Resolving Disruptions in Simple Temporal Problems

Negative Cycle Elimination Algorithms for Weighted, Directed Graphs

Lian Ien Oei
Resolving Disruptions in Simple Temporal Problems

Negative Cycle Elimination Algorithms
for Weighted, Directed Graphs

Master’s Thesis in Computer Science

Lian lën Oei

9616687

15th May 2009

Algorithmics group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Abstract

Simple Temporal Problems (STPs) can be used for representation of and reasoning with temporal constraint satisfaction problems. In dynamic environments, it often happens that – after a problem has been modelled as an STP – a change occurs in one of the constraints. This can lead to an inconsistent situation, in which case there are negative cycles in the distance graph representation of the STP.

In this thesis, we discuss heuristic algorithms for solving such disruptions in STPs. More specifically, we try to find a minimum set of constraints that has to be changed to remove all negative cycles from the distance graph. First, we give an overview of known approaches for solving disruptions in STPs. After that, we present some new algorithms. We look at the basic case, in which constraints may entirely be removed from the graph. Then, we discuss the case in which preferences are added to the model. Furthermore, we investigate the situation in which a single constraint is known to be the source of the inconsistency, and we want to repair the STP without changing this specific constraint. Finally, we empirically evaluate the performance of our algorithms for the basic case, and for repairing an STP with a known source of inconsistency. In these experiments, our new algorithms are shown to outperform existing algorithms.
Acknowledgements

I would like to take this opportunity to thank all the people who supported me during this thesis project. First and foremost, I thank Cees Witteveen, for his guidance and support throughout this project. He has always been a great source of inspiration for me. Second, I thank Renze Steenhuisen, for all his advice and great feedback on my work. I thank the other members of my graduation committee, for taking the time to familiarise themselves with my work.

I thank Pieter Buzing, who acted as my daily supervisor in the early stages of this project, for familiarising me with the subject and the basics of scientific research in the first place. I thank Léon Planken, for sharing his source code with me. Without his help, I would never have been able to finish all the experiments on time. I thank all my colleagues from the Algorithmics group, the people from the PDS group, the support staff, all my colleague students from the “aquarium” and Cynthia, for the enjoyable time they gave me.

Furthermore, I thank Sjoerd, for always being there for me and never losing faith in me, and my parents, who have always supported me in everything I did.

Lian Ien Oei

Delft, The Netherlands
15th May 2009
A.3 PRI-CYCLE ......................................................... 86
A.4 Tarjan’s algorithm ................................................. 87
A.5 YTO algorithm ..................................................... 89
A.6 Edmonds-Karp ...................................................... 92

B Detailed discussion of experiments 95
B.1 Experiment group 1: Basic case ................................. 95
B.2 Experiment group 2: A known source of inconsistency .... 116
### List of Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG-CYCLE (NC)</td>
<td>15</td>
</tr>
<tr>
<td>CRI-CYCLE (CC)</td>
<td>16</td>
</tr>
<tr>
<td>Top-down algorithm for the basic case (TD)</td>
<td>26</td>
</tr>
<tr>
<td>Bottom-up algorithm for the basic case (BU)</td>
<td>27</td>
</tr>
<tr>
<td>Top-down algorithm with post-processing for two preference levels</td>
<td>31</td>
</tr>
<tr>
<td>Repairing an STP with a known source of inconsistency (KSI)</td>
<td>39</td>
</tr>
<tr>
<td>Floyd-Warshall</td>
<td>84</td>
</tr>
<tr>
<td>Incremental Full Path Consistency (IFPC)</td>
<td>85</td>
</tr>
<tr>
<td>PRI-CYCLE (PC)</td>
<td>86</td>
</tr>
<tr>
<td>Tarjan’s algorithm</td>
<td>87</td>
</tr>
<tr>
<td>Young-Tarjan-Orlin (YTO)</td>
<td>90</td>
</tr>
<tr>
<td>Find-arc-key($a_{ij}, G$)</td>
<td>91</td>
</tr>
<tr>
<td>Edmonds-Karp</td>
<td>93</td>
</tr>
<tr>
<td>Find-augmenting-path($G, rc, s, t$)</td>
<td>94</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Schiphol Airport ........................................... 2

2.1 Simple Temporal Network ................................. 7
2.2 Distance graph ............................................. 7
2.3 $d$-graph .................................................. 8
2.4 Semi-convex and non-semi-convex functions .......... 10
2.5 Amsterdam canals ......................................... 11
2.6 Preference function ....................................... 12
2.7 Minimum vs. minimal set ................................. 13
2.8 Inconsistent distance graph ............................. 21
2.9 Positive arcs influence cycle ratios ................. 21

4.1 Number of vertices and clauses for DTP and jobshop benchmarks 46
4.2 Summary statistics of Experiment group 1A ........... 51
4.3 Experiment 1A.1: Random STPs with original benchmark weights 52
4.4 Experiment 1A.4: Random STPs with mostly positive arcs with few inconsistencies ........................................... 54
4.5 Experiment 1B.1: Practical scheduling problems with original benchmark weights ........................................... 55
4.6 Results of Experiment group 1B .......................... 56
4.7 Experiment 1B.6: Practical scheduling problems with mostly positive arcs with one inconsistency source (percentage) 58
4.8 Results of Experiment group 2A ......................... 68

B.1 Experiment 1A.1: Random STPs with original benchmark weights 97
B.2 Experiment 1A.2: Random STPs with only negative arcs .... 98
B.3 Experiment 1A.3: Random STPs with a tight schedule with few inconsistencies ........................................... 99
B.4 Experiment 1A.3: Random STPs with a tight schedule with few inconsistencies ........................................... 100
B.5 Experiment 1A.4: Random STPs with mostly positive arcs with few inconsistencies ........................................... 101
B.6 Experiment 1A.5: Random STPs with a tight schedule with one inconsistency source ........................................... 102
B.30 Experiment 2B.3: Practical scheduling problems with a tight schedule with a very small disruption . . . . . . . . . . . . . . . . . . . 129
B.31 Experiment 2B.4: Practical scheduling problems with mostly positive arcs with a very small disruption . . . . . . . . . . . . . . . . 130
B.32 Experiment 2B.5: Practical scheduling problems with a tight schedule with a big disruption . . . . . . . . . . . . . . . . . . . . . . . . . 131
B.33 Experiment 2B.6: Practical scheduling problems with mostly positive arcs with a big disruption . . . . . . . . . . . . . . . . . . . . . . . . . 132
## List of Tables

2.1 All-pairs-shortest-paths matrix ................................................. 9  
4.1 Summary statistics of Experiment 2A ................................. 66  
4.2 Summary statistics of Experiments 2B.1 and 2B.3 with identical random seeds ................................................. 70  
B.1 Summary statistics of Experiment 2A.1: Random STPs with a tight schedule ................................................. 118  
B.2 Summary statistics of Experiment 2A.2: Random STPs with mostly positive arcs ................................................. 119  
B.3 Summary statistics of Experiment 2A.3: Random STPs with a tight schedule with a very small disruption ................................................. 120  
B.4 Summary statistics of Experiment 2A.4: Random STPs with mostly positive arcs with a very small disruption ................................................. 121  
B.5 Summary statistics of Experiment 2A.5: Random STPs with a tight schedule with a big disruption ................................................. 122  
B.6 Summary statistics of Experiment 2A.6: Random STPs with mostly positive arcs with a big disruption ................................................. 124  
B.7 Summary statistics of Experiment 2B.1: Practical scheduling problems with a tight schedule ................................................. 126  
B.8 Summary statistics of Experiment 2B.2: Practical scheduling problems with mostly positive arcs ................................................. 128  
B.9 Summary statistics of Experiment 2B.3: Practical scheduling problems with a tight schedule with a very small disruption ................................................. 129  
B.10 Summary statistics of Experiment 2B.4: Practical scheduling problems with mostly positive arcs with a very small disruption ................................................. 130  
B.11 Summary statistics of Experiment 2B.5: Practical scheduling problems with a tight schedule with a big disruption ................................................. 131  
B.12 Summary statistics of Experiment 2B.6: Practical scheduling problems with mostly positive arcs with a big disruption ................................................. 132
Chapter 1

Introduction

Sea, sea, nothing but sea. . . A bit weary, Alice looks through the window. She was lucky to get one of the last window seats. If only there would be anything more interesting to see than just water. Going on vacation is really great, but why does the journey always have to take so long? Then, suddenly a feeling of excitement runs through her body as she sees the Dutch coastline. Finally! This means, that in about 15 minutes, her plane will be landing at Schiphol Airport.

Alice looks at her watch. 9:45 AM. Nice on time. In her head she repeats the plan she has made for the rest of the day. After the plane has landed at 10:00 AM, she will need at least three hours to pick up her luggage, take the train to Amsterdam, and get to her hotel. Then, she wants to freshen up herself and take rest for at least three hours, because the journey was very tiring. Will she have enough time left in the afternoon to explore the city for two to four hours, before having dinner at 06:30 PM?

In many complex processes, planning and scheduling play a very important role. Often, sophisticated schedules have to be developed, in order to meet the high efficiency requirements and deal with the limited availability of resources. Many of these processes can be modelled as temporal problems. Alice’s planning problem is just a very simple example of such a problem. In general, a temporal problem is a problem in which temporal constraints are involved. Examples of such constraints are: “The plane lands at 10:00 AM.”, “Alice can pick up her luggage after the plane has landed.”, and “She wants to explore the city for two to four hours.”. As these examples show, temporal constraints may be formulated in a ordinal or metric way, or may contain references to absolute time. Given such a temporal problem, the goal is to answer questions such as: “At what time can Alice arrive at her hotel?” or “Will she have enough time to explore the city?”.

Alice attempts to find a schedule for her plan. If the plane lands at 10:00 AM, and it takes her three hours to pick up her luggage and get to the hotel, then she will be there at 01:00 PM. Then, she will freshen up herself and take a rest until 04:00 PM. Thus, she will have two and a half hours left, to explore the city before dinner, which fits neatly into her plan. She is glad to see that her plan is feasible.
The Simple Temporal Problem (STP) formalism, which has been developed by Dechter, Meiri and Pearl [10], has proved to be a very popular formalism for representation of and reasoning with temporal information. Many temporal problems can be modelled as an STP. They can be solved by finding a solution to their STP representation.

1.1 A disruption

The plane has landed, and all passengers have gotten out of the aircraft. Thanks to the clear signs, Alice has easily found her way to the baggage hall. Now she is waiting for her suitcase to arrive. Impatiently, she looks at her watch. It is 12:30 PM already. Most other passengers have left, and there has not arrived any new luggage on the conveyor belt for a long time. She is getting worried. What could have happened to her suitcase?

In dynamic environments, it often happens that something changes compared to the original problem situation, which leads to a change in the set of constraints. As a result of such a change, the STP can become inconsistent, which means that it is impossible to find a solution to the problem. If an STP is inconsistent, then there are negative cycles in its distance graph representation. In order to still be able to find a solution to the problem, the STP has to be repaired. This means that constraints have to be changed or removed from the distance graph, in order to remove all negative cycles from the graph.

Of course, if one small thing changes compared to the original situation, we do not want to adapt the entire plan we had already made. Instead, we would like
to be able to reuse as much of our plan as possible. Therefore, we want to find a minimum set of constraints that has to be changed to free the distance graph of negative cycles.

In finding such a minimum set, however, we have to keep in mind that the constraints in the graph have a meaning. They are used to model a real-world problem. This means that often we cannot simply remove any constraint from the graph, for this could lead to a plan that cannot be executed in practice. Therefore, we need to indicate which constraints can be changed, and by which amount they can be changed, by adding the notion of preferences to the model.

Also, in many situations there is much more information available than just the inconsistent graph. Often, there used to be a consistent graph, that has become inconsistent because of a change in one single constraint. In such situation, the knowledge of this previously consistent graph and the cause of the inconsistency, can be applied to reduce the search space.

At 12:45 PM, Alice does not expect anymore that her suitcase will still arrive. She decides to contact the luggage handling company, and report her suitcase as missing. After all the paperwork has been completed, it is 01:30 PM when she can finally continue her journey. She expects that now she will arrive at her hotel at 03:00 PM at the earliest. Being two hours delayed, she will never be able to execute the entire schedule she has made on the plane. Thus, she will have to adapt her initial plan. She does have to go to her hotel in Amsterdam anyway, so eliminating the remainder of her journey is not an option. However, she can decide to solve the problem by taking a shorter rest break, by postponing the exploration of the city until tomorrow, or by dining a little later.

1.2 Overview and contributions

In this thesis, we discuss heuristic algorithms for finding a minimum set of constraints that has to be changed to free the distance graph of negative cycles. The remainder of this document is structured as follows.

In Chapter 2, we start with the definitions and notations used in this document. We formally describe the Simple Temporal Problem formalism, which is the basis for our work. We discuss methods for solving STPs, and extensions of the formalism with preference information. Then, we introduce the problem of solving disruptions in STPs. We give an overview of existing approaches for removing negative cycles from a distance graph. We evaluate these approaches, and give a motivation for our research.

In Chapter 3, we present our new algorithms for solving disruptions in STPs. We first discuss the basic case, in which we simply want to find a minimum set of arcs that has to be removed from the graph. After that, we extend the model with preference information. We discuss the situation in which there are two preference levels, before generalising to multiple preference levels. Furthermore, we discuss the situation in which a change in a single constraint is known to be the source of
the inconsistency.

In Chapter 4, we evaluate the performance of our new algorithms through empirical research. In several experiments, we compare our algorithms for the basic case and our algorithm for repairing an STP with a known source of inconsistency with existing algorithms. We test their performance on a variety of test cases taken from two benchmark sets, which represent random STPs and practical scheduling problems.

Finally, in Chapter 5, we end with a summary, our conclusions and a list of the topics for future work.
Chapter 2

Background

This chapter defines the Simple Temporal Problem (STP) formalism and introduces several ways of representing STPs in Section 2.1. Then, it describes how such problems can be solved in Section 2.2. Section 2.3 describes an extension of the formalism with preference information. Section 2.4 formally defines the problem of resolving disruptions in STPs, which we investigate in this thesis. In Section 2.5, we give an overview of existing approaches for solving disruptions in STPs. Finally, in Section 2.6 we evaluate these existing approaches, and give a motivation for our own research.

2.1 The Simple Temporal Problem

In 1991, Dechter, Meiri and Pearl [10] developed the Simple Temporal Problem (STP) formalism for representation of and reasoning with temporal problems. In this section we give a formal definition of the model, and show several ways of representing an STP in a directed graph.

2.1.1 Model

An STP \( S = \langle X, C \rangle \) consists of a finite set of time point variables \( X = \{x_0, x_1, \ldots, x_n\} \) (representing events), and a set of binary constraints \( C = \{c_{ij} \mid i, j \in 0, 1, \ldots, n; i \neq j\} \) between these variables. A constraint \( c_{ij} \) is represented by an interval \( I_{ij} = [lb_{ij}, ub_{ij}] \), that indicates the allowed distances between the time points \( x_i \) and \( x_j \). This is often denoted as \( lb_{ij} \leq x_j - x_i \leq ub_{ij} \) or \( x_j - x_i \in [lb_{ij}, ub_{ij}] \), and means that \( x_j \) has to occur at least \( lb_{ij} \) and at most \( ub_{ij} \) time units after \( x_i \).

When modelling a temporal plan as an STP, two time point variables are assigned to each action in the plan: one represents the start of the action, and one represents its end. The time point \( x_0 \) is a special time point, called the temporal reference point, that denotes a fixed point in time (such as the beginning of the execution of the plan). Usually it is assigned the value 0. A unary constraint, which restricts the
domain of a single variable \(x_i\), can be expressed as a binary constraint \(c_{0i}\) between the temporal reference point and \(x_i\). This way, it is possible to assign absolute times to events.

**Example 2.1** Alice’s problem of Chapter 1 can be modelled as an STP as follows:

There are five time point variables \(X = \{x_0, \ldots, x_4\}\), where:

- \(x_0\) = the temporal reference point,
- \(x_1\) = the plane’s arrival at Schiphol Airport,
- \(x_2\) = Alice’s arrival in her hotel room,
- \(x_3\) = start of exploration of the city,
- \(x_4\) = start of dinner.

If the temporal reference point \(x_0\) refers to the current time (9:45 AM) and is assigned the value 0, then the following constraints (expressed in number of minutes) exist between these time point variables: \(C = \{c_{01}, c_{12}, c_{23}, c_{34}, c_{04}\}\), where:

- \(c_{01} = [15, 15]\),
- \(c_{12} = [180, \infty]\),
- \(c_{23} = [180, \infty]\),
- \(c_{34} = [120, 240]\),
- \(c_{04} = [525, 525]\).

### 2.1.2 Simple Temporal Network and distance graph

There are several types of directed graphs to represent an STP. In each of these, the vertices represent the time point variables and the arcs represent the constraints. A Simple Temporal Network (STN) or directed constraint graph is a directed graph in which each arc \(x_i \rightarrow x_j\) corresponds to the constraint \(c_{ij}\) and is labelled by the interval \([lb_{ij}, ub_{ij}]\). In a distance graph each arc \(x_i \rightarrow x_j\) is labelled by a weight \(w_{ij}\) and represents the linear inequality \(x_j - x_i \leq w_{ij}\). The constraint \(lb_{ij} \leq x_j - x_i \leq ub_{ij}\) can be written as two inequalities \(lb_{ij} \leq x_j - x_i\) (or alternatively \(x_i - x_j \leq -lb_{ij}\)) and \(x_j - x_i \leq ub_{ij}\). This constraint is represented by the arc \(x_i [lb_{ij}, ub_{ij}] \rightarrow x_j\) in an STN, and by the two arcs \(x_i \xrightarrow{ub_{ij}} x_j\) and \(x_j \xleftarrow{-lb_{ij}} x_i\) in a distance graph. The weight of an arc \(a\) in the distance graph is also denoted as \(w(a)\). Figure 2.1 shows the STN of the STP in Example 2.1 and Figure 2.2 shows its distance graph.

In this document, the terms “STP” and “STN” will be used interchangeably. The same goes for the terms “variable” and “vertex”, and for the terms “constraint” and “arc”. If the term “graph” is used without further indications, this will refer to the distance graph. If another type of graph is meant, this will be made clear from the context in which the term is used. Furthermore, \(n\) will be used to denote the number of vertices in the graph \((n = |V|)\), and \(m\) will be used to denote the number of arcs in the graph \((m = |A|)\). Thus, if there exist at most 1 forward arc and 1 backward arc between each two vertices, then \(m\) is bounded above by \(2n(n - 1) = O(n^2)\).
2.1.3 \textit{d-graph}

The weight (or length) of a directed path in the distance graph is equal to the sum of the weights of the arcs on this path. Similarly, the weight $w(C)$ of a directed cycle $C$ in the distance graph is equal to the sum of the weights of the arcs in $C$. If in the distance graph, there exists a path from $x_i$ to $x_j$, then (implicitly) there also exists a constraint between $x_i$ and $x_j$, equal to the weight of this path.

A \textit{d-graph} is a complete, directed graph, in which each arc $x_i \rightarrow x_j$ is labelled by the weight of the shortest path $d_{ij}$ from $x_i$ to $x_j$. The \textit{d-graph} can be found by solving the all-pairs-shortest-paths (APSP) problem, which is to find the shortest path between each pair of vertices in the graph. In Section 2.2 we will discuss algorithms for solving the APSP problem. The network that is represented by the \textit{d-graph} is also called the \textit{minimal network}. Figure 2.3 shows the \textit{d-graph} of the STP in Example 2.1.

The arcs in the distance graph express an upper bound for the allowed temporal distance between two variables. Since the arcs in the \textit{d-graph} are labelled by the length of the shortest path between two variables, they give the lowest upper bound of the allowed time distance between these variables. This information is important when finding solutions to an STP, as will be shown in the next section.
2.2 Solving STPs

In this section, we will first show how one can detect whether an STPs is consistent or not. Then, we will discuss algorithms for solving STPs.

2.2.1 Solutions of an STP

A tuple \( T = (\tau_0, \tau_1, \ldots, \tau_n) \) is called a solution if the assignment \( \{ x_0 = \tau_0, x_1 = \tau_1, \ldots, x_n = \tau_n \} \) satisfies all the constraints in \( C \). As said before, usually \( \tau_0 = 0 \). An STP is consistent if at least one solution exists.

**Theorem 2.1 (Consistency (see [10][21]))** An STP is consistent if, and only if, its distance graph contains no negative cycles.

One can easily see that this is true, because the inequality corresponding to a path in the distance graph can be obtained by taking the sum of the inequalities of the arcs on the path. A negative cycle from \( x_i \) to itself would result in the inequality \( x_i - x_i < 0 \), which cannot be satisfied. Conversely, if there are no negative cycles, then the shortest path between each pair of vertices is well defined. For each two time points \( x_i \) and \( x_j \) these shortest paths satisfy \( d_{0j} \leq d_{0i} + a_{ij} \), so the inequality \( d_{0j} - d_{0i} \leq a_{ij} \) holds. Thus the tuple \( (0, d_{01}, \ldots, d_{0n}) \) is a solution to the STP.

**Example 2.2** At 01:30 PM, Alice is finally ready to leave the airport and continue her journey to Amsterdam. At that moment, she expects that she will arrive at her hotel at 03:00 PM at the earliest. This means that the additional constraint \( c_{02} = [315, \infty] \) is added to the STP. This constraint is represented by the two arcs \( x_0 \xrightarrow{\infty} x_2 \) and \( x_2 \xrightarrow{-315} x_0 \) in the distance graph. Now, a negative cycle \( x_0 - x_4 - x_3 - x_2 - x_0 \) of weight \( 525 - 120 - 180 - 315 = -90 \) exists in the graph, which means that the STP has become inconsistent.
Table 2.1: All-pairs-shortest-paths matrix of Example 2.1.

<table>
<thead>
<tr>
<th></th>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_0$</td>
<td>0</td>
<td>15</td>
<td>225</td>
<td>405</td>
<td>525</td>
</tr>
<tr>
<td>$x_1$</td>
<td>-15</td>
<td>0</td>
<td>210</td>
<td>390</td>
<td>510</td>
</tr>
<tr>
<td>$x_2$</td>
<td>-195</td>
<td>-180</td>
<td>0</td>
<td>210</td>
<td>330</td>
</tr>
<tr>
<td>$x_3$</td>
<td>-375</td>
<td>-360</td>
<td>-180</td>
<td>0</td>
<td>150</td>
</tr>
<tr>
<td>$x_4$</td>
<td>-525</td>
<td>-510</td>
<td>-300</td>
<td>-120</td>
<td>0</td>
</tr>
</tbody>
</table>

Once we know that an STP is consistent, we want to find an assignment that satisfies the constraints. Here, the importance of the $d$-graph becomes clear. A common way for constructing a solution, is to choose one variable, assign a value to it, and then proceed with the next variable, until all variables have been assigned a value. However, one has to take into account that the value that will be assigned to a new variable, has to be consistent with the values chosen for all previous variables. It is not sufficient to check the constraints in the distance graph, because these constraints may be too loose: it may be possible to choose a value that is consistent with the constraints in the distance graph, but is inconsistent with the values that have already been chosen.

The $d$-graph, on the other hand, does provide the necessary information. In the $d$-graph, every locally-consistent partial solution (i.e., an assignment of values to a subset of the variables that satisfies the constraints between the variables in this subset) can be extended to a globally consistent solution. This means that the order in which values are assigned to the variables does not matter. For this reason, “solving an STP” can be achieved by determining its minimal network.

### 2.2.2 Finding solutions

The $d$-graph can be constructed by solving the all-pairs-shortest-paths (APSP) problem, which is to find the shortest path between each pair of vertices. This can be done by applying the $O(n^3)$ Floyd-Warshall algorithm [5, 12, 26] (see Appendix A.1) to the distance graph, which results in the APSP matrix. Table 2.1 shows the APSP matrix of Example 2.1. The Floyd-Warshall algorithm can also be used to detect negative cycles, by simply checking the sign of the diagonal elements $d_{ii}$ of this matrix, thereby checking the consistency of the STP.

There are alternatives to Floyd-Warshall, such as the Bellman-Ford algorithm [1, 14], or the $\triangle$STP algorithm [27]. However, it is beyond the scope of this thesis to discuss them all.

### 2.3 STPs with preferences

Sometimes, certain solutions are more preferable than others. For example, Alice wants to take a rest break for at least three hours, but she also wants to have two to
four hours left, to explore the city before dinner. If she does not have enough time
to fulfil both these constraints, then she prefers to take a shorter rest break, rather
than not go into the city at all.

To express that certain values are more preferable than others, Khatib, Morris,
Morris and Rossi [18] have added preferences to STPs. They do this by adding
to each constraint a preference function, that maps each temporal value in the in-
terval identified by that constraint to a preference value. This preference value is
a measure of the desirability of that temporal value. The goal now is to gener-
ate solutions to these problems that are globally preferred, which means that they
simultaneously meet all local preference criteria as well as possible.

In a Simple Temporal Problem with Preferences (STPP), the hard constraints
\( c_{ij} = I_{ij} \) are replaced by soft constraints \( c_{ij} = (I_{ij}, f_{ij}) \), where \( f_{ij} : x \in I_{ij} \rightarrow P \)
is a function and \( P \) is a set of preference values. A solution to an STPP is
a complete assignment of times to variables, that satisfies all temporal distance
constraints. Each solution has a global preference value, obtained by combining
the local preference values found in the constraints. The optimal solutions of an
STPP are those solutions that have the best preference value.

Whereas normal STPs are polynomially solvable, general STPPs are NP-hard
problems [18]. For some subclasses of STPPs, however, tractability can be proved.
Under certain conditions, this is the case for STPPs with semi-convex preference
functions.

**Definition 2.1 (Semi-convex function (see [18]))** A semi-convex function \( f : \mathbb{R} \rightarrow \mathbb{R} \) is one such that, for all constants \( c \), the set \( \{ X : f(X) \geq c \} \) forms a single interval in \( \mathbb{R} \).

Some examples of semi-convex functions are given in Figure 2.4 (examples
taken from [18]).
An STPP with semi-convex preference functions can be solved by searching the space of projected STPs. Here, the projected STPs are defined by the preference functions associated with each STPP constraint. For every preference level \( p \), the projected STP at level \( p \) can be obtained by restricting all constraints to their (sub)interval for which the preference value is at least \( p \). If all preference functions are semi-convex, then every STPP constraint will map to at most one STP constraint, thus resulting in an STP.

An optimal solution can be found by searching for the maximum \( p \) such that there exists a consistent projected STP at level \( p \). If there is a finite set of preference levels \( P \), then this solution can be found in \( O(\log(|P|) \times n^3) \) time, by applying a binary search over the preference levels.

**Example 2.3** The city of Amsterdam is famous for its many historic canals. This afternoon, Alice would like to take a cruise with a canal boat, as this is one of the best ways to explore these canals. She needs at least two and a half hours to walk from her hotel to the departure place of the canal boats, go with the cruise, and then walk to the restaurant. Thus, even though she has planned to explore the city for two to four hours, she prefers to have at least two and a half hours time. This can be expressed by adding the following preference function to constraint \( c_{34} \): 

\[
 f_{13}(a) = \begin{cases} 
 1 & a \in [120, 150) \\
 2 & a \in [150, 240] 
\end{cases}
\]

This function is shown in Figure 2.6.

### 2.4 Disruptions in STPs

Suppose a disruption occurs that leads to an inconsistency of the STP. This means, that there are negative cycles in its distance graph. To resolve the inconsistency, we
can repair the STP by removing all negative cycles from the graph. In this process, we would like to retain as much of the original situation as possible. Thus, we want to find a minimum set of constraints that has to be changed to free the distance graph of negative cycles.

2.4.1 Problem definition and complexity

The problem of finding a minimum set of constraints that has to be changed to free the distance graph of negative cycles, can be formulated formally as follows:

**Problem 2.1** Given a directed graph \( G = \langle V, A \rangle \) with weights \( w(a) \in \mathbb{R} \) for each \( a \in A \), find a minimum subset of arcs \( A' \subseteq A \) such that \( A' \) contains at least one arc from every directed cycle \( C \) in \( G \) with weight \( w(C) < 0 \).

Suppose an algorithm for solving Problem 2.1 has found a subset of arcs \( A' \subseteq A \) such that \( A' \) contains at least one arc from every negative cycle. Then, we can repair the STP by either removing all arcs in \( A' \) from \( G \), or changing their weights in such a way that all cycles become non-negative.

It is not difficult to see that Problem 2.1 is an NP-hard problem. If all cycles in the graph have a negative weight, the problem is equal to the minimum feedback arc set problem.

**Problem 2.2 (Minimum feedback arc set)** Given a directed graph \( G = \langle V, A \rangle \), find a minimum subset of arcs \( A' \subseteq A \), such that \( A' \) contains at least one arc from every directed cycle in \( G \).

The decision version of the minimum feedback arc set problem is a well-known NP-complete problem [15]. Since until now nobody has been able to find a polynomial time solution to any NP-complete problem, our working hypothesis is that it is undoable to solve this problem exactly for larger instances. Therefore, in this research, we restrict ourselves to heuristic algorithms that find a minimal solution.
Figure 2.7: The set \{AB, AD\} is a minimal, but not a minimum set. The set \{CA\} is both a minimum and a minimal set.

### 2.4.2 Minimum vs. minimal set

There are two types of “smallest” solution sets. A *subset minimal set* (or *minimal set*) is a set from which no element can be removed while still constituting a solution. This is, for example, a subset of arcs whose removal will free the graph from all negative cycles, from which it is impossible to remove any arc, without leaving at least one negative cycle in the graph.

There may be multiple of such minimal sets, and these may have different sizes (i.e., they contain a different number of arcs). However, one can unambiguously determine what the smallest size is among all these different sizes. A set that has the smallest size among all minimal sets, is called a *cardinal minimal set* (or *minimum set*).

**Example 2.4** *In the graph of Figure 2.7, the set of arcs \{AB, AD\} is a minimal, but not a minimum set, whose removal will free the graph of all negative cycles. Removing the arcs AB and AD will free the graph of all negative cycles, and it is not possible to put either of these back into the graph, for then there would still be a negative cycle left. However, even a smaller set exists. The set \{CA\} is also a solution to the problem, and has only one element instead of two. One can easily see that it is impossible to free the graph from all negative cycles with an even smaller amount of arcs. Therefore, the set \{CA\} is both a minimum and a minimal set.*

In this thesis, when we use the term *minimal set*, this will always stand for a subset minimal set. When we use the term *minimum set*, this will always stand for a cardinal minimal set. In the cases, where we use the term *optimal solution*, this means a minimum set. In our solutions, we will try to find a minimal, but not necessarily minimum, set.
2.5 Known approaches for solving disruptions in STPs and STPPs

In this section, we give an overview of existing approaches for finding a subset of constraints whose weights have to be changed to remove all negative cycles from a distance graph. Furthermore, we describe how, until now, preference information has been used for plan repair.

2.5.1 Removing negative cycles

Whereas the minimum feedback arc set problem has been studied intensively, until now, there has not been much research on finding a minimum (or minimal) subset of constraints whose weights have to be changed to make a distance graph free of only the negative cycles. As far as we know, only Dasdan has developed (heuristic) algorithms for solving this specific problem.\(^1\) Preliminary versions of his algorithms have been published in 2002 in [6, 7]. Here, an overview will be given of his results as described in [9] in 2009.

In this overview article, three algorithms are described: NEG-CYCLE, CRI-CYCLE and PRI-CYCLE. Each of these finds a set of constraints, whose weights have to be changed to make a distance graph free of negative cycles, in strongly polynomial time. The algorithms differ from each other in the way they choose the constraints in this set. In order not to change the meaning of the constraints, all algorithms fulfil the following two requirements:

1. Only the weights of negative arcs are changed.
2. Negative arcs are never made positive.

Here, we will describe each of these algorithms briefly, and present the full versions of the NEG-CYCLE and CRI-CYCLE algorithms. The full version of the PRI-CYCLE algorithm is given in Appendix A.3.

The NEG-CYCLE algorithm

The NEG-CYCLE\(^2\) algorithm, shown in Algorithm 1, is the first algorithm described by Dasdan. This algorithm iterates until the distance graph \(G\) is free of negative cycles. At each iteration it selects a negative cycle, using Tarjan’s negative cycle detection algorithm [4, 25]. The full version of Tarjan’s algorithm is given in his original publications, Dasdan used a different, yet equivalent formulation of the same problem. In his version, all arcs in the graph are inversed and their weights are negated. Therefore, in the original versions of the algorithms the objective was to remove all positive cycles from the graph.

\(^2\)Originally this algorithm was called POS-CYCLE, as its goal was to remove all positive cycles from the graph. Since here we want to remove all negative cycles, we have renamed it to NEG-CYCLE.
given in Appendix A.4. Tarjan’s algorithm runs in $O(nm)$ time and returns a negative cycle, if there exists one in the graph. From this cycle, the NEG-CYCLE algorithm selects negative arcs and sets their weight to 0, until the weight of the cycle has become non-negative. This is repeated until there are no more negative cycles left in the graph. At the end, the algorithm returns the list $L$ of arcs whose weight has to be changed.

Algorithm 1: The NEG-CYCLE algorithm (NC).

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$L = \emptyset$;</td>
</tr>
<tr>
<td>2</td>
<td>repeat</td>
</tr>
<tr>
<td>3</td>
<td>$\langle \text{found}, C \rangle = \text{Tarjan}(G)$;</td>
</tr>
<tr>
<td>4</td>
<td>if (found) then</td>
</tr>
<tr>
<td>5</td>
<td>rem = $w(C)$;</td>
</tr>
<tr>
<td>6</td>
<td>while (rem &lt; 0) do</td>
</tr>
<tr>
<td>7</td>
<td>Select one arc $a$ on $C$ with $w(a) &lt; 0$;</td>
</tr>
<tr>
<td>8</td>
<td>rem = min(0, rem - $w(a)$);</td>
</tr>
<tr>
<td>9</td>
<td>$w(a) = 0$;</td>
</tr>
<tr>
<td>10</td>
<td>$L = L \cup {a}$;</td>
</tr>
<tr>
<td>11</td>
<td>until (not found) ;</td>
</tr>
<tr>
<td>12</td>
<td>return $L$;</td>
</tr>
</tbody>
</table>

The NEG-CYCLE algorithm runs in $O(nm)$ time if the input graph $G$ is consistent, because then Tarjan’s algorithm has to be executed only once. If $G$ is inconsistent, then the negative cycles have to be removed by changing the weight of arcs on the cycle to zero. Since there are $m$ arcs in the graph, the algorithm iterates at most $m$ times. In each of these iterations, Tarjan’s algorithm is executed. Thus, the time complexity of the NEG-CYCLE algorithm is $O(nm^2)$.

The CRI-CYCLE algorithm

The NEG-CYCLE algorithm has two weaknesses: (i) it selects negative cycles in an arbitrary order, and (ii) to ensure a strongly polynomial runtime, it always needs to change the weight of an arc into zero. This may lead to changing the weight of an arc by an amount that is much larger than is needed for zeroing the weight of the cycle. The CRI-CYCLE algorithm does not have these weaknesses.

Whereas the NEG-CYCLE algorithm selects negative cycles in an arbitrary order, the CRI-CYCLE algorithm (shown in Algorithm 2) selects them in a more sophisticated way, using the property of cycle means. The cycle mean $\lambda(C)$ of a cycle $C$ in $G$ is the average weight of the arcs in $C$ (i.e., $\lambda(C) = w(C)/|C|$). The minimum cycle mean $\lambda^*(G)$ of a graph $G$ is the minimum value of $\lambda(C)$ for all cycles $C$ in $G$. The CRI-CYCLE algorithm selects negative cycles in a way that minimizes the number of arcs whose weight needs to be changed, by selecting cycles with minimum cycle mean.
Algorithm 2: The CRI-CYCLE algorithm (CC).

**input**: A distance graph \( G = (V, A) \)

**output**: A list \( L \) of arcs whose weight has to be changed

1. \( L = \emptyset \);
2. repeat
3. \( \langle \lambda^*(G), C \rangle = \text{YTO}(G) \);
4. if \( \lambda^*(G) < 0 \) then
5. \( \text{rem} = w(C) \);
6. while \( \text{rem} < 0 \) do
7. Select one arc \( a \) on \( C \) with \( w(a) < 0 \);
8. if \( \text{rem} < w(a) \) then
9. \( \text{rem} = \text{rem} - w(a) \);
10. \( w(a) = 0 \);
11. else
12. \( w(a) = w(a) - \text{rem} \);
13. \( \text{rem} = 0 \);
14. \( L = L \cup \{a\} \);
15. until \( \lambda^*(G) \geq 0 \);
16. return \( L \);

cycles \( C \) in \( G \). A cycle whose mean is equal to the minimum cycle mean is called a *critical cycle*. A critical cycle of \( G \) can be found in \( O(nm) \) time, for example by using the Young-Tarjan-Orlin algorithm (YTO) [8, 28] algorithm\(^3\). The full version of the YTO algorithm can be found in Appendix A.5. Since \( G \) has at least one negative cycle if and only if \( \lambda^*(G) < 0 \), this algorithm can also be used to simultaneously check the consistency of the network.

The CRI-CYCLE algorithm iterates until the distance graph \( G \) is free of negative cycles. At each iteration it selects a critical cycle \( C \) from \( G \), using the YTO algorithm. From this critical cycle, it selects negative arcs \( a \) and changes their weight, until the weight of the cycle has become zero. The weight of these arcs is set to 0, if the (remaining) weight of the cycle is less than the weight of the arc, or to \( w(a) \) – the remaining weight of the cycle, otherwise. This is repeated until \( \lambda^*(G) \geq 0 \). At the end, the algorithm returns the list \( L \) of arcs whose weight has to be changed.

\(^3\)In Dasdan’s implementation, the Young-Tarjan-Orlin algorithm (YTO) [8, 28] algorithm is used for finding critical cycles. Actually this algorithm has a (in some cases) slightly higher worst-case time complexity of \( O(nm + n^2 \log n) \). However, it turned out to perform better in practice than critical-cycle algorithms that do have a time complexity of \( O(nm) \) [8]. In the remainder of this document, the YTO algorithm will also be used for finding critical cycles. For computation of the time complexity of the algorithms in which it is used, we assume that finding a critical cycle costs \( O(nm) \) time. The YTO algorithm can be replaced by any other critical-cycle algorithm that indeed has an \( O(nm) \) time complexity, to obtain this complexity bound.
Since the algorithm chooses a critical cycle every time (instead of an arbitrary one), a strongly polynomial runtime can be guaranteed without the need of always zeroing the weight of an arc. A drawback of this, however, is that the same arc may be chosen multiple times, until a maximum of \( n \) times. For this reason, the CRI-CYCLE algorithm has a worse time complexity than the NEG-CYCLE algorithm. To see why this is the case, note that, given an arc \( a \) on a critical cycle \( C \), then for every other negative cycle \( C' \) on \( a \) where \( |C'| \leq |C| \), subtracting \( w(C) = |C| \lambda^*(G) \) from \( w(a) \) makes \( C' \) non-negative. Since the maximum number of arcs of a cycle \( C \) in \( G \) is \( n \), this can happen at most \( n \) times for a single arc. Thus, for \( m \) arcs this can happen at most \( nm \) times. If the remaining weight of the cycle \( \text{rem} \) is less than the weight of the arc \( w(a) \), then \( w(a) \) is set to zero and the arc will never be chosen again. This can happen at most \( m \) times, since there are \( m \) arcs. Thus, the worst-case scenario occurs if the algorithm always chooses an arc whose weight is larger than the remaining weight of the cycle (since then the arc may be chosen again in a later iteration). As mentioned before, this can happen at most \( nm \) times. Thus, the main loop of CRI-CYCLE iterates at most \( nm \) times. Since each iteration takes \( O(nm) \) time for finding a critical cycle, the CRI-CYCLE algorithm runs in \( O(nm) \) time if the input graph \( G \) is consistent, and in \( O(n^2m^2) \) time if \( G \) is inconsistent.

The PRI-CYCLE algorithm

The PRI-CYCLE algorithm is equal to the CRI-CYCLE algorithm, but it combines the notions of weight and priority in its procedure to select the arcs whose weights should be changed. In the PRI-CYCLE algorithm, apart from the weight \( w \), also a priority \( \pi \) is associated with each arc in the graph. A priority is a positive integer value, \( 1 \leq \pi \leq \pi_{\text{max}} \). The smaller the number, the higher the priority, i.e., a higher priority arc should be considered for a weight change before a lower priority arc.

Notice, that there is a difference between the priorities used in this algorithm, and the preferences of STPPs (described in Section 2.3). In contrast to the preference values we have seen before, here only one priority value is assigned to each arc, instead of an entire preference function that is assigned to each constraint. The priority values denote an ordering between different arcs, which indicates which arcs should be changed first, and which should rather keep their current weight. They do not contain any information about the amount by which the weights of these arcs should be changed. Preference values contain information about different values a constraint may take, and how preferred each of these values is. However, they do not explicitly register an ordering between the different arcs.

Instead of cycle means, the PRI-CYCLE algorithm uses cycle ratios to select a cycle that should be zeroed. The cycle ratio \( \rho(C) \) of a cycle \( C \) in \( G \) is defined as \( \rho(C) = w(C)/\pi(C) \). Here, the weight \( w(C) \) of the cycle is the sum of the weights of the arcs in the cycle, and the priority \( \pi(C) \) of the cycle is the sum of the priorities of the arcs in the cycle. The minimum cycle ratio \( \rho^*(G) \) of a graph \( G \) is the minimum value of \( \rho(C) \) for all cycles \( C \) in \( G \). A cycle whose ratio is equal to
the minimum cycle ratio is called a critical cycle, and can also be found using the YTO algorithm [8, 28].

The time complexity analysis of the PRI-CYCLE algorithm is very similar to that of the CRI-CYCLE algorithm. Given an arc \( a \) on a critical cycle \( C \), then for every other negative cycle \( C' \) on \( a \) where \( \pi(C') \leq \pi(C) \), subtracting \( w(C) = \pi(C)\rho^*(G) \) from \( w(a) \) makes \( C' \) non-negative. The worst case occurs when all cycles are visited in the order of their decreasing priority. Since there are \( \pi_{\text{max}} \) priority levels for an arc, and a cycle can have at most \( n \) arcs, the priority of a cycle can be at most \( \pi_{\text{max}} n \). Thus, a single arc can be chosen at most \( \pi_{\text{max}} n \) times.\(^4\) For \( m \) arcs this can happen at most \( \pi_{\text{max}} nm \) times, so the main loop of the algorithm iterates at most \( \pi_{\text{max}} nm \) times. Since each iteration takes \( O(nm) \) time, the PRI-CYCLE algorithm runs in \( O(nm) \) time if the input graph \( G \) is consistent, and in \( O(\pi_{\text{max}} n^2 m^2) \) time if \( G \) is inconsistent. Without loss of generality, we can assume that \( \pi_{\text{max}} = O(m) \) (because \( m \) priority values are sufficient to distinguish \( m \) arcs by priority). Assuming that \( \pi_{\text{max}} = O(m) \), the time complexity of the PRI-CYCLE algorithm is \( O(n^2 m^3) \).

2.5.2 Plan repair using preferences

In this section, we describe existing plan repair techniques that use the preference information in an STP with preferences. Buzing, ter Mors and Witteveen [2] have described how an STPP can be applied in plan repair. They assume a finite number of preference levels. They state that plans at preference level \( p < p_{\text{max}} \) are in fact a relaxation of the optimal plan. This property may be used to adapt an existing plan if during the actual execution phase unexpected events occur, that lead to inconsistency of the plan that is being executed. If a constraint is changed during plan execution, it may be possible to repair the plan in constant time by simple returning to a lower preference level. For this it is necessary to store all results of the binary search, for which additional space of \( O(\log |P|) \) is needed, where \( P \) is the set of preference levels. Repairing an STP that has become inconsistent can then be done in \( O(\log |P|) \) time. Should the intermediate results not be stored, then they have to be computed during the search, in which case repairing the STP takes \( O(\log(|P|) \times n^3) \) time.

2.6 Discussion of existing approaches

In this section, we discuss the known approaches for solving disruptions in STPs and STPPs, described in Section 2.5. We conclude with a motivation for our own research.

\(^4\)In [9], Dasdan claims that the priorities of a cycle can range from 1 to \( \pi_{\text{max}} \). However, we believe that this is incorrect. If the priorities of an arc can range from 1 to \( \pi_{\text{max}} \), and there can be at most \( n \) arcs in a cycle, then the priority of a cycle can be at most \( \pi_{\text{max}} n \). Thus, each arc can be chosen at most \( \pi_{\text{max}} n \) times, instead of only \( \pi_{\text{max}} \) times. Therefore, the claims in our proof differ by a factor \( n \) from Dasdan’s claims.
2.6.1 Evaluation of algorithms for removing negative cycles

We first evaluate Dasdan’s algorithms for removing negative cycles from the distance graph – i.e., the NEG-CYCLE, CRI-CYCLE and PRI-CYCLE algorithms, which we described in Section 2.5.1. There are some flaws in Dasdan’s approach – at least when his models are seen in the context of temporal planning. Here, first the meaning of Dasdan’s additional requirements will be explained, as seen in the context of STPs. Then, a short overview will be given of the points at which his algorithms can be improved.

Meaning of Dasdan’s additional requirements in STPs

As mentioned in Section 2.1, a constraint \( c_{ij} = [lb_{ij}, ub_{ij}] \) between the two variables \( x_i \) and \( x_j \) can be represented in the distance graph by two arcs \( x_i \xrightarrow{ub_{ij}} x_j \) and \( x_j \xleftarrow{-lb_{ij}} x_i \). This means that (in general) in the distance graph, all arcs with negative weights represent the lower bounds of an interval, and all arcs with positive weights represent the upper bounds of an interval.

Dasdan has imposed the following two requirements upon his algorithms:

1. Only the weights of negative arcs are changed.
2. Negative arcs are never made positive.

In the context of STPs, changing only the weights of negative arcs means that only the lower bounds of the intervals are extended. If you associate a constraint with the duration of a task, it means that the task is allowed to last shorter than originally planned. Thus, making a negative arc less negative corresponds with speeding up a task. Changing the weight of an arc into zero means that the task can have a zero-length duration. However, it can never be the case that an interval is extended.

The requirement that negative arcs are never made positive, implies that the precedence relations between all tasks are preserved. Or more precisely, a zero-length duration means that both time points can also coincide. But if in the original problem, \( x_j \) should occur after \( x_i \), it can never happen that in the new situation \( x_j \) precedes \( x_i \).

Points for improvement of Dasdan’s algorithms

Here, the most notable points at which Dasdan’s algorithms can improved be will be listed.

- *It is not realistic to change only negative weight arcs.*

It is not realistic to speed up all tasks, and especially not to have no limits on the amount by which the duration of a task can be decreased. This is what you are actually doing when you make negative weight arcs less negative:
you lower the lower bounds of the intervals. And especially demanding from a task that it should have a zero-length duration cannot be achieved in any realistic scenario. Also, there is no universal rule that forbids any interval to be extended. Therefore, there is no reason why only the weights of arcs with a negative weight may be changed.

- **It makes no sense to divide weights by priorities.**

You cannot just divide the weight of the arcs by their priorities (as Dasdan does in the PRI-CYCLE algorithm), and still get any meaningful value. E.g., suppose you have a task with a duration of 24 minutes and a priority of 6. This task has a weight/priority ratio of 24/6 = 4. What happens if you split this task into two tasks, each with a duration of 12 minutes? If you still want them to have the same priority of 6, you suddenly get a weight/priority ratio of (12+12)/(6+6) = 2. If, on the other hand, you want to keep the same ratio, their priorities would suddenly become 3 instead of 6. Neither one makes sense.

- **The priorities of blacklisted arcs should not be able to influence the arc selection procedure.**

The PRI-CYCLE algorithm does not take into account the fact that all positive arcs are already blacklisted beforehand, by allowing only negative arcs to change. Since this is the case, their priorities should not be able to influence the cycle ratios (which at this moment they do). It is not logical to allow the priority of a positive arc to make the difference between the selection of which negative arc should be changed.

- **The algorithms do not have to return a minimal set.**

The algorithms return a set of constraints of which changing their weights will resolve the inconsistency. However, there are no guarantees for the quality of the solutions found. It may be possible, that the algorithms return a solution that is not a minimal set. E.g., in the graph of Figure 2.7 it is possible that the algorithms first resolve the negative cycle $A - B - C - A$ by changing the weight of arc $AB$, and then resolve the negative cycle $A - D - C - A$ by changing the weight of arc $CA$. Then, the solution found by the algorithms will consist of the set $\{AB, CA\}$, whereas changing only arc $CA$ would be enough to solve the problem. Thus, the arc $AB$ is redundant in this set.

**Example 2.5** When, at 01:30 PM, Alice is finally ready to take the train to Amsterdam, she expects that she will arrive at her hotel at 03:00 PM at the earliest. In Example 2.2, we saw that her planning problem has become inconsistent, which manifested itself in the negative cycle $x_0 - x_4 - x_3 - x_2 - x_0$ of weight $525 - 120 - 180 - 315 = -90$ in the distance graph. This situation is shown in Figure 2.8. The negative cycle is indicated in bold.
Figure 2.8: A negative cycle in the distance graph indicates an inconsistency in Alice’s plan. Dasdan’s algorithms will never consider a change of arc $x_0 \rightarrow x_4$, because its weight is positive.

Figure 2.9: In the PRI-CYCLE algorithm, the priorities of the blacklisted positive arcs influence the cycle ratios, and thus the choice of which negative arc should be changed first.

**Dasdan’s algorithms will solve this inconsistency by eliminating one of the negative arcs. This means that either Alice’s journey to Amsterdam, her rest break, or her exploration of the city will be cancelled. However, even though it is a valid option in reality, the algorithms will never consider the possibility of postponing her dinner, because this constraint is expressed by an arc with a positive weight.**

**Example 2.6** The graph in Figure 2.9 has two simple cycles. The cycle $A - B - A$ has a cycle ratio of $(1 - 2)/(1 + 2) = -1/3$. The cycle $B - C - B$ has a cycle ratio of $(1 - 2)/(3 + 1) = -1/4$. Thus, the cycle $A - B - A$ is the only critical cycle, and will be repaired first. Since the arc $AB$ has a positive weight, it cannot be selected for a weight change. This means that the weight of arc $BA$ will be changed first, even though the arc $CB$ has a higher priority.

Thus, the most notable points at which Dasdan’s algorithms can be improved are the way in which certain arcs are blacklisted, the way in which priority information is applied, and the quality of the solution returned by the algorithms.

### 2.6.2 Evaluation of plan repair using preferences

The idea of Buzing, ter Mors and Witteveen is to recover from a disruption by letting the entire STP revert to a lower preference level. The advantage of this approach is that repair can be done very fast. The disadvantage, on the other hand, is
that possibly the preference level is lowered of many more constraints than needed to solve the disruption. In the worst case, it may be enough to lower only a single constraint. Instead the preference level of all constraints is lowered, which is an overhead of \( m - 1 \) constraints.

### 2.6.3 Motivation for our research

There has not been much research yet on algorithms for freeing a distance graph of all negative cycles. And those few algorithms that do exist, have a lot of flaws – at least when seen in the context of temporal planning. The main points at which existing algorithms can be improved, are the criteria for choosing the arcs of which the weights may be changed, and the amount by which their weights may be changed. Furthermore, it would be nice if one could give guarantees about the minimality of the solutions.

Since the situation may be different for each individual planning problem, it is not possible to formulate one universal rule that dictates which changes are allowed, and which are not. Therefore, additional information is needed to be able to make better substantiated choices. This is possible by adding preference information to the model.

An existing approach that uses preference information for plan repair, requires that all constraints in the graph are changed, in order to solve a disruption. This will result in a large overhead, if changing only a small subset of constraints is enough to resolve the inconsistency. Thus, it would be interesting to investigate whether it is possible to repair the STP by lowering the preference levels of only a subset of constraints.

In our research, we will investigate how to repair an STP with preference information. Our goal is to find a minimal set of constraints that have to be changed to remove all negative cycles from the graph. We aim at improving existing negative cycle elimination algorithms, by using a model that allows us to make better substantiated choices, while ensuring that the set of arcs that has to be changed is minimal.

Another interesting topic, which has not been investigated yet, is the following problem. In many situations, when a disruption occurs, there is more information available than just the inconsistent network. Often, there used to be a consistent plan, that has become inconsistent due to a change in one of the constraints. In such a situation, the plan has to be adapted to incorporate this change. Also here, we would like to retain as much of our original plan as possible. Thus, we would also like to find a minimum set of constraints that has to be changed, in order to remove all negative cycles from the graph. This time, however, there is the additional restriction that the constraint that has just caused the inconsistency should not be changed.

We will investigate how, with minor modifications, the existing algorithms and our algorithms for freeing regular STPs of all negative cycles, can be applied to find a solution to this problem. Furthermore, we will investigate how one can apply the
additional information about the previous, consistent situation and the source of the inconsistency, to guide the search for a solution and reduce the search space.
Chapter 3

New algorithms for resolving disruptions in STPs

In this chapter, we discuss new algorithms for resolving disruptions in STPs. Our goal is to find a minimal subset of arcs that have to be changed, in order to remove all negative cycles from the distance graph. We start with the basic case, in which we are allowed to simply remove any arc from the graph. After that we extend the STP model with preference information. This allows us to take into account the amount by which the weight of each arc may be changed. First, we describe the case in which there are two preference levels. Then, we extend the model by allowing multiple preference levels. Finally, we investigate the situation in which one constraint is known to be the source of the inconsistency.

3.1 Basic case

Before we extend the STP model with preference information, we will first have a look at how regular STPs can be repaired. In this section, we discuss algorithms for finding a minimal subset of arcs that have to be removed from the distance graph, in order to make the graph free of negative cycles. We investigate the basic case, in which we assume that all constraints are equally important, and that a constraint is either left in the graph, or removed from it completely. Thus, it is not possible to only change its weight.

When trying to find a minimal subset of arcs that have to be removed from the distance graph, to free the graph of negative cycles, two main approaches are possible: a top-down approach, and a bottom-up approach. In the top-down approach, we start with the inconsistent graph that has to be repaired. From this graph, we iteratively remove arcs until there are no negative cycles left. This is very similar to the way Dasdan’s algorithms work. In the bottom-up approach, we start with an “empty” graph, containing only the vertices of the graph that has to be repaired, but no arcs. To this graph we add the arcs one at a time, unless this would lead to an inconsistency. The arcs that cannot be added without yielding an inconsistency,
Algorithm 3: The top-down algorithm for the basic case (TD).

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$L = \emptyset$;</td>
</tr>
<tr>
<td>2</td>
<td>repeat</td>
</tr>
<tr>
<td>3</td>
<td>$(\text{found}, C) = \text{Tarjan}(G)$;</td>
</tr>
<tr>
<td>4</td>
<td>if (\text{found}) then</td>
</tr>
<tr>
<td>5</td>
<td>Select an arc $a$ on $C$;</td>
</tr>
<tr>
<td>6</td>
<td>$L = L \cup {a}$;</td>
</tr>
<tr>
<td>7</td>
<td>Remove $a$ from $G$;</td>
</tr>
<tr>
<td>8</td>
<td>until (not found)</td>
</tr>
<tr>
<td>9</td>
<td>return $L$;</td>
</tr>
</tbody>
</table>

are the arcs that should be removed from the original inconsistent graph.

3.1.1 Algorithms

In this section, we will discuss algorithms for the top-down approach and for the bottom-up approach.

Top-down algorithm

Our top-down algorithm is a variation of the NEG-CYCLE algorithm. The algorithm iterates until the distance graph $G$ is free of negative cycles. At each iteration, it selects a negative cycle $C$ from $G$, using Tarjan’s algorithm. From this cycle, it selects an arc and removes this arc from the graph, thereby breaking the negative cycle. Then, it continues with the next negative cycle. This is repeated until the graph is free of negative cycles. At the end, the algorithm returns the list $L$ of arcs that have to be removed from the graph, in order to free the graph of negative cycles. The entire algorithm is shown in Algorithm 3.

To prove the complexity of this algorithm, note that Tarjan’s algorithm finds a negative cycle in $O(nm)$ time. For each cycle returned by Tarjan’s algorithm, one arc $a$ is removed from the graph. Since there are $m$ arcs in the graph, this can happen at most $m$ times. Thus, the time complexity of the entire algorithm is $O(nm^2)$.

Algorithm 3 has one drawback. It may be possible that after the algorithm has ended, the list $L$ found by the algorithm contains redundant arcs, which could be returned to the graph again without causing an inconsistency. This is the case if the algorithm first removes one arc $a_1$ to eliminate a negative cycle $C_1$, and then removes another arc $a_2$ that is also on $C_1$, to eliminate another negative cycle $C_2$. In that case, it might have been sufficient to remove only arc $a_2$, since the removal of that arc would eliminate both $C_1$ and $C_2$. Thus, although the removal of all arcs
Algorithm 4: The bottom-up algorithm for the basic case (BU).

**input**: A distance graph \( G = (V, A) \).

**output**: A list \( L \) of arcs that has been removed from the graph.

1. \( G' = (V, A') \);
2. foreach \( (a'_{ij} \in A') \) do
3.   if \( (i == j) \) then
4.     \( w(a'_{ij}) = 0; \)
5.   else
6.     \( w(a'_{ij}) = \infty; \)
7. \( L = \emptyset; \)
8. foreach \( (a_{ij} \in A) \) do
9.   if \( (w(a_{ij}) + w(a'_{ji}) \geq 0) \) then
10.      IFPC\( (G', a_{ij}) \);
11.   else
12.      \( L = L \cup a_{ij}; \)
13. \( G = G' \);
14. return \( L \);

in \( L \) does free the graph of negative cycles, the set of arcs found by the algorithm does not have to be minimal. To obtain a minimal solution, a post-processing phase would be necessary, in which for all arcs in \( L \) the algorithm would check whether they could be returned to the graph again. Such post-processing phase would be very similar to the way the bottom-up approach works, which will be discussed next.

**Bottom-up algorithm**

The bottom-up algorithm works the other way around. It starts with a graph \( G' \) that contains only the vertices of the inconsistent graph \( G \), but not its arcs. During the entire algorithm, \( G' \) will represent a minimal network. For each arc \( a \) in \( G \), the algorithm checks whether it is possible to add this arc to \( G' \). If this is possible without yielding an inconsistency, arc \( a \) is added to \( G' \). This can be done using the Incremental Full Path Consistency (IFPC) algorithm [20] (see Appendix A.2). At the end, the algorithm returns the list \( L \) of arcs that could not be added to \( G' \), and thus should be removed from \( G \) to free the graph of negative cycles. The entire algorithm is shown in Algorithm 4.

Initialising the distances in \( G' \) costs \( O(n^2) \) time. Since the graph \( G' \) represents a minimal network, checking whether an arc \( a \) can be added to it without yielding an inconsistency, can be done in constant time. The IFPC algorithm runs in \( O(n^2) \) time. Since there are \( m \) arcs in \( A \), and this has to be done for each arc, the main loop of the algorithm iterates \( m \) times. Thus, the run time complexity of Algorithm
Since every arc $a$ in $A$ is checked, and added to $G'$ if this is possible, the list $L$ returned by the algorithm will only contain arcs that would yield an inconsistency if added to $G'$. Thus, the list $L$ returned by the algorithm does not contain any redundant elements. This means that, unlike the top-down algorithm, the bottom-up algorithm is guaranteed to return a minimal set.

### 3.2 Two preference levels

Removing constraints from the graph completely is the fastest way to break a negative cycle, and may be nice in theory. In reality, however, we always have to keep in mind that these constraints have a meaning. This means that usually we cannot simply remove any constraint from the graph and still be guaranteed to result in a plan that is feasible in practice. Thus, we need a way to express which constraints may be changed, and by which amount they may be changed. This can be achieved by adding the notion of preferences to the model. In this section, we will show how the algorithms described in Section 3.1 can be adapted, such that they can be used to repair an STP with two preference levels. We will first describe the framework that we use, and then discuss the algorithms.

#### 3.2.1 Framework

In this section, we extend the STP model with preference information. There are two preference levels $p$, which are denoted by the integer values 1 and 2. Here, 1 is the lowest preference level and 2 is the highest preference level. Solutions at level 2 are more preferred than solutions at level 1.

For each arc $a$, two weights are available. We use $w_i(a)$ to denote the weight of arc $a$ at preference level $i$. We assume that, for each arc $a$, $w_1(a) \geq w_2(a)$. Thus, the constraints at level 1 can be seen as a relaxation of the constraints at level 2. All solutions at level 2 are also solutions at level 1, but level 1 may allow more solutions than level 2.

At each moment we associate a preference level with each arc. The arcs at preference level 1 will be referred to as hard arcs, and arcs at preference level 2 will be referred to as soft arcs. The current preference level of $a$ will be denoted by $p(a)$. The current weight of $a$ will be denoted by $w(a)$. Thus, if $p(a) = i$, then $w(a) = w_i(a)$. In our solutions, we will consider each arc in the distance graph separately. Thus, it may be possible that we assign a different preference level to the arc that represents the lower bound of an interval associated with a constraint, than to the arc that represents the upper bound of the same interval.

We assume that in the original, inconsistent situation, all arcs have preference level 2. Furthermore, we assume that the STP is consistent at preference level $p = 1$. This means that the STP is consistent if all arcs $a$ are assigned the weight $w_1(a)$. This means that we are always able to find a solution to our problem by lowering
the preference level of one or more constraints. The goal of our algorithms is to find a minimal subset of arcs, whose preference level has to be decreased to level 1, in order to remove all negative cycles from the graph. Should the STP even be inconsistent at level $p = 1$, then it requires domain specific knowledge to solve the problem. This falls outside the scope of our work.

### 3.2.2 Algorithms

The idea of our algorithms for solving disruptions in an STP with two preference levels is fairly straightforward. Instead of removing an arc from the graph completely, we now decrease its preference level from 2 to 1. Since we know that the STP is consistent at preference level 1, this will be sufficient for repairing the STP.

**Top-down algorithm**

For the top-down algorithm, this means that line 7 of Algorithm 3 is replaced by “$p(a) = 1$,” (which results in $w(a)$ being set to $w_1(a)$). With this change, the algorithm will be able to repair the STP with two preference levels in $O(nm^2)$ time. However, it also possible to optimise the algorithm a little further. When decreasing the preference level of this single arc, there is no guarantee that the negative cycle that was found is no longer negative. If after changing the preference level of this arc the cycle is still negative, we can select a different arc on the same cycle immediately, instead of running the TARJAN algorithm again to look for another negative cycle. This is similar to the way Dasdan’s NEG-CYCLE algorithm works. Even though with this change the time complexity of the algorithm is still $O(nm^2)$, in many cases it will run faster than before, since checking whether the cycle is still negative can be done much faster than finding a new negative cycle.

**Bottom-up algorithm**

For the bottom-up algorithm, this means that we start with computing the minimal network for a graph $G'$ in which all arcs are set to preference level 1. Then for each arc we increase its preference level to 2, if this does not lead to an inconsistency, or we add the arc to the list $L$ of arcs whose preference level should be decreased, otherwise. Even though this algorithm is very similar to the bottom-up algorithm for the basic case (without preference levels), it has a different time complexity. This is due to the fact that we have to initialise the distances in the graph $G'$. Where we could simply initialise all distances with $\infty$ in $O(n^2)$ time in the previous version, we now have to run an all-pairs-shortest-paths algorithm. This takes $O(n^3)$ time if the Floyd-Warshall algorithm is used.\(^1\) After the initialisation of these distances, the algorithm also takes $O(n^2m)$ time to check for each arc whether its prefer-

---

\(^1\)Subcubic APSP algorithms also exist. At this moment the fastest known subcubic APSP algorithm has a running time approximating $O(n^3/\log^2 n)$[3], which may still be worse than $O(n^2m)$. 

29
ence level can be increased. Thus, the total time complexity of the algorithm is $O(n^3 + n^2m)$.

**Top-down algorithm with post-processing**

Algorithm 5 shows the top-down algorithm with a bottom-up post-processing phase for repairing an STP with two preference levels. The algorithm is built up from the top-down algorithm and the bottom-up algorithm described above. It consists of two phases. In the first top-down phase, it iterates until the distance graph $G$ is free of negative cycles. At each iteration, it selects a negative cycle $C$ from $G$, using Tarjan’s algorithm. From this cycle, it selects a soft arc and changes it into a hard arc. Thus, the preference level of this arc is set from 2 to 1. It continues selecting soft arcs on this cycle and changing them into hard arcs, until the weight of the cycle has become non-negative. Then it continues with the next negative cycle. This is repeated until the graph is free of negative cycles. However, the list $L$ may contain more arcs than needed to achieve this goal. Therefore, the second bottom-up phase is a post-processing phase, in which for each arc $a$ in $L$, the algorithm tries to revert the preference level of the arc to 2. If this is possible without yielding an inconsistency, this is done and the arc $a$ is removed from the list $L$ again. At the end, the algorithm returns the list $L$ of arcs that have to be changed from soft into hard arcs.

Since we know that the STP is consistent at preference level $p = 1$, it is always possible to make a negative cycle non-negative by changing soft arcs into hard arcs. Also, we know that there is a finite number of soft arcs that can be changed into hard arcs. Since at each step we decrease this number by 1, the algorithm is guaranteed to terminate and will always find a solution.

To prove the complexity of this algorithm, note that Tarjan’s algorithm finds a negative cycle in $O(nm)$ time. From each cycle returned by Tarjan’s algorithm, at least one soft arc is changed into a hard arc. Since there are at most $m$ soft arcs, there are at most $m$ iterations needed to remove all negative cycles. Thus, the first phase of the algorithm takes $O(nm^2)$ time. Then the $O(n^3)$ Floyd-Warshall algorithm is used to compute the minimal network. In the second phase, for each arc $a$ in $L$ the $O(n^2)$ IFPC algorithm is applied. Since there are at most $m$ arcs in $L$, this phase takes at most $O(n^2m)$ time. Thus, the total time complexity of the algorithm is $O(nm^2 + n^3 + n^2m)$.

### 3.2.3 Discussion

This time bound seems a lot worse than the complexity of only the bottom-up approach, whereas both find a minimal solution. Thus, one could wonder why to consider this algorithm, instead of simply using the bottom-up algorithm. However, these complexity bounds are based on worst case situations. In practice, the running time of the algorithm largely depends on the amount of arcs found by the algorithm. E.g., suppose that in the top-down phase $k$ arcs are found that have to
Algorithm 5: The top-down algorithm with post-processing for repairing an STP with two preference levels.

**Input**: A distance graph $G = (V, A)$ with 2 preference levels.

**Output**: A list $L$ of arcs whose weight has to be changed

1. $L = \emptyset$
2. repeat
   3. $(\text{found, } C) = \text{Tarjan}(G)$;
   4. if (found) then
      5. $\text{rem} = w(C)$;
      6. while ($\text{rem} < 0$) do
         7. Select an arc $a$ on $C$ with $p(a) = 2$;
         8. $\text{rem} = \text{rem} - w_2(a) + w_1(a)$;
         9. $p(a) = 1$;
      10. $L = L \cup \{a\}$;
   11. until (not found);
12. $G' = \text{Floyd-Warshall}(G)$;
13. foreach $(a_{ij} \in L)$ do
   14. if ($w_2(a_{ij}) + w(a_{ji}') \geq 0$) then
      15. $p(a) = 2$;
      16. Add $w_2(a_{ij})$ to $G'$ using IFPC;
      17. $L = L \setminus \{a\}$;
18. return $L$;

be changed, then main loop of the top-down phase will iterate $k + 1$ times, and thus take $(k + 1) \times O(nm)$ time. Then the list $L$ contains $k$ arcs, so the main loop of the bottom-up phase will iterate $k$ times, which takes $k \times O(n^2)$ time. Thus, if $k$ is very small compared to the input size, and may be ignored in our complexity analysis, then the running time will be $O(nm + n^3 + n^2)$ (or even less if a subcubic APSP algorithm is used). On the other hand, the bottom-up algorithm always has to consider all arcs, so it will always take $O(n^2m)$ time. Thus, for small values of $k$ and large values of $m$, a top-down algorithm with a bottom-up post-processing phase may be faster.

### 3.3 Multiple preference levels

An STP with two preference levels can be used to indicate which constraints can be changed, and by which amount they can be changed, such that it will always be possible to execute the resulting plans in practice. However, the expressiveness of a model with only two preference levels is still very limited. Often, it is possible to nuance the gradation of how preferred certain solutions are. Some solutions
are most preferred, others are a bit less preferred, and still others may also be feasible, but are actually not preferred at all. In this section, we extend the model by allowing multiple preference levels, to increase its expressiveness. We present two algorithms for repairing an STP with multiple preference levels. Furthermore, we show how our approach can be used for solving an STP with priority information.

3.3.1 Framework

In this section, we extend the framework of Section 3.2 to include multiple preference levels. Here, there is a finite number of preference levels $p$. The set of all preference levels is indicated by $P$, where $P$ is the set of integer values in the range $[1, p_{\text{max}}]$. $1$ is the lowest preference level and $p_{\text{max}}$ is the highest preference level. The lower the preference level, the harder the constraint. The constraints at level $p = 1$ are the hardest constraints, and should not be changed at all. The number of preference levels is the same for all arcs. Also in this section, we assume that the STP is consistent at preference level $p = 1$.

A semi-convex preference function is associated with each constraint: $c_{ij} = \langle I_{ij}, f_{ij} \rangle$, where $f_{ij} : x \in I_{ij} \to P$. This means that at a higher preference level, the interval is narrowed down to a single, possibly smaller subinterval. The largest weight that an arc $a$ can have at preference level $p$ will be denoted by $w_p(a)$.

At each moment we associate a preference level with each arc. The current preference level of $a$ will be denoted by $p(a)$. The current weight of $a$ will be denoted by $w(a)$. If $p(a) = 1$, then $w(a) = w_1(a)$. Also here, we will consider each arc in the distance graph separately. Since all solutions at a higher preference level are also valid solutions at a lower preference level, and all preference functions are semi-convex, each combination of an upper and a lower bound of an interval will always form a single interval as well, irrespective of the preference level that is chosen for each arc.

3.3.2 Goal of our algorithms

Here, we discuss the goal of our algorithms for repairing an STP with multiple preference levels. In general, our objective is to find a subset of arcs, whose preference level has to be decreased, and the amount by which the preference level of each arc has to be decreased, such that all negative cycles will be removed from the graph. We aim to find a solution, such that it is not possible to decrease the preference level of any arc by a smaller amount, without leading to an inconsistency. The question now is, which approach should be taken for choosing the arcs whose preference levels should be decreased, and the amount by which those preference levels should be decreased.

With two preference levels, the goal of our algorithms was to find a minimal subset of arcs, whose preference level had to be decreased, in order to remove all negative cycles from the graph. There was a clear distinction between the arcs whose preference level could still be lowered, and the arcs that could not be changed any-
more. Thus, it was immediately evident whether an arc was eligible for inclusion in the subset of arcs that should be changed or not.

With multiple preference levels, however, there are various possible strategies for choosing the arcs whose preference level should be lowered. One can make a trade-off between lowering the preference level of arcs that have a high preference level, and further lowering the preference level of arcs that already have a lower preference level. There are several possibilities for the objective function that could be maximised.

One possibility is to still find a minimal subset of arcs, whose preference level has to be decreased, in order to remove all negative cycles from the graph. This means that further lowering the preference level of an arc whose preference level has already been lowered before, should be preferred over lowering the preference level of an arc that has not been changed yet. Decreasing the preference level of an arc by a larger amount, means that this arc is able to make a higher contribution to resolving negative cycles. Thus, in many cases, the result of this approach will be, that a small number of arcs is chosen, and their preference levels are decreased by a large amount. This approach implies a large disadvantage for only a couple of arcs. This may be a problem if, for example, different arcs are the responsibility of different agents, and fairness is considered more important than the number of agents that are affected.

Another option, therefore, is to try to share the “damage” as fairly as possible between all arcs. In that case, the objective function would be to maximise the number of arcs that have a preference level of at least $p$, on the condition that the number of arcs that have a preference level of at least $p - 1$ is maximised. This means that lowering the preference level of an arc with a high preference level should be preferred over lowering the preference level of an arc with a low preference level.

We will discuss algorithms for these two approaches. Of course, more advanced approaches are also possible. One could, for example, introduce weight factors to make a trade-off between lowering the preference level of multiple arcs that have a high preference level, and further lowering the preference level of one arc that already has a lower preference level. This falls outside the scope of our work, and is left for further research.

### 3.3.3 Algorithms

In this section, we discuss two algorithms for solving disruptions in an STP with multiple preference levels. The first algorithm minimises the number of affected arcs. The second algorithm maximises fairness.

**Minimising the number of affected arcs**

The idea of the algorithm that minimises the number of affected arcs is as follows. First, we find a minimal subset of arcs whose preference level has to be decreased
to level 1, in order to remove all negative cycles from the graph. This can be done using one of the algorithms for repairing an STP with two preference levels, described in Section 3.2, by considering the values at level \( p_{\text{max}} \) to be the values for the soft constraints, and the values at level 1 to be the values for the hard constraints. Then, for each arc in the subset that was found, we increase its preference level again to the highest possible preference level at which the resulting STP is still consistent. This preference level can be found by applying a binary search over the preference levels.

This algorithm first applies an algorithm for repairing an STP with two preference levels. If, for example, the bottom-up approach is used, this takes \( O(n^3 + n^2 m) \) time. For each arc in the subset that was found, a binary search over the preference levels is performed, and the preference level of the arc is increased to the appropriate level. If the minimal network is known, then finding the right preference level for a single arc can be done in \( O(\log(p_{\text{max}})) \) time. Increasing the arc’s preference level and recomputing the minimal network can be done in \( O(n^2) \) time, using the IFPC algorithm. Since there are at most \( m \) arcs in the subset, this takes \( O(m(\log(p_{\text{max}}) + n^2)) \) time. Thus, the time complexity of the entire algorithm is \( O((n^3 + n^2 m + m(\log(p_{\text{max}}) + n^2)) \), if a bottom-up approach is used for finding the subset of arcs that has to be changed.

Maximising fairness

The idea of the algorithm that maximises fairness is as follows. First, we apply a binary search over the preference levels, to find the highest preference level \( p \) at which the projected STP is still consistent. Then, we try for each arc to increase its preference level to level \( p + 1 \). This can be done using one of the algorithms for repairing an STP with two preference levels, described in Section 3.2. This approach is applied recursively on the arcs whose preference level could still be increased in the previous iteration. Thus, in each iteration we try to increase the preference level of as many arcs as possible by 1.

If an \( O(n^3) \) algorithm is used for checking the consistency of an STP, then finding the highest preference level at which the STP is still consistent takes \( O(n^3 \log p_{\text{max}}) \) time. Then, for a maximum of \( p_{\text{max}} - 1 \) iterations, an algorithm for repairing an STP with two preference levels is applied. If the bottom-up approach is used, this takes \( O(p_{\text{max}}(n^3 + n^2 m)) \) time. Thus, the time complexity of the entire algorithm is \( O((n^3 \log p_{\text{max}}) + (p_{\text{max}}(n^3 + n^2 m))) \), which is equal to \( O(p_{\text{max}}(n^3 + n^2 m)) \).

### 3.3.4 Modelling priority information

In this section, we will show how an STP with priority information can be modelled as an STP with multiple preference levels. STPs with priorities, to which Dasdan’s PRI-CYCLE algorithm is applied, have been introduced in Section 2.5.1. In an STP with priorities, a priority value \( \pi(a) \) is associated with each arc. These priority
values are positive integer values, $1 \leq \pi(a) \leq \pi_{\text{max}}$. They denote an ordering between different arcs, which indicates which arcs should be changed first, and which should rather keep their current weight. Arcs with a smaller priority number should be considered for a weight change before arcs with a higher priority number.

Now, we will show how an STP with $\pi_{\text{max}}$ different priority values can be modelled as an STP with $p_{\text{max}} = \pi_{\text{max}} + 1$ preference levels. The vertices and arcs in the graph with preferences are the same as in the graph with priorities. However, instead of a weight and a single priority value, a preference function is associated with each arc, which specifies the weight of the arc at each preference level. For each arc $a$, the weight $w_i(a)$ at preference level $p(a) = i$ is defined as follows:

$$w_i(a) = \begin{cases} 
\infty & \text{if } 1 \leq i \leq (p_{\text{max}} - \pi(a)) \\
\pi(a) & \text{if } (p_{\text{max}} - \pi(a)) < i \leq p_{\text{max}} 
\end{cases}$$

Here, $w_\pi(a)$ is the weight of arc $a$ in the STP with priorities. If the weight of an arc is $\infty$, then all values are allowed, which is equal to the arc being removed from the graph. Thus, informally one can say that at the $\pi(a)$ highest preference levels, the arc exists in the graph. If the preference level is lower than that, then the arc is removed from the graph. Arcs with a small priority number only exist at high preference levels. This corresponds with the idea that they should be considered for a weight change first.

The STP with priority information can now be solved by applying an algorithm that maximises fairness to this STP with preferences. Such algorithm maximises the number of arcs that have a preference level of at least $p$, on the condition that the number of arcs that have a preference level of at least $p - 1$ is maximised. This means that arcs with a priority value of $\pi$ will only be changed if it is not possible to solve the problem by changing arcs with a smaller priority number. Thus, arcs with a smaller priority number are indeed considered for a weight change before arcs with a larger priority number.

It may be possible that the user does not want to remove the arcs from the graph completely, but wants to specify the amount by which the arc may be changed. This can be done by replacing the value of $\infty$ for preference levels $1 \leq p(a) \leq (p_{\text{max}} - \pi(a))$ by the appropriate weight. As long as the STP is consistent at preference level $p = 1$, it will be possible to solve the problem. With this change, priority information is in fact combined with two preference levels.

### 3.4 A known source of inconsistency

Until now, we have looked at algorithms for repairing an inconsistent STP, without paying attention to the source of the inconsistency. We tried to find a minimal subset of arcs that has to be changed to make the distance graph free of negative cycles, by simply looking at the inconsistent STP. In many real-life situations, however, we have much more information than that. Often, it happens that we have already
made a schedule for a consistent situation and then, after we made that schedule, one constraint changes and causes the STP to become inconsistent. A comparable situation occurs, when we want to know what the cost is of changing a certain constraint. In such situation, we want to know how we can repair the STP while staying as close to our original schedule as possible. Thus, we also want to find a minimum subset of constraints that has to be changed to make the distance graph free of negative cycles. However, this subset should not contain the constraint that has just caused the inconsistency.

In this section, we investigate how, by allowing minor modifications, the general purpose algorithms described earlier in this document, can be applied to solve this problem. Furthermore, we present a new algorithm for repairing an inconsistent STP (without preference information), for which a change in only a single arc \( a_c \) has caused the STP to become inconsistent.

Once these algorithms have returned their solution, the user can decide whether he wants to adopt this solution by changing the constraints in the set that has been returned by the algorithm. Alternatively, he can try to solve the problem by revising and changing the constraint that has caused the inconsistency.

### 3.4.1 Framework

As a starting point, we assume that information is available about the arc \( a_c \) that has caused the inconsistency, as well as information about a previously consistent situation. We denote the consistent situation we had before the change by the subscript \( \text{old} \), and the inconsistent situation we got after the change by the subscript \( \text{new} \). The inputs for the algorithm are the distance graph of the previous situation \( G_{\text{old}} = \langle V_{\text{old}}, A_{\text{old}} \rangle \), the arc \( a_c \) that has been changed, and its new weight \( w_{\text{new}}(a_c) \). The vertex at the head of \( a_c \) is denoted by \( x_h \), and the vertex at the tail of \( a_c \) is denoted by \( x_t \). Thus we have \( a_c = x_t \xrightarrow{w_{\text{new}}(a_c)} x_h \). We assume that the \( d \)-graph (minimal network) for the previous situation is given. The shortest path from a vertex \( x_i \) to a vertex \( x_j \) in the previous situation is denoted by \( d_{\text{old}}(i, j) \).

### 3.4.2 Applying general purpose algorithms

In contrast to the problems we have investigated so far, for the problem we investigate in this section, we would like to find solutions that exclude a certain arc \( a_c \). Here, we will show how the general purpose algorithms we have discussed earlier in this document can be applied for finding such solutions. We assume that arc \( a_c \) is not a self-loop, and thus that all negative cycles contain at least two arcs. If \( a_c \) would be a self-loop, then it would be impossible to find a solution set that excludes this arc.

In order to exclude a specific arc from the solution set, the algorithms have to be able to treat this arc in a different way than the other arcs. Some of the algorithms, namely Dasdan’s PRI-CYCLE algorithm and our algorithms for STPs with preferences, have means of their own to make a distinction between different arcs. We
allow minor modifications to the algorithms that do not have any expressive power to make this distinction.

Algorithms that have to be adapted

The basic idea of these minor modifications, to distinguish the source of the inconsistency from the other arcs in the graph, is to add specific rules that ensure that this arc always remains in the graph. For the algorithms that iteratively choose a negative cycle and eliminate it from the graph (i.e., NEG-CYCLE, CRI-CYCLE, and the top-down algorithms for the basic case), this means that from these negative cycles different constraints are chosen to be changed or removed. This can be done by adding a simple check after the line that selects an arc \(a\) on the negative cycle \(C\). If \(a = a_c\), then we skip this arc and select a different arc. Since all negative cycles contain at least two arcs, the top-down algorithms will always be able to remove a different arc on the cycle, and thus to find a solution that excludes the source of inconsistency from the solution set.

For Dasdan’s NEG-CYCLE and CRI-CYCLE algorithms, on the other hand, this may not always work. Since these algorithms never really remove an arc from the graph, but only set their weights to zero, it can happen that, even if the weights of all other negative arcs on the cycle are set to zero, the remaining weight of the cycle is still negative. Therefore, for these algorithms, if we select arc \(a_c\) from the negative cycle, we do not skip it completely, but put it back to the end of the list of arcs that will be considered for a weight change. This way, it will still be possible to resolve the negative cycle by changing the weight of arc \(a_c\), but only if this has turned out to be impossible by changing the weights of other arcs. This means that solution sets returned by these algorithms may still include \(a_c\), but only as a last resort, if it is impossible to solve the problem in a different way.

The bottom-up algorithm for the basic case can be adapted in such a way that arc \(a_c\) is added to the graph \(G'\) first. Since at that moment no other arcs have been added to the graph yet, this will always be possible without creating any negative cycles. Once the arc has been added to \(G'\), it will not become part of the solution set anymore.

PRI-CYCLE

The PRI-CYCLE algorithm already has the notion of priorities, to distinguish between different arcs. In this algorithm, apart from the weight \(w\), also a priority \(\pi\) is associated with each arc in the graph. A priority is a positive integer value, \(1 \leq \pi \leq \pi_{max}\). The smaller the number, the higher the priority, i.e., a higher priority arc should be considered for a weight change before a lower priority arc. We can express our problem in this model by assigning priority \(\pi = 2\) to arc \(a_c\), and priority \(\pi = 1\) to all other arcs in the graph. Even though Dasdan does not explicitly mention it in his algorithm, we can assume that the selection procedure for selecting an arc \(a\) on the negative cycle \(C\) will select higher priority arcs before
lower priority arcs. This way, the PRI-CYCLE algorithm behaves in a way similar to our adapted versions of the NEG-CYCLE and CRI-CYCLE algorithms. The only point at which these algorithms differ from each other, is the way in which a negative cycle is chosen to be eliminated. The PRI-CYCLE algorithm uses cycle ratios, whereas NEG-CYCLE selects negative cycles in an arbitrary order, and CRI-CYCLE uses cycle means. The PRI-CYCLE algorithm may also return solution sets that do contain the source of inconsistency $a_c$.

**Algorithms for STPs with preferences**

In an STP with preferences, our problem can be modelled using two preference levels. At preference level 2, all arcs have the same weights as in the original STP. The fact that we do not want to change the source of inconsistency, can be modelled by assigning to arc $a_c$ the same weight at preference level 1 as it has at level 2. This way, even if its preference level should be lowered, it will always keep the same weight. All other arcs are assigned the weight $\infty$ at level 1, to indicate that they will be removed from the graph if their preference level is lowered. Alternatively, it is possible to assign them a different weight at level 1, if we want to restrict the amount by which these arcs can be changed. If at level 1 the other arcs are assigned the weight $\infty$, then the behaviour of these algorithms should be the same as that of our adapted algorithms for the basic case. As long as the STP remains consistent at preference level 1, the algorithms for STPs with preference levels will always be able to find a solution that does not change the weight of arc $a_c$.

3.4.3 **KSI algorithm for a known source of inconsistency**

We have shown how the problem of repairing an STP with a known source of inconsistency can be solved using the general purpose algorithms we have described in this thesis so far. These algorithms, however, do not use any of the knowledge that is available about the previous, consistent situation. In this section, we present the KSI algorithm: a new algorithm for repairing an STP with a known source of inconsistency that does take into account this additional information. Before we discuss this algorithm, we first observe a few things.

- **All negative cycles have to contain $a_c$.** Since the change in arc $a_c$ has caused the network to become inconsistent, all negative cycles have to contain $a_c$. If there would be negative cycles that would not contain $a_c$, then the network would already have been inconsistent before the change.

- **All negative cycles consist of $a_c$ and a path from $x_h$ to $x_t$.** This immediately follows from the previous observation.

- **A min-cut from $x_h$ to $x_t$ in the subgraph containing only the negative cycles is a solution to our problem.** Since we cannot remove $a_c$ itself, the only way we can get rid of a negative cycle is to remove an arc on the path from $x_h$.
**Algorithm 6** The KSI algorithm for repairing an STP with a known source of inconsistency.

- **input**: A distance graph $G_{old} = (V_{old}, A_{old})$ and its $d$-graph with values $d_{old}(i, j)$, an arc $a_c$ that has been changed and its new weight $w_{new}(a_c)$.
- **output**: A list $L$ of arcs whose weight has to be changed

1. $L = \emptyset$
2. $V_{nc} = \{x_h, x_t\}$
3. $A_{nc} = \{a_c\}$
4. forall $(x_i \in V)$ do
   5. if ($(w_{new}(a_c) + d_{old}(x_h, x_i) + d_{old}(x_i, x_t)) < 0$) then
      6. $V_{nc} = V_{nc} \cup x_i$
   7. forall $(a_{ij} \in A \mid x_i, x_j \in V_{nc})$ do
      8. if ($(w_{new}(a_c) + d_{old}(x_h, x_i) + w(a_{ij}) + d_{old}(x_j, x_t)) < 0$) then
         9. $A_{nc} = A_{nc} \cup a_{ij}$
10. $G_{nc} = (V_{nc}, A_{nc})$
11. $L = \text{Mincut}(G_{nc}, x_h, x_t)$
12. return $L$

To $x_t$. If we do this for all such paths, which means we have found a cut from $x_h$ to $x_t$, we have removed all negative cycles. Thus, we can solve the problem by finding a min-cut from $x_h$ to $x_t$ in the subgraph containing only the negative cycles.

The idea of our algorithm is now as follows. First, we construct the subgraph containing only the vertices and arcs involved in a negative cycle. Then, we apply a min-cut algorithm to this subgraph.

For construction of the negative cycles subgraph, we use the fact that in any consistent situation, there is always a shortest path from $x_h$ to another vertex $x_i$, or from another vertex $x_i$ to $x_t$, that does not contain $a_c$. Should $a_c$ be part of such a path, then this path would contain a cycle. Since the graph is consistent, such cycle can never be negative. Thus, by removing this cycle from the path, we can always obtain a path with the same or shorter length that does not contain $a_c$. Similarly, there will always consist a shortest path from $x_h$ to $x_i$, that does not contain any outgoing arc from $x_i$, and there will always consist a shortest path from $x_i$ to $x_t$ that does not contain any incoming arc to $x_i$. This means that we can use the values $d_{old}(h, i)$ and $d_{old}(i, t)$ in our check for the existence of negative cycles in the new situation.

Algorithm 6 shows our KSI algorithm for repairing an STP in which a change in exactly one arc $a_c$ has caused the STP to become inconsistent. It finds a subset of arcs that has to be changed, in order to repair this STP, that does not contain
this arc $a_c$. The algorithm first gets the subset of all vertices that lie on a negative cycle. A vertex $x_i$ is on a negative cycle if and only if the sum of the new weight of the changed arc $w_{\text{new}}(a_c)$, and the shortest path from $x_h$ to $x_t$ through $x_i$ in the old situation ($d_{\text{old}}(x_h, x_i) + d_{\text{old}}(x_i, x_t)$) is negative. Then, it checks for all arcs between the vertices in the subset it just found, whether they are on a negative cycle. Together, these two subsets of vertices and arcs form the subgraph $G_{nc}$, containing only the negative cycles. Finally, it runs a min-cut algorithm on this subgraph, to find a min-cut from $x_h$ to $x_t$ in $G_{nc}$.

The problem of finding a minimum cut from a source vertex $s$ to a sink vertex $t$ in a directed graph (i.e., $x_h$ and $x_t$ in our case), has been studied intensively in literature. Given a partition of the set of vertices $V$ into two sets $A$ and $B$, such that $s \in A$ and $t \in B$, then the weight of this cut is the sum of the weights of all arcs that go from a vertex in $A$ to a vertex in $B$. A minimum cut is a cut whose weight is smallest among all possible cuts of $V$. Since min-cut algorithms usually find a cut of minimum weight, and we want to find a cut with a minimum number of arcs, we first have to translate our data to make it suitable as input for a min-cut algorithm. This can be done by setting the weights of all arcs in $G_{nc}$ to 1. After the min-cut algorithm has found a solution, this solution can be translated back into the arcs from the original graph.

The most well-known algorithm for solving the min-cut problem is the Ford-Fulkerson algorithm, which has first been published in 1956 by L.R. Ford and D.R. Fulkerson [13]. The Edmonds-Karp algorithm [11], which is explained in detail in Appendix A.6, is a specialisation of this algorithm. Many other (and faster) algorithms for solving the minimum cut problem, or the related maximum flow problem, also exist. However, it goes beyond the scope of this thesis to discuss them all. An overview of existing algorithms and their time bounds can be found in [16].

If the minimal network of the previous situation is known, then finding the subset of vertices $V_{nc}$ can be done in $O(n)$ time. Finding the subset of arcs can be done in $O(m)$ time. Changing the weights of all arcs in $G_{nc}$ into 1, and back into their original weights, can be done in $O(m)$ time. Thus, the total complexity of the KSI algorithm is $O(n+m+\text{the complexity of the min-cut algorithm})$. If the Edmonds-Karp algorithm is used, then a min-cut can be found in $O(nm^2)$ time. However, if a faster algorithm is used, then this time bound will be improved. The runtime complexity of the KSI algorithm is mainly determined by the complexity of the min-cut algorithm. If the change affects only a small part of the network, then the subgraph containing only the negative cycles can be much smaller than the original graph. Then, finding a min-cut can be done even faster compared to the size of the original input.
3.5 Discussion of new algorithms

In this chapter, we have presented several new algorithms for resolving inconsistencies in STPs. We have described two main approaches for finding a set of constraints that have to be changed to remove all negative cycles from the distance graph, namely the top-down approach and the bottom-up approach. A bottom-up post-processing phase can be applied to the set found by the top-down algorithm, in order to guarantee a minimal solution. The bottom-up algorithm always finds a minimal solution. We have described how these algorithms can be adapted to solve an STP with two preference levels. Then, we showed how the model can be extended by allowing multiple preference levels. We described two approaches for solving an STP with multiple preference levels. One minimises the number of affected arcs, the other maximises fairness. Furthermore, we showed how an STP with priority information can be modelled as an STP with multiple preference levels. Finally, we described how we can repair an STP in the situation where a change in one constraint is known to be the source of the inconsistency. We showed how, with minor modifications, the general purpose algorithms can be applied to solve this problem. After that, we presented a new algorithm that also takes into account additional information about the previous consistent situation.

Our algorithms improve upon existing approaches by not imposing any unfounded additional requirements upon the solutions, such as blacklisting all positive arcs beforehand, or without any reason always preserving the ordering of time points. Instead, the use of preference information allows the user to model in a substantiated way which changes are allowed and which are not. Furthermore, our algorithms are guaranteed to always find a minimal set of arcs. Thus, unlike existing algorithms, it is not possible that they return a solution that contains redundant arcs.

An STP with priority information can be modelled as an STP with multiple preference levels. An STP with multiple preference levels is more expressive than an STP with priorities, as it allows the user to specify the amount by which the weight of an arc may be changed. Our approach improves upon the PRI-CYCLE algorithm, by guaranteeing that a minimal set of arcs with a large priority number is changed. Arcs with a small priority number are always considered for a weight change before arcs with a large priority number. Thus, our approach applies the priority information in a better way than the PRI-CYCLE algorithm.
Chapter 4

Experiments and evaluation

In this chapter, we evaluate the performance of our algorithms. We have conducted experiments to test whether or not the algorithms we developed achieve the goals for which they were designed in practical situations. Here, we describe these experiments, and analyse our main results. A detailed description of the results of all individual experiments can be found in Appendix B.

The structure of this chapter is as follows. First, we describe our experimental setup. We describe the test cases we have used, and briefly describe the experiments. Then, for each group of experiments, we describe our research questions and expected results, we give a detailed description of the experimental setup, and we present and analyse the most important results of the experiments. Finally, we end with a summary of our results.

4.1 Experimental setup

In this section, we give an overview of our experimental setup. We describe our performance measure, our implementation, and the test cases we have used. Furthermore, we briefly introduce the experiments we have conducted.

4.1.1 Performance measure

We evaluate the performance of the algorithms by comparing the quality of the solutions found. This quality is measured by the number of arcs that are changed (or removed) by the algorithms, in order to repair the inconsistent STP. The smaller this number of changed arcs is, the better the algorithm performs.

4.1.2 Implementation

All algorithms were implemented in Java. We used a framework set up by Léon Planken, which he developed for evaluation of the techniques described in [20]. We adapted his implementation to our needs, and added the algorithms we wanted
to evaluate. For several existing algorithms (Bellman-Ford, Floyd-Warshall and IFPC), we used Planken’s implementation, which was already present.

In our implementations, for all variants of the NEG-CYCLE algorithm, the PRI-CYCLE algorithm and the top-down algorithm, an extra vertex and \( n \) zero-weighted, outgoing arcs from this vertex to all other vertices, were added to the STP (where \( n \) is the total number of vertices in the original STP). This additional vertex was used as the source vertex for the Tarjan and YTO algorithms. This is necessary, to ensure that all other vertices can be reached from the source vertex. If there are vertices in the graph that cannot be reached from the source vertex, it may be possible that Tarjan’s algorithm and the YTO algorithm do not find any negative cycles, even though they do exist in the graph. This addition is explicitly mentioned as a part of the YTO algorithm as described in [8]. Since the additional vertex has only outgoing arcs, this vertex and the additional arcs will never be part of any cycle in the graph. Thus, the additional arcs will never be changed or removed by the algorithms, and will never be part of any solution set found by these algorithms. In any situation where a total number of arcs mentioned, this total number of arcs does not include the \( n \) additional arcs.

A side effect of this addition, is that all vertices will be visited immediately in the first pass of the TARJAN algorithm. This means that the algorithm becomes less sensitive to “unlucky” choices of the source vertex. In the original graph it could be possible that a small cycle remains undetected for a long time, because of such an unlucky choice of the source vertex. In the graph with the additional vertex and arcs, however, small cycles are always detected in early passes of the algorithm.

An alternative for this addition, is to first split the graph into strongly connected components, and then apply Tarjan’s algorithm to each of these components. Splitting a graph into strongly connected components can be done is \( O(n + m) \) time [24]. This is less than the runtime complexity of Tarjan’s algorithm, which takes \( O(nm) \) time. Splitting the graph into strongly connected components first, could reduce the runtime of the algorithms, since the components could be smaller than the original graph.

Since in this study, we measure the performance of the algorithms by the number of arcs that are changed by the algorithms, and not by the runtimes of the algorithms, we have not spent any attention to trying to optimise the algorithms for speed. This means that at some points, for ease of implementation, we have chosen to implement easier, but slower solutions, instead of faster, but more complicated solutions that achieve the same goals.

In our implementations, the algorithms do not have any special additions to randomise the selection procedures. Thus, when iterating the sets of vertices or arcs, or when selecting an arc from a negative cycle, the algorithms simply proceed with the first eligible element, instead of randomly selecting the next element. As a consequence of this, the results may be influenced by the order in which the data is stored, and the solutions of the different algorithms for the same test instance may diverge less than they would have if the selection procedures would be randomised.
4.1.3 Test cases

We have used two different types of test cases, namely randomly generated STP instances, and practical scheduling problems, to test the performance of our algorithms. The random STPs allow us to examine the behaviour of the algorithms regardless of any structure imposed upon the graph. With practical scheduling problems, we can evaluate the performance of the algorithms in typical situations for which the STP formalism has been designed in the first place.

The test cases we used, originate from two benchmark sets for SMT solvers from SMT-LIB\(^1\) [22] that are representative of these types of problems. All problem instances in these benchmark sets consist of a conjunction of clauses, each of which consists of a disjunction of literals. Each literal is a linear inequality of the form \(x - y \leq a\), where \(x\) and \(y\) are integer variables and \(a\) is an integer constant. A solution to such a problem instance, is an assignment of values to variables, such that from each clause at least one literal is satisfied. From these benchmarks, STP instances can be obtained by selecting exactly one literal from each clause.

The two benchmark sets we used, are DTP benchmarks and job shop benchmarks. The **DTP benchmarks** contain randomly generated problem instances for testing solvers of the Disjunctive Temporal Problem [23]. The STP instances generated from these benchmarks contain arcs with a random weight in the range \([-100, 100]\). The **job shop benchmarks** represent practical scheduling problems. The STP instances generated from these benchmarks contain arcs with weights in a relatively small range. Also, they contain a large amount of negative weight arcs, compared to the amount of arcs with a positive weight.

Figure 4.1 shows a graph in which, for all test cases in these benchmark sets, the number of clauses is plotted against the number of variables. For the DTP benchmarks, all instances contain 35 variables. Their number of clauses ranges from 175 to 245. For the jobshop benchmarks, the number of variables ranges from 5 to 241. The number of clauses ranges from 10 to 3960. When generating STP instances from these benchmark sets, if there were multiple constraints between the same two variables, we always stored the tightest constraint, and discarded the others. This way, STPs were generated that contained at most one forward arc and one backward arc between each two vertices. Because of this, the actual number of arcs in the STPs may be a little lower than the number of clauses displayed in Figure 4.1.

4.1.4 Experiments

The experiments we conducted can be divided into the following two groups:

1. **Basic case.** We compared Dasdan’s NEG-CYCLE algorithm with our top-down, top-down with post-processing, and bottom-up algorithms for the basic case.

\(^1\)SMT-LIB is a library of benchmarks for Satisfiability Modulo Theories.
2. A known source of inconsistency. We compared Dasdan’s NEG-CYCLE and PRI-CYCLE algorithms and our top-down algorithm with post-processing and bottom-up algorithm for the basic case with our algorithm for a known source of inconsistency.

These experiments will be described in detail in the remainder of this chapter.

4.2 Experiment group 1: Basic case

In Experiment group 1, we evaluate the performance of our algorithms for the basic case, which were described in Section 3.1. We investigate the effect of Dasdan’s additional requirements that only the weights of negative arcs are changed, and that negative arcs are never made positive. Additionally, we investigate the effect of our guarantee that our top-down with post-processing and bottom-up algorithms always return a minimal set.

To this end, we compared Dasdan’s NEG-CYCLE algorithm with our top-down, top-down with post-processing, and bottom-up algorithms for the basic case. Furthermore, we included in our tests a version of the NEG-CYCLE algorithm with a post-processing phase, and versions of our algorithms in which only negative arcs are removed from the graph.

4.2.1 Research questions

Our algorithms for the basic case improve upon Dasdan’s NEG-CYCLE algorithm on two points. First, they do not impose any unfounded additional requirements upon the solution, such as blacklisting all positive arcs beforehand, or without any reason always preserving the ordering of time points. Instead of only changing
the arcs’ weights to zero, they completely remove the arcs from the graph. Second, the top-down algorithm with post-processing and the bottom-up algorithm are guaranteed to always find a minimal set of arcs. Whereas Dasdan’s NEG-CYCLE algorithm may return a non-minimal set of arcs, these two algorithms are guaranteed to always return a minimal set. The rationale behind our choices, is that we want to find solutions that consist of a smaller number of arcs than the solutions returned by Dasdan’s algorithm.

We tested whether this goal is achieved, by comparing the size of the solutions found by Dasdan’s NEG-CYCLE algorithm with the size of the solutions found by our algorithms for the basic case. We also included in our tests a version of the NEG-CYCLE algorithm with a post-processing phase, and versions of our algorithms which only negative arcs are removed from the graph. This enables us to compare the different variables independently of each other. In the post-processing phase of the NEG-CYCLE algorithm with post-processing, we restore the weights of the changed arcs to their old value, if this can be done without yielding an inconsistent STP. This way, we create a minimal solution for the NEG-CYCLE algorithm.

We ran these algorithms on many different test cases with different characteristics. We varied the degree of inconsistency of the graph, and the ratio between positive and negative arcs. This way, we investigated the influence of the characteristics of the input data on the performance of the algorithms.

4.2.2 Experimental setup

Algorithms

We implemented and tested the following eight algorithms:

- NEG-CYCLE (NC)
- NEG-CYCLE with post-processing (NCPP)
- Top-down algorithm for the basic case (TD)
- Top-down algorithm for the basic case removing only negative arcs (TDNA)
- Top-down algorithm with post-processing for the basic case (TDPP)
- Top-down algorithm with post-processing for the basic case removing only negative arcs (TDPPNA)
- Bottom-up algorithm for the basic case (BU)
- Bottom-up algorithm for the basic case removing only negative arcs (BUNA)

The NEG-CYCLE algorithm (NC) (see Section 2.5.1) was developed by Dasdan. It finds a negative cycle using Tarjan’s algorithm. Then it selects negative arcs from that cycle and changes their weight into zero. This is repeated until the network
has become consistent. The algorithm has the two additional restrictions that only
the weights of negative arcs are changed, and that negative arcs are never made
positive.

The NEG-CYCLE algorithm with post-processing (NCPP) is the same as the
NEG-CYCLE algorithm. Only at the end a post-processing phase is added, in
which the arcs whose weight can be reverted to the original value, without yielding
an inconsistency, are changed back to the original value.

The top-down algorithm (TD) (see Section 3.1.1) finds a negative cycle using
Tarjan’s algorithm. Then it selects an arc from that cycle and removes the arc from
the graph. This is repeated until the network has become consistent. Thus, it works
in a way that is very similar to the NEG-CYCLE algorithm. Only, it does not have
the two additional restrictions.

The top-down algorithm with post-processing (TDPP) is the same as the top-
down algorithm, but also in this case a post-processing phase is added in which
arcs that can be put back into the graph without yielding an inconsistency are put
back into the graph. See Section 3.2.2 for the version of this algorithm for STPs
with two preference levels.

The bottom-up algorithm (BU) (see Section 3.1.1) starts with an “empty” copy
of the graph, that contains only the vertices of the original graph, but not its arcs.
Then for each arc in the set of arcs, it adds the arc to the graph, unless this leads to
an inconsistency. The arcs that cannot be added to the graph are the arcs that have
to be removed from the original graph.

The versions of the top-down algorithms and the bottom-up algorithm that re-
mov e only negative arcs are identical to their counterparts, but with the additional
restriction that only negative arcs may be removed.

Test cases

The test cases we used are random STPs and practical scheduling problems, which
were taken from the DTP and job shop benchmark sets described in Section 4.1.
From the test cases in these benchmark sets, we generated STP instances by ran-
domly selecting one linear inequality from each clause, such that the resulting STP
was inconsistent. Then, we manipulated the arc weights in the following six ways,
to create STP instances with a similar topology, but with different degrees of in-
consistency, and different ratios between positive and negative arcs:

1. *Original benchmark weights.* We kept the original weights of the benchmark
   set.

2. *Only negative arcs.* We set all arc weights to -1. This way, we created an
   STP with only negative arcs, and thus a very high degree of inconsistency.

3. *Tight schedule with few inconsistencies.* We randomly chose an assignment
   of values to time point variables, and set all arc weights to the smallest value
   for which this assignment was a solution. Then, we randomly introduced
inconsistencies in 2% of the arcs, by lowering their weights by a random amount. This way, we created an STP with a tight schedule and an approximately equal number of positive and negative arcs, with a only a few sources of inconsistency.

4. *Mostly positive arcs with few inconsistencies.* We randomly chose a positive weight for all arcs. Then, we randomly introduced inconsistencies in 2% of the arcs, by lowering their weights to a value such that the constraint could not be satisfied anymore. This way, we created an STP with mostly positive arcs, in which all negative arcs were a source of inconsistency.

5. *Tight schedule with one inconsistency source.* We randomly chose an assignment of values to time point variables, and set all arc weights to the smallest value for which this assignment was a solution. Then, we randomly introduced an inconsistency in one single arc, by lowering its weight to a value such that the constraint could not be satisfied anymore. This way, we created an STP with a tight schedule and an approximately equal number of positive and negative arcs, in which a single arc was the source of the inconsistency.

6. *Mostly positive arcs with one inconsistency source.* We randomly chose a positive weight for all arcs. Then, we randomly introduced an inconsistency in one single arc, by lowering its weight to a value such that the constraint could not be satisfied anymore. This way, we created an STP with mostly positive arcs, in which a single arc was the source of the inconsistency.

4.2.3 Expected results

We expected that our algorithms would always find a minimal solution, whereas the NEG-CYCLE algorithm may also find non-minimal solutions. Furthermore, we expected that, due to Dasdan’s additional requirements, the NEG-CYCLE algorithm may need to change more than one arc to remove a single negative cycle. In the special case where a cycle contains only negative arcs, the NEG-CYCLE algorithm has to change the weights of all arcs, whereas our algorithms only have to remove one arc. However, since changing the weights of multiple arcs may also affect more other negative cycles, we did not know what the net effect of these additional requirements would be on the total amount of arcs changed by the algorithm for random graphs.

For the test cases with only negative arcs, we expected the NEG-CYCLE algorithms to perform very badly. On the other hand, for the test cases with mostly positive arcs, we expected the algorithms that change only negative arcs to perform very well. For the test cases with a single source of inconsistency, it is possible to compare the solutions found by the algorithms with the optimal solution. We expected that for the tight schedules, with an almost equal number of positive and negative arcs, and a single source of inconsistency, the algorithms would often find non-optimal solutions. On the other hand, for the STPs with mostly positive arcs
in which a single negative arc is the source of the inconsistency, we expected the algorithms that change only negative arcs to find the optimal solution. For these test cases, we expected that the algorithms that change both positive and negative arcs may also find non-optimal solutions.

We hoped that the solutions found by our algorithms would also be at least as small as the solutions found by NEG-CYCLE. However, this does not necessarily have to be the case. The size of the solutions found, largely depends on the order in which the arcs are chosen, and there are no guarantees that the order applied by our algorithms is at least as good as (or better than) the order applied by the NEG-CYCLE algorithm. We expected that especially the bottom-up algorithms might show very different results compared to the other algorithms, since they apply a very different approach.

4.2.4 Results

In this section, we describe the results of the experiments in Experiment group 1. First, in Experiment group 1A, we describe the results for the random STP instances. Then, in Experiment group 1B, we describe the results for the practical scheduling problems. Here, we present our main results. A detailed discussion of the results of all individual experiments can be found in Appendix B.1.

1A: Random STPs

In Experiment group 1A, we evaluate the performance of our algorithms for the basic case for random STPs. The test cases for these experiments were generated from the DTP benchmark set. The six experiments in this group are numbered 1A.1 to 1A.6. They differ from each other in the weights that were assigned to the arcs. This has been described in Section 4.2.2. In this section, we will first give a global overview of our main results. After that, we will zoom in on some individual experiments, to find more detailed answers to our research questions, and to point out some remarkable results.

Global overview  From the results of our experiments, it turned out that the algorithms we tested can be divided into four groups, according to their performance. The NC group contains the NEG-CYCLE algorithms (NC and NCPP). The TD group contains the top-down algorithms without post-processing (TD and TDNA). The TDPP/BUNA group contains the top-down algorithms with post-processing (TDPP and TDPPNA) and the bottom-up algorithm removing only negative arcs (BUNA). The BU group contains the only bottom-up algorithm (BU).

Figure 4.2 shows boxplots with summary statistics for these four groups. In this figure, the NC group is red, the TD group is blue, the TDPP/BUNA group is magenta and the BU group is dark grey. On the vertical axis, the number of changed (or removed) arcs is displayed, as a percentage of the total number of arcs in the STP (excluding any additional arcs, added by the NEG-CYCLE and
Figure 4.2: Summary statistics of Experiment group 1A. The four groups of algorithms are displayed on the horizontal axis. The changed arcs are displayed on the vertical axis as a percentage of the total number of arcs in the STP.
top-down algorithms). In this figure, we can see that for most test cases, the NC group performs worse, the TD group performs mediocre, and TDPP/BUNA and BU groups perform best. Thus, for most test cases, our algorithms indeed find solutions that consist of a smaller number of arcs than the solutions returned by Dasdan’s NEG-CYCLE algorithm. Only for STPs with mostly positive arcs, in which all negative arcs are a source of inconsistency (Experiments 1A.4 and 1A.6), the BU group performs worse than all other groups. On the other hand, this group performs better than all other groups for STPs with tight schedules.

**Guarantee of a minimal solution** One of our research questions is what the effect is of the guarantee of always finding a minimal solution. Does the fact that the TDPP and BU algorithms always find a minimal solution always lead to better results than the NC and TD algorithms? And can the performance of the NC algorithm be improved by adding a post-processing phase, thereby guaranteeing a minimal solution?

To answer these questions, we will look at the results of Experiment 1A.1 (Random STPs with original benchmark weights), which are shown in Figure 4.3. In this figure, each data point indicates the result of one specific algorithm for one specific test case. The total number of arcs in the STP is displayed on the horizontal axis. The (absolute) number of changed arcs is displayed on the vertical axis. This means that all data points of one test case are displayed in the same column. Thus, the data points in the figure should be related to points either above or below them (but not to points on their left-hand or right-hand side).

In this figure, we can see that, as we expected, the TDPP algorithms always find smaller solutions than the TD algorithms. In fact, it could already be seen in Figure 4.2, that the TDPP/BUNA group yields much better results than the TD.
group. Thus, adding a post-processing phase to the top-down algorithm, in order to guarantee a minimal solution, is indeed very useful.

Quite surprisingly, however, Figure 4.3 shows that, even though the NCPP algorithm is guaranteed to find a minimal solution, it hardly yields any better results than the NC algorithm. Similar results are found for the other experiments. The difference between the NC and NCPP algorithms is always quite small, compared to the difference between these algorithms and other algorithms. Thus, the reason why NEG-CYCLE performs worse than our algorithms does not lie in the fact that it does not guarantee a minimal solution. Rather, we should seek its cause in the additional requirements Dasdan imposed upon the solutions. These imply that in his solutions only the weights of negative arcs are changed, and that these arcs are only changed into zero, instead of completely removed from the graph.

**Changing only negative arcs** This leads us to the following question: what is the effect of the requirement that only the weights of negative arcs are changed? In Figure 4.3, we can see that, for our algorithms, the difference between the regular versions and the versions in which only negative arcs may be changed is very small. In general, for all experiments, the difference between these two versions of one algorithm is much smaller than the difference between the different groups of algorithms. Also, it differs from experiment to experiment, whether the regular version, or the negative arcs version of an algorithm performs better. Thus, we conclude that the main reason for NEG-CYCLE’s worse performance lies in the restriction that the weights of the arcs are only changed into zero, instead of completely removed from the graph.

When looking at the results of Experiment 1A.2 (Random STPs with only negative arcs) in Figure 4.2, we can see that this restriction indeed has a large impact upon the size of the solution. Since in this experiment, all arc weights are negative, all algorithms will always find a solution that includes only negative arcs. Yet, the NEG-CYCLE algorithms find much larger solutions than all other algorithms, in many cases even changing the weights of all arcs in the graph.

The only exception to our observation that changing only the weights of negative arcs does not make very much difference, is the fact that for all STPs with mostly positive arcs, the BU algorithm performs worse than all other algorithms. This can be seen in the results of Experiment 1A.4 (Random STPs with mostly positive arcs with few inconsistencies), which are shown in Figure 4.4.

This can be explained by the fact that for all algorithms that change only the weights of negative arcs, the arcs that are a source of inconsistency are exactly the only arcs that may be changed by the algorithms. Thus, these algorithms will always immediately find the sources of inconsistency. Furthermore, the top-down algorithms have information about the location of negative cycles, whereas the bottom-up algorithm does not have this. Thus, it cannot apply this information in its procedure for selecting which arcs to remove from the graph.

Thus, for STPs with many positive arcs, the requirement of changing only the
weights of negative arcs, is indeed a good addition. Such cases are not very likely to occur in practise, though, as they do not specify any ordering between time points.

1B: Practical scheduling problems

In Experiment group 1B, we evaluate the performance of our algorithms for the basic case for practical scheduling problems. The test cases for these experiments were generated from the jobshop benchmark set. The six experiments in this group are numbered 1B.1 to 1B.6. They also differ from each other in the weights that were assigned to the arcs. This has been described in Section 4.2.2.

In general, the results of Experiment group 1B are very similar to the results of Experiment group 1A. In some cases, the differences between the performance of the different groups of algorithms are larger for practical scheduling problems than for random STPs, especially for larger STP instances. We attribute this to the fact that the practical scheduling problems themselves are much more structured than the random STPs. We will come back to this later in this section.

In this section, we will only discuss the most interesting results of this group of experiments. First we will discuss the results of Experiment 1B.1 (Practical scheduling problems with original benchmark weights). From a practical point of view, this is the most interesting experiment of all experiments in Experiment group 1, as it represents the most realistic type of problems. After that, we discuss how the performance of the algorithms is for STP instances of different input sizes. We were unable to make these observations for the random STPs, as they all have approximately the same size.

Figure 4.4: Experiment 1A.4: Random STPs with mostly positive arcs with few inconsistencies (absolute number of changed arcs).
Practical scheduling problems with original benchmark weights  From a practical point of view, Experiment 1B.1 (Original benchmark weights) is the most interesting experiment, as it represents the most realistic type of problems. Figure 4.5 shows the results of this experiment. In this figure, the total number of arcs in the STP is displayed on the horizontal axis. The (absolute) number of changed arcs is displayed on the vertical axis. The red line indicates the total number of arcs in the graph (i.e., the maximum amount of arcs that can be changed or removed).

When comparing the results of Experiment 1B.1 to the results of Experiment 1A.1 (Random STPs with original benchmark weights) in Figure 4.3, we can see that for this experiment, the distinction between the different groups of algorithms is much clearer than for Experiment 1A.1. This can be explained by the nature of these problems. The practical scheduling problems contain a large amount of negative arcs, compared to the amount of arcs with a positive weight. Therefore, the results are more similar to the results of random STPs with only negative arcs (Experiment 1A.2), than to the results of random STPs with random weight arcs (Experiment 1A.1).

Different input sizes  Figure 4.6 shows the results of Experiment group 1B. In this figure, the number of arcs that have been changed is displayed as a percentage of the total amount of arcs in the STP. Showing the relative number of arcs that have been changed makes it easier to compare the performance of the algorithms for different input sizes. Please note that the scales of the axes may vary, depending on the results of the tests. A red line in the figures indicates the total amount of arcs in the STP. Larger versions of these graphs can be found in Appendix B.1.

In this figure, we can see the trends that become visible as the STP instances become larger. In general, the results of the different algorithms are closer for smaller
Figure 4.6: Results of Experiment group 1B. The total number of arcs in the STP is displayed on the horizontal axis. The changed arcs are displayed on the vertical axis as a percentage of the total number of arcs in the STP.
instances than for larger instances. The larger the instances become, the better we are able to distinguish different groups of algorithms. This may be explained by the fact that these STP instances are very structured. The larger the instances become, the more clearly visible the influence of this structure becomes in the results. Also, for larger instances, the results may be less sensitive to specific choices of the random number generator that chose which arcs to include in the graph, and which weights to assign to these arcs.

For STPs with mostly negative arcs, we can make a very clear distinction between the different groups. As the instances become larger, the size of the results converges to a fixed percentage of the total number of arcs in the graph. The NC group performs worse, often changing the weights of all arcs in the graph. The TD group performs mediocre, changing around 60% of the arcs. The TDPP/BUNA and BU groups perform best, changing around 40% of the arcs.

In Experiment 1B.3 (Tight schedule with few inconsistencies), the variance between the results for different instances is much bigger. This may be explained by the fact that for these instances, changes in a small number of arcs are the source of many negative cycles. Because it is hard to find those specific arcs that have caused the inconsistencies, the results may be more sensitive to specific choices of the random number generator. In this experiment, we can see a remarkable difference between the performance of (most notably) the NEG-CYCLE algorithm for small instances and for larger instances, which we are unable to explain. Further research is required to find an explanation for these results. For both experiments with a tight schedule (Experiments 1B.3 and 1B.5), the percentage of changed arcs remains within the same range as the size of the instances increases.

In Experiment 1B.4 (Mostly positive arcs with few inconsistencies), we can see that for larger instances not only the bottom-up algorithm (BU) performs worse than the other algorithms, but also the top-down algorithm without post-processing (TD). However, the results of the TD algorithm are not as bad as the results of the BU algorithm. The quality of their solutions seems to decrease as the instances become larger. Further research would be required, to see how these trends continue for even larger instances.

In Experiment 1B.6 (Mostly positive arcs with one inconsistency source), on the other hand, the percentage of changed arcs decreases as the instances become larger. This is better visible in Figure 4.7, which shows the results for this experiment for only the larger instances. In fact, for all these instances, the number of arcs removed by the bottom-up algorithm is bounded above by 5 arcs. In this experiment, all other algorithms always found a solution that contained only a single arc.

Comparing the solutions to the optimal solution Apart from being able to define an ordering between the algorithms, based upon their performance, we would also like to know how the quality of their solutions is compared to the optimal solution. In particular, we would like to know whether the guarantee of a minimal
solution also leads to solutions whose size is (close to) optimal.

With the results of these experiments, we can compare the size of the solutions found by the algorithms to the size of the optimal solution (i.e., the size of the minimum set of arcs that has to be changed to remove all negative cycles). The size of the optimal solution can be derived from the way we introduced the inconsistencies in the STPs. For the STPs with few inconsistencies (Experiments 1B.3 and 1B.4), we do not know the exact size of the optimal solution, but we do have an upper bound for it, which is 2% of the total number of arcs in the STP. For the STPs with one inconsistency source (Experiments 1B.5 and 1B.6), we know that the optimal solutions contain only one single arc.

For STPs with mostly negative arcs, we do not have the size of the optimal solution. However, we can see that the algorithms in the NC and TD groups always find non-optimal solutions, because the solutions found by the TDPP/BUNA and BU groups are smaller.

For STPs with a tight schedule with few inconsistencies (Experiment 1B.3), none of the algorithms finds optimal solutions. Also, many algorithms find solutions that are much worse than the optimal solution. For STPs with a tight schedule with one inconsistency source (Experiment 1B.5), the algorithms are more often able to find the optimal solution. However, all algorithms also find non-optimal solutions in many cases, which often contain many more arcs than the optimal solution.

For STPs with mostly positive arcs, in which all negative arcs are a source of inconsistency (Experiments 1B.4 and 1B.6), all algorithms that change only the weights of negative arcs can only change the arcs that were a source of inconsistency. Thus, these algorithms always find a (close to) optimal solution for these
STPs. Also the TDPP algorithm finds similar results. The TD and BU algorithms also find non-optimal solutions. The quality of the solutions found by the BU algorithm is much worse than the quality of the solutions found by the TD algorithm.

Thus, the fact that some of the algorithms are guaranteed to find a minimal solution, is no guarantee that also a minimum solution will be found. For STPs with a tight schedule with few inconsistencies, all algorithms find solutions that are larger than the optimal solutions. For STPs with mostly positive arcs, many algorithms are able to find an optimal solution. However, this is rather due to the special structure of these STP instances and the fact that these algorithms may only change the weights of negative arcs, than to the algorithms’ guarantee of minimality.

4.2.5 Conclusions

In this group of experiments, we have compared Dasdan's NEG-CYCLE algorithm with our top-down algorithm, top-down algorithm with post-processing and bottom-up algorithm for the basic case. We have tested the performance of these algorithms on many different test cases, which represented random STPs and practical scheduling problems. For most experiments, the results for these two types of STPs were very similar. In some cases, the differences between the performance of the different algorithms were larger for practical scheduling problems than for random STPs. We attribute this to the fact that the practical scheduling problems themselves are much more structured than the random STPs.

We have seen that in most test cases, our algorithms outperform Dasdan’s NEG-CYCLE algorithm. For the practical scheduling problems taken from the jobshop benchmark set, the size of the solution sets of our best-performing algorithms is only approximately 40% of the size of the solution sets found by NEG-CYCLE. Only in the specific case that almost all negative arcs are a source of inconsistency, the NEG-CYCLE algorithm performs better than the top-down algorithm and the bottom-up algorithm, and the same as the top-down algorithm with post-processing. Such cases are not very likely to occur in practise, though, as they do not specify any ordering between time points.

Adding a post-processing phase to the NEG-CYCLE algorithm does not improve its performance very much. Even with a post-processing phase, which creates a minimal solution for the NEG-CYCLE algorithm, it still performs worse than our algorithms. The main reason for its worse performance lies in the restriction that the weights of negative arcs are only changed into zero, instead of being completely removed from the graph.

A very important factor for the performance of the algorithms is the ratio between positive and negative arcs in the graph. As the ratio of negative arcs increases, the differences between the algorithms become larger. The NEG-CYCLE algorithm performs very badly if there are many negative arcs in the graph, and very well if there are only a few negative arcs. The bottom-up algorithm performs very well if there are many negative arcs in the graph, but very poorly if there are only a few negative arcs. The top-down algorithm without post-processing performs
mediocre in all cases. The top-down algorithm with post-processing performs very well in almost all cases. For STPs with a tight schedule and an almost equal number of positive and negative arcs, and few sources of inconsistency, however, the bottom-up algorithm performs better than the other algorithms.

The fact that some of the algorithms are guaranteed to find a minimal solution, is no guarantee that also a minimum solution will be found. For STPs with a tight schedule with few inconsistencies, all algorithms find solutions that are larger than the optimal solutions. For STPs with mostly positive arcs, in which all negative arcs are a source of inconsistency, many algorithms are able to find an optimal solution. However, this is rather due to the special structure of these STP instances and the fact that these algorithms may only change the weights of negative arcs, than to the algorithms’ guarantee of minimality.

From the results in these experiments, we conclude that the top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU) are our preferred algorithms for finding a minimal subset of arcs that have to be removed from the distance graph, in order to make the graph free of negative cycles. Which algorithm should be used, depends on the structure of the graph. In these experiments, we have seen that for graphs with a tight schedule and an almost equal number of positive and negative arcs, and few sources of inconsistency, the bottom-up algorithm performs better. On the other hand, for graphs with many positive or many negative arcs, the top-down algorithm with post-processing performs best. Further research would be required to identify exactly the parameters that determine which of these two algorithms performs best.

4.3 Experiment group 2: A known source of inconsistency

In Experiment group 2, we evaluate the performance of our KSI algorithm for repairing an STP with a known source of inconsistency, which was described in Section 3.4.3. We compared our KSI algorithm with Dasdan’s NEG-CYCLE and PRI-CYCLE algorithms. Also, we compared it with our top-down algorithm with post-processing and our bottom-up algorithm for the basic case.

4.3.1 Research questions

When repairing an STP with a known source of inconsistency, our KSI algorithm takes into account additional information about the previous, consistent situation. In these experiments, we investigate whether the use of this additional information leads to solutions that are equally good as or better than the solutions found by the general purpose algorithms, which use only the information in the inconsistent STP, when applied to this same problem.

To this end, we compared our KSI algorithm with Dasdan’s NEG-CYCLE and PRI-CYCLE algorithms, and with our top-down algorithm with post-processing and bottom-up algorithm for the basic case. Of these algorithms, we used the
versions that were described in Section 3.4.2. As we explained there, the NEG-CYCLE algorithm, the top-down algorithm with post-processing, and the bottom-up algorithm were adapted slightly, such that (if possible) they exclude the arc that caused the inconsistency from the solution set. We assumed that, when selecting an arc from the negative cycle, the PRI-CYCLE algorithm considers negative arcs with a higher priority before those with a lower priority.

4.3.2 Experimental setup

Algorithms

We implemented and tested the following five algorithms:

- NEG-CYCLE (NC)
- PRI-CYCLE (PC)
- Top-down algorithm with post-processing for the basic case (TDPP)
- Bottom-up algorithm for the basic case (BU)
- Algorithm for a known source of inconsistency (KSI)

The NEG-CYCLE algorithm (NC) was developed by Dasdan. It finds a negative cycle using Tarjan’s algorithm. Then it selects negative arcs from that cycle and changes their weight into zero. This is repeated until the network has become consistent. The algorithm has the two additional restrictions that only the weights of negative arcs are changed, and that negative arcs are never made positive. We adapted the algorithm slightly, such that the arc that caused the inconsistency is the last negative arc on a negative cycle, to be considered for a weight change.

The PRI-CYCLE algorithm (PC) was also developed by Dasdan. It uses cycle ratios \( \rho(C) = \frac{w(C)}{\pi(C)} \) to select a cycle that should be zeroed (where \( w(C) \) is the weight of the cycle, and \( \pi(C) \) is its priority). It finds a cycle with the minimum cycle ratio using the YTO algorithm. Then it selects negative arcs \( a \) from this cycle and changes their weight, until the weight of the cycle has become zero. The weight of these arcs is set to 0, if the (remaining) weight of the cycle is less than the weight of the arc, or to \( w(a) - \) the remaining weight of the cycle, otherwise. This is repeated until the graph is free of negative cycles. We modelled the STP with a known source of inconsistency in such a way that the arc that has caused the inconsistency is assigned the priority \( \pi = 2 \), and all other arcs are assigned the priority \( \pi = 1 \). We assumed that, when selecting an arc from the negative cycle, the PRI-CYCLE algorithm considers negative arcs with a higher priority (i.e. a smaller number) before those with a lower priority. The PRI-CYCLE algorithm also has the additional restrictions that only the weights of negative arcs are changed, and that negative arcs are never made positive.

The top-down algorithm with post-processing (TDPP) finds a negative cycle using Tarjan’s algorithm. Then it selects an arc from that cycle and removes the arc
from the graph. This is repeated until the network has become consistent. After that, it enters a post-processing phase, in which arcs that can be put back into the graph without yielding an inconsistency are put back into the graph. We adapted the algorithm slightly, such that the arc that caused the inconsistency is never removed from the graph.

The bottom-up algorithm (BU) starts with an “empty” copy of the graph, that contains only the vertices of the original graph, but not its arcs. Then for each arc in the set of arcs, it adds the arc to the graph, unless this leads to an inconsistency. The arcs that cannot be added to the graph are the arcs that have to be removed from the original graph. We adapted the algorithm slightly, such that the arc that caused the inconsistency is always the first one to be added to the “empty” copy of the graph. This way, we ensured that this arc is never removed from the original graph.

The KSI algorithm for a known source of inconsistency (KSI) first constructs a subgraph of the original graph, containing only the vertices and arcs involved in a negative cycle. Then, it runs the Edmonds-Karp min-cut algorithm on this subgraph, to find a min-cut from the head of the arc that has caused the inconsistency to the tail of that arc. The arcs in the min-cut are the arcs that have to be removed from the original graph, in order to repair the inconsistency.

Performance measure

We measured the performance of the solutions of these algorithms by counting the number of arcs that are changed (or removed) by the algorithms. The smaller this number of changed arcs is, the better the algorithm performs. A solution that does not include the arc that was the source of the inconsistency always has a higher quality than a solution that does include this arc, regardless of the total number of arcs in the solution sets.

Test cases

The test cases we used are random STPs and practical scheduling problems, which were taken from the DTP and job shop benchmark sets described in Section 4.1. From the test cases in these benchmark sets, we generated STP instances by randomly selecting one linear inequality from each clause. Then, we manipulated the arc weights, to create consistent STP instances with different ratios between positive and negative arcs. Finally, for each instance, we lowered the weight of a single arc such that the STP became inconsistent. By varying the amount by which the weight of the arc was lowered, we created STP instances with disruptions of different degrees.

1. **Tight schedule.** We randomly chose an assignment of values to time point variables, and set all arc weights to the smallest value for which this assignment was a solution. Then, we randomly introduced an inconsistency in one
arc, by lowering the weight of this arc by a random amount. This way, we
created an STP with a tight schedule, with an approximately equal number
of positive and negative arcs, with a single source of inconsistency.

2. Mostly positive arcs. We randomly chose a positive weight for all arcs. Then,
we randomly introduced an inconsistency in one arc, by lowering the weight
of this arc by a random amount to a value such that the constraint could not
be satisfied anymore. This way, we created an STP with only positive arcs,
except for a single negative arc that is the source of the inconsistency.

3. Tight schedule with a very small disruption. We randomly chose an assign-
ment of values to time point variables, and set all arc weights to the smallest
value for which this assignment was a solution. Then, we randomly intro-
duced an inconsistency in one arc, by lowering the weight of this arc to the
largest possible value that still caused an inconsistency. This way, we created
an STP with a tight schedule, with a very small disruption.

4. Mostly positive arcs with a very small disruption. We randomly chose a
positive weight for all arcs. Then, we randomly introduced an inconsistency
in one arc, by lowering the weight of this arc to the largest possible value that
still caused an inconsistency. This way, we created an STP with only positive
arcs, except for a single negative arc that is the source of the inconsistency,
with a very small disruption.

5. Tight schedule with a big disruption. We randomly chose an assignment of
values to time point variables, and set all arc weights to the smallest value
for which this assignment was a solution. Then, we randomly introduced an
inconsistency in one arc, by lowering the weight of this arc to a value that
was smaller than -(the sum of the absolute values of the weights of all other
arcs in the graph). This way, we created an STP with a tight schedule, with
a very big disruption.

6. Mostly positive arcs with a big disruption. We randomly chose a positive
weight for all arcs. Then, we randomly introduced an inconsistency in one
arc, by lowering the weight of this arc to a value that was smaller than -(the
sum of the absolute values of the weights of all other arcs in the graph). This
way, we created an STP with only positive arcs, except for a single negative
arc that is the source of the inconsistency, with a very big disruption.

4.3.3 Expected results

We expected that our algorithms would always be able to find a solution that ex-
cludes the arc that was the source of the inconsistency. We expected that the NEG-
CYCLE and PRI-CYCLE algorithms would also find solutions that include the arc
that was the source of the inconsistency. We hoped that the KSI algorithm would

63
always find a set whose size is equal to or smaller than the solutions found by the other algorithms. However, it is not guaranteed that this is always the case.

Of the algorithms we compared in these experiments, we expected that the algorithm for a known source of inconsistency (KSI) would perform best, since it applies information about the previous, consistent situation. We expected that the top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU) would be the second best, since they are both guaranteed to return a minimal solution, and don’t impose any additional restrictions upon the solutions. We expected that the NEG-CYCLE algorithm (NC) and the PRI-CYCLE algorithm (PC) would perform worst, since they both impose additional restrictions upon the solution set. These additional restrictions imply that only negative arcs are considered for a weight change, and that negative arcs are never made positive.

We expected that for the STPs with a tight schedule and a single source of inconsistency, the NEG-CYCLE and PRI-CYCLE algorithms would still be able to find a solution to a subset of the test instances. However, for the STPs with only positive arcs, except for one single arc that is the source of the inconsistency, the arc that is the source of the inconsistency is the only arc that may be changed by the NEG-CYCLE and PRI-CYCLE algorithms. Therefore, for these test cases, we expected them not to find any solutions at all, excluding the source of inconsistency.

4.3.4 Results

In this section, we describe the results of the experiments in Experiment group 2. First, in Experiment group 2A, we describe the results for the random STP instances. Then, in Experiment group 2B, we describe the results for the practical scheduling problems. A detailed discussion of the results of these experiments can be found in Appendix B.2.

2A: Random STPs

In Experiment group 2A, we evaluate the performance of our KSI algorithm for repairing an STP with a known source of inconsistency for random STPs. The test cases for these experiments were generated from the DTP benchmark set. The test cases differ from each other in the weights that were assigned to the arcs. This has been described in Section 4.3.2. In this section, we will first give a global overview of our results. Then we will have a more detailed look at some aspects of these results.

Global overview The algorithms we tested in these experiments can be divided into three groups, according to their performance. The first group, containing the NEG-CYCLE (NC) and PRI-CYCLE (PC) algorithms, performs very badly. The second group, containing the top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU), performs mediocre. The third group, containing only the KSI algorithm (KSI), performs very well. Our KSI algorithm always
found a solution with the smallest size among any of the solutions returned by the algorithms, that did not include the source of inconsistency. This was the case for all test instances in all experiments.

Table 4.1 shows the summary statistics of the experiments in this group. For each experiment, these statistics show for how many instances the algorithms found a solution that was smallest among the solutions found by any of the algorithms, for how many instances the algorithms found a solution that was larger than this solution, and for how many instances the algorithms were unable to find a solution excluding the source of inconsistency. In this table, we can see that the NC and PC algorithms are often unable to find a solution that excludes the source of inconsistency. If they find a solution that excludes this source, in most cases it is a larger solution. The TDPP and BU algorithms are always able to find a solution that excludes the source of inconsistency. Yet, in many cases they still find larger solutions. In general, they perform better for STPs with mostly positive arcs and small disruptions, than for tight schedules. The KSI algorithm always finds the smallest solution.

**Solutions including the inconsistency source** In many cases, the NC and PC algorithms find a solution including the source of inconsistency. We will now investigate in which cases they find such a solution, and explain why this is the case.

In Table 4.1, we can see that, as we expected, the NC and PC algorithms always find a solution including the source of inconsistency for all STPs with mostly positive arcs. In these STPs, the source of inconsistency is the only negative arc in the graph, and therefore the only arc that may be changed by the algorithms. Thus, the solutions that the NC and PC algorithms found for the STPs with mostly positive arcs always contain just one single arc, which is exactly the source of inconsistency.

Also for STPs with a tight schedule with a big disruption (Experiment 2A.5), the NC and PC algorithms always find a solution that includes the source of inconsistency. This can be explained by the value that has been chosen for the weight of the arc that has caused the inconsistency. The new weight of this arc has been chosen such that it was less than -(the sum of the absolute value of the weights of all other arcs in the graph), thereby representing a very big disruption. This means that changing the weights of all other negative arcs on a negative cycle into zero will never be enough to compensate for the amount by which the source of the inconsistency has been changed when causing the disruption. Thus, the source of the inconsistency is the only arc that is left, whose weight change can resolve the inconsistency. In contrast to the STPs with mostly positive arcs, however, the solutions returned by these algorithms for these STPs often contain more than just one single arc. In this experiment, the NC and PC have first tried to resolve the inconsistency by changing the weights of all other negative arcs on the negative cycle. Only after they failed to resolve the inconsistency that way, they have resorted to changing the weight of the source of inconsistency.
Table 4.1: Summary statistics of Experiment 2A. For each experiment, these statistics show the number of test instances for which each algorithm has found a smallest solution among the solutions found by any of the algorithms, a larger solution, or a solution that included the arc that was the source of the inconsistency.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>19</td>
<td>60</td>
</tr>
<tr>
<td>Larger solution</td>
<td>15</td>
<td>15</td>
<td>47</td>
<td>41</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>45</td>
<td>45</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) 2A.1: Tight schedule.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>0</td>
<td>0</td>
<td>51</td>
<td>52</td>
<td>60</td>
</tr>
<tr>
<td>Larger solution</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>60</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) 2A.2: Mostly positive arcs.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>1</td>
<td>6</td>
<td>11</td>
<td>15</td>
<td>60</td>
</tr>
<tr>
<td>Larger solution</td>
<td>44</td>
<td>39</td>
<td>49</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(c) 2A.3: Tight schedule with a very small disruption.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>0</td>
<td>0</td>
<td>51</td>
<td>47</td>
<td>60</td>
</tr>
<tr>
<td>Larger solution</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>60</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(d) 2A.4: Mostly positive arcs with a very small disruption.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>0</td>
<td>0</td>
<td>22</td>
<td>21</td>
<td>60</td>
</tr>
<tr>
<td>Larger solution</td>
<td>0</td>
<td>0</td>
<td>38</td>
<td>39</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>60</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(e) 2A.5: Tight schedule with a big disruption.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>0</td>
<td>0</td>
<td>33</td>
<td>15</td>
<td>60</td>
</tr>
<tr>
<td>Larger solution</td>
<td>0</td>
<td>0</td>
<td>27</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>60</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(f) 2A.6: Mostly positive arcs with a big disruption.
For STPs with a tight schedule and a smaller disruption (Experiments 2A.1 and 2A.3), the NC and PC algorithms found a solution including the source of inconsistency for only a part of the test cases. For these experiments, they always found such a solution for the same test instances. The reason for this is, that these test instances contain at least one cycle for which changing the weights of all other negative arcs on the cycle into zero, is not enough to make the weight of the cycle non-negative. Thus, changing the weight of the source of inconsistency is the only solution that remains for eliminating the negative cycle. The smaller the disruption is, the more often these algorithms are able to resolve all negative cycles without changing the source of inconsistency. Therefore, the number of solutions including the source of inconsistency is much smaller for Experiment 2A.3 than for Experiment 2A.1.

**Quality of larger solutions** The next thing we look at, is the quality of the solutions returned by the algorithms that are larger than the smallest solution returned by any of the algorithms. If the algorithms return a larger solution, how many arcs does this solution contain compared to the smallest solution?

Figure 4.8 shows the results for all experiments in Experiment group 2A. In this figure, the total number of arcs in the STP is displayed on the horizontal axis. This number excludes any additional arcs, added by the NC, PC and TDPP algorithms. The number of changed (or removed) arcs is displayed on the vertical axis. This means that all data points of one test case are displayed in the same column. Thus, the data points in the figures should be related to points either above or below them (but not to points on their left-hand or right-hand side). Please note that the scales of the axes may vary, depending on the results of the tests. The test instances for which the NC and PC algorithms found a solution including the arc that caused the inconsistency are omitted from the figure. This is the reason why for these algorithms no data points are shown for Experiments 2A.2, 2A.4, 2A.5 and 2A.6. Larger versions of these graphs can be found in Appendix B.2.

In this figure, we can see that for the STPs with a tight schedule (Experiments 2A.1, 2A.3 and 2A.5), the larger solutions found by the algorithms often contain many more arcs than the smallest solution. Also, the quality of the larger solutions found by the NC and PC algorithms is worse than the quality of the larger solutions found by the TDPP and BU algorithms. This can be explained by the fact that the NC and PC algorithms only change the weights of the arcs into zero, instead of completely removing them from the graph. Therefore, they often need to change the weights of more arcs than the TDPP and BU algorithms to eliminate a negative cycle.

For STPs with mostly positive arcs and small disruptions (Experiments 2A.2 and 2A.4), both the TDPP and BU algorithms often find an smallest solution. When they find a larger solution, this solution often contains just a few more arcs than the smallest solution. An explanation for this could be that for these instances, the number of negative cycles and the number of arcs in these cycles remain quite
Figure 4.8: Results of Experiment group 2A. The total number of arcs in the STP is displayed on the horizontal axis. The changed arcs are displayed on the vertical axis as a percentage of the total number of arcs in the STP.
small. Therefore, the number of possible solutions may also be quite small.

For STPs with mostly positive arcs and a big disruption (Experiment 2A.6), the BU algorithm often finds larger solutions that contain many more arcs than the smallest solution. The number of arcs in the larger solutions found by the TDPP algorithm, on the other hand, remains relatively small. The reason why the TDPP algorithm performs better than the BU algorithm, could be that the TDPP has a means for reverting bad choices in the post-processing phase, whereas the BU algorithm cannot reconsider a bad choice after it has been made.

A quite remarkable result in Figure 4.8 is the difference between the results of the TDPP algorithm for Experiments 2A.5 and 2A.6. When comparing the results of these experiments, we can see that the BU algorithm has approximately the same performance for both experiments. The TDPP algorithm, on the other hand, performs much better for Experiment 2A.6 than for Experiment 2A.5. For both these experiments, the weight of the arc that has caused the disruption was chosen such, that all cycles in which this arc participates have become negative cycles. Thus, the differences in the performance of the TDPP algorithm are not caused by any differences in the size of the smallest solution or the number of negative cycles in the graph. Thus, we conclude that the performance of the TDPP algorithm is also influenced by the ratio between positive and negative arcs in the graph, since at this moment that is the only remaining difference we can see between the test cases of these two experiments. The TDPP algorithm does not only perform worse for STPs with many negative cycles than for STPs with only a few negative cycles, but it also performs worse for STPs with more negative arcs than for STPs with less negative arcs. We have not been able to explain why this is the case, though. Further research would be needed to check the validity of our conclusions, and to find an explanation for these results.

2B: Practical scheduling problems

In Experiment group 2B, we evaluate the performance of our KS algorithm for repairing an STP with a known source of inconsistency for practical scheduling problems. The test cases for these experiments were generated from the job shop benchmark set. Also in this experiment group, the test cases differ from each other in the weights that were assigned to the arcs. This has been described in Section 4.3.2.

In general, the results of Experiment group 2B are almost identical to the results of Experiment group 2A. Therefore, the results of this group of experiments will not be discussed in full detail. We will only point out some interesting differences.

In this group of experiments, we could observe the performance of the different algorithms for different input sizes. For Experiments 2B.1, 2B.3, 2B.5 and 2B.6, the number of arcs in the larger solutions increases as the instances become larger. For these experiments, most of the smallest solutions are found for the smaller test instances. An explanation for this could be that for very small instances the optimal solution may be the only available solution, whereas for larger instances, there may
Table 4.2: Summary statistics of Experiments 2B.1 and 2B.3 with identical random seeds for both experiments. The results of the TDPP algorithm are different for both experiments, whereas the results of the BU algorithm are the same. We have not been able to explain this difference.

4.3.5 Conclusions

In this group of experiments, we have investigated the problem of repairing an STP for which a change in a single arc is known to be the source of the inconsistency, and we want to repair the STP without changing the arc that has caused the in-
consistency. We have seen that the algorithms we have tested can be divided into three groups, according to their performance. The first group, containing the NEG-CYCLE (NC) and PRI-CYCLE (PC) algorithms, performs very badly. The second group, containing the top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU), performs mediocre. The third group, containing only the KSI algorithm (KSI), performs very well. The results of the experiments are very similar for the random STPs and for the practical scheduling problems.

The NEG-CYCLE and PRI-CYCLE algorithms perform very badly in all experiments. As we expected, when the rest of the graph contains only positive arcs, these algorithms are unable to find a solution that excludes the source of inconsistency. However, even for STPs with a tight schedule with a very small disruption (i.e., the experiments in which these algorithms performed best), they are unable to find such a solution in about 30% of the cases. In almost cases where they were able to exclude the source of inconsistency from the solution set, they found a larger solution, whose quality was worse than that of larger solutions found by other algorithms. The additional restrictions that positive arcs are never changed, and negative arcs are never made positive, are too restrictive in combination with our requirement that the source of inconsistency should be excluded from the solution set. Thus, we conclude that the NEG-CYCLE and PRI-CYCLE algorithms are unsuitable for finding solutions to the problem of repairing an STP with a known source of inconsistency.

The performance of the top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU) depends on the characteristics of the STP that has to be repaired. For STPs with a tight schedule, they are able to find pretty good solutions for very small test instances. For larger instances, however, they perform very badly. On the other hand, for STPs with mostly positive arcs, both algorithms perform very well as long as the number of negative cycles in the graph and the size of the optimal solution remains low. For STPs with mostly positive arcs and big disruptions, these algorithms perform worse. For these types of STPs, the BU algorithm has the same performance as for the STPs with a tight schedule. The TDPP algorithm, on the other hand, performs better for STPs with mostly positive arcs and a big disruption, than for STPs with a tight schedule. It finds more smallest solutions, and also the quality of its larger solutions is not as bad as for the STPs with a tight schedule. We have not been able to find a satisfactory explanation for the differences in the performance of the TDPP algorithm for different test cases. Also, we found some inexplicable differences between the performance of the TDPP algorithm for tight schedules with a very small disruption and for tight schedules with a little larger disruption. Further research would be required to fully explain these results. However, we do not deem this very important, as the results of the KSI algorithm are better anyway.

Our KSI algorithm, for repairing an STP with a known source of inconsistency, has found a solution with the smallest size among the solutions found by any of the algorithms for all test cases. For many test cases, it also found better solutions than all other algorithms. Therefore, this algorithm is clearly superior in repairing STPs
with a known source of inconsistency, where the source of inconsistency should be excluded from the solution set. We conclude that its approach of finding a min-cut in the subgraph containing only the negative cycles is a good heuristic for finding an optimal solution. There is one side note to this, however. The KSI algorithm is not guaranteed to always find the best solution. Additional research would be required to find out in which cases this algorithm performs worse than the other algorithms.

4.4 Summary

In this chapter, we have conducted several experiments, to evaluate the performance of our algorithms in practical situations. We have implemented several of our algorithms and tested their performance on many different test cases, taken from two benchmark sets. The STP instances we used represented random STPs and practical scheduling problems.

In Experiment group 1, we evaluated the performance of our algorithms for the basic case. To that end, we compared Dasdan’s NEG-CYCLE algorithm with our top-down, top-down with post-processing and bottom-up algorithms for the basic case. We have seen that our algorithms outperform Dasdan’s NEG-CYCLE algorithm in almost all cases. The main reasons for this, are the additional restrictions the NEG-CYCLE algorithm imposes upon the solutions. These imply that positive arcs are never changed, and that negative arcs are never made positive. Only in the very specific case, that almost all negative arcs are a source of inconsistency, the NEG-CYCLE algorithm performs better than the top-down algorithm and the bottom-up algorithm, and the same as the top-down algorithm with post-processing. Such cases are not very likely to occur in practice, though, as they do not specify any ordering between time points. For graphs with a tight schedule and an almost equal number of positive and negative arcs, and few sources of inconsistency, the bottom-up algorithm is our preferred algorithm for finding a minimal subset of arcs that have to be removed from the distance graph, in order to make the graph free of negative cycles. For graphs with many positive or many negative arcs, the top-down algorithm with post-processing is the algorithm of our choice. Further research would be required to identify exactly the parameters that determine which of these two algorithms performs best.

In Experiment group 2, we have investigated the problem of repairing an STP for which a change in a single arc is known to be the source of the inconsistency, and we want to repair the STP without changing the arc that has caused the inconsistency. To that end, we compared our KSI algorithm for repairing an STP with a known source of inconsistency with Dasdan’s NEG-CYCLE and PRI-CYCLE algorithms, and with our top-down algorithm with post-processing and bottom-up algorithm for the basic case. We have seen that these algorithms can be divided into three groups, according to their performance. The NEG-CYCLE and PRI-CYCLE algorithms perform very badly. The top-down algorithm with post-processing and
the bottom-up algorithm perform mediocre. The KSI algorithm performs very well. Since the KSI algorithm is the only algorithm that found the smallest solution for all test cases, it clearly outperforms all other algorithms. Therefore, it is our preferred algorithm for repairing an STP with a known source of inconsistency.
Chapter 5

Discussion

In this chapter, we first summarise our results from previous chapters, and present our conclusions. After that, we list some topics that could be interesting for future research.

5.1 Summary and conclusions

Many temporal problems can be modelled as a Simple Temporal Problem (STP). A solution to an STP can be found if the network is consistent, which is the case if its distance graph is free of negative cycles. In dynamic environments, a previously consistent STP can become inconsistent due to a change in one of the constraints. Then, this STP has to be repaired by removing all negative cycles from the graph. During this repair, we aim to stay as close to our original solution as possible, thus changing a minimum set of constraints. However, we also have to take into account which constraints may be changed, and by which amount they may be changed.

In this thesis, we investigated heuristic algorithms for finding a minimum set of constraints that has to be changed to free the distance graph of negative cycles.

In Chapter 2, we introduced the STP formalism, and formally defined the problem of resolving disruptions in STPs. We gave an overview of existing algorithms for finding a subset of constraints whose weights have to be changed, in order to remove all negative cycles from a distance graph. It turned out that this problem has not been studied very intensively yet. The only algorithms we have found, that address this problem, are the NEG-CYCLE, CRI-CYCLE and PRI-CYCLE algorithms, which were developed by Dasdan.

We identified several shortcomings of these algorithms, and indicated the main points at which these algorithms can be improved. These are mainly the criteria for choosing the arcs of which the weights may be changed, and the amount by which their weights may be changed. On these two points, Dasdan’s algorithms make choices and impose additional restrictions upon the solutions that are not grounded upon any requirements imposed by the problem that is modelled in the
STP. This can be improved by not restricting the solutions beforehand, but instead using information from the problem domain to specify which changes are allowed and which are not. Furthermore, the existing algorithms do not give any guarantees about the minimality of the solutions.

In Chapter 3, we presented several new algorithms for resolving disruptions in STPs. First, we discussed algorithms for the basic case, in which any arc may be removed from the graph completely, in order to repair the STP. We described two algorithms: a top-down algorithm (which can be extended with a post-processing phase), and a bottom-up algorithm. These algorithms improve upon existing approaches by not imposing any unfounded additional requirements upon the solutions, such as blacklisting all positive arcs beforehand, or without any reason always preserving the ordering of time points. Furthermore, the top-down algorithm with post-processing and the bottom-up algorithm are guaranteed to always find a minimal set of arcs.

After that, we adapted the algorithms for the basic case, such that they could be applied to repair an STP with two preference levels. By adding preference information to the model, we are able to model additional information about the problem that is represented in the STP. This enables us to make better substantiated choices about which changes are allowed and which are not. Thus, we improve upon existing algorithms by using information from the problem domain to guide our choices, instead of making choices that do not have any foundation in the problem at hand.

We extended this model by generalising to multiple preference levels, thereby increasing the expressiveness of our model. We presented two algorithms for repairing STPs with multiple preference levels. The first algorithm minimises the number of affected arcs. The second algorithm maximises fairness. We also showed how an STP with multiple preference levels can be used to model priority information. Our approach improves upon the PRI-CYCLE algorithm, by applying the priority information in a better way. Furthermore, our model is more expressive than an STP with priorities, as it allows the user to specify the amount by which the weight of an arc may be changed.

Finally, we investigated the problem of repairing an STP for which one single constraint is known to be the source of the inconsistency, and we want to repair the STP without changing the constraint that has just caused the inconsistency. To the best of our knowledge, this problem has not been studied in literature before. We showed how the general purpose algorithms, described earlier in this thesis, can be applied for solving this problem. After that, we presented the KSI algorithm: a new algorithm for repairing an STP with a known source of inconsistency, which also takes into account information that is available about the previous, consistent situation.

In Chapter 4, we evaluated the performance of our algorithms for the basic case and our KSI algorithm, for repairing an STP with a known source of inconsistency.
We have conducted several experiments in which we compared the quality of the solutions, in terms of the number of arcs that are changed (or removed) by the algorithms. The smaller this number of arcs is, the better the algorithms perform. We tested the algorithms on many different test cases taken from two benchmark sets, which represented random STPs and practical scheduling problems.

In Experiment group 1, we compared Dasdan’s NEG-CYCLE algorithm with our top-down, top-down with post-processing, and bottom-up algorithms for the basic case. In these experiments, our algorithms found better solutions than the NEG-CYCLE algorithm in almost all cases. Only in the very specific case that almost all negative arcs are a source of inconsistency, the NEG-CYCLE algorithm also performed well. For graphs with a tight schedule and an almost equal number of positive and negative arcs, and few inconsistencies, the bottom-up algorithm is our preferred algorithm for finding a minimal subset of arcs that have to be removed from the distance graph, in order to make the graph free of negative cycles. For graphs with many positive or many negative arcs, the top-down algorithm with post-processing is the algorithm of our choice. Further research would be required to identify exactly the parameters that determine which of these two algorithms performs best.

In Experiment group 2, we compared Dasdan’s NEG-CYCLE and PRI-CYCLE algorithms and our top-down algorithm with post-processing and bottom-up algorithm for the basic case with our KSI algorithm for a known source of inconsistency. In these experiments, the KSI algorithm found the best solution for all test cases, thereby clearly outperforming all other algorithms.

Summarising these results, we conclude that our algorithms improve upon existing approaches for resolving disruptions in STPs. We have shown that we are able to find better solutions than existing algorithms, not only in theory but also in practise, by not imposing any ungrounded additional restrictions upon the solutions. Furthermore, we have shown that by adding preference information to the model, we can increase its expressiveness. This allows us to model additional information about the problem, which enables us to make better substantiated choices. Finally, we have addressed the problem of repairing an STP with a known source of inconsistency, which has not been investigated before. We have shown that, by using information about the previous, consistent situation, we are able to find better solutions than algorithms that do not use this additional information.

### 5.2 Future work

In this section, we list some topics that could be interesting for future research.

**Top-down algorithm with post-processing vs. bottom-up algorithm** In our experiments, we have seen that the top-down algorithm with post-processing and the bottom-up algorithm perform better than the NEG-CYCLE algorithm and the top-
down algorithm without post-processing. However, we have not been able to identify exactly in which case which of these two algorithms performs best. Further research would be required to identify exactly the parameters that determine the performance of these algorithms, in order to be able to decide when which of these two algorithms should be used.

**Empirical evaluation of runtimes** In our experiments, we have limited ourselves to comparing the performance of the algorithms in terms of the number of arcs that are changed by the algorithms in the solutions they found. We have not spent any attention to the time that the algorithms need for finding their solutions. However, an algorithm that returns a larger solution set may still be preferred over an algorithm that returns a smaller solution set, if it is able to find a solution significantly faster. Thus, for a complete evaluation of the performance of the algorithms, it is also desirable to compare their runtimes. To that end, the implementations of all algorithms would have to be optimised for speed, and their runtimes would have to be compared.

**Negative cycle and arc selection heuristics** At this moment, the order in which the negative cycles are repaired, and the order in which arcs are chosen, are very arbitrary. Since these orders have a large impact on the performance of the algorithms, it would be very interesting to investigate whether it is possible to perform these selection procedures in a smarter way, such that “important” cycles and arcs are selected first. Ideally, it would be possible to give guarantees about the approximation factor that can be obtained.

**Combine preferences and known source of inconsistency** Now that we have discussed algorithms for repairing an STP with preferences, and for repairing a regular STP for which one constraint is known to be the source of the inconsistency, a logical step would be to combine these two cases. Thus, it would be interesting to investigate how information about the source of the inconsistency can be used in an STP with preferences. In such situation, it will not be enough to simply find a min-cut in the graph of negative cycles, and decrease the preference levels of the arcs that form the min-cut, because this may not be sufficient to break all negative cycles.

**Advanced approaches for multiple preference levels** In this research, we have discussed two algorithms for repairing an STP with multiple preference levels. As mentioned in Section 3.3, more advanced approaches are also possible. One could, for example, introduce weight factors to make a trade-off between lowering the preference level of multiple arcs that have a high preference level, and further lowering the preference level of one arc that already has a lower preference level.
Multiple constraints have changed  Another topic that would be interesting to investigate, is the case in which multiple constraints have changed and caused the STP to become inconsistent. The goal would then be to find a minimum subset of constraints that have to be changed, that does not contain any of these constraints that have just caused the inconsistency. Also in this case, it would not be sufficient to simply find a min-cut, because there is no single source and sink of the network. Besides, the information about the minimal network of the previously consistent situation may be outdated, because the shortest paths might go through one of the constraints that have changed.
Bibliography


Appendix A

Existing algorithms

In this appendix, we give the full versions of the following existing algorithms:

<table>
<thead>
<tr>
<th>Section</th>
<th>Algorithm</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>Floyd-Warshall</td>
<td>All-pairs-shortest-paths</td>
</tr>
<tr>
<td>A.2</td>
<td>IFPC</td>
<td>Incremental Full Path Consistency</td>
</tr>
<tr>
<td>A.3</td>
<td>PRI-CYCLE</td>
<td>Solving disruptions in an STP with priorities</td>
</tr>
<tr>
<td>A.4</td>
<td>Tarjan</td>
<td>Negative cycle detection</td>
</tr>
<tr>
<td>A.5</td>
<td>YTO</td>
<td>Minimum cycle ratio</td>
</tr>
<tr>
<td>A.6</td>
<td>Edmonds-Karp</td>
<td>Min-cut</td>
</tr>
</tbody>
</table>
A.1 Floyd-Warshall

The Floyd-Warshall algorithm [5, 12, 26] (shown in Algorithm 7) solves the all-pairs-shortest-paths (APSP) problem, which is to find the shortest path between each pair of vertices in a graph. By applying the Floyd-Warshall algorithm to the distance graph, one can compute the minimal network (or \(d\)-graph) of an STP. The runtime complexity of this algorithm is \(O(n^3)\).

<table>
<thead>
<tr>
<th>Algorithm 7: The Floyd-Warshall algorithm.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong>: A distance graph (G = (V, A))</td>
</tr>
<tr>
<td><strong>output</strong>: A (d)-graph</td>
</tr>
<tr>
<td>1 <strong>for</strong> ((i = 1 \text{ to } n)) <strong>do</strong></td>
</tr>
<tr>
<td>2 (d_{ii} = 0;)</td>
</tr>
<tr>
<td>3 <strong>for</strong> ((i, j = 1 \text{ to } n)) <strong>do</strong></td>
</tr>
<tr>
<td>4 (d_{ij} = a_{ij};)</td>
</tr>
<tr>
<td>5 <strong>for</strong> ((k = 1 \text{ to } n)) <strong>do</strong></td>
</tr>
<tr>
<td>6 <strong>for</strong> ((i, j = 1 \text{ to } n)) <strong>do</strong></td>
</tr>
<tr>
<td>7 (d_{ij} = \min(d_{ij}, d_{ik} + d_{kj});)</td>
</tr>
<tr>
<td>8 <strong>return</strong> (d;)</td>
</tr>
</tbody>
</table>
A.2 Incremental Full Path Consistency (IFPC)

The Incremental Full Path Consistency algorithm (IFPC) [20] (shown in Algorithm 8) is an incremental algorithm for solving the all-pairs-shortest-paths (APSP) problem, which is to find the shortest path between each pair of vertices in a graph. Given an STP for which the all-pairs-shortest-paths are known (i.e., the minimal network or $d$-graph) and a new constraint $a_{ab}$, which has to be added to the STP, the algorithm adds $a_{ab}$ to the STP and computes the all-pairs-shortest-paths after addition of the new constraint. The runtime complexity of this algorithm is $O(n^2)$.

**Algorithm 8**: The Incremental Full Path Consistency algorithm (IFPC).

**Input**: A $d$-graph $G = (V, A)$ and a new arc $a'_{ab}$

**Output**: CONSISTENT if $a'_{ab}$ has been added to $G$, which is again minimal; INCONSISTENT otherwise

1. if $(w(a'_{ab}) + w(a_{ba}) < 0)$ then
   2. return INCONSISTENT;
3. if $(w(a'_{ab}) \geq w(a_{ab}))$ then
   4. return CONSISTENT;
5. $w(a_{ab}) = w(a'_{ab})$;
6. $I = \emptyset$;
7. $J = \emptyset$;
8. foreach $(x_k \in V, x_k \neq x_a, x_b)$ do
   9. if $(w(a_{kb}) > w(a_{ka}) + w(a_{ab}))$ then
      10. $w(a_{kb}) = w(a_{ka}) + w(a_{ab})$;
      11. $I = I \cup k$;
   12. if $(w(a_{ak}) > w(a_{ab}) + w(a_{bk}))$ then
      13. $w(a_{ak}) = w(a_{ab}) + w(a_{bk})$;
      14. $J = J \cup k$;
15. foreach $(i \in I, j \in J, i \neq j)$ do
   16. if $(w(a_{ij}) > w(a_{ia}) + w(a_{aj}))$ then
      17. $w(a_{ij}) = w(a_{ia}) + w(a_{aj})$;
18. return CONSISTENT;
A.3 PRI-CYCLE

In Algorithm 9, the full version of Dasdan’s PRI-CYCLE algorithm is given. This algorithm removes all negative cycles from an STP with priority information. The PRI-CYCLE algorithm is described in detail in Section 2.5.1.

<table>
<thead>
<tr>
<th>Algorithm 9: The PRI-CYCLE algorithm (PC).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong>: A distance graph ( G = (V, A) )</td>
</tr>
<tr>
<td><strong>output</strong>: A list ( L ) of arcs whose weight has to be changed</td>
</tr>
<tr>
<td>1. ( L = \emptyset; )</td>
</tr>
<tr>
<td>2. repeat</td>
</tr>
<tr>
<td>3. ( \langle \rho^*(G), C \rangle = YT\sigma(G); )</td>
</tr>
<tr>
<td>4. if ( \rho^*(G) &lt; 0 ) then</td>
</tr>
<tr>
<td>5. ( \text{rem} = w(C); )</td>
</tr>
<tr>
<td>6. while ( \text{rem} &lt; 0 ) do</td>
</tr>
<tr>
<td>7. Select one arc ( a ) on ( C ) with ( w(a) &lt; 0 );</td>
</tr>
<tr>
<td>8. if ( \text{rem} &lt; w(a) ) then</td>
</tr>
<tr>
<td>9. ( \text{rem} = \text{rem} - w(a); )</td>
</tr>
<tr>
<td>10. ( w(a) = 0; )</td>
</tr>
<tr>
<td>11. else</td>
</tr>
<tr>
<td>12. ( w(a) = w(a) - \text{rem}; )</td>
</tr>
<tr>
<td>13. ( \text{rem} = 0; )</td>
</tr>
<tr>
<td>14. ( L = L \cup {a}; )</td>
</tr>
<tr>
<td>15. until ( \rho^*(G) \geq 0 )</td>
</tr>
<tr>
<td>16. return ( L; )</td>
</tr>
</tbody>
</table>
A.4 Tarjan’s algorithm

Tarjan’s algorithm [4, 25] is both a single source shortest path algorithm and a negative cycle detection algorithm. It can also be used for checking the consistency of an STP. The goal of this algorithm is to compute the shortest paths $d(x_i)$ from a source vertex $x_s$ to all other vertices $x_i \in V$. During this process, it builds a graph $G_p$ of parent pointers. The parent $\text{parent}(x_i)$ of a vertex $x_i$ is the predecessor of $x_i$ on the (so far known) shortest path from $x_s$ to $x_i$. A cycle in the parent graph $G_p$ implies the existence of a negative cycle in the original STP. Thus, for any consistent STP, the graph $G_p$ is a tree. During the construction of the parent graph, Tarjan’s algorithm checks for the occurrence of cycles in $G_p$.

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{input}: A distance graph $G = (V, A)$, and a source vertex $x_s \in V$
\State \textbf{output}: A list $L$ of arcs on the negative cycle or \text{CONSISTENT}
\State $L = \emptyset$
\State \hspace{1em} \textbf{foreach} ($x_i \in V$) \textbf{do}
\State \hspace{2em} $d(x_i) = \infty$
\State \hspace{2em} $\text{parent}(x_i) = \text{NULL}$
\State \hspace{2em} $\text{status}(x_i) = \text{unreached}$
\State \hspace{1em} \hspace{1em} $d(x_s) = 0$
\State \hspace{1em} \hspace{1em} $\text{status}(x_s) = \text{labelled}$
\State \hspace{1em} \hspace{1em} $Q = x_s$
\State \hspace{1em} \hspace{1em} \textbf{while} ($Q \neq \emptyset$) \textbf{do}
\State \hspace{2em} \hspace{2em} $x_i = \text{Get-head}(Q)$
\State \hspace{2em} \hspace{2em} \textbf{if} ($\text{status}(x_i) == \text{labelled}$) \textbf{then}
\State \hspace{2em} \hspace{3em} \textbf{foreach} ($a_{ij} \in A$) \textbf{do}
\State \hspace{2em} \hspace{3em} \hspace{1em} \textbf{if} ($d(x_i) + w(a_{ij}) < d(x_j)$) \textbf{then}
\State \hspace{2em} \hspace{3em} \hspace{2em} \textbf{if} ($x_i \in \text{subtree rooted at } x_j$) \textbf{then}
\State \hspace{2em} \hspace{3em} \hspace{3em} \hspace{2em} $L = \text{cycle from } x_i \text{ to } x_j \text{ to } x_i$
\State \hspace{2em} \hspace{3em} \hspace{3em} \hspace{2em} \textbf{return} $L$
\State \hspace{2em} \hspace{3em} \hspace{3em} \hspace{1em} \hspace{1em} \textbf{else}
\State \hspace{2em} \hspace{3em} \hspace{3em} \hspace{2em} \hspace{1em} $d(x_j) = d(x_i) + w(a_{ij})$
\State \hspace{2em} \hspace{3em} \hspace{3em} \hspace{2em} $\text{status}(x_j) = \text{labelled}$
\State \hspace{2em} \hspace{3em} \hspace{3em} \hspace{2em} \text{Add } x_j \text{ to } Q$
\State \hspace{2em} \hspace{3em} \hspace{3em} \hspace{2em} $\text{parent}(x_j) = x_i$
\State \hspace{2em} \hspace{3em} \hspace{3em} \hspace{2em} \text{Disassemble subtree rooted at } x_j$
\State \hspace{2em} \hspace{3em} \hspace{3em} \hspace{1em} \hspace{1em} $\text{status}(x_i) = \text{scanned}$
\State \hspace{2em} \hspace{1em} \hspace{1em} \textbf{else}
\State \hspace{2em} \hspace{1em} \hspace{1em} $d(x_j) = d(x_i) + w(a_{ij})$
\State \hspace{2em} \hspace{1em} \hspace{1em} $\text{status}(x_j) = \text{labelled}$
\State \hspace{2em} \hspace{1em} \hspace{1em} \text{Add } x_j \text{ to } Q$
\State \hspace{2em} \hspace{1em} \hspace{1em} $\text{parent}(x_j) = x_i$
\State \hspace{2em} \hspace{1em} \hspace{1em} \text{Disassemble subtree rooted at } x_j$
\State \hspace{1em} \hspace{1em} \textbf{return} \text{CONSISTENT}$
\end{algorithmic}
\end{algorithm}

Tarjan’s algorithm is shown in Algorithm 10. In this algorithm, a status
status\(x_i\) \in \{\text{unreached, labelled, scanned}\} is associated with each vertex \(x_i\). A scan operation on a vertex \(x_i\) implies that all outgoing arcs \(a_{ij}\) from \(x_i\) are checked. If the shortest known path \(d(x_j)\) from \(x_s\) to \(x_j\) is longer than the weight of the shortest known path \(d(x_i)\) to \(x_i\) + the weight of arc \(a_{ij}\), then \(d(x_j)\) is updated to this smaller value. The status unreached means that for this vertex no distance information is available yet, or that the information is outdated. The status labelled means that new distance information has become available, but the vertex has not been scanned yet. The status scanned means that the vertex has already been scanned, and that its distance information has not been updated afterwards.

The algorithm maintains a FIFO queue \(Q\) of labelled vertices (which still have to be scanned). In each iteration, it takes the next vertex to be scanned, \(x_i\), from the head of the queue. For each outgoing arc \(a_{ij}\), it checks whether the distance information to \(x_j\) can be updated. If this is the case, then it first checks whether \(x_i\) occurs in the subtree rooted at \(x_j\) in the currently known parent graph \(G_p\). If \(x_i\) occurs in this subtree, then the STP contains a negative cycle, which is returned by the algorithm. Otherwise, the distance information \(d(x_j)\) is updated, \(x_j\) is marked as “labelled” and added to the queue. Furthermore, the subtree rooted at \(x_j\) in \(G_p\) is disassembled, and vertices in this subtree are marked as “unreached”. This is done, because the distance information that was available is now outdated. If all vertices in the queue have been scanned, and no negative cycle has been found, then the STP is consistent.
A.5 YTO algorithm

The Young-Tarjan-Orlin algorithm (YTO) [8, 28] (shown in Algorithm 11) finds a cycle with the minimum cycle ratio $\rho^*(G)$ of a graph $G$. This is the minimum value of $\rho(C) = w(C)/\pi(C)$ for all cycles $C$ in $G$ (where $w(C)$ is the weight of the cycle and $\pi(C)$ is the priority of the cycle). The YTO algorithm can also be used for finding a cycle with the minimum cycle mean, by assigning the priority $\pi(a) = 1$ to all arcs $a$ in the graph. The pseudocode in Algorithm 11 is taken from [8].

The YTO algorithm was originally designed for solving the parametric shortest paths problem. In this problem, a parametrised value is subtracted from the weight of each arc in the graph. The original weight $w(a)$ of an arc $a$ is replaced by the value $w(a) - r\pi(a)$. The goal of the algorithm is to find the maximum value of $r$, for which the graph with the replaced arc weights has no negative cycles. This is the largest value of $r$, for which $d(x_j) \leq d(x_i) + (w(a_{ij}) - r\pi(a_{ij}))$ for each arc $a_{ij} \in A$. Here, $d(x_i)$ is the weight of a minimum-weight path from the source vertex $x_s$ to $x_i$. This maximum value of $r$ is exactly equal to the minimum cycle ratio $\rho^*(G)$. Thus, the YTO algorithm can also be used for computing the minimum cycle ratio.

The algorithm associates a vertex key $vk(x_i)$ with each vertex $x_i$, and an arc key $ak(a_{ij})$ with each arc $a_{ij}$. The vertex key of a vertex $x_i$ is the arc with the minimum arc key among all incoming arcs of $x_i$. The arc key of an arc is computed by the function $\text{Find-arc-key}$ (shown in Function $\text{Find-arc-key}$).

Just like Tarjan’s algorithm, the YTO algorithm constructs a graph of parent pointers. The parent $\text{parent}(x_i)$ of a vertex $x_i$ is the predecessor of $x_i$ on the path from the source vertex $x_s$ to $x_i$ in this parent graph. The algorithm maintains two values for each vertex $x_i$. $d(x_i)$ is the sum of the weights $w(a)$ of the arcs $a$ on the path from $x_s$ to $x_i$ in the parent graph. $p(x_i)$ is the sum of their priorities $\pi(x_i)$.

The algorithm maintains a priority queue $H$ of $\langle \text{arc key}, \text{vertex key} \rangle$ tuples. This data structure contains for each vertex $x_i$ the corresponding vertex key $vk(x_i)$ (i.e., the arc with the minimum arc key value among all incoming arcs of $x_i$), and the arc key value $ak(vk(x_i))$ of this vertex key. The priority queue is keyed on the arc keys. Thus, the function $\text{PQ-find-min}$ returns the tuple with the smallest arc key value in the queue. $\text{PQ-insert}$ adds an entry to the queue, and $\text{PQ-delete}$ removes a given entry from the queue.

In each iteration, the algorithm takes the tuple $\langle r, a_{uv} \rangle$ with the smallest arc key from the priority queue $H$. If the parent of $x_u$ can be set to $x_v$ without resulting in a cycle in the parent graph, this is done, and the data of all vertices in the subtree rooted at $x_v$ are updated. The algorithm terminates if the value of $r$ is infinite, in which case the graph does not contain any cycles, or if $x_u$ is in the subtree rooted at $x_v$. In that case, the cycle with the minimum cycle ratio is found and returned by the algorithm.
Algorithm 11: The Young-Tarjan-Orlin algorithm (YTO).

**input**: A distance graph \( G = (V, A) \) with priorities \( \pi(a) \) for each arc, and a source vertex \( x_s \in V \) with outgoing arcs \( a_{si} \) to all other vertices \( x_i \in V \)

**output**: A list \( L \) of arcs on the critical cycle

1. \( L = \emptyset \);
2. \( \text{foreach} \ (x_i \in V) \) \ do
   3. \( d(x_i) = 0; \ \ p(x_i) = 1; \ \ \text{parent}(x_i) = x_s; \ \ vk(x_i) = \text{NULL}; \)
4. \( p(x_s) = 0; \)
5. \( \text{foreach} \ (a_{ij} \in A) \) \ do
   6. \( \text{Find-arc-key}(a_{ij}, G); \)
   7. \ if \ (vk(x_j) == NULL or ak(a_{ij}) < ak(vk(x_j))) \ then
      8. \( \text{PQ-insert}(H, (\infty, \text{NULL})); \)
   9. \( \text{foreach} \ (x_i \in V) \) \ do \( \text{PQ-insert}(H, (\text{ak}(vk(x_i)), vk(x_i))); \)
10. \ while \ (true) \ do
   11. \( \langle r, a_{uv} \rangle = \text{PQ-find-min}(H); \)
   12. \ if \ (r == \infty or x_u \in \text{subtree rooted at} x_u) \ then
   13. \( \text{minimumCycleRatio} = r; \)
   14. \ if \ (r \neq \infty) \ then \ L = \text{cycle from} x_u \text{ to} x_v \text{ to} x_u; \)
   15. \ return \( L; \)
   16. \( \text{foreach} \ (x_i \in \text{subtree rooted at} x_v) \) \ do
      17. \( d(x_i) = d(x_i) + (d(x_u) + w(a_{uv}) - d(x_v)); \)
      18. \( p(x_i) = p(x_i) + (p(x_u) + \pi(a_{uv}) - p(x_v)); \)
   19. \( \text{parent}(x_v) = x_u; \)
   20. \( \text{foreach} \ (x_i \in \text{subtree rooted at} x_v) \) \ do
      21. \( \text{ak}(vk(x_i)) = \infty; \)
      22. \( \text{foreach} \ (\text{incoming arc} a_{ji} \text{ of} x_i) \) \ do
         23. \( \text{Find-arc-key}(a_{ji}, G); \)
         24. \ if \ (ak(a_{ji}) \leq \text{ak}(vk(x_i))) \ then
            25. \( \text{PQ-delete}(H, (\text{ak}(vk(x_i)), vk(x_i))); \)
            26. \( \text{PQ-insert}(H, (\text{ak}(vk(x_i)), vk(x_i))); \)
      27. \( \text{foreach} \ (\text{outgoing arc} a_{ij} \text{ of} x_i) \) \ do
         28. \( \text{Find-arc-key}(a_{ij}, G); \)
         29. \ if \ (ak(a_{ij}) \leq \text{ak}(vk(x_j))) \ then
            30. \( \text{PQ-delete}(H, (\text{ak}(vk(x_j)), vk(x_j))); \)
            31. \( \text{PQ-insert}(H, (\text{ak}(vk(x_j)), vk(x_j))); \)
Function Find-arc-key \((a_{ij}, G)\)

**input**: An arc \(a_{ij}\), and a distance graph \(G = \langle V, A \rangle\) with priorities \(\pi(a)\) for each arc

**output**: The value of the arc key of \(a_{ij}\)

1. \(\Delta p = p(x_i) + \pi(a_{ij}) - p(x_j);\)
2. if \((\Delta p > 0)\) then
3. \(\Delta d = d(x_i) + w(a_{ij}) - d(x_j);\)
4. return \(\Delta d/\Delta p;\)
5. else
6. return \(\infty;\)
A.6 Edmonds-Karp

The Edmonds-Karp algorithm [11] is an algorithm for computing the maximum flow from a source vertex \( s \) to a sink vertex \( t \) in a directed graph with capacities for all arcs. This algorithm is a specialisation of the more famous Ford-Fulkerson algorithm [13]. The goal of the algorithm is to find the maximum amount of flow that can be sent through the network from \( s \) to \( t \) without exceeding the capacities of the arcs. The max-flow min-cut theorem [13] says that the maximum flow value is equal to the capacity of the minimum cut. We will first explain how the algorithm works. After that, we will show how it can be used for finding a min-cut.

The basic idea of the algorithm is as follows: As long as there is a path from \( s \) to \( t \), with available capacity on all its arcs, we send flow along this path. Then, we continue with the next path, and so on, until it is no longer possible to send more flow through the network. A breadth-first search is used to find the next path.

The Edmonds-Karp algorithm is shown in Algorithm 13. For each arc \( a_{ij} \), the algorithm maintains the value \( \text{flow}_{ij} \), which is the amount of flow that is already sent along this arc. Furthermore, it maintains the capacities \( r_{ij} \) of the arcs in the residual graph. This residual graph contains the same set of vertices as the original graph. The residual capacity of an arc is the capacity that is “leftover” after subtraction of the flow that has already been sent along this arc \( (r_{ij} = c(a_{ij}) - \text{flow}_{ij}) \). Also, for each arc \( a_{ij} \) along which a flow \( \text{flow}_{ij} > 0 \) is sent, the residual graph contains a backward arc with a residual capacity \( r_{ji} \) equal to the flow of \( a_{ij} \) \( (r_{ji} = \text{flow}_{ij}) \). As long as there is a path from \( s \) to \( t \) with a positive capacity in the residual graph (i.e., an augmenting path), it is possible to send more flow from \( s \) to \( t \) in the original graph. The backward arcs make it possible to “undo” the flow that has been sent along \( a_{ij} \) earlier, by pushing it backward.

The algorithm repeatedly calls the function \text{Find-augmenting-path}. This function finds an augmenting path in the residual graph, using a breadth-first search. During this search, it constructs a parent graph. The augmenting path can be found by following the path from sink vertex \( t \) back to source vertex \( s \) in this parent graph. The capacity \( m \) of the path is equal to the minimum capacity of any arc on this path. After it has found an augmenting path, the algorithm updates the flow values of the arcs on this path, and the capacities of the arcs in the residual graph. The algorithm terminates if no new augmenting path can be found. The runtime complexity of this algorithm is \( O(nm^2) \).

Even though the Edmonds-Karp algorithm was originally designed to find the value of the maximum flow or minimum cut in the graph, its results can also be used to find the set of arcs that constitute this minimum cut. This can be done as follows. At the moment the Edmonds-Karp algorithm terminates, we have a residual graph in which there is no path from \( s \) to \( t \). Let \( A \) be the set of vertices that can be reached from \( s \) in this residual graph, and let \( B \) be the set of all other vertices. Since there is no path from \( s \) to \( t \), it is always the case that \( s \in A \) and \( t \in B \). Thus, \( (A, B) \) is a valid \( s-t \)-cut. Now, the set of arcs in the min-cut are exactly those arcs that go from a vertex in \( A \) to a vertex in \( B \) (see [19] for a proof).
Algorithm 13: The Edmonds-Karp algorithm.

\textbf{input}: A directed graph \( G = (V, A) \) with a capacity \( c(a) \) for each arc, a source vertex \( s \in V \) and a sink vertex \( t \in V \)

\textbf{output}: The value of the maximum flow \( \text{maxflow} \), the amount of flow sent along the arcs \( \text{flow} \), and the capacities of the arcs in the residual graph \( \text{rc} \)

1. \( \text{maxflow} = 0; \)
2. \textbf{foreach} \((x_i, x_j \in V)\) \textbf{do} \[ \text{flow}_{ij} = 0; \]
3. \hspace{1em} \textbf{if} \((a_{ij} \in A)\) \textbf{then} \[ \text{rc}_{ij} = c(a_{ij}); \]
4. \hspace{1em} \textbf{else} \[ \text{rc}_{ij} = 0; \]
5. \textbf{while} \( (\text{true}) \) \textbf{do} \[ \langle m, \text{parent} \rangle = \text{Find-augmenting-path}(G, \text{rc}, s, t); \]
6. \hspace{1em} \textbf{if} \((m == 0)\) \textbf{then} \[ \text{break}; \]
7. \hspace{1em} \text{maxflow} = \text{maxflow} + m;
8. \hspace{1em} \text{x}_j = t;
9. \hspace{1em} \textbf{while} \((x_j \neq s)\) \textbf{do} \[ \text{x}_i = \text{parent}(x_j); \]
10. \hspace{2em} \text{flow}_{ij} + = m;
11. \hspace{2em} \text{flow}_{ji} - = m;
12. \hspace{2em} \text{rc}_{ij} - = m;
13. \hspace{2em} \text{rc}_{ji} + = m;
14. \hspace{2em} \text{x}_j = x_i;
15. \hspace{1em} \textbf{return} \langle \text{maxflow}, \text{flow}, \text{rc} \rangle;
Function \textit{Find-augmenting-path}(G, rc, s, t)

\textbf{input}: A directed graph $G = (V, A)$ with a capacity $c(a)$ for each arc, the residual capacities $rc$, a source vertex $s \in V$ and a sink vertex $t \in V$

\textbf{output}: The capacity of the path found, and the parent graph $parent$

$\begin{array}{l}
\text{foreach} \ (x_i \in V) \ \text{do} \\
\quad \text{status} \ (x_i) = \text{unreached}; \\
\quad \text{parent} \ (x_i) = \text{NULL}; \\
\quad \text{minCapacity} \ (x_i) = \infty; \\
\text{status} \ (s) = \text{labelled}; \\
\text{Q} = \{s\}; \\
\text{while} \ (Q \ \text{is not empty}) \ \text{do} \\
\quad x_i = \text{Get-head}(Q); \\
\quad \text{foreach} \ (x_j \in V) \ \text{do} \\
\quad \quad \text{if} \ (rc_{ij} > 0 \ \text{and} \ \text{status} \ (x_j) == \text{unreached}) \ \text{then} \\
\quad \quad \quad \text{parent} \ (x_j) = x_i; \\
\quad \quad \quad \text{minCapacity} \ (x_j) = \min(\text{minCapacity}(x_i), rc_{ij}); \\
\quad \quad \quad \text{status} \ (x_j) = \text{labelled}; \\
\quad \quad \quad \text{if} \ (x_j \neq t) \ \text{then} \\
\quad \quad \quad \quad \text{Add} \ x_j \ \text{to} \ Q; \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \quad \text{return} \ (\text{minCapacity}(t), \text{parent}); \\
\quad \text{return} \ (0, \text{parent}); 
\end{array}$
Appendix B

Detailed discussion of experiments

In this appendix, we discuss the results of all our experiments in detail. First, we discuss the results of Experiment group 1 (Basic case). Then, we discuss the results of Experiment group 2 (A known source of inconsistency). An overview of these experiments, the experimental setup, and our main conclusions have been described in Chapter 4.

B.1 Experiment group 1: Basic case

In Experiment group 1, we evaluated the performance of our algorithms for the basic case, which were described in Section 3.1. This experiment group has been described in Section 4.2.

We implemented and tested the following eight algorithms: NEG-CYCLE (NC), NEG-CYCLE with post-processing (NCPP), the top-down algorithm for the basic case (TD), the top-down algorithm with post-processing for the basic case (TDPP), the top-down algorithm for the basic case removing only negative arcs (TDNA), the top-down algorithm with post-processing for the basic case removing only negative arcs (TDPPNA), the bottom-up algorithm for the basic case (BU), and the bottom-up algorithm for the basic case removing only negative arcs (BUNA).

The test cases we used are random STPs (Experiment group 1A) and practical scheduling problems (Experiment group 1B). For both these groups we conducted the following six experiments:

1. Original benchmark weights
2. Only negative arcs
3. Tight schedule with few inconsistencies
4. Mostly positive arcs with few inconsistencies
5. Tight schedule with one inconsistency source

6. Mostly positive arcs with one inconsistency source

The results of these experiments are shown in Figures B.1 to B.21. In these figures, the total number of arcs in the STP is displayed on the horizontal axis. This number excludes any additional arcs, added by the NEG-CYCLE and top-down algorithms. The number or percentage of changed or removed arcs is displayed on the vertical axis. This means that all data points of one test case are displayed in the same column. Thus, the data points in the figures should be related to points either above or below them (but not to points on their left-hand or right-hand side). Please note that the scales of the axes may vary, depending on the results of the tests. A red line in the figures indicates the total amount of arcs in the STP (i.e., the maximum amount of arcs that can be changed or removed).

1A: Random STPs

Here, we describe the test results for Experiment group 1A: Random STPs. The test cases for these experiments were generated from the DTP benchmark set.
Figure B.1 shows that for random STPs, with weights from the original benchmark instances, the algorithms can roughly be divided into three groups, according to their performance. The group with the worst performance contains the algorithms NEG-CYCLE (NC) and NEG-CYCLE with post-processing (NCPP). The group with the best performance contains the algorithms top-down with post-processing (TDPP), top-down with post-processing removing only negative arcs (TDPPNA), bottom-up (BU) and bottom-up removing only negative arcs (BUNA). The middle group, containing the algorithms top-down (TD) and top-down removing only negative arcs (TDNA), performs better than the NEG-CYCLE algorithms, but worse than the top-down algorithms with post-processing and the bottom-up algorithms.

As we expected, the top-down algorithms with post-processing always find a smaller solution than the top-down algorithms without post-processing. Quite surprisingly, however, even though the NEG-CYCLE algorithm with post-processing is guaranteed to find a minimal solution, it hardly yields any better results than the regular NEG-CYCLE algorithm. Thus, for these test cases, the reason why NEG-CYCLE performs worse than our algorithms does not lie in the fact that it does not guarantee a minimal solution. Rather, we should seek its cause in the additional requirements Dasdan imposed upon the solutions. These requirements imply that in his solutions only the weights of negative arcs are changed, and that these arcs are only changed into zero, instead of completely removed from the graph.
1A.2: Only negative arcs  For random STPs with only negative arcs (and therefore a high degree of inconsistency), Figure B.2 shows a division into the same three groups. Only here, the gaps between the different groups are much larger. Especially the NEG-CYCLE algorithms perform very badly, in many cases even changing the weights of all arcs in the STP. This figure shows that for these test cases, Dasdan’s requirement that negative arcs are never made positive has a very big impact on the size of the solution set. The algorithms that change only the negative arcs have the same performance as the versions that change both positive and negative arcs, which is only natural, since all arcs are negative.
**1A.3: Tight schedule with few inconsistencies**  
Figure B.3 shows a much more chaotic picture for random STP instances with a tight schedule (and thus an almost equal number of positive and negative arcs), and only a few sources of inconsistency. Also here, the NEG-CYCLE algorithms seem to perform worse than the other algorithms. The performance of the other algorithms, however, varies a lot from case to case. This can be explained by the huge randomness of these test cases, which makes it very hard to determine a good strategy for finding those few arcs that are a source of inconsistency.

As the picture is so cluttered, it is very hard to see in this figure whether or not some algorithms perform better than others. Therefore, Figure B.4 shows the same data summarised in a boxplot. In this figure, we can see that also for this type of STPs the ordering we have seen before remains valid. The NEG-CYCLE algorithms perform worst, the top-down algorithms without post-processing perform mediocre, and the top-down algorithms with post-processing and the bottom-up algorithms perform best. For this type of STPs, the bottom-up algorithm performs better than the top-down algorithm with post-processing.
Figure B.4: Summary statistics of Experiment 1A.3: Random STPs with a tight schedule with few inconsistencies. On the vertical axis, the number of changed arcs is displayed as a percentage of the total number of arcs in the STP.
1A.4: Mostly positive arcs with few inconsistencies  

Figure B.5 shows that for STPs with mostly positive arcs and few inconsistencies, in which all negative arcs are part of a negative cycle, most algorithms find the same small amount of arcs. Only the bottom-up algorithm finds much larger solution sets than the other algorithms.

This can be explained by the fact that, unlike the other algorithms, the bottom-up algorithm does not have any information about the location of negative cycles. Thus, it cannot apply this information in its procedure for selecting which arcs to remove from the graph, and which arcs to leave in the graph.
Figure B.6: Experiment 1A.5: Random STPs with a tight schedule with one inconsistency source.

**1A.5: Tight schedule with one inconsistency source**  Figure B.6 shows that for STPs with a tight schedule, in which one single arc is the source of the inconsistency, in many cases the algorithms do not find an optimal solution. Often these non-optimal solutions are much worse than the optimal solutions - even containing up to one third of the total number of arcs in the graph, whereas removing one single arc would have been enough to repair the inconsistency.

From what can be seen in the figure, it seems that overall the NEG-CYCLE algorithms find more non-optimal solutions, and also non-optimal solutions with a lower quality, than the other algorithms. The bottom-up algorithms seem to find less non-optimal solutions, and also the quality of these non-optimal solutions is not as bad as that of the solutions found by the other algorithms.
Figure B.7: Experiment 1A.6: Random STPs with mostly positive arcs with one inconsistency source.

1A.6: Mostly positive arcs with one inconsistency source  Figure B.7 shows that for STPs with only positive arcs, except for a single negative arc that is the source of the inconsistency, almost all algorithms find an optimal solution in all cases. The only exception is the bottom-up algorithm, which in some cases finds a non-optimal solution.

The range of the number of arcs in non-optimal solutions found by the bottom-up algorithm for these test cases, is much smaller than the range of the number of arcs in non-optimal solutions found in Experiment 1A.5 (Random STPs with a tight schedule with one inconsistency source). This can be explained by the fact that the constraints in the previous experiment are much tighter than in this experiment. Thus, a change in one single arc in the previous experiment will often lead to more negative cycles, and also negative cycles with more arcs, than a change in one single arc by a random value in the same range in this experiment.
IB: Practical scheduling problems

Here, we describe the test results for Experiment group 1B: Practical scheduling problems. The test cases for these experiments were generated from the job shop benchmark set. First, we show the summary statistics for this experiment group. Then, we discuss the individual experiments. For each experiment in this group, we added two figures. In the first figure, the absolute number of arcs that have been changed by the algorithms is shown. In the second figure, the same data is displayed. Only here, the number of arcs that have been changed is displayed as a percentage of the total amount of arcs in the STP. This makes it easier to compare the performance of the algorithms for different input sizes.

Summary statistics Figure B.8 shows the summary statistics for Experiment group 1B.1. The results of this experiment group are very similar to the results of Experiment group 1A.1 in Figure 4.2. Two important differences can be observed. In the first place, the results of Experiment 1B.1 (Practical scheduling problems with original benchmark weights) are worse than the results of Experiment 1A.1 (Random STPs with original benchmark weights). Because the STPs of Experiment 1B.1 contain a relative large amount of negative arcs, the results are more similar to those of the STPs with only negative arcs.

In the second place, these summary statistics have much more outliers than the summary statistics of Experiment group 1A. The reason for this is the fact that the smallest test instances in this experiment group are very small compared to the other instances, and the instances of Experiment group 1A. If the total amount of arcs in the graph is very small (i.e., containing only 8 or 9 arcs for the smallest instances), then a solution set that contains only one or two arcs is already relatively large compared to the total number of arcs in the graph.
Figure B.8: Summary statistics of Experiment group 1B. The four groups of algorithms are displayed on the horizontal axis. The changed arcs are displayed on the vertical axis as a percentage of the total number of arcs in the STP.
1B.1: Original benchmark weights  Figures B.9 and B.10 show the results for the practical scheduling problems with the original benchmark weights. As indicated in Section 4.1.3, these instances contain arcs with weights in a relatively small range. Also, they contain a large amount of negative weight arcs, compared to the amount of arcs with a positive weight.

The figures show a division into the same three groups we have seen for the random STPs with only negative arcs. The group with the worst performance contains the algorithms NEG-CYCLE (NC) and NEG-CYCLE with post-processing (NCPP). The group with the best performance contains the algorithms top-down with post-processing (TDPP), top-down with post-processing removing only negative arcs (TDPPNA), bottom-up (BU) and bottom-up removing only negative arcs (BUNA). The middle group, containing the algorithms top-down (TD) and top-down removing only negative arcs (TDNA), performs better than the NEG-CYCLE algorithms, but worse than the top-down algorithms with post-processing and the bottom-up algorithms.

When comparing the results in Figure B.9 to the results of Experiment 1A.1 (Random STPs with original benchmark weights) in Figure B.1, we can see that for the practical scheduling problems with original benchmark weights, the distinction between the three groups is much clearer than for random STPs with original benchmark weights. This can be explained by the nature of these problems. The practical scheduling problems contain a large amount of negative arcs, compared to the amount of arcs with a positive weight. Therefore, the results are more similar to the results of random STPs with only negative arcs (Experiment 1A.2), than to the results of random STPs with random weight arcs (Experiment 1A.1).

When looking at Figure B.10, we can see that for smaller instances, the results of the different algorithms are much closer than for larger instances. The larger the instances become, the bigger the gap between the different groups becomes. This can be explained by the fact that for smaller instances, there is a relatively larger amount of positive arcs compared to the amount of negative arcs, than for larger instances.

A remarkable result for this experiment, is that the top-down algorithm that removes only the negative arcs performs worse than the regular top-down algorithm. This can be explained by the special structure of these instances, in which the positive arcs are often part of many more negative cycles than the negative arcs.

1B.2: Only negative arcs  Figures B.11 and B.12 show almost similar results for practical scheduling problems with only negative arcs. This is in line with our expectations, since the original benchmark already contained mostly negative arcs with small weights.

Compared to the results of Experiment 1B.1 (Practical scheduling problems with original benchmark weights), here the results are even more extreme. Just like we saw for random STPs with only negative arcs in Experiment 1A.2, also here the NEG-CYCLE algorithms often find solution sets containing all arcs in the graph.
Furthermore, there is no difference anymore between the performance of the TD and TDNA algorithms, because now all arcs are negative.

In Figure B.12, the differences between the smaller instances and the larger instances are much smaller than in Figure B.10. Since in this experiment all arcs are negative, the ratio between positive and negative arcs is the same for all test instances. For smaller STPs, the differences between several instances of the same size are still larger than for larger STPs, though. An explanation for this may be, that smaller STPs are more sensitive to specific choices of the random STP generator than larger instances. It could also be the case, however, that there is another cause for these differences. The idea that our explanation may not be sufficient, has been inspired by some remarkable results of Experiment 1B.3, which we have not been able to explain. This experiment will be discussed next.

1B.3: Tight schedule with few inconsistencies

The results for the practical scheduling problems with an almost equal number of positive and negative arcs, and only a few sources of inconsistency, are shown in Figures B.13 and B.14. These figures show a division into roughly the same three groups we have seen before, of NC and NCPP performing worst, TD and TDNA performing a little better, and TDPP, TDPPNA, BU and BUNA performing best, but the differences between these groups are much smaller than in the previous two cases.

However, the picture is a lot less chaotic than the picture for the random STP instances with only a few inconsistencies. This can be explained by the fact that the graphs of these test cases themselves are much more structured, since they represent practical planning problems.

A remarkable result in Figure B.14, is the huge jump in the percentage of arcs changed by (most notably) the NEG-CYCLE algorithms, around STPs with a total number of 500 arcs. For smaller STPs, the percentage of arcs that is changed is much smaller than for larger STPs. We have not been able to explain this result. Further research would be required, to find out what may be the reason for this.

In Figure B.14, we can see that the solutions found by the algorithms, often contain many more arcs than the optimal solution. Even though we do not know the exact amount of arcs in the optimal solution, there is something we can say about its size. When we created the STP instances, we introduced inconsistencies in 2% of the arcs in the graph. Thus, we know that the number of arcs in the optimal solution can be at most 2% of the total amount of arcs in the graph. This means that we have an upper bound for the size of the optimal solution.

In Figure B.14, we can see that none of the algorithms finds optimal solutions. Also, we can see that many algorithms find solutions that are much worse than the optimal solution.
Figure B.9: Experiment 1B.1: Practical scheduling problems with original benchmark weights (absolute number of changed arcs).

Figure B.10: Experiment 1B.1: Practical scheduling problems with original benchmark weights (percentage of the total number of arcs that have been changed).
Figure B.11: Experiment 1B.2: Practical scheduling problems with only negative arcs (absolute number of changed arcs).

Figure B.12: Experiment 1B.2: Practical scheduling problems with only negative arcs (percentage of the total number of arcs that have been changed).
Figure B.13: Experiment 1B.3: Practical scheduling problems with a tight schedule with few inconsistencies (absolute number of changed arcs).

Figure B.14: Experiment 1B.3: Practical scheduling problems with a tight schedule with few inconsistencies (percentage of the total number of arcs that have been changed).
1B.4: Mostly positive arcs with few inconsistencies  Figures B.15 and B.16 show that for practical scheduling problems with mostly positive arcs and few inconsistencies, in which all negative arcs are part of a negative cycle, most algorithms find the same small amount of arcs. Only the bottom-up algorithm finds much larger solution sets. This was also the case for random STPs with mostly positive arcs. A remarkable difference with the random STPs, is that here the top-down algorithm without post-processing also clearly performs worse than the other algorithms. However, its performance is still much better than that of the bottom-up algorithm. The fact that the top-down algorithm without post-processing performs worse than the other algorithms, becomes clearer visible as the STP instances become larger. This may be the reason why its worse performance remained undetected for the random STPs: those STP instances are much smaller than instances of the practical scheduling problems.

The results, which show that the bottom-up algorithm and the top-down algorithm without post-processing perform worse than the other algorithms, can be explained as follows. For these specific test cases, changing only the negative arcs is a very good strategy. Therefore, all algorithms that change only negative arcs perform very well. Even though the top-down algorithm with post-processing does not apply this strategy, it does at some point include many of these negative arcs in its solution set. In the post-processing phase, it will then remove most positive arcs again, because they have become redundant. The top-down algorithm and the bottom-up algorithm, on the other hand, have no means to revert a choice they have made. The top-down algorithm still performs better than the bottom-up algorithm, because, unlike the latter, it can apply information about the location of negative cycles in its selection procedure of which arcs to change.

In Figure B.16, we can see that in this experiment most algorithms do find an optimal (or close to optimal) solution. This can be explained by the fact that, for these instances, all negative arcs are arcs in which an inconsistency was introduced. Those are the only arcs that can be changed by the algorithms that change only the weights of negative arcs, and which will thus be found by these algorithms.

1B.5: Tight schedule with one inconsistency source  Figures B.17 and B.18 show the results for the practical scheduling problems with a tight schedule, in which one single arc is the source of the inconsistency. The results in this figure are very similar to the results of Experiment 1A.5 (Random STPs with a tight schedule with one inconsistency source). Also here, many algorithms do not find an optimal solution. The number of arcs in non-optimal solutions increases as the total number of arcs in the STP increases. It remains around the same percentage of the total amount of arcs in the graph for test instances of different sizes. The NEG-CYCLE algorithms seem to perform worse than the other algorithms, and the bottom-up algorithms seem to perform better than the other algorithms.
Figure B.15: Experiment 1B.4: Practical scheduling problems with mostly positive arcs with few inconsistencies (absolute number of changed arcs).

Figure B.16: Experiment 1B.4: Practical scheduling problems with mostly positive arcs with few inconsistencies (percentage of the total number of arcs that have been changed).
Figure B.17: Experiment 1B.5: Practical scheduling problems with a tight schedule with one inconsistency source (absolute number of changed arcs).

Figure B.18: Experiment 1B.5: Practical scheduling problems with a tight schedule with one inconsistency source (percentage of the total number of arcs that have been changed).
Figure B.19: Experiment 1B.6: Practical scheduling problems with mostly positive arcs with one inconsistency source (absolute number of changed arcs).

**1B.6: Mostly positive arcs with one inconsistency source**  Figures B.19 to B.21 show the results for practical scheduling problems with only positive arcs, except for one single negative arc that is the source of the inconsistency. The results in these pictures are also very similar to the results of Experiment 1A.6 (Random STPs with mostly positive arcs with one inconsistency source). Almost all algorithms find an optimal solution in all cases. The only exception is the bottom-up algorithm, which in some cases finds a non-optimal solution.

In contrast to the results of Experiment 1B.5 (Practical scheduling problems with tight constraints with one inconsistency source), here the number of arcs in non-optimal solutions does not increase as the total number of arcs in the STP increases. Instead, it is bounded above by the very small amount of 5 arcs.

Since the data in Figure B.20 is very hard to read, because of large percentages for very small instances, the values for larger instances are shown again in Figure B.21. In that figure, we can see that the percentage of arcs that has been changed in the non-optimal solutions found by the bottom-up algorithm is very small, and decreases as the instances become larger. Thus, even though the quality of the solutions found by the bottom-up algorithm is worse than the quality of the solutions found by the other algorithms, we conclude that, for these instances, the bottom-up algorithm does not perform that much worse than the other algorithms.
Figure B.20: Experiment 1B.6: Practical scheduling problems with mostly positive arcs with one inconsistency source (percentage of the total number of arcs that have been changed).

Figure B.21: Experiment 1B.6: Practical scheduling problems with mostly positive arcs with one inconsistency source (percentage of the total number of arcs that have been changed (only large instances)). This figure is an enlargement of Figure B.20.
B.2 Experiment group 2: A known source of inconsistency

In Experiment group 2, we evaluate the performance of our algorithm for repairing an STP with a known source of inconsistency, which was described in Section 3.4.3. This experiment group has been described in Section 4.3.

We implemented and tested the following five algorithms: NEG-CYCLE (NC), PRI-CYCLE (PC), the top-down algorithm with post-processing for the basic case (TDPP), the bottom-up algorithm for the basic case (BU), and the KSI algorithm for a known source of inconsistency (KSI).

The test cases we used are random STPs (Experiment group 2A) and practical scheduling problems (Experiment group 2B). For both these groups we conducted the following six experiments:

1. Tight schedule
2. Mostly positive arcs
3. Tight schedule with a very small disruption
4. Mostly positive arcs with a very small disruption
5. Tight schedule with a big disruption
6. Mostly positive arcs with a big disruption

The results of these experiments are shown in Figures B.22 to B.33. In these figures, the total number of arcs in the STP is displayed on the horizontal axis. This number excludes any additional arcs, added by the NEG-CYCLE, PRI-CYCLE and top-down algorithms. The number of changed or removed arcs is displayed on the vertical axis. This means that all data points of one test case are displayed in the same column. Thus, the data points in the figures should be related to points either above or below them (but not to points on their left-hand or right-hand side). Please note that the scales of the axes may vary, depending on the results of the tests. The test instances for which the NEG-CYCLE and PRI-CYCLE algorithms were unable to find a solution that excludes the arc that caused the inconsistency, are omitted from the figures.

We also added summary statistics that show for how many instances the algorithms found a smallest solution among all solutions found by any of the algorithms, for how many instances the algorithms found a larger solution, and for how many instances the algorithms were unable to find a solution excluding the source of inconsistency.
2A: Random STPs

Here, we describe the test results for Experiment group 2A: Random STPs. The test cases for these experiments were generated from the DTP benchmark set. The test cases differ from each other in the weights that were assigned to the arcs. In the first experiment, there is a tight schedule with a single source of inconsistency. In the second experiment, all arcs have positive weights, except for a single negative arc that is the source of the inconsistency. In the subsequent four experiments, we investigate graphs with these two types of weights, and either a very small or a very big disruption.

2A.1: Tight schedule

Figure B.22 and Table B.1 show the results for Random STPs with a tight schedule. These results show that the algorithms can be divided into three groups, according to their performance. The first group, containing the NEG-CYCLE (NC) and PRI-CYCLE (PC) algorithms, performs very badly. The second group, containing the top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU), performs mediocre. The third group, containing only the KSI algorithm (KSI), performs very well.

The NC and PC algorithms have an almost similar performance. They both perform very badly. In 75% of the test instances, they were unable to find a solution that did not include the source of inconsistency. In all other cases, they found a non-optimal solution, which often contained many more arcs than the smallest solution found by any of the algorithms. The algorithms are always unable to find a solution excluding the source of inconsistency for the same test instances. The reason for this is, that these test instances contain at least one cycle for which changing the weights of all other negative arcs on the cycle into zero, is not enough to make the weight of the cycle non-negative. Since the algorithms cannot change the weights of positive arcs, and cannot make any negative arcs positive, changing the weight of the source of inconsistency is the only solution that remains for eliminating the negative cycle.

The TDPP and BU algorithms also have a very similar performance. Their performance differs from case to case. These two algorithms found a smallest solution for approximately one third of the test instances. In case they found a larger solution, however, they often found a solution containing many more arcs than the smallest solution. In general, the quality of the larger solutions found by these algorithms is not as bad as the quality of the larger solutions found by the NC and PC algorithms. Overall, the performance of the TDPP and BU algorithms is very similar.

Our KSI algorithm always finds a smallest solution, thereby clearly performing better than all other algorithms.

2A.2: Mostly positive arcs

Figure B.23 and Table B.2 show the results for Random STPs with only positive arcs, except for one single arc that is the source of inconsistency. These results show that the algorithms can be divided into the
same three groups, according to their performance. The NEG-CYCLE (NC) and PRI-CYCLE (PC) algorithms perform very badly. The top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU) perform mediocre. The KSI algorithm (KSI) performs very well.

As we expected, the NC and PC algorithms never find a solution that excludes the source of inconsistency. In fact, they always find a solution that contains only the source of inconsistency. The TDPP and BU algorithms find a smallest solution in approximately 85% of the test cases. In case they found a larger solution, this solution contained only a few more arcs than the smallest solution. The KSI algorithm always finds a smallest solution.

The performance of the TDPP and BU algorithms for these test cases is a lot better than for the test cases of Experiment 2A.1 (Random STPs with a tight schedule). The reason for this may be that the disruptions that were introduced lead to less negative cycles than in the previous experiment, because the constraints in these

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>19</td>
<td>60</td>
</tr>
<tr>
<td>Larger solution</td>
<td>15</td>
<td>15</td>
<td>47</td>
<td>41</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>45</td>
<td>45</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table B.1: Summary statistics of Experiment 2A.1: Random STPs with a tight schedule.
test cases are not as tight as they were in the previous experiment. In the following experiments, we further investigate whether this could be a good explanation for the differences between the results of these two experiments.

2A.3: Tight schedule with a very small disruption Figure B.24 and Table B.3 show the results for Random STPs with a tight schedule with a very small disruption. These results show that the algorithms can be divided into the same three groups we have seen before. The NEG-CYCLE (NC) and PRI-CYCLE (PC) algorithms perform very badly. The top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU) perform mediocre. The KSI algorithm (KSI) performs very well.

When comparing these results to the results of Experiment 2A.1 (Random STPs with a tight schedule), we can see that the TDPP, BU and KSI algorithms have approximately the same performance for both experiments. The NC and PC algo-
changed arcs

Figure B.24: Experiment 2A.3: Random STPs with a tight schedule with a very small disruption.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>1</td>
<td>6</td>
<td>11</td>
<td>15</td>
<td>60</td>
</tr>
<tr>
<td>Larger solution</td>
<td>44</td>
<td>39</td>
<td>49</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. incon</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table B.3: Summary statistics of Experiment 2A.3: Random STPs with a tight schedule with a very small disruption.

The NC and PC algorithms, however, perform a lot better for this experiment than for Experiment 2A.1. Now, they are unable to find a solution that excludes the source of inconsistency in 25% of the test cases (this was 75% in Experiment 2A.1). Also, the quality of the larger solutions is not as bad as it was before, and in a few cases they were even able to find a smallest solution. This can be explained by the fact that the disruption that was introduced in this experiment was only very small. This means that the negative cycles in this experiment were less negative than in Experiment 2A.1, and therefore, it was more often possible to resolve the inconsistencies by changing only the weights of other negative arcs on the negative cycles into zero (instead of also changing the weight of the source of inconsistency). Yet, even though the performance of the NC and PC algorithms is much better than it was before, it is still a lot worse than the performance of the other algorithms.
Figure B.25: Experiment 2A.4: Random STPs with mostly positive arcs with a very small disruption.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>0</td>
<td>0</td>
<td>51</td>
<td>47</td>
<td>60</td>
</tr>
<tr>
<td>Larger solution</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>60</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table B.4: Summary statistics of Experiment 2A.4: Random STPs with mostly positive arcs with a very small disruption.

**2A.4: Mostly positive arcs with a very small disruption** Figure B.25 and Table B.4 show the results for Random STPs with mostly positive arcs with a very small disruption. These results show that the algorithms can be divided into the same three groups we have seen before. The NEG-CYCLE (NC) and PRI-CYCLE (PC) algorithms perform very badly. The top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU) perform mediocre. The KSI algorithm (KSI) performs very well.

The results of this experiment are very similar to the results of Experiment 2A.2 (Random STPs with mostly positive arcs). This affirms the presumption that the disruption introduced in Experiment 2A.2 was relatively small, given the flexibility of the STP.
Figure B.26: Experiment 2A.5: Random STPs with a tight schedule with a big disruption.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>0</td>
<td>0</td>
<td>22</td>
<td>21</td>
<td>60</td>
</tr>
<tr>
<td>Larger solution</td>
<td>0</td>
<td>0</td>
<td>38</td>
<td>39</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. incons. source</td>
<td>60</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table B.5: Summary statistics of Experiment 2A.5: Random STPs with a tight schedule with a big disruption.

2A.5: Tight schedule with a big disruption  Figure B.26 and Table B.5 show the results for Random STPs with a tight schedule with a big disruption. These results show that the algorithms can be divided into the same three groups we have seen before. The NEG-CYCLE (NC) and PRI-CYCLE (PC) algorithms perform very badly. The top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU) perform mediocre. The KSI algorithm (KSI) performs very well.

When comparing these results to the results of Experiment 2A.1 (Random STPs with a tight schedule) and Experiment 2A.3 (Random STPs with a tight schedule with a very small disruption), we can see that the TDPP, BU and KSI algorithms have approximately the same performance for all experiments. The NC and PC algorithms, however, perform a lot worse for this experiment. Here, they are unable to find a solution that excludes the source of inconsistency for all test cases.

The performance of the NC and PC algorithms, can be explained by the value
that has been chosen for the weight of the arc that has caused the inconsistency. The new weight of this arc has been chosen such that it was less than -(the sum of the absolute value of the weights of all other arcs in the graph), thereby representing a very big disruption. This means that changing the weights of all other negative arcs on a negative cycle into zero will never be enough to compensate for the amount by which the source of the inconsistency has been changed when causing the disruption. Thus, since positive arcs cannot be changed, and negative arcs can never be made positive, the source of the inconsistency is the only arc that is left, whose weight change can resolve the inconsistency.

The results for these test cases differ from the results of Experiments 2A.2 and 2A.4 (Random STPs with mostly positive arcs). In those experiments, the arc that caused the inconsistency was the only arc that was changed by the NC and PC algorithms, because all other arcs were positive and could therefore not be changed. In this experiment, on the other hand, the NC and PC have first tried to resolve the inconsistency by changing the weights of all other negative arcs on the negative cycle. Only after they failed to resolve the inconsistency that way, they have resorted to changing the weight of the source of inconsistency. Therefore, in this experiment they changed the weight of more than just a single arc.

2A.6: Mostly positive arcs with a big disruption

Figure B.27 and Table B.6 show the results for random STPs with mostly positive arcs with a big disruption. Just like in Experiments 2A.2 (Random STPs with mostly positive arcs) and Experiment 2A.4 (Random STPs with mostly positive arcs with a small disruption), the NEG-CYCLE and PRI-CYCLE algorithms are always unable to find a solution that excludes the source of inconsistency, and the KSI algorithm always finds a smallest solution. The results for the top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU), however, differ from the results of the previous experiments.

Both the TDPP algorithm and the BU algorithm found more larger solutions than in Experiments 2A.2 and 2A.4. However, the TDPP algorithm found larger solutions in 45% of the cases, whereas the BU algorithm found larger solutions in 75% of the cases. Also, the quality of the larger solutions found by the BU algorithm is worse than the quality of the larger solutions found by the TDPP algorithm. This may be explained as follows. Although the ratio of positive and negative arcs is the same for this experiment and Experiments 2A.2 and 2A.4, the test instances in this experiment contain more negative cycles than the test instances of the previous experiments. This is the reason why both the TDPP algorithm and the BU algorithm perform worse in this experiment than in the previous experiments. The reason why the TDPP algorithm performs better than the BU algorithm, could be that the TDPP has a means for reverting bad choices in the post-processing phase, whereas the BU algorithm cannot reconsider a bad choice after it has been made.

However, this does not explain the difference between the results of the TDPP algorithm for this experiment and for Experiment 2A.5 (Random STPs with a tight
Figure B.27: Experiment 2A.6: Random STPs with mostly positive arcs with a big disruption.

Table B.6: Summary statistics of Experiment 2A.6: Random STPs with mostly positive arcs with a big disruption.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>0</td>
<td>0</td>
<td>33</td>
<td>15</td>
<td>60</td>
</tr>
<tr>
<td>Larger solution</td>
<td>0</td>
<td>0</td>
<td>27</td>
<td>45</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>60</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

When comparing the results of this experiment to the results of Experiment 2A.5, we can see that the BU algorithm has approximately the same performance for both experiments. The TDPP algorithm, on the other hand, performs much better for this experiment than for Experiment 2A.5. For both these experiments, the weight of the arc that has caused the disruption was chosen such, that all cycles in which this arc participates have become negative cycles. Thus, the differences in the performance of the TDPP algorithm are not caused by any differences in the size of the optimal solution or the number of negative cycles in the graph. Thus, we conclude that the performance of the TDPP algorithm is also influenced by the ratio between positive and negative arcs in the graph, since at this moment that is the only remaining difference we can see between the test cases of these two experiments. The TDPP algorithm does not only perform worse for STPs with many negative cycles than for STPs with only a
few negative cycles, but it also performs worse for STPs with more negative arcs than for STPs with less negative arcs. We have not been able to explain why this is the case, though. Further research will be needed to check the validity of our conclusions, and to find an explanation for these results.

2B: Practical scheduling problems

Here, we describe the test results for Experiment group 2B: Practical scheduling problems. The test cases for these experiments were generated from the job shop benchmark set.

2B.1: Tight schedule

Figure B.28 and Table B.7 show the results for practical scheduling problems with a tight schedule. These results show a division into the same three groups we have seen in the previous experiments. The NEG-CYCLE (NC) and PRI-CYCLE (PC) algorithms perform very badly. The top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU) perform mediocre. And the KSI algorithm (KSI) performs very well.

The results of this experiment are quite similar to the results of Experiment 2A.1 (Random STPs with a tight schedule). The NC and PC algorithms fail to find a solution that excludes the source of inconsistency in 78% of the cases. In almost all other cases, they find a larger solution that often contains many more arcs than the smallest solution found by any of the algorithms. The TDPP and BU algorithms find a smallest solution in 34% of the cases. When they find a larger solution, this solution often contains many more arcs than the smallest solution. The quality of the larger solutions found by the TDPP and BU algorithms, however, is not as bad as the quality of the larger solutions found by the NC and PC algorithms. The KSI algorithm always finds a smallest solution.

In this experiment, we can see how the performance of the algorithms is for STP instances of different sizes. We were unable to make these observations for the random STPs, as they all have approximately the same size. The difference between the number of arcs in the solution found by the KSI algorithm and the larger solutions found by the other algorithms increases, as the size of the test instances increases. The larger solutions found by the NC and PC algorithms are almost always worse than the larger solutions found by the TDPP and BU algorithms.

For both the TDPP and BU algorithms, most of the smallest solutions are found for the smaller test instances. For very small test instances, both algorithms often find a smallest solution. For larger test instances, both algorithms almost always find a larger solution. An explanation for this could be that for very small instances the optimal solution may be the only available solution, whereas for larger instances, there may also be non-optimal alternatives. The more non-optimal alternatives there are, the harder it is to accidentally find an optimal solution.
Figure B.28: Experiment 2B.1: Practical scheduling problems with a tight schedule.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>1</td>
<td>2</td>
<td>41</td>
<td>40</td>
<td>120</td>
</tr>
<tr>
<td>Larger solution</td>
<td>25</td>
<td>24</td>
<td>79</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>94</td>
<td>94</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table B.7: Summary statistics of Experiment 2B.1: Practical scheduling problems with a tight schedule.
2B.2: Mostly positive arcs  
Figure B.29 and Table B.8 show the results for practical scheduling problems with only positive arcs, except for one single arc that is the source of inconsistency. These results again show a division into the same three groups we have seen in the previous experiments. The NEG-CYCLE (NC) and PRI-CYCLE (PC) algorithms perform very badly. The top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU) perform mediocre. And the KSI algorithm (KSI) performs very well.

The results of this experiment are quite similar to the results of Experiment 2A.2 (Random STPs with mostly positive arcs). The NC and PC algorithms never find a solution that excludes the source of inconsistency. In fact, they always find a solution that contains only the source of inconsistency. The TDPP and BU algorithms find a smallest solution in approximately 97% of the test cases. In case they found a larger solution, this solution contained only a few more arcs than the smallest solution. The KSI algorithm always finds a smallest solution.

Compared to the results of Experiment 2B.1 (Practical scheduling problems with a tight schedule), here the quality of the larger solutions found by the TDPP and BU algorithms does not decrease as the size of the test instances increases. An explanation for this could be that for these test instances, the number of negative cycles and the number of arcs in the optimal solution remain quite small, even for larger test instances. Therefore, the number of possible solutions may also be quite small.

2B.3: Tight schedule with a very small disruption  
Figure B.30 and Table B.9 show the results for practical scheduling problems with a tight schedule and a very small disruption. These results again show a division into the same three groups we have seen in the previous experiments. The NEG-CYCLE (NC) and PRI-CYCLE (PC) algorithms perform very badly. The top-down algorithm with post-processing (TDPP) and the bottom-up algorithm (BU) perform mediocre. And the KSI algorithm (KSI) performs very well.

The results of this experiment are very similar to the results of Experiment 2A.3 (Random STPs with a tight schedule with a very small disruption). The NC and PC algorithms are able to find more solutions excluding the source of inconsistency than in Experiment 2B.1 (Practical scheduling problems with a tight schedule).

Quite remarkably, however, the TDPP and BU algorithms seem to find less smallest solutions than in Experiment 2B.1. To rule out any coincidences caused by randomness, we repeated Experiment 2B.1 with the random seed of Experiment 2B.3, and vice versa. In both cases, the BU algorithm found solutions with the same sizes for all test instances. The TDPP algorithm, however, found in both cases less smallest solutions for the STPs with a tight schedule with a very small disruption, than for the STPs with a tight schedule and a little larger disruption. We have not been able to explain why this is the case. Further research would be required to find an explanation for these results.
Figure B.29: Experiment 2B.2: Practical scheduling problems with mostly positive arcs.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>0</td>
<td>0</td>
<td>117</td>
<td>116</td>
<td>120</td>
</tr>
<tr>
<td>Larger solution</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>120</td>
<td>120</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table B.8: Summary statistics of Experiment 2B.2: Practical scheduling problems with mostly positive arcs.
Figure B.30: Experiment 2B.3: Practical scheduling problems with a tight schedule with a very small disruption.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>8</td>
<td>10</td>
<td>22</td>
<td>33</td>
<td>120</td>
</tr>
<tr>
<td>Larger solution</td>
<td>69</td>
<td>67</td>
<td>98</td>
<td>87</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>43</td>
<td>43</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table B.9: Summary statistics of Experiment 2B.3: Practical scheduling problems with a tight schedule with a very small disruption.
2B.4: Mostly positive arcs with a very small disruption  

Figure B.31 and Table B.10 show the results for practical scheduling problems with only positive arcs, except for one single arc that is the source of inconsistency, with a very small disruption.

A very interesting result of this experiment, is the fact that here both the TDPP and the BU algorithms have an almost perfect score. Only the BU algorithm found a larger solution for one of the test instances. Thus, for this type of STPs, with very loose schedules and very small disruptions, not only the KSI algorithm, but also the TDPP and BU algorithms are very well usable.
Figure B.32: Experiment 2B.5: Practical scheduling problems with a tight schedule with a big disruption.

Table B.11: Summary statistics of Experiment 2B.5: Practical scheduling problems with a tight schedule with a big disruption.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>0</td>
<td>0</td>
<td>42</td>
<td>44</td>
<td>120</td>
</tr>
<tr>
<td>Larger solution</td>
<td>0</td>
<td>0</td>
<td>78</td>
<td>76</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>120</td>
<td>120</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

2B.5: Tight schedule with a big disruption  Figure B.32 and Table B.11 show the results for practical scheduling problems with a tight schedule with a big disruption. The results of this experiment are very similar to the results of Experiment 2A.5. Also here, the NEG-CYCLE and PRI-CYCLE algorithms fail to find a solution excluding the source of inconsistency in all cases. The top-down algorithm with post-processing and the bottom-up algorithm find a smallest solution in approximately 36% of the cases. The KSI algorithm finds a smallest solution in all cases.

2B.6: Mostly positive arcs with a big disruption  Figure B.33 and Table B.12 show the results for practical scheduling problems with mostly positive arcs with a big disruption. These results are very similar to the results of Experiment 2A.6 (random STPs with mostly positive arcs with a big disruption). Also here, the
Figure B.33: Experiment 2B.6: Practical scheduling problems with mostly positive arcs with a big disruption.

Table B.12: Summary statistics of Experiment 2B.6: Practical scheduling problems with mostly positive arcs with a big disruption.

<table>
<thead>
<tr>
<th></th>
<th>NC</th>
<th>PC</th>
<th>TDPP</th>
<th>BU</th>
<th>KSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest solution</td>
<td>0</td>
<td>0</td>
<td>74</td>
<td>39</td>
<td>120</td>
</tr>
<tr>
<td>Larger solution</td>
<td>0</td>
<td>0</td>
<td>46</td>
<td>81</td>
<td>0</td>
</tr>
<tr>
<td>Solution incl. inconsistency source</td>
<td>120</td>
<td>120</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NEG-CYCLE and PRI-CYCLE algorithms are always unable to find a solution that excludes the source of inconsistency, the top-down algorithm with post-processing and the bottom-up algorithm find smallest solutions for only a subset of the test cases, and the KSI algorithm always finds a smallest solution. And just like in Experiment 2A.6, the top-down algorithm with post-processing (TDPP) performs better than the bottom-up algorithm (BU).

We assume that the differences between the performance of the TDPP algorithm and the BU algorithm can be explained in a similar way as for Experiment 2A.6. Apparently, the BU algorithm performs worse as the number of negative cycles in the graph increases. The TDPP algorithm, on the other hand, keeps performing relatively well, as long as the number of negative arcs in the graph remains low.