# Robust Solutions for the Resource-Constrained Project Scheduling Problem

*Understanding and Improving Robustness in Partial Order*

*Schedules produced by the Chaining algorithm*

Daan Wilmer

# Robust Solutions for the Resource-Constrained Project Scheduling Problem

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Daan Wilmer
born in Vlaardingen, the Netherlands

TU Delft

Algorithmics Group
Software and Computer Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`alg.ewi.tudelft.nl`

# Robust Solutions for the Resource-Constrained Project Scheduling Problem

| | |
|---|---|
| Author: | Daan Wilmer |
| Student id: | 1358510 |
| Email: | daan@daanwilmer.nl |

**Abstract**

Robustness is essential for schedules if they are being executed under uncertain conditions. In this thesis we research robustness in Partial Order Schedules, which represent sets of solutions for instances of the Resource-Constrained Project Scheduling Problem. They can be generated using a greedy procedure called Chaining, which can be easily adapted to use various heuristics. We use an empirical method to gain understanding of robustness and how the chaining procedure adds to this.

Based on the findings of an exploratory study we develop three models, each capturing aspects of robustness on a different level. The first model describes how a single activity is affected by various disturbances. The second model predicts how structural properties of Partial Order Schedules can reduce the effect of these disturbances. The third model describes how heuristics for the chaining procedure can influence these properties.

Using experimental evaluation, we found that the model is not complete. Experimental results did conform to the expectations set by the third model, but not of the second model. We therefore suspect that it is too simplistic for accurate predictions, but since it does match earlier observations we believe it is a good starting point for further understanding.

# Contents

# List of Figures

# Chapter 1

# Introduction

Logistics has been one of the underpinnings of any successful organization, and scheduling is one of the core practices of logistics. Its importance is especially clear when multiple activities or processes compete for the same resource. Take for example a railway network, where only one train at a time can use a particular section of railway track. Some trains can go faster than others, or stop at more or fewer stations. To make the network operate smoothly, a careful schedule has to be made to meet all demands.

Another area where logistics, and especially scheduling, play a very important role, is manufacturing and maintenance. On the one hand, products are becoming more and more complex. Cars, for example, continue to be developed with more luxury and safety features every year. Where the earliest cars only focused on being able to drive without feeling too much of the bumps in the road, most of today's new cars have several sensor-triggered airbags, climate control and satellite navigation included. On the other hand there is a drive to improve efficiency. Efficient scheduling can reduce the time or resource requirements, allowing manufacturing and maintenance to become cheaper or easier to scale up.

These developments increase the complexity of the schedules and add more constraints on the result. This makes it harder to produce an acceptable schedule, which is why it is an interesting research topic for algorithmics. One particular problem model we find interesting is the Resource-Constrained Project Scheduling Problem, or RCPSP. This problem focuses on projects which consists of activities, which require time and resources and might be ordered. An example of this problem is car maintenance: activities like replacing a tyre require resources in the form of people and tools, and you can only replace a tyre once you have taken the wheel from the car. As cars become increasingly complex, and garages often handle multiple cars at the same time, efficient schedules become harder to create. Similar developments happen in other areas: products become more complex while manufacturing and maintenance facilities tend to scale up for increased potential efficiency. The ubiquity of this problem, as well as the opportunity to reduce waste through improved efficiency is what makes this problem so interesting to us.

Another issue, besides complexity, is the probability of disturbances. It happens all too often that projects are over time, over budget, or both. This is caused by activities taking more time than expected, causing dependent activities to be delayed, and so on. This calls for robust scheduling: scheduling in such a way that schedules can be adapted to these disturbances

One method for robust scheduling is to provide a set of schedules as solution for a scheduling problem, as opposed to only a single schedule. This way, the best schedule for certain conditions can be picked from a set, which can be done much faster than generating an entirely new schedule. This set of solutions could be represented as a partial order schedule, which is in fact a scheduling problem of its own. The advantage of the partial order schedule is that an actual schedule can be generated in little time. However, it is infeasible to create a partial order schedule that represents the complete set of solutions to a scheduling problem. Instead we will have to create a partial order schedule such that the set of solutions is as good as possible within reasonable time.

Several methods have been proposed for creating a Partial Order Schedule, one of which is called chaining. It has proven to be a quite simple and flexible, yet effective method for creating Partial Order Schedules. However, its effectiveness has so far only been measured using metrics that are thought to correlate with robustness, but this correlation has not been proven.

Our goal in this thesis is therefore to perform chaining in such a way that the resulting Partial Order Schedule is as robust as possible, by understanding how different methods in chaining affect the robustness of the resulting POS. In order to do so we propose the following research questions:

1. What is robustness?
2. To what extent are Partial Order Schedules robust?
3. What is the relationship between the method used for chaining and the robustness of the resulting Partial Order Schedule?
4. How can methods for creating Partial Order Schedules be adapted to create more robust Partial Order Schedules?

The rest of this thesis is structured as follows. In Chapter 2 we will define the necessary concepts and definitions, including RCPSP, Robustness, and Partial Order Schedules. In Chapter 3 we will then try to find answers to our research questions, as well as related topics, in existing literature. In Chapter 4 we will summarize the remaining questions, and the method for answering these.

In Chapter 5 we will perform an exploratory study to gather observations regarding our research questions. We will then propose three models in Chapter 6 to explain our observations. These models will then be evaluated using experiments in Chapter 7. Finally we will summarize our conclusions and future work in Chapter 8.

# Chapter 2

# Concepts and Definitions

First of all, we introduce the reference framework of this thesis. In this chapter we will state and define the concepts we will use in the rest of our thesis. Some of these concepts are taken from existing work, others are developed anew. When needed, definitions are adapted or refined to fit consistently in this work.

We will first introduce the concepts and definitions of scheduling, including our reference problem of RCPSP. Then we will define robustness, and how this definition maps to our definitions of schedules. This will provide the background information needed for our research.

## 2.1 Scheduling

The research area of our thesis is scheduling, meaning the assignment of times to a set of events. Such an event could be the start of a process or a deadline for delivery of a product.

To narrow down the field, we research the subtopic of project scheduling. In this subtopic, the events that are scheduled are start times of activities. These activities have a certain duration, and may depend on other activities.

Our reference problem in this subtopic is RCPSP, or the Resource-Constrained Project Scheduling problem. In this section we will first define RCPSP, after which we will define the Partial Order Schedule, or POS, which is a form to represent solutions to scheduling problems. Finally we will present general strategies for creating a POS.

### 2.1.1 Resource-Constrained Project Scheduling Problem

The reference problem in our thesis is the Resource-Constrained Project Scheduling Problem, or RCPSP for short. An RCPSP instance consists of a project, which is a set of activities with precedence constraints, and resources. There can be any finite number of resources, each with a finite capacity. The activities have a certain duration, and during that duration they require a certain amount of some of the resources which are released afterwards. Each precedence constraint dictates an ordering between one pair of activities, meaning that one cannot start before the other activity has finished. The goal is to create a schedule that satisfies all constraints and minimizes the makespan.

For a formal definition, we first define an RCPSP instance in Definition 1:

**Definition 1.** An RCPSP instance is a tuple $\langle A, P, R, c, d, q \rangle$, where $A$ is a set of activities, $P \subseteq A \times A$ is the set of precedence constraints, and $R$ is the set of resources. For a resource $r \in R$, let $c_a$ be the capacity of that resource. For an activity $a \in A$, let $d_a \in \mathbb{N}_{>0}$ be the duration of that activity and let $q_{a,r}$ be the resource requirement of that activity for resource $r \in R$.

In order to define what a valid schedule is in Definition 4, we first define an assignment of start times to activities in Definition 2 and the makespan of such an assignment in Definition 3.

**Definition 2.** An assignment of start times to activities is a $s : A \to \mathbb{R}$.

**Definition 3.** The makespan of an assignment of start times to activities $s$ for a set of activities $A$ is $mks(s) = \max_{a \in A}(s(a) + d_a)$.

**Definition 4.** Given an assignment of start times to activities $s$ for an RCPSP instance $I = \langle A, P, R, c, d, q \rangle$. Let $\delta(t) = \{a \in A | s(a) \leq t < s(a) + d_a\}$ be the set of activities that are active at time $t$.

The assignment of start times to activities $s$ is a valid schedule if and only if the following two equations hold.

$$\forall t \in [0, mks(s)] : \forall r \in R : \sum_{a \in \delta(t)} q_{a,r} \leq c_r \tag{2.1}$$

$$\forall (a_i, a_j) \in P : s(a_i) + d_{a_i} \leq s(a_j) \tag{2.2}$$

Given these definitions, we can state the goal of RCPSP very concisely in Definition 5.

**Definition 5.** The goal of the Resource-Constrained Project Scheduling Problem is to create a valid schedule for an RCPSP instance such that the makespan is minimal.

For reasoning on RCPSP instances, we define the set of successors and the total set of successors of an activity as follows:

**Definition 6.** Given a set of activities $A$ and a set of precedence constraints $P$, the set of successors of an activity $a \in A$ is equal to

$$succ(a) = \{x \in A | (a, x) \in P\} \tag{2.3}$$

**Definition 7.** Given a set of activities $A$ and a set of precedence constraints $P$, the total set of successors of an activity $a \in A$ is equal to

$$succ^*(a) = succ(a) \cup \bigcup_{x \in succ(a)} succ^*(x) \tag{2.4}$$

And similarly, for the predecessors of an activity:

**Definition 8.** Given a set of activities $A$ and a set of precedence constraints $P$, the set of predecessors of an activity $a \in A$ is equal to

$$pred(a) = \{x \in A | (x,a) \in P\} \tag{2.5}$$

**Definition 9.** Given a set of activities $A$ and a set of precedence constraints $P$, the total set of predecessors of an activity $a \in A$ is equal to

$$pred^*(a) = pred(a) \cup \bigcup_{x \in pred(a)} pred^*(x) \tag{2.6}$$

As defined in Definitions 2 and 4, a schedule is a single assignment of start times to activities — there is only one time assigned to each activity. For this reason, this type of schedule is also called a fixed-time schedule.

In contrast, there are also situations where it is desired that multiple start times can be assigned to activities, but still in a schedule-like manner. For example, an interval schedule assigns not a single time but an interval to an activity. These intervals are chosen such that for each activity any time point in its interval can be chosen, but the resulting fixed-time schedule is a valid schedule. This is desired to give work crews some form of autonomy in performing an activity, while retaining control over the entire process, as proposed in Wilson et al. (2013).

### 2.1.2 Partial Order Schedule

A Partial Order Schedule is another type of schedule which does not provide fixed start times for activities. Instead, a POS defines start times of activities relative to eachother. More specifically: it uses precedence constraints to only specify the ordering of some pairs of activities. This leaves flexibility to derive multiple different fixed-time schedules from the Partial Order Schedule, or POS.

For this reason, POSes are used as solutions for RCPSP instances. In order to be usable as a solution for a certain RCPSP instance, a POS must be constructed such that any fixed-time schedule which is valid for the POS must also be a valid schedule for the RCPSP instance. This allows for more flexibility than a fixed-time schedule.

We can define also define the structure and use of a POS formally, for more formal reasoning. From Policella et al. (2009) we derive Definition 10.

**Definition 10.** Given an RCPSP instance $I = \langle A, P, R, c, d, q \rangle$, a Partial Order Schedule for this instance consists of the tuple $\langle A, P \cup U, d \rangle$, where $U$ is a set of additional precedence constraints chosen such that every assignment of start times to activities $s$ which satisfies the precedence constraints is also a valid fixed-time schedule for the original instance $I$.

$$\forall (a_i, a_j) \in P \cup U : s(a_i) + d_{a_i} \leq s(a_j) \tag{2.7}$$

### Earliest Start Schedule

In order to use the POS as solution for an RCPSP instance, we need to be able to derive a fixed-time schedule from it. A special schedule is the earliest start schedule, which is the schedule such that every activity starts at its earliest start time, as defined in Policella et al. (2009). We define it as follows:

**Definition 11.** Given a Partial Order Schedule $\langle A, P \cup U, d \rangle$, its earliest start schedule $\hat{s}$ is an assignment of start times to activities such that every activity is started as early as possible without violating any precedence constraints.

$$\forall a \in A : \hat{s}(a) = \max(0, \max_{x \in pred(a)} (\hat{s}(x) + d_x)) \tag{2.8}$$

This earliest start schedule can be constructed in linear time, by traversing all activities in topological order: an ordering such that an activity occurs in this ordering after all its predecessors and before all its successors. A simple algorithm to create a topological order can be found by treating the activities and precedence constraints as vertices and edges, respectively, of a directed graph. Without loss of generality we can assume that this graph contains no cycles, forming a directed acyclic graph, for which a linear algorithm for creating a topological ordering has been demonstrated in Kahn (1962). Using this topological ordering, we can follow the Definition 11 to determine the start time of each activity based on its predecessors in a total time of $O(|A| + |P \cup U|)$, which is linear.

We can assume that this graph contains no cycles, because otherwise the finishing time of one activity in that cycle would have to be less than or equal to the start time of that same activity. This is only possible if all activities in that cycle have duration 0, which is conflicting with Definition 1. If there is a cycle with activities which have a larger duration, no fixed-time schedules are consistent with the POS.

### Constructing a POS

Following Definition 10, a Partial Order Schedule is constructed by adding additional precedence constraints $U$ to the set of precedence constraints $P$ of the instance, such that these additional precedence constraints also prevent resource conflicts — situations where the resources required by the active activities exceed the available resource capacity. An easy solution is to construct a total ordering, allowing no activities to run concurrently. Constructing other POSes with more desirable qualities, e.g. a shorter makespan, is non-trivial.

Several methods have been proposed for creating a Partial Order Schedule of good quality. According to Policella et al. (2007), these can be categorized in two general strategies: an "outside-in" approach, and an "inside-out" approach. The outside-in approach is the oldest, and works by adding (or "posting") precedence constraints to the set of additional constraints, reducing the set of possible fixed-time schedules until all of them are valid for the RCPSP instance. This method is called precedence constraint posting, or PCP.

The inside-out method is a newer method, which works in two steps. In the first step, a fixed-time schedule is created as solution for the RCPSP instance. Next, this fixed-times schedule is used to create a partial order schedule. This method is called Solve-and-Robustify.

**Precedence Constraint Posting** Precedence Constraint Posting is an iterative method of creating a Partial Order Schedule for an RCPSP instance. It works by adding detecting resource conflicts and adding new constraints to reduce these conflicts, until no more resource conflicts are found.

One method to do this, proposed in Cesta, Oddi and Smith (1998), uses resource profiles to detect resource conflicts and then resolves them using a heuristic approach. A resource profile is a description of the resource usage over time. Since the start times for activities in a POS are not fixed, Cesta et al. use an upper bound resource profile and a lower bound resource profile. The lower bound resource profile counts activities only at time points when they must be scheduled because of resource constraints, while the upper bound resource profile counts activities at all time points where they could be scheduled.

The upper bound resource profile is used to detect possible resource conflicts, while the lower bound resource profile is used to prioritize conflicts. When it is decided which conflict to solve, it is reduced or resolved by adding a precedence constraint chosen to maximize slack, or the temporal distance between activities.

Another method focuses on critical sets: sets of activities that, when looking at the precedence constraints only, can be scheduled concurrently, but would violate the resource constraints when they were scheduled concurrently. These sets can be detected by solving a minimum-flow problem, as proposed in Lombardi and Milano (2012). To solve a resource conflict, a minimal critical set is created from the critical set: a critical set that, when one of the activities is removed, is no longer critical. It is therefore sufficient to post one precedence constraint between two activities in this MCS to eliminate it.

**Solve-and-Robustify** Solve-and-Robustify is a two step method for creating a Partial Order Schedule. The first step is to solve the RCPSP instance to create a fixed-time schedule. Next, this fixed-time schedule is “robustified”, or transformed into a partial order schedule.

In the “solve” step, a fixed-time schedule is created. Since this schedule is later “robustified” into a POS, this schedule is one of the schedules that can later be derived from the POS. When this method was introduced in Policella et al. (2007), the ESTA algorithm was used to create the fixed-time schedule. However, since there is no coupling between the algorithms for the “solve” and “robustify” steps other than the fixed-time schedule, the ESTA algorithm can be replaced by any algorithm that can solve RCPSP instances.

In the “robustify” step, the created fixed-time schedule is transformed into a Partial Order Schedule. Again, there are multiple approaches for doing this, for example using IP-based approaches similar to Braeckmans et al. (2005). We do, however, want to focus on the greedy algorithm that was introduced in Policella et al. (2007), called chaining.

In chaining, the focuses on the resources, rather than the activities. This method is based on the idea that, during execution of a schedule, each resource unit is used by a series, or chain, of activities. For example, a single screwdriver might be used in different activities of maintenance. Through the subsequent use of this resource unit, the activities are automatically sequenced and dependent on each other.

This idea is made explicit in the algorithm. For each resource unit a chain is created. Then, activities are, in chronological order, assigned to one chain for each unit of each resource they require, such that two activities which are scheduled concurrently cannot be assigned to the same chain. The chains are then fixed by adding precedence constraints between each pair of consecutive activities in this chain. This way, activities which need the same resources cannot be scheduled concurrently and therefore do not compete with each other.

A partial order schedule constructed using this method always contains the fixed-time schedule used as input for chaining. However, which other schedules can be derived from the POS depends on its structure. This structure is determined greatly by the choice of which activities to assign to which chain.

When the method was first introduced in Policella et al. (2007), the assignment of activities to chains was done randomly. Later, two heuristics were introduced: maxCC and minID, in Policella et al. (2009). These heuristics are both aimed at reducing the number of added precedence constraints.

The maxCC heuristic, which stands for "Maximize Common Chains", tries to maximize common chains between pairs of activities. The idea is that activities often require more than one unit of a resource and therefore have to be assigned to several chains. Once an activity is added to one chain, this heuristic tries to find other chains which have the same activity as latest assigned activity.

The minID heuristic, which stands for "Minimize InterDependencies", tries to reuse existing precedence constraints. That is: it tries to assign the activity to a chain where the previous activity already is a predecessor of the new activity, requiring no new precedence constraints.

These two heuristics are combined to form a policy for assigning new activities to chains, called maxCCminID. It first tries to use the maxCC heuristic, if that does not work it tries the minID heuristic, and otherwise a random activity is chosen

## 2.2 Robustness

As explained, our interest with respect to scheduling is robustness. Therefore, we use this section to explain robustness. We will start by defining the general concept of robustness, after which we will apply this definition to scheduling. Using these definitions, we will answer the first two research questions: "What is robustness?", and "To what extent are Partial Order Schedules robust?"

### 2.2.1 The General Concept of Robustness

To define robustness, we turn to the Oxford English Dictionary. It defines *robust*, when referring to an object, as "sturdy in construction" or "able to withstand or overcome adverse conditions". The same dictionary states that *to withstand* means "to remain undamaged or unaffected by".

With slightly different wordings, we therefore define *robustness* as follows:

**Definition 12.** Robustness of an object is the ability of that object to retain or regain its properties during or following adverse conditions.

### 2.2.2 Robustness of Schedules

This definition has two abstract terms in it: “its properties” and “adverse conditions”. In order to apply this definition to schedules, we have to specify the properties of a schedule, and the adverse conditions that can be encountered during execution. We wil start with the latter part.

#### Adverse Conditions

During execution of a schedule, conditions might be different than initially expected. For example, activities may take longer due to unforeseen circumstances, or resources become (temporarily) unavailable due to damage or theft. We will model this as a disturbance in the instance:

**Definition 13.** A disturbance of an RCPSP instance $I = \langle A, P, R, c, d, q \rangle$ is a mapping $f$ from that instance to another instance $f(I) = \langle A, P, R, c', d', q' \rangle$, which is called the disturbed instance.

Note that disturbances can only affect the resource capacity, activity duration, or resource requirement; we assume that the structure of the instance remains unchanged. Also note that the nature of these disturbances is not specified, and that they might not be adverse. However, in this thesis we will focus only on adverse disturbances, i.e. disturbances $f$ such that there exists an assignment $s$ of start times to activities which is valid for $I$ but invalid for $f(I)$.

#### Desirable Properties of Schedules

The second part to adapt to scheduling is “properties”. Of course, a schedule can have many properties, but we will focus on desirable properties. To know which desirable properties a schedule can have, we look at the use of these schedules. The first use of schedules is as solution for an RCPSP instance. Since the goal for RCPSP is to create a valid schedule with minimum makespan, this provides the first two properties of a schedule: validity, and makespan.

Validity is an essential property of schedules: if an assignment of start times to activities is not a valid schedule, there will be problems during execution. Either the resource demand exceeds the capacity or precedence constraints are violated, or both. The main reason to create a schedule is to avoid these problems, so ending up with an invalid assignment of start times to activities — regardless of other properties — defeats the purpose of scheduling altogether.

The second desirable property of a schedule is a short makespan. RCPSP is an optimization problem, of which a short makespan is the objective. This does not only apply to finding fixed-time solutions, but also to the schedules created for the disturbed instance: the closer to optimal the makespan is, the better.

Looking further, we find these schedules being used to schedule projects. From this context, we can derive a third desirable property: stability, or the lack of change from an original schedule. Having to reschedule fewer activities is desirable in an environment where rescheduling is costly, for example when activities include external parties.

We define stability to be the fraction of activities which do not have to be rescheduled because of a disturbance, as described in Definition 14.

**Definition 14.** Given a baseline schedule $s_0$ and a schedule under disturbance $s_d$ for a set of activities $A$, stability is equal to

$$\sigma(s_0, s_d) = \frac{|\{a \in A \mid s_d(a) = s_0(a)\}|}{|A|} \tag{2.9}$$

Note that we chose in this definition to keep it purely activity-based instead of also including a temporal aspect, i.e. the amount of change. This is motivated because the temporal aspect is already included in the makespan, and partly because we want to emphasize the use case of having external contractors where the *act* of rescheduling costs money, instead of the *amount* an activity has been moved.

### 2.2.3 Robustness of a Partial Order Schedule

With both robustness and partial order schedules defined, we can now combine the definitions to provide an answer to Research Question 2: "To what extent are Partial Order Schedules robust?"

To answer this question, we have to understand how adverse conditions affect the schedules that can be obtained from the partial order schedule. We do this by examining the three types of disturbance, as defined in Definition 13: disturbances in resource capacity, disturbances in resource requirements, and disturbances in activity duration.

#### Disturbance in Resource Capacity

When the capacity of a resource is reduced, it cannot be guaranteed that the set of solutions represented by the POS still contains a valid solution. This is because the POS does not contain information on the resources; only the solutions to possible resource conflicts, or which activities to order sequentially and in what order, are retained. Any information about the exact resource usage is not present anymore, so it is impossible to know from the POS alone whether a solution is still valid with this modification. This can be checked using the original RCPSP instance, but ensuring a valid solution — constructing a new, valid solution when another solution is invalid — requires a new scheduling effort, effectively creating a new Partial Order Schedule.

When the capacity of a resource is increased, the set of solutions represented by the POS is still valid. However, there is no way of taking advantage of the extra capacity, for the same reason as the previous case: there is no information regarding resources.

In either case, partial order schedules cannot be considered robust under disturbances in resource capacity.

### Disturbances in Resource Requirements

When the resource requirements are disturbed, the same problem arises as with the disturbances in resource capacity: there is no concrete information on the resource capacities and requirements in the POS.

When resource requirements are increased, there is no way of determining whether or not there is enough capacity to handle this increase. This means that there is no guarantee that the fixed-time schedules which can be derived from the POS are still valid. Similarly, when the resource requirements are reduced, none of the schedules represented by the POS become invalid. However, there is no way to take advantage of this reduced requirement.

We therefore do not consider partial order schedules robust under disturbances in resource capacity.

### Disturbances in Activity Duration

When activity durations are disturbed, the Partial Order Schedule can be used to easily generate a fixed-time schedule for the disturbed instance. Suppose we have the POS $\langle A, P \cup U, d\rangle$ for the instance $I = \langle A, P, R, c, d, q\rangle$, and let $f(I) = \langle A, P, R, c, d', q\rangle$ be the disturbed schedule, with only changed activity durations. We can then change the POS to $\langle A, P \cup U, d'\rangle$ in order to have a Partial Order Schedule describing a set of solutions for $f(I)$. These solutions can again be generated in linear time using the algorithm as described in Section 2.1.2.

Although this is only a small adaptation, it is still sufficient to created valid schedules. The set $P$ of precedence constraints is unchanged by the disturbance, so this can also remain the same in the POS. The resource capacity and resource requirement also remain unaffected, so any combination of concurrently scheduled activities that is acceptable for $I$ is also acceptable for $f(I)$. Since the possible combinations depend only on the precedence constraints in the POS, and not on activity durations, we do not have to change the set of added precedence constraints $U$ either. The only thing that we need to change is therefore the set of activity durations, to ensure that schedules reserve enough time for activities with extended durations.

Using this, we know that validity of a Partial Order Schedule is guaranteed under temporal disturbances. The other two desirable properties, short makespan and stability, are not directly applicable to Partial Order Schedules. Makespan is calculated using a fixed-time schedule, and stability depends on a fixed-time schedule as well as a baseline schedule. Therefore, we use the earliest start schedules to define the makespan of a POS in Definition 15 and the stability of a disturbed POS in Definition 16.

**Definition 15.** The Makespan of a Partial Order Schedule is equal to the makespan of its earliest start schedule.

**Definition 16.** Let $\langle A, P \cup U, d \rangle$ be a Partial Order Schedule and let $\hat{s}_0$ be its earliest start schedule. Let $\langle A, P \cup U, d' \rangle$ be the same POS but adapted for disturbances and let $\hat{s}_d$ be its earliest start schedule. The stability of this disturbed POS is equal to the stability of its earliest start schedule compared to the earliest start schedule of the non-disturbed POS:

$$\sigma(\hat{s}_0, \hat{s}_d) = \frac{|\{a \in A \mid \hat{s}_d(a) = \hat{s}_0(a)\}|}{|A} \tag{2.10}$$

Using these definitions, we see that makespan and stability of a Partial Order Schedule under disturbances depend on its earliest start schedule, which is in turn a function of both the precedence constraints and activity durations. Since the durations can be affected by disturbances, the desirable properties of a disturbed POS are dependent on both the structure of the POS and the disturbances.

However, we would like to increase robustness of the Partial Order Schedules. In order to measure this increase, we need to compare POSes with each other. For comparing POSes with each other on a purely structural level, we therefore define the comparison of robustness as follows:

**Definition 17.** One Partial Order Schedule is more robust than another Partial Order Schedule if, under equal disturbance, the makespan and stability of the first POS are at least as good as those of the second schedule and at least one of these metrics is better for the first than for the second POS.

Note that this does not specify which of two Partial Order Schedules is more robust if one has better makespan and the other better stability; this is because this is impossible to tell without knowing the relative importance of both metrics. Also note that this makes the notion “more robust” dependent on the kind of disturbance: under one particular disturbance, one Partial Order Schedule might seem more robust than another, while this order might be reversed when the disturbance is changed.

Using this information, we can now answer Research Question 2 as follows: partial order schedules are only robust under disturbances in activity duration, and they can be more or less robust depending on their structure and the circumstances in which they are executed.

# Chapter 3

# Related Work

In this chapter we will consult existing work on our two remaining research questions:

3 What is the relationship between the method used for chaining and the robustness of the resulting Partial Order Schedule?

4 How can methods for creating Partial Order Schedules be adapted to create more robust Partial Order Schedules?

We will try to find an answer for these questions in three parts. In the first part we will describe measures of robustness, and how these can be influenced. In the second part we will describe measures which are related to robustness, but do not measure robustness itself. Of these measures, we will also discuss the links to robustness: argumentation used to link them, correlations found, and so on. Finally we will describe methods shown to improve some of metrics in Partial Order Schedules.

## 3.1 On-line Robustness Measures

In this section we describe measures we can use to measure robustness in an on-line setting. This means that these measures indicate robustness when a schedule has been executed with disturbances. They can indicate robustness by measuring the impact of disturbances on the quality of the schedule; the more robust a schedule is, the smaller the impact on quality.

### 3.1.1 Number of Delays

The first robustness measure is the number of delays, or the number of activities of which the start time is delayed. This measure is also used in Wilson et al. (2013) Given the earliest start schedule for the undisturbed POS $\hat{s}_0$ and the earliest start schedule for the POS under disturbance $\hat{s}_d$, the number of delays is calculated as follows:

$$numDelays = |\{a \in A | \hat{s}_d(a) > \hat{s}_0(a)\}| \tag{3.1}$$

Ignoring the factor $|A|$, this is the exact opposite of the stability of a schedule. We therefore consider this metric to be a good measure for this aspect of robustness.

### 3.1.2 Sensitivity

A metric that is related to the Number of Delays is sensitivity, a measure introduced in Rasconi, Cesta and Policella (2008). Instead of only counting the number of delayed activities, it also takes the amount of delay into account. This measure calculates the mean difference between the scheduled and executed start time of each activity in the Partial Order Schedule. Given the earliest start schedule for the undisturbed POS $\hat{s}_0$ and the earliest start schedule for the POS under disturbance $\hat{s}_d$, the sensitivity is calculated as follows:

$$snty = \frac{1}{|A|} \sum_{a \in A} (\hat{s}_d(a) - \hat{s}_0(a)) \tag{3.2}$$

Similar to the Number of Delays, we consider this measure to be a good measure for (the lack of) stability.

### 3.1.3 ΔMakespan

Finally, we found a measure that captures makespan, which is the main quality measure of schedules. In order to capture the deterioration in makespan, we take the difference of the makespans of the earliest start schedules of the undisturbed POS and the POS under disturbances. However, if the delays are dependent on the durations of activities, this metric would deem POSes with shorter durations as naturally more robust, regardless of their structure. To overcome this, we normalize the difference in makespan by the sum of activity durations. Given the earliest start schedule for the undisturbed POS $\hat{s}_0$ and the earliest start schedule for the POS under disturbance $\hat{s}_d$, Δmakespan is calculated as follows:

$$\Delta Mks = \frac{mks(\hat{s}_d) - mks(\hat{s}_0)}{\sum_{a \in A} d_a} \tag{3.3}$$

Since this metric captures the difference in makespan, we deem it a good measure for the difference in makespan.

## 3.2 Off-line Robustness Predictors

While on-line measures are very good at measuring robustness by the impact of disturbances, they require that the schedule is executed with a certain disturbance to be able to measure something. Being able to robustness in a schedule without having to execute the schedule and without being tied to a specific disturbance would be a great improvement. However, a measure that accurately captures robustness — at least following our definition — has not yet been found. Instead, we will focus in this section on offline measures, i.e. measures that do not require execution, that predict or are associated with robustness.

### 3.2.1 Slack-Based Robustness Measures

In Chtourou and Haouari (2008), the authors have presented a set of twelve measures for robustness in Fixed Time Schedules. The measures are all based on slack, which is defined for an activity as “the time that an activity *i* [...] can slip without delaying the start of any of its immediate successors, while upholding resource feasibility”. In other words: the amount of time that an activity can be postponed without affecting any other activity.

This can be translated to Partial Order Schedules, using its earliest start time $\hat{s}$.

$$slack(a) = \min_{x \in succ(a)} (\hat{s}(x) - (\hat{s}(a) + d_a)) \tag{3.4}$$

In their paper, the authors propose three different variants of this base metric: one is slack as described here, the second is a binary value that is set to one if and only if the slack is greater than zero, and the third is the minimum of slack and a fraction of the activity’s duration based on the expected delay. The slack of the complete schedule is calculated by taking the weighted sum of the slack values of the activities, using one of four weightings. In the first weighting all activities weigh equal, the second weighting counts each activity once for each of its successors, the third weighting multiplies the slack value by the number of resource units the activity requires, and the final weighting is the combination of the second and third: the weights are the multiplication of the number of successors and the number of resource units of an activity.

In the paper, the authors evaluate the robustness measures in combination with a set of priority rules for creating a schedule. The priority rules are used in the Serial Generation Scheme, or SGS, as described in Hartmann and Kolisch (2000) to create different schedules. To briefly summarize SGS: it is a greedy algorithm for scheduling events one at a time. During execution, the algorithm maintains a set of activities of which all predecessors are already scheduled. From this set, one activity is picked at random, with probabilities indicated by the priority rules, and scheduled at the earliest resource-feasible time. After this, the set of available activities is updated and the process repeats until all activities are scheduled.

In order to evaluate the robustness measures, the authors use the SGS to generate a large number of schedules. From these schedules, the schedules with the lowest makespan are selected and from these, the schedule that is the least robust according to the metric. Then, another set of schedules is generated, from this set they select the schedules with a makespan less than or equal to the “non-robust schedule”, and from this selection they pick the most robust schedule (again, according to the metric). This way, two schedules with comparable makespan but the largest possible difference in robustness are selected.

These schedules are then tested for robustness by increasing the duration of a randomly selected portion of activities by a certain amount. New, delayed schedules are then created using a method similar to SGS, where the chronological order of the original schedules instead of a stochastic process is used to determine the order of scheduling the activities. The schedule of which the ordering produced the lowest makespan schedule in this step is considered the most robust.

Among other results, they found that most of their measures are correlated with robustness. That is: of the two schedules with the lowest and highest found values of the robustness metric, the schedule with the highest value was also considered more robust in the evaluation. Unfortunately, their results cannot be directly translated to our work, mostly because of the difference in the problem domain. Not only is the type of schedule used in their work different, our definition of robustness is also broader than what is measured in this work to be robustness. However, it is interesting to see whether similar results can be obtained, and what this tells about robustness of POSes.

### 3.2.2 Robustness Measures in Partial Order Schedules

Another set of robustness measures is the set used by Policella et al. (2004, 2007, 2009). Similar to the measures used by Chtourou and Harouari, they are not used as robustness measures but as an indication. This is best formulated in Policella et al. (2007):

> *We expect a flexible schedule to be easy to change, and the intuition is that the degree of flexibility in this schedule is indicative of its robustness.*

#### Flexibility

One measure used by Policella et al. is called flexibility. This measure was introduced in Aloulou and Portmann (2003) as sequential flexibility, or $Flex_{seq}$. It is originally defined as being “equal to the number of non-oriented edges in the transitive graph representing the partial order”. Policella et al. (2007) slightly changed the definition of this metric to the following equation:

$$flex_{seq} = \frac{|\{(a_i, a_j) \mid a_i \nprec a_j \wedge a_j \nprec a_i\}|}{n(n-1)} \tag{3.5}$$

In this equation, $n$ stands for the number of activities in the schedule, and $a_i \nprec a_j$ means that $a_i$ is not necessarily scheduled before $a_j$. The set $\{(a_i, a_j) | a_i \nprec a_j \wedge a_j \nprec a_i\}$ is therefore the set of all pairs of activities which are not ordered relative to each other — either one can be scheduled first, or they can be scheduled concurrently.

Where Aloulou and Portmann intended to count the number of non-ordered pairs of activities, Policella et al. decided to normalize this metric by dividing by $n(n-1)$ — the total number of pairs of activities. This distinction is important, and allows the metric to be compared between partial order schedules with different numbers of activities. Therefore we prefer the definition of Policella et al. over that by Aloulou and Portmann.

#### Fluidity

Fluidity is another metric used by Policella et al., intended to measure the amount of movement that is allowed for pairs of activities. It is introduced in Cesta, Oddi and Smith (1998), originally called Robustness. It is defined as follows:

$$fldt = \sum_{i=1}^{n} \sum_{j=1 \wedge i \neq j}^{n} \frac{slack(a_i, a_j)}{H \times n \times (n-1)} \times 100 \tag{3.6}$$

In this equation, $n$ is again the number of activities, $H$ is the horizon or the sum of all durations, and $slack(a_i, a_j)$ is the amount of relative movement possible between the two activities. The denominator, $H \times n \times (n-1)$, is again here for normalization to ensure that schedules with different numbers of activities can still be compared.

This measure is interesting, because it focuses on absorbing disturbances locally — preventing other activities to be affected by a single disruption. However, this particular measure is designed for the RCPSP/max problem, where constraints can also have a minimum and maximum time between activities instead of just representing an ordering. Therefore, the slack metric cannot be applied to our case, where the only constraints are precedence constraints.

### Pairwise Float

A metric that combines elements of the previous two is called Pairwise Float. It is introduced in Braeckmans et al. (2005) as an objective function for creating robust schedules using (mixed) integer programming. Similar to chaining, their method also uses a fixed-time schedule to create a partial order schedule. This metric is therefore based on the fixed-time schedule, as well as the partial order schedule it measures.

Slightly deviating from the original notation, we can define pairwise float for a pair of activities $(a_i, a_j)$ as follows:

- If the pair $(a_i, a_j)$ is directly connected by a single, non-transitive precedence constraint, the pairwise float is defined as $PF_{a_i,a_j} = s(a_j) - (s(a_i) + d_{a_i})$.
- If the pair $(a_i, a_j)$ is connected by a series of precedence constraints to form a path $\pi = [a_i, a_{x_1}, a_{x_2}, \dots, a_{x_n}, a_j]$ instead of a single, non-transitive constraint, the pairwise float is determined by the pairwise float of the path. The pairwise float of a path $\pi = [a_1, a_2, \dots, a_{n-1}, a_n]$ can be defined as

  $$PF_\pi = \sum_{i=1}^{n-1} PF_{a_i,a_{i+1}}$$

  If $\Pi$ is the set of all paths from $a_i$ to $a_j$, then the pairwise float is defined as the minimum of those paths:

  $$PF_{a_i,a_j} = \min_{\pi \in \Pi} PF_\pi$$
- If the pair $(a_i, a_j)$ is not related by precedence constraints but the two activities are not scheduled concurrently in the fixed-time schedule, i.e. $s(a_i) + d_{a_i} \leq s(a_j)$, the pairwise float is set to a certain constant: $PF_{a_i,a_j} = C$.
- In any other case, $PF_{a_i,a_j} = 0$.

The pairwise float of the schedule is then defined as follows:

$$PF = \sum_{a_i,a_j \in A} PF_{a_i,a_j} \tag{3.7}$$

The value of this metric is highly dependent on the value of $C$: if $C$ is small, the pairwise float of connected pairs of activities is the main contributor to this value. If $C$ is large, the number of pairs of activities which are not related by precedence constraints becomes a more dominant contributor. Good values for $C$ are not given, except that the authors use the value $C = 10$ in their own experiments — without mentioning how they came to that value.

#### Disruptibility

Disruptibility is a metric introduced by Policella, Smith and Oddi (2004). It measures, for each activity, the amount of moving space, but takes into account the number of activities that are affected by that movement. It is defined as follows:

$$dsrp = \frac{1}{n} \sum_{i=1}^{n} \frac{lst_{a_i} - est_{a_i}}{num_{changes}(a_i, \Delta_{a_i})} \tag{3.8}$$

In this equation, $n$ is again the number of activities, $est_{a_i}$ and $lst_{a_i}$ are, respectively, the earliest and latest start time of $a_i$, and $num_{changes}(a_i, \Delta_{a_i})$ stands for the number of activities which are affected by postponing $a_i$ by $\Delta_{a_i}$. As example, the authors use $lst_{a_i} - est_{a_i}$ as value for $\Delta_{a_i}$. The latest start time of an activity $lst_{a_i}$ does require the activity to have a deadline, be it a deadline on the activity itself, on one of its successors or on the entire schedule. This way we know how far back we can push the activity without making the schedule invalid.

This measure again focuses on local absorption of disturbances, but this time also suggesting that it is not forbidden to move other activities. It is however unwanted to postpone other activities, therefore affecting more activities decreases the value for this measure. This metric needs some adaptation for RCPSP — it, too, is created for RCPSP/max — but it might be a good measure for robustness.

## 3.3 Influences of Algorithms on Robustness

Many authors have put effort into creating more robust schedules, using one or more of the mentioned metrics as target measure. In this section, we will summarize the most important of these efforts.

### 3.3.1 Improving robustness in Fixed-Time Schedules

Chtourou and Harouari optimize for robustness measures using a basic black-box optimization scheme: producing many different solutions and pick the one with best value. Depending on the used priority rules, a difference in predicted robustness (using one of the measures) would match a difference in evaluated robustness.

### 3.3.2 Maximizing Flexibility in Chaining

As mentioned, Policella et al. (2007, 2009) aim for flexibility and fluidity in order to increase robustness of schedules. Not only does their method, solve-and-robustify, in itself produce Partial Order Schedules that have more flexibility and fluidity than previous, comparable algorithms, the authors also developed two heuristics to guide the chaining process.

#### Maximize Common Chains

The first method is called MaxCC, which stand for Maximize Common Chains. This heuristic aims to reduce the number of added precedence constraints by maximizing the number of chains it has in common with each predecessor. If one chain is selected, the next chain is selected such that it has the same last activity — i.e. the activity has the same predecessor in this chain as in the previously chosen chain. That way, both chains require the same precedence constraint to be added.

When compared to randomly selecting a chain, without regard for earlier decisions, this can greatly reduce the number of different predecessors for each activity. As a result, this heuristic decreases the number of precedence constraints, meaning an increase in flexibility and fluidity — after all, each added precedence constraint can only reduce fluidity.

#### Minimize InterDependencies

The second method is called MinID, which stands for Minimize InterDependencies. This heuristic aims to reduce the number of added precedence constraints by reusing existing precedence constraints. If there is a chain such that the last activity on that chain is already a predecessor of the current activity, this activity is added to that chain. In other words: when assigning activity $a$ to a chain, it tries to find a chain $x$ such that $last(x) \in pred^*(x)$.

This reduces the number of add precedence constraint, since there already is a precedence constraint between the two subsequent activities. This increases the flexibility and fluidity of the resulting POS.

#### MaxCCminID

Furthermore, this heuristic can be used in combination with the previous heuristic, forming a heuristic called maxCCminID. This heuristic dictates that, for each chain, first maxCC is used to try and find a suitable chain. If that does not yield a result, for example because no previous chain was selected, the minID heuristic is used to find a preferred chain. Only if that also does not yield a result, a chain is selected randomly. This heuristic combines the benefits of both heuristic, and increases the flexibility and fluidity of the resulting POS even more.

#### Bias toward Flexibility or Fluidity

In these heuristics, there is still a stochastic component. Policella et al. try to utilize this stochastic component by using black-box optimization to introduce a small bias towards either Flexibility or Fluidity. They do this by generating several Partial Order Schedules and choosing the POS with the highest value for one of the two metrics. However, the effect of this bias is quite small compared to the effect of the two heuristics.

### 3.3.3 Optimizing Metrics using Mixed Integer Programming

In Braeckmans et al. 2005, the authors use (mixed) integer programming to optimize a certain heuristic, in order to decrease the expected value of sensitivity. This expected value is based on some assumptions on the expectation of the delays. In their research, the authors compare several objective functions for integer programming to each other and to the chaining algorithms by Policella et al.

Following the intuition that flexibility has a positive impact on robustness, they introduce the MinEA objective function. This function minimizes the number of Extra Arcs, or the number of added precedence constraints. Using it increases the stability of the produced POS in their results compared to the POS produced using chaining.

The next objective function that Braeckmans et al. use and examine is the MaxPF objective function. This objective function maximizes the Pairwise Float, which, as discussed earlier, incorporates aspects of both flexibility and temporal slack. Compared to the MinEA objective function, POSes which are optimized for this function have even more stability.

Finally, the authors introduce an objective function that aims to optimize the expected stability itself. It is called MinED, for Minimize Expected Disturbance, and uses a simplified model of stability to optimize the expected stability. This produced even more stable POSes compared to the MaxPF objective function. However, this method was very computationally expensive, making it not very usable for practical applications.

### 3.3.4 Greedy Algorithm using Simulation to Increase Stability

Following the research using integer programming, in the same paper Braeckmans et al. develop MABO: Myopic Activity-Based Optimization. This is a greedy algorithm which considers all activities once and one at a time, similar to the chaining approach.

For each activity it first determines from which other activities resources are needed. This is chosen by taking the available resources from the activity's predecessors, supplemented with available resources from other activities when needed. If there are multiple candidate activities to choose from, a small-scale simulation is used to determine the most stable activities. If there are more resources available from these activities than is needed, the resources are taken from the activities with the fewest successors. This is done to increase the availability of resources for those successors.

This algorithm was also evaluated to test both the execution time and stability. The stability of the produced POSes was roughly equal to the stability of the POSes created using MinED — the most successful of the integer programming-based solutions. The execution time, on the other hand, was quite low: in the order of magnitude that one could expect from a simple greedy algorithm, as compared to the much longer execution time of the MinED algorithm.

# Chapter 4

# Problem Statement and Methodology

Having defined our problem domain and the work done in other research, in this section we will state our problem statement and how we will approach this. We will do this by re-iterating our research questions, and break down how much of this has been answered and what questions still need to be answered. Following the detailed problem statement, we will discuss how we plan to answer these questions.

## 4.1 Detailed Problem Statement

We defined our research questions as follows:

1. What is robustness?
2. To what extent are Partial Order Schedules robust?
3. What is the relationship between the method used for chaining and the robustness of the resulting Partial Order Schedule?
4. How can methods for creating Partial Order Schedules be adapted to create more robust Partial Order Schedules?

Questions 1 and 2 have been answered in Chapter 2. To summarize the answers: robustness of an object is the ability of that object to retain or regain its properties during or following adverse conditions. For scheduling this means that a robust schedule needs little or no modification to be usable if, during execution, properties of the instance are different than expected during planning. Partial Order Schedules are able to cope with temporal disturbances: if the durations of activities change, the POS can be used to quickly create a new fixed-time schedule. The properties of this fixed-time schedule depend on the structure of the POS and the disturbance.

This leaves questions 3 and 4 left for answering. In Chapter 3, part of the answers were given, which we will summarize here. There are several proposed relationships between the method used for chaining and the robustness of the resulting POS. These are slack, flexibility, fluidity, and pairwise float; each of these have demonstrated to have — at least in some cases — some positive correlation with robustness. Flexibility and fluidity can be increased using the maxCC and minID heuristic, for the other measures no specific algorithms have been proposed.

Most of the algorithms that are proposed try to increase some metric using some form of optimization scheme, instead of devising specific algorithms that exploit the problem structure. This includes using an IP solver, as Braeckmans et al. did for the minEA and maxPF metrics, or using a black-box optimization approach of generating several slightly different solutions and picking the most suitable one, as was done for many of the other metrics. A special subcategory of this can be found in the MinED priority rule and the MABO algorithm, both proposed in Braeckmans et al. (2005), which use simulation to measure the effects of a certain choice. Although they can be very effective and, thanks to continuing development on specialized solvers, quite fast, these algorithms teach us very little on the nature of robustness.

Filling in the partial answers found in Chapter 3, there are some gaps left to fully answer the research questions. In order to use earlier findings for constructing an answer to our third research question, we need to answer the following research questions:

- Are the slack-based metrics from Chtourou and Harouari and their results with respect to robustness transferable to Partial Order Schedules and our definition of robustness?
- Are Flexibility and Fluidity actually related to robustness?
- Are the metrics used by Braeckmans et al. and their results with respect to stability transferable to our definition of robustness?

If we have an answer to these questions, we have to answer the following questions to complete the answers to research questions 3 and 4.

- Is there an underlying mechanism in the relationship between the method used for chaining and the robustness of the POS that causes the found correlations, and if so: what is it?
- Can we use the found results to adapt chaining in such a way that it produces more robust partial order schedules?

## 4.2 Methodology

In order to find answers to these questions, we used an empirical approach to examine the behaviour of the algorithm and its results under different circumstances. We will first expand on this approach and the motivations for this approach. After that, we will explain our implementation of this approach in a simulation environment.

### 4.2.1 Empirical approach

The experimental approach we used is based on the approaches by Hooker (1995) and McGeoch (1996). The idea of this approach is that algorithms can exhibit a certain behaviour, which can be studied to learn about the workings of the algorithms. In this approach we follow the well-known scientific method of observation, hypothesising, prediction, and experimental validation.

We chose to use this approach because the problem is very complex: there are many activities which can all be delayed, and these delays can influence other activities in different ways. Using an analytical approach would probably provide some insights on the low-level behaviour, but on a higher level the interactions between activities become hard to predict. An empirical approach might show us the results of these interactions, providing us with some insights on how the algorithm works on a higher level.

### 4.2.2 Implementation of the Emprical Approach

First of all, we performed an exploratory study to gather observations. The experiments in this study are derived from existing work, which is redone to get observations from a consistent framework and within our scope. We list the experiments and their results in Chapter 5.

Using the observations form this study, we created a model to provide an answer to the questions. The model of this step, which is the "hypothesising" part of the scientific method, should be consistent with current observations, as well as provide new insights. In order to accomplish this, the model should describe the inner workings of the algorithm, explaining why the algorithm exhibits certain behaviour rather than simply listing this behaviour. We describe this model in Chapter 6.

From this model we can derive predictions on the chaining algorithm: they predict behaviour of the algorithm under unknown conditions. These predictions can then be tested by setting up the required conditions in an experiment and recording the behaviour. If this behaviour matches the predicted behaviour, we can add confidence in the validity of the model. However, if this match is not there, this means that the model is invalid and another model needs to be found that does match this new observed behaviour. The predictions we made and the experiments we performed to test these predictions are described in Chapter 7.

### 4.2.3 Simulation Environment

An important distinction in the work of McGeoch is the notion of what an algorithm is. In their view, an algorithm is only an abstract idea. For using it in a system we implement it, and for investigating it we simulate the algorithm. Although the simulation might consist of the same implementation as real-world usage, the main difference with a simulation is that it is focused on understanding the algorithm. This means that in the simulation it is more important to have controlled conditions and reproducable and detailed outcomes than to have good performance in terms of time and memory. This is done in a simulation environmet.

Our simulation environment consists of three parts: the algorithm input, an implementation of the algorithm itself, and an assessment of the output. Since the chaining algorithm consists of two steps, the "solve" step and the "robustify" step, the entire system forms a pipeline of four steps. In our environment, the intermediate results between each step are saved. This allowed us to reuse these results when variation was needed only in later steps in the pipeline, isolating the difference in those steps from difference produced by a stochastic process in an earlier step of the pipeline. Furthermore, analysis can be done on a very low level without having to redo a simulation.

We designed the steps of our pipeline to have an easily interchangeable implementation, to easily vary the parameters. Following McGeoch (1996), we denote each combination of parameters as a design point.

### Pipeline Details

As input we used the PSPLIB benchmark set for RCPSP, as introduced in Kolisch and Sprecher (1996). This benchmark sets consists of four instance sets: J30, J60, J90, and J120. The instance sizes of these are respectively 30, 60, 90, and 120 activities. For the "solve" step, we created fixed-time schedules for these RCPSP instances using a very simple Serial Generation Scheme.

Using the fixed-time schedule, the next component creates a partial order schedule. The main implementation of this component uses chaining, since it is the subject of our study, but we also have an alternate implementation using Mixed Integer Programming. These implementations are described in more detail in Appendix A.

Finally, the output assessment is probably the most important part in this scheme. We assessed the robustness of the Partial Order Schedules by simulating delays and measuring the impact of these delays. The delays we introduced are relative to the activity duration, following the real world where long activities can be delayed more than short activities.

The delays come from a list of real numbers which denote the ratios of the simulated duration compared to the expected duration. For example, a ratio of 1 means that the duration remains the same, while a ratio of 1.3 indicates that an activity's duration is increased by 30%. We call a single list of ratios a delay pattern. For testing, we generated several sets of delay patterns, which we call delay sets, in advance. These sets all have delay patterns which are randomly generated using the same probability distribution, ensuring that this probability distribution is quite thoroughly tested. Generating these delay sets in advance instead of generating the delays on the fly increases reproducability of our experiments.

For diversity in our research, we generated several delay sets. Since we could not find consensus on distribution of activity delays, we chose four delay sets with delays generated from four different probability distributions.

**exp2** A delay set with factors chosen from an exponential distribution with $\lambda = 2$, with a minimum factor of 1. The idea of an exponential distribution comes from the fact that, if an activity is delayed, a small delay is more likely than a large delay. Setting a minimum of 1 makes sure there are no activities being shorter, which is outside of the scope of the research. In this set, on average 14% of activities is delayed, with a mean factor of 1.51. This results in a mean delay of 7.1% over all activities.

**fixed_50_30** A delay set where one half of the activities is delayed by 30 percent, or $P(x = 1) = 0.5$ and $P(x = 1.3) = 0.5$. This distribution is inspired by the delay distributions used in Wilson et al. (2013), and is meant as an example where many activities are delayed but only by a small amount.

**unif_80_5** A delay set where twenty percent of activities is delayed, by a factor chosen from a uniform distribution between 1 and 5. This distribution has a slightly larger portion of activities delayed than the *exp2* delay set, but a smaller portian than the *fixed_50_30* set. The delays are quite large, to simulate a scenario where there is a lot of delay, testing the robustness of schedules under quite extreme conditions.

**gauss_02** A delay set where delays are chosen according to a gaussian distribution with $\mu = 1$ and $\sigma = 0.2$. This is the only set where factors smaller than 1 occur, and it is chosen for exactly that reason. However, since the main goal of our thesis entails robustness in adverse conditions, test results with this delay sets will be considered auxiliary — they may be informative, but will not be used for the main goal.

In a simulation, we adapt the POS by changing the durations according to the delay pattern, as described in Section 2.2.3, and record the resulting earliest start schedule. On this delayed schedule, we measure robustness using $\Delta$makespan and number of delays. These are two of the measures we found in literature and described in . Note that we do not use sensitivity, but only the other two metrics. We opted not to include this metric, because the temporal aspect is already included in $\Delta$makespan and because we want to prioritize the number of delays over the amount of delay. The motivation for this prioritization comes from scheduling projects with external contractors. In this case the act of rescheduling an activity can be very costly, regardless of the difference between the old and new start time of the activity.

In addition to the metrics we classified as on-line robustness measures, we also measure several off-line robustness predictors as described in Section 3.2. The purpose of these metrics is to find correlations between these predictors and our on-line robustness measures, helping to answer our research questions. We measured flexibility on the POSes, as defined by Policella et al., and the slack-based metrics by Chtourou and Haouari on the earliest start schedule of the POSes. We chose not to use the other metrics, since they require also a deadline for the schedule. Setting this deadline to the makespan of the earliest start schedule would produce a metric very similar to one of the slack-based metrics. Choosing a deadline larger than this would require us to make a choice of deadline, introducing another variable into our research.

All of the code is currently available, with the current version released in Wilmer (2014) and the latest version availble on GitHub[1]. This ensures reproducibility of our experiments and, hopefully, encourages other researchers to perform experiments in the same field.

[1]`https://github.com/dwilmer/rcpsp-testing-framework`

# Chapter 5

# Exploratory Study

The first step towards the understanding of robustness in partial order schedules and chaining, is to make observations on the behaviour to find patterns. For this purpose, we performed an exploratory study. This is a study where we vary parameters and capture the results, simply to get observations on the algorithm's behaviour.

In this study, we tested all combinations of instance set (J30, J60, J90, or J120) and delay set (as described in Section 4.2.3) in combination with several different chaining heuristics, unless otherwise specified. These heuristics come from related work, as described in Section 3.3.2. We call them basic chaining, where the chain assignment is not guided by any heuristic, and maxCC, minID, and maxCCminID for the three heuristics. First we compare the robustness between between the heuristics, after which we focus on the relationships with the robustness predictors flexibility and slack.

We determine whether any differences are significant using significance testing. This test can determine the probability that two sets of test data come from the same stochastic process, i.e. the probability that the results are a product of chance rather than a difference in the process. This probability is called the $p$-value, and for a difference in two sets to be significant this value should be below some threshold. Common values for this threshold are 0.05, 0.01, and 0.001, meaning that there a chance less than respectively 5%, 1%, or 0.1% that the results are accidental rather than systemic.

The significance test we use is the Wilcoxon signed-rank test, as described in Wilcoxon (1945). This test is used for paired data that is not normally distributed. The pairing comes from the fact that we compare the results on a per-instance basis, as opposed to viewing the results as a whole. This reduces the impact of large differences between instances, and instead focuses on differences between runs per instance.

## 5.1 Robustness Differences between Algorithms

The first thing we want to explore, is whether there are actually differences in the robustness of the Partial Order Schedules generated using the different methods. Therefore we compare the four chaining methods from Policella et al. (2009), and compare robustness between them. For this we measure and compare both Δmakespan and the number of violations.

### 5.1.1 Δmakespan

In Figure 5.1 we compare the normalized makespan increase of basic chaining, maxCC, minID, and maxCCminID, using the J30 instance set and exp2 delay set.

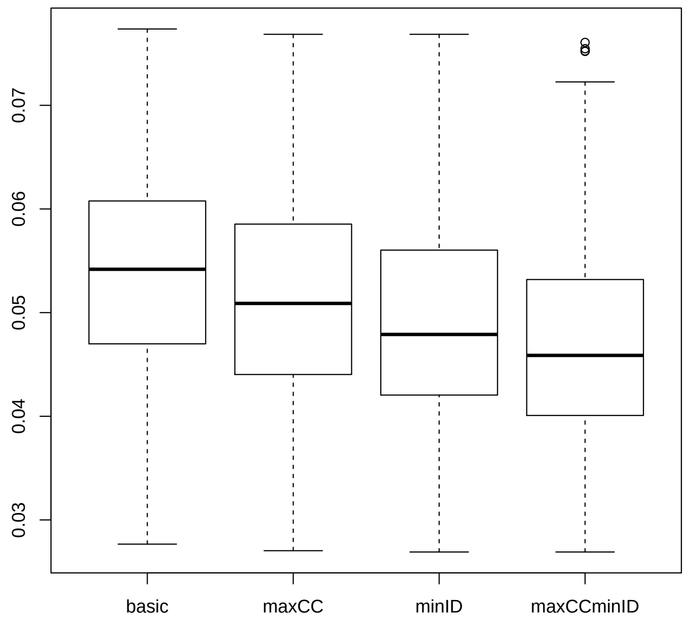


Figure 5.1: Comparison of normalized makespan increase *J30 instance set, exp2 delay set*

This shows that there clearly is a difference between the Δmakespan for the different chaining methods. Remarkably, the figure for the instance sets with larger instances (J60, J90 and J120) looks nearly identical, as shown in Figure B.1 in Appendix B. There is, however, more variation in the comparison of Δmakespan when using different delay sets. The figures for all different delay sets, using the J30 instance set, can be found in Figure B.2 in Appendix B.

Significance testing using the Wilcoxon signed-rank test shows that the differences are all statistically significant with a $p$-value of $p < 0.001$.

### 5.1.2 Number of Violations

In Figure 5.2 we compared the number of violations of basic chaining, maxCC, minID, and maxCCminID, using the J30 instance set and exp2 delay set.

Again, there is a clear difference in the number of violations occurring in the simulated executions of POSes created using different approaches. There are some more differences in the plots for different instance sets or delay sets, but the trend is in all cases the same.

Using the Wilcoxon signed-rank test, we found that the difference between basic chaining and maxCC, and the difference between minID and maxCCminID are statistically significant with a p-value of $p < 0.001$. The difference between maxCC and minID is not

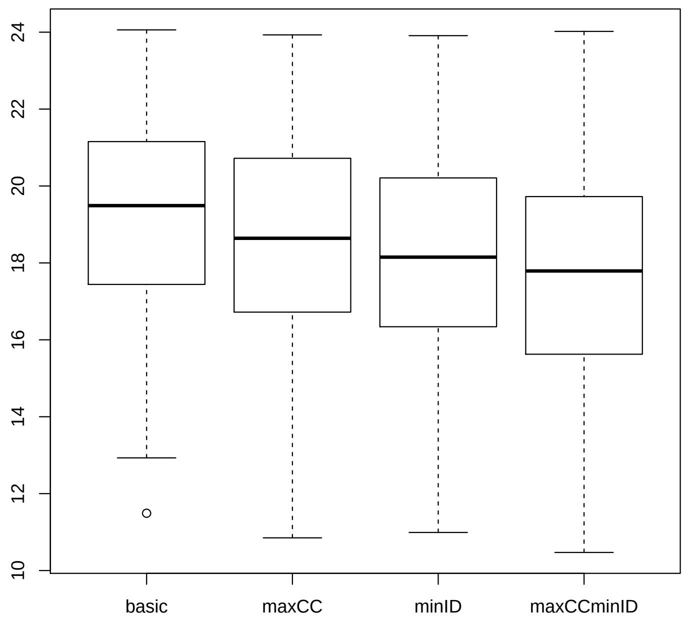


Figure 5.2: Comparison of number of violations *J30 instance set, exp2 delay set*

statistically significant with $p < 0.001$ when tested using one of the following three parameter combinations:

- J30 instance set and fixed_50_30 delay set, this produces a p-value of $p = 0.039$;
- J90 instance set and unif_80_5 delay set, this produces a p-value of $p = 0.001$;
- J120 instance set and fixed_50_30 delay set, this produces a p-value of $p = 0.147$.

Although the difference is not significant in all cases, these values are low enough that we can still state that, overall, minID tends to produce more robust POSes than maxCC.

## 5.2 Flexibility

To investigate the origin of the trend described in the previous section, we look for measures which are correlated with robustness. The first measure we investigated is flexibility, which is used by several authors as a surrogate measure for robustness. For example, Policella et al. (2007) justify this use as follows:

*We expect a flexible schedule to be easy to change, and the intuition is that the degree of flexibility in this schedule is indicative of its robustness.*

For this reason, we first investigated flexibility and its correlation to robustness.

### 5.2.1 Differences in Flexibility

In Figure 5.3 we compared the flexibility of the Partial Order Schedules created using different methods. For this plot, we again used the J30 instance set. As can be seen in Figure B.3 in the Appendix, the plots are quite similar for the other instance sets.

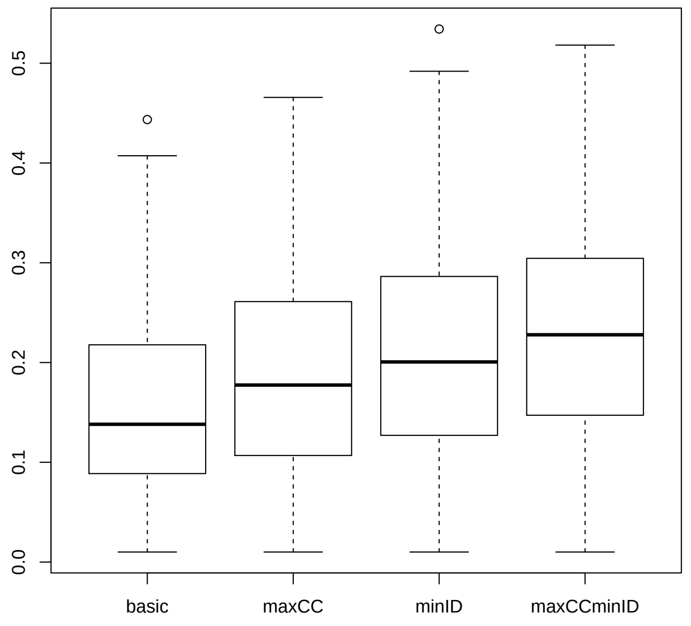


Figure 5.3: Comparison of Flexibility *J30 instance set*

In the plot of Figure 5.3, we observe three important facts. First of all, there is a clear trend that maxCC tends to produce more flexible POSes than basic chaining, minID tends to produce more flexible POSes than maxCC, and maxCCminID tends to produce more flexible POSes than minID. Given that the methods which tend to produce more flexible POSes also tend to produce more robust POSes, as shown in Figures 5.1 and 5.2, this is the first indication that the intuition from Policella et al. (2007) might be correct.

The second observation is that the variance of flexibility for a given instance set and chaining method is very large compared to the differences of flexibility between methods. This is similar to the large variances in $\Delta$makespan and number of violations.

Finally, we observe that the flexibility ranges from values barely larger than 0 to just over 0.6. This means that some of the partial order schedules are actually an almost complete ordering, while in other POSes the activities are, on average, connected to less than half of the other activities. We suspect that this large range is partly due to the large differences in instances in the PSPLIB library, as intended by Kolisch and Sprecher.

In addition to the observations in the plot, we used the Wilcoxon signed-rank test to test whether the differences are significant. With a p-value of $p < 0.001$, the difference in flexibility is significant.

### 5.2.2 Correlation Testing

To further investigate a possible connection between flexibility and robustness, we investigated correlation. We did this by measuring flexibility and robustness in many different POSes, generated using the different chaining approaches found in literature.

#### Δmakespan

We first tested the correlation between flexibility and makespan increase, using the four test sets and four delay sets. The correlation values using these sixteen different settings are listed in Table C.1 in the appendix. Most values indicate strong correlations: values are mostly between $-0.7$ and $-0.95$. The single exception arises when taking instances from the j30 set and testing using the exp2 delay set, and this will be investigated further. Note that these values are negative, indicating that Δmakespan decreases as flexibility increases, meaning that robustness increases as flexibility increases.

In Figure 5.4, we plotted the flexibility and Δmakespan for different Partial Order Schedules. The instances were taken from the J120 test set, and the used delay set is the unif_80_5 set. Four different chaining methods were used: basic chaining, maxCC, minID, and maxCCminID. The points corresponding to POSes created using these methods are respectively coloured black, red, green, and blue.

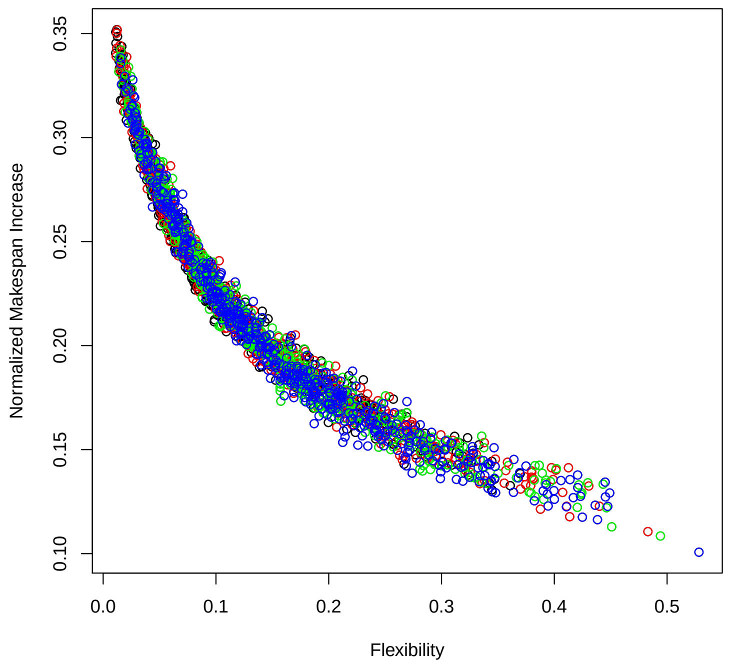


Figure 5.4: Normalized Makespan Increase compared to Flexibility *J120 instance set, unif_80_5 delay set*

Looking at the plot in Figure 5.4, a number of things can be observed. First of all, the points are concentrated around a single curve, shaped like a hyperbola. Secondly, no real distinction can be made between the different chaining methods: POSes created using different techniques can be found all around the curve. Plots for other instance sets and delay sets show the same shape, except when — again — using the j30 instance set and exp2 delay set, which we will also investigate further.

#### Number of Violations

The next test to find a correlation between flexibility and robustness, is to find a correlation between flexibility and the number of violations.

In Figure 5.5, we plotted the flexibility and number of violations for different partial order schedules. The instances were taken from the J120 test set, and execution was simulated using the unif_80_5 delay set. Four different chaining methods were used: basic chaining, maxCC, minID, and maxCCminID. The points corresponding to the POSes created using these methods are respectively coloured black, red, green, and blue.

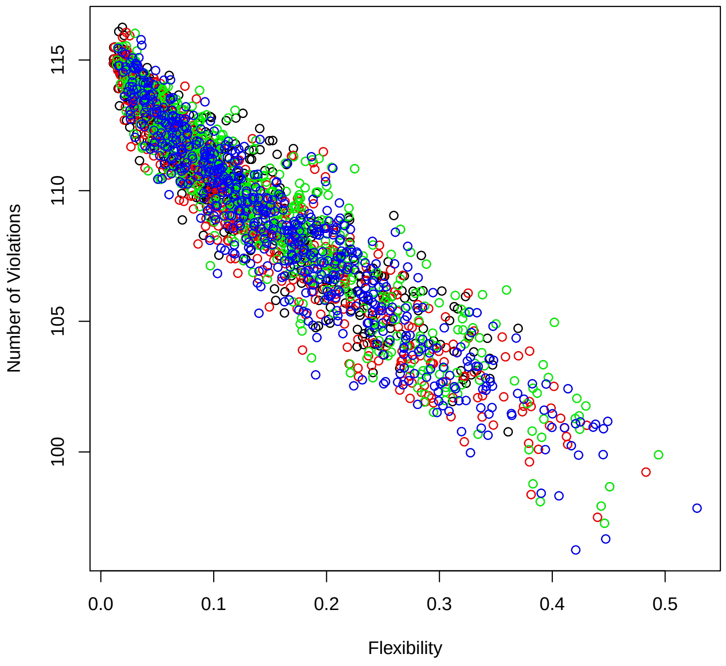


Figure 5.5: Number of Violations compared to Flexibility *J120 instance set, unif_80_5 delay set*

This plot shows an elongated cloud, showing a seemingly linear connection between the flexibility and the number of violations. This is in contrast with the hyperbola in the previous figure. In addition, the point cloud is less dense than the one found in Figure 5.4.

### 5.2.3 Comparison Testing

In order to minimize the influence of the instances, and instead highlight the differences between chaining procedures, we used comparison testing. In these tests, we compare partial order schedules to other POSes that are created for the same instance. One chaining method serves as a baseline, so that other chaining methods can be measured relative to this baseline instead of using absolute values. The idea is that properties of the instance that influence the partial order schedule, influence all POSes. Using one of these POSes as a baseline therefore filters our this influence, showing only the influence of using a different chaining method.

We perform these tests by dividing the values of the measures of the POSes by the values of the measures of the baseline POS. These ratios are then used as coordinates for plotting the points. Using this methods we can compare the change in two measures.

For example, suppose we want to compare maxCCminID to basic chaining, with the latter being the baseline. For one specific instance, maxCCminID might produce a POS with flexibility 0.42 and the basic chaining procedure results in a POS with flexibility 0.21. During simulation, the normalized makespan increase for the baseline POS might be 0.2, with the POS created by maxCCminID measures 0.15. In the comparison, we then plot a point at coordinates $(\frac{0.42}{0.21}, \frac{0.15}{0.2})$, being $(2, 0.75)$.

For easier understanding, we draw a horizontal line where the value on the vertical axis is 1.0, and a vertical line where the value on the horizontal axis is 1.0. This divides the area into four quadrants: the upper left, upper right, bottom left, and bottom right quadrant. This division does not add information, it merely makes it easier to read the information.

### ΔMakespan

In Figure 5.6, we plotted the flexibility and Δmakespan for different Partial Order Schedules, compared to that of a Partial Order Schedule created using basic chaining. Again, we used the J120 instance set and the unif_80_5 delay set. We used basic chaining as baseline to compare maxCC, minID, and maxCCminID, with the points being coloured red, green, and blue, respectively.

In the plot of Figure 5.6 we see that the points are clustered in an elongated cloud, ranging from the point $(1,1)$ to the bottom right. However, the hyperbola is absent, as is the strong concentration around a single line. The differences between chaining methods are, however, more distinct: maxCC (coloured red) does not improve flexibility as much as maxCCminID (coloured blue), compared to basic chaining.

### Number of Violations

In Figure 5.7, we plotted the flexibility and number of violations for different partial order schedules, using the partial order schedules created by basic chaining as baseline. For good comparison, we again used the J120 instance set and unif_80_5 delay set. We used basic chaining as baseline to compare maxCC, minID, and maxCCminID, with the points being coloured red, green, and blue, respectively.

Quite noticeable is that the difference in number of violations is only a few percent; the vast majority of points is between 0.98 and 1.00 on the vertical axis. However, far more striking is the observation that there is only a weak correlation between the increase in flexibility and the decrease in number of violations. This is also the first time we observe

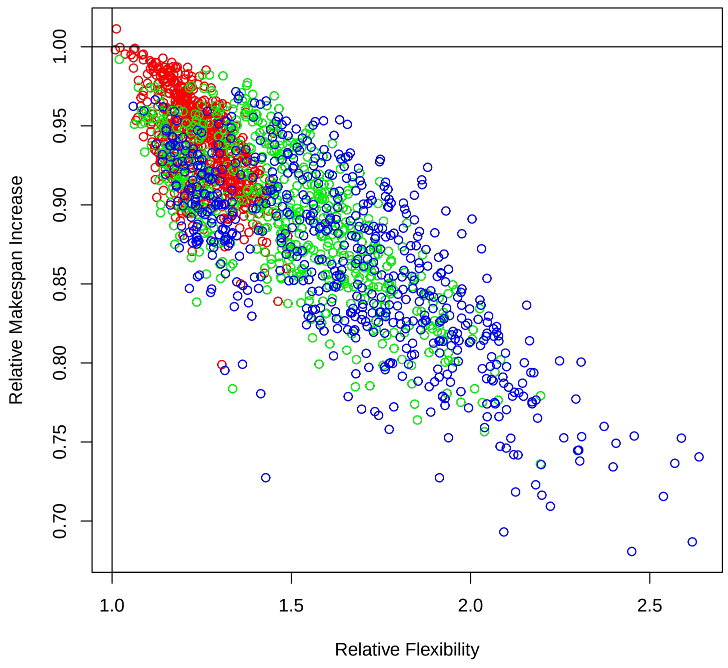


Figure 5.6: Normalized Makespan Increase relative to baseline, compared to Flexibility relative to baseline *J120 instance set, unif_80_5 delay set*

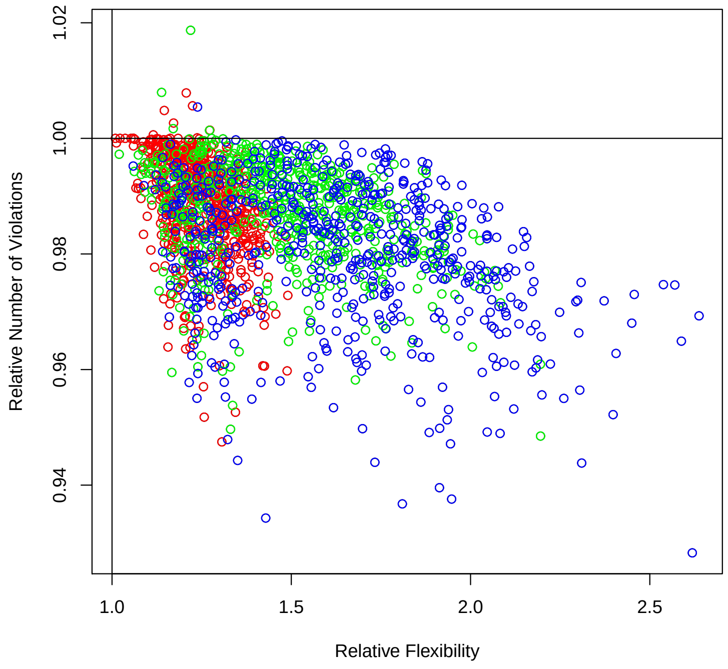


Figure 5.7: Number of Violations relative to baseline, compared to Flexibility relative to baseline *J120 instance set, unif_80_5 delay set*

points in the upper right quadrant, meaning that there are POSes for which the flexibility is higher than the baseline, but so is the number of violations.

### 5.2.4 Optimizing for Flexibility

Curious whether flexibility is an important property of robust schedules or merely a side product of the chaining procedure, we created POSes to optimize for flexibility. We did this using Mixed Integer Programming, in a manner similar to the approach in Braeckmans et al. (2005).

Using a MIP solver, we created Partial Order Schedules optimized for the lowest number of precedence constraints under transitive closure, thus optimizing for flexibility. This is described in more detail in Section A.2 in the appendix. We limited the amount of time available for each instance to 15 minutes, forcing calculation on some of the instances to stop before optimality was proven. This time limit also prevented us from trying larger instances than the J30 instance set, to prevent calculation on too large a portion of the instances being cut short. However, we are confident that the results approach optimality, for they were more flexible than POSes created by the other methods.

#### Δmakespan

In Figure 5.8, we compare flexibility and Δmakespan of the optimized POS to the POS created using maxCCminID. We used the J30 instance set, as mentioned, and the exp2 delay set.

A clear observation can be made that in the majority of cases, both the flexibility increased and Δmakespan decreased. However, there seems to be little correlation between the amount of flexibility increase and the amount of Δmakespan decrease. Additionally, there are also many cases where both flexibility and Δmakespan increased.

#### Number of Violations

In Figure 5.9, we compare flexibility and number of violations of the optimized POS to the POS created using maxCCminID. We used the J30 instance set, and the exp2 delay set.

It is quite clear that the optimization for flexibility leads, in most cases, to a reduction in the number of violations, when compared to POSes created using maxCCminID. However, the correlation between flexibility and the number of violations is not very strong — although stronger than that between flexibility and Δmakespan. Still, there are many cases where both flexibility and Δmakespan both increase.

#### Performance Ratios

As a side study, we also investigated the performance ratio of the heuristics with respect to flexibility. The performance ratios calculated for all instances of the J30 set are shown in a box plot in Figure 5.10. With this we assume that we created POSes with the highest possible flexibility, although this may be disputed for some instances due to the calculation not reaching proven optimality, as discussed earlier.

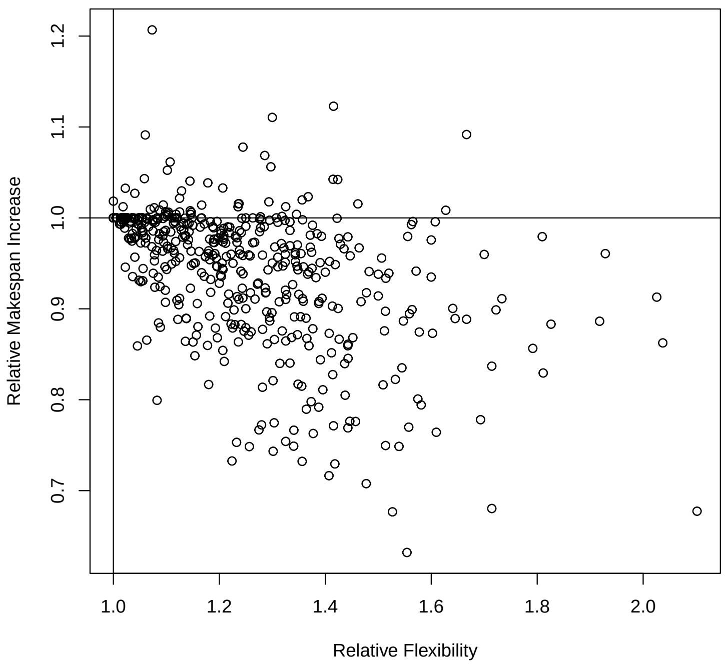


Figure 5.8: Flexibility and Normalized Makespan Increase of Optimized POS compared to maxCCminID

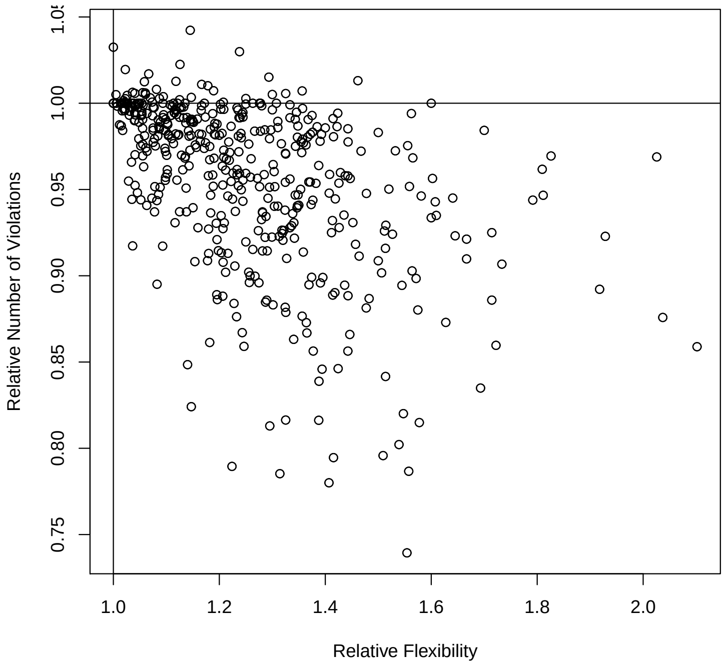


Figure 5.9: Flexibility and Number of Violations of Optimized POS compared to maxCC-minID

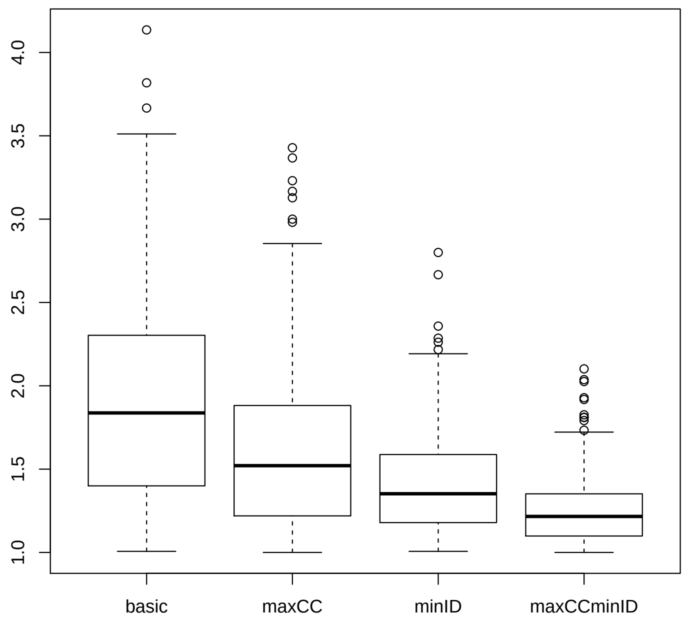


Figure 5.10: Performance Ratio of Chaining Methods with respect to Flexibility

With this plot, it is clear that maxCCminID performs much better in terms of flexibility than basic chaining. However, there is still room for improvement: on average, the flexibility can be improved by around twenty percent by assigning activities to different chains.

### 5.2.5 Discussion of Observations

From the correlations found between flexibility and the two robustness measures, we can conclude that it is reasonable to assume a connection between flexibility and robustness. However, there are many examples where increasing flexibility does not increase robustness. Therefore we can conclude that other factors are in play.

An interesting observation was the presence of the hyperbola in Figure 5.4. We suspect this is caused by the average concurrency, or the average number of concurrent activities or paths. The reasoning is as follows. We measure Δmakespan as the difference between the original makespan and the makespan under execution, divided by the horizon for normalization — not by the original makespan. This way, every delay is counted equally, regardless of the quality of the original schedule.

When two activities take longer than expected and they are on the same path, the delays add up. However, if they are concurrent, the delays do not add up; instead the delay of a common successor is determined by the largest delay of the two. Since the probability of two randomly selected activities to be concurrent is equal to $\frac{1}{concurrency}$, on average only the delays of $\frac{1}{concurrency}$ of the delays add up. We have observed an almost linear relation between flexibility and the average concurrency, calculated using $concurrency = \frac{horizon}{makespan}$. However, we could not find a mechanism to explain this linear relationship.

## 5.3 Slack

Another important metric that is associated with robustness is slack. It is defined for an activity as the amount of time by which it can be postponed without disturbing other activities. For a schedule, it is the summed slack of all activities. For a partial order schedule, it is the slack of the earliest start time schedule.

In our study, we used the metrics described in Chtourou and Haouari (2008). These metrics are based on one of three ways of measuring slack, weighted using one of four weightings. The metrics go unnamed in the original paper, but we name them for ease of understanding.

### 5.3.1 Metric details

These metrics are already described in Section 3.2, but for the interest of clarity we elaborate a bit on that description here. The three base metrics we called slack, binary slack, and capped slack. The first is simply the time available for an activity to be postponed without any other activities to be affected. The second is a binary variable having value 1 when the slack is greater than zero, and 0 otherwise. Finally, capped slack is the minimum of the slack and a fraction of the activity's processing time. In the original paper, this fraction is set to the expected delay. However, in order to reduce complexity and keep the research generic, we decided to fix this factor to 1.

The four weightings are as follows: unweighted, weighted by number of successors, weighted by resource usage, and weighted by number of successors and resource usage. The first is simple: all activities are weighted equally. The second weighting is also simple: the more successors an activity has, the more the slack counts towards the total of the schedule. The third weighting uses the summed resource usage as a measure: the more resources an activity requires, the more the slack counts towards the total of the schedule. In the final weighting, the last two are combined: the slack is counted once for each successor for each resource unit — in other words, the number of successors is multiplied by the number of resource units, and that number is used to weight the activity.

### 5.3.2 Correlation Test

The first test we perform is a correlation test. In this test we calculate the correlation between the different slack metrics and the two robustness metrics, for each combination of instance set and delay set.

In Appendix C we listed, among others, tables with these correlation values. In these tables we list the correlation between all slack metrics and one of the robustness metrics, for all of the instance sets and one of the delay sets. For example, Table C.3 shows the correlation between the different slack metrics and the number of violations for all instance sets, for the exp2 delay set.

These tables contain a vast amount of data: eight tables, each containing 48 numbers for different correlations, coming to a total of 384 values. We can, however, summarize the data per topic, by looking for patterns.

### Pattern in Weighting by Resources

The most notable pattern is that the slack metrics using the first two weightings, unweighted and weighted by number of successors, have a medium to strong negative correlation with both the number of violations and Δmakespan. This contrasts with metrics based on the other two weightings, which assigns weights to activities according to resource usage, or resource usage and the number of successors. Metrics bases on these weightings are at best weakly negatively correlated with the number of violations or Δmakespan, at worst they are strongly positively correlated with the robustness measures. Since the number of violations and Δmakespan actually measure the absence of robustness — as robustness increases, these metric decrease — this strongly suggests that slack metrics based on robustness usage cannot be used for this purpose.

One possible explanation for the bad performance of the weighting by resource usage, is that the weights may be much larger than the slack in the system. This causes the metrics to measure something that more closely resembles resource usage than slack — which might also explain the positive correlations. However, we can see in Tables C.6 and C.10 that, when using the gauss_1_02 delay set, the unweighted metrics are more strongly correlated with robustness than the metrics weighted by number of successors.

### Patterns in Slack Variants

In the remaining half of the data, it is much harder to find patterns. When ranking the metrics on correlation strength, some patterns begin to emerge. In Table 5.1 we show for each metric in how many parameter combinations it was the metric with the strongest correlation with the robustness metrics, and in how many it was the metric with the weakest correlation.

| | Number of Violations | | Δmakespan | |
|---|---|---|---|---|
| | Strongest | Weakest | Strongest | Weakest |
| Unweighted Slack | 6 | 1 | 6 | 2 |
| Weighted Slack | 4 | 0 | 2 | 0 |
| Unweighted Binary Slack | 1 | 2 | 0 | 2 |
| Weighted Binary Slack | 1 | 13 | 4 | 11 |
| Unweighted Capped Slack | 2 | 0 | 3 | 1 |
| Weighted Capped Slack | 2 | 0 | 2 | 0 |

Table 5.1: Number of times a metric is the most strongly correlated and most weakly correlated metric, per robustness metric

From this table, we can make some interesting observations. Looking at the "Δmakespan" column, we see that Weighted Binary Slack is the metric with the strongest correlation in 4 of the design points, and the metric with the weakest correlation in nearly all of the other design points. Investigating the correlation tables, we can see in Table C.8 that Weighted Binary Slack is the metric with the strongest correlation with Δmakespan in the four parameter combinations that contain the fixed_50_30 delay set. Furthermore, Table C.7 shows us the only other case when it is *not* the metric with the weakest correlation with Δmakespan: using the J60 instance set and the exp2 delay set — and in this case, it is the metric with the second weakest correlation.

Interestingly enough, Table C.4 shows that weighted binary slack is only weakly correlated with the number of violations when using the fixed_50_30 delay set — except when using the J30 instance set. This is in contrast with the metric's relatively strong correlation with with Δmakespan, as discussed earlier.

When testing using another delay set than the fixed_50_30 set, the unweighted and weighted slack tend to be more strongly correlated with robustness, but this varies with testing conditions. It is therefore hard to find explicit other patterns.

### 5.3.3 Conclusions

Slack is correlated to robustness, although this correlation is not as strong as the correlation between flexibility and robustness. Furthermore, depending on the delay pattern, it is either the total slack that has a stronger correlation to flexibility or the number of activities with slack. This shows us that robustness is not a fixed property of a schedule, but that it might depend on — or even be tailored for — the kind of delays.

# Chapter 6

# Robustness Model

Having found these correlations in the exploratory study, we now want to understand why these correlations exist, and how we can use this to our advantage. In other words: how does chaining produce robust partial order schedules, and how can we exploit this?

In this chapter, we try to answer this question by proposing three models. The first model, the delay propagation model, describes how a single activity is affected by disturbances. The second model, which we call the robustness model, uses the delay propagation model to describe what makes a partial order schedule robust. The third and final model, called the chain selection model, uses the robustness model to describe the strategy that heuristics should use in order to create robust partial order schedules.

## 6.1 Delay Propagation Model

We introduce the Delay Propagation as a basic model of how delays of activities affect other activities, and how the delays of multiple activities interact. We describe this model using two example scenarios, with respectively one and two delayed activities, and then extending this scenario to the general case.

In each scenario, we start with a Partial Order Schedule $\langle A, P \cup U, d \rangle$ and its earliest start time schedule $s$. We then introduce one or more delays by changing $d$ to $d'$, which changes the earliest start time schedule to $s'$. This new schedule $s'$ will be described in terms of $s$, to get an intuitive understanding of the changes caused by the delays.

### 6.1.1 Single Delay

We first examine the scenario where only one activity is delayed. Let this activity be $x$, so $d'_x > d_x$ and $\forall a \in A \setminus \{x\} : d'_a = d_a$. Let this increase be $\Delta = d'_x - d_x$. Since the start time of $x$ is still the same, the end time of this activity is now postponed by $\Delta$.

Assuming all successors of $x$ are scheduled immediately after $x$, this means that their start time has to be postponed by $\Delta$ as well to satisfy all precedence constraints. Since their start time is postponed by $\Delta$, their end time is also postponed by $\Delta$. Once again assuming no time between the end time of one activity and the start time of its successor, the successors' successors have their start times postponed, and so on. This means that, in order to satisfy all precedence constraints, we must construct $s'$ as follows:

$$s'(a) = \begin{cases} s(a) & : a \notin succ^*(x) \\ s(a) + \Delta & : a \in succ^*(x) \end{cases} \tag{6.1}$$

Notice that each activity passes this delay on to its successors. We call this delay propagation:

**Definition 18.** Delay Propagation is the process whereby the delay of an activity causes successors of that activity to be delayed, which in turn can cause their successors to be delayed, and so on.

Of course, the assumption that the successors of $x$ immediately follow $x$ does not always hold. Suppose $y$ is one of the successors of $x$. There might be some time between the end time of $x$ and the start time of $y$, let this be the margin $m_{x,y}$. Then, when $x$ is delayed by $\Delta$, the start time of $y$ only needs to be postponed by $\max(\Delta - m_{x,y}, 0)$ in order to satisfy the precedence constraint. If $z$ is one of the successors of $y$, and there is a margin $m_{y,z}$ between $y$ and $z$, then $z$ needs to be postponed by $\max(\Delta - m_{x,y} - m_{y,z}, 0)$ in order to satisfy the precedence constraints.

It gets more complicated when there are not one, but two concurrent activities between $x$ to $z$: the delay of both of these could affect $z$, depending on the margins. Suppose there is also an activity $w$, which is a successor of $x$ and a predecessor of $z$, which has margin $m_{x,w}$ between $x$ and $w$ and margin $m_{w,z}$ between $w$ and $z$. The end time of activity $w$ is now postponed by $\max(\Delta - m_{x,w}, 0)$.

Since the start time of activity $c$ needs to be later than both $y$ and $w$, its start time needs to be postponed by $\max(\Delta - m_{x,y} - m_{y,z}, \Delta - m_{x,w}, m_{w,z}, 0)$. If we rewrite this to $\max(\Delta - \min(m_{x,y} + m_{y,z}, m_{x,w} + m_{w,z}), 0)$, it shows that this reduction in delay is actually equal to the pairwise float. This measure, as previously described in Section 3.2.2, measure the amount of float available between two activities: the amount of time the one activity can be postponed without affecting the other activity. We can therefore change our construction of $s'$ to be:

$$s'(a) = \begin{cases} s(a) & : a \notin succ^*(x) \\ s(a) + \max(\Delta - PF_{x,a}, 0) & : a \in succ^*(x) \end{cases} \tag{6.2}$$

### 6.1.2 Two Delays

Next, we examine the scenario where two activities are delayed, to know how they interact. Suppose two activities $x, y \in A$ are delayed by respectively $\Delta_x$ and $\Delta_y$. The effects on successors of $x$ and $y$ depend on whether the activities are independent or not.

Suppose that $x$ and $y$ are independent, so $x \notin succ^*(y) \wedge y \notin succ^*(x)$. The activities do not affect each other, but common successors can be affected by both. Let $z \in succ^*(x) \cap succ^*(y)$ be one such common successor. This activity can be either postponed as a result of $x$ being delayed, or as a result of $y$ being delayed — whichever affects it the most. The start time of $z$ therefore has to be postponed by $\max(\Delta_x - PF_{x,z}, \Delta_y - PF_{y,z}, 0)$. To summarize this in our construction of $s'$:

$$s'(a) = \begin{cases} s(a) & : a \notin succ^*(x) \cup succ^*(y) \\ s(a) + \max(\Delta_x - PF_{x,a}, 0) & : a \in succ^*(x) \setminus succ^*(y) \\ s(a) + \max(\Delta_y - PF_{y,a}, 0) & : a \in succ^*(y) \setminus succ^*(x) \\ s(a) + \max(\Delta_x - PF_{x,a}, \Delta_y - PF_{y,a}, 0) & : a \in succ^*(x) \cap succ^*(y) \end{cases} \tag{6.3}$$

Suppose that $x$ and $y$ are not independent and $y \in succ^*(x)$. In that case, the start time of $y$ is already postponed by $\max(\Delta_x - PF_{x,y}, 0)$. Adding the delay, we have to postpone the end time of $b$ by $\max(\Delta_x - PF_{x,y}, 0) + \Delta_y$. Any successors of $y$ are automatically successors of $x$, so $succ^*(y) \subset succ^*(x)$. The set of common successors is therefore equal to $succ^*(x) \cap succ^*(y) = succ^*(y)$. Combining this increased delay of $y$ with the previous results, we can now construct $s'$ as follows:

$$s'(a) = \begin{cases} s(a) & : a \notin succ^*(x) \cup succ^*(y) \\ s(a) + \max(\Delta_x - PF_{x,a}, 0) & : a \in succ^*(x) \setminus succ^*(y) \\ s(a) + \max(\Delta_x - PF_{x,a}, \max(\Delta_x - PF_{x,y}, 0) + \Delta_y - PF_{y,a}, 0) & : a \in succ^*(y) \end{cases} \tag{6.4}$$

Note that the delay of $y$ might not be the factor increasing the start time of an activity $a \in succ^*(y)$: it is still possible that there is another path between $x$ and $y$ with less pairwise float, and therefore affecting $a$ more.

### 6.1.3 Multiple Delays

This model can be extended to the general case, with $d'_a \geq d_a$ for every $a \in A$. Using the definition of a valid schedule, we can simply define $s'$ as follows:

$$s'(a) = \begin{cases} 0 & : |pred(a)| = 0 \\ \max_{p \in pred(a)}(s'(p) + d'_p) & : \text{otherwise} \end{cases} \tag{6.5}$$

In order to increase the understanding of the impact of all delays, we can also define $s'$ in terms of the $s$ and a function of the introduced delays. First of all, we define the set of delayed activities:

$$delayed = \{a \in A \mid d'_a > d_a\} \tag{6.6}$$

Now we can use this to define the start delay of an activity as a function of all delayed predecessors:

$$SD_a = \max(\max_{x \in pred^*(a) \cap delayed}(SD_x + (d'_x - d_x) - PF_{x,a}), 0) \tag{6.7}$$

We can now construct $s'$ as a function of $s$ and the start delay, as simple as $s'(a) = s(a) + SD_a$.

## 6.2 Robustness Model

Robustness is, as defined in Section 2.2, the ability of an object to retain or regain its properties during or following adverse conditions. For Partial Order Schedules, we then defined that more robust schedules are schedules that are more stable and have less makespan increase under the same adverse conditions.

Using Delay Propagation to describe the underlying mechanism, we now propose a model that describes how properties of a partial order schedule influence its robustness. We will also use this to explain the observations.

### 6.2.1 Size of *pred*$^*$

From the Delay Propagation model, we can see that the start delay of an activity $a$ depends on the elements of $pred^*(a) \cap delayed$. If this set is empty, the start of $a$ is not delayed. Any activity that is added to this set might delay the start of $a$, or increase it if it already is delayed, depending on the probability of different delay lengths.

If we could reduce the size of $pred^*(a) \cap delayed$, we reduce the expected value of the start delay, and maybe bring it to zero. Since we cannot predict anything on the set of delayed activities, we can only try to reduce the set of delayed predecessors of $a$ by reducing the size of $pred^*(a)$. Reducing the average size of $pred^*(a)$ reduces the expected start delay of all activities, which in turn increases stability (since fewer activities are expected to be delayed) and reduces Δmakespan, since delays have lower probability to add up.

#### Relationship to *flex*

Flexibility, when measured in *flex*, is equal to the fraction of unrelated pairs of activities. Rewriting from Equation 3.5, an activity is — on average — connected to $(1 - flex)(n-1)$ other activities. This means that the average value of $pred^*(a) + succ^*(a) = (1 - flex)(n - 1)$. Since every occurrence of activity $a$ in $succ^*(b)$ also means an occurrence of $b$ in $pred^*(a)$, we can state that on average $pred^*(a) = \frac{1}{2}(1 - flex)(n-1)$. This explains the strong correlation between flexibility and robustness.

### 6.2.2 Pairwise Float

As we can observe in equation 6.7, pairwise float has a dampening effect on delays. This leads us to conclude that an increase in pairwise float will, in general, have a positive effect on robustness.

However, this is limited by the objective of minimizing makespan in the fixed-time schedule. After all, reducing the makespan also reduces the average time between activities.

#### Relationship to slack

Slack is defined as the amount of time an activity can be postponed without disturbing any of its successors. This is the same as the minimum of the pairwise floats between an activity and its successors. Multiplying the slack of an activity by the number of successors of that activity therefore provides a lower bound on the pairwise float between that activity and its successors. This relationship is a possible explanation for the observations in Section 5.3.

**Relationship to $flex$**

From Equation 6.7 we can derive that the effect of the delay of activity $x \in pred^*(a)$ on $a$ is completely negated if $PF_{x,a} \geq SD_x + (d'_x - d_x)$. This is effectively the same as $x \notin pred^*(a)$: in neither case, $SD_a$ is increased as an effect of the delay of $x$. A large enough pairwise float might therefore be used to simulate the effects of *flex*.

This is what is used in Braeckmans et al. (2005) to incorporate flexibility into their pairwise float heuristic: by defining the pairwise float of an unrelated pair of activities as a large value. However, the exact value of $d'_x$ is only known during execution. It is therefore impossible to know this without making any assumptions on the probability distribution of $d'_x$.

### 6.2.3 Guidelines for Increasing Robustness

Using the observations and relations, we can propose guidelines on how the metrics should be influenced for increased robustness.

We can first summarize the observed relations. Flexibility can be increased to decrease the number of violations and $\Delta$makespan. Pairwise float can be increased to decrease $\Delta$makespan, and — if large enough — to decrease the number of violations. However, this only happens when $PF_{x,a} \geq SD_x + (d'_x - d_x)$, which is the same condition under which pairwise float and flexibility can be exchanged.

Therefore, we propose the following guidelines:

- Both flexibility and pairwise float should be increased, if possible.
- Depending on the expected delays, increasing flexibility should get priority over increasing pairwise float. Only when an increase in pairwise float is sufficient to negate delays, should pairwise float be given priority over flexibility. In practice, where an optimized fixed-time schedule limits the available time, pairwise float should only be prioritized when delays are small and few.

## 6.3 Chain Selection Model

In this section we will discuss the influence of chain selection on robustness. Since we learned that flexibility should, in general, be prioritized over pairwise float, we currently focus only on increasing flexibility.

In order to increase flexibility, we need to decrease the number of constraints under transitive closure. This is an important distinction from reducing the number of constraints: it might be that replacing two constrains by another constraint actually increases the number of constraints under transitive closure.

Chaining is a greedy procedure which traverses the activities in the order of their start time. This results in a fast and simple procedure, but prevents us from using more powerful techniques such as backtracking (which could become very computationally expensive). Trying to increase the number of of constraints under transitive closure therefore becomes a balancing act between minimizing the number of constraints for the current activity and its expected impact on further activities.

### 6.3.1 Local Optimization

First we look at local optimization of flexibility, by which we mean reducing $pred^*(a)$ for a single activity $a$. For this we assume that all activities that are chained before $a$ are already fixed and we ignore activities to be scheduled after $a$. This means that $a$ needs to be assigned to chains such that the lowest number of activities is added to $pred^*(a)$. In other words: if $last(x)$ is the last activity on chain $x$, select a set of chains $X$ to minimize the following equation:

$$(\{last(x) \mid x \in X\} \cup \bigcup_{x \in X} pred^*(last(x))) \setminus pred^*(a) \tag{6.8}$$

There is a technique to reduce the problem size, by choosing a chain $x$ such that $(\{last(x)\} \cup pred^* x \in X) \setminus pred^*(a) = \varnothing$ — in other words, $x \in pred^*(a)$. This is exactly what is described by the minID heuristic: if possible, chains are selected such that the last activity on the chain is already a predecessor of $a$. This explains the improved flexibility of minID compared to basic chaining, and of maxCCminID compared to maxCC.

This technique might be sufficient to completely solve the instance and assign $a$ to chains without increasing the number of predecessors. When this is not the case, we can use heuristics to assign chains with a small number of added predecessors. However, this might not be an optimal solution.

One heuristic, maxCC is very simple: if the previously selected chain is called $x$, the next chain $y$ will be selected such that $last(x) = last(y)$ — if that is possible. This way, as many chains with the same last activity as $x$ are added, effectively adding only $pred^*(x)$ to the set of predecessors. Compared to basic chaining, where any chain is picked from the set of available chains, this decreases the set of $pred^*(x)$, resulting in an increased flexibility. This explains the differences in flexibility between basic chaining and maxCC, and the differences in flexibility between minID and maxCCminID.

The selection of the first chain in maxCC is not specified; this is actually done using other heuristics. In maxCCminID this is done using minID, if possible, but otherwise it is selected randomly. This first activity may be selected in a smarter way, using other heuristics.

### 6.3.2 Impact Reduction

When an assignment $a$ is assigned to a set of chains, the choice of chains can influence the choice of chains of activities scheduled later. After all, assigning activity $a$ to a chain $x$ makes that chain unavailable for activities that are scheduled concurrently with $a$ and changes the implications if a later activity is also assigned to $x$. We therefore aim for impact reduction: assigning $a$ to a set of chains such that later activities can be optimized further.

However, we want to keep this goal from reducing the local solution quality (i.e. increasing the size of $pred^*(a)$).

First, we consider the chains that $a$ is assigned to, and what this assignment means for these chains. Any activities that are assigned to any of the chains that $a$ is assigned to, get $a \cup pred^*(a)$ added to their set of predecessors. This is the only thing that can be influenced by chaining choice — end time and number of chains are, after all, determined by the fixed time schedule and the problem instance, respectively. This is another reason not to increase $pred^*(a)$ when trying to reduce the impact the chaining choice on other activities.

Next, we consider the chains that $a$ is not assigned to. These are the chains that activities that are scheduled concurrently with $a$ can be assigned to. Let $b$ be such an activity.

When we assign $b$ to a set of chains, we want to do so while adding the lowest number of predecessors to $pred^*(b)$. However, we do not know $pred^*(b)$ — remember that we are reasoning about the future event of assigning $b$ to a set of chains, while in the process of assigning $a$ to chains. We therefore use heuristics to reduce the expected increase in $pred^*(b)$.

First of all, we would like as many chains as possible to be already in $pred^*(b)$, so we can assign $b$ to these chains "for free". We argue that the probability of a non-chosen chain to be in $pred^*(b)$ can be improved by not choosing activities with the most successors. In other words: if there is no difference between two chains for the size of $pred^*(a)$, we choose the chain with the least successors which are not yet assigned to chains.

Secondly, for the other chains, we want to reduce the expected number of activities added to $pred^*(b)$. Again, we don't know which activities are in $pred^*(b)$ so which activities actually contribute most to this. We argue that it is therefore best for $b$ to be able to choose from sets of activities with the same last activity that are as large as possible.

The first strategy to do this, is to increase the average number of available chains per activity. We can do this by using all chains from a single activity, for example using maxCC. In order for this to work, we need to assign $a$ to chains such that the last activity of these chains has fewer available chains than $a$ needs — although this might be impossible without increasing the number of predecessors of $a$.

Another strategy is to try to increase the number of chains available with one activity as last activity at the expense of the number of chains available with another activity as last activity. This increases the number of large sets of chains with the same last activity, at the cost of also creating some small sets of chains with the same activity. The idea is that this way, the probability of $b$ needing only a few activities is reduced. We also hope that the small sets of chains with the same last activity are not needed, or needed at a later moment.

# Chapter 7

# Model Testing

The models proposed in the previous chapter are merely that: proposals. Although the models provide an explanation for our observations so far, we want to use the models for further predictions. In this chapter, we will investigate the validity of our models by deriving predictions from them and testing these.

## 7.1 Prediction and Experimental Design

We evaluate our model by deriving a new heuristic aimed at increasing robustness and measuring if it actually does. We use the chaining model to create this new heuristic, but because it is based on the robustness model we test that model in the same time.

### 7.1.1 MaxChains Heuristic

In Section 6.3 we discussed how flexibility might be increased by decreasing the number of added predecessors as a result of adding an activity to a chain. We introduce a simple heuristic, maxChains, that aims to do this indirectly: by choosing a chain such that its latest activity is also the latest activity of the highest number of other chains, the number of added direct predecessors is reduced.

By combining this with maxCCminID, we get a metric we call maxCCminIDmax-Chains. We can describe it as follows:

- Select a chain $x$ such that $last(x) = last(prev)$, where $prev$ is the previously selected chain (maxCC).
- If that is not possible, select a chain such that $last(x) \in pred^*(a)$ (minID).
- If that is not possible, determine for each activity in how many chains it is the latest activity. Of these activities, select the activities with the most chains of which it is the latest. Select a chain $x$ such that $last(x)$ is one of these activities (maxChains).

### 7.1.2 Expected Results

This heuristics minimizes, for each activity individually, the number of direct predecessors for that activity. Since each direct predecessors might also add a number of indirect predecessors, we suspect that, in general, this also decreases the number of indirect predecessors for that activity. This effect should increase flexibility.

However, the choice for the activity with the most available chains might be disadvantageous locally, when it is also the activity with the most added predecessors. Other activities might be the latest activity on fewer chains (and providing fewer resources), but if they contribute fewer predecessors it might be better to choose those instead. Furthermore, choosing an activity with more chains than necessary prohibits later activities, which might require more chains, from selecting all these chains. This might result in this later activity to add two precedence constraints, while the current activity might be satisfied with only one precedence constraint when choosing a different activity. These two phenomena have a negative effect on flexibility

We suspect that the influence of these effects depend per instance, and that in some cases it will produce a more flexible POS compared to maxCCminID, and in some cases it will produce a less flexible POS. Since the positive effect happens in all cases but the negative effects only occur in certain conditions, we suspect that flexibility will increase in more cases than it will decrease.

**Prediction 1.** Partial Order Schedules created using maxCCminIDmaxChains are more flexible than Partial Order Schedules created using maxCCminID for the same instance and using the same fixed-time schedule.

Following the model of Section 6.2, we predict that the change in flexibility will be roughly comparable to the change in robustness.

**Prediction 2.** Partial Order Schedules created using maxCCminIDmaxChains are more robust than Partial Order Schedules created using maxCCminID for the same instance and using the same fixed-time schedule, and this change in robustness is similar to the change in flexibility.

## 7.2 Measured Results

Using our testing environment as described in Section 4.2, we performed several simulations with partial order schedules created using maxCCminID and maxCCminIDmaxChains.

### 7.2.1 Flexibility

We first measured the difference in flexibility between partial order schedules created using the two different heuristics. In Figure 7.1, we plot the flexibility of POSes created using this heuristic, relative to the flexibility of POSes created using the maxCCminID heuristic. The part of the box plot above the dashed line at 1.0 represent instances where the flexibility of the POS created using maxCCminIDmaxChains was more than the flexibility of the POS created using maxCCminID.

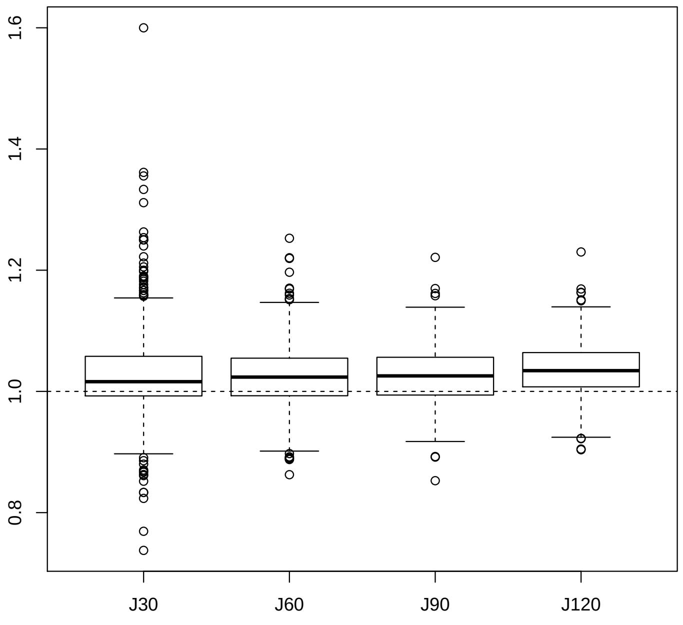


Figure 7.1: Flexibility of POSes generated using maxCCminIDmaxChains relative to flexibility of POSes generated using maxCCminID for the same instance, for different instance sets.

We can clearly see that, for all instance sets, the majority of instances benefits from using maxCCminIDmaxChains instead of maxCCminID, in terms of flexibility. It hardly needs stating that the increase in flexibility proved significant, using the Wilcoxon signed-rank test, with a p-value orders of magnitude smaller than 0.001. The flexibility increased 2.8%, 2.5%, 2.6%, and 3.5% for the J30, J60, J90, and J120 instance sets, respectively. Given that, for the J30 instance set, the optimized POSes have 25% more flexibility than those created using maxCCminID, we can conclude that this simple heuristic bridged one tenth of the gap between maxCCminId and the optimum — quite a good first step.

**Observation 1.** Partial Order Schedules created using maxCCminIDmaxChains are, on average, more flexible than Partial Order Schedules created using maxCCminID for the same instance and using the same fixed-time schedule.

### 7.2.2 Robustness

However, our goal is not flexibility but robustness. We predicted that this increase in flexibility should also increase robustness — and therefore reduce the makespan increase and the number of delays. In order to measure whether this was changed significantly we measured the mean change and the probability of such a change for each combination of instance set and delay set.

The result is shown in Table 7.1. In this table, the values of the metrics of the POSes created using maxCCminIDmaxChains are compared to those of the POSes created using maxCCminID in the columns “Relative Value”. A value less than 1 shows that the metric is decreased, a value greater than 1 that the metric increased. In the column “p-value”, we show the probability that the values of the metrics of the POSes created using the different heuristics actually have the same distribution, calculated using the Wilcoxon signed-rank test. The lower the value, the more significant the difference is, with 0.05 being an often-used significance threshold: if the value is lower than this value, the difference is considered significant. Due to rounding, values less than $5 \times 10^{-6}$ are rounded to 0.

| | | Δmakespan | | Number of Violations | |
|---|---|---|---|---|---|
| Instance Set | Delay Set | Relative Value | p-value | Relative Value | p-value |
| J30 | exp2 | 0.994 | 0.00045 | 0.996 | 6e-05 |
| | fixed_50_30 | 0.998 | 0.0302 | 0.999 | 0.0437 |
| | unif_80_5 | 0.994 | 1e-05 | 0.996 | 0 |
| | gauss_1_02 | 0.99 | 0.00617 | 0.994 | 7e-04 |
| J60 | exp2 | 1.005 | 0.94547 | 0.998 | 0.03948 |
| | fixed_50_30 | 1.004 | 0.34849 | 0.999 | 0.04839 |
| | unif_80_5 | 0.998 | 0.00538 | 0.998 | 0 |
| | gauss_1_02 | 1.034 | 0.58681 | 0.998 | 0.15701 |
| J90 | exp2 | 1.012 | 0.00836 | 1.001 | 0.82891 |
| | fixed_50_30 | 1.008 | 0.00025 | 1 | 0.54755 |
| | unif_80_5 | 1.001 | 0.96367 | 0.999 | 0.0014 |
| | gauss_1_02 | 0.979 | 0.00012 | 1.005 | 0.02206 |
| J120 | exp2 | 1.001 | 0.13056 | 0.999 | 0.00101 |
| | fixed_50_30 | 1.003 | 0.62097 | 1 | 0.12493 |
| | unif_80_5 | 0.996 | 0 | 0.999 | 0 |
| | gauss_1_02 | 1.012 | 0.9693 | 1 | 0.33298 |

Table 7.1: Change in Δmakespan and number of violations, and probability that a change of this magnitude is the product of chance

Unfortunately, the results in this table are not as positive as we expected. Of the sixteen combinations of instance set and delay set, we found six where both Δmakespan and the number of violations were significantly decreased with a p-value of $p < 0.05$. In five more cases we can observe that one of the metrics was decreased significantly, but the other was not. That means that in five cases, there was no significant decrease in either Δmakespan or the number of violations, or even an increase of one or both of these metrics. We can therefore not conclude that the maxCCminIDmaxChains metrics produces more robust partial order schedules than maxCCminID.

**Observation 2.** Partial Order Schedules created using maxCCminIDmaxChains are not significantly more robust than Partial Order Schedules created using maxCCminID for the same instance and using the same fixed-time schedule.

## 7.3 Results Discussion

Our first prediction, Prediction 1, is matched by Observation 1. This strengthens our belief in the model as described in Section 6.3.

However, Prediction 2, is refuted by Observation 2. This means that there is an error in either the model, the prediction, or the experiment. Reasoning brings us to the following three possible errors:

1. The prediction is wrong, and differences in pairwise float are large enough to negate the effect of increased flexibility. This might be the case if the heuristic selects chains with a later end time, which is not unlikely since the chains with later activities are less likely to be selected by previous activities — simply because there are fewer activities between that activity and the current activity. If this is the case, there should be a negative correlation between flexibility and pairwise float.

2. The test is not accurate enough, due to too little variation in delay patterns or too few delay patterns. We deem this not very likely, since our delay sets seemed accurate enough for our previous experiments. However, if this is the case, it would be remedied by using a delay set with more delay patterns.

3. The model as described in Section 6.2 is incomplete or (partially) incorrect.

### 7.3.1 Investigation of Pairwise Float

First, we investigated the first option. Tests investigating a correlation between flexibility and the connected total float (which is the sum of pairwise float of all connected pairs of activities) showed a positive correlation. In other words: more flexibility is coupled with more, not less, pairwise float. We can therefore dismiss this option.

### 7.3.2 Investigation of Delay Set Accuracy

To investigate the second option, we created two delay sets containing 1000 instead of 100 delay patterns, one with the unif_80_5 distribution and one with the fixed_50_30 distribution. Tested using these sets instead of the smaller delay sets, we found that the relative change in $\Delta$makespan or number of violations does not change, but that the p-value is reduced. This indicates that the use of larger delay sets reduces the noise from the delay sets, causing the reduced p-value, but averaged over all POSes this does not change the measured robustness. We therefore dismiss this option as probable cause for the bad results. Furthermore, we see this as validation that the delay sets using 100 delay patters are sufficient for (the majority of) our research.

### 7.3.3 Investigation of Model Incompleteness

The final option is that the Robustness Model described in Section 6.2 is incorrect or incomplete. We assume that the Delay Propagation Model, described in Section 6.1, is correct and complete, because it is derived from the definition of Partial Order Schedules and Fixed Time Schedules. We believe that the cause of Prediction 2 being refuted is not in the Chain Selection Model, described in 6.3, since that model does not describe the relation between flexibility and robustness.

Having eliminated all other plausible causes for the refutation of Prediction 2, we must conclude that the Robustness Model must be incomplete or incorrect. In this model, we identified the size of $pred*$, later substituted by flexibility, and pairwise float as factors that influence robustness. However, we suspect that there is another factor which affects robustness, which is not included in our model. This way, the effect of a small increase in flexibility can be overshadowed by the effects of this other factor, making it undetectable.

Moreover, our model only looks at properties of individual activities, while in the Partial Order Schedule they are connected to form a network. We suspect that the overlooked factor can be found in the structure of this network, rather than properties of single activities. This is an effect that has been demonstrated before: that the behaviour of a network is more complicated than the sum of its parts.

# Chapter 8

# Discussion

In this chapter we will briefly summarize our contributions and discuss them, including the research domain and threats to validity. Using this as a starting point, we will describe topics for future research.

## 8.1 Contributions

We have made several contributions on several aspects of our research topic. Using a testing framework, which we also made available for public use, we tested some assumptions that were made in related work. But, maybe more valuable, we proposed several models on the workings of chaining and robustness.

### 8.1.1 Findings

We found that there is a strong correlation between flexibility and robustness in Partial Order Schedules. This has long been believed, but has to our knowledge not yet been investigated thoroughly. This finding is only valid for the tested delay patterns and instances and under the assumption that robustness can be measured using makespan increase and the number of violations. However, we find our choices in delay patterns, instances, and assumptions sufficiently defendable and general to believe that they can be used outside of this scope.

Furthermore, we found that slack-based metrics are weakly correlated with robustness in Partial Order Schedules. This somewhat contradicts previous beliefs, which uses slack-based metrics as a substitute for robustness — although that was used to measure robustness in Fixed-Time Schedule instead of Partial Order Schedules.

### 8.1.2 Model Proposals

We proposed, first of all, the delay propagation model. This model shows how an activity is affected by the delays in the system. Since it is derived solely from the definition of Partial Order Schedules and Earliest Start Time Schedules, it is not bound to any assumptions we made in our experiments.

Based on that, we proposed a robustness model. This model explains most of our observations, although it proved to be insufficient to explain Observation 2. We suspect that it might still be usable when fine precision is not necessary, or to serve as basis for a better model.

Finally, we proposed a model that describes how heuristics for the chaining method could be adapted to improve flexibility. This model has had little testing, but so far fits all observations. If it does hold up to more rigorous testing, our hope is that it could be adapted when a better robustness model is found in order to help create better chaining heuristics.

### 8.1.3 Methodology

We demonstrated that models about algorithm behaviour can be found using empirical evaluation. This is not a novel idea, but it not frequently used in algorithms research: most research happens non-empirically, and most empirical research focuses on performance rather than understanding.

In order to aid future researchers in investigating the Resource-Constrained Project Scheduling Problem, we made our testing platform publicly available through GitHub. This also aids in the reproducibility of our work: everyone can use our code to perform the same experiments. Furthermore, the code has a modular structure, making it easier to only change certain parts of the process, making it a good starting point for future research.

## 8.2 Future Work

In writing a thesis, there is only so much time available for research. In this section we list the research we would like to do, given enough time and resources[1]. On one hand we are interested in broadening the scope of our research and to investigate the applicability in real-life situations. On the other hand we would like to continue refining the models we currently have, and work from them to create a deeper understanding of the algorithm. Finally, we would like to see the (possibly refined) models being used to create new algorithms which produce more robust partial order schedules.

### 8.2.1 Scope of the Research

The scope of our research is limited to the delay patterns and instances discussed in Section 4.2. We encourage future researchers to investigate good delay patterns and instances, e.g. delay patterns and instances that match real-world conditions, and investigate the applicability of our research to those delay patterns and instances.

Furthermore, we have selected makespan increase and number of violations as metrics to measure robustness. In certain use cases, other metrics may be more applicable. We would be happy to see whether our findings on robustness still hold when other metrics are used to measure robustness.

[1] no pun intended

### 8.2.2 Testing and Refining the Models

The three models we proposed would benefit from testing and refinement. A thorough evaluation using more tests would benefit all three of them, but especially our robustness model could be refined and expanded to better predict the robustness of schedules. Since the chain selection model is based on the robustness model, a new version of this would have to be developed to suit an improved robustness model.

We suggest that future research also uses the delay propagation model in order to inspect robustness of a partial order schedule, rather than simulation. This would entail assuming a probability distribution for delays and calculating probability distributions for the delays of activities based on these. We suspect that, in combination with visualization, this could be a very effective tool for investigating how delays interact on the level of the POS instead of the level of the activity.

### 8.2.3 Model Application for Algorithm Design

If the proposed models, or future refined versions, are deemed valuable enough, we hope they can be used to guide algorithm design. This includes the proposal of new heuristics.

Based on our chain selection model for heuristics to increase flexibility, as described in Section 6.3, we also designed some heuristics that we did not have time to implement and evaluate, but may be of help for future research.

- Select an activity to be the predecessor based on the number of available chains per added predecessor.
- Instead of choosing a chain such that its latest activity has the most available chains, select a chain such that its latest activity has sufficient available chains.
- Choose a chain such that the last activity on that chain has the fewest successors. This is inspired by MABO, as discussed in Section 3.3.4.

# Bibliography

Aloulou, M. and M. Portmann (2003). “An efficient proactive reactive scheduling approach to hedge against shop floor disturbances”. In: *Proceedings of the 1st Multidisciplinary International Conference on Scheduling: Theory and Applications, MISTA*, pp. 336–362 (cit. on p. 16).

Braeckmans, K. et al. (2005). “Proactive resource allocation heuristics for robust project scheduling”. In: pp. 1–22 (cit. on pp. 7, 17, 19, 20, 22, 35, 45).

Cesta, A., A. Oddi and S. F. Smith (1998). “Profile-Based Algorithms to Solve Multiple Scheduling Problems Capacitated Metric”. In: (cit. on pp. 7, 16).

Chtourou, H. and M. Haouari (2008). “A two-stage-priority-rule-based algorithm for robust resource-constrained project scheduling”. In: *Computers & Industrial Engineering* 55.1, pp. 183–194. DOI: 10.1016/j.cie.2007.11.017 (cit. on pp. 14, 25, 38).

Hartmann, S. and R. Kolisch (2000). “Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem”. In: *European Journal of Operational Research* 127.2, pp. 394 –407. DOI: 10.1016/S0377-2217(99)00485-3 (cit. on p. 15).

Hooker, J. N. (1995). “Testing heuristics: We have it all wrong”. In: *Journal of Heuristics* 1.1, pp. 33–42. DOI: 10.1007/BF02430364 (cit. on p. 22).

Kahn, A. B. (1962). “Topological sorting of large networks”. In: *Communications of the ACM* 5.11, pp. 558–562. DOI: 10.1145/368996.369025 (cit. on p. 6).

Kolisch, R. and A. Sprecher (1996). “PSPLIB — A project scheduling problem library”. In: *European Journal of Operational Research* 96, pp. 205–216 (cit. on pp. 24, 30).

Lombardi, M. and M. Milano (2012). “A min-flow algorithm for Minimal Critical Set detection in Resource Constrained Project Scheduling”. In: *Artificial Intelligence* 182-183, pp. 58–67. DOI: 10.1016/j.artint.2011.12.001 (cit. on p. 7).

McGeoch, C. C. (1996). “Feature ArticleToward an Experimental Method for Algorithm Simulation”. In: *INFORMS Journal on Computing* 8.1, pp. 1–15. DOI: 10.1287/ijoc.8.1.1 (cit. on pp. 22–24).

Policella, N., S. F. Smith and A. Oddi (2004). “Generating Robust Schedules through Temporal Flexibility”. In: *ICAPS*, pp. 209–218 (cit. on p. 18).

Policella, N. et al. (2004). “Generating Robust Partial Order Schedules”. In: pp. 496–511 (cit. on p. 16).

Policella, N. et al. (2007). "From Precedence Constraint Posting to Partial Order Schedules". In: *Ai Communications* 20.3, pp. 1–17 (cit. on pp. 6–8, 16, 18, 25, 29, 30).

Policella, N. et al. (2009). "Solve-and-robustify". In: *Journal of Scheduling* 12.3, pp. 299–314. DOI: 10.1007/s10951-008-0091-7 (cit. on pp. 5, 6, 8, 16, 18, 19, 27, 61).

Rasconi, R., A. Cesta and N. Policella (2008). "Validating scheduling approaches against executional uncertainty". In: *Journal of Intelligent Manufacturing* 21.1, pp. 49–64. DOI: 10.1007/s10845-008-0172-7 (cit. on p. 14).

Wilcoxon, F. (1945). "Individual Comparisons by Ranking Methods". In: *Biometrics Bulletin* 1.6, pp. 80–83 (cit. on p. 27).

Wilmer, D. (2014). *RCPSP Testing Framework*. DOI: 10.5281/zenodo.13309 (cit. on p. 25).

Wilson, M. et al. (2013). "Enhancing flexibility and robustness in multi-agent task scheduling". In: *Proceedings OPTMAS workshop* (cit. on pp. 5, 13, 25).

# Appendix A

# Implementation Details

In this appendix we will describe some implementation details. Although they may be interesting to the reader, they are not required to understand the rest of the thesis.

## A.1 Chaining Implementation

Our implementation of the chaining procedure differs slightly from the pseudocode as described in Policella et al. (2009). In our implementation, we view the heuristics as filters to narrow down the set of possible chains. Using some simple commands, we combine simple filters to form a complicated filter function. From the filtered set of candidate chains, we randomly select one chain.

We have several simple filter functions, forming the building blocks of the large filter function. Among other we have implemented a maxCC and a minID function to use these metrics to select a chain. The first filter we use is a filter called filterByTime, which removes all chains of which the latest activity ends later than the activity which currently needs to be assigned to a chain.

These building blocks can then be combined using a “sequence” function, applying the second filter to the results of the first one, or the “if-then-else” combinator. This “if-then-else” combinator applies the “if” filter to the set of chains and checks if the result is non-empty. If that is the case, the “then”-filter is applied to the result and that is returned. Otherwise, the “else”-filter is applied to the original input, and that result is returned. In combination with an identity filter, which returns the input set, this combinator can be used for all purposes we have encountered. For example, we implemented the maxCCminID heuristic as the following filter combination:

```
sequence(
    filterByTime,
    ifThenElse(
        maxCC, identity,
        ifThenElse(
            minID, identity,
            identity
            )
        )
)
```

## A.2 Translation of POS creation to Mixed Integer Problem

The basic idea behind the MIP instance was similar to the chaining procedure: using a fixed-time schedule, every activity can use resources from any other activity that finishes before its start time. Encoding the problem to MIP was done using two types of variables:

- $r_{a,b,k} \in \mathbb{R}^+$ indicating the resource flow from activity $a$ to activity $b$ of resource $k$. In other words: activity $b$ uses $r_{a,b,k}$ units of resource $k$ that activity $a$ also uses. There is one such variable for every $a, b \in ACT, k \in [1,R] | s(a) + t_a \leq s(b)$.
- $l_{a,b} \in (0,1)$ indicating that there is non-zero resource flow from activity $a$ to activity $b$. In other words: activity $b$ uses at least one unit resource that activity $a$ also uses. This means that activity $a$ should be scheduled before activity $b$ in the POS, creating a precedence constraint. There is one such variable for every $a, b \in ACT | s(a) + t_a \leq s(b)$.

To encode the structure of the problem, the following constraints were added to MIP instance:

1. $$\forall a \in ACT, k \in [1,R] : \sum_{b \in ACT | s(b)+t_b \leq s(a)} r_{b,a,k} = req_a[r] \tag{A.1}$$
   This constraint enforces that the total amount of resources that $a$ uses is equal to its requirements.

2. $$\forall a \in ACT, k \in [1,R] : \sum_{b \in ACT | s(a)+t_a \leq s(b)} r_{a,b,k} = req_a[r] \tag{A.2}$$
   This constraint enforces that the total amount of resources from $a$ that can be used by other activities is equal to its requirements.

3. $$\forall a, b \in ACT | s(a) + t_a \leq s(b) : M \times l_{a,b} - \sum_{k \in [1,R]} (r_{a,b,k}) \geq 0 \tag{A.3}$$
   In this constraint we set $M = \sum(CAP)$. This constraint enforces that $l_{a,b}$ will not be 0 — and therefore it has to be 1 1 — when any of the $r_{a,b,k}$ variables are non-zero.

4. $$\forall a, b, c \in ACT | s(a) + t_a \leq s(b) \wedge s(b) + t_b \leq s(c) : 2l_{a,c} - l_{a,b} - l_{b,c} \geq 0 \tag{A.4}$$
   This constraint ensures that for every pair of (posted) precedence constraints $(a,b)$ and $(b,c)$, the transitive constraint $(a,c)$ is also set.

5. $$\forall (a,b) \in P : l_{a,b} = 1 \tag{A.5}$$

   This constraints ensures that all precedence constraints from the original instance are taken into account.

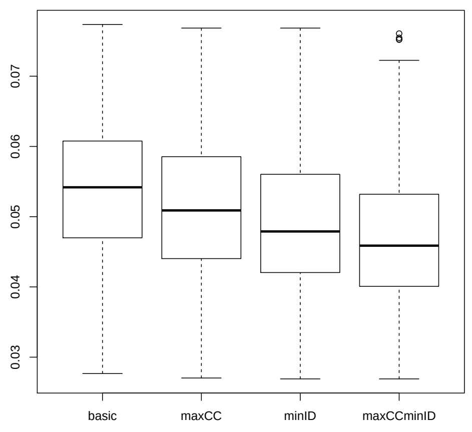
Since $flex_{seq}$ is equal to the portion of pairs of activities that are not related by one or more precedence constraints, maximizing this metric is the same as minimizing the number of activities that are related by one or more precedence constraints. Since every pair of activities that is related by a precedence constraint corresponds to a value of 1 for variable $l_{a,b}$, we choose our objective function as follows:

$$Z = \sum_{a,b \in ACT | s(a)+t_a \leq s(b)} (l_{a,b}) \tag{A.6}$$

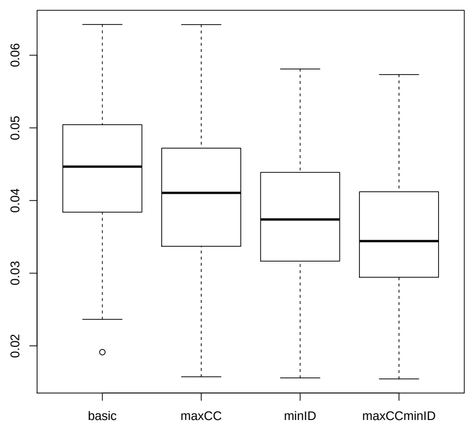
This makes our goal the following: Minimize $Z$ as defined in Equation A.6 under the constraints as defined in Equations A.1, A.2, A.3, A.4, and A.5.

# Appendix B

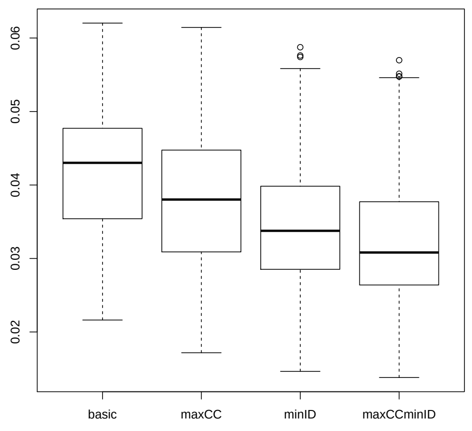
# Graphs



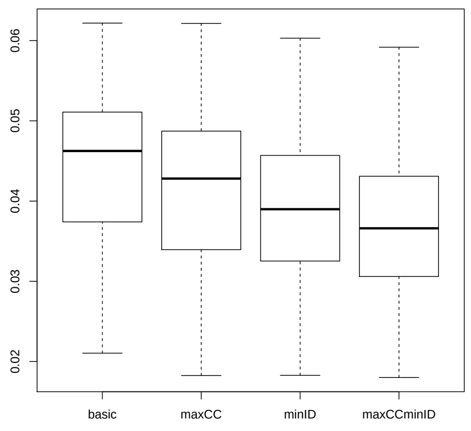
(a) J30 instance set



(b) J60 instance set

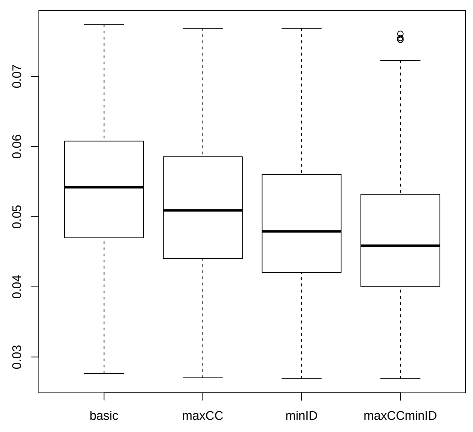


(c) J90 instance set

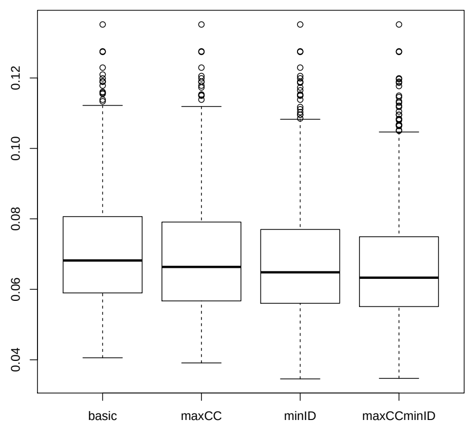


(d) J120 instance set

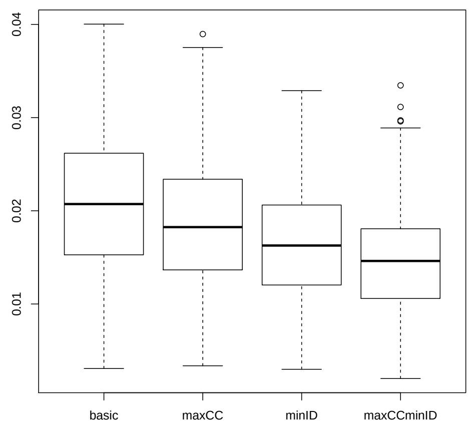
Figure B.1: Comparison of Normalized Makespan Increase for different instance sets, using the exp2 delay set



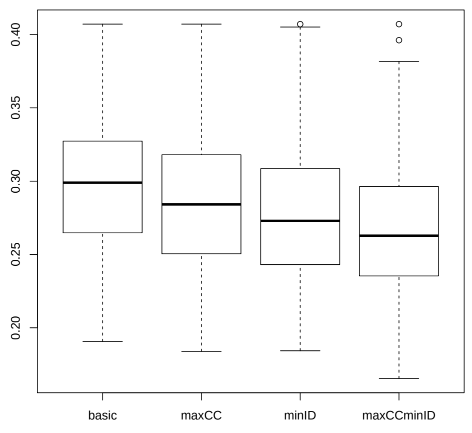
(a) Exp2 delay set



(b) Fixed_50_30 delay set

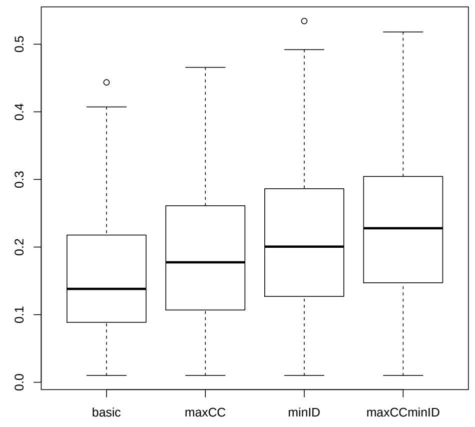


(c) Gauss_1_02 delay set

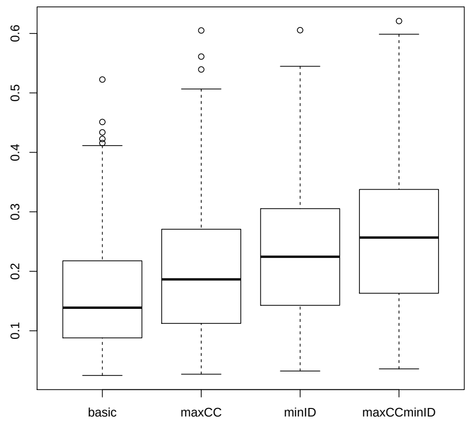


(d) Unif_80_5 delay set

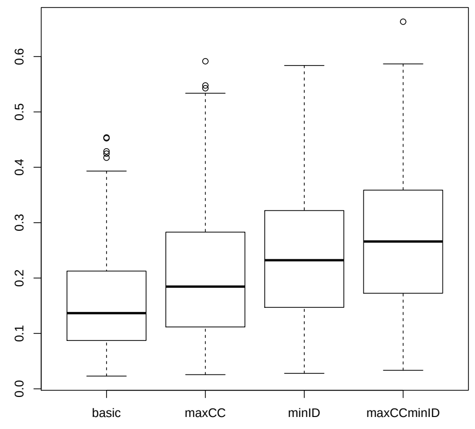
Figure B.2: Comparison of Normalized Makespan Increase for different delay sets, using the J30 instance set



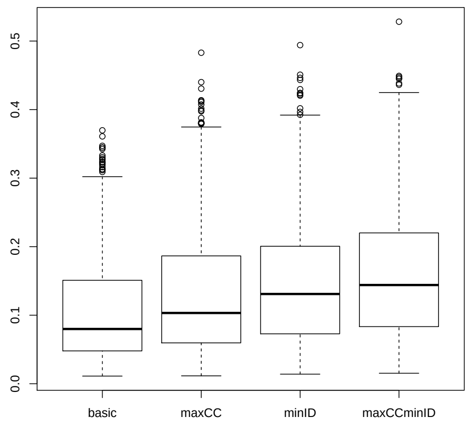
(a) J30 instance set



(b) J60 instance set



(c) J90 instance set



(d) J120 instance set

Figure B.3: Comparison of Flexibility for different Instance sets

# Appendix C

# Tables

| | j30 | j60 | j90 | j120 |
|---|---|---|---|---|
| exp2 | −0.854 | −0.901 | −0.921 | −0.923 |
| fixed_50_30 | −0.842 | −0.848 | −0.843 | −0.821 |
| unif_80_5 | −0.923 | −0.944 | −0.936 | −0.925 |
| gauss_1_02 | −0.343 | −0.708 | −0.816 | −0.794 |

Table C.1: Correlations between Flexibility and normalized makespan increase, for different test sets and instance sizes

| | j30 | j60 | j90 | j120 |
|---|---|---|---|---|
| exp2 | −0.884 | −0.923 | −0.943 | −0.928 |
| fixed_50_30 | −0.815 | −0.840 | −0.853 | −0.823 |
| unif_80_5 | −0.946 | −0.947 | −0.948 | −0.941 |
| gauss_1_02 | −0.340 | −0.605 | −0.712 | −0.628 |

Table C.2: Correlations between $flex_{seq}$ metric and number of delayed activities, for different test sets and instance sizes

| Instance Set | Weighting | Slack | Binary Slack | Capped Slack |
|---|---|---|---|---|
| J30 | Unweighted | −0.532 | −0.582 | −0.551 |
| | Weighted by #successors | −0.559 | −0.523 | −0.535 |
| | Weighted by total resources | −0.145 | 0.095 | −0.007 |
| | Weighted by #succ × resources | −0.155 | 0.006 | −0.051 |
| J60 | Unweighted | −0.640 | −0.611 | −0.642 |
| | Weighted by #successors | −0.653 | −0.434 | −0.616 |
| | Weighted by total resources | −0.209 | 0.242 | 0.026 |
| | Weighted by #succ × resources | −0.160 | 0.209 | 0.057 |
| J90 | Unweighted | −0.686 | −0.621 | −0.672 |
| | Weighted by #successors | −0.669 | −0.398 | −0.608 |
| | Weighted by total resources | −0.269 | 0.235 | −0.035 |
| | Weighted by #succ × resources | −0.163 | 0.268 | 0.065 |
| J120 | Unweighted | −0.687 | −0.642 | −0.697 |
| | Weighted by #successors | −0.714 | −0.513 | −0.702 |
| | Weighted by total resources | −0.100 | 0.506 | 0.194 |
| | Weighted by #succ × resources | 0.012 | 0.453 | 0.264 |

Table C.3: Correlations between slack-based metrics and number of violations for various instance sets using exp2 delay set.

| Instance Set | Weighting | Slack | Binary Slack | Capped Slack |
|---|---|---|---|---|
| J30 | Unweighted | −0.394 | −0.459 | −0.437 |
| | Weighted by #successors | −0.423 | −0.465 | −0.441 |
| | Weighted by total resources | −0.132 | 0.030 | −0.046 |
| | Weighted by #succ × resources | −0.160 | −0.075 | −0.098 |
| J60 | Unweighted | −0.472 | −0.473 | −0.463 |
| | Weighted by #successors | −0.495 | −0.411 | −0.476 |
| | Weighted by total resources | −0.212 | 0.120 | −0.020 |
| | Weighted by #succ × resources | −0.192 | 0.069 | −0.022 |
| J90 | Unweighted | −0.509 | −0.459 | −0.506 |
| | Weighted by #successors | −0.496 | −0.354 | −0.477 |
| | Weighted by total resources | −0.264 | 0.132 | −0.078 |
| | Weighted by #succ × resources | −0.184 | 0.136 | −0.005 |
| J120 | Unweighted | −0.490 | −0.470 | −0.506 |
| | Weighted by #successors | −0.529 | −0.480 | −0.555 |
| | Weighted by total resources | −0.113 | 0.364 | 0.124 |
| | Weighted by #succ × resources | −0.060 | 0.261 | 0.128 |

Table C.4: Correlations between slack-based metrics and number of violations for various instance sets using fixed_50_30 delay set.

| Instance Set | Weighting | Slack | Binary Slack | Capped Slack |
|---|---|---|---|---|
| J30 | Unweighted | −0.545 | −0.522 | −0.532 |
| | Weighted by #successors | −0.545 | −0.442 | −0.581 |
| | Weighted by total resources | −0.113 | 0.166 | 0.048 |
| | Weighted by #succ × resources | −0.095 | 0.081 | 0.023 |
| J60 | Unweighted | −0.600 | −0.540 | −0.583 |
| | Weighted by #successors | −0.599 | −0.385 | −0.548 |
| | Weighted by total resources | −0.203 | 0.236 | 0.035 |
| | Weighted by #succ × resources | −0.145 | 0.195 | 0.065 |
| J90 | Unweighted | −0.620 | −0.535 | −0.594 |
| | Weighted by #successors | −0.601 | −0.358 | −0.540 |
| | Weighted by total resources | −0.273 | 0.206 | −0.047 |
| | Weighted by #succ × resources | −0.173 | 0.224 | 0.049 |
| J120 | Unweighted | −0.630 | −0.576 | −0.640 |
| | Weighted by #successors | −0.659 | −0.491 | −0.656 |
| | Weighted by total resources | −0.103 | 0.488 | 0.189 |
| | Weighted by #succ × resources | −0.004 | 0.411 | 0.237 |

Table C.5: Correlations between slack-based metrics and number of violations for various instance sets using unif_80_5 delay set.

| Instance Set | Weighting | Slack | Binary Slack | Capped Slack |
|---|---|---|---|---|
| J30 | Unweighted | −0.585 | −0.459 | −0.546 |
| | Weighted by #successors | −0.528 | −0.195 | −0.393 |
| | Weighted by total resources | −0.086 | 0.253 | 0.091 |
| | Weighted by #succ × resources | 0.007 | 0.307 | 0.162 |
| J60 | Unweighted | −0.780 | −0.683 | −0.792 |
| | Weighted by #successors | −0.759 | −0.250 | −0.659 |
| | Weighted by total resources | −0.218 | 0.347 | 0.027 |
| | Weighted by #succ × resources | −0.092 | 0.443 | 0.174 |
| J90 | Unweighted | −0.835 | −0.761 | −0.831 |
| | Weighted by #successors | −0.810 | −0.333 | −0.704 |
| | Weighted by total resources | −0.259 | 0.300 | −0.037 |
| | Weighted by #succ × resources | −0.122 | 0.434 | 0.143 |
| J120 | Unweighted | −0.816 | −0.742 | −0.819 |
| | Weighted by #successors | −0.795 | −0.337 | −0.709 |
| | Weighted by total resources | −0.201 | 0.464 | 0.091 |
| | Weighted by #succ × resources | 0.009 | 0.594 | 0.319 |

Table C.6: Correlations between slack-based metrics and number of violations for various instance sets using gauss_1_02 delay set.

| Instance Set | Weighting | Slack | Binary Slack | Capped Slack |
|---|---|---|---|---|
| J30 | Unweighted | −0.560 | −0.579 | −0.654 |
| | Weighted by #successors | −0.570 | −0.492 | −0.600 |
| | Weighted by total resources | −0.202 | 0.080 | −0.114 |
| | Weighted by #succ × resources | −0.182 | 0.016 | −0.114 |
| J60 | Unweighted | −0.672 | −0.630 | −0.400 |
| | Weighted by #successors | −0.681 | −0.450 | −0.673 |
| | Weighted by total resources | −0.326 | 0.168 | −0.100 |
| | Weighted by #succ × resources | −0.260 | 0.161 | −0.039 |
| J90 | Unweighted | −0.699 | −0.632 | −0.696 |
| | Weighted by #successors | −0.692 | −0.437 | −0.648 |
| | Weighted by total resources | −0.369 | 0.163 | −0.148 |
| | Weighted by #succ × resources | −0.266 | 0.203 | −0.030 |
| J120 | Unweighted | −0.709 | −0.664 | −0.731 |
| | Weighted by #successors | −0.731 | −0.554 | −0.737 |
| | Weighted by total resources | −0.206 | 0.451 | 0.093 |
| | Weighted by #succ × resources | −0.090 | 0.395 | 0.172 |

Table C.7: Correlations between slack metrics and Δmakespan for various instance sets using exp2 test set.

| Instance Set | Weighting | Slack | Binary Slack | Capped Slack |
|---|---|---|---|---|
| J30 | Unweighted | −0.292 | −0.449 | −0.430 |
| | Weighted by #successors | −0.364 | −0.510 | −0.479 |
| | Weighted by total resources | −0.114 | −0.036 | −0.119 |
| | Weighted by #succ × resources | −0.177 | −0.156 | −0.190 |
| J60 | Unweighted | −0.341 | −0.402 | −0.387 |
| | Weighted by #successors | −0.396 | −0.487 | −0.469 |
| | Weighted by total resources | −0.174 | 0.045 | −0.057 |
| | Weighted by #succ × resources | −0.213 | −0.074 | −0.125 |
| J90 | Unweighted | −0.394 | −0.372 | −0.400 |
| | Weighted by #successors | −0.422 | −0.448 | −0.443 |
| | Weighted by total resources | −0.225 | 0.093 | −0.075 |
| | Weighted by #succ × resources | −0.212 | −0.006 | −0.085 |
| J120 | Unweighted | −0.408 | −0.399 | −0.431 |
| | Weighted by #successors | −0.465 | −0.551 | −0.524 |
| | Weighted by total resources | −0.053 | 0.336 | 0.142 |
| | Weighted by #succ × resources | −0.070 | 0.135 | 0.058 |

Table C.8: Correlations between slack metrics and Δmakespan for various instance sets using fixed_50_30 delay set.

| Instance Set | Weighting | Slack | Binary Slack | Capped Slack |
|---|---|---|---|---|
| J30 | Unweighted | −0.563 | −0.529 | −0.605 |
| | Weighted by #successors | −0.565 | −0.461 | −0.555 |
| | Weighted by total resources | −0.172 | 0.123 | −0.039 |
| | Weighted by #succ × resources | −0.152 | 0.041 | −0.056 |
| J60 | Unweighted | −0.582 | −0.536 | −0.590 |
| | Weighted by #successors | −0.598 | −0.445 | −0.592 |
| | Weighted by total resources | −0.278 | 0.157 | −0.059 |
| | Weighted by #succ × resources | −0.238 | 0.101 | −0.042 |
| J90 | Unweighted | −0.583 | −0.511 | −0.571 |
| | Weighted by #successors | −0.582 | −0.424 | −0.551 |
| | Weighted by total resources | −0.321 | 0.151 | −0.105 |
| | Weighted by #succ × resources | −0.242 | 0.131 | −0.033 |
| J120 | Unweighted | −0.592 | −0.541 | −0.604 |
| | Weighted by #successors | −0.626 | −0.539 | −0.645 |
| | Weighted by total resources | −0.137 | 0.437 | 0.140 |
| | Weighted by #succ × resources | −0.066 | 0.314 | 0.152 |

Table C.9: Correlations between slack metrics and Δmakespan for various instance sets using unif_80_5 delay set.

| Instance Set | Weighting | Slack | Binary Slack | Capped Slack |
|---|---|---|---|---|
| J30 | Unweighted | −0.577 | −0.433 | −0.578 |
| | Weighted by #successors | −0.492 | −0.164 | −0.396 |
| | Weighted by total resources | −0.161 | 0.208 | −0.002 |
| | Weighted by #succ × resources | −0.021 | 0.282 | 0.110 |
| J60 | Unweighted | −0.778 | −0.691 | −0.794 |
| | Weighted by #successors | −0.741 | −0.320 | −0.672 |
| | Weighted by total resources | −0.359 | 0.218 | −0.116 |
| | Weighted by #succ × resources | −0.220 | 0.307 | 0.041 |
| J90 | Unweighted | −0.781 | −0.709 | −0.774 |
| | Weighted by #successors | −0.745 | −0.375 | −0.668 |
| | Weighted by total resources | −0.391 | 0.179 | −0.167 |
| | Weighted by #succ × resources | −0.243 | 0.295 | 0.016 |
| J120 | Unweighted | −0.799 | −0.749 | −0.823 |
| | Weighted by #successors | −0.780 | −0.437 | −0.745 |
| | Weighted by total resources | −0.322 | 0.426 | −0.005 |
| | Weighted by #succ × resources | −0.106 | 0.515 | 0.214 |

Table C.10: Correlations between slack metrics and Δmakespan for various instance sets using gauss_1_02 delay set.