Re-scheduling the Railway Traffic using Parallel Simulated Annealing and Tabu Search:
A comparative study

Sebastian Boman & Petrit Gerdovci

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden
This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Engineering: Computer Science and Engineering. The thesis is equivalent to 20 weeks of full-time studies.

Contact Information:
Author(s):
Petrit Gerdovci
E-mail: gerdovci90@gmail.com

Sebastian Boman
E-mail: sebastian.boman.91@gmail.com

University advisor:
Prof. Håkan Grahn
Faculty of Computing

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57
Abstract

Context. This study has been conducted in the area of train re-scheduling. One of the most common types of disturbance scenarios are trains which have deviated from their originally planned arrival or departure times. This type of disturbance is of today handled manually by the train dispatcher, which in some cases can be cumbersome and overwhelmingly complex to solve. Therefore, there is an essential need for a train re-scheduling decision support system.

Objectives. The aim of the study is to determine if parallel adaptations of simulated annealing (SA), and tabu search (TS) are able to find high quality solutions for the train re-scheduling problem. The study also aims to compare the two proposed meta-heuristics in order to determine the more adequate algorithm for the given problem.

Methods. To answer the research question sequential and parallel versions of the algorithms were implemented. Further the research methodology of choice was experiment, were the meta-heuristics are evaluated based on 10 disturbance scenarios.

Results. Parallel simulated annealing (PSA) is overall the better performing algorithm, as it is able to reduce the total delay by 585 seconds more than parallel tabu search (PTS) for the 10 disturbance scenarios. However, PTS is able to solve more conflicts per millisecond than PTS, when compared to their sequential versions.

Conclusions. We conclude that both the parallel versions perform better than their sequential versions. Further, PSA is clearly able to outperform PTS in terms of minimizing the accumulated delay. One observation is that the parallel versions are not reaching their max efficiency per thread, this is assumed to be caused by the RAM. For future work we propose further investigation of why we are not reaching the max efficiency per thread, and further improvement of algorithm settings.

Keywords: Parallel Computing, tabu search, simulated annealing, train re-scheduling.
# Contents

Abstract i

1 Introduction 1
   1.1 Contributions and current limitations ................. 2
   1.2 Thesis Outline ..................................... 3

2 Problem Formulation 5
   2.1 Aims and Objectives ............................... 5
   2.2 Research Questions ................................ 6
   2.3 Research Methodology ............................... 6
      2.3.1 Implementation ................................ 7
      2.3.2 Experiment ..................................... 7
      2.3.2.1 Experimental Design ......................... 8
      2.3.2.2 The timetable and infrastructure data-set ... 9
   2.3.3 Data collection .................................. 9
   2.3.4 Data analysis and Synthesis ...................... 9

3 Graph Theory and Job Shop Scheduling 11
   3.1 Basics on Graph Theory ............................ 11
      3.1.1 Neighbourhood ................................. 11
      3.1.2 Transitions ................................... 12
   3.2 Job Shop Scheduling ................................ 13
      3.2.1 The no-wait job shop ............................ 14
      3.2.2 Disjunctive graph ............................... 14
      3.2.3 Methods for solving the job shop problem .... 16
      3.2.3.1 Branch-and-bound ............................ 17
      3.2.3.2 Shifting bottleneck heuristic ............... 17
      3.2.3.3 Heuristic Methods ........................... 18

4 The Swedish Railway Network 19
   4.1 Definition of Terms and Metrics in the Railway Network .... 19
   4.2 The Railway Timetable .............................. 21
      4.2.1 Train Delays ................................. 21
      4.2.1.1 Cost ..................................... 22
4.3 Solving train conflicts .................................................. 23
  4.3.1 Meet conflict ......................................................... 23
  4.3.2 Pass conflict ......................................................... 23
  4.3.3 Capacity conflict .................................................. 24

4.4 Mapping to the JSSP .................................................... 25

4.5 The Railway Conflict Graph .......................................... 26
  4.5.1 The solution space ................................................ 26

4.6 Train Re-scheduling Strategies ...................................... 27

5 Meta heuristics used in railway scheduling .......................... 30
  5.1 Simulated Annealing .................................................. 30
    5.1.1 Historical Background ........................................ 30
    5.1.2 Algorithm Description ....................................... 30
      5.1.2.1 Cooling Schedule ...................................... 31
      5.1.2.2 Equilibrium ............................................ 32
  5.2 Tabu Search .......................................................... 32
    5.2.1 Historical Background ........................................ 32
    5.2.2 Algorithm Description ....................................... 33
      5.2.2.1 Tabu List and Memory Structures ...................... 34
      5.2.2.2 Aspiration Criteria .................................... 35
  5.3 Genetic Algorithm .................................................. 35
    5.3.1 Historical Background ........................................ 35
    5.3.2 Algorithm Description ....................................... 36
      5.3.2.1 Encoding ............................................... 37
      5.3.2.2 Selection ............................................... 37
      5.3.2.3 Crossover ............................................... 38
      5.3.2.4 Mutation ............................................... 38

6 Distributed and Parallel Computing .................................. 39
  6.1 Computer architectures ............................................ 39
  6.2 Shared memory systems ............................................ 41
  6.3 Distributed memory systems ........................................ 42
  6.4 Parallel Simulated annealing and Tabu search .................... 43
    6.4.1 Parallel TS .................................................... 43
      6.4.1.1 Parallel cost function evaluation and neighbourhood 44
      6.4.1.2 Parallel decomposition of the problem ............... 44
      6.4.1.3 Parallel solution domain exploration ............... 44
    6.4.2 Parallel SA .................................................... 45
      6.4.2.1 Simultaneous independent searches ................... 45
      6.4.2.2 Simultaneous corresponding searches ................ 46
        6.4.2.2.1 Temperature updating strategies ............... 46
7 Related Work
7.1 Job Shop Scheduling ........................................ 47
7.2 Meta heuristics applied in Railway scheduling .......... 47

8 Implementation
8.1 System Overview ............................................ 50
8.2 Data loading ................................................ 51
  8.2.1 Train timetable ........................................ 52
  8.2.2 Railway Infrastructure ................................. 52
8.3 Conflict Detection and Resolution loop(CDRL) .......... 53
  8.3.1 Conflict Detection ..................................... 54
    8.3.1.1 Inserting Subsequent Events ..................... 54
    8.3.1.2 Checking for Conflicts .......................... 54
    8.3.1.3 Description of Conflict detection algorithms . 55
  8.3.2 Conflict Resolution ................................... 56
    8.3.2.1 Candidate List Sorting Strategy .................. 56
    8.3.2.2 The Conflict Resolution Table and the Neighbour-
      hood .................................................. 57
    8.3.2.3 Description of Conflict repair algorithms ....... 57
8.4 Heuristic Based Resolution(HBR) .......................... 59
  8.4.1 Cost Function .......................................... 60
  8.4.2 Parallel Simulated Annealing Algorithm ............. 60
  8.4.3 Parallel Tabu Search Algorithm ....................... 65
  8.4.4 System Restrictions ................................... 68
    8.4.4.1 Train restrictions .............................. 68
    8.4.4.2 Infrastructure restrictions ...................... 69
8.5 Output and System Validation ............................... 69
  8.5.1 Timetable Conflict detection .......................... 69
  8.5.2 Timetable Validation of the train events ............. 69
  8.5.3 Timetable No wait constraint .......................... 69

9 Results .......................................................... 70
9.1 Experiment Details ......................................... 70
  9.1.1 Algorithm parameter settings ........................ 72
9.2 Experimental Results ....................................... 72
  9.2.1 Model Evaluation ...................................... 73
  9.2.2 Meta-heuristics Evaluation ............................ 73
    9.2.2.1 Meta-heuristic improvement from initial solution 74
    9.2.2.2 Conflicts resolved by sequential and parallel im-
      plementations ....................................... 78
  9.2.3 Memory Usage .......................................... 78
10 Analysis

10.1 Reflection of the model ........................................ 80
10.2 Reflection of the meta-heuristics ................................. 80
   10.2.1 Solution Optimality evaluation ........................... 81
      10.2.1.1 Tabu Search ........................................ 81
      10.2.1.2 Simulated annealing ................................. 81
      10.2.1.3 Tabu search and Simulated Annealing ............... 82
10.2.2 Improvement from initial solution evaluation .............. 83
10.2.3 Conflicts solved evaluation ................................ 83
10.2.4 Memory Usage evaluation ................................. 84

11 Conclusions and Future Work ................................. 85
   11.1 Conclusion .................................................. 85
   11.2 Future Work ................................................ 86

References ...................................................... 88
## List of Figures

2.1 Flowchart of the research methodology ........................................... 6  
3.1 Directed Graph ................................................................. 12  
3.2 Transition Example ............................................................ 13  
3.3 No wait constraint ............................................................... 14  
3.4 Disjunctive Graph example ..................................................... 15  
4.1 Overview of surveillance area Björnfjell-Riksgränsen-Kiruna-Boden- 
Luleå ................................................................. 20  
4.2 Depiction of a meet conflict and solutions ................................... 23  
4.3 Depiction of a pass conflict and solutions ................................... 24  
4.4 Depiction of a capacity conflict and solutions ............................. 24  
4.5 A illustration of a timetable with one conflict at station S2-S3 ........ 26  
4.6 Depiction of a conflict graph with one conflict ............................ 27  
5.1 Example of a roulette wheel selection ....................................... 37  
6.1 Classifications of computer architectures ................................. 40  
6.2 Example of a multiprocessor system ....................................... 41  
6.3 Example of a distributed memory system ................................. 43  
8.1 Train re-scheduling system overview ...................................... 51  
8.2 Overview of system parser ................................................... 51  
8.3 Overview of the re-scheduling process. Depicting the active, candidate, 
and train event lists ....................................................... 53  
8.4 Illustration of Conflict resolution table(CRT) ............................ 57  
9.1 Overview over the improvement of cost for SA for scenario 1 ....... 75  
9.2 Overview over the improvement of cost for TS for scenario 1 ....... 75  
9.3 Overview of improvement from initial of the meta-heuristics ......... 77  
9.4 PSA Memory Usage for an instance of Scenario 1 ....................... 78  
9.5 PTS Memory Usage for an instance of Scenario 1 ....................... 79
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Task chain and processing times</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>A timetable with one conflict</td>
<td>25</td>
</tr>
<tr>
<td>6.1</td>
<td>Flynn’s taxonomy</td>
<td>39</td>
</tr>
<tr>
<td>6.2</td>
<td>Race condition</td>
<td>42</td>
</tr>
<tr>
<td>9.1</td>
<td>Experimental disturbance scenarios</td>
<td>71</td>
</tr>
<tr>
<td>9.2</td>
<td>Model and system performance</td>
<td>73</td>
</tr>
<tr>
<td>9.3</td>
<td>Overview of Solution Optimality for SA</td>
<td>74</td>
</tr>
<tr>
<td>9.4</td>
<td>Overview of Solution Optimality for TS</td>
<td>74</td>
</tr>
<tr>
<td>9.5</td>
<td>Overview of the number of conflicts solved per millisecond (ms) for the meta-heuristics</td>
<td>78</td>
</tr>
</tbody>
</table>
List of Algorithms

1. Generic Simulated Annealing ............................................. 31
2. Generic Tabu Search ......................................................... 33
3. Get Best Neighbour (GBT) ................................................... 34
4. Generic Genetic Algorithm (GA) ......................................... 36
5. Step to conflict (STC) algorithm ........................................ 55
6. Check conflicted events (CCE) algorithm .............................. 56
7. Greedy Solver algorithm ................................................... 58
8. Repair conflict (RC) algorithm ............................................ 59
9. Parallel SA Initialization .................................................... 61
10. Parallel Simulated Annealing (PSA) .................................... 62
11. Process Communication .................................................... 64
12. Generate random neighbour ............................................. 65
13. Parallel TS Initialization ................................................... 66
14. Parallel Tabu Search (PTS) ............................................... 67
15. Parallel Tabu Search Get Best Neighbour ............................ 68
Chapter 1

Introduction

This thesis is conducted in the area of optimization. More precisely in the area real-time re-scheduling of trains during disturbances. The growth of freight- and passenger train traffic, has resulted in an increasing complexity when scheduling existing railroad traffic. One of the most common type of disturbance in the railroad are trains which have deviated from their originally planned arrival time.

Even with proper timetables which take buffer times into consideration, unpredictable and small delays can in some cases lead to a high accumulated delay in the overall railway system. Buffer times are used to improve the robustness of timetables, it can be considered as a small time added in the timetable in order to absorb small delays. Late departed trains can take advantage of buffer times to reduce the risk of introducing further disturbances in the railway network. The buffer times simultaneously reduce the overall efficiency of the railway network, as trains do not take advantage of the buffer times, unless the train has been delayed.

In the year 2014, 78.1% of all freight and 90.3% of all passenger trains had a delay of more than 5 minutes. The Swedish transport administration is eager to minimize both the occurrence and the amount of delay for all railway traffic. Disturbances in the timetable have to be dealt with effectively by re-defining the timetable to reduce the overall delay and therefore the associated penalty costs for operators and infrastructure providers. The reason why this poses a problem is due to the fact that the task of re-scheduling is complex and time consuming. The railway network and its constraints result in a problem which is not solvable in linear time, which means it is not feasible to evaluate all possible solutions. The problem is further considered to be NP-hard. Therefore, in this thesis we are concerned with finding the best solution during a certain time frame.

The increased demand of track capacity has lead to problems for the Swedish railway operators and they are in need of a decision support system (DSS), which can assist the train dispatchers to manage the delays and make intelligent decisions in real-time. As of today, railway traffic administrations are challenged with achieving high punctuality and ensuring a safe traffic system for attractive freight and passenger transport services. Re-scheduling is still today done manually without any computational DSS, despite the evident benefits of past research.
efforts in academia.

In the thesis an optimization-based re-scheduling support system for railway traffic networks is implemented. The DSS system uses two meta-heuristic search functions: Simulated annealing, and tabu search. These heuristics are used in order to find optimized timetables within a reasonable time. We further optimize the two meta-heuristics by adopting parallel computing, this is to evaluate if timetables of higher quality can be found within the same given time frame. In this thesis, finding solutions of higher quality using parallel techniques is referred to as improving solution optimality.

The development of the real-time railway traffic re-scheduling DSS is composed of a set of challenges:

(a) Modeling the re-scheduling system to emulate the real world railway. This requires to determine which set of attributes are of relevance, and how to solve different disturbances scenarios by redefining the timetable.

(b) As the problem is NP-Hard, we need to limit the search space to areas of relevance (i.e. prune the search space).

(c) We are faced with the problem of how to adopt the meta-heuristics for the given problem. Furthermore the main focus and problem to be tackled is how parallelism can be achieved for the meta-heuristics.

1.1 Contributions and current limitations

In our thesis we will investigate how meta-heuristic search functions perform, in a simplified model of the single track railway re-scheduling problem. The problem in Sweden’s railway is very specific due to unique properties which increase the complexity. Two properties can be delimited which make the Swedish railway network different from other countries. The railway allows for both freight and passenger trains to share the same railway network, although this property exists in some other European countries. The surveillance area has ore trains which are long and heavy, and which we generally do not want to stop during transport.

Furthermore, we are addressing a problem which has not been evaluated with the usage of parallel implementations of Tabu search (TS) and Simulated Annealing (SA). Thus, parallel versions of the two proposed meta-heuristics have in past academic work never been compared in the area of train re-scheduling. This will address the uncertainty if adopting parallelism can outperform the sequential versions, in terms of minimizing the accumulated delay. This thesis further gives insight on how the parallel and sequential versions of the SA and TS algorithms, compares to a simple greedy solver. Past academic work show heuristic approaches are among the most widely used methods for train re-scheduling[20].
Moreover, results obtained in studies evaluating TS and SA exclusively might not yield the same outcome as our proposed study. Due to how the constraints for the problem have been set, how we have chosen to model the problem, and how the search space has been pruned. Therefore, we need to address this uncertainty and compare the two heuristics within this unexplored research area of parallel computing. As concluded in [20], these algorithms are widely used heuristics when re-scheduling railway networks. Thus, due to SA and TS popular usage and compelling results, make them highly adequate for addressing the proposed problem.

With the problem and challenges formally defined, we in the following present the structure of the thesis.

1.2 Thesis Outline

The thesis presents a model which maps the single track railway re-scheduling problem into a special case of the job shop scheduling problem. The mapping allows us to benefit from the numerous of methods used to solve the job shop scheduling problem. In addition to the model, two meta-heuristics have been successfully applied to the re-scheduling problem. Both sequential and parallel version of simulated annealing and tabu search are in this thesis evaluated.

The thesis along with providing new research in the area of train scheduling, provides the concepts of how to adopt the meta-heuristics for the problem tackled here or similarly complex problems. In the following we present the highlights of each thesis chapter:

**Chapter 1:** Introduces the railway re-scheduling problem, and also gives the outline for the thesis. The chapter also gives the reader insight to the current limitations of the presented academic work, and the contributions of the thesis.

**Chapter 2:** The problem is in this chapter formally described, and the research method used in this thesis is presented.

**Chapter 3:** Gives the reader insight to the graph theory, and the job shop scheduling problem. The chapter concludes with discussing common methods for solving the job shop scheduling problem.

**Chapter 4:** Describes the characteristics of the railway network. Further the chapter later describes how the railway re-scheduling problem can be modelled as a special case of the job shop scheduling problem. The chapter gives a overview of the literature of different cost functions, and railway re-scheduling strategies.
Chapter 5: Introduces the meta heuristics: Simulated annealing, tabu search, and genetic algorithm. For each heuristic a brief historical overview is given, further descriptions of the methods are given in detail.

Chapter 6: Describes two different parallel computing techniques: distributed, and shared memory. The chapter later discusses numerous of widely used parallel adaptations of simulated annealing and tabu search.

Chapter 7: Gives the reader insight of past academic work in the area of train scheduling and the job shop problem.

Chapter 8: Describes the implementation of the proposed model, and how the meta-heuristics have been adopted to the model. The choices along the thesis are in this chapter motivated.

Chapter 9: Presents the results from the parallel and sequential implementation for 10 different selected disturbance scenarios.

Chapter 10: This chapter presents the analysis of the experiments. We further in this chapter compare both the algorithms against each other for a set of given test cases.

Chapter 11: In this final chapter we summarize the highlights of the thesis. The developed meta-heuristics are in this chapter discussed, and future research is proposed.
Chapter 2

Problem Formulation

The following chapter introduces the formal aims and objectives of the thesis. Further the research methodology is presented, which is to be applied in order to answer our research questions.

2.1 Aims and Objectives

The main aim of this study is to evaluate if parallel implementations of simulated annealing and tabu search, are able to find a feasible solutions within a short computation time for train re-scheduling.

Real time re-scheduling of delayed trains in railroads is a necessity. As of today, railway dispatchers are re-scheduling delayed trains manually. Thus, they are in need of real-time railway rescheduling applications. In order to accomplish our main objective, a number of sub-objectives need to be completed beforehand. These sub-objectives are as follows:

- Analyze and determine specific characteristics of the Swedish railways to design the simplified model.
- Design the model based on the set of determined attributes.
- Implement a model of the railway infrastructure, and validate the model.
- Determine how simulated annealing and tabu search can be adopted for the problem.
- Implement sequential and parallel versions of simulated annealing and tabu search.
- Investigate how both algorithms can be optimized for the specific model and get a better understanding for the search space generated by the model.
- Compare both algorithms with each other.
2.2 Research Questions

From the previous section, we discussed our goal was to implement a simplified model of the railway problem. Thereafter, evaluate the heuristics simulated annealing (SA) and tabu search (TS) on the model. This leads us to our research questions:

**RQ1** - How can railway re-scheduling be done effectively in real-time, with the usage of parallel simulated annealing (PSA) and tabu search (PTS)?

**RQ1.1** - What is an efficient approach of adopting parallel computing for simulated annealing and tabu search to improve solution optimality?

**RQ1.2** - How does the solution optimality of PSA compare to PTS?

2.3 Research Methodology

In this thesis, an implementation and experiment will be conducted. It is not trivial as how to adopt the proposed meta-heuristics. In order to tackle this problem, a literature survey will be conducted. In the literature survey we will further focus on which parallel approaches are feasible to adopt within the scope of this master thesis. Our research is divided into five steps, as depicted in figure 2.1.

![Figure 2.1: Flowchart of the research methodology](image)

In the illustrated process, we have an iterative process, were we study literature to identify how to adopted the meta-heuristics for railway re-scheduling.
Further we study different parallel techniques which could be applied to the algorithms. If the implementation was successfully carried out we move to the experiments. Otherwise we go back to continuing implementing the system. The data collected from the experiments then lay basis for drawing our conclusions and answering the research questions.

In modern science, it is of significance for the results of experiments to be reproducible. Lewis et al. [41] claims:

“The use of precise, repeatable experiments is the hallmark of a mature scientific or engineering discipline” Lewis, 1991

Therefore, a careful experimental design is a necessity for future scientist to reproduce and verify our results. In the following we present what research methodologies will be applied in the different parts of the study.

2.3.1 Implementation

In regards to the implementation, the aim is to improve the proposed heuristic search functions (i.e. TS and SA). This will be done by adopting parallel computing techniques, such as multi-threading. The goal of the implementation will be to evaluate if we are able to find better solutions using a multi-threaded approach. Improving solution optimality of these heuristics, is of importance. Since this will allow for real-time re-scheduling of trains by minimizing the total delay. Furthermore we will be implementing the actual system for reading the train timetables and infrastructure of the railway network. The implementation of the system will lay the basis (i.e. back-end) in order to evaluate the both algorithms. Moreover, we need to identity conflicts in the train timetable. By identifying the various of conflicts which have occurred, we are able to build parts of the search space for finding feasible solutions. Performance of the heuristics search functions, will be evaluated based on which algorithm finds the best solution within a pre-defined execution time frame.

2.3.2 Experiment

The choice of research methodology, as stated, is experiment. It is chosen since it is the dominant research methodology in the area of parallel computing and optimization. Moreover there exists an uncertainty of which of the heuristics function are more adequate for the railway re-scheduling problem. To address this uncertainty, we might in combination with observation, discover unexpected and useful results. Which would later open doors for new areas of which research is needed. The metrics which are useful to consider in this study, and (in fact) are measurable have been identified as:
Chapter 2. Problem Formulation

- **The accumulated delay for trains** - The solution to the train re-scheduling problem is the new generated time-table. Each of the algorithms are to determine a feasible conflict free time-table. The accumulated delay for trains in the time-tables are to be compared, in order to determine which algorithm finds the better solution.

- **Processing time** - The processing time (CPU time) can be considered as the time of an algorithm given certain input. Measurements on the CPU time is difficult and can be unreliable, due to different reasons: CPU speed, sizes of caches, and the load of the system. In this thesis our focus lies in measuring CPU time for the time needed to find a feasible solution, and the time taken to build the model.

- **Memory requirements** - The random-access memory (RAM), can be considered as data storage units. The process of re-scheduling the train traffic is heavily demanding on the RAM. It is therefore useful to consider memory as an metric when evaluating the algorithms.

These metrics will be used when evaluating performance of the algorithms. The accumulated delay metric is chosen since we need to re-schedule delayed trains in real-time. The reason choosing experiment as our research methodology over a survey, or case study is that they do not apply in this context.

### 2.3.2.1 Experimental Design

The specifications of the PC hardware, which will be used for the experiment, are as follows:

- Intel i7-3630QM processor
- 8GB RAM
- 500GB hard drive

The implementation will be done using Java SE JDK 1.8, we will moreover use the built in multi-threaded API to achieve higher performance. The experiments will be conducted on the operating system (OS) Ubuntu 14.10.

The execution time of the algorithms will be limited to a certain time limit (e.g. 30 seconds). We will afterwards evaluate the results of the algorithms based on the best solution. Meaning the algorithm which has the lowest accumulated delay, is the better performing one.

JVM monitor will be used to monitor how much memory is being used by the application, in order to give estimates of peak memory and overall memory usage. Other tools of less significance which will be used are Python and Bash script, which are useful for unit-testing the output timetable.
2.3.2.2 The timetable and infrastructure data-set

The input data which will be used for our experiments is represented in two separate data sets. One describes the rules for how trains are allowed to travel, i.e., how the railway network is structured. The second data set describes how and when the trains are supposed to move forward and stop in the railway network. The network data set contains information regarding locations (travel destinations) and how they are interconnected. The infrastructure further shows which track segments are placed between different locations, this is denoted as network links (NWK). This makes it possible to view the locations as nodes and the network links as arcs in a network graph. The train movements are described by which train should depart and arrive from which location. This is done by defining train stops and train movements separately.

2.3.3 Data collection

From conducting the experiment we will gather the results of the sequential and parallel versions of the algorithms. Our data collection, will primarily focus on gathering the accumulated delay for the algorithms during a defined time limit. Moreover, we will also gather the number of conflicts solved by each of the algorithms. In order to draw conclusions of how efficient the parallel implementations are compared to the sequential ones. Comparing the number of conflicts resolved by each algorithm, illustrates roughly how much work each of the implementations are performing.

Furthermore, we will evaluate the proposed algorithms for multiple runs and in this thesis 100 runs were chosen. This is due to the algorithms being random, and 100 runs gives a more accurate representation of the actual cost of the solutions found. Applying parallelism, might not yield a better solution within a reasonable time. Therefore, applying parallel computing is of importance to address this uncertainty.

Finally we will evaluate the model, in terms of how long it takes to build the model as well as the time it takes to generate an initial solution. The initial solution generated by the greedy solver will later on be compared to the meta-heuristics search functions, in order to demonstrate how the heuristics improve the solution.

2.3.4 Data analysis and Synthesis

When the data from the experiments have been gathered, a thorough evaluation will be performed. Our primarily focus will be to evaluate the optimality of the solutions found by the heuristics for a certain time frame. This will be accomplished by gathering the results of both the sequential and parallel versions
into tables and graphs, whereby we can illustrate the potential benefits of the meta-heuristics.

The heuristics are abbreviated by the usage of a capital letters in the beginning to denote the version, meaning: sequential(S) and parallel(P). We will then analyse the results for the following versions and experiments:

(a) SSA vs PSA
(b) STS vs PTS
(c) PTS vs PSA

The objective with the application of parallel computing, is to evaluate if we are able to find better solutions than the sequential versions of the meta-heuristics. Finally with the results we will be able to answer our research questions.
Chapter 3
Graph Theory and Job Shop Scheduling

This chapter gives the reader an insight to the classical Job Shop Scheduling problem, and graph theory. This prior knowledge is a necessity for the reader to be aware of before continuing with the rest of the paper.

3.1 Basics on Graph Theory

The concept of graph theory was first introduced by the Swiss mathematician Leonhard Euler[19]. In Euler’s study, he solves a bridge crossing problem in the city of Königsberg. In the context of graph theory, a graph \( G \) comprises of a set of vertices \( V \) and a set of edges \( E \). The vertex is drawn as a dot or node, the edges connects two vertices in the form of a line. For instance in figure 3.1, the set of \( V \) can be formulated according to equation 3.1.

\[
V = \{A, B, C, D, E\} \quad (3.1)
\]

The set of \( E \), can be formulated according to equation 3.2.

\[
E = \\{\{A, B\}, \{A, C\}, \{A, D\}, \{B, D\}, \{C, D\}, \{D, E\}\} \quad (3.2)
\]

Unitedly \( V \) and \( E \) form the graph. There exists two types of graphs, directed and un-directed graphs. In the illustrated example we show a directed graph, meaning the edges are not bi-directional.

3.1.1 Neighbourhood

In graph theory the neighbourhood of vertex \( V \), is the adjacent vertices connected to \( V \) by the edges. In figure 3.1, the neighbourhood of vertex \( A \) are the vertices \( B, C, D \).
In the aforementioned example, the neighborhood does not contain the vertex $A$ itself. This is said to be an open neighborhood. It is further possible to include the vertex $A$, this is referred to as an closed neighborhood. The neighborhood of $V$ is denoted as $N(V)$.

### 3.1.2 Transitions

A transition\(^1\) perturbs one solution to another, within the same neighborhood. A solution consists of a set of $E$, the cost of the solution is donated by the weight $W$ of the set (equation 3.3).

\[
\sum_{i=m}^{n} W_i
\]  

(3.3)

In figure 3.1, say node $A$ is the start node and $E$ is the goal node. A solution $S$ can thereby be formulated as follows, $S = \{A, B, D, E\}$. A transition can described with the following characteristics:

- Edges which two solutions have in common, $(S \cap S')$.
- Edges which have been deleted from $S$, $(S \setminus S')$.
- Edges which have been added in the permuted solution $S'$, $(S' \setminus S)$.

From figure 3.2, we can demonstrate a concrete example of how a transition from a initial solution $S$, to a new solution $S'$ appears in a simple graph. The dashed lines illustrate potential transitions from one vertex. The filled line illustrates our current solution (figure a). Choosing a different vertex in the neighbourhood, yields a new solution (figure b). From figure c we show which edges have been added, and the edges which have been removed in solution $S'$.

\(^1\)Also denoted move or alternatively change.
3.2 Job Shop Scheduling

Scheduling can be considered as a decision-making process, in which we are concerned with allocating limited resources to tasks over time. There exists no exact definition of resources and task, regarding the subject of scheduling. These may vary depending on the problem at hand. In the manufacturing context, the resources can be machines. In this thesis resources are considered as train tracks, moreover tasks are considered as train trips which are to be scheduled.

The Job Shop Scheduling Problem (JSSP), is considered to be a classical NP-hard[40] problem. The JSSP is closely related to the process of re-scheduling trains, and therefore receives attention in this paper. From Zhengcheng and Shuang[65], the problem can be formulated as following. We have \( n \) set of jobs, which are denoted by equation 3.4.

\[
J = \{ J_1, J_2, ..., J_n \} \tag{3.4}
\]

Each job is to be processed by one machine, denoted by equation 3.5.

\[
M = \{ M_1, M_2, ..., M_n \} \tag{3.5}
\]
Each machine is associated with a set of tasks, as denoted by constraint 3.6.

\[ T = \{ T_1, T_2, ..., T_n \} \]  

(3.6)

Meaning \( m(T) \in M \) and each job consists of \( k \) tasks, \( J \in \{ T_1, ..., T_k \} \). If we have precedence constraints between the tasks, e.g. task \( T_i \) and \( T_j (T_i \rightarrow T_j) \). This means task \( T_i \) has to be completed before \( T_j \) can start.

Each job is to be processed in order by order, meaning no intersection is allowed. The jobs are to be processed during a certain time period on a given machine. Moreover, one job should from start to finish not be interrupted by other jobs. The goal of the scheduling is to minimize the completion time, also referred to as the make span.

Widmer and Hertz[64] were one of the earliest to apply tabu search in scheduling.

### 3.2.1 The no-wait job shop

The no-wait job shop (NWJS) is a verision of the classical of the job shop scheduling problem. In the NWJS an additional constraint is added, namely two subsequent tasks in a job must be processed without any interruption. It is well known that the NWJS is also NP-hard[29].

Mascis and Pacciarelli [44] described the NWJS problem by means of generalizing the disjunctive graph, which they denote as the alternative graph. Additionally Mascis and Pacciarelli, define the no wait constraint as having a pair of consecutive tasks in a job. Say task \( T_j \) and \( T_i \), we further assume \( T_i \) must start being processed \( k \) time units after the completion of \( T_j \) (i.e. \( K \geq 0 \)). This is denoted as perishability constraint, since it represents the fact of job deterioration when being stored for longer than \( k \) time units. In figure 3.3, we can represent this constraint with a pair of fixed edges \((j,i)\) and \((i,j)\) which have a processing length of \( p_i \) and \(-p_i - k\), respectively. If \( k = 0 \) we have a tight no-wait constraint.

![Figure 3.3: No wait constraint](image)

In the next section we will show how a job shop problem can be modeled using a disjunctive graph.

### 3.2.2 Disjunctive graph

Egon Balas in 1969[8], formulated a elegant representation of the JSSP with the usage of a disjunctive graph. A disjunctive graph can be used to model a system
of task, which are to be scheduled with respect to time constraints which are to be considered. Two distinct characteristics of the disjunctive graph, can be described as follows:

- If task $T_j$ must be processed before $T_i$, this is represented as a directed edge from $T_j$ to $T_i$.
- If task $T_j$ and $T_i$ can be processed in either order, but not simultaneously. This is represented by a undirected edge from $T_j$ to $T_i$.

From Blazewicz, Pesch, and Sterna\cite{6} we can derive an example to illustrate how to model task scheduling using a disjunctive graph.

**Example** Consider a job shop problem, which consists of three machines $M = \{M_1, M_2, M_3\}$, and three jobs $J = \{J_1, J_2, J_3\}$. Each of the set of jobs can be described with the following task chain: $J_1 = \{T_1 \rightarrow T_2 \rightarrow T_3\}$, $J_2 = \{T_4 \rightarrow T_5\}$ and $J_3 = \{T_6 \rightarrow T_7 \rightarrow T_8\}$. In table 3.1 we present for any task $T_i$, which machine is required and the total processing time.

<table>
<thead>
<tr>
<th>$T_i$</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M(T_i)$</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M3</td>
<td>M2</td>
<td>M2</td>
<td>M1</td>
<td>M3</td>
</tr>
<tr>
<td>$p_i$</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

The corresponding disjunctive graph for the given problem is illustrated in figure 3.4. The vertices represent the different tasks, which are to be scheduled. The edges represent the total processing time for each task. We can divide the graph into three parts, which are to be taken care of one machine respectively. The upper nodes, are to be taken care of machine $M_1$. The nodes in the middle are to be processed by $M_2$, and the nodes in the lower part are to be processed by $M_3$. 

![Figure 3.4: Disjunctive Graph example](image-url)
The disjunctive graph contains all information needed in order to find a partial or complete solution of the *JSSP*. Therefore, it is of significance for the graph to be represented properly in order for algorithms to efficiently solve the problem.

### 3.2.3 Methods for solving the job shop problem

Solving the *JSSP* entails making scheduling decisions by defining the order of tasks requiring the same machine, where the disjunctive graph $G(S) = (V, C \cup D)$ for the *JSSP* is defined as the following:

- $V$: Set of tasks
- $C$: Set of directed edges reflecting the job-order of tasks
- $D$: Set of undirected edges reflecting tasks requiring the same machine

Since for every pair of tasks that require the same machine there is an undirected, so-called disjunctive edge. Turning undirected (disjunctive) edges into directed ones represents the process of solving the *JSSP*. A selection $S$ is defined as a set of these formerly undirected, now directed edges. Any selection $S$ is called a complete selection defining a feasible schedule[13] if:

- every undirected edge has been is turned into a directed one
- the resulting graph $G(S) = (V, C \cup S)$ is acyclic

Additionally there are two special vertices: The source 0 and the sink *. Whereby the processing times of source 0 and the sink * are equal to zero. The sink vertex has no neighbours in the directed graph since it is the last vertex. For a given schedule, i.e. a complete selection $S$, the maximal completion time of all tasks $C_{\text{Max}}$ is equal to the length of the longest weighted path from the source 0 to the sink * in the acyclic graph $G(S)$.

Solving the *JSSP* is a challenging task, certainly when a limited time frame is given. A well known benchmarking problem featuring the *JSSP* was introduced by Fisher and Thompson[21] in 1963 consisting out of ten jobs and ten machines. Since its introduction, the benchmarking problem has received a lot of attention by researchers. Thereby stimulating the development of many different solving strategies[10]. Many of the strategies are based on enumerative algorithms. Other strategies include mixed integer linear programming, constraint programming and heuristic based approaches. There also exist hybrid strategies using a meta-heuristic framework such as SA and TS.

The solving strategies can be categorized into complete and incomplete methods. While complete methods can be used to prove optimality, incomplete methods are able to produce satisfying solutions within a reasonable amount of time which often is of interest for many practical real-time applications.

Widely used methods for solving the *JSSP* are briefly described in the following.
3.2.3.1 Branch-and-bound

Branch-and-Bound (BB) is an algorithm design paradigm for discrete and combinatorial optimization problems which can be used to prove optimality. It belongs to the class of enumerative algorithms and was first proposed by A. H. Land and A. G. Doig in 1960 [38] for discrete programming, and has become the most commonly used tool for solving NP-hard optimization problems such as the JSSP.

The name "branch and bound" was first introduced by Little [1] in his work on the traveling salesman problem. There exist several BB algorithms for solving job-shop scheduling problems for exact solutions [13]. For several years the most efficient algorithm was developed by McMahon and Florian [45]. In 1987 Carlier and Pinson [14] presented an algorithm for solving the 10 x 10 benchmark problem and proving optimality of the solution.

The BB algorithm performs a systematic enumeration of feasible solutions by organizing them and forming a search tree. The root of the search tree represents the full set of feasible solutions while branches in the search tree entail a subset of such. Any decision made regarding the sequence of a task is referred to as a node in the search tree. The exploration of nodes in the search tree therefore represents the exploration of subsets of the solution set, which in turn are representing a partial schedule. In this way, each task is sequenced after another thereby constructing these partial schedules until no tasks are left to be sequenced.

In contrast to a simple depth-first search, the BB algorithm reduces the number of search nodes to be generated by applying a branching and a bounds scheme. The branching scheme allows the algorithm to only expand on nodes that are likely to lead to goal solutions. The bounds scheme however, dynamically defines restrictions to an explored node. In that way the algorithm is able to step back from further evaluating branches of a node violating the introduced bound constraints and continue the search at a more promising node. In literature, the process of stepping back from the expansion of nodes and continuing in another part of the search tree, is often referred to as backtracking.

The application of both schemes enables to reject the exploration of fruitless nodes of the search tree and thereby decreasing the number of nodes visited drastically, compared to a simple depth-first search. The search ends if all nodes, i.e. all decisions regarding the sequence tasks, have been implicitly or explicitly been explored.

3.2.3.2 Shifting bottleneck heuristic

The shifting bottleneck heuristic from Adams, Balas and Zawak [2] work is considered to be a powerful procedure for solving the JSSP. The algorithm aims at minimizing the total make span of the JSSP i.e. the completion time for all jobs. When jobs are competing for the same resources in the JSSP some machines can be identified as to be bottlenecks for the processing of tasks [10].
The general approach of the shifting bottleneck heuristic is to solve for each machine a one-machine scheduling problem to optimality. It is assumed that a lot of scheduling decisions for the various one-machine schedules coincide with an optimal solution for the global schedule. The shifting bottleneck heuristic is an iterative heuristic and determines at each iteration the schedule for an additional machine. Already scheduled machines are changed by using re-optimization. Bottleneck machines are scheduled first whereby a measure of the bottleneck quality for a machine is used which corresponds with the maximal completion time.

3.2.3.3 Heuristic Methods

Unlike complete methods like $BB$, incomplete methods cannot guarantee reaching an optimal solution and therefore not prove optimality. However, many heuristic methods have been shown to find optimal or near-optimal solutions for particular instances of problems. Incomplete methods generally encounter the problem of easily getting stuck in local minima, which results in the algorithm finding sub-optimal solutions.

Incomplete methods can be divided into two stages. The first stage performs a local search where a solution is perturbed into a new feasible solution. The second stage is the process for escaping such local minima. This is done using a meta-heuristic guiding the local search when it gets stuck in local minima. Thereby the meta-heuristics allow the search continue in other areas of the search space where it is likely to find better solutions.
Chapter 4

The Swedish Railway Network

In the following chapter we present different definitions regarding the railway network. We then discuss how the railway scheduling problem can be mapped to the job shop scheduling problem, and introduce the conflict graph.

4.1 Definition of Terms and Metrics in the Railway Network

Along the thesis a number of technical terms and metrics will be discussed. These are described in the following:

Meet-point - Locations where two trains may pass simultaneously, e.g. train stations.

Single railway track - A structure consisting of rails, which is shared by traffic in both directions.

Track segment - The segment of the railway which is delimited by two meet points. The number of tracks determines how many trains are allowed to travel the segment simultaneously (i.e. the capacity).

Signal blocks - The section of the railway which is separated by signals.

Train conflict - We can divide train conflicts into three distinct cases: meet, capacity, and pass conflicts. See section 4.3.

Minimum headway - The minimum time required for keeping a safe distance between two trains on a single track.

Delay info - The retardation time of a train, meaning the amount of time a train deviates from its planned arrival and/or departure time.

Minimum dwell time - The minimum time a train has to wait in a train station, before being allowed to continue traveling. Trains often scheduled to wait longer than the minimum dwell time, meaning delayed trains can accommodate for lost time if needed.
**Minimum section run-time** - The minimum time a train is scheduled to occupy a section. Trains are once again often scheduled to occupy sections longer than the minimum section run-time, thus this permits a delayed train to make up for lost time.

The railway network can be divided into two distinct categories: train stations(locations), and network links. In order for a train to travel from one location to another, the train has to travel through the network links connecting the two locations. In figure 4.1¹ we depict the railway network of the traffic area Boden. This is moreover the surveillance area of focus in this thesis.

![Diagram of the Swedish Railway Network](http://www.trafikverket.se/contentassets/8017cbf14eeb43abb055c14afe53e57c/tlo_boden.pdf)

Figure 4.1: Overview of surveillance area Björnfjell-Riksgränsen-Kiruna-Boden-Luleå

¹Source provided by Trafikverket: [http://www.trafikverket.se/contentassets/8017cbf14eeb43abb055c14afe53e57c/tlo_boden.pdf](http://www.trafikverket.se/contentassets/8017cbf14eeb43abb055c14afe53e57c/tlo_boden.pdf)
4.2 The Railway Timetable

If the train dispatchers only task was to monitor scheduled trains, the dispatchers job would be straightforward. In these cases all the planned arrival and departure times, would not cause any conflicts in the railway network. What makes a dispatchers job difficult is handling scenarios which are not planned for. These might include demands on track usage which are not currently planned in the railway timetable, or the need to re-plan to responded to unanticipated events. Examples of unplanned scenarios are listed in the following:

- The need to plan for unscheduled trains.
- The need to requests for time for maintenance work on tracks.
- The dispatchers has to re-plan for trains which are delayed. This is a very common scenario in the railway traffic.
- The dispatcher has to re-route trains during track outages.

The dispatchers have only two tools for handling these types of scenarios and monitoring the network, mainly the railway timetable and train diagrams. The railway timetable is a schedule for trains which are to travel the railway infrastructure. The timetable contains arrival and departure times for each train, not only for stations but also for intermediate stations.

The train diagram is a graphical representation of the timetable, the advantage with a train diagram is that is makes detecting conflicts relatively simple for a train dispatcher.

4.2.1 Train Delays

Train deviations from the intended timetable can both assume positive and negative values. Positive train deviations are referred to as train delays while negative deviations as lead. Deviations are measured in time units; often as minutes. These deviations can either result from the re-scheduling of trains or introduced disturbances that have occurred. Trains are considered to be delayed if their deviations exceed a specified positive value depending on the country and type of train. In Sweden, trains are generally being considered as delayed if the delay at their final destination is above 3 minutes. Since this is considered a negligible delay threshold.

The delay for a certain train at a point in its schedule, is determined by the difference between the intended arrival time and the new arrival time, i.e. by subtracting the new arrival time from the intended arrival time. The train delay can be referred to as the tardiness of a train, and is denoted as $D$. A train dispatcher will generally aspire to reduce the tardiness of trains in the schedule when considering different re-scheduling possibilities. When different
re-scheduling possibilities are evaluated they are compared against one another with regard to their cost which is affected by the delay of trains in a schedule.

The delay of a train is defined as following. If $c_i$ denotes the planned train trip completion time, and $C_i$ the actual train trip completion time, then the delay $D_i$ of train $T_i$ is defined as $D_i = C_i - c_i$.

4.2.1.1 Cost

The cost of schedules serves as an optimization criteria for search algorithms aspiring to find better re-scheduling solutions. The cost of a schedule is calculated with use of a cost function, i.e. objective function. There exist various cost functions that consider a range of attributes where the number of trains delayed, the severity of the delay(amount of time), and which type of trains are delayed, are frequently included attributes. A cost function should in general cover the needs and preferences of the railway network that is to be re-scheduled, since the process of re-scheduling aims at minimizing that cost.

The following delay costs are frequently used in literature[57]:

**MD:** The maximum delay(MD) can be calculated according to the constraint: $MD = \max(D_1, D_2, ..., D_n)$

The criterion minimizes the maximum delay only. As a result many trains might have a deviation from their intended schedule. This criterion is suitable for cases where passenger trains are of higher priority.

**WMD:** The weighted maximum delay(WMD) can be calculated according to the constraint: $WMD_{max} = \max(w_1D_1, w_2D_2, ..., w_nD_n)$

This criterion can be of interest for railway networks that encounter mixed traffic conditions. The weights $w_i$ are often corresponding to which type a train belongs to. Meaning that freight trains may share one weight, while passenger trains receive another.

**TD:** The total delay(TD) can be calculated according to the constraint: $TD = \sum_{i=1}^{n} D_i$

This criterion minimizes the delay for all trains.

**TWD:** The total weighted delay(TWD) can be calculated according to the constraint: $TWD = \sum_{i=1}^{n} w_iD_i$ This criterion may be used to minimize the total delay for all trains where in railway networks with mixed traffic conditions. Trains are receiving a weight, i.e. the prioritization of trains or type of trains.

**NTD:** The number of trains delayed(NTD) can be calculated according to the constraint: $NTD = \{ T_i \in T | D_i > 0 \}$ This criterion aims at minimizing the number of trains delayed in the schedule.
4.3 Solving train conflicts

The conflicts which are detected in the railway timetable, can be categorized into three different types[3]: meet, pass, and capacity conflicts. In the following we describe each type of conflict and the variety of solutions which can be imposed to resolve the conflicts. It is important to note that solving a conflict in one part of the schedule, might impose a new conflict further down in the timetable. This is not taken into consideration in the given examples.

4.3.1 Meet conflict

The first type of conflict involves two trains which collide. Thus, we have two possible solutions for a meet conflict. In figure 4.2(top figure) we illustrate train $i$ and $j$, which collide in a meet conflict. Followed by the top figure, we illustrate two possible solutions for the conflict. The two possible solutions are either to let the inbound or outbound train wait. In the illustrations the y-axis represents different stations and track segments, and the x-axis represents the elapsed time. From the figure we can tell both of the train are meant to wait in station 2 for a period of time. As train $i$ and $j$ are driving in opposite directions, they collide between station 1 and 2. The two possible solutions for the problem result in two different delays for respective train. The dashed lines represent the earlier path of the train, which lead to a conflict. The filled lines in the bottom figures illustrate the new valid solution to the problem.

![Figure 4.2: Depiction of a meet conflict and solutions](image)

4.3.2 Pass conflict

The pass conflict once again involves train $j$ and $i$. This type of conflict is more challenging to solve, since the running times of trains have to be considered. We assume trains leave their stations as soon as possible, in order to minimize their delay. Therefore when a faster train tries to bypass a slower train, the fast train
has to decelerate in order to avoid red lights. In figure 4.3, we in the top plot illustrate a pass conflict, where the fast train $j$ tries to pass the slow train $i$. Below the top plot, two possible solutions are proposed. The left bottom plot, illustrate an example of how train $j$ had to be slowed down, in order to avoid stopping in the middle of a track segment. The right bottom plot, illustrate an example of where we delay the slow train $i$ in order to let the fast train $j$ bypass it.

![Figure 4.3: Depiction of a pass conflict and solutions](image)

### 4.3.3 Capacity conflict

Finally, we have the capacity conflict which is the most complex to plan for. This type of conflict occurs when we have a train station, which can hold two trains(capacity of 2), and a third trains tries to arrive at the very same station. In figure 4.4, we illustrate a capacity conflict, and three possible solutions. We can derive that if a train station has a capacity of $N$, we have $N + 1$ possible solutions when a capacity conflict occurs. The policy for solving this type of conflict is to let one of the three trains, wait in the previous station. Until one of the trains in the full station departs.

![Figure 4.4: Depiction of a capacity conflict and solutions](image)
4.4 Mapping to the JSSP

In the following we present how to map the railway re-scheduling problem into a special case of job shop scheduling problem (JSSP). This can be achieved by treating each train trip as a job, which is to be scheduled on railway tracks. The tracks in this case are treated as resources. Further each train trip may consist of many tasks, which requires the allocation of distinct points in the railway line in order to travel between points. The distinct points can either be train stations or a track segments separated by two signals. Given that a resource has a capacity of one, a conflict occurs when two tasks try to occupy the same resource at the same time.

The extension to the classical JSSP in the railway re-scheduling problem, is due to the fact of more than more task being able to occupy a resource as long as the capacity allows for it. Formally stating in the railway re-scheduling problem, a train trip $J_i$ can be described as $J_i = \{T_{i1}, T_{i2}, \ldots, T_{ik}\}$. Where the $T_{ik}$ is the $k^{th}$ task of $J_i$. Each task $T_{ij}$ has a scheduled departure time $d_{ij}$ and processing time $p_{ij}$. In addition, each task requires processing on a unique set of tracks (i.e. machines), meaning $m_i \in M$.

The tasks of a job are to be scheduled so that no delay exists between the end time of a task and the start time of the next task. Given the fact that a subsequent task must start immediately after its antecedent, we have a no-wait constraint (see section 3.2.1 pp. 14).

**Example** As the definitions of what resources and task are in the context of railway scheduling have been defined, we now visually illustrate how the railway scheduling can be mapped to the JSSP. In table 4.1 a set of tasks are assigned to different machines. For each task we have a start time (departure time), and a processing time.

<table>
<thead>
<tr>
<th>Trip≡Job</th>
<th>Machine sequence</th>
<th>Departure</th>
<th>Processing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$m_8, m_7, m_6, m_5, m_4, m_3, m_2, m_1$</td>
<td>170</td>
<td>47, 96, 11, 27, 15, 49, 20, 21</td>
</tr>
<tr>
<td>2</td>
<td>$m_3, m_4, m_5, m_6, m_7, m_8$</td>
<td>381</td>
<td>49, 15, 27, 11, 96, 47</td>
</tr>
</tbody>
</table>

Figure 4.5 shows a train diagram of the present values in table 4.1. The illustration shows the different resources on the y-axis, and the elapsed time in seconds in the x-axis. Between station S2 and S3, the inbound and outbound train collide. Resulting in a conflict, in section 4.3 on page 23 we showed how this type of conflict can be solved.
4.5 The Railway Conflict Graph

Zwaneveld et al.[66] proposed the conflict graph. In which each node corresponds to a potential train route in the track topology. As the focus of the study is to re-schedule train which are delayed, and not re-route trains we need to re-formalize the definition of the conflict graph.

4.5.1 The solution space

With the previous established knowledge, we now introduce the conflict graph. Which represents the solution space in this thesis.

Definition 4.1. (Conflict Graph)

Given a set of track segments $R$ and a set of conflicts $C$ an disjunctive graph $G = (V, E)$ is called a conflict graph if:

- We have a one-to-one mapping between $R$ and $V$.
- $V$ and $E$ are finite sets.
- Directed edges represents task dependencies.
- Un-directed edges represents task which can be processed in either order.
From table 4.1 on page 25, we demonstrated an example of an infeasible timetable. In the example we had train \( i \) and \( j \), which had different train trips. Train \( i \) had the following tasks to carry out \( J_i = \{T_1, T_2, \ldots, T_8\} \), and train \( j \) had \( J_j = \{T_9, T_{10}, \ldots, T_{14}\} \). In figure 4.6, we show how the very same example can be depicted as an conflict graph. The directed edges represent the task dependencies and include the respective processing time. A undirected edge (colored in red) represents a conflict, where two (or more) tasks may be processed in either order.

In order to solve the conflict in figure 4.6, we have to either delay the processing time of task \( T_7 \) or \( T_9 \) as they are competing for the same limited resource. The two different solutions can be formalized as follows:

1. Task \( T_7 \) is delayed until task \( T_9 \) is done, meaning we delay the predecessor of \( T_7 \). The new processing time of its predecessor is \( T_6p = 15 + 49 \).

2. Task \( T_9 \) is delayed until task \( T_7 \) is done, meaning we delay the departure time of \( T_9 \). The new processing time of its departure is \( T_0p = 381 + 49 \).

4.6 Train Re-scheduling Strategies

Iqbal, Grahn, and Krasemann[32] proposed four different train re-scheduling strategies. These strategies are of importance, and strategy \( s_0 \) was chosen for this thesis. For motivations of the choice refer to Chapter 8 Implementation. In order to explain each of the strategies we first need to define a set of attributes and concepts.

e is an event of specific train, which is to represent a train movement at specified track of a section during a discrete time interval

\( t_{\text{start}}^e \) the planned start time (departure time) of train event \( e \)

\( t_{\text{stop}}^e \) the planned end time (arrival time) of train event \( e \)

\( t_{\text{runtime}}^e \) the minimum time train event \( e \) occupies a section

\( t_{\text{buffertime}}^e \) the additional time which is added to \( t_{\text{runtime}}^e \), to make timetables more robust to disturbances
\( t_{e}^{\text{release}} \) the time when train event \( e \) releases a track

\( t_{e}^{\text{min}_{-}\text{start}} \) the earliest time for when train event \( e \) can start

\( t_{e}^{\text{deviation}} \) the deviation time of train event \( e \), calculated as \( t_{e}^{\text{min}_{-}\text{start}} - t_{e}^{\text{start}} \)

\( s_{e} \) the section which is occupied by train event \( e \), where \( s_{e} \in R \)

\( NC \) an ordered list of of \( n \) candidates \( C \), where \( e \in \{C_{0}, C_{1}, ..., C_{n}\} \)

\( t_{e'} \) the subsequent event of train A to be executed

\( t_{e''} \) the subsequent event of train B to be executed

In each of the strategies, we have a pre-processing phase in order to detect trains which have collided. At the time of the disturbance a candidate list \( NC \) is calculated, which is sorted according to one of the four re-scheduling strategies. The time of the disturbance is denoted as \( T_{0} \). In the second phase, feasible candidates are executed one by one. The candidate list is updated with the next events of trains and re-sorted.

With each of the set of attributes and concepts formally defined, we now introduce the four strategies:

**Strategy \( s_{0} \) -** In the first strategy events which are in the candidate list are sorted according to earliest start time. Thus, the criteria for sorting can be formulated as: \( t_{e'}^{\text{min}_{-}\text{start}} < t_{e''}^{\text{min}_{-}\text{start}} \). Where \( e' \) and \( e'' \) represent two different trains in the \( NC \) list. If two train events have the same start time, \( t_{e'}^{\text{min}_{-}\text{start}} = t_{e''}^{\text{min}_{-}\text{start}} \), then they are stored according to: \( t_{e'}^{\text{min}_{-}\text{start}} + t_{e'}^{\text{runtime}} < t_{e''}^{\text{min}_{-}\text{start}} + t_{e''}^{\text{runtime}} \).

**Strategy \( s_{1} \) -** As the first strategy \( s_{0} \) did not take into account the release time of a track, this second strategy was developed. The motivation behind the strategy is that one train with a long section run-time might delay other trains significantly. The release time is calculated as follows: \( t_{e}^{\text{min}_{-}\text{start}} + t_{e}^{\text{stop}} - t_{e}^{\text{start}} \). The strategy then sorts the \( NC \) list according to the condition: \( t_{e'}^{\text{release}} < t_{e''}^{\text{release}} \).

**Strategy \( s_{2} \) -** This strategy aims to make use of the timetables built in buffer times. The buffer time is calculated as: \( t_{e}^{\text{buffer time}} = t_{e}^{\text{stop}} - t_{e}^{\text{start}} - t_{e}^{\text{runtime}} \). The advantage of this strategy, is that is able to fully utilize the buffer times to minimize the delay of trains. A comparison between two events is done accordingly to the condition: \( t_{e'}^{\text{min}_{-}\text{start}} + t_{e'}^{\text{buffer time}} < t_{e''}^{\text{min}_{-}\text{start}} + t_{e''}^{\text{buffer time}} \).

For cases where we have no buffer time, we sort events according to strategy \( s_{0} \).
**Strategy** $s_3$ - Lastly, we have a strategy which is based on minimum section run-time, which is intended to minimize the delay of trains. The strategy compares two train events accordingly: $t_{e''}^{\text{min\_start}} + t_{e''}^{\text{runtime}} < t_{e'}^{\text{min\_start}} + t_{e'}^{\text{runtime}}$. For larger delays $s_3$ sorts according to strategy $s_0$. 
Chapter 5
Meta heuristics used in railway scheduling

In this chapter we present three prominent meta heuristics used in railway scheduling. The heuristics are considered to be local search methods. Thus, the methods move from a solution to another one, in their neighbourhood according to some well-defined rules.

5.1 Simulated Annealing

Simulated Annealing (SA) is a probabilistic meta-heuristic, which has been applied to solve a vast range of optimization problems. The heuristic stems from the process of annealing in metallurgy. Whereby a material is heated and in a controlled manner is cooled down to a certain temperature. As the heuristics converges to random improvements, SA is expected to find near optimal or true optimum as the temperature reaches a cool state. The following will give a historical background, and a general description of the SA algorithm.

5.1.1 Historical Background

The SA algorithm origins from a reworking of the Metropolis-Hastings (MH) algorithm, which was developed and described by Metropolis et al. [47] in 1953. The MH algorithm was formally invented to find feasible solutions, for problems which are not analytically solvable.

Kirkpatrick, Gelatt and Vecchi in the year of 1983 [35], later adopted the algorithm in order to solve the classical travelling salesman problem (TSP). Moreover, Vlado Černý in 1985 [67] independently described the SA algorithm. Černý applied the meta-heuristic to solve the TSP, with the usage of the probability granted by the Boltzmann-Gibbs distribution. Results showed the simple algorithm was able to find near optimal and even true optimum for the given problem.

5.1.2 Algorithm Description

The pseudo-code for the SA method can be found in algorithm 1, which is derived from [42]. The heuristic firstly generates a initial solution (S), which could be
chosen randomly or according to some procedure. SA then perturbs the initial solution slightly; this permuted solution is denoted $S'$. The SA procedure does this in order to compare whether $S'$ has a lower cost than $S$. If this is the case, the current solution is replaced by the solution with lower cost.

A key element of the algorithm is the temperature, $T$. When we have a high value for $T$, solutions of higher cost are more likely to be accepted. With each iteration of the algorithm, $T$ decreases. Leading the SA algorithm to become more greedy towards the end of its search. The probability function for accepting lower quality solutions is known as the Boltzmann distribution, which is formulated in equation 5.1.

$$e^{\frac{\text{Cost}(S) - \text{Cost}(S')}{T}}$$  

(5.1)

The algorithm further has a condition, where it at random uniformly generates a value between 0 and 1. This random value is checked against the calculated Boltzmann distribution, in order to decide if the new solution is accepted or not.

**Algorithm 1** Generic Simulated Annealing

1: procedure SA() ⬤ Procedure SA
2: Setting initial and final temperatures
3: $S \leftarrow$ getInitialSolution()
4: $T \leftarrow T_{initial}$
5: while $T > T_{final}$ do ⬤ Temperature cycles
6:     while StoppingCriteria do ⬤ Metropolis cycles
7:         $S' \leftarrow$ PickRandomState($N(S)$) ⬤ Pick random neighbour of $S$
8:         if $\text{Cost}(S') < \text{Cost}(S)$ then
9:             $S \leftarrow S'$
10:         else
11:             $r \leftarrow \text{Random}(0, 1)$
12:             if $r < e^{\frac{\text{Cost}(S) - \text{Cost}(S')}{T}}$ then
13:                 $S \leftarrow S'$
14:         end
15:     end
16:     Decrease $T$
17: end

5.1.2.1 Cooling Schedule

The cooling schedule directly affects the run time of the SA algorithm. While a slow cooling schedule allows the algorithm to converge to the global minimum, the computation time increases drastically. The cooling schedule is defined by the initial temperature and the cooling factor used. A high initial temperature allows the algorithm to accept inferior solutions. Accepting seemingly worse solutions,
can open up parts of the search space where good solutions or even the optimal solution might be found. Therefore the initial temperature has to be carefully chosen, in order to enable the algorithm to be able to explore different parts of the search space. The temperature reduction can either be performed at every metropolis cycle or at every temperature cycle. By updating the temperature outside the metropolis cycles the temperature can be seen as to be reduced in levels instead of being decreased more fluently.

5.1.2.2 Equilibrium

An equilibrium criteria may be included in the SA algorithm. Equilibrium is considered to be reached when the algorithm is unable to improve the solution for a specified number of iterations. The detection of a state of a reached equilibrium can be used to terminate the search before the originally intended cooling schedule is finished. It can be further used to decide whether a transition to a lower temperature level shall be made, i.e. being used as StoppingCriteria therefore influencing the number of metropolis cycles.

5.2 Tabu Search

Tabu Search (TS) is a local search optimization technique, widely used for combinatorial optimization problems. The meta-heuristic is able to overcome local optimality limitation, with the usage of its forbidden list (tabu list). The tabu list includes moves, which have been previously examined. The purpose of maintaining a selective memory of the search trajectory, is to prevent the heuristic from re-visiting examined solution states. The heuristic has no sophisticated elements, as Simulated Annealing. The following will give a historical background of TS, and a description of the algorithms elements.

5.2.1 Historical Background

The origins of TS, extends back to the late 1960’s and early 1970’s. But was first created by Fred Glover in 1986[23]. The heuristic was later formalized in [24, 26]. As of today, main references in the TS area refer to the latter case. A updated review of the TS algorithm was later imposed by Glover et al. in [27].

The heuristic has been applied to a range of problems. These include scheduling problems, e.g. time tabling problems and job shop flow sequencing. Moreover, graph problems such as graph coloring and the traveling sales man problem (TSP). The problems are considered to be NP-hard combinatorial optimization problems. Meaning, it is not possible to compute a solution within polynomial time.

The first parallel implementation of TS and SA can be found in Malek et al.[43], which was applied to TSP. Results showed TS consistently outperformed
SA, in regards to computation time. Moreover, both of the algorithms yield similar solutions to the given problem.

### 5.2.2 Algorithm Description

The TS algorithm can be viewed as an iterative search technique, which explores a set of problem solutions, denoted $X$. The method modifies a initial solution, denoted $S$, by applying moves in the neighbourhood $N(S)$. In order to obtain a new solution, denoted $S'$. The moves are applied until the stopping criteria is met, with the aim to reach a optimal or near optimal solution. The moves are evaluated according to a objective function $f(S)$, which is to be minimized.


**Algorithm 2** Generic Tabu Search

1: procedure TS()  
2: \hspace{1cm} iteration ← numberOfIterations  
3: \hspace{1cm} $S ← \text{getInitialSolution}()$  
4: \hspace{1cm} TabuList ← NULL  
5: \hspace{1cm} while iteration ≠ StoppingCriteria do  
6: \hspace{2cm} $S' ← \text{getBestNeighbour}(N(S), \text{TabuList}, S)$  
7: \hspace{2cm} if Cost($S'$) < Cost($S$) then  
8: \hspace{3cm} $S ← S'$  
9: \hspace{2cm} TabuList ← $S$  
10: \hspace{2cm} iteration ← iteration −−  
11: \hspace{1cm} end

In algorithm 3 we search for the best candidate in the neighborhood. This is done by iterating over the possible transitions. We collect candidate moves which we store in a candidate list. Thereafter, we from the candidate list select the best candidate according to a objective function $f(S)$ which is to be minimized. Lastly, we return our permuted solution $S'$ and later compare it to our current best solution.
We begin with the basic elements of the TS algorithm. Starting with the tabu list and memory structure, and lastly discuss aspiration criteria.

5.2.2.1 Tabu List and Memory Structures

The tabu list contains a record of moves in the order they have been taken, or in some memory structures complete solutions. The moves are added to the end of the list, and removed from start of the list. If a solution or move appears in the tabu list, it cannot be re-visited. Hence, the name tabu(taboo) list.

One of the main difficulties of TS, is keeping track of visited solutions. As explained by Glover[25], there exists three types of memory structures for TS:

1. **Short-Term Memory** As memory requirements of the TS algorithm rapidly becomes prohibitive. A structure to remedy this limitation, is to make use of a short-term memory. One simple implementation is to store recently visited solution in the list. However, storing and comparing complete solution might be too demanding on memory and computation time. To accommodate for high computation and memory demands, a popular alternative method is to use transition abstraction. Which stores the transitions rather than solution states. Since classifying a certain move in general only requires to keep track of a set attributes. As an example, we can keep track of hashtags which represent transitions between two states.

2. **Intermediate-Term Memory** In order to solve harder problems, and escape from visiting solution of lower quality or avoid spending too much time in a certain region. One might employ intermediate-term memory. Intensification rules are used in this memory structure with the objective to store high-quality solutions. This makes it possible to more accurately investigate areas of the search space where the best solutions were identified. One simple intensification scheme consists of returning the best current solution found.
3. **Long-Term Memory**

The last memory structure makes use of *diversification rules*. These rules help to guide the search into new search regions. This can be achieved through restarting the algorithm. Or by *strategic oscillation*, which modifies the objective function to balance how much time i spent in regions of the search space.

**5.2.2.2 Aspiration Criteria**

One important aspect of *TS* is the usage of aspiration criteria, which aims to override the tabu status of a move. One of the simplest and most popular aspiration criteria involves allowing forbidden moves, which improves the current best known solution. This might be desirable in situations where the transition $s_1 \rightarrow s_2$ is forbidden. The state $s_2$ now becomes unexplored, and might potentially yield to be the best solution found so far.

### 5.3 Genetic Algorithm

The *Genetic Algorithm*(*GA*) method is inspired by Darwin’s theory of evolution, and aims to find solutions to NP-hard problems.

#### 5.3.1 Historical Background

In the 1950’s and the 1960’s computer scientists independently investigated how evolution could be used as an optimization tool, for solving engineering problems. The idea behind this research was to develop an optimization technique, which purpose was to generate a population of candidate solutions based on natural genetic variation and natural selection.

Ingo Rechenberg in the years of 1965 and 1973\cite{53, 52} introduced "evolution strategies"(Evolutionsstrategie in the German). Rechenberg used this method in order to optimize real-valued parameters for airfoils. The idea was then further researched and developed by Schwefel\cite{54}.

Genetic algorithm was invented by John Holland, and Kenneth A De Jong. Holland, his students, and colleagues at the University of Michigan in the 1960s and 1970s investigated how evolution could be adopted as an optimization tool.

*GA* became popular through Hollands book *Adaptation in Natural and Artificial Systems* (1975)\cite{31}. What differentiated *GA* from evolution strategies, was that *GA* was applicable to solve a range of problems rather than a specific problem. De Jong is also considered as a originator of the *GA* algorithm, as De Jong in 1975 published his book *Adaptation of the behavior of a class of genetic adaptive systems.*
5.3.2 Algorithm Description

The GA technique is inspired by nature, hence it is useful to introduce biological terminology which are part of the method. From Melanie Mitchell[48], we present the basics of GA, and biological terminology in the context of the method.

Every living organism consists of cells, the cells in turn consist of the same or different set of chromosomes\(^1\). The chromosomes can be further divided into genes, you may consider a gene as encoding a trait(e.g. eye color). The variations of possible traits, such as blue, brown, and hazel eye color, is referred to as alleles. Moreover, the locus of a gene refers to a position in the chromosome.

The GA method, as described in algorithm 4, starts with generating an initial population, which contains possible solutions. The possible solutions are referred to as chromosomes. GA, afterwards attempts to evolve this population through iterations(generations). The chromosomes are evaluated according to a fitness functions, during each generation. For the method to generate new chromosomes(offspring), we merge two existing chromosomes from the current population, using the crossover operation. The method later modifies a chromosome using the mutation operator. To form a new generation we select chromosomes, this includes parents and offspring, according to their fitness value. Chromosomes which have a higher fitness value, have a higher probability of being selected. After several generation, the method is expected to converge to a optimum or near optimal solution, for the given problem. Choosing the stopping criteria is often problem dependent, one simple approach is to stop when we have \(G\) non-improving generations. Alternatively choosing a fixed number of generations.

Algorithm 4 Generic Genetic Algorithm(GA)

\[
\begin{align*}
1: & \quad \textbf{procedure} \ \text{GA}() \quad \triangleright \text{Procedure GA} \\
2: & \quad t \leftarrow 0 \\
3: & \quad \text{initPopulation}(P(t)) \\
4: & \quad \text{evalPopulation}(P(t)) \\
5: & \quad \textbf{while} \ \text{stoppingCriteria} \ \textbf{do} \\
6: & \quad \quad P_p(t) \leftarrow P(t).\text{selectParents}() \\
7: & \quad \quad P_c(t) \leftarrow \text{crossover}(P_p) \\
8: & \quad \quad \text{mutate}(P_c(t)) \\
9: & \quad \quad \text{evalPopulation}(P_c(t)) \\
10: & \quad \quad P(t + 1) \leftarrow \text{buildNextGeneration}(P_c(t), P(t)) \\
11: & \quad \quad T \leftarrow T + + \\
12: & \quad \textbf{end}
\end{align*}
\]

\(^1\)The string of DNA, also defined as an organisms blue print.
5.3.2.1 Encoding

The very first thing to consider when implementing the GA method is how the solutions should be encoded. The most common approach is to represent the chromosomes as a string of bits, this is referred to as binary encoding. This type of encoding creates many possible chromosomes, even when only having a small number of alleles. The downside with this encoding is that it might not be a natural approach for all types of problems.

A alternative encoding scheme is to use permutation encoding, where each chromosome is represented as an array of numbers which indicate an order of sequence. This type of encoding is useful for problems like the travelling salesman problem. Where each chromosome holds a sequence of cities in the order the salesman will visit them.

5.3.2.2 Selection

This operator selects the fittest chromosome, which is to be used as parent to create the offspring. Chromosomes with a higher fitness value, are more likely to be selected as parents. There however exists different types of methods for chromosome selection.

The Roulette wheel selection works as the name suggest. The candidate chromosomes each occupy their own slice in an imaginary roulette wheel in proportion to their fitness score. Chromosomes in the wheel are selected through generating a random number, which specifics a spot in the wheel. If we have a high difference in the fitness score, chromosomes with a lower fitness score are less likely to be selected. Figure 5.1 illustrates the relationship between chromosomes and their fitness score in a roulette wheel.

![Roulette wheel selection](image-url)
An alternative method, which is less likely to neglect chromosomes with lower fitness score is *rank selection*. This method will rank each of the chromosomes, according to their fitness score. The chromosome with lowest fitness score will receive a new fitness of 1, the second worst a fitness of 2, and the best chromosome receives a fitness of \( N \) (number of chromosomes in the population). In the figure 5.1, the chromosome with highest fitness value will have the probability \( \frac{4}{1 + 2 + 3 + 4} = 0.4 \) of being selected. The chromosome with the lowest fitness value would have the probability \( \frac{1}{1 + 2 + 3 + 4} = 0.1 \) of being selected.

### 5.3.2.3 Crossover

The crossover operator randomly selects a locus, and exchanges sequences before and after the locus of two selected chromosomes to generate two off-springs. To illustrate this we take the strings 10000100 and 11111111, the chromosomes are crossed over in the third locus. This will produce the offspring 10011111 and 11100100. How crossover is done further depends on the type of encoding used, the aforementioned example depicted a binary encoding crossover.

### 5.3.2.4 Mutation

In order for the algorithm to escape local optima, it introduces a form of randomness. The mutation operator randomly selects a position in the chromosome and flips the bit. For example, the second position is chosen in the string 00000100, the mutation will afterwards yield the new string 01000100.
In this chapter we present two parallel approaches, shared- and distributed memory. Many difficulties arise when introducing parallel computing to applications. A general description is in the following described, moreover difficulties and problems which occur in parallel systems are discussed. Lastly we introduce parallel adaptations of simulated annealing and tabu search, which are widely applied in past academic work.

6.1 Computer architectures

It is useful to classify the different parallel computer architectures into categories according to certain characteristics. A simple model is given by Flynn’s taxonomy[22]. The taxonomy distinguishes parallel computers according to the global control, the data, and control flows. The four categories which can be classified are presented in table 6.1.

<table>
<thead>
<tr>
<th></th>
<th>Single Instruction</th>
<th>Multiple Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple Data</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

The different categories which are listed in Flynn’s taxonomy can be explained as follows:

1. **Single-Instruction, Single-Data (SISD):** A sequential computer architecture which makes no use of parallelism. In this architecture we have one processing element, which accesses a single program and the data storage. In each step, the SISD architecture loads an instruction and the corresponding data, and executes the instruction. The calculated results are stored back in the data storage. An example of a SISD architecture are uni-processor machines. In figure 6.1a we depict an example of a SISD system.
2. **Multiple-Instruction, Single-Data (MISD):** Multiple processing elements exist in this architecture. Each processing element has access to its own private program memory. However, the processing elements share a common data storage. In each step, the processing elements load an instruction (possibly different instructions) from their private memory, and they load the very same data element from the data storage. The processing elements then execute their loaded instructions in parallel on the identical data element. In Figure 6.1c we depict an example of a MISD system.

3. **Single-Instruction, Multiple-Data (SIMD):** Multiple processing elements are present, each of which have exclusive access to a shared or distributed data memory. In SIMD the processing elements have access to a common program memory, where instructions are fetched from. In each step, the processing element fetch the same instruction, which is to be executed in parallel on different data elements. An example of a SIMD is the Graphics processing unit (GPU). In Figure 6.1b we depict an example of a SIMD system.

4. **Multiple-Instruction, Multiple-Data (MIMD):** Multiple processing elements exists, each of which have access to separate instructions, and separate distributed or shared program and data-memory. In each step, the MIMD architecture loads a separate instruction and separate data element, and executes the instructions in parallel on the data elements. The calculated results are stored back to the data storage. An example of a MIMD architecture are shared memory and distributed memory systems. In Figure 6.1d we depict an example of a MIMD system.

Figure 6.1: Classifications of computer architectures
6.2 Shared memory systems

Gene Amdahl[4] in 1967, argued that single computers had reached its limits. Amdahl further stated in order to achieve significant improvements, interconnection of multiple computers which corporate to solve problems are in need. Amdahl is well known for the discovery of the speedup formula, known as Amdahl’s law, presented in constraint 6.1. The law indicates the expected speedup of the relation between a parallelized and serial implemented algorithm.

\[
S(n) = \frac{1}{(r_s) + \frac{r_p}{n}}
\]

(6.1)

Where \( r_s \) represents the ratio of the serial portion of a program, and \( r_p \) the ratio parallel portion. Moreover, \( n \) represents how many processors are used.

From Rauber and Rünger[51], we can derive basic theory regarding computers with a physically shared memory. These types of computer architectures are referred to as shared memory machines (SMMs). SMMs can from figure 6.2, be depicted as a set of processors which are interconnected to the very same physically shared memory.

Each processor in turn contains four cores. The processors are able to exchange data, through read and write operations to the shared memory (main memory). When several processors access shared variables in main memory concurrently, a problem denoted as race condition occurs. To illustrate a race condition, we can refer to table 6.2. In this example we have two threads \( T_1 \) and \( T_2 \) running concurrently. Both of the threads access the value of variable \( a \), each thread now has its own copy of the variable. As the two threads increase the value of the

![Figure 6.2: Example of a multiprocessor system](image-url)
variable, only their local copy is increased. When the threads write the updated value back to memory, the value of $a$ is incorrectly one. This is due to the race condition, when in fact the value of $a$ should have been updated to two.

Table 6.2: Race condition

<table>
<thead>
<tr>
<th>Thread 1($T_1$)</th>
<th>Thread 2($T_2$)</th>
<th>Variable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>read value</td>
<td>$a = 0$</td>
<td>$a = 0$</td>
</tr>
<tr>
<td>increase value</td>
<td>$a = 1$</td>
<td>$a = 1$</td>
</tr>
<tr>
<td>write back</td>
<td>$a = 1$</td>
<td>$a = 1$</td>
</tr>
</tbody>
</table>

Therefore, in order to keep main memory consistent and up to date, synchronization between access to shared variables is needed.

Sutter and Larus[58] debate today’s programming languages are not adequate for transforming serial applications to parallel. They further discuss the difficulty is not designing the multi-core systems. But actually how programming should be done, to efficiently utilize the CPU. Edward Lee[39] states automatic parallelism only achieves modest parallelism. Henceforth, a proper design for parallel implementation is a must. Lee further discusses the difficulties when applying parallelism to applications, are introduced by the abstraction of simple instructions. These instructions include: semaphores, mutual-exclusion locks, and monitors. Arvind, Nikhil, and Pingali[5] further argue it is very difficult to achieve parallelism, efficiency and ease of coding simultaneously.

From these studies, we can draw the conclusion that applying parallelism is not trivial. Therefore a careful design must be selected when implementing parallel versions of $SA$, and $TS$.

6.3 Distributed memory systems

The other parallel technique involves computers with physically distributed memory, also addressed as distributed memory machines(DMMs). DMMs consist of a number of independent processing elements(nodes), which are interconnected through a network. In order to allow the nodes to transfer data across the network. Each node $n_i$ has its own processor $P_i$, and main memory $M_i$, meaning $n_i = P_i + M_i$. A example of a DMM is illustrated in figure 6.3.
6.4 Parallel Simulated annealing and Tabu search

In order to adopt parallel computing, we in the following present common approaches which have been widely used. The approaches discussed include both distributed and shared memory techniques regarding parallel implementations of simulated annealing and tabu search.

6.4.1 Parallel TS

There exist several approach as how to adopt parallelism in the TS algorithm. Four strategies which are mainly pointed out are as follows:

1. Adopting parallelism in the cost function evaluation. Henceforth, instead of having a single thread calculate the cost of the current solution, we use multiple threads to calculate the cost. Thereby, we minimize the computation time. Drawbacks with the parallelism in the cost function, is that it will not improve overall solution optimality of the search.

2. Adopting parallelism when examining and evaluating the neighbourhood. In this approach, we evaluate the neighbours of our current solution in parallel. This will result in faster computation times when identifying the best neighbour. The approach also lacks the ability to improve the solution...
optimality, as it focuses on decomposing the work of merely evaluating the
neighbours.

3. Decomposing the problem into smaller sub-problems, whereby course grain
parallelism can be adopted.

4. Adopting parallelism in the exploration of the solution domain. This is
done by maintaining dependent or independent search paths. This type
of parallelism has the advantage of minimizing the communication over-
head of threads. It is further well suited for Multiple Instructions Multiple
Data (MIMD) parallel computers.

6.4.1.1 Parallel cost function evaluation and neighbourhood

The first two strategies were proposed by Jaishankar Chakrapani and Jadranka
Skorin-Kapov[15]. In the sequential version of the TS algorithm, Chakrapani and
Skorin-Kapov argue evaluating the neighborhood at each iteration might result
in being computationally too demanding. Pairwise exchanging moves in order to
find the best neighbour involves the evaluation of $O(n^2)$ exchanges. Evaluating
the neighbourhood in parallel requires the evaluation of $O(\log n)$ exchanges. This
is due to independently evaluating a subset of the neighbourhood. The authors
employ both intensification and diversification in their parallel implementation
of TS. Their intensification phase is started after the 25th iteration, allowing the
algorithm to move back to the best solution found so far. The diversification
starts after the 50th iteration, and is performed for a fixed 10 iterations. This
phase is meant to guide the search towards unexplored areas of the search space.

6.4.1.2 Parallel decomposition of the problem

The third parallel strategy of the TS algorithm was used by Taillard[59] to solve
the vehicle routing problem. Taillard proposes two different partition methods,
one based on the euclidean distance and the other based on the shortest path
distance. The approach involves distributing the number of cities to visit around
central depots, the cities are distributed according to the aforementioned parti-
tioning methods.

6.4.1.3 Parallel solution domain exploration

The fourth parallel approach of the TS algorithm was used by Smail Niar and
Arnaud Freville[49]. The parallel implementation was used in order to solve the
0-1 Multidimensional Knapsack Problem. In this parallel adaptation, we create
dependent or independent parallel processes, which are denoted as parallel search
threads. Each of the search threads execute the TS algorithm using a different
set of parameters. The chosen parameters will determine the behaviour of the
algorithm in each processor. The parameters which are of relevance are: the size tabu list, the number of neighbours evaluated, the move which is chosen at each iteration, and using a different set of initial solutions.

6.4.2 Parallel SA

Several approaches exist for adopting parallelism for the SA heuristic. Greening[28] introduced a taxonomy which divides parallel annealing techniques into three major classes:

1. Serial-like algorithms.
2. Altered generated algorithms.
3. Asynchronous algorithms.

Generally, parallel approaches for SA can either be used to speedup the sequential annealing process or to achieve a higher quality of solutions to a problem. Speeding up the sequential process approaches have a limited scope for large scale parallelism[16] and are more suited for single machines rather than clusters. More distributed approaches aiming at finding higher quality of solutions instead, are taking advantage of the observation that a good initial solution results in faster convergence. A higher diversity in the search space enhances the probability of finding better solutions. These approaches are highly applicable to distributed memory architectures as well. Cases of super-linear speedups have been reported by Ram and Sreenivas[50] for finding the same quality of solution in comparison to sequential SA. In order to achieve super-linear speedups with parallel SA(PSA), an optimal solution has to be identifiable as such. Distributed approaches are therefore more popular than speeding up the sequential annealing process.

The most commonly used parallel approaches for identifying higher quality solutions are described in the following.

6.4.2.1 Simultaneous independent searches

A very unproblematic and generally applicable type of parallelism for SA is the IIP method[55]. The algorithm is executed identically (except for different initial solutions) and independently on the several nodes available simultaneously. The communication between the nodes is kept to a minimum. For problems where an optimal solution can be recognized as such, the nodes communicate only when one locates the optimal solution and the remaining are told to stop. According to Shonkwiler and Van Vleck[56] it can be shown theoretically that the IIP method generally can achieve super-linear speedup. For problems where an optimal solution is not always identifiable, two stopping methods can be applied. We can stop the processes either by stopping when the first process has finished its cooling schedule, or by letting all processes finish their searches undisturbed.
6.4.2.2 Simultaneous corresponding searches

A more advanced variant of IIP starts several instances of SA with different initial solutions which then, after a fixed number of iterations, exchange their best solution found so far[43]. All the nodes accept the best solution shared, and continue applying the SA technique on the newly accepted best solution. The process is continued for a certain number of times, after which each node continues working independently until completion.

6.4.2.2.1 Temperature updating strategies

There exist different strategies on how the temperature should be updated upon nodes adapting results from another node. It has been shown that SA is very sensitive and an ongoing annealing process generally should not be disturbed. The adapted solution however, might not correspond to the current annealing temperature. Following strategies for temperature updating have been presented by Malek[43];

1. Continue with the same temperature after swapping.
2. Update temperatures in proportion to change in cost.
3. Reset temperature to starting value.

Results obtained by Malek have shown that a modification of the third strategy to be very effective. The modification entails that each process is annealed at a faster rate, by using a lower initial temperature and higher cooling rate. Which allows each process to complete its search before results are shared. The temperature is reset to its starting value, causing the annealing process to begin all over again from the adapted solution now serving as new initial solution. Repetitive solutions are prevented by memorizing the previously shared solution. The approach of restarting stochastic search algorithms such as SA has been researched by Mendivil and Shonkwiler[46] and has been shown to potentially be very effective.
Chapter 7

Related Work

This chapter gives the reader insight to past academic work conducted in the area of train scheduling and job shop scheduling.

7.1 Job Shop Scheduling

Wojciech Bozejko and Jarosla Pempera[11], implemented a parallel tabu search algorithm. In regards to the Flow Shop Problem, which is a special kind of case of the Job Shop scheduling problem. Wojciech Bozejko and Jarosla Pempera conducted a experiment where they evaluated tabu search for three different cases: sequential, multi-run and parallel. They concluded the multi-run tabu search obtained the best results on average. Whereas the parallel implementation obtained the best results for the first 30 test cases (total of 50). Wojciech Bozejko and Jarosla Pempera suggested to further improve the parallel tabu search as future work. For instance having a dynamic tabu list, instead of static. Furthermore making several moves simultaneously per iteration for the parallel algorithm.

Bozejko, Pempera, and Smutnicki[12] applied three algorithms based on simulated annealing, on the job shop scheduling problem. The first method was the classical simulated annealing. The second approach denoted as PSA-R, is a parallel implementation which randomly selects candidate neighbours, which are to be evaluated in parallel. The last approach denoted as PSA, simply generates several solutions using a single core, and evaluates the objective value of the solutions in parallel. Results showed PSA-R outperformed both the classical, and the PSA algorithm. Using 4 cores, the PSA-R algorithm was able to achieve superlinear speedup of 4.7 to 5.6. The additional speedup was acquired due to the utilization of the MMX instruction, which rules out branches in the executing code.

7.2 Meta heuristics applied in Railway scheduling

Wei Fang et al.[20] studied which types of algorithms are frequently used when re-scheduling railway networks. The results showed, most of the reviewed studies used Heuristic approaches for re-scheduling, such as: Simulated annealing, greedy algorithm, tabu search, and genetic algorithm.
T. K. Ho and T. H. Yeung[30] applied three different heuristics methods: Genetic algorithm, simulated annealing, and tabu search. In order to tackle conflicts which occur in railroad junctions. The center of focus was evaluating the algorithms based on: computation time and the solutions optimality. T. K. Ho and T. H. Yeung concluded that all three algorithms provide a similar balance in computation time and optimality. As future work they proposed further experimentation of parameters of Genetic algorithms, such as mutation and cross-over operations.

Mohammad T. Isaai and Madan G. Singh [34] have evaluated simulated annealing(SA) and tabu search(TS) in a hybrid approach with another heuristic to create predictive schedules. The problem described are trains that interact in a railway network that consists mostly out of single-track railway with some double-track blocks. They have evaluated three algorithms. One constraint-based heuristic and two hybrid methods, which are combinations of tabu search and simulated annealing with the heuristic. Known timetables of passenger trains on Iran’s railway network are compared to the generated schedules of the algorithms. They conclude that the algorithms perform quite well and future work could be done regarding the combination of two meta-heuristics.

Francesco Corman et al.[17] implemented a tabu search and truncated branch and bound algorithm, in order to re-direct trains in the Dutch railway network during rail operations. The algorithms are compared in various of scenarios, where they include multiple late trains and complex disturbances. Results showed tabu search found near optimal results for a small number of instances. For a large number of instances the algorithm finds 15\% better solutions, than results achieved from running tabu search for 180 s. As future work Francesco Corman et al. discussed improving tabu search. In regards to a large number of instances, to find near optimal solution within short computation times. Furthermore, they aim taking into account other factors, such as minimizing energy consumption.

Francesco Corman et al.[18] further developed a real-time re-scheduling Decision Support System(DSS). With the aim of minimizing accumulated delay, using Automatic Route Setting(ARI-like), tabu search and Branch and Bound(BB). Computational results showed TS was able to minimize accumulated delay more than 11 \%, in comparison to BB. Moreover, TS minimized accumulated delay more than 36 \%, when compared to the ARI-like system. On the downside, TS often reached the time limit of 130 s. For future work, Francesco Corman et al. propose integrating the system into larger frameworks with multiple dispatch areas.

Lake et al.[37] compared TS, SA, and a local search method for the railway track maintenance problem. The methods are evaluated against a problem with twenty track segments, and a seven-day scheduling period. Results showed SA is the best performing method, by obtaining the best final results, which was measured from the difference between the initial solution cost.

The proposed problem was originally addressed by Johanna Törnquist Krasemann[36],
where a greedy algorithm was implemented to tackle the railway issue specifically for Sweden. Results showed the greedy algorithms is able to find feasible solutions within a short time frame. As future work Krasemann suggested to apply parallel computing techniques, in order to find more optimal solutions in a shorter time frame.

Johanna Törnquist has been working on the re-scheduling problem prior to that[61] by introducing two meta-heuristics in attempt to more efficiently optimize the search space. Those two meta-heuristics are tabu search and simulated annealing. Her results indicate that tabu search performs better than simulated annealing for the model of a single-track system not taking into account that trains can take over each other for the Swedish Banverket.

The problem was later revisited by Syed M. Z. Iqbal et al.[33], where they implemented a parallel depth-first search (DFS) branch-and-bound (B&B) algorithm. Syed M. Z. Iqbal et al. concluded that the parallel implementation improves the solutions found for five different disturbance scenarios. Furthermore, as stated in [33] the problem is very specific due to the Swedish railroad infrastructure. In contrast to other countries, the Swedish railroad network allows for bi-directional traffic. Moreover, Swedish railroads allow for track swapping in double-tracked railways. Lastly, a commonly used re-scheduling strategy during disturbances is to allow both tracks for traffic in one direction. All the aforementioned factors, contribute to making the problem more difficult to solve.
Chapter 8

Implementation

In this chapter we describe our implementation of the re-scheduling of conflicted timetables. The chapter is structured as follows. In section 8.2, we describe the data loading module and the data structures used. Section 8.3 describes how conflicts are detected and resolved. In Section 8.4 we introduce the implementations of TS and SA. Finally section 8.5 gives the reader insight on how the system is validated.

8.1 System Overview

In our implementation, there are two parts which can be distinguished (see fig. 8.1):

- The Conflict Detection & Resolution loop (CDRL)
- The Heuristic Based Resolution (HBR)

The CDRL module is responsible for identifying conflicts within the timetables, which are read by the data loading module. The CDRL has two primary objectives, one is resolving the detected conflicts according to a conflict resolution table. The second objective is to create a conflict resolution table if an initial solution is being generated. A conflict resolution table contains for each conflict, the train chosen to delay and the alternative trains we could have delayed. The CDRL and HBR module both communicate through the conflict resolution table.

The HBR module receives the conflict resolution table and perturbs the resolution by altering one or multiple entries in the conflict resolution table in accordance to the possible choices. The perturbed resolution, represented as conflict resolution table, is received by the CDRL as input. The CDRL attempts to create a new feasible solution based on the perturbed conflict resolution table received from the HBR. Entries which were not utilized by the CDRL are removed from the table upon returning the newly created resolution to the HBR. The process ends when the meta-heuristics TA or SA, have reached their stopping criteria, and the best resolution found so far is returned.
Chapter 8. Implementation

8.2 Data loading

The input data is divided into two different parts: the train timetable, and the railway infrastructure. The implemented system loads the available data in plain text format, which is locally stored on the hard drive. As shown in figure 8.2, the data is afterwards parsed in order to form a two data structures which are to represent the timetable and railway infrastructure.
8.2.1 Train timetable

The timetable information is parsed and stored in containers specifically designed to hold information of train events. Each container is responsible for storing information for one specific train. When the parsing and storing process is completed, each container will hold all the train events of one designated train it was assigned to store.

Each train container will store the following attributes for each of the events (terminology discussed in sec. 4.1):

- **TSP or TMV:** A event in the timetable can either be a train stop (TSP) or train movement (TMV).
- **Internal identifier:** The internal identifier specifies which train the event belongs to.
- **Event identifier:** Each event is in addition associated with an event ID.
- **Current Destination:** The current destination of the train.
- **Travel Destination:** The travel destination of the train.
- **Departure time:** Is specified both in seconds from midnight and according to the 24-hour clock format.
- **Arrival time:** Is specified both in seconds from midnight and according to the 24-hour clock format.
- **Dwell time:** The time a train has to wait before continuing, is also read by the data loading module.
- **Planned stop:** Specifies if a train has a planned stop in the passing station.
- **Section run-time:** The time a train has to occupy a section.

8.2.2 Railway Infrastructure

The infrastructure data is parsed and stored in a container designed to hold multiple entries of track segment information. Each track segment entry can be identified by a start and end point and has an attribute defining its capacity i.e. the maximum number of trains allowed on the track segment simultaneously.

The track segment container will in the end store the following attributes for each entry (terminology discussed in sec. 4.1):

- **NWK or LOC:** An entry can either be a network link (NWK) or location (LOC). Network links are used to connect different locations together in the infrastructure.
Start location: Determines the starting point of a track segment.
End location: Determines the end point of a track segment.
Number of tracks: Specifies the number of tracks available for the track segment.
Headway: The internal identifier specifies which train the event belongs to.

8.3 Conflict Detection and Resolution loop (CDRL)

The CDRL module is responsible for the detection, and resolution of conflicts. In figure 8.3, we illustrate how the three main containers interplay. The train event list is responsible for storing the data which is read from the data loading module. In the initial phase, the very first events of train A-D are loaded into the candidate list. In the figure events which have been selected are darkened. The active list invariably selects the very first element in the candidate list. The process is repeated until any of the following conditions have been met:

- A train delay has been detected
- A conflict has been detected
- The very last element in the candidate list has been executed

![Figure 8.3: Overview of the re-scheduling process. Depicting the active, candidate, and train event lists](image)

The CDRL ends its processing when the last entry in the candidate list has been executed without inducing a new conflict. In the following we go more in depth of the main containers, and describe the greedy solver algorithm.
8.3.1 Conflict Detection

Train events are executed according to their order in the candidate list. This is accomplished by removing the first entry in the candidate list and placing it into the active list. The subsequent event for the selected train is then inserted to the candidate list. Further the active list is kept updated by removing events if their end time exceeds the newly added events start time. As example we can look at train Bs container (see fig. 8.3.1), if event $B_2$ were to be executed. Its successor $B_3$ is then loaded to the candidate list.

8.3.1.1 Inserting Subsequent Events

Insertions to the candidate list are fulfilled by performing a sorted insertion at the appropriate position in the list. Thereby the candidate list is kept sorted at all times, which alleviates the need to re-sort the candidate list altogether. Initial sorting and sorted insertion are carried out according to sorting strategy $s_0$ (discussed in sec. 4.6).

When inserting a subsequent event of an event currently being executed, the subsequent events departure time must match the executed events arrival time. For events not deviating from their originally intended times, no additional action is needed.

For deviating i.e. re-scheduled events however, the altered arrival time has to match the subsequent start time. This is achieved by introducing departure- and arrival-delay as attributes to train events. Since trains are not allowed to depart earlier than their earliest departure time for planned stop events, only positive values for the delay attributes are allowed.

Events with a $\text{departuredelay} > 0$ will receive their arrival-delay depending on their minimum run-time or minimum dwell time on the track segment which corresponds to the buffer time.

8.3.1.2 Checking for Conflicts

The active list, at all times, contains train events being active on the railway network. With the active list, we can accordingly determine if executing an event renders a conflict and what events are affiliated with the conflict at the current state of event execution. The information regarding which events are affiliated with the conflict will be used in the conflict resolution process.

To determine if events currently active on the railway network are conflicting, each member of the active list is examined for how many other entries in the active list are occupying the same track segment. If the number of entries sharing a track segment exceeds the allowed maximum capacity of the track segment, a conflict has been detected.
8.3.1.3 Description of Conflict detection algorithms

With the general concepts for detecting conflicts introduced, we now in algorithm 5 present how we progress until a conflict occurs.

Algorithm 5 Step to conflict (STC) algorithm

1: procedure STC
2: if CandidateList.SIZE > 0 then
3: event ← CandidateList.remove(0)
4: setActiveOnGraph(event)
5: ActiveList.addsorted(event)
6: for i to ActiveList.SIZE do
7: if ActiveList.get(i).EndTime < event.EndTime then
8: ActiveList.remove(i)
9: updateCost(event)
10: nextEvent ← event.getNext()
11: rescheduled ← updateArrDepTime(next)
12: if rescheduled then
13: RescheduledList.add(next)
14: CandidateList.addSorted(next)
15: if !checkConflictedEvents() then
16: StepToConflict()
17: else
18: GreedySolver()  \[\triangleright\text{All train events have been evaluated}\]
19: else
20: seRepairValid()
21: end

In the STC algorithm we first ensure the candidate list is not empty (line 2). If this is the case we have stepped through all the train events, meaning either all conflicts have been resolved or none were detected. Thus, we set that the timetable is a valid schedule (line 20). The first element from the candidate list is put in the active list, and set active on a point in the infrastructure (line 3-5). Further other events in the active list are removed if their end time is lower than the executing events start time (line 6-8). The executing events successor is loaded and its start and end time is updated if needed, and is later stored in the candidate list (line 10-14). We then check if we have a conflict between the active events. If no conflict was detected we make a recursive call to the STC algorithm. Otherwise we call the greedy solver which task is to repair the conflict.

In algorithm 6 the Check conflicted events (CCE) algorithm is presented.
Algorithm 6 Check conflicted events (CCE) algorithm

1:  procedure CCE
2:     conflictDetected ← false
3:     conflictedTrains ← 0
4:     for i to ActiveList.SIZE do
5:         trackInfo ← ActiveList.get(i).getActiveOnTrack()
6:         for j to ActiveList.SIZE do
7:             if ActiveList.get(j).getActiveTrack() = trackInfo then
8:                 conflictedTrains ++
9:                 TroubledEvents.addSorted(ActiveList.get(j))
10:        if conflictedTrains > trackInfo.nrOfTracks() then
11:            conflictDetected ← true
12:            Break
13:     return conflictDetected

CCE is used to check whether the events in the active list are occupying more physical resources, than provided by the railway infrastructure. Each event in the active list is iteratively checked against the remained events, with the objective to determine if they occupy the same rail track (line 4-12). If the number of trains occupying the very same rail track exceeds the capacity limit, the algorithm will indicate this by returning a true flag.

8.3.2 Conflict Resolution

A conflict is resolved by re-scheduling an event to be executed after the first event ending on the conflicted track segment, meaning the time when the track segment capacity allows the event to be re-scheduled to enter the segment. Depending on the sorting strategy for the candidate list, the order in which train events are executed will differ.

8.3.2.1 Candidate List Sorting Strategy

There exists multiple strategies for sorting the candidate list, we have selected strategy $s_0$. As concluded by Krasemann[62], the strategy is very effective in finding an initial solution. In contrary to the other sorting strategies. The disadvantage of sorting strategy $s_0$ lies in the effectiveness of finding further improvements of the current solution. As we are concerned with letting the greedy solver finding a initial solution fast, then allowing the meta-heuristics perturb the solution. Sorting strategy $s_0$ emerged as the most adequate strategy to apply in order to solve the re-scheduling problem. Krasemann ultimately stated strategies which were based on earliest start time and earliest track release time, had the best average performance. This further limited the choice to either strategy $s_0$ or $s_1$. 
8.3.2.2 The Conflict Resolution Table and the Neighbourhood

In our implemented system, we consider the neighbourhood of a solution a point in the timetable were we have a conflict. In each conflict which occurs, we are given a set of choices of what trains to reschedule. We formally define the different set of choices as the neighbourhood of a solution. As explained in chapter 4, this can be accomplished in a various of ways depending on the type of conflict. When a initial solution is generated by a greedy search algorithm, each of the choices along the search trajectory is stored in the conflict resolution table (CRT). Unitedly with the selected choices, the set of possible choices is also stored in the CRT. In order to identify train events, we generate unique identifiers for the events. This is done by combining the train identifier along with the event identifier. As each train has its own unique train ID, and a set of events which are to be scheduled. The combination of the two identifiers, will allow us to determine exactly which trains and at which event stage they have conflicted. In figure 8.4 we illustrate the concept of the CRT, were each previous re-scheduled train is stored in the left column, and the possible swaps are stored in the right column. The train IDs and events IDs are separated by a red delimiter.

<table>
<thead>
<tr>
<th>Previous Wait ID</th>
<th>Possible Swap IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conflict 1</td>
<td>91</td>
</tr>
<tr>
<td>Conflict 2</td>
<td>100</td>
</tr>
<tr>
<td>Conflict 3</td>
<td>80</td>
</tr>
</tbody>
</table>

Figure 8.4: Illustration of Conflict resolution table (CRT)

8.3.2.3 Description of Conflict repair algorithms

The implemented parallel and serial version of SA and TS, both communicate with the greedy algorithm presented in algorithm 7. The communication is performed through he conflict resolution table, which is modified by the meta-heuristics. If conflicted events have been solved in the past, we follow rules provided in the CRT (line 2-9). Further the conflicted event (troubled list) are prioritized according to condition 8.1.

\[ \text{Event.MakeUpTime}() - \text{Event.ArrivalTime}() \]  \hspace{1cm} (8.1)

The make up time, determines the total buffer time of the event. With this prioritizing, we ensure that trains which can catch up the most lost time are first in the troubled list. The Greedy solver then branches out for different possible
solutions, and continues until either we a dead end or all conflicts have been resolved (line 11-20).

Algorithm 7 Greedy Solver algorithm

1: procedure GREEDYSOLVER

2: if FindResolution(Swap, CRT, TroubledEvents) then

3:   waitingTrainID ← Swap.ID

4:   if RepairConflict(watingTrainID) then

5:     StepToConflict()

6:   else

7:     Break

8:   if RepairValid() then

9:     Break

10: else

11:   for i to TroubledEvents.SIZE do

12:     CRT.add(troubledEvents.get(i))

13:     waitingEvent ← troubledEvents.get(i)

14:     waitingTrainID ← waitingEvent.getPredecessor().ID

15:     if RepairConflict(watingTrainID) then

16:       StepToConflict()

17:     else

18:       Break

19:     if RepairValid() then

20:       Break

21: end

In algorithm 8 the repair conflict (RC) method is presented. After an event has been selected to be delayed, its unique identifier is passed to the RC algorithm. The objective of the method is to backtrack, to a state where we have found the event which is to be delayed (line 10-12). Pseudo-code for backtrack algorithm is omitted, as it can be considered as a reverse implementation of the STC algorithm.

The RC method further identifies a event which occupies the conflicted track the shortest (line 5-9). After we have identified the $event_{\text{minDepIndex}}$, and backtracked to the correct state. We updated the departure time of the event to be delayed, with the departure time of $event_{\text{minDepIndex}}$, the headway time, and the current delay (line 13-16).
Chapter 8. Implementation

Algorithm 8 Repair conflict (RC) algorithm

1: procedure RepairConflict(watingTrainID)
2:     for i to BackTrackList.SIZE do
3:         CandidateList.addSorted(BackTrackList.get(i))
4:     minDepTime ← ∞
5:     for i to TroubledEvents.SIZE do
6:         if TroubledEvents.get(i).ID ≠ watingTrainID then
7:             if TroubledEvents.get(i).endTime < minDepTime then
8:                 minDepIndex ← i
9:                 minDepTime ← TroubledEvents.get(i).endTime
10:            while CandidateList.get(LastIndex).ID ≠ watingTrainID do
11:                if !BackTrack() then
12:                    return false
13:        delayTime = TroubledEvents.get(minDepIndex).endTime
14:        delayTime ←= CandidateList.get(LastIndex).endTime + HeadWay
15:        delayTime ←= CandidateList.get(LastIndex).delayedArrivial
16:            CandidateList.get(LastIndex).setDelayedArrivial(delayTime)
17:     for i to BackTrackList.SIZE do
18:        CandidateList.addSorted(BackTrackList.get(i))
19:     return true

8.4 Heuristic Based Resolution (HBR)

This section describes the implemented algorithms: Simulated annealing, and tabu search. Which are based on the classical implementations presented in chapter 5. As there exists a wide range of search algorithms which have been applied both the JSSP and train re-scheduling. Our motivation for choosing to limit the scope of the thesis to meta-heuristics, is due to the fact of the problem size. Re-scheduling trains during disturbances is NP-hard, therefore a complete search is not feasible for this problem. Hence, meta-heuristics provide guidance for the search process and aids in directing the search to more promising areas of the search space. Moreover, heuristic based algorithm are able to obtain optimal solutions or near-optimal solutions in the limited time[20].

There exists three prominent meta-heuristics which have been applied to the problem of re-scheduling trains: Simulated annealing, tabu search, and genetic algorithm. All of which have a form of sophisticated guidance for solving optimization problems. As our focus lies in re-scheduling trains in a single-track
Chapter 8. Implementation

railway, we will most of the time have two to three trains which are in conflict. Henceforth, making genetic algorithm inadequate for the given problem, as there exist no need for GAs nature inspired operators when we have a choice of two to three trains to re-schedule.

8.4.1 Cost Function

Since the timetable data used for the study does not include preferences of the train dispatchers regarding the priority of trains, we are limited to select between a set of cost functions. We are therefore unable to consider cost functions which are based on weighted train delays. Presented in the following is the selected objective function, which only considers attributes of the trains. The selected cost function is inspired from Krasemann’s[63] total delay(TD)(see sec. 4.2.1.1) function. One modification from Krasemann’s objective function is that it only accumulates delays above a threshold of 3 minutes. The total delay of all trains at their end station is accumulated and denoted as \( \sum_{n=1}^{n} (D_i + 3) \). One advantage of the TD function is that it considers the delays for the whole railway system, and not focusing on singular trains. The cost function is further representative of how railway dispatchers are concerned of minimizing the total accumulated delay in the railway network. The number of delayed trains function is was not selected since it does not take into account the severity of occurred disturbances.

8.4.2 Parallel Simulated Annealing Algorithm

The parallel implementation for the simulated annealing algorithm is the variation of the simultaneous corresponding searches approach presented by Malek(see sec. 6.4.2.2). The parallel method is chosen since it has the property of exploring the search space independently in parallel, thereby achieving diversity in the search space. Periodical exchanges of the best solutions found so far are performed to avoid getting stuck in unfavorable parts of the search space. Each run of the SA differs both in run time and solutions visited due to its probabilistic properties. Performing a SA search multiple times in parallel on the same problem increases the accuracy of solutions to a problem, which is meant as its proximity to the global optimal solution. A drawback to this distributed approach is the uneven load balance among the different annealing processes. Since every annealing process differs in run time and they are allowed to finish before sharing their best solution, this can render several processes waiting for the last one to finish execution. The distributed character of this parallelization approach renders it fit for MIMD architectures as well.

Other methods focusing on speeding up the sequential implementation of SA achieve only mediocre speedups and/or are not well suited for our problem of train re-scheduling. A speedup by move acceleration would not be beneficial, since the calculation of the cost is not dividable into sub problems as the cost is
calculated during the search in our model. Evaluating moves in parallel would be possible for the given problem and yield a speedup for the sequential algorithm. Approaches aiming at preserving the convergence characteristics of the sequential algorithm evaluate only non-interacting moves for acceptance. The identification of those is a difficult problem [9]. Other approaches allow the evaluation of interacting moves for acceptance based on inaccurate information as a result of errors in cost calculations. Even though the convergence characteristics of the parallel algorithm are affected, the approach has the maximum scope for parallelism for speeding up the sequential algorithm. However, a speedup of the sequential algorithm would not improve the solution quality. The sequential algorithm, even when a slower cooling schedule is applicable due to the speedup achieved, might still converge to parts of the search space where good solutions, yet not the best, are found. By running asynchronous searches in parallel, the probability of converging to an advantageous part of the search space is enhanced and finding better or even optimal solutions becomes more likely. Asynchronous searches in parallel therefore render a more stable behaviour.

For our implementation parallelization is achieved by starting each SA process with the same initial solution and letting each process finish its cooling schedule asynchronously. Upon completing the cooling schedule, which in turn is determined by a specified run time, current results are shared among the processes. The best solution among those shared solutions is then accepted by all processes as the new current solution. The temperature is reset to its starting value and the asynchronous annealing process starts all over again. After a predefined number of synchronizations, the parallel SA search is finished. The communication is handled via a master process depicted as master node.

Algorithm 9 Parallel SA Initialization

1: procedure INIT PARA SA()
2: masterNode ← masterNode() \Comment{Master process for communication}
3: nrOfThreads ← maxThreads \Comment{Set the number of threads}
4: for iteration ≠ nrOfThreads do
5: \quad \quad G ← BuildGraph() \Comment{Build infrastructure}
6: \quad \quad T ← ReadTimeTable() \Comment{Parse timetable}
7: \quad \quad D ← ReadDelayTable() \Comment{Parse delay table}
8: \quad \quad S ← getInitialSolution(G, T, D)
9: \quad \quad ParaSA(initTemp, numberOfSync, runTime, S, masterNode)
10: \quad iteration ← iteration + + \Comment{Increment number of iterations}
11: end

In algorithm 9 the initializing for the parallel SA is presented. A master process is started that is used to handle the communication between threads (line 2). Each child process receives a copy of the train timetable, delay-table and
infrastructure data. Then the initial solution is generated by the greedy solver with the data passed (line 7) and the parameters for the parallel SA are set (line 8). Those are the initial temperature, number of synchronizations, the initial solution and the run time between synchronizations. The parallel SA also receives a reference to the master node needed for communication.

**Algorithm 10** Parallel Simulated Annealing (PSA)

1: \textbf{procedure} \textsc{ParaSA} (pInitTemp, pNrOfSync, pRunTime, pInitSolution, pMaster) 
2: \hspace{1em} \textit{best} $\leftarrow \infty$ 
3: \hspace{1em} \textit{initTemp} $\leftarrow$ pInitTemp 
4: \hspace{1em} \textit{parallelSyncIterations} $\leftarrow$ pNrOfSync 
5: \hspace{1em} \textit{executionTime} $\leftarrow$ pRunTime 
6: \hspace{1em} \textit{S} $\leftarrow$ initialSolution 
7: \hspace{1em} masterNode $\leftarrow$ pMaster 
8: \hspace{1em} for \textit{i} $<$ parallelSyncIterations do 
9: \hspace{2em} \textit{T} $\leftarrow$ initTemp \hspace{1em} \(\triangleright\) Set the temperature to starting point 
10: \hspace{2em} \textbf{while} \textit{executionTime} $>$ 0 do 
11: \hspace{3em} \textit{timeIterationStarted} $\leftarrow$ currentSystemTime() 
12: \hspace{3em} \textit{S}' $\leftarrow$ getRandomNeighbour\((N(S), S)\) \hspace{1em} \(\triangleright\) Random feasible neighbour 
13: \hspace{3em} \textbf{if} \textit{Cost(S')} $<$ \textit{best} \textbf{then} \hspace{1em} \(\triangleright\) Update the best cost found so far 
14: \hspace{3em} \hspace{1em} \textit{best} $=$ \textit{Cost(S')} 
15: \hspace{3em} \textbf{if} \textit{Cost(S')} $<$ \textit{Cost(S)} \textbf{then} \hspace{1em} \(\triangleright\) Accept a better solution 
16: \hspace{3em} \hspace{1em} \textit{S} $\leftarrow$ \textit{S}' 
17: \hspace{3em} \textbf{else} 
18: \hspace{4em} \textbf{if} Random(0, 1) $<$ \textit{e}^{\textit{Cost(S')}−\textit{Cost(S)}/\textit{T}} \textbf{then} 
19: \hspace{4em} \hspace{1em} \textit{S} $\leftarrow$ \textit{S}' \hspace{1em} \(\triangleright\) Possibly accept an inferior solution 
20: \hspace{4em} \hspace{1em} \hspace{1em} \textit{deltaT} $\leftarrow$ currentSystemTime() − timeIterationStarted 
21: \hspace{4em} \hspace{1em} \hspace{1em} \textit{executionTime} $\leftarrow$ \textit{executionTime} − \textit{deltaT} 
22: \hspace{4em} \hspace{1em} \hspace{1em} \textit{deltaCooling} $\leftarrow$ \((1/\textit{initTemp})^{1/(\textit{executionTime}/\textit{deltaT})}\) \hspace{1em} \(\triangleright\) Calculate cooling factor for elapsed time of current iteration 
23: \hspace{3em} \hspace{1em} \textit{T} $\leftarrow$ \textit{T} * \textit{deltaCooling} \hspace{1em} \(\triangleright\) Cooling the temperature 
24: \hspace{3em} masterNode.sendIsFinished() 
25: \hspace{3em} masterNode.sendResult(S) \hspace{1em} \(\triangleright\) Send result to master node 
26: \hspace{3em} \textbf{while} masterNode.getShouldWait() \textbf{do} \hspace{1em} \(\triangleright\) Wait for other threads 
27: \hspace{3em} \hspace{1em} Sleep(1) 
28: \hspace{3em} \hspace{1em} \textit{S} $\leftarrow$ CopyOf(masterNode.getBestResult()) \hspace{1em} \(\triangleright\) Set the best shared result as new initial solution 
29: \hspace{3em} masterNode.sendAllDone(best) 
30: \hspace{1em} \textbf{end}
Our parallel implementation of SA for the train re-scheduling problem is shown in algorithm 10. Since the different SA search threads are started asynchronously and independently, the implementation of the parallel SA is quite similar to the classical implementation. The main difference are the sharing of current solutions and the resetting of the temperature upon synchronization.

As the classical implementation, the SA process consists of an inner loop, which can be referred to as \textit{metropolisCycles} which is nested within an outer loop, often referred to as \textit{temperatureCycles}. The outer loop controls the number of synchronizations between the asynchronous SA processes. For each iteration of the outer loop, the current solution of the SA process shares its current solution by sending it to the master node (line 25) and waits for other processes to reach the same state of execution (line 26-27). When the master node has collected the current solutions from all SA processes, the processes are allowed to continue and accept the best among shared solutions returned by the master node as new current solution (line 28). The best solution found during the entire search is stored (line 13-14), since the SA does not necessarily converge to that best solution at the end of the search. When an asynchronous process has performed all iterations of the outer loop, it informs the master node (line 29) and sends the best solution found during the search. When all asynchronous processes have reached that point, the parallel SA is finished.

Within the inner loop, a neighbour to the current solution is generated using the "get random neighbour" function which receives the conflict resolution table (\textit{CRT}) and the current solution as input. The CRT is denoted as \textit{N(S)} in our implementation. The newly generated neighbour solution and current solution are evaluated against one another with regard to their calculated cost. The neighbour solution is either accepted if lower in cost or if being inferior, to a probability which is calculated using the boltzmann distribution (line 18-19).

The stopping criteria (line 9) determining the number of metropolis cycles being made is not set to a fixed number of iterations. Since we are operating within a limited time frame, the algorithm performs as many iterations as possible within a specified time (line 10). The elapsed execution time therefore acts as an stopping criteria for the inner loop. By letting the execution time determine the number of iterations, instead of waiting for each of the processes to reach equilibrium, a better load balance can be expected, since execution time for a SA process to reach equilibrium can vary greatly. Since the number of temperature reductions, i.e. iterations, is determined by the execution time, the cooling factor cannot be static. The final temperature shall reach the value of 1 under all circumstances. The cooling factor is therefore calculated by how much time has elapsed since the last iteration (line 23).
The inter-process communication for the parallel implementation of the SA search is presented in algorithm 11. An additional process handles the communication between the independent search processes which is referred to as masterNode. The intention of the masterNode is to eliminate race conditions when the parallel SA processes share their current solutions and except the best among those. Before the independent searches of parallel SA are started, the masterNode is initialized(line 1). Procedures denoted as synchronized entail that the procedures are atomic and not being executed simultaneously by multiple processes. Instead they are executed sequentially by queuing and letting the processes wait that are attempting to execute the procedure. Thereby the solutions between processes can be shared and the best one distributed without any inconsistency. Since the inter-process communication rarely occurs and the synchronized procedures have a fast execution time due to their simplicity, the resulting communication overhead between processes is low.
Algorithm 12 Generate random neighbour

1: procedure getRandomNeighbour(N(S), S)
2:     randomIndex ← Random(0, N(S).nrOfConflicts)
3:     currentWaitingTrain ← N(S).waitingTrainAtIndex[randomIndex]
4:     trainToWait ← Random(N(S).conflictedTrainsAtIndex[randomIndex])
5:     do
6:         S′ ← Swap(S, currentWaitingTrain, randomTrainToWait)
7:     while !N(S′).Valid()
8:     return S′

In algorithm 12 we present the "get random neighbour" function. An existing solution is perturbed by performing a random swap from all possible swaps in the neighbourhood. A swap determines which train shall wait in order to resolve a conflict i.e. which train will be re-scheduled and therefore further delayed. The neighbourhood $N(S)$ of a solution is represented by a conflict resolution table which contains the conflicts, the conflicting trains for each conflict, as well as the chosen train to re-schedule in order to solve the conflict. A random swap is performed by selecting a random index in the conflict resolution table(line 2) and changing which train shall wait to resolve the conflict at the selected index. This is done by randomly selecting a train among the trains involved in the conflict(line 4), under the condition that the new waiting train is not the same train as before. Since not all possible moves result in feasible schedules, the algorithm selects moves at random until a feasible schedule has been generated(line 5-7).

8.4.3 Parallel Tabu Search Algorithm

In the following we present the parallel implementation of tabu search. The chosen parallel adaptation is the "Parallel solution domain exploration"(discussed in section 6.4.1). This parallel adaptation is chosen since it allows for independent tabu search processes to explore parts of the search space. In our implementation this is achieved by providing each TS process with its own unique initial solution. Furthermore, the parallel adaptation is well suited for MIMD computer architectures.

In comparison to the other widely used approaches, the chosen parallel approach focuses on improving solution optimality. Evaluating the cost function in parallel would not yield to be beneficial for our given problem, as calculation of the cost cannot be divided into sub problems since the cost is calculated during the search. Further evaluating the neighbourhood in parallel might result in faster computation times, however it does not improve overall solution optimality. Decomposing the problem into sub-problems might yield to be a difficult task to achieve in our given problem, since not every choice in the along the search trajectory results in a feasible(valid) solution. Which might result in some
threads being jobless during the search. It order to make this kind of parallel implementation work properly, a lot of communication overhead and synchronization is introduced, which will slow down overall performance. Finally, having a shared tabu list among the processes would require considerable communication overhead and synchronization between the processes. A shared tabu list would further increase the probability of being stuck in a local optima, as all processes are forbidding moves. Which might result in areas being unexplored considering tabu search will always select the best move, leaving out moves of lower quality. This is a huge downfall, since tabu search can forbid a transition to a partially explored branch. Hence, the "parallel solution domain exploration" is selected as the parallel technique for the TS algorithm.

In algorithm 13 we present the initialization phase for the parallel TS algorithm. We first load the train timetable, delay-table, and infrastructure data in order to pass these into our greedy solver(line 7). The greedy solver then generates a initial solution, which is then perturbed(line 8) for each TS process to have its own initial solution. Lastly we start a TS process with its own initial solution and tabu list size(chosen randomly).

### Algorithm 13 Parallel TS Initialization

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>procedure InitParaTS</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>nrOfThreads ← maxThreads</td>
<td>Set the number of threads</td>
</tr>
<tr>
<td>3</td>
<td>problemSize ← totalNrEvents</td>
<td>Set problem size</td>
</tr>
<tr>
<td>4</td>
<td>for iteration ≠ nrOfThreads do</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>G ← BuildGraph()</td>
<td>Build infrastructure</td>
</tr>
<tr>
<td>6</td>
<td>T ← ReadTimeTable()</td>
<td>Parse timetable</td>
</tr>
<tr>
<td>7</td>
<td>D ← ReadDelayTable()</td>
<td>Parse delay table</td>
</tr>
<tr>
<td>8</td>
<td>S ← getInitialSolution(G, T, D)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>S' ← Permutation(S)</td>
<td>Perturb initial solution</td>
</tr>
<tr>
<td>10</td>
<td>TabuSize ← 0.3 * Random(problemSize, problemSize + 10)</td>
<td>Set list size at random</td>
</tr>
<tr>
<td>11</td>
<td>PTS(TabuSize, S')</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>iteration ← iteration ++</td>
<td>Increment number of iterations</td>
</tr>
<tr>
<td>13</td>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

In algorithm 14 the adaptation of TS for the train re-scheduling problem is presented. The neighbour as earlier stated, is generated using the conflict resolution table, this is used in order to evaluate possible moves in the neighbourhood. In the pseudo-code the CRT is referred to as \( N(S) \).

As with the classical implementation, we evaluate the best neighbours by passing the neighbourhood, the current solution, and the tabu list to the "get best neighbour" function(line 6). We further employ intermediate-term memory, this is accomplished by storing a limited set of the best solutions found. The
intensification phase is started when a given set of iterations have passed, were we have no improvement of the current best solution. One of the best solutions found is then set to the current solution, in order for the search to further investigate areas of higher quality.

In Algorithm 14, we introduce the "get best neighbour" function. The classic two-change\(^1\) neighbourhood is employed, in order to find solutions of higher quality. A "move" in this context consists of exchanging two elements of the current solution. An exchange will determine which train is to be re-scheduled (i.e., delayed further), and what train is to follow its planned arrival and departure time.

In each iteration we examine all possible moves in the neighborhood, and choose the best non-tabu element. This exchange process is performed until the permutation becomes the new working solution. The neighbourhood \(N(S)\) will contain the number of conflicts, and the number of trains which have conflicted at each conflict stage. Each of the neighbours are evaluated based on the cost function \(f(\text{candidate})\), which purpose is to find the best neighbour (line 5). The candidates in the neighbourhood are evaluated one by one. If the best candidate found is non-tabu we swap the current solution (line 7), and store the unique identifier of the move (e.g., \(\text{Train}_i \rightarrow \text{Train}_j\)) in the tabu list. We then use the step to conflict function to repair the new proposed solution. If we have a non-valid repair, we forbid the move and try another move (line 9-11). We otherwise

\(^{1}\) Also denoted as swap
check if the new proposed solution is better, and updated the current solution if this is the case (line 12-14).

Algorithm 15 Parallel Tabu Search Get Best Neighbour

1:  procedure ParaTSGBN(N(S), TabuList, initialSolution)
2:     S ← initialSolution
3:     for N(S).nrOfConflicts do
4:         for i to N(S).nrOfConflictedTrains do
5:             if f(Train_i) < f(Train_i+1) then  ▷ Check better move
6:                if !TabuList(Train_i, Train_j) then
7:                   S' ← Swap(S, Train_i, Train_i + 1)  ▷ Perturb solution
8:                   STC()  ▷ Repair timetable
9:                if !N(S').Valid() then  ▷ Check if valid repair
10:                   TabuList(Train_i, Train_i + 1)  ▷ Update tabu list
11:                   Continue  ▷ Check if we have a better solution
12:             if Cost(S') < Cost(S) then
13:                S ← S'
14:                TabuList(Train_i, Train_i + 1)  ▷ Update tabu list
15:     return S

8.4.4 System Restrictions

In the following we present how we have simplified the problem, and the corresponding constraint are explained.

8.4.4.1 Train restrictions

Resolving a conflict consists of re-timing a trip at its departure time, up to the point the conflict is resolved. The strategy stems from the knowledge of train operators not wanting to plan for passenger trains being delayed after their departure time. The selected strategy has its drawbacks, as it will settle for when a conflict is resolved immediately. This will result in some areas of the search space not being explored. However at the same time the strategy provides an effective approach for limiting the search space, as exploring areas beyond the settle point will cause the problem to be infeasible to solve within reasonable time.

Furthermore, in the implemented system a train is not allowed to take advantage of the timetables buffer time, unless its delayed. Lastly a train events end time is immediately followed by the next event start time, in the sorted event list.
8.4.4.2 Infrastructure restrictions

The infrastructure consists of a limited set of resources. Train events are therefore not to exceed the track capacity of the infrastructure. Further each train event is allocated precisely one track per section it occupies.

8.5 Output and System Validation

The implementation of the railway re-scheduling system in the end outputs a new feasible train timetable. It is therefore of importance to validate both the timetable and the validity of the implementation.

In the following we present three different techniques which have been employed during the development of the system, in order to validate the output.

8.5.1 Timetable Conflict detection

Each new timetable which is generated is passed to the conflict detection algorithm (STC algorithm). The very same process is therefore applied as when repairing a conflict to the new timetable which has been generated by the algorithms: the greedy solver, simulated annealing, or tabu search. If a conflict is detected in the timetable, then this is considered a non-valid timetable. Otherwise we move on to the second test of validating each train event.

8.5.2 Timetable Validation of the train events

The second technique employed consists of checking that all of the trains events are present in the new generated timetable. This is done by firstly calculating the number of events present in the original conflicted timetable, and then matching the calculated value to the number of events in the new generated timetable. Lastly we check if all the event identifiers are unique, this will test if we have redundancy in the new timetable. If the new timetable passes both of the aforementioned steps, then we move on to the final test which is the "no wait constraint" test.

8.5.3 Timetable No wait constraint

The final test which is employed on the timetable is the "no wait constraint" test. In the following test, we check that one train events departure time matches the subsequent events arrival time. This process is performed iteratively throughout the whole timetable. If the "no wait constraint" condition is satisfied by all train events, then the timetable passes this test. If all three of the mentioned test are passed by the timetable, then we have a valid timetable.
Chapter 9

Results

In this chapter we apply the implemented versions of simulated annealing and tabu search for the railway re-scheduling problem. The main objective of the experiments is to determine if parallel versions of simulated annealing and tabu search are able to increase solution optimality.

9.1 Experiment Details

In table 9.1 we have selected 10 disturbance scenarios. For each scenario we include a unique identifier, the time of the disturbance($T_0$), a general description, and lastly we specify the problem size. The problem size specifies the total number of trains, the number of active trains in parenthesis, and the number of active events. The purpose of the experiments is primarily to evaluate how much disturbed situations can be improved with the usage of the SA and TS algorithms. Secondly we are concerned with the effectiveness of adopting parallel computing to the algorithms.

Each of the disturbance scenarios share a common set of characteristics. These characteristics are selected according to reasonable intervals. For each scenario we decide the following:

- Which train is to be delayed.
- Which event of the train is to be delayed.
- The magnitude of the disturbance, i.e. how long we should delay the train.

The 10 unique disturbance scenarios are each to be investigated by both sequential and parallel versions of the meta heuristics. Each of the algorithms are given a limited execution time, in this case 30 seconds. Due to the randomness of the meta-heuristics, we collect the necessary data from 100 runs for each disturbance scenario. Important to note is that the parallel versions are executed with the usage of 4 threads. Further when each algorithm has completed 100 runs of each scenario we collect the following data:
Chapter 9. Results

Best Value: Denotes the absolute best value obtained from all runs of the experiments.

Average Value: Denotes the average of the best value obtained in each of the runs.

Worst Value: Denotes the absolute worst of the best values obtained in all of the runs.

Table 9.1: Experimental disturbance scenarios

<table>
<thead>
<tr>
<th>Scenario</th>
<th>$T_0$</th>
<th>Disturbance description</th>
<th>Problem-size (#trains/#events)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13:29</td>
<td>Train 93 departs 26 minutes late from Riksgränsen</td>
<td>103(61)/2281</td>
</tr>
<tr>
<td>2</td>
<td>10:36</td>
<td>Train 96 departs 9 minutes late from Luleå</td>
<td>103(76)/2923</td>
</tr>
<tr>
<td>3</td>
<td>11:02</td>
<td>Train 96 departs 10 minutes late from Boden</td>
<td>103(74)/2840</td>
</tr>
<tr>
<td>4</td>
<td>01:06</td>
<td>Train 4005 departs 2 hours 13 minutes late from Riksgränsen</td>
<td>103(103)/4344</td>
</tr>
<tr>
<td>5</td>
<td>09:49</td>
<td>Train 7155 departs 9 minutes late from Kiruna malmbangård</td>
<td>103(79)/3068</td>
</tr>
<tr>
<td>6</td>
<td>13:24</td>
<td>Train 9914 departs 15 minutes late from Kiruna malmbangård</td>
<td>103(61)/2003</td>
</tr>
<tr>
<td>7</td>
<td>13:24</td>
<td>Train 9914 departs 20 minutes late from Kiruna malmbangård</td>
<td>103(61)/2003</td>
</tr>
<tr>
<td>8</td>
<td>19:05</td>
<td>Train 9964 departs 58 minutes late from Luleå</td>
<td>103(27)/666</td>
</tr>
<tr>
<td>9</td>
<td>05:21</td>
<td>Train 19941 departs 16 minutes late from Lappberg</td>
<td>103(99)/3861</td>
</tr>
<tr>
<td>10</td>
<td>11:48</td>
<td>Train 9106 departs 1 hour and 7 minutes late from Notviken</td>
<td>103(72)/2690</td>
</tr>
</tbody>
</table>

Furthermore, we will investigate how the greedy solver calculates the solutions values for the different disturbance scenarios. Lastly we can summarize the attributes which are of interest for the experiment as follows:

- Improvement from the initial solution generated by the greedy solver.
- Total cost for the sequential and parallel versions of the meta-heuristics, based on the objective function(TD+3 see sec. 8.4.1).
Chapter 9. Results

- Total search time for finding an initial solution.
- Total time to build the model.
- The memory footprint of the application.
- The efficiency of the parallel implementation compared to the sequential. Efficiency is calculated as the total number of conflicts resolved per millisecond.

9.1.1 Algorithm parameter settings

For the TS algorithm, its stopping criteria was set to a limited execution time. The tabu list was in the experiment set to be 30% of the total problem size. In this thesis the problem size is viewed as the total number of events for the scenario. The parallel TS introduces a random seed, whereby the tabu size can be larger than 30% of the problem size. The list length for PTS can at maximum be a length of 10 greater than 30% of the problem size. Both the versions of TS have a fixed number for iterations without improvement, this value is set to 30. After 30 iterations without any improvement in solution optimality, we start the intensification phase.

For both the sequential and parallel SA algorithm the initial temperature was set to 10000 and the cooling factor was set to reach a final temperature of 1. The stopping criteria for the sequential algorithm was set to be reached after 30 seconds. Since the parallel SA resets the temperature to the initial value after synchronization and the cooling schedule is reset, the stopping criteria was set to 7.5 seconds. This way the parallel processes reach a total run time of 30 seconds for the number of 4 synchronizations i.e. temperature cycles. The level of parallelism was chosen of 4 threads for the 4-core CPU in use. Since the cooling factor is calculated by the algorithm itself depending on the time elapsed since the last feasible schedule was explored, no parameter settings are required. These values were obtained conducting smaller experiments and yield satisfactory results for the chosen scenarios in the experiment. These smaller experiments, which are not included in this thesis due to low significance, were performed on the 10 chosen scenarios.

Lastly, in order to run sequential versions of SA and TS, we run the algorithms using one thread.

9.2 Experimental Results

We conclude the chapter with providing the results of each of the experiments conducted. The experiments clearly show the benefits in having a railway re-scheduling decision support system.
Chapter 9. Results

9.2.1 Model Evaluation

In table 9.2 we present the computation times for building the model and finding an initial solution. Further the cost of initial solutions calculated by the greedy solver (GS) for each scenario is likewise included. The time to build the model includes the following:

- The time to read the infrastructure, timetables, and delay table.
- The time to step up until a conflict has been detected.

Whereas the computation time for GS includes the time taken to generate an initial solution, from the point the conflict was detected.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Model Computation time(ms)</th>
<th>GS Computation time(ms)</th>
<th>Initial solution(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>62</td>
<td>174</td>
<td>1257</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>53</td>
<td>461</td>
<td>3500</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>611</td>
<td>10217</td>
</tr>
<tr>
<td>5</td>
<td>152</td>
<td>192</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>61</td>
<td>189</td>
<td>1407</td>
</tr>
<tr>
<td>7</td>
<td>62</td>
<td>200</td>
<td>1748</td>
</tr>
<tr>
<td>8</td>
<td>87</td>
<td>48</td>
<td>1671</td>
</tr>
<tr>
<td>9</td>
<td>28</td>
<td>112</td>
<td>1038</td>
</tr>
<tr>
<td>10</td>
<td>56</td>
<td>209</td>
<td>5231</td>
</tr>
</tbody>
</table>

9.2.2 Meta-heuristics Evaluation

Due to the probabilistic characteristics of the parallel implementations, the results in table 9.3 and 9.4 are obtained by running each of the disturbance scenarios 100 times. In table 9.3, an overview for the results of the SA algorithm is presented. Whereby each disturbance scenario is solved by the sequential SA(SSA), and the parallel SA(PSA). For each scenario we present a best, average, and worst value (given in seconds) of the minimized delays.
Table 9.3: Overview of Solution Optimality for SA

<table>
<thead>
<tr>
<th>Scenario</th>
<th>SSA(s)</th>
<th></th>
<th>PSA(s)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Avg</td>
<td>Worst</td>
<td>Best</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>78</td>
<td>267</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>3766</td>
<td>4072</td>
<td>4186</td>
<td>3555</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>150</td>
<td>186</td>
<td>515</td>
<td>150</td>
</tr>
<tr>
<td>7</td>
<td>491</td>
<td>531</td>
<td>737</td>
<td>491</td>
</tr>
<tr>
<td>8</td>
<td>1572</td>
<td>1580</td>
<td>1658</td>
<td>1572</td>
</tr>
<tr>
<td>9</td>
<td>143</td>
<td>143</td>
<td>143</td>
<td>143</td>
</tr>
<tr>
<td>10</td>
<td>3793</td>
<td>3842</td>
<td>3938</td>
<td>3793</td>
</tr>
</tbody>
</table>

In table 9.4 we correspondingly present the results of the sequential TS(STS), and the parallel TS(PTS).

Table 9.4: Overview of Solution Optimality for TS

<table>
<thead>
<tr>
<th>Scenario</th>
<th>STS(s)</th>
<th></th>
<th>PTS(s)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Avg</td>
<td>Worst</td>
<td>Best</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4125</td>
<td>4125</td>
<td>4125</td>
<td>3929</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>150</td>
<td>150</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>7</td>
<td>491</td>
<td>491</td>
<td>491</td>
<td>491</td>
</tr>
<tr>
<td>8</td>
<td>1671</td>
<td>1671</td>
<td>1671</td>
<td>1572</td>
</tr>
<tr>
<td>9</td>
<td>429</td>
<td>429</td>
<td>429</td>
<td>143</td>
</tr>
<tr>
<td>10</td>
<td>3847</td>
<td>3847</td>
<td>3847</td>
<td>3793</td>
</tr>
</tbody>
</table>

9.2.2.1 Meta-heuristic improvement from initial solution

In figure 9.1 and 9.2 we illustrate how SA and TS improves the cost from the initial solution. The average of the best found cost is depicted in the figures for the meta-heuristics. Figure 9.1 furthermore illustrates the convergence behaviour of PSA by depicting the average cost of accepted solutions, i.e. the average cost of schedules explored during the search.
In figure 9.1 we can observe how the improvement of the best solution found corresponds to the number of iterations past. *PSA* converges significantly faster and seemingly to a lower value than *SSA*. The inferior solutions accepted by the *PSA* algorithm seem to decrease in both cost and number, indicated by the decreasing height and width of the four peaks for every iteration. The shared solution upon synchronization acting as new initial solution seems to result in a faster convergence.

Figure 9.1: Overview over the improvement of cost for SA for scenario 1

Figure 9.2: Overview over the improvement of cost for TS for scenario 1
Chapter 9. Results

The improvement of cost for PTS and STS (fig. 9.2), shows the differences in cost for both TS versions. One observation is that PTS is able to find better solutions per iteration. However, it converges to the optimal solution later than STS for the given scenario.

In figure 9.3 we depict the reduction between the average value (see tab. 9.4 and 9.3) of the meta-heuristics and the initial solution cost found from the greedy solver. The reduction is shown for each of the 10 scenarios for the sequential and parallel TS and SA. The higher the bar, the better the algorithm is performing in comparison to the initial solution cost.
Figure 9.3: Overview of improvement from initial solution cost of the meta-heuristics.
Chapter 9. Results

9.2.2.2 Conflicts resolved by sequential and parallel implementations

In table 9.5 we show how many conflicts the meta-heuristics solve per millisecond, i.e. we illustrate the efficiency of the methods. The efficiency of the methods is measured according to how many conflicts they manage to solve over time (ms). Results of the efficiency are collected for both the sequential and parallel versions of the heuristics. The results are gathered as an average value from the execution of 100 runs for each scenario. With the execution of 4 threads, we therefore expect a maximum efficiency of 4.

Table 9.5: Overview of the number of conflicts solved per millisecond (ms) for the meta-heuristics

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Conflicts solved/ms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SSA</td>
</tr>
<tr>
<td>1</td>
<td>7.1</td>
</tr>
<tr>
<td>2</td>
<td>N/A</td>
</tr>
<tr>
<td>3</td>
<td>7.3</td>
</tr>
<tr>
<td>4</td>
<td>7.0</td>
</tr>
<tr>
<td>5</td>
<td>7.0</td>
</tr>
<tr>
<td>6</td>
<td>7.4</td>
</tr>
<tr>
<td>7</td>
<td>7.3</td>
</tr>
<tr>
<td>8</td>
<td>4.7</td>
</tr>
<tr>
<td>9</td>
<td>6.8</td>
</tr>
<tr>
<td>10</td>
<td>6.9</td>
</tr>
</tbody>
</table>

9.2.2.3 Memory Usage

In figure 9.4 and 9.5 we present the memory usage for PSA and PTS. The figures illustrate how memory usage relates to the execution time of roughly 30 seconds.

Figure 9.4: PSA Memory Usage for an instance of Scenario 1
From figure 9.4, we can observe the application reaches a peak memory usage of roughly 750 MB. Significant drops of memory usage can further be observed throughout the execution time.

The *PTS* algorithm likewise consumes roughly 750 MB at most (fig. 9.5). We can determine we are allocating less memory with the *PTS* algorithm. This can be shown by the frequency of peaks in the figures, less peaks are due to less re-allocation of the memory.

Furthermore, we can show from a run of scenario 1 the garbage collector is freeing a total of 72.95 GB for the execution of 30 seconds for *PSA*. For a small scale experiment with the experimental PC, writing an integer of size 4096 to an array-list in Java has been calculated to yield a throughput of roughly 2.5 GB/s. Resulting in a total throughput of 75 GB for the execution of 30 seconds.
Chapter 10

Analysis

In the following chapter the results from the experiments are evaluated. The chapter gives insight to how both the sequential and parallel versions perform for the 10 given scenarios.

10.1 Reflection of the model

It can be observed from the Model and system performance table (tab. 9.2) that building the model is generally less time consuming than finding the initial solution with the greedy solver. Two exceptions can be found for scenario 2 and 8. Various reason can be established as why this is the case. One of the main reason can be based on the fact that the complexity of the disturbance is very low, leading to much less work for the greedy solver. As seen in scenario 2, the greedy solver is able to find the optimal solution of 0 cost. Another reason which can be established is how late the disturbance occurs, meaning at which point \( T_0 \) has been set. Depending on how late in the time schedule the disturbance occurs, the model has to step further in the timetable, before the conflict(s) have been detected. Meaning a disturbance which occurs later in the timetable, will result in lesser work for the greedy solver.

As can be observed from the total final delay cost for the greedy solver, we are able to find feasible solution quickly. Some even with optimal results, as with scenario 2 and 5. Although the disturbance delay for both of these scenarios was set at 9 minutes. In contrast to other disturbance scenarios which are generally set to be much more severe.

10.2 Reflection of the meta-heuristics

The meta-heuristics are in this section formally evaluated, in order to conclude if parallelism benefits the train re-scheduling problem.
10.2.1 Solution Optimality evaluation

10.2.1.1 Tabu Search

The results of solution optimality for the sequential and parallel tabu search are presented in table 9.4. We can observe that generally the average cost of \(PTS\) for each scenario, is lower than the best found cost of \(STS\). We can further state the sum of all average values for the scenarios is 270 seconds less for \(PTS\). Two derogations are scenario 1 and 2, where both the average and the worst cost of \(PTS\) is much higher than \(STS\). This deviation can be explained as the parallel implementations workload per thread not being completely linear. As the sequential version of \(TS\) in this case is able to perform more work than one thread of \(PTS\), thereby managing to find a better swap through its current best solution. The \(STS\) therefore has the advantage of being faster in some cases. One explanation as why the workload is not scaling linearly for each thread is since they are working on their own copy of the timetables, and infrastructure. This consumes more RAM, and since Java has been used for the implementation we speculate the garbage collector(GC) combined with a limited memory bandwidth might slow down the performance for \(PTS\). As the GC needs to take care of more objects, with each added thread which has its independent data. The GC poses a problem for the parallel implementation chosen for \(TS\), as the GC dynamically chooses when to collect the unused objects. Further, experimentation on the GC settings have shown to have a great impact on the performance of the application.

We can further observe that we are able to find solutions of higher quality with the parallel implementation within the same time frame. This is due to the fact of each executing thread having its own unique initial solution, whereby a form of diversification arises. This allows the threads to potentially explore different parts of the search space. From the data we can conclude introducing a form of randomness to the initial solution, yields to be beneficial for the train re-scheduling problem. We further can identify cases were the worst found solution by \(PTS\) is the absolute best found solution of \(STS\), for instance scenario 8 and 9.

Evaluating the absolute best values found of \(PTS\) and \(STS\), we are able to determine \(PTS\) is better at finding solutions of higher quality. However \(PTS\) is more unstable, as the efficiency is not scaling as expected. The \(STS\) is always able to maintain the same best, average, and worst cost for each scenario. \(STS\) is further the most stable implementation considering the implementations of \(TS\) and \(SA\).

10.2.1.2 Simulated annealing

The results collected from the experiments regarding the solution optimality for both the sequential and parallel \textit{Simulated annealing} are shown in table 9.3. The average cost of \(PSA\) is lower or equal for all 10 scenarios which tells us that the solutions found on average are generally lower in cost compared to \(SSA\). It can
further be observed that the sum of average values for all scenarios for PSA is 574 seconds lower than for SSA. The worst solution found for all scenarios except 4 and 10, correspond to the best solution found. PSA finds the best solution at every run thus improving the solution quality. We can therefore conclude that PSA has a more stable behaviour than SSA. For scenario 4 the PSA heuristic finds a lower best cost than SSA does. Furthermore, PSA finds a lower cost solution on average than the best solution cost found by SSA for scenario 4.

The reason for the more stable behaviour and the ability to find lower cost solutions can be explained by mainly two reasons. PSA is able to search a larger portion of the search space since the asynchronous searches are executed in parallel. Although an annealing process is able to find satisfying solutions, it does not necessarily converge to the part of the search space where the best solutions are found. This is due to the relatively fast cooling schedule that has to be executed within the limited time frame of 30 seconds and the probabilistic properties regarding the acceptance of inferior solutions. The parallel annealing processes can converge to different parts of the search space hence achieving diversity in the search space. Therefore the probability of at least one process converging to the most promising part of the search space and finding better solutions or even the optimal, enhances the overall solution quality. The resetting of the temperature allows only few synchronizations to be effective since the annealing process can be viewed as to being restarted at every synchronization. The chosen number of synchronizations of 4 is seemingly low, yet gave the best results during our small scale parameter optimization. Other temperature updating strategies might allow more frequent synchronizations to be beneficial. Finally we can conclude PSA performs better in comparison to SSA.

10.2.1.3 Tabu search and Simulated Annealing

Comparing PSA and PTS, we can observe that PSA is better performing at reducing the accumulated delay. Calculating the sum of the average values for each scenario, we can conclude PSA is able to reduce the total delay 585 seconds more than PTS.

Since the obtained data and six of the selected scenarios are derived from the work of Krasemann[63]. We now present how our developed algorithms compare to the results of Krasemann. In our study the six common disturbance scenarios are as follows: 1, 2, 3, 5, 6, and 7. We can observe TS and SA are able to find the optimal solution for the problems in all besides one case. In scenarios 7 we are obtaining a best value of 491 seconds, while Krasemann is able to further minimize the delay to 450 seconds. This deviation might be caused by how we have chosen to limit the search space. Further how the constraints for the railway system have been set.

In Krasemann’s[61] study, TS is able to outperform SA. This is however inconsistent with what we conclude from our study. One deviation is that our data-set
is concerned with traffic in the north of Sweden, and Kraseman’s data focuses on the stretch between Karlskrona and Kristianstad. As we are addressing different parts of Sweden, the complexity for solving the problem therefore varies. Further we are not addressing the same type of disturbance scenarios as tackled in [61]. In [34, 30] they concluded SA and TS both perform equally well, for the single tracked railway scheduling problem. Lake et al.[37] argue SA is better performing than TS, for the railway maintenance problem. We can therefore conclude there is a form of inconsistency when evaluating these two meta-heuristics. Motivations as why this might be the case are different, few have already been discussed. Other reasons might be how a move is defined, and how past academic work have chosen to define their neighbourhood.

10.2.2 Improvement from initial solution evaluation

In figure 9.2 we observe the cost improvement for TS per passing iteration. We can conclude PTS is per iteration better at finding solutions of higher quality. However, in this given scenario the STS is able to converge to 0 faster than PTS. This is be to expected from the results gathered in table 9.4, as PTS has a worse average cost than STS for scenario 1.

Figure 9.1 shows the cost improvement for SA over the number of iterations. It can be observed that the PSA both converges faster and seemingly to a lower solution cost compared to SSA. From that we can conclude that PSA is better at finding solutions of higher quality while the number of iterations needed to achieve that quality is decreased. The PSA therefore shows a higher efficiency in traversing the search space.

The temperature updating strategy of resetting the temperature at the periodical synchronization between the processes can somewhat be viewed as restarting the SA processes. The annealing processes therefore start accepting inferior solutions with a high probability. Restarting SA can in some cases be beneficial since going back to a good solution, i.e. the one received at synchronization, might be more effective than moving from the current state. As can be observed from figure 9.1, the new initial solution obtained during synchronization does lead to a faster convergence indicated by the acceptance lower cost solutions. We can observe that the 4 peaks depicting accepted solutions are becoming smaller, both in height and width, for every synchronization.

10.2.3 Conflicts solved evaluation

From table 9.5 we present the efficiency of the parallel implementations in comparison to the sequential versions. We can observe for the TS algorithm that the parallel version is performing roughly 2.5 - 3.5 times more work than the sequential version. One deviation can be perceived for scenario 9, were the parallel version is solving 8 times more conflicts than the sequential version. This
can be explained as we have a forbidden list (tabu list) in the TS algorithm, this hinders the conflict counter from incrementing if it tries to re-do forbidden moves. Thereby, yielding the advantages of the parallel implementation in cases where one thread is getting stuck in already explored areas of the search space.

For SA we can observe that the parallel implementation performs about 2.1 - 3.0 times more work than the sequential version for the 10 selected scenarios. On average, the PSA performs roughly 2.66 times more work. The lowest efficiency for PSA can be found in scenario 8, where a factor of 2.1 is achieved. The larger number of train events to be re-scheduled renders the scenario more memory demanding than other scenarios. While SA struggles at solving conflicts for scenario 8, TS manages to solve as many conflicts as in other scenarios. This can be delimitied to the use of the different neighbourhood functions of the algorithms. The getBestNeighbour of TS contra the getRandomNeighbour of SA allows the TS to create schedules closer to the intended schedule, resulting in less complex conflicts therefore enabling the TS to perform as expected.

Different speculations arise as why we the parallel versions are not performing 4 times more work, with the execution of 4 threads. The main bottleneck can be delimitied to the RAM, as the problem is memory demanding.

10.2.4 Memory Usage evaluation

The memory usage of PSA and PTS can be observed in figure 9.4 and 9.5. From the figures we can determine that both algorithms are heavily memory demanding in terms of re-allocation of memory. Every time the GC decides to free the memory of unused objects, a significant drop in the used memory can be observed. We speculate that the memory bandwidth is the main bottleneck in the system when running multiple search processes in parallel. We further speculate this may be caused by the fact that a dominant portion of the application is allocated on the heap. Whereas it would be ideal allocating more objects on the stack, which is cheaper and de-allocation of the objects would be free on the stack.

Furthermore we can observe that PSA re-allocates more memory than PTS. One reason can be delimitied to the usage of the tabu list, which prevents PTS from revisiting already explored areas of the search space. This can cause the PTS to spend processing time without generating new schedules using the greedySolver, while PSA constantly evaluates new schedules without any interruption other than that at synchronization.

Lastly, we can conclude the GC is freeing 72.95 GB for the execution of 30 seconds. Having a limitation of 75 GB memory throughput, we are safe to assume we are maximizing the memory bandwidth usage for the parallel implementations. As the application is not only cornered with freeing memory, but also allocating memory.
Chapter 11
Conclusions and Future Work

In this final chapter we conclude the outcomes and contributions of this thesis. In addition, we once again highlight how this thesis differentiates from past academic work done in the area of train re-scheduling. Lastly, future research is proposed based on the outcome of the experiments.

11.1 Conclusion

In this thesis we have studied the area of optimization, more precisely the area of real-time re-scheduling of trains. Two different meta-heuristics, simulated annealing and tabu search, have in this thesis been developed. Both sequential and parallel versions of the meta-heuristics have been thoroughly evaluated. Their design, implementation, and final results are in the thesis presented. As of today, train re-scheduling is done manually by train dispatchers. This task can become increasingly complex and cumbersome for dispatchers to handle. Thus, there is an essential need for a decision support system for assisting the train dispatchers. Very limited research has been done in the area of parallelism and re-scheduling of trains. In the past academic work no attempt has been made in analysing parallel simulated annealing and tabu search within the area of train re-scheduling. We therefore in this thesis cover this limitation and present the effectiveness of parallelism for train re-scheduling.

The results of the experiments showed we are able to reduce the total accumulated delay with the developed parallel meta-heuristics. The parallel implementation chosen for TS starts independent search processes, where each tabu processes has its own unique initial solution to diverse the search. For SA independent searches are started in parallel and due to its fundamental probabilistic character diversity is achieved. The SA search processes synchronize after a time period to share the best solution found, which aims to guide the searches to areas of higher quality. Due to the probabilistic nature of the parallel implementations, we in this thesis have focused on the average of the best values found. From the average we can conclude PTS is able to reduce the sum of the cost for all scenarios by 270 seconds, when compared to STS. Further the results showed PTS is able to solve roughly 2.5 to 3.5 times more conflicts per milliseconds than STS. We
can conclude PTS is generally better at finding solutions for each passing iteration. Moreover, PSA is able to reduce the sum of the total delay by 574 seconds in consideration to SSA. PSA is further able to perform 2.1 to 3.0 times more work than the sequential version. Finally the results showed SSA is more stable, and converges faster to an near optimal solution than SSA. Comparing PTS and PSA, we are able to determine that PSA is able to reduce the total delay by 585 seconds more than PTS. We can therefore state PSA is able to outperform PTS, as PSA is able to greatly reduce the accumulated delay for the 10 given scenarios.

We further conclude one effective approach for limiting the search space, is to re-time a trip at its departure time, up to the point when the conflict is resolved. This provides a varuous approach, as the repair conflict function will settle immediately after a feasible repair of the timetable. If we were not to prune the search space as explained, the space would become inhibitable to find any solutions in.

### 11.2 Future Work

The performance of TS is very dependent on the tabu list size, and the number of iterations used as stopping criteria. In the conducted experiments a fixed length for the tabu list has been used. Using a tabu list length which dynamically changes might yield to be a better approach. Further the chosen design for PTS has one main drawback, as only one thread receives a solution of high quality. The rest of the threads receives a randomly perturbed solution, which solution quality might yield to be very low.

For SA, the performance is dependent on the used temperature and the cooling factor used for the given problem. These settings for TS and SA have received very little focus in this thesis, as the parameters have been set according to past academic work. The main bottleneck for the chosen parallel implementations has been delimited to the RAM. Finally our suggestions for future work can be summarized as follows:

- Further experimentation on the settings for TS and SA.
- Implement a permutation function for PTS, which generates solutions of roughly equal quality for each thread. One suggestion is to generate the initial solution according to different train prioritizations.
- Investigate how different synchronization strategies for PSA can be used to improve the efficiency of the search. Areas of interest are how the temperature update upon synchronization is handled, how the the identification of a state of equilibrium might be used to result in a higher efficiency, and how the stopping criteria for the different search processes could handle the request for synchronization.
• Investigate if the main bottleneck for the parallel implementation is the RAM. By for instance, implementing the system in the programming language C, or C++.

• Further improve the model to support data-sets for double tracked railway system, as this study focused on single tracked railway systems.
References


References


