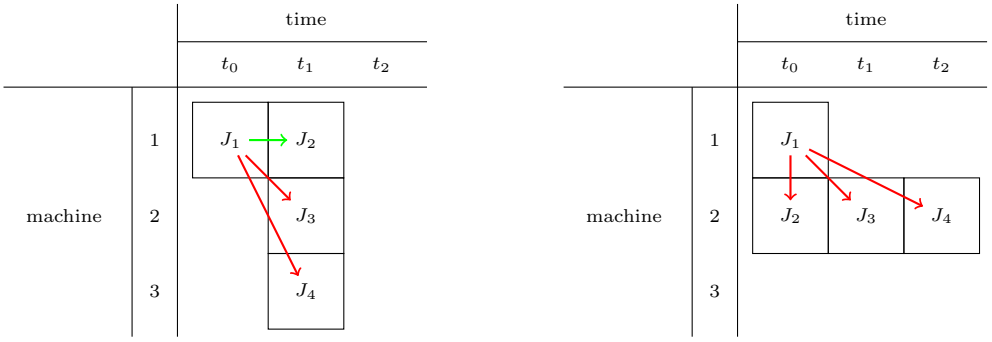
Along with the simplification of precedence relations to correlations in section 1.3, an example use case was given in the form of scheduling related computations over several computers. In that scenario, tasks might benefit from the efforts of other tasks — and it could very well be possible that some tasks benefit more from this effect than others. It could therefore be beneficial to the robustness of the resulting schedule to fixate some relations in favour of others. The inclusion of precedence relations in the previous chapter does not change this, as we can still prefer the jobs of a relation scheduled on the same machine over the jobs of some other relation — although the partial ordering imposed by the precedence relations must be honoured at all times.

To model this preference for the fixation of a specific relation, we introduce a weight for each relation r, represented by the property w_r. Technically, this property could have any real number as its value, although we assume positive values, as those are the most common, indicating how much we would like to the relation fixated. A negative number would indicate the opposite, i.e. making us prefer not having the jobs of this relation on the same machine. Using zero for the weight of a some relation means we do not reward its fixation in any way; in this case we do not even need to try to fixate it, as this would never be beneficial. Note that up to now, we acted as if all weights were set to 1.

6.2 Better approximating $\sum \gamma$

With the introduction of fixations in section 2.2, the objective of the search became to find a schedule with the most fixated relations. While this does give an upper bound on $\sum \gamma$ — recall that γ_{jk} was defined to take a value of 1 if job j itself is not scheduled on machine k, but has at least one successor that is, and 0 otherwise — it does not necessarily minimise that term, which Hoppenbrouwer (2011) introduced to maximise robustness.

It is even possible for a schedule to have more relations fixated but a lower $\sum \gamma$, as figure 6.1 demonstrates. The first schedule in this figure has one fixated relation, between J_1 and J_2, but a non-fixated relation to both jobs scheduled on machines two and three, i.e. $\gamma_{12} = \gamma_{13} = 1$, leading to a $\sum \gamma$ score of 2. Schedule 6.1b on the other hand, has none of its three relations fixated, making it inferior when using the original objective — however, since it only has non-fixated relations to jobs on machine two, its $\sum \gamma$ is equal to 1, meaning this schedule could just as well be considered superior to the first. This makes sense from the viewpoint of delay propagation, since



(a) $\sum -f_r w_r = -1$, $\sum \gamma = 2$ (b) $\sum -f_r w_r = 0$, $\sum \gamma = 1$

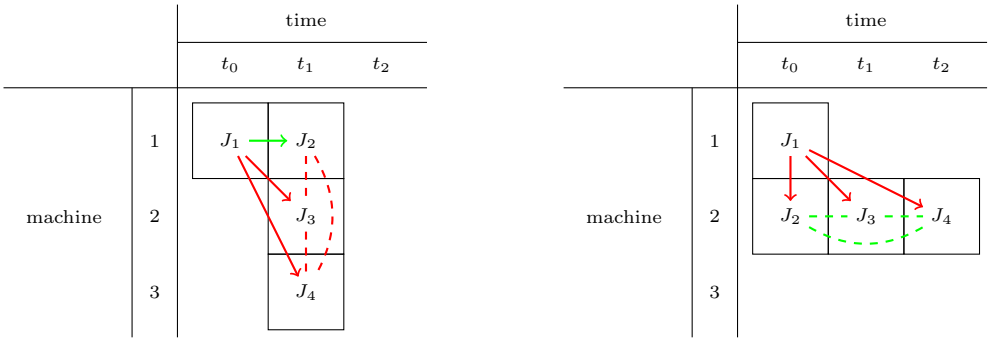
Figure 6.1: Two example schedules with four jobs and three relations, with $w_r = 1$.

in schedule 6.1a two other machines would have to wait when the first gets delayed, whereas this would only affect one other machine in the second schedule.

6.2.1 Additional correlations

To address this issue, we add some extra relations to the problem instance. For every job j, we add a *correlation* between each possible pair of elements in the set $succ(j)$, without adding any additional partial ordering constraints. In the example given above, we would thus add the correlations $(J_2 \leftrightarrow J_3)$, $(J_2 \leftrightarrow J_4)$ and $(J_3 \leftrightarrow J_4)$, all between successors of J_1. If there already is a precedence relation between some combination of jobs, we leave out the additional correlation for that pair, e.g. if the previous example contained the relation $(J_2 \rightarrow J_4)$, we would only add the correlations $(J_2 \leftrightarrow J_3)$ and $(J_3 \leftrightarrow J_4)$. Augmenting the problem in this way effectively turns the set of successors of each job into a clique, giving the solver a higher incentive to schedule the involved jobs on the same machine.

Because we do not want these additional correlations to overshadow the original precedence relations in the objective function, we include the former with smaller weights. For our example to work, the three correlations have to outweigh the one precedence relation, otherwise the first schedule would still be superior. We therefore set the weights to half the average weight of the precedence relations, i.e. to $\frac{1}{2}$ when those weights are still set to 1 — using a different weight ratio would obviously lead to different results; the higher the weights of the additional correlations, the more the search algorithms focuses on cliques, specifically large cliques. Figure 6.2 shows the result of this extension, and demonstrates how the second schedule now is better according to both metrics.

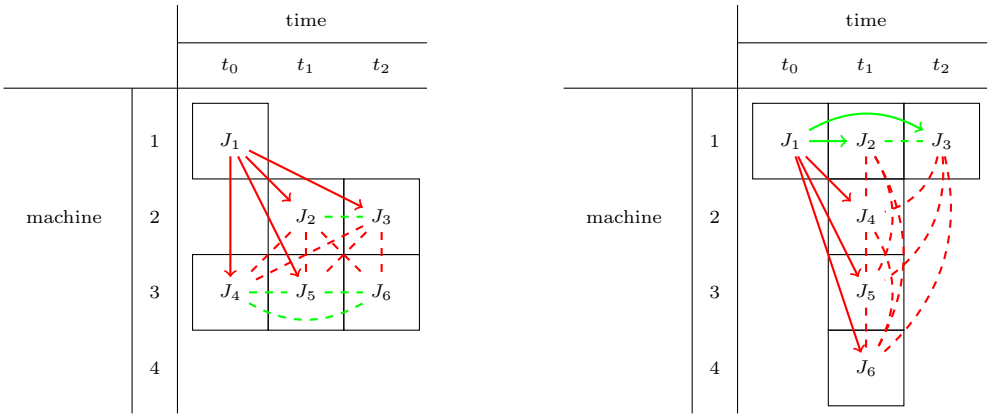


(a) $\sum -f_r w_r = -1$, $\sum \gamma = 2$ (b) $\sum -f_r w_r = -\frac{3}{2}$, $\sum \gamma = 1$

Figure 6.2: The example schedules of figure 6.1 augmented with additional correlations.

Note that this augmentation of the objective function is still an approximation of $\sum \gamma$, and the values of these metrics can still differ. Figure 6.3 shows two example schedules with six tasks, of which jobs two through six are successors of J_1. In schedule 6.3a, there are possibilities for propagations of delays to two machines, whereas this number is three in the second example. On the other hand, when considering the summed weight of the fixated relations, the search algorithms will prefer example 6.3b over the first, which is not what we wanted. The difference between these two schedules is fairly small however, and considering the amount of added relations it is quite likely that a fixation is generated that has even more jobs bound to a single machine.

To completely avoid these situations in which a better fixation actually has a worse $\sum \gamma$, we would have to set the weights of the additional correlations equal to the weight of the precedence relations. However, it is questionable whether that is desirable, as that might focus on these additional correlations too much to be beneficial for the robustness. Also, with the introduction of weighted relations in the previous section, precedence relations can be set to have non-uniform arbitrary weights, so that is not an option. We therefore leave the weights of the additional correlations at half the average weight of the precedence relations, which still gives us the improvement over the original situation of figure 6.1.



(a) $\sum -f_r w_r = -\frac{4}{2}$, $\sum \gamma = 2$ (b) $\sum -f_r w_r = -2\frac{1}{2}$, $\sum \gamma = 3$

Figure 6.3: Two example schedules with five jobs, four precedence relations and six correlations.

In the remainder of this chapter we present the updated search algorithms for finding the best fixations, where we continue to use R_p to indicate the set of original precedence relations and introduce R_c to represent the set of additional correlations. We also redefine R as the union of these two sets, i.e. $R = R_p \cup R_c$.

6.3 Solving the extended problem to optimality

To solve the problem with the mentioned extensions to optimality, we update the ILP presented in section 2.3 by adding a time index to its variables. This makes it very similar to the updated verifier ILP, as it then also has x_{jkt} indicating whether job j is scheduled to start on machine k at time t. The other variable, y_{ij}, is still used to fixate jobs i and j on the same machine, with the only difference that $r : (i, j)$ now comes from the union of R_p and R_c. We include the weights of the relations in the objective function, leading to the following formulation. The first three constraints are the same as those in section 5.3, stating that every job is to be included once (6.1), each machine is executing at most one job at a single point in time (6.2), and the precedence relations are to be honoured (6.3). The last two ensure that fixated relations are to be executed by the same machine (6.4 & 6.5). Also here, the availability of jobs is taken into account by the definition of the time index in each constraint.

$$\max \sum_{r:(i,j)\in R} y_{ij} w_r \quad \text{s.t.}$$

$$\sum_{k=1}^{m} \sum_{t=r_j}^{\min(C,\bar{d}_j)-p_j} x_{jkt} = 1 \qquad \forall j \in \{1,\dots,n\} \tag{6.1}$$

$$\sum_{j=1}^{n} \sum_{t'=\max(r_j,t+1-p_j)}^{\min(t,\min(C,\bar{d}_j)-p_j)} x_{jkt'} \leq 1 \qquad \forall k \in \{1,\dots,m\}\ \forall t \in \{0,\dots,C\} \tag{6.2}$$

$$\sum_{k=1}^{m} \sum_{t=r_j}^{\min(C,\bar{d}_j)-p_j} x_{jkt}t - \sum_{k=1}^{m} \sum_{t=r_i}^{\min(C,\bar{d}_i)-p_i} x_{ikt}t \geq q_{ij} \qquad \forall r:(i,j) \in R_p \tag{6.3}$$

$$\sum_{t=r_i}^{\min(C,\bar{d}_i)-p_i} x_{ikt} - \sum_{t=r_j}^{\min(C,\bar{d}_j)-p_j} x_{jkt} \leq 1 - y_{ij} \quad \forall k \in \{1,\dots,m\}\ \forall r:(i,j) \in R \tag{6.4}$$

$$\sum_{t=r_j}^{\min(C,\bar{d}_j)-p_j} x_{jkt} - \sum_{t=r_i}^{\min(C,\bar{d}_i)-p_i} x_{ikt} \leq 1 - y_{ij} \quad \forall k \in \{1,\dots,m\}\ \forall r:(i,j) \in R \tag{6.5}$$

$$x_{jkt} \in \{0,1\} \qquad \forall j \in \{1,\dots,n\}\ \forall k \in \{1,\dots,m\}\ \forall t \in \{r_j,\dots,\min\left(C,\bar{d}_j\right)-p_j\}$$
$$y_{ij} \in \{0,1\} \qquad \forall r:(i,j) \in R$$

As searching for the optimal fixation using this ILP continues to be far too inefficient, the rationale for using the decomposition approach still stands. The next section describes how we adapt the local search approaches for the extended problems.

6.4 Extending the search algorithms

To incorporate the weighted relations into the genetic algorithm and simulated annealing approach, we only need to change their objective function into the following, and no other modifications have to take place:

$$\min \sum_{r \in R} -f_r w_r.$$

The better approximation of $\sum \gamma$ comes for free with the newly introduced additional correlations, as these immediately become part of the solution and are eligible for fixation.

6.4.1 Merging job-sets

The strategy for determining which sets of jobs to unify while combining fixations during the genetic algorithm, or while changing the current solution of the simulated annealing algorithm by merging sets used to be similar. The program would check whether merging two sets was possible by ensuring the sum of their processing times stayed within the maximum makespan. While this is still a valid criterion, we can extend that check by seeing whether we can construct a roster for the combined set of jobs, in which the starting times of all included jobs have a valid value and all precedence relations are honoured.

To this end we use the constraint programming implementation used by the new pricing heuristic introduced in section 5.4.3. Whenever two sets of jobs are selected for a merge, we invoke the CP solver to create a roster that includes all those jobs. If we manage to find a roster, we unify the two sets; otherwise we abort the merge — then we continue like before. Also, if a roster is returned, we immediately add it to the indicator LP, making solving it easier once we get to the evaluation of the fixation that is being generated.

Chapter 7

Additional search

During the evaluation of the solutions generated by the search algorithms, many single machine rosters are generated as columns for the indicator LP. In this chapter we show how we can use these columns to look for fixations in a different manner.

7.1 Combining columns

By combining all the single machine rosters we have generated while evaluating fixations, and selecting an appropriate subset of these, we might be able to find fixations the search algorithms have missed. With the goal of finding the best fixation, we thus have to set the objective to maximising the sum over the weights of relations that are fixated in each column. To achieve this, we add a parameter w_s to every known column s, that is calculated by summing up the weights of the relations of which both jobs are included, i.e. $w_s = \sum_{r \in \{r \mid r:(i,j) \in R \wedge a_{is} \wedge a_{js}\}} w_r$.

Using S' to indicate the set of known columns, we formulate the following integer linear programming problem. Although the objective is different, the first two constraints of this formulation are the same as the ones for the master problem of the indicator LP, described in section 5.4.1 — stating that all jobs should be included exactly once (7.1), and that the precedence relations are to be honoured (7.2). Additionally, we have to enforce that no more columns are chosen than there are machines available (7.3).

$$\max \quad \sum_{s \in S'} x_s w_s \quad \text{s.t.}$$

$$\sum_{s \in S'} x_s a_{js} = 1 \qquad \forall j \in \{1, \ldots, n\} \tag{7.1}$$

$$\sum_{s \in S'} x_s a_{js} S_{js} - \sum_{s \in S'} x_s a_{is} S_{is} \geq q_{ij} \qquad \forall r : (i,j) \in R_p \tag{7.2}$$

$$\sum_{s \in S'} x_s \leq m \tag{7.3}$$

$$x_s \in \{0,1\} \qquad \forall s \in S'$$

7.2 Implementation

Because this ILP needs the columns that were generated during the execution of the other search algorithms, it is not possible to run it separately. It could however come up with solutions that were missed by the local search heuristics, so running it afterwards might lead to improvements. When the genetic search and simulated annealing algorithms are done, all columns that were generated are passed to this approach. Should an improved solution be found, it is added to the genetic populations, to give them a better starting point for their next run.

To determine the effectiveness of this approach, we also backported its implementation to the original code by Van Roermund (2013) that worked on the simplified problems, to see how this extension improves the quality of found solutions. The results of these experiments can be found in section 9.3.5.

Part III

Results

Chapter 8

Parameters

Before putting the presented approach to the test, we first need to set appropriate values for its parameters. To do this, we run the program multiple times on a set of problems, each time with a different setting, and choose the best performing value as the default. The first section describes the problem instances that were used for these and further experiments. The remaining sections describe the various parameters that were tested, belonging to the different search algorithms.

8.1 Problem instances

The problem instances we used to test the approach on have all been randomly generated. Each instance is titled nJ-rR-mM, where n indicates the number of jobs, r the number of relations and m the number of machines — e.g. 30J-40R-8M is a problem with 30 jobs, 40 relations and 8 machines. The processing times were picked between 1 and 20, the release dates of jobs were randomly chosen from the interval $[0, \frac{n}{2}]$, and the deadlines were randomly set ranging from the release date plus the processing time to some big number. The precedence relations were added between randomly selected pairs of jobs, with weights of 1 and the value of q_{ij} randomly chosen between 0 and the processing time of the predecessor job, i.e. within $[0, p_j]$.

8.2 Stop count

The stop count parameter determines the number of iterations, in which no improvements to a solution are found, after which a search algorithm is stopped. This setting is therefore applicable to both the genetic and simulated annealing local search. Choosing a value that makes it come out too low would make the algorithm give up the search while there are still possibilities for improvements. If set too high however, it would keep searching for better fixations long after the best solution was found, only wasting time. We therefore expect there to be a certain threshold, above which the quality of solutions is no longer improved. To find the height of this threshold, we set the stop count to a very high value, and keep track of the number of iterations between improvements.

When looking at the found numbers, shown in figure 8.1, we can see that for the genetic algorithm, the threshold lies higher for instances with more machines — for the tested instances with only 2 machines, at most one iteration passed in which no improvement was made, whereas this was up to 70 iterations with 8 machines. For the simulated annealing algorithm we see a similar pattern, although the difference is larger — up to 330 iterations without improvements passed, after which a better solution was found. We therefore choose the value of this parameter to be $15m - 20$ for the genetic search and $60m$ for the simulated annealing algorithm. These values allow for a large enough stop count to not affect the quality of the solutions, while still keeping the total runtime of the algorithm in check.

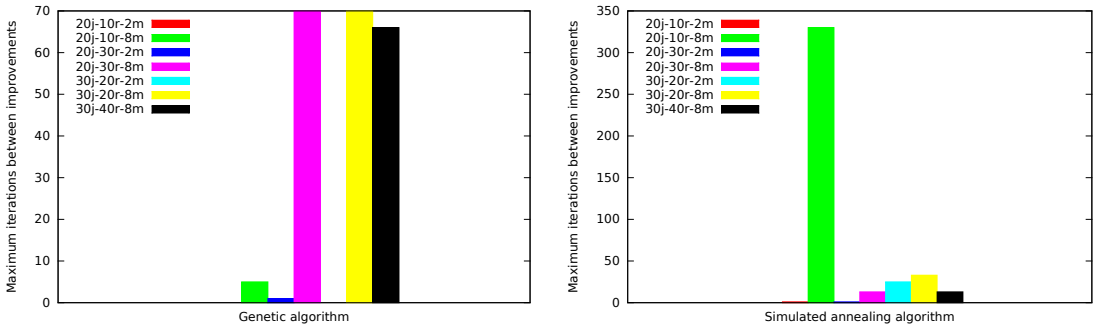


Figure 8.1: Experiments for the stop count parameter.

8.3 Genetic local search

In this section, we explore the performance of the genetic algorithm, by tweaking its parameters. During these experiments the simulated annealing algorithm was not used to improve the fixations found by the genetic algorithm, to get a clear view of the effect of these settings on the quality of the solutions. For each parameter we show two charts, demonstrating the effect of a setting on both the quality of the solutions and the runtime of the algorithm. The multi start approach was kept, each result represents the best of three separate runs; the time is the total time taken by those three runs.

8.3.1 Elitist selection

This parameter determines the number of best solutions in the population that are unconditionally copied to the new generation, as defined in section 4.1.1. Setting it to zero means that every iteration the entire population is thrown away after selecting the parents, and therefore provides no guarantee that the best solution of the next generation is as good as the current best. On the other hand, setting it to the size of the population minus one means that only the worst solution in the population is replaced during an iteration, and might smother the diversity of the solutions in the population. With a population-size of 10, the tested values for the elitist selection count are 0, 1, 3, 5, 7 and 9.

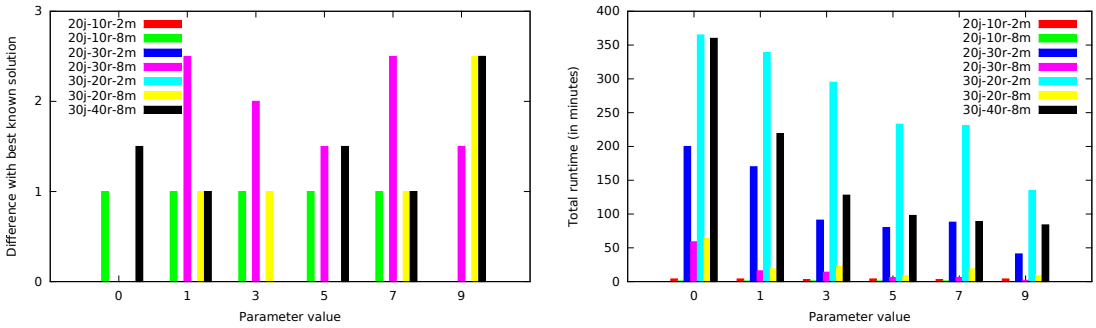


Figure 8.2: Experiments for the elitist selection count parameter.

As can be seen from the chart depicting the runtime in figure 8.2, the higher the value of this parameter, the faster the populations converge. However, the average distance to the best known solution also increases slightly with a higher setting. We therefore choose the middle ground, 5 — half the size of the population — as the default value for this parameter.

8.3.2 Combiner inclusion chance

When combining parent solutions to create new fixations, the common parts of the parents are copied with a certain chance, which is governed by this parameter. With a low value, only little information is transfered between generations, leading to a longer time before a population converges. However, a high setting might ensure that once some relation is fixated by all solutions in the population, no new solutions can be generated in which that relation is not fixated, thereby leading to premature convergence. The tested settings are 0.7, 0.8, 0.85, 0.9 and 1.

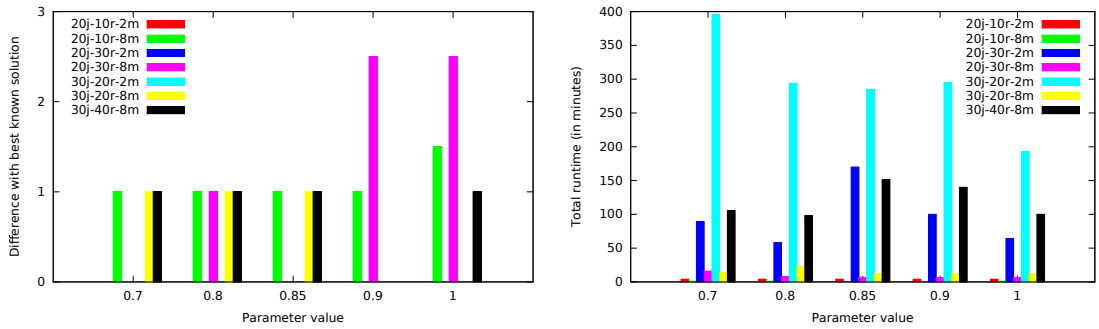


Figure 8.3: Experiments for the combiner inclusion chance parameter.

Figure 8.3 shows how setting the combiner inclusion chance to a higher value indeed leads to a faster converging population. The distance to the best known solutions also increases when the value approaches 1, confirming our assumption. We therefore choose 0.8 as the default value for this parameter, as it has one of the lowest runtimes and still few errors — off-by-one errors can be explained due to the random nature of the search algorithms, making this the best setting.

8.4 Simulated annealing

We also investigate the performance of the simulated annealing search algorithm, by changing its parameters. Since the simulated annealing algorithm is normally intended to be run after the genetic search, we do these experiments by running simulated annealing on a fixed solution that is somewhat close to the best known fixation for each instance.

8.4.1 Cooling schedule

As described in section 4.2.2, the cooling schedule governs the temperature during the run of the simulated annealing algorithm, and consists of a starting temperature, a modifier and the cool count, i.e. the number of iterations after which the current temperature is decreased by multiplying it with the modifier. Setting the temperature too high, or not lowering it enough, might make the search slow to converge, whereas starting off with a very low temperature, or decreasing it too fast, will allow the algorithm to explore only a small part of the search space.

In section 4.2.1 we introduced two different changes for modifying fixations, i.e. merge two sets of jobs or scattering one set. Since only the latter can — and always will — lead to a decrease in the fitness value of the current solution, this is the only occasion the temperature will be needed. As the sizes of the sets that are scattered are proportionate to the number of relations in the problem instance, we make the starting temperature dependent on that number. The tried settings are 0.5-0.9-25, 0.5-0.95-10, 1-0.9-25 and 1-0.95-10, where a-b-c indicates $a \cdot |R|$ as the starting temperature and b as the modifier that is applied to the current temperature every c iterations.

The charts in figure 8.4 show how a higher starting temperature leads to a longer runtime, but also to slightly worse results. Lowering the temperature more often but more gradually seems to slightly decrease the effectiveness, although the effect is not very large. We therefore pick 0.5-0.9-25 as the default cooling schedule, as it is the fastest and has the best results.

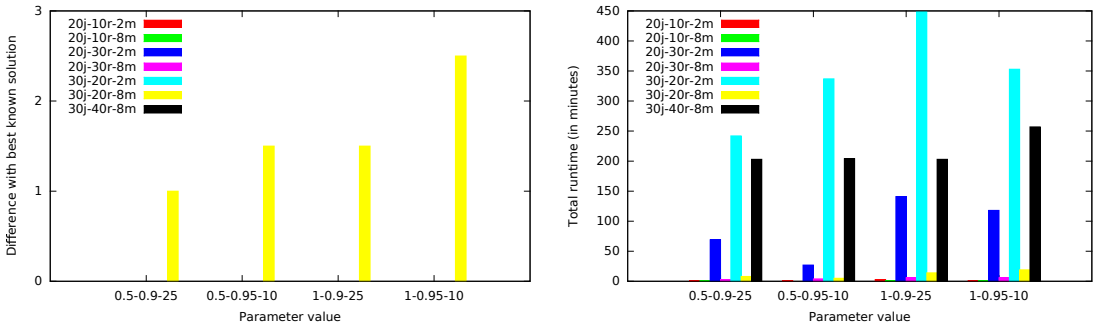


Figure 8.4: Experiments for the cooling schedule parameter.

8.4.2 Changer ratio

As described above, we defined two changes to fixations during the simulated annealing search algorithm. Since the scattering of a set turns every job in that set into a singleton, it would take the size of that set minus one merges to return to the original solution. And as the size of the sets of jobs is expected to grow with the size of the problem, we make the ratio for choosing between the two changes dynamic and dependent on the number of relations in the problem instance. This parameter only influences the initial choice of which change to apply; if there are no sets to merge or the scattering of a set is rejected based on the current temperature, the other change is tried anyway. The tried values are 0.2, 0.5 and 1, where a value r results in a $r \cdot |R| : 1$ ratio for choosing the merging change.

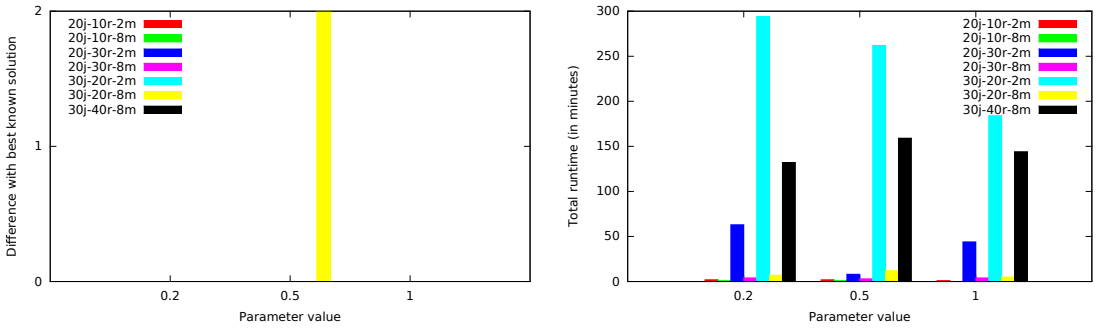


Figure 8.5: Experiments for the changer ratio parameter.

Figure 8.5 shows how different values for this parameter make little difference to the effectiveness of the search algorithm. We thus pick the value 1 as the default setting, as that has the smallest average runtime, although other values might work as well.

8.5 Pricing heuristic retries

The last parameter we investigate controls the application of the pricing heuristic. When solving the indicator LP to evaluate the feasibility of a fixation, we use the shadow prices of the current solution to generate a new column. This can either be done by the optimal approach, which is another ILP by itself, or the pricing heuristic, which solves the LP relaxation of that ILP. After finding a solution to the latter, the heuristic tries to construct a column by rounding the relaxed variables either up or down, and, to maximise the chance of finding a column that improves the solution of the indicator LP, it does so multiple times. This parameter controls the number of times the pricing heuristic will try to build a column from the result of its LP relaxation.

We test this parameter by measuring the time needed to evaluate a single fixation that is close to the best know solution. The tested values are 0, 1, 3, 5 and 9, where 0 means that the pricing heuristic is completely skipped and only the optimal solution to the pricing problem is determined. Each experiment has been repeated five times; the results are displayed in figure 8.6, where the colored bars represent the averages and the black error bars show the minimal and maximum results.

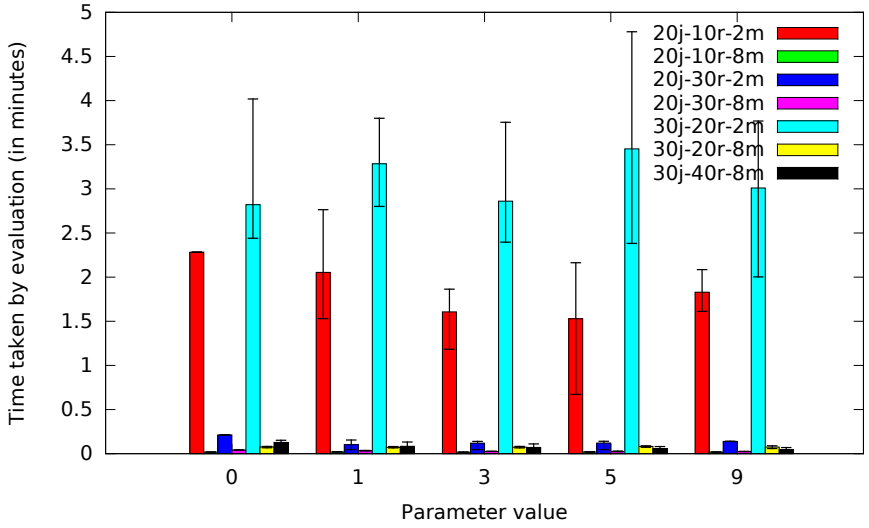


Figure 8.6: Experiments for the pricing heuristic retry parameter.

We can see that not using the pricing heuristic at all, i.e. setting this parameter to 0, still gives us a decent running time for evaluating solutions. However, setting it to 3 gives the overall quickest results, so we will choose this as the default value. Another noticeable thing from this chart is the fact that evaluating a fixation for a problem instance with many machines takes much less time than doing so for an instance with only few machines. We will return to this observation when discussing the results in the next chapter.

Chapter 9

Experiments

This chapter describes the experiments that we carried out to prove the effectiveness of the presented approach. We start with a description of the simplification of the problem instances we needed to compare with optimal solutions, followed by the detailing of the various statistics that were gathered during the execution of the program. The results are presented in the last section, combined with an interpretation and explanation of the tables that can be found in the appendices.

9.1 Simplified problem instances

In order to determine the effectiveness of our approach, we need to compare the solutions it finds to the optimal ones. Unfortunately, running the ILP defined in section 6.3 on the generated problem instances proved intractable — trying to solve 20J-30R-2M to optimality crashed after more than 1000 minutes, because it had exhausted all the 8GB of available memory in the system running the experiments. We therefore decided to generate a second set of instances, which were simplified in order to determine the optimal solutions. For these problem instances, the processing times of all tasks were set to 1, the release dates to 0 and the deadlines equal to the number of jobs. The q_{ij} values of precedence relations were still randomly chosen from the range $[0, p_j]$, but because of the unit processing times this becomes $[0, 1]$. We refer to these simplified instances as nJ-rR-mM-SIMPLE.

9.2 Statistics

Various statistics have been gathered from the experiments, which are presented in the tables found in the appendices. In the first set of tables, each instance is presented using up to four rows, each one with a different value for the maximum makespan. The tables start with the following three columns:

Problem	The name of the instance;
C	The maximum makespan used when solving the problem;
Best	Indicates the value of the best solution found for this C;

Timing statistics

The next 9 columns contain information on the amount of time spent on operations:

Total	This indicates the total time spent on the indicator LP, including the next 6 columns;
Solving	This is the time taken by the Gurobi solver on the indicator LP;
Pricing: Heuristic	The time spent on solving the pricing problem heuristic;

Pricing: Optimal	The time spent on solving the ILP implementation of the pricing problem;
Columns: −	The time spent removing columns at the end of evaluating a fixation;
Columns: ±	The time spent re-adding columns that were previously removed;
Columns: ↻	The time spent updating columns at the start of evaluating a fixation;
CS	The time taken by the column-based search of chapter 7;
ILP	Indicates the total time taken by solving the verifier ILP.

Counting statistics

The last 9 columns contain statistics on how many times operations were executed:

Evals	The number of fixations that were evaluated by the indicator LP;
Iters	The total number of iterations made by the indicator LP;
Pricing: Heuristic	The number of times the heuristic for the pricing problem returned a new column, over the total number of times the pricing problem was called;
Pricing: Optimal	The number of times the integer linear programming implementation of the pricing problem was successfully solved, over the total number of times it was called;
Columns: +	The total number of unique columns added to the indicator LP;
Columns: −	The total number of times a column was removed;
Columns: ±	The total number of times a column, that was previously removed, was re-added;
LB	The number of evaluations that were aborted because the lower bound was higher than the number of available machines;
ILP	The number of fixations checked by the verifier ILP.

9.3 Results

When looking at the tables in the appendices, the following things can be noticed.

9.3.1 Effectiveness of the approach

Appendix A, starting at page 50, contains the tables with the statistics of the experiments as described above. One thing that attracts the attention is the last column of each of those tables, showing the number of times the verifier ILP was called, which mostly contains ones — even more so for the simplified instances. This means that the verifier ILP is able to prove the feasibility of the best fixation found by the search algorithms most of the time, and thus that the indicator LP does not return many false positives when evaluating a fixation. Because of its formulation as a LP-relaxation, we are ensured that it can never return false negatives, i.e. it will never claim a fixation is not feasible while in fact it is. We can thus conclude that the assumption made in section 3.2 — that finding a solution to the LP, which minimises the number of used machines, is a good indicator of feasibility — is valid for these problems. The fact that the verification step succeeds does not mean that the verifier ILP can be skipped however, for it is this step that actually generates a schedule, whereas the evaluation LP uses fractions of columns to determine the alleged feasibility of a fixation and does not build a complete roster.

The few instances where the ILP had to be invoked many times before finding a feasible fixation though, the time taken by those checks is immense compared to when run only once — when a solution is being verified, all possible schedules need to be checked before we can conclude its infeasibility, which can take a lot of time. It would therefore be good to ensure the number of possibly infeasible solutions that are checked is minimised when implementing this approach for real problems — see section 10.3.3 for more thoughts on this.

Also evident from these tables, is the application of the intermediate lower bound on the indicator LP — topping at 4628 times for the first run of 40J-50R-8M — and the effectiveness

of the pricing heuristic. When looking at for example the eighth entry of table A.1, i.e. 20J-30R-2M with $C = 120$, we see it takes 297072ms for 3030 runs, against 8569449ms for 1502 runs of the optimal pricing problem — meaning the heuristic is almost 60 times faster to run, which corresponds to the other entries in the table. Note that this does not mean that the columns found by the heuristic implementation of the pricing problem actually improve the solution of the LP — we looked at this earlier in section 8.5, and came to the conclusion that including the heuristic definitely improves the runtime of the approach.

9.3.2 Evaluations with zero iterations

In the statistics tables, rows can be found where the number of iterations of the indicator LP is very low. This is possible because the program first tries to solve the LP using the columns that have already been generated during previous evaluations. If this succeeds and the number of machines needed is small enough, no iterating needs to be done, as we already deem the given fixation feasible. The number of evaluations is usually quite low in those cases as well, which can be explained by the fact that the program also caches the evaluation results, meaning fixations that have previously been checked and deemed feasible do not have to be evaluated again — solutions that were deemed infeasible are also cached, but cleared when the value of C is increased, as those fixations then could be feasible. The low values of both the number of iterations and evaluations indicate that the search algorithms were unable to find any fixations that were an improvement over the best known solution at that stage, most likely because they were at a (local) optimum.

9.3.3 Comparison of results

The next set of tables, found in appendix B at page 53, displays the comparison of solutions found by our approach and the optimal ones. The cells showing the value of the found solutions have been colour-coded to allow for a quick overview of the divergence from the optimal value. A green cell indicates that the optimal solution was found, a lime one means there was a difference of 0.5 or 1; yellow means that the search algorithm could find a value of up to 2 less than the optimal value; orange and red cells indicate differences of up to 3 and more than 3 respectively. Note that these tables only contain the simplified problem instances ($p_j = 1, r_j = 1, \bar{d}_j = n$).

Without exceptions, the heuristic approach is able to come to, or very close to, the optimal value for all instances. The minor offsets can be explained by the random nature of the local search algorithms, and re-running the search might lead to the optimum being found that time. This means that our approach is a very good alternative to the ILP that solves problems to optimality.

9.3.4 Comparison of running times

In those same tables, the total time it took to find the solutions is provided for comparison between runs of the same algorithm, to show which instances are harder than others. Comparing the times of the optimal search and the heuristic can be misleading, as the stopping criteria for the local search algorithms have been set in a very general way. For tackling a specific problem, the parameters of the genetic and simulated annealing algorithms could be tailored to minimise the running time; or we could give it a target value that is no higher than the optimal value, so it can stop searching after finding a good enough solution.

Note that for smaller instances, solving it to optimality still is more efficient — for larger instances however, especially those with many machines and more precedence relations than jobs, the tables show that the running time of the ILP drastically increases, quickly making it intractable. For example, have a look at the running time of the optimality ILP for the instances 30J-40R-*M-SIMPLE; it perfectly shows how incrementing the number of available machines increases the time needed to find an optimal solution.

The running time of the heuristic approach also grows with the size of the instances, but is less dependent on the number of machines. The exception to that is when there are only two

machines, in which case the search algorithms need way more time to come up with a solution. When looking at the 30J-40R-*M-SIMPLE instances, we see that the time needed to run the search algorithms actually decreases when the number of available machines grows. Although this does not show for the other simplified instances, it is a pattern that is better visible in table A.1 showing the statistics for the non-simplified problem instances, where the larger instances with only few machines could not be solved within 24 hours, whereas those same instance with more machines could. This is most probably caused by the indicator LP being able to more easily combine columns when the number of allowed machines is higher, leading to a quicker conclusion on the feasibility of a fixation.

Another advantage of our approach is the fact that it needs less system resources to run than the ILP implementation — whereas the latter often ran out of memory on larger instances, our approach never needed more than 1GB of memory, even for the largest instances that were executed — meaning much larger problems can be tackled before running into hardware limitations. These observations, along with the conclusion that our approach comes very close to the optimum solution, makes it a most adequate alternative to the ILP, especially for larger instances with many machines.

9.3.5 The improved genetic algorithm

The tables in appendix C, at page 55 and onwards, show the results of the improved genetic algorithm as presented in chapter 4, compared to the naive method introduced by Van Roermund (2013) — with and without the column search ILP of chapter 7 — and the optimal solution for each problem instance. These experiments were run on the original problem instances, with the relations still as mere correlations, and are therefore not comparable to the instances discussed earlier. The format used to indicate them, i.e. nJ-rR-mM and nJ-rR-mM-SIMPLE, is identical; also here the simplified versions have all processing times set to 1.

The goal of these experiments is to show the effectiveness of the improvements presented in this paper. Each table displays the time it took to find the solutions and the number of relations that were fixated in those solutions. Table C.1 holds the results for problems with 10 jobs, table C.2 does the same for instances with 25 jobs, and table C.3 has those with 60 jobs. The columns showing the number of found fixated relations are colour-coded in the same way as the previously discussed tables.

The first thing that draws the attention here is the fact that the improved genetic algorithm always finds an optimal solution, and does so quicker than the old naive method came up with an answer. Only for small instances the ILP that solves the problem to optimality is quicker, but its running time explodes when tasked to solve larger problems, proving the improved GLS much more stable in its running time.

The results of the naive algorithm were taken directly from Van Roermund (2013), to give a basis for comparison. Extending that heuristic with the column based ILP of chapter 7 shows some merit, but it never comes close to the improved version. The only major differences it shows are when the naive approach fell short of the optimal solution by a long way — i.e. instances 25J-60R-2M and 60J-150R-*M — where the extension is able to make a relatively big improvement, but is still far from the optimum.

The performance of the column-based search in the experiments displayed in appendices A and B was comparable to this, i.e. only when the search algorithms failed to come up with a reasonably good solution, this column-based ILP could present an improving solution; otherwise it just came to the same fixation as the searches had returned. Looking at the difference in the running times for similar sized problems, e.g. 60J-60R-8M of both appendix B and C, we can see how adding precedence relations and job availabilities makes the problems so much harder to solve — both optimally and heuristically.

Chapter 10

Conclusion

In this final chapter we give a summary of the entire approach as presented in this work, and we draw the conclusions from the results of the experiments. The last section gives pointers for future research into this field.

10.1 Summary of the approach

In this thesis we set out to construct an approach for maximising the robustness of schedules. We have done this by fixating relations and thereby forcing the involved jobs to be executed by the same machine. To traverse the search-space and find the best fixations, we defined a genetic local search algorithm and a simulated annealing approach. Any found solution was then evaluated by a linear programming formulation based on column generation, which tries to minimise the number of machines needed to schedule all jobs while enforcing the fixation. By re-using the solver for that LP, we were able to quickly evaluate thousands of solutions, and identify those that give the highest robustness.

In summary, the complete approach for finding a schedule that maximises robustness using the presented method works as follows:

1. We determine the value of C by solving $\mathrm{P}|r_j, \bar{d}_j, prec|C_{\max}$;
2. We run the search algorithms, thereby tackling $\mathrm{P}|r_j, \bar{d}_j, prec, C_{\max} \leq C|\sum -f_r$;
   - Every iteration of the search algorithms generates one or more fixations, which are evaluated using the indicator LP;
     - Every iteration of the indicator LP, we solve the pricing problem to generate new columns;
     - When the indicator LP has been solved, we draw a verdict on the feasibility of the evaluated fixation;
   - After a set amount of iterations in which no improvement was made, the search algorithms return the best solution encountered;
3. We solve the verifier ILP to check whether the returned solution is feasible — if it is not, we try the next best solution, until we have found a fixation that is;
4. Optionally, we increase the value of C and repeat from step 2, until we have a good idea of the interplay between the maximum makespan and the achieved robustness.

10.2 Conclusion

Following the interpretations of the results of the experiments in the previous chapter, we can conclude that the presented approach performs very well. Although it is not very quick on smaller problems, because of the overhead of the local search algorithms, and it can take a lot of time to solve larger instances with only a few machines, it excels at solving problems with many machines. Seeing that those instances are also the weak spot of the ILP that solves them to optimality, because of the symmetry in its formulation, this makes our approach an excellent heuristic for solving those problems.

10.3 Future work

In order to enhance the usefulness of the approach presented in this thesis, there are several extensions that are worthwhile to investigate in future works.

10.3.1 Other types of precedence relations

As mentioned in section 5.1, we only looked at precedence relations of the form $S_j - S_i \geq q_{ij}$, with $q_{ij} \geq 0$, and did not consider other types of relations. Ignoring the constraint that q_{ij} is positive for a moment, we could rewrite the other types quite easily, making them all alike:

$$\begin{aligned} & & S_j - S_i &\geq q_{ij} \\ S_j - S_i &\leq q_{ij} \quad \Rightarrow & -S_j + S_i &\geq -q_{ij} \\ S_j - S_i &= q_{ij} \quad \Rightarrow & -S_j + S_i &\geq -q_{ij} \ \wedge \ S_j - S_i \geq q_{ij} \end{aligned}$$

However, instead of simply including these rewritten forms into the constraints, we could come to a better approach by taking a closer look at what these relations actually indicate. The included precedence relations, with positive values for q_{ij}, signify a minimal time between the starting times of jobs i and j. The second type indicates the opposite, i.e. a maximum time between the starting of jobs i and j. And the last of these relations, with their equality condition, demand a precise difference in the starting times of involved jobs. This makes the last type much harder to satisfy from the viewpoint of robustness, as the possibility of delay propagations now work both ways — a delay of the execution of job i can have as much impact on a schedule as the postponing of job j. As expected, the second type has the delay propagations working only one way, but opposite from the type that was implemented — whenever job j is delayed and there is no more slack time, we also need to postpone job i, even though it is scheduled before j, which might not even be possible, as it could have already started, making it impossible to fulfill that constraint.

It would therefore make sense to treat these relations differently when fixating them, to take full advantage of their meaning. Seeing that the third type poses the most restrictions on the schedules, it would be best to first fixate as many of these relations as possible. Since the second type can be harder to overcome when delays actually take place, we would want to fixate these next, before moving on to the first type. A weighted sum over the value of these fixations might be useful as an objective function, although the exact interplay between the terms should be looked into.

10.3.2 Specific problem instances

As mentioned in section 9.3.4, we could significantly improve the running time of the program when targeting a specific problem. The problem instances in this work were chosen randomly, with a wide range of parameters, to investigate the effectiveness of the approach on various types of problems. Knowing that our heuristic is most effective on larger instances with many machines, this problem-type can be further explored, to get the most out of the strengths of this method. Trying our approach on some real-world instances might have some value as well, and give more insight in its applications.

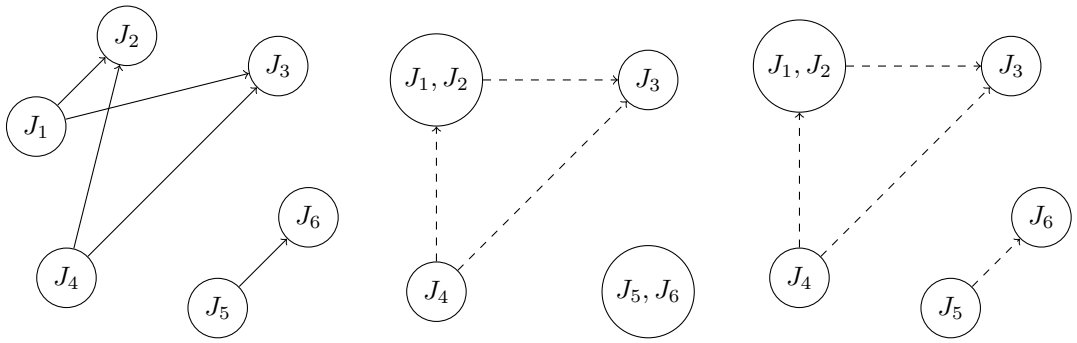
Also, the precedence relations of the instances that were experimented on have all been added between random jobs, leading to no specific structure. As an alternative, we could generate these relations in such a way that it forms a tree — possibly an in-tree or an out-tree — or even other types of graphs. The performance of the presented approach might be affected differently by various types of dependency graphs, which makes for another interesting piece of research.

10.3.3 Extending the verifier ILP

The verifier ILP, as described in sections 3.1 and 5.3, is invoked after the search algorithms return a list of the solutions found, to determine whether they are actually feasible. As noted in the previous chapter, when many fixations turn out to be infeasible, the process of finding one that is can take a lot of time. This part of the approach could be accelerated by preprocessing the ILP model and thereby reducing the search space.

Instead of adding a variable for scheduling each job on each machine, we could for make a graph. In this graph, we merge jobs that are fixated together into a single node, and add edges for every relation that is not fixated. Since we are verifying the feasibility of the best fixation found, we assume no trivial improvements are possible, i.e. we are not able to improve the solution by simply fixating any single relation that was not already fixated. Using this assumption, we know that the edges in the graph we just created indicate that the connected sets of jobs must be scheduled on different machines. And since the machines are identical, it does not matter which machine execute which jobs. We can therefore simply restrict the jobs in a node to a specific machine, and the jobs of a connected node to another machine, etcetera. Each job still needs its own set of variables, as the precedence relations still have to be enforced within the ILP, but restricting some jobs to only one machine significantly diminishes the search space, leading to a faster conclusion on the feasibility of the evaluated fixation.

Let us look at the example in figure 10.1 with 6 jobs and 5 precedence relations — $(J_1 \to J_2)$, $(J_1 \to J_3)$, $(J_4 \to J_2)$, $(J_4 \to J_3)$ and $(J_5 \to J_6)$. For the fixation $[1, 0, 0, 0, 1]$, the graph described above is shown in figure 10.1b. When constructing the ILP, we would restrict jobs 1 and 2 to the first machine, job 3 to the second and job 4 to the third machine. Jobs 5 and 6 can be scheduled on any machine, although we still demand that they end up on the same one. Doing so decreases the possible schedules the ILP solver can consider while solving this instance, lowering its runtime, but does not create a chance of a false negative — if a feasible schedule is possible for the given fixation, it can still be found after this preprocessing step.



(a) Problem instance.

(b) Preprocessed graph for the fixation $[1, 0, 0, 0, 1]$.

(c) Preprocessed graph for the fixation $[1, 0, 0, 0, 0]$.

Figure 10.1: Example problem with 6 jobs and 5 precedence relations.

Note that when the preprocessed graph is a forest, like in figure 10.1c, we can only restrict the jobs of one of its trees to specific machines. To decrease the search space as much as possible, we should select the largest tree for this, and add some additional constraints that demand that jobs involved in non-fixated relations are not scheduled on the same machine.

Bibliography

Van den Akker, J.M., J.A. Hoogeveen, and J.W. van Kempen (2012). "Using column generation to solve parallel machine scheduling problems with minmax objective functions". In: *Journal of Scheduling* 15.6, pages 801–810.

Bazaraa, M.S., J.J. Jarvis, and H.D. Sherali (2005). *Linear programming and network flows*. 3rd. Wiley.

Galler, B.A. and M.J. Fisher (1964). "An improved equivalence algorithm". In: *Communications of the ACM* 7.5, pages 301–303.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. 1st. Addison-Wesley Professional.

Graham, R.L., E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan (1979). "Optimization and approximation in deterministic sequencing and scheduling: a survey". In: *Annals of Discrete Mathematics* 5.2, pages 287–326.

Gurobi (2009). *Gurobi Optimizer*. `http://www.gurobi.com/products/gurobi-optimizer/`.

Hoogeveen, J.A. (2005). "Multicriteria scheduling". In: *European Journal of Operational Research* 167.3, pages 592–623.

Hoppenbrouwer, D.J. (2011). "Robust parallel machine scheduling with relations between jobs". Master's thesis. Universiteit Utrecht.

IBM (2009). *ILOG CPLEX Optimization Studio*. `http://www.ibm.com/software/integration/optimization/cplex-optimization-studio/`.

De Jong, K.A. (1975). "Analysis of the behavior of a class of genetic adaptive systems". PhD thesis.

Kirkpatrick, S., C.D. Gelatt Jr., and M.P. Vecchi (1983). "Optimization by simulated annealing". In: *Science* 220.4598, pages 671–680.

Lenstra, J.K. and A.H.G. Rinnooy Kan (1978). "Complexity of scheduling under precedence constraints". In: *Operations Research* 26.1, pages 22–35.

Van Roermund, D.J. (2013). *Robust scheduling of correlated jobs*. Experimentation project, Universiteit Utrecht.

Part IV

Experimental data

Appendix A

Statistics

See section 9.2 for the description of these tables. If less than three rows are shown for an instance, it means that the heuristic approach was able to find a solution with all relations fixated, so increasing the maximum makespan will not allow for any improvements. The empty rows, containing only *n/a*, belong to instances that could not be run within 24 hours — tuning the parameters of the search algorithms to each specific instance would lead to better results, but for these experiments we only used the settings as presented in chapter 8.

Problem	C	Best	Timing statistics (in ms)									Counting statistics								
			LP							CS	ILP	LP								ILP
			Total	Solving	Pricing		Columns					Evals	Iters	Pricing		Columns			LB	
					Heuristic	Optimal	−	±	↻					Heuristic	Optimal	+	−	±		
20J-10R-2M	107	11.5	194555	41	7992	186466	4	2	4	18	1713	8	110	64/112	48/48	187	208	23	0	1
20J-10R-5M	45	2.0	100915	1143	22986	75845	131	560	26	135	148346	563	798	215/1241	590/1026	3289	99882	96635	14	92
	46	5.0	98594	911	20924	76087	98	383	14	92	33004	427	790	181/1161	619/980	2799	68116	65305	40	7
	47	6.5	84931	648	15447	68423	56	218	18	52	304	365	557	118/848	446/730	2160	43211	41048	85	1
20J-10R-8M	43	5.5	81508	968	22497	57228	115	453	37	109	213	477	795	218/1214	578/996	3099	79846	76789	20	1
	44	5.5	83280	948	23256	58403	85	386	16	87	276	410	860	287/1270	573/983	3177	74232	71057	33	1
	45	6.5	79727	860	18132	60176	61	327	25	61	275	463	668	191/995	483/804	3159	67539	64381	24	1
20J-30R-2M	120	38.0	8870069	1394	297072	8569449	127	1364	32	1524	1029	1256	2707	1528/3030	1251/1502	4167	51060	46926	32	1
	122	38.0	775206	197	37712	737159	8	50	10	59	1212	509	331	226/382	127/156	717	1827	1080	2	1
	124	38.0	475735	140	33672	441819	10	18	14	54	1284	546	279	208/319	92/111	642	1607	990	3	1
20J-30R-5M	51	15.0	1278450	10052	225085	1017431	1223	22655	280	790	83516	4771	5570	1459/9084	4211/7625	14548	1029171	1014639	287	222
	52	18.5	609198	3080	124902	475296	278	4670	142	266	7256	2515	2919	860/4917	2143/4057	6088	165399	159308	286	20
	53	22.0	751995	4722	145527	590907	516	9062	186	526	308	3658	3324	1131/5558	2337/4427	8539	382075	373536	220	1
20J-30R-8M	46	16.5	895939	11632	189468	668026	1196	23656	350	391	368	6683	4353	872/8754	3538/7882	14295	1034199	1019925	460	1
	47	17.0	570503	5002	124799	431608	493	7455	231	222	1169	3992	2735	596/5421	2193/4825	8975	315595	306614	238	3
	48	17.0	743062	5546	149159	576514	592	9924	229	176	793	4360	3153	537/6285	2672/5748	9652	384311	374663	288	2
30J-20R-2M	158	22.5	19460433	3344	965166	18487647	210	2720	26	2864	19048	866	5223	3102/5451	2199/2349	6587	140251	133666	58	1
	161	22.5	9633308	2353	571909	9056489	135	1615	33	1170	3755	668	2884	1888/3078	1106/1190	3694	108543	104859	19	1
	164	22.5	7091189	1926	420450	6666600	113	1516	27	647	3407	566	2030	1298/2250	859/952	2833	85471	82641	15	1
30J-20R-5M	64	17.5	4303843	8241	208329	4072534	1095	12387	142	922	8433	2952	3826	1900/5216	2148/3316	10310	1178101	1167811	394	1
	65	18.5	1784103	3802	92801	1682607	404	3894	89	319	9241	1849	1666	848/2334	1007/1486	6456	403288	396825	146	1
	66	19.5	1140813	2170	57600	1078655	195	1785	78	113	3440	1431	996	674/1440	477/766	4633	185657	181041	203	1
30J-20R-8M	41	8.5	988917	27893	217806	703262	4537	33316	502	591	196054	9055	2618	1199/10148	1502/8949	26154	4783790	4757660	1140	15
	42	11.5	511945	10320	115507	373531	1461	10106	271	186	1297	5023	1475	714/5207	823/4493	13327	1382750	1369423	568	1
	43	12.5	299602	5651	71010	216381	680	5159	178	134	2410	3449	1017	553/3202	534/2649	8425	674807	666381	453	1
30J-40R-2M	*n/a*																			
30J-40R-5M	64	38.0	39699022	27044	1728037	37867941	2172	68043	402	3418	2117	4805	17672	5585/21555	12143/15970	25572	1770719	1745165	557	1
	65	39.0	14809415	8017	811360	13972403	794	14505	228	1447	1501	2705	6819	1597/9172	5256/7575	9658	487075	477419	510	1
	66	39.0	10579574	5026	582188	9982867	367	7426	176	801	2003	1865	4969	1098/6555	3906/5457	7544	235554	228015	359	1
30J-40R-8M	64	38.0	44328545	31198	1702216	42508280	2585	78326	453	3680	2030	5210	17772	5289/21756	12534/16467	29295	2199347	2170072	485	1
	65	39.0	18419460	10729	961985	17422135	917	20784	264	2264	1714	3389	8851	1848/11749	7033/9901	13392	635855	622465	558	1
	66	39.0	15496184	7777	813594	14659014	744	12988	229	1099	2077	2645	6321	1418/8436	4944/7018	11884	419705	407828	645	1
40J-30R-2M	*n/a*																			
40J-30R-5M	*n/a*																			
40J-30R-8M	50	23.5	3783038	37004	367777	3291921	5260	77726	712	1515	10892	10874	5777	2332/11688	3869/9356	27158	5332939	5305806	1527	1
	51	24.5	1216352	12569	142707	1040154	1522	17913	407	452	4831	5951	1984	934/4347	1334/3413	14320	1508957	1494633	570	1
	52	24.5	848504	8684	111860	715708	907	10228	339	252	10889	5274	1230	746/3515	721/2769	12117	865580	853464	1298	1
40J-50R-2M	*n/a*																			
40J-50R-5M	*n/a*																			
40J-50R-8M	58	30.0	16735995	133379	1449854	14733347	16163	392095	2391	6362	3981555	17680	18264	7103/30934	11605/23831	57649	15080187	15022562	4628	84
	59	33.5	7613978	36476	670006	6806478	5024	90826	1525	2451	1484354	10530	6470	2364/13700	4468/11336	24945	4029580	4004631	2888	21
	60	34.5	3474190	15967	357724	3063712	2085	32045	888	895	14521310	6650	2947	1364/7032	1842/5668	15281	1630183	1614899	1559	108

Table A.1: Statistics for non-simplified problem instances.

Problem	C	Best	Timing statistics (in ms)									Counting statistics								
			LP							CS	ILP	LP								
			Total	Solving	Pricing		Columns					Evals	Iters	Pricing		Columns			LB	ILP
					Heuristic	Optimal	−	±	↻					Heuristic	Optimal	+	−	±		
20J-10R-2M-SIMPLE	10	10.0	6697	240	4545	1632	35	147	8	84	27	304	307	243/324	81/81	822	18290	17475	0	1
	11	11.0	104	2	100	0	0	0	0	3	30	1	8	9/9	0/0	23	16	0	0	1
20J-10R-5M-SIMPLE	4	6.5	14811	791	9719	3650	120	351	25	57	20	599	487	154/855	361/701	2237	66310	64108	25	1
	5	8.5	6155	351	3726	1871	33	98	15	61	20	372	282	75/358	227/283	1623	21374	19741	2	1
	6	9.5	3517	240	2190	971	19	54	9	20	22	229	155	73/193	105/120	1544	10183	8664	0	1
20J-10R-8M-SIMPLE	4	6.5	14096	838	8614	3976	99	375	31	53	32	678	448	74/795	379/721	2980	63591	60639	29	1
	5	8.5	7561	484	4468	2338	26	146	19	70	33	440	331	77/437	276/360	2561	27260	24697	1	1
	6	9.5	3496	242	2221	894	14	62	10	24	36	237	145	62/182	105/120	2162	10243	8113	0	1
20J-30R-2M-SIMPLE	10	32.0	160996	1834	84902	72090	81	1343	44	1482	41	1442	3129	2142/3446	1138/1304	4810	40520	35714	0	1
	11	32.5	83576	999	47368	34418	32	311	20	724	55	1098	2036	1526/2169	622/644	2805	13637	10836	3	1
	12	33.0	86583	1012	59965	24765	25	367	37	488	56	956	1773	1464/1968	502/504	2490	14253	11761	0	1
20J-30R-5M-SIMPLE	6	23.5	396471	10081	150323	215208	755	17712	206	880	33	3643	8208	1888/10400	6456/8512	14660	601136	586495	5	1
	7	26.0	168095	3351	66452	92241	228	4745	122	619	291	2370	3630	1186/4646	2633/3460	7700	139783	132073	28	3
	8	28.5	179006	3260	83775	85845	251	4775	150	848	58	2773	3299	1617/4316	2068/2699	7925	140290	132371	3	1
20J-30R-8M-SIMPLE	6	23.5	507836	14262	191435	268326	1129	29561	277	1253	47	4734	10494	2185/13349	8459/11164	19802	951774	931988	13	1
	7	26.0	206823	5490	78231	111776	432	9612	188	854	451	3800	3856	1381/5240	2698/3859	12372	341171	328796	12	3
	8	28.5	166283	3560	79364	76341	273	5524	179	960	99	2949	3238	1594/4355	2051/2761	9494	186381	176888	4	1
30J-20R-2M-SIMPLE	15	21.5	4307	47	2870	1317	4	14	3	29	102	16	110	95/118	23/23	253	552	305	0	1
30J-20R-5M-SIMPLE	6	18.5	26090	2324	15144	6085	245	1928	86	192	101	2027	747	621/1168	300/547	5943	208732	202809	25	1
	7	19.5	5289	1142	2840	471	76	590	54	89	60	1463	157	196/232	36/36	4587	56912	52320	0	1
	8	19.5	5173	1258	2556	515	79	610	68	78	115	1365	137	174/206	32/32	3998	54519	50521	0	1
30J-20R-8M-SIMPLE	4	17.0	40471	4087	20573	12362	419	2513	191	113	55	4093	572	107/1698	488/1591	11600	333243	321661	65	1
	5	18.0	24251	2574	11815	7755	202	1512	142	76	55	2941	505	101/956	432/855	9627	172720	163089	66	1
	6	18.5	10861	1678	4858	3140	117	852	83	51	84	2297	216	99/379	165/280	8260	89010	80749	20	1
30J-40R-2M-SIMPLE	15	48.5	1763274	8007	844553	902748	179	5271	73	8436	1493	1688	10512	7098/10885	3690/3790	13259	100992	87739	56	6
	16	49.5	940203	5675	614860	315450	95	2249	59	4409	137	1297	7486	5824/7803	1960/1983	9017	48884	39870	6	1
	17	50.0	692340	3444	414995	271149	65	1572	55	3031	151	1077	4617	3643/4910	1255/1268	5734	38234	32505	1	1
30J-40R-5M-SIMPLE	6	32.5	1120624	44462	361868	612720	3308	92632	637	3285	78	7545	16571	3929/21931	13430/18002	30660	3117501	3086846	1268	1
	7	34.5	310182	5886	105666	188767	454	7903	256	1632	466	3070	3945	1346/6126	2814/4780	9291	287518	278251	1422	3
	8	38.0	495839	7091	219835	255836	493	10782	301	1813	472	3494	4380	1994/6488	2813/4494	11050	337203	326141	1430	3
30J-40R-8M-SIMPLE	5	29.0	531386	30597	220506	203896	2583	69857	624	1441	70	9505	10105	1936/15601	8314/13665	26576	2449472	2422912	2483	1
	6	32.5	422670	12028	156500	226822	1056	23691	440	1279	120	5619	6411	1183/9425	5388/8242	15894	844399	828494	1466	1
	7	34.5	402219	7858	128842	249609	665	13469	372	1244	902	4587	3902	1119/6750	2980/5631	14102	452215	438131	2102	4
40J-30R-2M-SIMPLE	20	32.5	233108	2124	84069	144546	148	1571	34	1328	270	838	2480	1812/2567	747/756	3716	83849	80135	0	1
	21	32.5	105336	1383	37619	64884	87	1029	22	586	334	704	1345	1080/1440	356/360	2191	57822	55645	0	1
	22	33.5	3793	36	2318	1425	0	0	0	5	293	7	64	52/66	14/14	91	107	2	0	1
40J-30R-5M-SIMPLE	8	29.5	108347	6492	53402	39601	496	7345	179	1043	138	2826	2383	1629/3216	1232/1588	8731	464860	456144	48	1
	9	30.0	39449	3240	17935	15378	190	2247	150	274	242	2246	784	631/1016	368/385	5523	155992	150475	4	1
	10	30.5	24315	2418	14526	5324	139	1549	135	231	207	1888	488	504/665	161/161	5117	99299	94180	0	1
40J-30R-8M-SIMPLE	5	24.0	513492	72362	202250	152927	6307	75450	917	1099	79	11476	7285	2047/14694	5733/12647	32338	6520303	6487992	1638	1
	6	27.0	204955	15597	86705	81610	1335	17781	515	1065	119	6567	3264	664/5834	2686/5170	16991	1219931	1202933	664	1
	7	29.0	99334	7454	40974	42882	562	6522	272	361	139	3897	1704	435/2553	1372/2118	11834	493941	482108	108	1
40J-50R-2M-SIMPLE	20	56.5	3393036	11490	1477135	1894323	135	6620	91	7587	340	1708	12610	9048/12925	3854/3890	15905	69736	53835	8	1
	21	56.5	1763896	8124	1037373	714374	76	1695	49	4827	373	1226	8135	6487/8316	1823/1838	9850	35369	25519	1	1
	22	56.5	2055868	7612	1180086	862873	98	2979	68	4726	458	1227	7958	6346/8305	1929/1969	9700	40796	31095	3	1
40J-50R-5M-SIMPLE	8	41.0	6758597	387341	2039657	3740732	17308	551051	1604	16915	114	10903	59850	16072/67789	44952/51719	86893	15116574	15029697	129	1
	9	45.5	3109407	39856	1002495	1958008	3009	97702	1057	9393	238	6097	21568	5063/24915	17062/19852	32138	2191044	2158913	77	1
	10	47.5	1656397	15831	727677	875529	1354	32122	658	4152	715	4142	9439	3352/11149	6629/7797	16765	836800	820028	107	1
40J-50R-8M-SIMPLE	8	41.5	9005981	557294	2161373	5179701	29379	1047598	2872	18312	204	17151	74790	15148/84927	59963/69779	111905	24026108	23914219	142	1
	9	45.5	4071406	85672	1081157	2668866	8396	214900	2303	13334	512	9816	26315	5520/30885	21134/25365	45423	5839609	5794185	85	1
	10	47.5	2390489	29703	820562	1442491	4065	86431	1717	5695	564	7487	12708	3583/15603	9641/12020	26626	2282733	2256097	49	1

Table A.2: Statistics for simplified ($p_j = 1, r_j = 0, \bar{d}_j = n$) problem instances.

Appendix B

Interpreted results

See sections 9.3.3 and 9.3.4 for the description of these tables.

Problem	Total time		C	Value of fixated relations	
	Optimal	Heuristic		Optimal	Heuristic
20J-10R-2M-SIMPLE	0m3.332s	0m9.945s	10	10.0	10.0
			11	11.0	11.0
20J-10R-5M-SIMPLE	0m0.975s	1m1.329s	4	6.5	6.5
			5	8.5	8.5
			6	9.5	9.5
20J-10R-8M-SIMPLE	0m1.411s	1m29.583s	4	6.5	6.5
			5	8.5	8.5
			6	9.5	9.5
20J-30R-2M-SIMPLE	0m3.059s	5m44.777s	10	32.0	32.0
			11	32.5	32.5
			12	33.0	33.0
20J-30R-5M-SIMPLE	0m44.994s	13m30.345s	6	23.5	23.5
			7	26.0	26.0
			8	28.5	28.5
20J-30R-8M-SIMPLE	0m36.648s	16m31.281s	6	23.5	23.5
			7	26.0	26.0
			8	28.5	28.5

Table B.1: Interpreted results for simplified ($p_j = 1, r_j = 0, \bar{d}_j = n$) instances with 20 jobs.

Problem	Total time		C	Value of fixated relations	
	Optimal	Heuristic		Optimal	Heuristic
30J-20R-2M-SIMPLE	0m0.746s	0m5.400s	15	21.5	21.5
30J-20R-5M-SIMPLE	0m2.443s	1m36.935s	6	18.5	18.5
			7	19.5	19.5
			8	19.5	19.5
30J-20R-8M-SIMPLE	0m4.869s	3m20.735s	4	17.0	17.0
			5	18.0	18.0
			6	18.5	18.5
30J-40R-2M-SIMPLE	11m58.309s	57m7.741s	15	48.5	48.5
			16	49.5	49.5
			17	50.0	50.0
30J-40R-5M-SIMPLE	19m50.820s	34m11.604s	6	32.5	32.5
			7	35.0	34.5
			8	38.0	38.0
30J-40R-8M-SIMPLE	129m23.710s	25m29.471s	5	29.0	29.0
			6	32.5	32.5
			7	35.0	34.5

Table B.2: Interpreted results for simplified ($p_j = 1, r_j = 0, \bar{d}_j = n$) instances with 30 jobs.

Problem	Total time		C	Value of fixated relations	
	Optimal	Heuristic		Optimal	Heuristic
40J-30R-2M-SIMPLE	0m3.409s	5m55.130s	20	32.5	32.5
			21	32.5	32.5
			22	33.5	33.5
40J-30R-5M-SIMPLE	1m16.801s	4m1.367s	8	29.5	29.5
			9	30.0	30.0
			10	30.5	30.5
40J-30R-8M-SIMPLE	0m45.258s	17m16.292s	5	24.0	24.0
			6	27.0	27.0
			7	29.0	29.0
40J-50R-2M-SIMPLE	683m45.842s	120m48.166s	20	56.5	56.5
			21	56.5	56.5
			22	56.5	56.5
40J-50R-5M-SIMPLE	30m44.282s	195m48.664s	8	41.5	41.0
			9	45.5	45.5
			10	47.5	47.5
40J-50R-8M-SIMPLE	378m57.224s	263m51.832s	8	42.0	41.5
			9	45.5	45.5
			10	47.5	47.5

Table B.3: Interpreted results for simplified ($p_j = 1, r_j = 0, \bar{d}_j = n$) instances with 40 jobs.

Appendix C

Improved genetic algorithm

See section 9.3.5 for the description of these tables.

Problem	Total time				C	Number of fixated relations			
	Optimal	Naive	Naive+CS	Improved		Optimal	Naive	Naive+CS	Improved
10J-10R-2M	0m0.465s	0m3.909s	0m1.942s	0m8.556s	53	6	6	6	6
					54	7	7	7	7
					55	7	7	7	7
					56	7	7	7	7
10J-10R-2M-SIMPLE	0m0.446s	0m0.845s	0m1.049s	0m1.094s	5	7	7	7	7
					6	8	8	8	8
					7	8	8	8	8
					8	8	8	8	8
10J-10R-4M	0m0.687s	0m1.186s	0m2.701s	0m8.596s	27	2	2	2	2
					28	4	4	4	4
					29	4	4	4	4
					30	5	5	5	5
10J-10R-8M	0m0.554s	0m0.647s	0m1.587s	0m0.881s	19	2	2	2	2
					20	2	2	2	2
					21	2	2	2	2
					22	2	2	2	2
10J-25R-2M	0m0.540s	0m0.941s	0m1.592s	0m2.793s	42	15	13	12	15
					43	15	13	13	15
					44	15	13	13	15
					45	15	13	13	15
10J-25R-2M-SIMPLE	0m0.548s	0m1.073s	0m1.567s	0m1.638s	5	14	13	13	14
					6	16	13	13	16
					7	17	15	15	17
					8	19	16	17	19
10J-25R-4M	0m1.182s	0m0.900s	0m2.165s	0m10.267s	26	7	7	7	7
					27	8	7	7	8
					28	9	7	7	9
					29	9	7	7	9
10J-25R-8M	0m1.539s	0m0.996s	0m2.144s	0m1.553s	17	5	5	5	5
					18	7	5	7	7
					19	7	7	7	7
					20	7	7	7	7

Table C.1: Results of the improved genetic algorithm on the original instances with 10 jobs.

Problem	Total time				C	Number of fixated relations			
	Optimal	Naive	Naive+CS	Improved		Optimal	Naive	Naive+CS	Improved
25J-10R-2M	0m0.466s	0m0.660s	0m0.786s	0m0.540s	141	10	10	10	10
25J-10R-2M-SIMPLE	0m0.463s	0m0.734s	0m0.682s	0m0.483s	13	10	10	10	10
25J-10R-4M	0m0.487s	0m0.781s	0m0.831s	0m0.514s	71	10	10	10	10
25J-10R-8M	0m0.954s	0m1.894s	0m2.012s	0m15.810s	36	8	8	8	8
					37	9	9	9	9
					38	9	9	9	9
					39	9	9	9	9
25J-25R-2M	0m0.560s	0m5.794s	0m8.956s	0m9.817s	153	21	21	21	21
					156	22	22	22	22
					159	22	22	22	22
					162	22	22	22	22
25J-25R-2M-SIMPLE	0m0.572s	0m4.404s	0m3.076s	0m2.048s	13	23	23	23	23
					14	23	23	23	23
					15	23	23	23	23
					16	23	23	23	23
25J-25R-4M	0m1.248s	0m13.160s	0m18.425s	0m9.976s	77	17	17	17	17
					78	18	18	18	18
					79	18	18	18	18
					80	18	18	18	18
25J-25R-8M	0m17.241s	0m7.674s	0m16.423	0m59.069s	39	10	10	10	10
					40	11	10	10	11
					41	12	12	12	12
					42	12	12	12	12
25J-60R-2M	0m0.756s	0m1.850s	0m2.752s	0m7.515s	141	44	31	36	44
					143	45	31	36	45
					145	45	36	38	45
					147	45	36	38	45
25J-60R-2M-SIMPLE	0m0.721s	0m9.517s	0m4.085s	0m2.790s	13	46	45	43	46
					14	46	45	45	46
					15	46	45	45	46
					16	47	45	45	47
25J-60R-4M	0m7.397s	0m4.266s	0m6.807s	0m22.700s	71	33	26	26	33
					72	33	27	27	33
					73	34	27	27	34
					74	34	27	27	34
25J-60R-8M	47m43.342s	0m9.871	0m18.385s	1m50.012s	36	18	16	15	18
					37	20	17	16	20
					38	21	17	17	21
					39	22	18	18	22

Table C.2: Results of the improved genetic algorithm on the original instances with 25 jobs.

Problem	Total time				C	Number of fixated relations			
	Optimal	Naive	Naive+CS	Improved		Optimal	Naive	Naive+CS	Improved
60J-25R-2M	0m0.565s	0m46.418s	0m5.574s	0m0.842s	333	25	25	25	25
60J-25R-2M-SIMPLE	0m0.556s	0m24.541s	0m6.829s	0m0.685s	30	25	25	25	25
60J-25R-4M	0m0.598s	0m8.067s	0m2.728s	0m0.890s	167	25	25	25	25
60J-25R-8M	0m1.482s	0m22.465s	1m31.075s	0m3.793s	84	23	23	23	23
					85	23	23	23	23
					86	23	23	23	23
					87	24	24	24	24
60J-60R-2M	0m0.840s	20m43.988s	1m49.297s	0m18.278s	317	56	55	55	56
					323	56	55	55	56
					329	56	55	55	56
					335	56	55	55	56
60J-60R-2M-SIMPLE	0m0.832s	12m57.844s	2m2.859s	0m9.954s	30	56	55	55	56
					31	56	55	55	56
					32	56	55	56	56
					33	56	55	56	56
60J-60R-4M	0m2.729s	6m43.308s	2m44.254s	0m13.206s	159	51	49	51	51
					162	52	49	52	52
					165	52	50	52	52
					168	52	50	52	52
60J-60R-8M	22m35.319s	2m40.046s	3m1.706s	0m17.393s	80	44	40	40	44
					81	44	42	42	44
					82	44	42	43	44
					83	44	42	43	44
60J-150R-2M	0m9.646s	0m10.323s	0m9.542s	0m13.051s	313	116	89	103	116
					319	117	89	103	117
					325	117	89	103	117
					331	117	89	105	117
60J-150R-2M-SIMPLE	0m6.748s	2m21.628s	0m21.888s	0m7.058s	30	116	87	104	116
					31	117	87	105	117
					32	117	87	105	117
					33	118	87	105	118
60J-150R-4M	1000+m	7m18.352s	4m11.121s	0m50.246s	157	*n/a*	80	82	94
					160	95	80	82	95
					163	*n/a*	80	87	95
					166		80	87	96
60J-150R-8M	200+m	5m57.544s	6m28.629s	3m57.197s	79	*n/a*	58	56	73
					80		58	59	76
					81		58	59	76
					82		58	62	77

Table C.3: Results of the improved genetic algorithm on the original instances with 60 jobs.