Improving RCPSP algorithms using machine learning methods

F.P. Verburg

Master's Thesis

TUDelft

Improving RCPSP algorithms using machine learning methods

by

F.P. Verburg

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday June 21, 2018 at 10:00 AM.

Project duration:	September 1, 2017 – June 21, 2018	
Thesis committee:	Dr. M.M. de Weerdt,	TU Delft, supervisor
	Dr. N. Yorke-Smith,	TU Delft
	Dr. ir. S.E. Verwer,	TU Delft

TU Delft

Abstract

For performing technical maintenance, it is important to keep a detailed schedule of resources and temporal constraints. The Resource Constrained Project Scheduling Problem (RCPSP) is a well defined scheduling model with both resources and temporal constraints. Precedence Constraint Posting (PCP) is a technique to solve the NP-hard RCPSP problem, that currently uses heuristics for making decisions for selecting and resolving conflicts. Our work focuses on improving the quality of the solutions for PCP by replacing these heuristics by a machine learning classifier. In this work, several datasets are generated that are used for training classifiers. The performance of the PCP solver when replacing the heuristics with these classifiers is comparable with the performance of the solver when using heuristics, but on average it is slightly worse than the best performing heuristics. After implementing Monte Carlo simulation, we concluded that there was a slight, but statistically significant decrease in average makespan when using the machine learning classifiers for simulating the behaviour of the solver compared to the average makespan when using random heuristics for simulating the behaviour of the solver. However, future research is needed to further improve the performance of the machine learning classifiers, for which we propose a list of improvements based on our observations.

Preface

This work concludes my time as a student in Computer Science at the Delft University of Technology. During my time as student, I had the opportunity to develop many new skills, which helped me to understand the challenges that I faced during this thesis. However, this work would not have been possible without the valuable advice and support of the many people that have helped me during this research.

First of all, Mathijs de Weerdt, thank you for all the help you have offered me in order to help me during my thesis project. I really appreciated the weekly meetings in which we brainstormed about the project, the intermediate results and the next steps in the research. Furthermore, I would like to thank you for the advice you gave me about how to approach a thesis project and the tips for writing the thesis report.

I would like to thank Bob Huisman for offering me the opportunity to work on this project and providing me with an interesting and challenging thesis project. I also want to thank my colleagues at the Dutch Railways and students at the university for helping me with many small problems I encountered during this project. I appreciated the questions I could ask and the feedback I received from everyone.

Last, but certainly not least, I would like to thank the other members of my thesis committee, Neil Yorke-Smith and Sicco Verwer, for their efforts in reading and grading this thesis.

F.P. Verburg
Delft, June 2018

Contents

1 Introduction

NedTrain is a company that is part of the NS Group (Dutch Railways). The task of NedTrain is to achieve a high availability of the rolling stock of NS. The rolling stock consists of locomotives, railroad cars, coaches and wagons of NS. NedTrain is responsible for the maintenance, repair and refurbishment of the complete rolling stock. In order to perform these tasks, NedTrain has depots in locations all over the Netherlands. NedTrain would like to improve the process of maintenance scheduling. In this work we will focus on the technical scheduling of the maintenance process.

1.1. Maintenance scheduling

In order to perform technical maintenance, it is important to keep a detailed schedule of resource and deadline management. At NedTrain, there are several kinds of tasks present. First of all, there are recurring check-ups, which are predefined tasks that can consist of inspections, maintenance or cleaning. These check-ups are performed dependent on the mileage per train or after a certain amount of time after the previous check-up. Furthermore, when a train enters a depot for technical maintenance, additional unexpected repairs and small refurbishments can be performed. With all these different tasks, which all can require different resources, the planning for technical maintenance is complex. Therefor, when scheduling technical maintenance, several challenges arise.

First of all, the arrival and departure times of trains are very strict. Since around 1 million people travel by train every day, it is important to keep the passenger schedule as stable as possible. This results in the fact that trains need to be available at strict time slots, which forces all tasks to be completed before a train is needed in the passenger schedule. Furthermore, the railway network in the Netherlands is very complex and dense. Trains can only leave the depot and enter the railway network in a certain assigned time slot.

The second challenge of maintenance scheduling for NedTrain is the uncertainty that is present at the execution of maintenance projects. Many tasks only occur after inspection, so they are not known beforehand. Furthermore, the duration of the tasks heavily depends on the condition of the train, which is not available beforehand either. This may cause problems, especially in combination with the strict deadlines that trains in the maintenance depot have.

Besides these two challenges, every task within maintenance scheduling projects has both resource and precedence constraints. A task has a resource constraint if that specific task needs additional resources that have a certain amount of capacity. An example of a resource constraint present at NedTrain is the limited number of maintenance tracks. Tasks that require the use of such a maintenance track cannot be scheduled all at the same time. A precedence constraint requires a task to be executed after a certain other task. This can be for example the case when accessibility or safety reasons force a preparation task to be completed before the other tasks starts.

Currently, the planning of maintenance scheduling is mostly done by hand. By providing the schedule makers with better software tools, the process of scheduling can be improved drastically. However, instead of only providing the schedulers with a powerful solver, it is also important to provide them with more insight into the scheduling choices that can be made. So the software need to be both fast and interactive so it can provide both high quality and well understandable schedules.

1.2. Use of heuristics

In the work of Evers[11], it is already showed that the temporal constraints can be solved efficiently with the Simple Temporal Network, which is explained in more detail in Chapter 2. However, when resource constraints are taken into account, this Simple Temporal Network is not sufficient anymore. In this paper, it is also proven that the scheduling problem of NedTrain is a NP-complete problem, thus cannot be solved efficiently.

It is preferred to use an interactive approach in order to create schedules which are understandable by the schedule makers. Therefore, heuristic solvers are used, since when using these heuristics solvers, the decisions that are taken can be seen per step. This makes the process more understandable. Heuristic solvers are in general much faster in solving problems. Heuristic solvers attempt to find solutions of satisfactory quality using for example a rule of thumb, an educated guess, profiling or best practices. These solvers are fast and can operate on incomplete information, which can be used to adjust constraints until a suitable solution is found[31].

Heuristic solvers often do not find optimal solutions for difficult problems. Furthermore, a deep understanding of the problem is needed in order to be able to improve the quality of the solutions. Therefore, a well defined problem of the maintenance scheduling problem has to be used. The scheduling problem of NedTrain, with both temporal and resource constraints, is well suited for the Resource Constrained Project Scheduling Problem (RCPSP). This is a well defined model, first described by Prisker et al.[27], which is studied extensively and extensively explained later in this work.

1.3. Use of machine learning

Until now, when automating the process of maintenance scheduling, heuristics are used for decision making in every step of the solver. The heuristics make choices based on specific rules of thumb, an educated guess, profiling or best practices, based on specific features of the problem. Machine learning methods are suitable for discovering these kind of rules when providing these features. These models can find more and even more complex rules in order to make better decisions. We can explore the possibilities of replacing heuristics by a machine learning approach, in order to improve the decision making in the solver.

People are often prone to making mistakes during analyses or, possibly, when trying to establish relationships between multiple features. However, machine learning can often be successfully applied to these problems, improving the efficiency of systems and the designs of machines. Machine learning is for example already used for this kind of problems by using models for an ”Algorithm Selection Problem”[28]. For these kind of problems, the idea is that a machine learning model tries to predict the behaviour of a perfect oracle that selects the algorithm that preforms best for that specific problem. We therefor expect that applying a machine learning model that replaces the current heuristics can improve solutions for maintenance scheduling problems.

In order to use a machine learning model for replacing the heuristics, we have to determine which features we will use. Furthermore, we need to determine the best machine learning algorithm for this specific problem. Machine learning models could improve the selections that these heuristics make, resulting in an improvement of the performance of the solver for this kind of maintenance scheduling problems.

1.4. Research questions

Later in this work we discuss several algorithms for solving RCPSP instances. The goal of this research is to improve these algorithms by replacing the heuristics used by the algorithms, by a machine learning classifier. In order to achieve this, several research questions need to be answered. The research questions that are used for this research are explained below.

1.4.1. Feature selection

There are a lot of heuristics in order to select and resolve conflict pairs. However, it may be possible to improve these heuristics by looking at the properties of the tasks and/or task pairs. So the first part of this research will be to select the features that will be used to generate the dataset that will be used to train the classifier.

Research Question 1: Which properties of tasks and/or task pairs have an influence on selecting and resolving conflict pairs?

1.4.2. Method selection

After it is clear which properties have influence on the resulting schedule, a machine learning algorithm can be applied to make decisions on how these properties should be used to select and resolve conflict pairs. However, it is also crucial to choose the right machine learning method to classify the instances.

Research Question 2: Which machine learning method(s) is/are the most effective method(s) for solving RCPSP instances?

1.4.3. Improving RCPSP

After the dataset is generated and the machine learning method is chosen, the heuristics can be replaced by a machine learning method. However, it still needs to be validated that the machine learning method produces better solutions than the individual heuristics or if additional steps need to be taken in order to improve the results.

Research Question 3: Can a machine learning algorithm improve the choices of selecting and resolving conflict pairs?

In Chapter 2 we study the current and relevant literature available for the project scheduling problem. Next, in Chapter 3 a brief overview of different machine learning techniques is given. In Chapter 4, our approach for improving RCPSP by using machine learning is explained, together with the results of these experiments. Lastly, in Chapter 5, the conclusions of this work and the future work are described.

2 Background in maintenance scheduling

In this chapter, an overview of the current methods to solve the problem present at NedTrain will be given, in order to find out which methods we can use for improving the quality of the current solutions. We focus mainly on the Resource Constrained Project Scheduling Problem (RCPSP), which is the abstract problem behind the problem present at NedTrain, rather than the specific case of the NedTrain scheduling problem.

First of all, and introduction to constraint programming is given and RCPSP is explained extensively, in order to give a good overview of the several challenges that lie within this problem. Next, Precedence Constraint Posting (PCP) is explained as a method to solve the RCPSP problem instances. Furthermore, several PCP algorithms and related heuristics are explained.

2.1. Constraint Satisfaction Problem

Constraint Programming[29] is an approach to solving combinatorial search problems based on the Constraint Satisfaction Problem (CSP)[4]. Constraints are just relations and a CSP states which relations should hold among the given problem decision variables. CSP approaches have proven to be an effective way to model and solve complex scheduling problems.

The formal definition of the Constraint Satisfaction Problem, for example given by Staats[31], is a tuple $\langle V, D, C\rangle$ where:

- $V = \{v_1, v_2, ..., v_n\}$ is a set of n variables.
- $D = \{D_1, D_2, ..., D_n\}$ is a set of n corresponding domains for the variables in V where $v_i \in D_i$.
- $C = \{c_1, c_2, ..., c_m\}$ is a set of m constraints. Each constraint is a predicate (or function): $c_k : D_1 \times D_2 \times ... \times D_n \rightarrow \{1, 0\}$. In order words, each predicate is a function that takes a set of variable values and returns true if it satisfies the constraint and false if it does not.

Each instance of a CSP can be represented as a constraint graph, $G = (V, E)$. For each variable $v_i \in V$, there is a corresponding node in the graph. For every set of variables connected by a constraint $c_i \in C$, there is a corresponding edge. Temporal constraints in the Resource Constrained Project Scheduling Problem are binary and can be modelled as a graph. Another well known example of a binary CSP is the Simple Temporal Problem, which is explained in more detail in Section **??**.

A CSP can be solved by a procedure that consists of three steps. First, the input problem is checked for consistency. If at least one constraint is violated, the algorithm will be stopped and will fail. If the input problem is also a solution, the algorithm will be stopped and the problem will be returned as a solution. The next step is to select a constraint conflict by using a heuristic. Then, a value is chosen by a value ordering heuristic and this will be added to the problem. The last step is to recursively call the algorithm on the updated problem.

2.2. The Resource Constrained Project Scheduling Problem

As mentioned earlier, a well defined scheduling model is the Resource Constrained Project Scheduling Problem (RCPSP). In this section, the basic definition of RCPSP is given.

The first two basic elements of a RCPSP instance are tasks and resources. Each instance consists of a set of tasks that need to be executed and a set of resources that may be used while executing the task. Each task has a processing duration and each resource has a finite amount of capacity. Both tasks and resources are dependent on practical constraints. There are two general types of constraints, namely temporal constraints and resource constraints. Temporal constraints are constraints that restrict the start time of tasks based on time related properties, such like deadlines and precedence constraints, while resource constraints are associated with the resource properties of the problem instance, for example the amount of resources required by a task and the resource capacity.

The goal of RCPSP is to create a schedule where the runtime is as short as possible. In order words, the goal is to minimise the makespan of the schedule. A schedule is an assignment of starting times of tasks, in which all constraints are satisfied. With a feasible schedule, all tasks can be executed while it is guaranteed that there are enough resources available and all required precedence tasks are completed. The formal definition, as used by for example Staats[31], is as follows:

RCPSP An RCPSP instance is a tuple $P = \langle T, R, C \rangle$ where:

1. T is a set of n tasks. Each task t has a duration $d_i \in \mathbb{R}^{\geq 0}$ and zero or more resource requirements $req(r_k, t_i) \in \mathbb{N}^{\geq 0}$.
2. R is a set of m resources. Each resource r_k has a capacity $cap(r_k) \in \mathbb{N}^{\geq 0}$.
3. C is a set of precedence constraints between tasks ($t_i \in T \rightarrow t_j \in T$).

For each task and resource the following properties must always hold:

1. For each $(t_i \rightarrow t_j) \in C$, task t_i must be completed before task t_j can start.
2. For each resource $r_k \in R$ and concurrent set $T_c \subset T$ the following must always hold $cap(r_k) \geq \sum\limits_{t_i \in T_c} req(r_k, t_i)$, where T_c is a set of tasks that can be executed concurrently.

The associated solution of RCPSP is defined as follows:

Solution of an RCPSP instance Given an RCPSP instance $P = \langle T, R, C \rangle$. Each task $t_i \in T$ is given a start-time variable $s_1, ..., s_n$, where $s_i \in \mathbb{R}^{\geq 0}$. To solve an RCPSP instance one needs to create a schedule, which is a tuple of start-time assignments $S = (s_1, ..., s_n)$ (fixed-time schedule), for which all constraints are satisfied and the make-span of S is minimised.

As explained before, the precedence constraints of an RCPSP instance can be modelled as a graph. This is an acyclic graph, called the project graph[20]. In this project graph, the nodes correspond with the tasks in T and the edges correspond with the precedence constraints $t_i \rightarrow t_j$.

If the project graph has cycles, the RCPSP instance becomes unsolvable, because that would mean that there are at least two tasks that are constrained in two different orderings at the same time. Furthermore, if there is a single tasks which requires more resources than the total resource capacity, the RCPSP instance becomes unsolvable as well. When these two properties are not violated, it is trivial to find an arbitrary schedule without violating the precedence and resource constraints. However, it is also the goal to minimise the makespan of the schedule, which is a NP-hard problem.

The previous definition of RCPSP does not allow for tasks to define an earliest starting time or a deadline. Therefore, we will look at an extension of RCPSP, first presented by Hartmann et al[13] and explained under the name Task RCPSP by Staats[31]. In this extension of RCPSP, release times and deadlines are added to tasks, where the release time is the earliest starting time of the task and the deadline is the latest finishing time of the task. At NedTrain the release time is the time at which a train arrives at the depot and the deadline is when the train leaves the depot. Together with constraints modelled by the standard RCPSP definition, this gives a complete overview of the constraints required by NedTrain. The formal definition of Task RCPSP is as follows:

Task RCPSP A Task RCPSP instance P_t is an extension of an RCPSP instance $P = \langle T, R, C\rangle$, where each task $t_i \in T$ has a release time $rt(t_i) \in \mathbb{N}^{\geq 0}$ and a deadline $dl(t_i) \in \mathbb{N}^{\geq 0}$. It must always hold that a task t_i starts after $rt(t_i)$ and completes before $dl(t_i)$. To solve Task RCPSP one needs to create a schedule the same way as for standard RCPSP.

In RCPSP, the temporal constraints and resource constraints are divided into two layers. The temporal layer contains the temporal constraints and the resource layer contains the resource constraints.

2.2.1. Temporal layer

Temporal constraints define in what range a temporal value has to be, in other words, what are the possible starting times of a task. In Task RCPSP, there are two kind of temporal constraints, deadlines and precedence constraints. A deadline is defined as $s_i < dl(t_i)$, where s_i is the starting time of task t_i, with deadline $dl(t_i)$. Precedence constraints between task t_i and task t_j are defined as $t_i \prec t_j$, where task t_i has to be completed before task t_j is started.

Deadline constraints are relatively simple compared to precedence constraints, since deadline constraints only influence one variable. One precedence constraint is also simple, but a set of precedence constraints can create a complex network of interdependencies. The temporal constraints can be represented as a graph, where the nodes represent the tasks and the edges represent the precedence constraints. The temporal graph of Task RCPSP is only able to define that two tasks are executed before or after each other and not able to define a maximum or minimum distance.

2.2.2. Resource layer

As stated in the definition of RCPSP, resources in Task RCPSP are both renewable and multi-capacity. A renewable resource is a resource that becomes available again after a task is completed. Multi-capacity means that multiple tasks are allowed to use the resource.

One of the challenges in maintenance scheduling is finding resource over-allocation in a timely manner. It is simple to check if a resource is over-allocated, given a set of concurrent tasks T_c, and a resource r_k. If $req(r_k, t_i)$ is the amount of resource r_k that is needed by task $t_i \in T_c$ and $cap(r_k)$ is the capacity of resource r_k, then the set $A_c \subseteq T_c$ is resource consistent with respect to r_k if it holds that $cap(r_k) \geq \sum_{t_i \in A_c} req(r_k, t_i)$. However, there can be many different concurrent sets depending on the complexity of the temporal graph and finding and checking all possible conflict sets can become intractable with only a few tasks. There are multiple general approaches for identifying potential resource conflicts, for example clique-based approaches and profile-based approaches[2]. Clique-based approaches reduce to the maximum clique problem, which is a NP-complete problem. Resource-profile approaches, however, can be calculated in $O(n \times log(n))$ time, by iterating over a list of tasks sorted by earliest starting times, while maintaining an ordered set of active tasks sorted on finishing time with the current resource load[31]. We will therefore focus on resource-profile based approaches, since we aim solve the RCPSP problem instances in a timely manner.

The general idea of profile-based approaches is that it is built as a function of time and resource allocation. If a resource has a certain capacity, time points can be identified where the resource is over-allocated, these time points are called peaks.

2.3. Simple Temporal Problem

Most temporal models use variables to represent events or starting times of tasks. In order to solve these models, an assignment of starting times need to be found. The temporal problem instance in consistent if all temporal constraints are satisfied.

A way to model simple temporal problems is the Simple Temporal Problem (STP)[8]. This model only allows each precedence constraint to define one time interval between tasks. This interval defines the minimum and maximum amount of time that is allowed between tasks. A problem instance can be checked to be consistent in polynomial time[22]. The definition of a STP is as follows:

STP An STP instance is a tuple $P = \langle T, C\rangle$ where $T = \{\tau_1, ..., \tau_n\}$ is a set of n time point variables and C is a set of binary constraints. For each constraint $c_{ij} \in C$ with $c_{ij} = (a_{ij}, b_{ij})$ the following must hold: $-\tau_i + \tau_j \leq b_{ij}$ and $\tau_i - \tau_j \leq -a_{ij}$. A solution S for P is an assignment of values for the time point variables in T, such that all constraints hold.

The time points $\tau_1, ..., \tau_n$ have values that can be restricted using the given intervals, which allows for many temporal restrictions to be applied to the starting times of a task.

The STP can also be represented as a constraint graph. Such a graph is called a Simple Temporal Network (STN). In such a graph, the nodes represent the time points and the edges are labeled as constraints.

An STP can be solved in linear time, because each constraint can be represented as the following two inequalities: $-\tau_i + \tau_j \leq b_{ij}$ and $\tau_i - \tau_j \leq -a_{ij}$, where τ are continuous variables in the linear model. This model can be solved using linear solvers.

We could express the Task RCPSP problem in an STP, but as stated in the section before, Task RCPSP has enough expressive power to model the temporal constraints present at the scheduling problem at NedTrain. We therefore not need to model it using STP, but STP can still be useful for problems later in this work.

2.4. Interval schedules

The schedules that are defined in the solutions of RCPSP and STP both are fixed-time schedules. A fixed-time schedule is a mapping of tasks to starting times, where every task only has one time point at which it can start. This has the advantage that all temporal or resource information can be ignored as long as the tasks do not suffer any form of delay. But when there are even small delays, the schedule will not be useful anymore. The schedule cannot continue because it is not known how the delay will impact the other tasks. When working with dynamic schedules, fixed time schedules do not provide the flexibility that is needed. In order to solve this, the scheduling process can be split in two steps. First, the dynamic scheduling problem is partly solved, which still results in a dynamic schedule. In the second step , this dynamic schedule is converted into a fixed-time schedule. This conversion is a fast process, and therefore can be executed again when a delay has occurred. One partial solution representation is an interval schedule[10].

An interval schedule does not assign a single time value to a task, but instead assigns an interval of time points. The starting time of the task can be anything in this interval, without being worried about breaking any constraint. An interval schedule is defined as follows:

Interval schedule Given an STP instance $P = \langle T, C \rangle$. An interval schedule S is a set of intervals $S = \{I_1, ..., I_n\}$ for each time-point variable in T, with $I_i = [a_i, b_i]$ where $a_i \leq b_i$ and for all $c_{ij} \in C$ it holds that $b_i \leq a_j$. S is a valid interval schedule for P iff for all values between a_i, b_i is a possible assignment of the variable $\tau_i \in T$ without violating the temporal constraints in P.

Interval schedules have more flexibility than fixed-time schedules, but they do not retain the original precedence relations. This means that a schedule can still be feasible when one of the tasks is started outside it's interval. Partial Order Schedules[23] on the other hand does maintain structural data.

2.5. Partial Order Schedules

A Partial Order Schedule (POS) is introduced in order to provide more robustness and flexibility in a schedule. The idea of a POS is that it is only a partially ordered set of tasks. It differs from an STN since there are no fixed starting times, but it only uses simple precedence constraints. For every task it is defined which tasks has to be completed in order for the task to start. Delays can be dealt with very easily and quick in this way. A POS can also be seen as a graph that defines a set of feasible fixed-time schedules. During the execution of the project the set of solutions is reduced as tasks are completed and delays are spread to future tasks. The formal definition Policella[24] has given is as follows:

POS A POS, $G = \langle N, E \rangle$ is a directed graph, where N is a set of nodes representing temporal variables $\tau_1, ..., \tau_n$ and E is a set of edges representing binary precedence constraints between two temporal variables $(\tau_i \rightarrow \tau_j)$. A schedule S of G is an assignment of values to N for which all precedence constraints hold. G is a valid POS of a given RCPSP instance P iff all possible schedules of G also satisfies all constraints in P.

A POS is more robust since external changes can be include in the underlaying temporal network. The POS only contains resource feasible schedules, so the resource constraints can be ignored during the spreading to future tasks. At some point new external changes can no longer be spread, at which point a new schedule needs to be calculated from scratch. Solutions can be found using Precedence Constraint Posting, which is explained in the next section.

2.6. Precedence Constraint Posting

The difficult part of solving an RCPSP instance is the part with resource constraints. Temporal constraints are much easier to satisfy. Having only temporal constraints opposed to both temporal and resource constraints restricts the search space from NP to P. Precedence Constraint Posting (PCP) is a technique that simplifies resource constrained problems to problems containing only temporal constraints. This has as advantage that the resulting problem, with only temporal constraints, is relatively easy to solve. However, the step of converting the resource constraints to temporal constraints is still a complex problem.

The way PCP converts a problem with both resource and temporal constraints to a problem with only temporal constraints is by continually adding extra precedence constraints to the temporal layer, until all possible temporal constraints are also consistent with the resource constraints. This is done by adding a precedence constraint between two tasks that have a resource conflict. By adding the precedence constraint, the two tasks cannot be executed together, so they do not use the same resource.

A basic greedy PCP solver has a problem with both resource and temporal constraints as input and outputs a schedule. The greedy algorithm is an iterative algorithm. In every iteration, the algorithm finds all resource conflicts, select one conflict out of this set and after that solves the conflict by adding a precedence constraint. This is done until no more resource conflicts can be found, or until the problem becomes unsolvable. This algorithm has two points at which it has to make a search decision, namely selecting a conflict to solve and choosing a precedence constraint to solve this conflict.

The first step of PCP is to find resource conflicts. As stated before, we will focus on using a resource profile to achieve this. A resource set is usually a set of tasks competing for the same resource, also called a conflict set. A conflict set can be resolved by simply selecting two random tasks in the resource set and adding a precedence constraint between these two tasks. But selecting tasks randomly is not the most effective way of resolving conflict sets. The randomly posted constraint might have very little impact in solving the set.

When selecting a conflict or resolving a conflict, usually a least-commitment strategy is applied. A least-commitment strategy means that the choices are made so that the amount of restrictions added to a set of possible solutions is minimised. This allows the solver to avoid dead-ends and increases its chances of finding a solution.

There are two common approaches for conflict resolution, a Minimal Critical Set (MCS) approach and pair-wise conflict resolution. A Minimal Critical Set is a set of tasks where if all tasks are executed together, a resource constraint is violated, but any strict subset of this set of tasks does not exceed the resource capacity. It is proposed to use MCS's for conflict resolution by only solving conflicts that are also a MCS[18][3], in order to restrict the number of precedence constraints posted. However, finding the most critical MCS is a hard problem for larger scheduling instances, since there are an exponential number of MC's to the number of tasks in a resource peak.

Pair-wise conflict resolution is a much simpler approach to resolve conflicts. Given a conflict set, a set of conflict pairs will be created. All the tasks in the conflict set compete for the same resource. Resolving only some of the conflict pairs will resolve the conflict set. Conflict pairs will be resolved by posting a precedence constraint between the two tasks in the conflict pair. As stated by Staats[31], there are four cases of conflict pairs, containing task t_1 and task t_2, which can be used to prune the search space of conflict pairs:

Case 1: t_1 cannot be executed before t_2 and t_2 cannot be executed before t_1

Case 2: t_1 cannot be executed before t_2

Case 3: t_2 cannot be executed before t_1

Case 4: t_1 can be executed before t_2 and t_2 can be executed before t_1

By classifying all the conflicts early pruning can be done by removing all conflicts of Case 1 because any extra posted constraint will create an inconsistent temporal network. Cases 2 and 3 are trivial to resolve because only one ordering is possible. Case 4 conflicts are open to any ordering and often requires more calculations to resolve.

Finding all pair-wise conflicts is, opposed to using MCS's, a much simpler problem, since the number of conflict pairs is bounded by $O(n^2)$. Therefore, pair-wise conflict resolution is much faster than using MCS's, but it can post more precedence constraints than using MCS's. Using the right heuristics influences the effectiveness of the posting of precedence constraints. An overview of possible heuristics is given in Section 2.9

2.7. Flexibility

Flexibility can help to make better decisions in the heuristics to eventually make better schedules. For NedTrain, a flexible schedule should give management reliable information and give maintenance teams autonomy while minimizing the duration of the maintenance project. Flexibility can be defined as the number of possible concrete schedules represented in a flexible schedule, however it is not generally possible to count the exact number of possible schedules, so most flexibility metrics are estimations of flexibility. There are multiple metrics to measure the flexibility of a POS. One of these metrics is $flex_I$[31]. Each task $t_i \in T$, where T is a set of tasks in the scheduling problem instance, is assigned an interval $[a_i, b_i]$. Each task is free to start within this interval without taking the behaviour of other tasks into account. We can define that amount of independent slack or flexibility as follows: $flex_I = \sum\limits_{t_i \in T} (b_i - a_i)$

2.8. PCP algorithms

There are different existing algorithms to implement PCP. Two of the existing algorithms are the Conflict Free Solution Algorithm, and Solve and Robustify.

2.8.1. Conflict Free Solution Algorithm

The Conflict Free Solution Algorithm(CFSA) takes a bounding approach to solving RCPSP problems. The idea behind CFSA is to keep adding precedence constraints until there are no more resource constraints. In other words, all possible solutions allowed by the temporal constraints must not exceed the resource capacity. A resource profile is used to find pairs of tasks that cause resource conflicts. However, to be able to create a resource profile that can find the exact number of conflict pairs, it needs to know the exact starting time of tasks, but CFSA does not maintain the exact starting time of tasks, it only maintains temporal bounds in the form of an STN. With a temporal network it is impossible to construct an exact resource profile because of possible interactions between tasks. Instead CFSA maintains two profiles: the worst case profile (upper-bound) and a best case profile (lower-bound). Unfortunately, this approach has the tendency to post too many precedence constraints. The resulting POS has too many constraints and therefore represents only a small set of possible schedules.

2.8.2. Solve and Robustify

A second approach for solving RCPSP problems is Solve and Robustify, which is proposed by Policella et al.[25]. This approach is based on the idea to separate the phase of obtaining a problem solution and the phase of adding flexibility to the resulting schedule, which results in a Partial Order Schedule. So, Solve and Robustify first tries to find a single fixed-time solution and in the second phase, the schedule is expanded into a POS, as is illustrated in Figure 2.1. In this way, any state of the art solver that has been designed to obtain optimal solutions for a certain objective can be used to retrieve an schedule. After that, in the second step, additional flexibility can be added, without altering the makespan of the schedule.

For the first step, the Earliest Start Time Algorithm (ESTA) can be used and for the second step, the Chaining Algorithm can be used. Both algorithms are explained in more detail in the following sections.

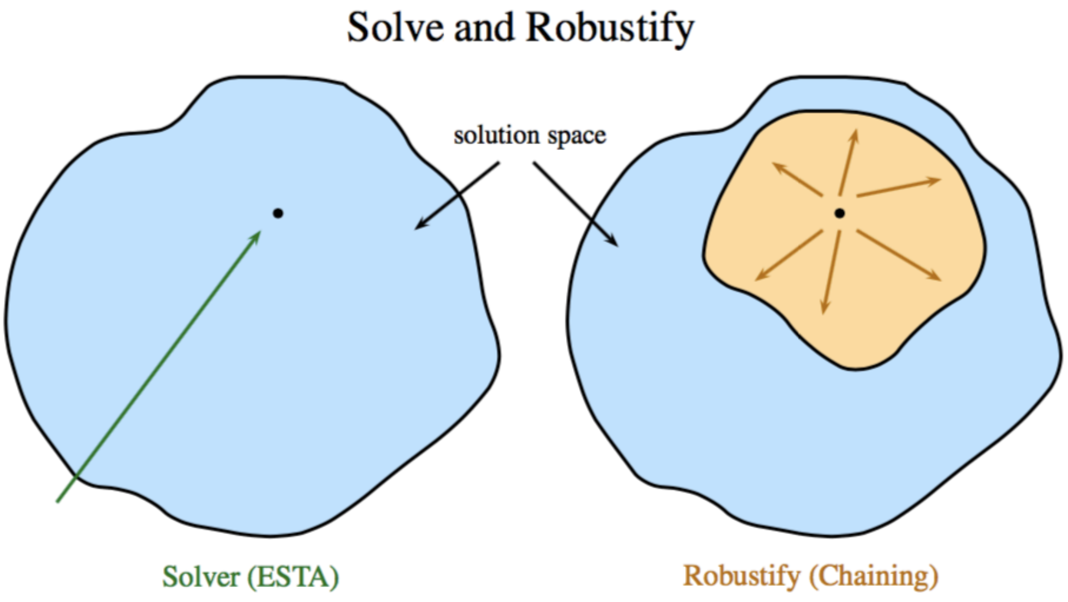

Figure 2.1: Graphical representation of Solve and Robustify[31].

Earliest Start Time Algorithm
The Earliest Start Time Algorithm(ESTA)[2] is an algorithm that was introduced by Cesta et al as an algorithm to solve the Multiple Capacitated Metric Scheduling Problem. This problem is very similar to the Resource Constrained Project Scheduling Problem and in later work it is showed that this algorithm can be used to solve RCPSP as well.

For the Earliest Start Time Algorithm, only the earliest start times are allowed by the temporal network. This results in only one fixed time schedule. This has as result that, contrarily to CFSA, an exact resource profile can be calculated. Therefor, it will be possible to select much more relevant precedence constraints. With an exact resource profile, ESTA can make better predictions for selecting resource peaks and constraints in order to minimise the resulting makespan.

For CFSA, every task has an interval in which the task can be executed. A consistent temporal solution for such a schedule can be retrieved by using only the earliest start times, which results in an Earliest Start Time Schedule(ESTS). It is also possible to use the Latest Start Time Schedule, but since we are interested in minimising the makespan, using the ESTS is more useful. The ESTS is temporal consistent, but it does not have to be resource consistent as well. However, it is a good start schedule for using a resource peak levelling algorithm for Precedence Constraint Posting.

Algorithm 1 shows the general process of ESTA. The first step is to calculate the Earliest Start Time Schedule. After that, the resource conflicts are calculated. If there aren't any resource conflicts, the schedule is also resource consistent, so the schedule can be returned. If the conflict set is unsolvable, for example when a conflict is found that cannot be put into sequence due to temporal restrictions, the schedule is considered unsolvable an a failure is returned. If this both doesn't happen, first a conflict is selected out of all conflicts in the resource conflicts and after that, a precedence constraint is selected, so which task has to be executed first out of the two tasks in the conflict. Finally the precedence constraint is posted to resolve it. This algorithm will start over until a resource consistent ESTS is found. It is important to choose the right conflicts and constraints, since the quality of the solution depends on the selection of these. There are several heuristics that make these kind of decisions, both for conflict selection as conflict resolution. An overview of such heuristics is given in Section 2.9.

Chaining Algorithm
The second step of Solve and Robustify is the robustify step. As mentioned earlier, this is done in order to add flexibility to the solution, by transforming the fixed time solution that is the result of the ESTA algorithm, to a POS. The chaining algorithm sees each resource as a distinct unit that flows through a sequence of tasks, called a chain. The algorithm creates such a flow by posting precedence

Algorithm 1 ESTA

```
1: **Input:** A problem $P$
2: **Output:** A schedule $S$
3: while $true$ do
4:    $S \leftarrow$ CalculateESTS($P$)
5:    $conflictSet \leftarrow$ FindResourceConflicts($S$)
6:    if $conflictSet = \emptyset$ then
7:       return $S$
8:    end if
9:    if Unsolvable($conflictSet$) then
10:      return $failure$
11:   end if
12:   $conflict \leftarrow$ SelectConflict($conflictSet$)
13:   $constraint \leftarrow$ SelectPrecedenceConstraint($conflict$)
14:   PostPrecedenceConstraint($P$, $constraint$)
15: end while
```

constraints. Two tasks that compete for the same resource unit will be added to the same chain and will therefor no longer be able to conflict with each other. Since there are as many chains as the number of resources, the resource capacities will never be exceeded.

A basic algorithm that can be used for chaining is introduced by Policella et al[24] and is showed in Algorithm 2. The Chaining Algorithm first sorts the tasks based on their start time, since using this approach it is guaranteed that each task that is added will not violate any temporal constraint. Adding tasks ordered by start time also ensures that there are enough free chains at the given start time of the task, which ensures that the makespan is not increased during the chaining process. After the sorting, for each resource a separate chain is created, containing the first task t_0. For every task t_i and for every chain, a precedence constraint is posted between t_i and the last task in the chain. Once all the tasks have been assigned to chains and all the needed precedence constraints have been posted, the temporal network will be a valid POS.

Algorithm 2 Chaining Algorithm

```
1: Input: A problem $P$ and one of its fixed-time schedules $S = s_1, ..., s_n$
2: Output: A POS in chain-form
3: $POS \leftarrow P$
4: Sort $S$ by start time of tasks in $S$
5: Initialise all chains with the start task $t_0$
6: for all resources $r_k$ do
7:    for all tasks $t_i$ do
8:       for 1 to $req(r_k, t_i)$ do
9:          $chain \leftarrow$ SelectChain($t_i, r_k$)
10:         $t_l \leftarrow$ last($chain$)
11:         $POS \leftarrow POS \cup \{t_l \prec t_i\}$
12:         $last(chain) \leftarrow t_i$
13:      end for
14:   end for
15: end for
16: return $POS$
```

2.8.3. Conclusion

In this section, two different Precedence Constraint Posting algorithms are explained, CFSA and Solve and Robustify. CFSA is a straightforward implementation of PCP, which reduces the RCPSP instance directly into a POS. However, since the input is not an fixed time schedule, it is not possible to calculate an exact resource profile. Therefor, the conflict pairs cannot be retrieved precisely. Solve and Robustify overcomes this problem by first converting the problem to a fixed time schedule, then solving the problem and after that add flexibility to the schedule.

Both methods can be used to solve an RCPSP instance. However, we want to have as much information as possible about the conflicts. This is desirable since decisions will be made based on the available information of the conflicts. Therefor, in this work we will be focusing on the Solve and Robustify algorithm for improving the solutions of RCPSP instances.

2.9. Heuristics for conflict selection and conflict resolution

It is important to select the right conflicts and constraints, since the resulting schedule depends on these selections. Heuristics are used in order to choose the conflicts and constraints. In this section, the heuristics for conflict selection and resolution will be described.

2.9.1. Heuristics for conflict selection

The conflict selection heuristics that are available are Min-Slack and ω_{res}. These heuristics are defined as follows.

Min-Slack

Min-Slack is a pair-wise resource conflict heuristic that is proposed by Smith[30]. This heuristic uses the amount of slack between two tasks in a conflict pair. The slack is used to rank the conflict pairs. Slack is the amount of time between the tasks. Slack is calculated using the earliest starting time (est), latest finishing time (lft) and duration (d) of tasks. The formula for calculating the slack between two tasks is:

$$slack(t_i \prec t_j) = lft(t_j) - est(t_i) - (d_i + d_j)$$

The heuristic uses this slack to rank the conflict pairs. The pair with the least amount of slack will be considered as the most critical, which results in the following formula:

$$\min_{(t_i,t_j)}\{\min(slack(t_i \prec t_j), slack(t_j \prec t_i))\}$$

The conflict with the least amount of slack is chosen first, because there is a higher probability that this conflict pair will contribute to an unsolvable resource peak later on in the PCP process.

Flexibility metric

Another pair-wise resource conflict heuristic is ω_{res}, which is proposed by Cesta[2]. This is a flexibility metric that also uses a temporal distance between two tasks. Just as with Min-Slack, the pair with the least amount of flexibility will be chosen to be resolved first. The formula of ω_{res} makes use of a distance graph, which uses start times and end times. Therefor, each task is split into a start time, $\tau_i = s_i$, and a end time, $\varepsilon_i = s_i + d_i$. Furthermore, if all conflict pairs in the peak are of case 4, which means that for every pair task 1 can be executed before task 2 and task 2 can be executed before task 1, ω_{res} uses an extra metric S in order to take both orderings of the tasks into account. Since both tasks can be executed before the other task, there are two possible orderings of which one can might be critical but the other not. The resulting formula is:

$$\omega_{res}(t_i,t_j) = \begin{cases} \frac{\min\{d(\varepsilon_i,\tau_j),d(\varepsilon_j,\tau_i)\}}{\sqrt{S}} & \text{if all conflicts are case 4} \\ \min\{d(\varepsilon_i,\tau_j), d(\varepsilon_j,\tau_i)\} & \text{else} \end{cases}$$

Where $S = \frac{\min\{d(\varepsilon_i,\tau_j),d(\varepsilon_j,\tau_i)\}}{\max\{d(\varepsilon_i,\tau_j),d(\varepsilon_j,\tau_i)\}}$.

2.9.2. Heuristics for conflict resolution

There are also several heuristics for conflict resolution, so which task must be executed first. The heuristics that are considered are max-slack, a makespan heuristic and the flexibility metrics $flex_I$, $RM1$ and RB. These heuristics are used to determine which of the two options remove the least solutions from the solution space. Max-slack uses the slack between the two tasks in the conflict, but the flexibility and makespan metrics use the flexibility, or makespan, of the problem after posting the constraint of task 1 $\prec$ task 2, versus the flexibility of the problem after posting task 2 $\prec$ task 1. The different heuristics are defined as follows.

Max-slack
This heuristic uses the amount of slack between the two tasks to determine which task must be executed first. This is based on the assumption that having more slack means that there is a larger amount of solutions. So removing the least amount of slack will remove the least amount of solution space. This results in the following constraint selection formula.

$$\text{selected constraint} = \begin{cases} \text{task 1} \prec \text{task 2} & \text{if } slack(\text{task 1, task 2}) > slack(\text{task 2, task 1}) \\ \text{task 2} \prec \text{task 1} & \text{else} \end{cases}$$

RM1
As stated earlier, the flexibility metrics calculate the flexibility of an entire problem, rather dan just two tasks. $RM1$ is a slack based metric proposed by Chtourou[7]. It sums up the slack of all tasks. The resulting formula for calculating $RM1$ of problem T is as follows.

$$RM1(T) = \sum_{t_i \in T} lst(t_i) - est(t_i)$$

RB
Another metric, which is proposed by Cesta[2], is RB. RB is the ratio between the average distances in time between tasks and the scheduling horizon. The scheduling horizon is the sum of the duration of all tasks in the problem. RB also uses a distance graph, and $d(t_i, t_j)$ is the temporal distance between the two tasks t_i and t_j. Given the horizon as H, the formula of RB of problem T is as follows.

$$RB(T) = 100 \times \sum_{t_i, t_j \in T, i \neq j} \frac{|d(t_i, t_j) + d(t_j, t_i)|}{H \times (n(n-1))}$$

$\mathbf{flex_I}$
Another metric is $flex_I$, which is proposed by Wilson[32]. This metric uses an interval schedule, which is used to calculate the amount of freedom the interval schedule has. This is also called the amount of independent slack. In an interval schedule, each task t_i has an interval $[a_i, b_i]$ in which the task can be executed without having to worry about exceeding any constraints. The formula for calculating $flex_I$ is as follows.

$$flex_I(T) = \sum_{t_i \in T} (b_i - a_i)$$

Makespan
The last metric we will use uses the makespan of the two resulting problems. The makespan of a problem is calculates as follows, where $eft(t_i)$ is the earliest finishing time of task t_i.

$$makespan(T) = \max_{t_i \in T}(eft(t_i))$$

3 Background in machine learning

As explained in the introduction, we would like to improve the constraint selection and conflict resolution by replacing the heuristics by a machine learning model. In this chapter, an introduction to machine learning is given, in order to be able to support the selection of the features and machine learning methods we use. We will use these methods later in this work to improve the current solver for the RCPSP problem.

We focus on machine learning classifiers, since the goal of the heuristics replacement is to classify each conflict and constraint in order to makes choices for selecting and resolving conflict pairs.

First of all, we give a general introduction to machine learning, after which we will explain several classifiers that we can use in more detail. At the end of this chapter, we select some machine learning models that could be effective for classifying the instances in our problem.

3.1. Introduction to machine learning

Machine learning is a field that often uses statistical models to give computers the ability to learn. This learning is done based on data that is provided, but without requiring the computer to be explicitly programmed to process the data. Each data example in a dataset is called an instance. Each instance in the dataset used by machine learning algorithms is represented using the same set of features. The features may be continuous, categorical or binary. In this work, we will work with instances with known labels. A label indicates in which corresponding class the instance belongs. These labels are the corresponding correct outputs. This kind of machine learning is called called supervised learning[16].

Inductive machine learning is the process of creating a classifier that can be used to generalise from new instances. The process of applying supervised machine learning to a real-world problem consist of multiple steps. When having a problem, the first important step is to identify the data and select the features that have influence on the comprehensibility of the classifier. It can also be possible that some extra data preprocessing is necessary for the classifier in order to be able to process the data. When the training set is generated, the appropriate machine learning algorithm with the right parameters needs to be selected. The model can be trained based on this training set, and is evaluated based on a different test set. Based on the results of this evaluation, some of the steps can be executed again. A graphical representation of this process is given in Figure 3.1.

The choice of which specific learning algorithm we should use is a critical step. The performance of the algorithm depends on the type of data and the type of features that is available. Therefor, we look at the properties of different algorithms and compare them. The different algorithms also need to be evaluated. In the next section, different evaluation methods are considered.

3.2. Evaluation of methods

Estimating the accuracy of a classifier is important not only to predict its future prediction accuracy, but also for selecting the correct machine learning classifier from a given set. It is important to avoid overfitting of the data. Any learned hypothesis h, is said to overfit training data if another hypothesis h' exists that has a larger error than h when tested on the training data, but a smaller error than h

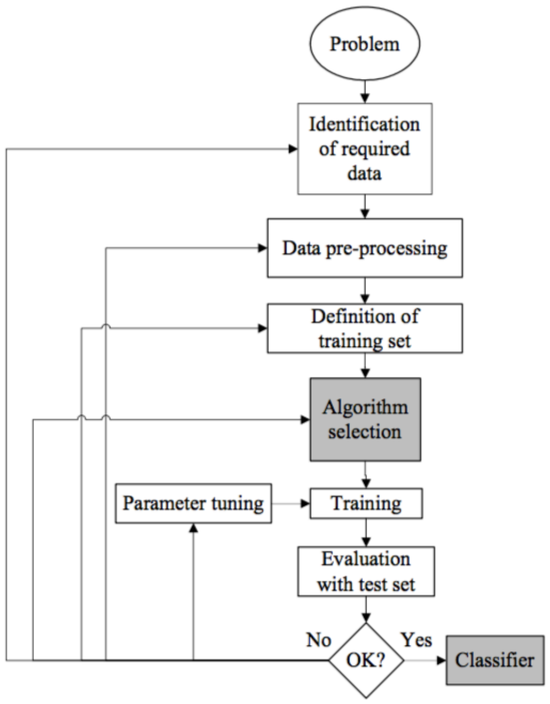

Figure 3.1: The process of supervised ML

when tested on the entire dataset. In order to avoid overfitting, k-fold cross-validation is used for the evaluation of the machine learning models that will be used. In k-fold cross-validation, the dataset D is randomly split in into k mutually exclusive subsets, which are the folds, $D_1, D_2, ..., D_k$ of approximately equal size. Then, the training and testing is done k times, each time $t \in \{1, 2, ..., k\}$. It is trained on $D \backslash D_t$ and tested on D_t [14].

Using this cross-validation, the confusion matrix and ROC curve are calculated. The confusion matrix is a matrix that visualises the performance of a machine learning algorithm. Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class. In this way, the amount of false negatives and false positives can be easily shown. A false negative is an instance that is classified as class 0, while it is actually member of class 1. A false positive is an instance that is classified as class 1, while it is actually a member of class 0.

The receiver operating characteristic (ROC) curve is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. With a classifier, the output can be a probability, a value between 0 and 1. Ideally you want to say that everything larger 0.5 is member of one class and everything less than 0.5 is member of the other class. This value is called the threshold. However, depending on the specific problem, you can be more concerned with false negatives or with false positives. The ROC curve is created by plotting the true positive rate against the false positive rate at various threshold settings, to take into account the sensitivity of the model in general. We also use the area under the curve (AUC) of the ROC curve as evaluation metric. The machine learning community most often uses the ROC AUC statistic for model comparison [12]. A reliable and valid AUC estimate can be interpreted as the probability that the classifier will assign a higher score to a randomly chosen positive example than to a randomly chosen negative example.

3.3. Parameter tuning

Besides choosing the best machine learning algorithm to achieve the best results, choosing the best parameters for the method is also important. Every machine learning method has several parameters that influence the performance of the model. Each parameter can be seen as a dimension on a graph with the values of a given parameter as a point along the axis. Therefor, n parameters will result in an n-dimensional hypercube of possible configurations of the algorithm. Tuning these parameters can therefor improve the performance of the model.

The goal of parameter tuning is to find a point in this n-dimensional hypercube in which the algorithm performs best. This can be done by using automated methods that uses a grid on the space of possibilities. However, overfitting of the model must be avoided. Therefor, it is important to use cross-validation in this step as well.

3.4. Overview of different machine learning algorithms

In order to choose the right machine learning algorithm to use in this research, we give an overview of different machine learning techniques in this chapter. Kotsiantis et al[16] have given an exhaustive overview of different techniques, which will be explained in this section.

3.4.1. Logic based algorithms

In this section we will concentrate on two groups of logical (symbolic) learning methods: decision trees and rule-based classifiers.

Decision trees

Decision trees are trees that classify instances by sorting them based on feature values. Each node in a decision tree represents a feature in an instance to be classified, and each branch represents a range of values that the node can assume. In the leaves, the class for an instance with the corresponding features is defined. Instances are classified starting at the root node and sorted based on their feature values. The problem of constructing optimal decision trees is an NP-complete problem and thus theoreticians have searched for efficient heuristics for constructing near-optimal decision trees. The feature that best divides the training data would be the root node of the tree. There are numerous methods for finding the feature that best divides the training data. The same procedure is then repeated on each partition of the divided data, creating sub-trees until the training data is divided into subsets of the same class.

The most straightforward way of tackling overfitting in decision trees is to pre-prune the decision tree by not allowing it to grow to its full size. Establishing a non-trivial termination criterion such as a threshold test for the feature quality metric can do that. Decision tree classifiers usually employ post-pruning techniques that evaluate the performance of decision trees, as they are pruned by using a validation set.

One of the most useful characteristics of decision trees is their comprehensibility. People can easily understand why a decision tree classifies an instance as belonging to a specific class. Since a decision tree constitutes a hierarchy of tests, an unknown feature value during classification is usually dealt with by passing the example down all branches of the node where the unknown feature value was detected, and each branch outputs a class distribution. The output is a combination of the different class distributions that sum to 1. The assumption made in the decision trees is that instances belonging to different classes have different values in at least one of their features. Decision trees tend to perform better when dealing with discrete/categorical features.

Learning set of rules

Decision trees can be translated into a set of rules by creating a separate rule for each path from the root to a leaf in the tree. However, rules can also be directly induced from training data using a variety of rule-based algorithms. Classification rules represent each class by disjunctive normal form (DNF). The goal is to construct the smallest rule-set that is consistent with the training data. A large number of learned rules is usually a sign that the learning algorithm is attempting to ”remember” the training set, instead of discovering the structure that the data governs.

The most useful characteristic of rule- based classifiers is their comprehensibility. In addition, even though some rule-based classifiers can deal with numerical features, some experts propose these features should be discretised before induction, so as to reduce training time and increase classification accuracy.

3.4.2. Perceptron based algorithms

Other well-known algorithms are based on the notion of perceptron. We can divide perceptron based algorithms into single layered and multilayered perceptrons.

Single layered perceptrons

A single layered perceptron can be briefly described as follows: If x_1 through x_n are input feature values and w_1 through w_n are the corresponding weights which are assigned to the features, typically real numbers in the interval [-1, 1], then the perceptron computes the sum of weighted inputs: $\sum_i x_i w_i$ and output goes through an adjustable threshold: if the sum is above threshold, the output is 1, otherwise the output is 0.

The most common way that the perceptron algorithm is used for learning from a batch of training instances is to run the algorithm repeatedly through the training set until it finds a weight vector and a threshold which is correct on all of the training set. This prediction rule is then used for predicting the labels on the test set.

Generally, all perceptron-like linear algorithms are anytime online algorithms that can produce a useful answer regardless of how long they run. The longer they run, the better the result they produce. Finally, perceptron-like methods are binary, and therefore in the case of multi-class problem one must reduce the problem to a set of multiple binary classification problems.

Multilayered perceptrons

Perceptrons can only classify linearly separable sets of instances. If a straight line or plane can be drawn to separate the input instances into their correct categories, input instances are linearly separable and training algorithm will find the right perceptron in order to find the solution. If the instances are not linearly separable learning will never reach a point where every instance is classified properly. Multilayered Perceptrons (Artificial Neural Networks (ANNs)) have been created to try to solve this problem.

A multi-layer neural network consists of large number of units (neurons) joined together in a pattern of connections. Units in a net are usually segregated into three classes: input units, which receive information to be processed; output units, where the results of the processing are found; and units in between known as hidden units. Feed-forward ANNs allow signals to travel one way only, from input to output.

First, the network is trained on a set of labeled data to determine input-output mapping. The weights of the connections between neurons are then fixed. These fixed weights are then used to determine the classifications of a new set of data.

During classification the signal at the input units propagates all the way through the net to determine the activation values at all the output units. Each input unit has an activation value that represents some feature external to the net. Then, every input unit sends its activation value to each of the hidden units to which it is connected. Each of these hidden units calculates its own activation value and this signal is then passed on to output units. The activation value for each receiving unit is calculated according to a simple activation function. The function sums together the contributions of all sending units, where the contribution of a unit is defined as the weight of the connection between the sending and receiving units multiplied by the sending unit's activation value. This sum is usually then further modified, for example, by adjusting the activation sum to a value between 0 and 1 and/or by setting the activation value to zero unless a threshold level for that sum is reached.

The multiple layers and the nonlinear activation function distinguishes the multilayered perceptron from a single layered perceptron, so it can also identify non binary problems.

ANNs have been successfully applied to many real-world problems but still, their most striking disadvantage is their lack of ability to reason about their output in a way that can be effectively communicated.

3.4.3. Statistic learning algorithms

Conversely to ANNs, statistical approaches are characterised by having an explicit underlying probability model, which provides a probability that an instance belongs in each class, rather than simply a classification. Bayesian networks are the most well known representative of statistical learning algorithms.

Naive bayesian network

Naive Bayesian networks (NB) are very simple Bayesian networks which can be used for simple classifier construction. It is a model that assigns class labels to problem instances. These instances are represented as vectors of feature values and the class labels is an finite set. Naive Byes classifiers assume that the value of each feature is independent from the value of any other feature. Many applications use the method of maximum likelihood for parameter estimation. The maximum likelihood estimator attempts to find the parameter values that maximise the likelihood function, given the observations.

The major advantage of the naive Bayes classifier is its short computational time for training. In addition, since the model has the form of a product, it can be converted into a sum through the use of logarithms - with significant consequent computational advantages. If a feature is numerical, the usual procedure is to discretise it during data pre-processing.

Instance-based algorithms

Another category under the header of statistical methods is Instance-based learning. Instance-based learning algorithms are lazy learning algorithms, as they delay the induction or generalisation process until classification is performed. Lazy-learning algorithms require less computation time during the training phase than eager-learning algorithms (such as decision trees, neural and Bayesian networks) but more computation time during the classification process. One of the most straightforward instance-based learning algorithms is the nearest neighbour algorithm.

The higher stability of nearest neighbour classifiers distinguishes them from decision trees and some kinds of neural networks. A learning method is termed ”unstable” if small changes in the training-test set split can result in large changes in the resulting classifier.

The major disadvantage of instance-based classifiers is their large computational time for classification. A key issue in many applications is to determine which of the available input features should be used in modelling via feature selection, because it could improve the classification accuracy and scale down the required classification time. Furthermore, choosing a more suitable distance metric for the specific dataset can improve the accuracy of instance-based classifiers.

3.4.4. Support Vector Machines

Using Support Vector Machines (SVMs) is another supervised machine learning technique. SVMs revolve around the notion of a ”margin”, either side of a hyperplane that separates two data classes. Maximising the margin and thereby creating the largest possible distance between the separating hyperplane and the instances on either side of it has been proven to reduce an upper bound on the expected generalisation error.

In the case of linearly separable data, once the optimum separating hyperplane is found, data points that lie on its margin are known as support vector points and the solution is represented as a linear combination of only these points. Other data points are ignored. Therefore, the model complexity of an SVM is unaffected by the number of features encountered in the training data (the number of support vectors selected by the SVM learning algorithm is usually small). For this reason, SVMs are well suited to deal with learning tasks where the number of features is large with respect to the number of training instances.

However, when non-separable data is involved, SVM’s can also be used. One solution to the inseparable data problem is to map the data onto a higher dimensional space. The separating hyperplane can then be defined on this higher dimensional space. With an appropriately chosen transformed feature space of sufficient dimensionality, any consistent training set can be made separable.

The training optimisation problem of the SVM necessarily reaches a global minimum, and avoids ending in a local minimum, which may happen in other search algorithms such as neural networks. However, the SVM methods are binary, thus in the case of multi-class problem one must reduce the problem to a set of multiple binary classification problems. Discrete data presents another problem, although with suitable rescaling good results can be obtained.

3.4.5. Ensemble learning

The ensemble learning methods are methods that generate many classifiers and aggregate their results[19]. Two well known methods are bagging and boosting, which are explained in more depth in the following sections.

Bagging

In bagging, successive prediction methods do not depend on earlier prediction methods. Each prediction method is independently constructed using a bootstrap sample of the data set. In the end, a simple majority vote is taken for prediction. An example of such a bagging method is a Random Forest classifier[19].

In random forest, each tree is constructed using a different bootstrap sample of the dataset. Furthermore, in each node of the tree, a subset of the available predictors is taken. The best among this subset of predictors is chosen at that specific node. This strategy results in a method that is robust against overfitting. After this step, there are multiple trees for the classification. The final prediction is done using an aggregation of these multiple trees, for example using majority vote.

Boosting

In boosting, successive prediction methods give extra weight to points incorrectly predicted by earlier predictors. In the end, a weighted vote is calculated to use as prediction. Gradient Boosting Decision Trees are trained iteratively. Each decision tree is trained to minimise a certain loss function, for example the mean squared error. This is done by recursively splitting the data such that maximises a certain criterion. This is done until some limit, for example the depth of the tree, is reached. This criterion is chosen in such a way that the loss function is minimised at each split.

The next tree is constructed in such a way that minimises the loss function when the outputs are combined with the outputs of the first tree. The hardest part is computing the best split for each tree. There is no analytical solution for choosing the best split.

An example of such a, improved, boosting method is XGBoost[6]. This method can construct boosted trees in an efficiently way and can furthermore operate in parallel.

XGBoost is an improved boosting algorithm, based on decision tree gradient boosting[34] and it can construct boosted trees in an efficiently way and can furthermore operate in parallel. The main goal of XGBoost is to optimise the value of the objective function.

3.4.6. Conclusions

In order to make a decision in which machine learning algorithm is suitable for our problem, we discuss some advantages and disadvantages of the different algorithms.

Generally, SVMs and neural networks tend to perform much better when dealing with multi-dimensions and continuous features. On the other hand, logic-based systems tend to perform better when dealing with discrete/categorical features. For neural network models and SVMs, a large sample size is required in order to achieve its maximum prediction accuracy whereas NB may need a relatively small dataset.

Instance-based algorithms are very sensitive to irrelevant features. Moreover, the presence of irrelevant features can make neural network training very inefficient, even impractical.

Lazy learning methods require zero training time because the training instance is simply stored. Naive Bayes methods also train very quickly since they require only a single pass on the data either to count frequencies (for discrete variables) or to compute the normal probability density function (for continuous variables under normality assumptions). Univariate decision trees are also reputed to be quite fast-at any rate, several orders of magnitude faster than neural networks and SVMs.

In our problem, we have mainly continuous attributes. Discrete attributes are attributes that can only take particular values. There may potentially be an infinite number of those values. An example of a discrete attribute is number of resources a task is dependent on. Continuous attributes are not restricted to defined separate values, but can occupy any value over a continuous range. Between any two continuous attribute value, there may be an infinite number of others. An example of a continuous attribute is the release time or deadline of a task.

So we are looking for an algorithm that can handle mainly continuous attributes. Decision trees are capable of dealing with both continuous as discrete attributes. Furthermore, Neural Networks and Support Vector Machines can also deal with continuous attributes. However, Neural Networks are sensitive to overfitting in relation to SVM's and Decision Trees. Furthermore, they have a lower tolerance for irrelevant and redundant features. Therefor, we will mainly focus on using Support Vector Machines and Decision trees. Furthermore, we will look at Tree Ensemble methods, which often have a better performance than normal Decision Trees.

4 Improving Solve and Robustify with a machine learning model

In this chapter, we describe how we have tried to improve the Solve and Robustify algorithm by replacing the heuristics with a machine learning classifier that makes better decisions than the heuristics. First, we have executed several experiments in order to decide which machine learning methods obtain the best results. We have used several datasets for this. After generating the datasets, we have trained several classifiers to replace the heuristics and evaluated their performances when they are used in the solver. Lastly, we have implemented Monte Carlo simulation for improving the performance of the solver that uses the machine learning classifiers.

4.1. Motivation

As explained before, there are multiple heuristics that can be used for conflict selection and conflict resolution, we have selected two heuristics for conflict selection and five heuristics for conflict resolution. So in total, there are ten different combinations of heuristics that can be used for the precedence constraint posting solver. However, there is no combination of heuristics that is significantly better than any other. The average makespan of the solved problems is similar. In fact, different combinations of heuristics perform best on different scheduling problem instances. Choosing one of these combinations is not a good approach. Indeed, it is the case that one combination is better than others at a specific problem instance, but it can be dramatically worse on other problem instances.

Since we want to obtain the lowest makespan for each problem, it is preferred to choose the right combination of heuristics for each problem. This is called an ”Algorithm Selection Problem”[28]. This determines which algorithm(s) should be run in order to minimise some performance objective, for example the minimum makespan. The ideal solution to this algorithm would be to consult an oracle that knows the makespan of each combination of heuristics for each problem instance, and then to select the combination of heuristics that has the lowest makespan. Unfortunately, since we cannot precisely determine an arbitrary makespan result without actually running the algorithm, computationally cheap, perfect oracles do not exist for NP-problems.

There are approaches that are based on an heuristic approximation to a perfect oracle. For example, Xu et al.[33] have used machine learning techniques to build a computationally inexpensive predictor of such an oracle on a given problem instance of a Satisfiability problem, based on features of the instance and the past performance of the algorithms. This solution is based on algorithm portfolios. This is the strategy of selection of one or more algorithms, out of a set of algorithms, to execute and use the results of the executions to obtain a final result. This selection is done, as mentioned earlier, based on machine learning techniques. Since the effectiveness of these techniques are demonstrated in extensive experimental results[33], this could be a useful strategy to apply on our problem as well.

However, these solvers use a fixed algorithm per problem. The solvers can additionally be improved by dynamically switching between search strategies. Each encountered subproblem is a different instance of the problem, where the structure could be changed drastically. For example the Dynamic Approach for Switching Heuristics (DASH)[9] uses a dynamic approach for solving Mixed-Integer Programming problems. This algorithm computes the features of the subproblem in each step. Based in these features, it determines the nearest cluster and based on this nearest cluster, it selects a heuristic for making decisions. This approach could also be used in order to improve the PCP problem solver.

However, we expect that the choices could be further improved by using a slightly different approach. As explained before, DASH, for example, makes choices based on the choice the heuristic, that is associated to the nearest cluster of the subproblem, makes. Our approach is slightly different, since we do not choose a heuristic that makes the decision, but we let the classifier make the decisions itself. All conflicts or constraints are classified into good and bad conflicts or constraints. The conflict or constraint that has the highest probability of being in the class of good choices is eventually chosen. So instead of using a machine learning model to select a heuristic, we will implement machine learning models to select conflicts and constraints directly.

4.2. Use of heuristic data as features

Our first research question addressed the problem of which properties of tasks and/or task pairs have an influence on selecting and resolving conflict pairs. In other words, which features are useful to use in the dataset that we use for our classifier.

In Section 2.9, heuristics that are used for both conflict selection and conflict resolution are explained. Based on these heuristics, selections for conflicts are made. The heuristics use an educated guess, based on different features. Each heuristic uses its own property of the conflicts and tasks in order to make these decisions, which can cause each heuristic to make other choices than the other heuristics. Therefor, different heuristics result in a different makespan, depending on the problem and the properties of the conflicts and tasks present at that problem. There is no combination of heuristics that always perform better than any other combination. It depends on the problem which heuristics perform best.

Because these different features of the conflicts and tasks, which are used by the different heuristics, all have problems on which they perform best, we have decided to use these features to create our datasets for both conflict selection and conflict resolution. The goal for the machine learning model to eventually find all best practices of the heuristics. In other words, learn why the choices that the best heuristic for that specific problem makes work best.

After the decision was made to use the features that the heuristics use for decision making, the dataset that was used for training the classifiers were made. There are two different classifiers required, one for conflict selection and one for conflict resolution. These datasets are created based on the selections that the different heuristics make. Per problem, we have calculated which combination, or which combinations, of heuristics for conflict selection and conflict resolution resulted in the lowest makespan. The combinations which resulted in the best makespan per problem are run again, and each choice that was made by the heuristics, is added as an instance to the dataset. The generation of the datasets is explained in more detail in Section 4.4 and Section 4.5.

However, in order to generate these datasets, we also needed a set of Resource Constrained Project Scheduling Problems. We have used the PSPLIB benchmark dataset for this, which is explained in more detail in the next section.

4.3. Benchmark dataset

In order to compare the results of the different heuristic combinations, we need to have a test set with RCPSP instances. Fortunately, there is a benchmark test set available for the Resource Constrained Project Scheduling Problem, which is often used in previous literature. This test set is called PSPLIB and is designed by Kolisch[15]. It provides the scientific community a way to compare RCPSP solvers with each other, therefore it also can be used to compare the results of our experiments. This PSPLIB data set contains both Multi Mode and Single Mode instances. Multi Mode instances are instances where it is possible for tasks to pick different resources, for example having different modes, where tasks in Single Mode instances have no choice and have to pick one specific resource. Since the problem that is present at NedTrain is a problem, with tasks, precedence constraints and a finite resource capacity, the Single Mode data instances are sufficient to test the problem. Therefore, we will use the Single Mode PSPLIB data set as benchmark for comparing the results of our experiments.

There are however several differences between the PSPLIB data set and the problem that is present at NedTrain. First of all, the tasks in PSPLIB data set do not have deadlines and release times. This problem is resolved by Staats[31], where every task has a release time of 0 and a deadline of 250 for the problems with 60 tasks. Earlier experiments has showed that all instances are still solvable when using these release times and deadlines.

The dataset of RCPSP instances from the PSPLib are split up into four benchmark sets of different sizes. Each benchmark set is named: j30, j60, j90 and j120, where the numbers refer to the number of tasks in each RCPSP instance. Each benchmark set has constant properties and variable properties. The constant properties are the number of tasks and resource types. Within each set there are 3 properties that are variable. These properties are:

NC Temporal network complexity, which is the average number of precedence constraints per task

RS Resource strength , which defines how scarce (not abundant) the resources are

RF Resource factor, which defines how many different resources one task requires.

There are 48 different combinations of these three properties in a benchmark set. Each combination has 10 problem instances, which gives each benchmark set 480 instances.

We have used the j30 benchmark set for our experiments. We have chosen this set, because with 480 problems with 30 tasks each, the datasets that are generated are already large, with hundreds of thousands of choices that are made by the heuristics. These datasets are large enough to train the classifiers.

4.4. Selecting machine learning method

For our second research question, we wanted to find out which machine learning method(s) is/are the most effective method(s) for solving RCPSP instances. Selecting the best machine learning methods for our problem is crucial for obtaining good results. The performance of the classification heavily depends on using the appropriate classifier.

In this section, we only determine which machine learning method performs best on the problem instances we have. The classifiers that we learn in this step are not yet suited for making decisions in the Solve and Robustify algorithm. However, the selection of machine learning methods that is done in this section is used in the experiments described Section 4.5, in which we have learned the classifiers that are suited for replacing the heuristics.

In order to determine which machine learning method(s) generate the best results, we have first tried to train classifiers that imitate the heuristics separately. This is done in order to determine which method(s) can deal with the heuristics data as samples the best. As explained in Section 3.4.6, we have used Support Vector Machines, Decision Trees, Random Forest and XGBoost for these experiments.

The selection of the machine learning method is done by learning all heuristics separately and determine which methods perform best for all heuristics. This is done in order to determine which machine learning methods are able to imitate the behaviour of all heuristics. As input for the models, we use the heuristic values of all heuristics as features, and the goal is to learn each heuristic separately, which labels based on the choices the each specific heuristic makes. The expectation is that the rules of thumb that are used by the heuristics can be learned perfectly, since there are only some simple rules that need to be learned, like choosing the sample with the smallest value.

This experiment is executed separately for conflict selection and conflict resolution. In the next sections, we explain how the datasets are generated and we discuss the results of the selection of the machine learning methods.

4.4.1. Conflict selection

As explained before, the datasets are generated by running the solver on PSPLIB instances and creating an instance per time the heuristic selects a conflict. The values of the heuristics are used as feature and the choices that the heuristics make are used as label.

For conflict selection, we have two datasets per step. One dataset for ω_{res} and one dataset for Min-Slack. All samples in both datasets have two features, both the ω_{res} value and the Min-Slack value of the conflict.

For all datasets, the labels of the instances are determined in the same way. There are two possible labels that a sample can have, 0 and 1. A label of 0 means that the corresponding conflict is not chosen and a 1 means that the corresponding conflict is chosen by the heuristic. An example of a part of the dataset that is used for learning the ω_{res} heuristic is showed below.

```
# 1:wres 2:slack
0 1:203.703215 2:234
0 1:215.608441 2:245
0 1:204.843843 2:237
0 1:223.494966 2:238
0 1:217.485632 2:235
0 1:207.060378 2:237
1 1:195.468668 2:234
```

After generating the datasets for both ω_{res} and Min-Slack, we have evaluated the classifiers that have been learned based on these datasets. The first results are not very promising. For example, the confusion matrix and ROC curve of the classifier for ω_{res} with a random forest classifier, after using cross-validation, result in a lot of false positives and false negatives, as can be seen in Figure 4.1 and Figure 4.2.

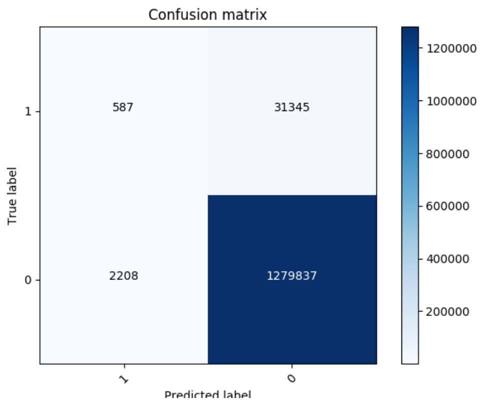

Figure 4.1: Imitating ω_{res}, confusion matrix for Random Forest classifier

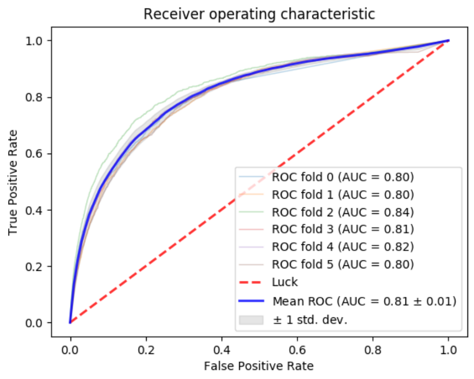

Figure 4.2: Imitating ω_{res}, ROC curve for Random Forest classifier

This poor performance has multiple reasons. These reasons are explained in the next sections.

Scaling instances

One of the reasons the classifiers do not perform perfectly, like we expected, is that for conflict selection, the heuristics use values that are relative to the values of other conflicts in the conflict set, as explained in Section 2.9.1. The conflict with the lowest heuristic value of the specific conflict set is chosen. Therefor, nothing useful can be said about a single sample for conflict selection. In order to overcome this, we have scaled the feature values per conflict set, to a value between zero and 1. The feature value of the conflict with the lowest heuristic value of the conflict set becomes zero and the feature value of the conflict with the highest heuristic value of the conflict set becomes one. The values of the other conflicts are scaled relatively to a value between zero and one. This results in all samples with feature value zero being assigned to class 1 and all other samples being assigned to class 0.

Handling imbalanced data

Besides the values that are relative to each other, there are also a lot more conflicts that are not chosen than conflicts that are chosen each conflict selection moment, there are lot more instances of class 0 than instances of class 1. The datasets for conflict selection are therefor quite imbalanced. In order to overcome this problem of imbalance, we have used an oversampling technique called SMOTE, which stands for Synthetic Minority Oversampling TEchnique, to create additional synthetic instances. We have chosen an technique that uses oversampling by adding synthetic instances, because oversampling by replication can lead to similar but more specific regions in the feature space as the decision region for the minority class[5]. This can potentially lead to overfitting on the multiple copies of minority class examples.

With SMOTE, the minority class is oversampled by taking the minority class sample and introducing synthetic samples based on these samples. Depending on the amount of oversampling that is required, neighbours from the k nearest neighbours are chosen randomly. The synthetic samples are then generated by taking the difference between the sample and its nearest neighbour, this difference is multiplied by a random value between 0 and 1 and this value is added to the sample. This approach is a good way to make the minority class become more general compared with replication of samples. The generated examples cause the classifier to create larger and less specific decision regions, instead of smaller and more specific regions.

Conflict selection after data preprocessing

After applying both the scaling and the oversampling on our datasets, we have learned the classifiers on the new datasets. An example of a part of the dataset that is used for learning the ω_{res} heuristic after scaling and oversampling is showed below.

```
# 1:wres 2:slack
0 1:0.293815 2:0
0 1:0.718603 2:1
0 1:0.334514 2:0.272727
0 1:1 2:0.363636
0 1:0.785582 2:0.090909
0 1:0.413601 2:0.272727
1 1:0 2xx:0
```

The results are as we expected, the heuristics can be imitated by a classifier perfectly. We have used cross-validation on our datasets to test the performance of the classifiers. All samples are classified correctly with each machine learning method, as can be seen for example for ω_{res} with XGBoost in Figure 4.3 and Figure 4.4.

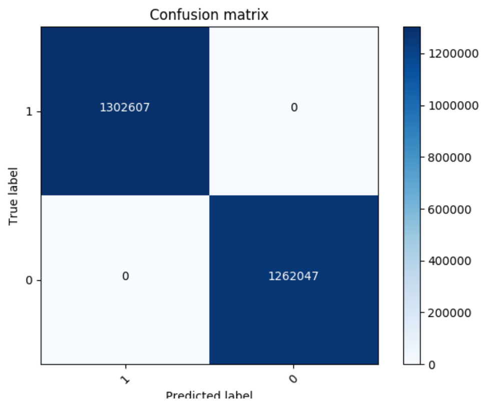

Figure 4.3: Imitating ω_{res} after scaling and oversampling, confusion matrix for XGBoost classifier

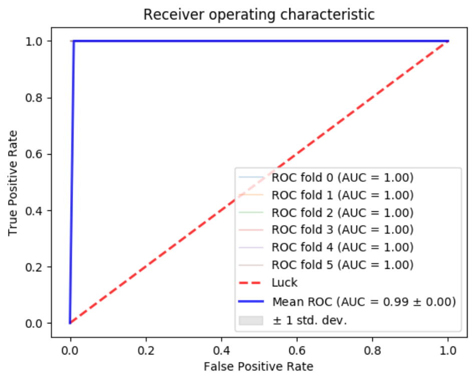

Figure 4.4: Imitating ω_{res} after scaling and oversampling, ROC curve for XGBoost classifier

4.4.2. Conflict resolution

For conflict resolution, the datasets are generated in the same way as for conflict selection, by running the solver on the PSPLIB dataset. For every conflict that is examined by the heuristics, a sample is added to the dataset. However, for conflict resolution we have generated five datasets per step, one dataset for each conflict resolution heuristic, namely Max-Slack, RM1, RB, $flex_I$ and makespan.

All samples in the datasets have ten features, the heuristic values of all conflict resolution heuristics for that specific conflict.

In the conflict resolution datasets, there are also two possible labels for a sample, also 0 and 1. The input of the constraint selection algorithm is a conflict with two tasks, t_1 and t_2. A sample with label 1 means that the resulting precedence constraint will be that t_1 will be executed before t_2 and a sample with label 0 means that t_2 will be executed before t_1.

An example of a part of the dataset that is used for learning the $flex_I$ heuristic showed below.

```
# 1:dij 2:dji 3:RM1_1 4:RM1_2 5:RB_1 6:RB_2 7:flexi_1
# 8:flexi_2 9:makespan_1 10:makespan_2
1 1:201 2:184 3:6565 4:6369 5:192.9268 6:183.1396 7:1905 8:1681 9:57 10:66
1 1:201 2:185 3:6557 4:6382 5:191.2924 6:182.8581 7:1905 8:1889 9:57 10:65
0 1:189 2:202 3:6441 4:6541 5:184.1990 6:188.9160 7:1893 8:1897 9:61 10:57
0 1:191 2:207 3:6391 4:6536 5:181.5589 6:188.0177 7:1855 8:1897 9:59 10:57
1 1:212 2:191 3:6533 4:6439 5:187.4236 6:183.3066 7:1893 8:1879 9:57 10:59
1 1:210 2:196 3:6529 4:6448 5:186.8176 6:180.0295 7:1893 8:1871 9:57 10:57
0 1:192 2:202 3:6420 4:6513 5:177.7688 6:185.1772 7:1858 8:1893 9:58 10:57
1 1:203 2:209 3:6550 4:6556 5:192.3255 6:192.7264 7:1894 8:1684 9:57 10:57
0 1:211 2:188 3:6542 4:6431 5:191.0848 6:187.0705 7:1689 8:1879 9:57 10:62
1 1:187 2:191 3:6385 4:6358 5:184.1429 6:182.2783 7:1877 8:1861 9:63 10:62
```

Not all classifiers were able to imitate the heuristics perfectly. Only the SVM and XGBoost classifiers were able to imitate the conflict resolution heuristics. The results for $flex_I$ and the SVM classifier can be seen in Figure 4.5 and Figure 4.6.

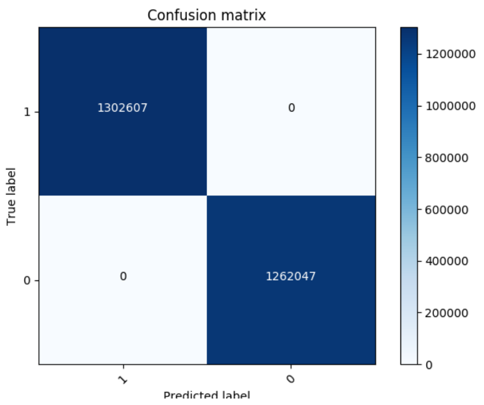

Figure 4.5: Imitating $flex_I$, confusion matrix for SVM classifier

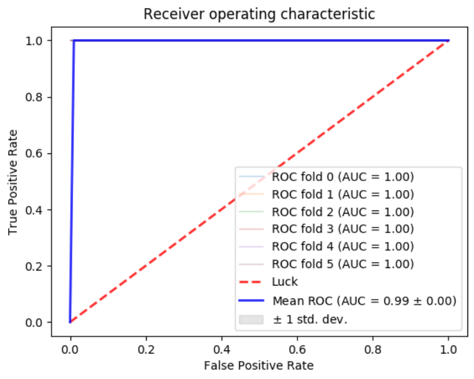

Figure 4.6: Imitating $flex_I$, ROC curve for SVM classifier

4.4.3. Conclusion

The heuristics can be imitated by machine learning classifiers for both conflict selection as conflict resolution. After some data preprocessing, scaling the features per conflict set and applying oversampling, each machine learning classifier can imitate the conflict selection heuristics perfectly. The rules of thumb that the conflict selection heuristics use are easy to learn for the different classifiers. For the conflict resolution heuristics however, only XGBoost and Support Vector Machines could imitate the heuristics nearly perfectly. The rules of thumb that the conflict resolution heuristics use are harder to learn for the Random Forest and Decision Tree classifiers, but still easy to learn for XGBoost and Support Vector Machines. So for our further research, we have focussed on XGBoost and Support Vector Machines classifiers, since these classifiers perform best on our datasets.

4.5. Training of model for replacing heuristics

After we have decided which machine learning methods we have used, we could start training the classifier which is used as replacement for the heuristics. For this classifier, as stated earlier, the goal is to learn the best practices of all heuristics, based on their heuristics feature values. In order to this, a dataset with the best practices is needed. We need only two datasets this time, one for conflict selection and one for conflict resolution.

In order to generate such a dataset, we first have run the problem set with all ten combinations of heuristics separately, in order to determine which combination, or combinations, of heuristics perform best on which problem. For the generation of the datasets, only the combinations which perform best are used. In the following sections, the generation of the datasets is explained in more detail.

4.5.1. Conflict selection

As stated before, only one classifier needs to be learned for conflict selection. The goal for the classifier is to learn the best practices from the ω_{res} and the Min-Slack heuristics. In order to generate the dataset, we have used the best performing combinations of heuristics per problem. While running these best combinations, every choice that the conflict selection heuristics make is saved as an instance in the dataset. For example, for the first problem in the benchmark test set, problem j301_1, the combination of the Min-Slack heuristic for conflict selection and the Max-Slack heuristic for conflict resolution results in the lowest makespan of 51. So in order to generate the dataset for conflict selection, we have run the solver with these heuristics for this specific problem, and for every step the Min-Slack heuristic is called, every conflict is saved as an instance in the dataset.

The dataset consists of, just as in the datasets for the second step, of two features, both the ω_{res} value and the Min-Slack value of the conflict. Just as before, there are two possible labels that a sample can have, 0 and 1. A label of 0 means that the corresponding conflict is not chosen and a 1 means that the corresponding conflict is chosen by the heuristic. An example of a part of the dataset, for problem j301_1, can be found below.

```
# 1:wres 2:slack
0 1:0.814033 2:0.846154
0 1:0.601177 2:0.384615
0 1:0.653788 2:0.461538
0 1:0.620828 2:0.230769
0 1:0.652197 2:0.384615
0 1:0.342146 2:0.307692
0 1:0.812947 2:0.384615
0 1:0.628188 2:0.923077
0 1:0.336483 2:0.461538
0 1:1.000000 2:0.769231
0 1:0.263479 2:0.230769
1 1:0.261197 2:0.000000
0 1:0.499304 2:0.615385
0 1:1.000000 2:0.769231
0 1:0.604496 2:0.307692
```

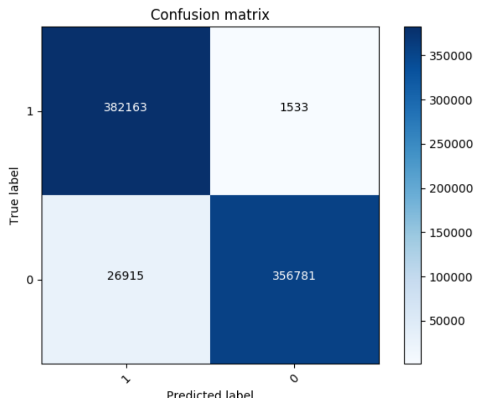

Figure 4.7: Confusion matrix of XGBoost classifier for conflict selection

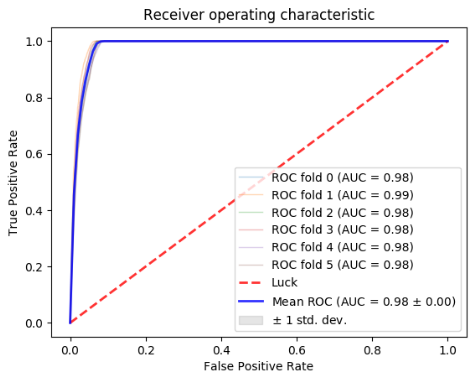

Figure 4.8: ROC curve of XGBoost classifier for conflict selection

After we have generated this dataset, we have used both SVM and XGBoost in order to learn a classifier. XGBoost performed best, especially after parameter tuning. Eventually we have used an XGBoost classifier, with a number of estimators, which is the number of boosted trees to fit, of 200 and a maximum depth, which is the maximum tree depth for base learners, of 10. This classifier performed quite well, with a relatively small amount of false positives and false negatives, which can be seen in Figure 4.7 and Figure 4.8.

4.5.2. Conflict resolution

For conflict resolution, there is also only one classifier needed. For this classifier, the goal is to learn from all best practices of the other heuristics too, this time from $flex_I$, $RM1$, RB, Max-Slack and Makespan. For the generation of this dataset, we have run the best combinations of all heuristics as well. For the same example of problem j301_1 as used for conflict selection, where the combination of the Min-Slack heuristic for conflict selection and the Max-Slack heuristic for conflict resolution results in the lowest makespan, we have run the solver with these heuristics for generating the dataset for conflict resolution as well. This time, for every step the Max-Slack heuristic is called, every selection made by the heuristic is saved as an instance in the dataset.

The dataset consists of 10 features, the heuristic values of all conflict resolution heuristics. In the dataset, a sample with label 1 means that the resulting precedence constraint will be that t_1 will be executed before t_2 and a sample with label 0 means that t_2 will be executed before t_1. An example of a part of the dataset, for problem j301_1, can be found below.

```
# 1:dij 2:dji 3:RM1_1 4:RM1_2 5:RB_1 6:RB_2 7:flexi_1
# 8:flexi_2 9:makespan_1 10:makespan_2
0 1:203 2:212 3:6796 4:6900 5:208.3303 6:214.2502 7:2199 8:2218 9:47 10:40
0 1:208 2:215 3:6842 4:6885 5:209.0562 6:212.8692 7:2196 8:2203 9:42 10:40
1 1:205 2:200 3:6832 4:6685 5:211.3130 6:198.3681 7:2203 8:1972 9:45 10:50
0 1:204 2:207 3:6792 4:6814 5:208.7626 6:209.0864 7:2192 8:1983 9:46 10:45
1 1:224 2:202 3:6813 4:6672 5:208.5017 6:199.4732 7:1983 8:1743 9:45 10:48
1 1:224 2:205 3:6811 4:6732 5:207.8843 6:202.7797 7:1981 8:1746 9:45 10:45
1 1:214 2:207 3:6792 4:6729 5:206.1479 6:202.6638 7:1973 8:1963 9:45 10:45
1 1:201 2:197 3:6712 4:6686 5:198.3568 6:201.8346 7:1965 8:1973 9:49 10:53
0 1:210 2:215 3:6667 4:6703 5:194.0776 6:196.6885 7:1951 8:1965 9:49 10:49
1 1:217 2:201 3:6699 4:6597 5:196.2676 6:188.9667 7:1961 8:1717 9:49 10:49
```

For this dataset, the XGBoost classifier worked best as well, as can be seen in Figure 4.9 and Figure 4.10. This time, after parameter tuning, we have use a number of estimators of 150 and maximum depth of 4. The number of false negatives and false positives is relatively low and the area under the ROC curve is really high, so the classifier has a good performance here as well.

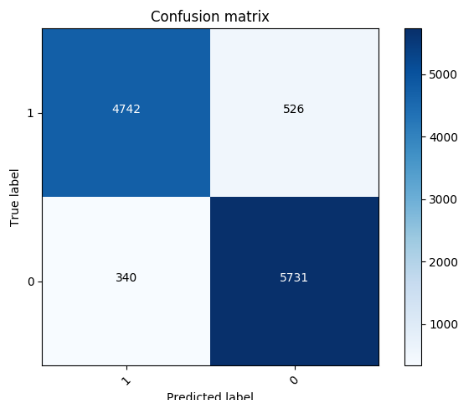

Figure 4.9: Confusion matrix of XGBoost classifier for conflict resolution

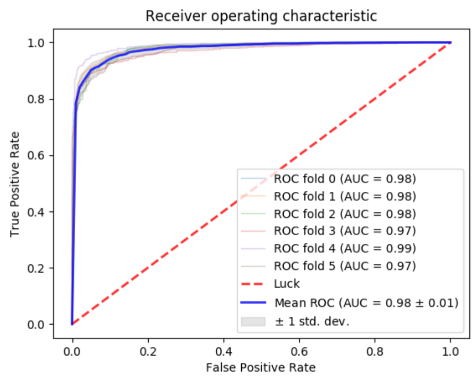

Figure 4.10: ROC curve of XGBoost classifier for conflict resolution

Table 4.1: The average makespans with all combinations of heuristics

Conflict selection heuristic	Constraint selection heuristic	Average makespan
ω_{res}	Max-Slack	71.870
Min-Slack	Max-Slack	71.670
ω_{res}	$flex_I$	72.308
Min-Slack	$flex_I$	73.047
ω_{res}	$RM1$	71.833
Min-Slack	$RM1$	71.662
ω_{res}	RB	71.943
Min-Slack	RB	72.054
ω_{res}	Makespan	85.845
Min-Slack	Makespan	85.147
ML conflict selection	ML constraint selection	71.850

4.6. Replacing heuristics by the model

In order to be able to answer research question 3, if a machine learning algorithm can improve the choices of selecting and resolving conflict pairs, we needed to implement the classifiers into the PCP solver. The classifiers replaced the heuristics in Algorithm 1, in the steps for SelectConflict and Select-PrecedenceConstraint. Per step in the solver, every heuristic feature is calculated and used as input for the classifiers.

For conflict selection, the ω_{res} and Min-Slack features are calculated for each conflict in the conflict set and per conflict, the classifier classifies the conflict. The goal is to choose the conflict that is classified with label 1, since a label 1 means that the corresponding conflict is chosen by the best practices of the heuristics. The classifier is able to return the probability of each instance of belonging to class 1. Therefor, the solver chooses the conflict that results in the highest probability for class 1 as to be chosen as conflict to resolve first.

For conflict resolution, the input is the conflict that is chosen in the previous step. For this conflict, the values for $flex_I$, $RM1$, RB, Max-Slack and Makespan are calculated and used as input for the classifier for conflict resolution. If the classifier classifies the instance as class 1, a constraint is posted that enforces the first task of the conflict to be executed before the second task. If the instance is classified as class 0, this will be the other way around.

In order to avoid overfitting, we have divided the problem set into 6 parts, each containing 80 problems. When running the solver on one of the parts, a classifier that is learned based on the data of the problems of the other 5 parts is used. Otherwise, we would use a classifier that is partly learned based on the problems that it is classifying, which could lead to influenced results.

After running the precedence constraint posting solver with the machine learning classifiers replacing the heuristics, we have compared the results with the results of using the heuristics. These results can be found in Table 4.1. In this table, the results of all combinations of heuristics are shown, with the average makespan of all 480 problems. In the last row, the average makespan of the solver that uses the machine learning classifiers instead of the heuristics is shown.

As can be seen, the solver with machine learning classifiers performs better than most combinations of heuristics, but there still are combinations that perform better than the classifiers in general, such as when using the solver with the combination of the Min-Slack heuristic for conflict selection and the $RM1$ heuristic for conflict resolution.

So the classifiers were not able to learn all best cases of the heuristics, while the classifiers performed well on the dataset. This can have several causes. First of all, it is possible that the more trivial choices, for example the choices that all heuristics make the same, can be learned really well, but the more difficult choices, where the heuristics make different choices, are learned more difficultly, while these are the most important choices to be learned. This is possibly caused by the problems having not enough tasks and therefor not being difficult enough to learn the difficult choices. In Section 4.7, we have used larger problems in order to overcome this.

Furthermore, there is a difference in approach in generating the datasets and using the classifiers for solving the problems. When generating the datasets, a fixed combination of two heuristics that performs best on that specific problem is used. However, different combinations of heuristics can perform best per step in the solver. Therefor, the classifier may be learned with the wrong input data. We have tried to overcome this with an alternative dataset, which is explained in Section 4.8.

4.7. Using problems with more tasks

In the previous sections, we have used the j30 PSPLIB benchmark test set for generating the datasets and running the solver. In order to test if problems with more tasks result in a better dataset and therefor better results when replacing the heuristics by the machine learning classifier, we have also tested using the j60 PSPLIB benchmark test set. In this test set, each problem had 60, instead of 30, tasks.

The training of the models is done exactly the same as the training of the j30 dataset. The dataset is build based on the best perform combination of heuristics for each problem. The performance of the classifier was equally good as the performance of the classifier for the j30 test set, as can be seen in Figure 4.11, Figure 4.12, Figure 4.13 and Figure 4.14.

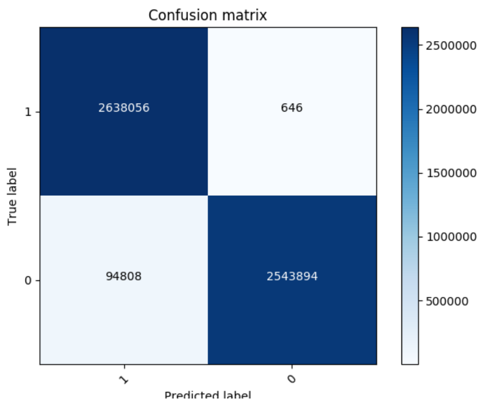

Figure 4.11: Confusion matrix of XGBoost classifier on the j60 test set for conflict selection

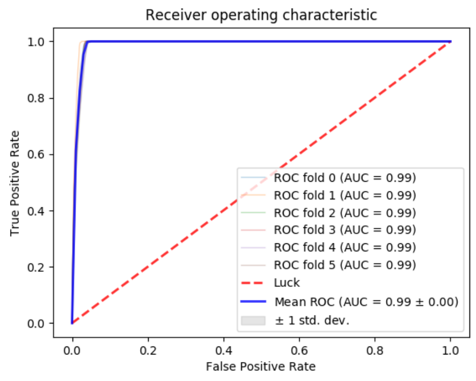

Figure 4.12: ROC curve of XGBoost classifier on the j60 test set for conflict selection

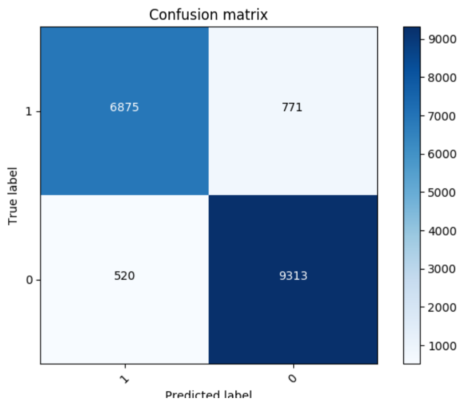

Figure 4.13: Confusion matrix of XGBoost classifier on the j60 test set for conflict resolution

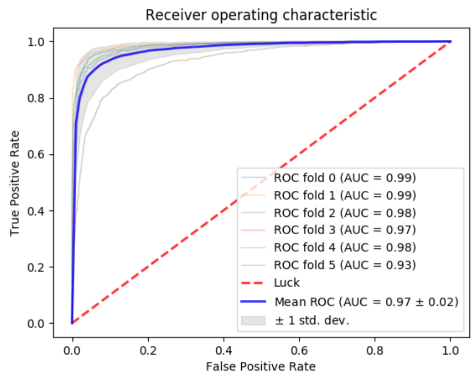

Figure 4.14: ROC curve of XGBoost classifier on the j60 test set for conflict resolution

After using the learned classifiers as replacement for the heuristics, the results are similar to the results after replacing the heuristics on the j30 test set. The average makespan when using the machine learning classifiers is lower than with most of the combinations of heuristics, but there are still some combinations that perform better than the classifiers in general. An overview of average makespan with all combinations of heuristics and the machine learning classifiers is given in Table 4.2.

Table 4.2: The average makespans with all combinations of heuristics on the j60 test set

Conflict selection heuristic	Constraint selection heuristic	Average makespan
ω_{res}	Max-Slack	102.825
Min-Slack	Max-Slack	102.185
ω_{res}	$flex_I$	104.264
Min-Slack	$flex_I$	103.272
ω_{res}	$RM1$	102.595
Min-Slack	$RM1$	101.818
ω_{res}	RB	102.831
Min-Slack	RB	101.993
ω_{res}	Makespan	126.854
Min-Slack	Makespan	122.533
ML conflict selection	ML constraint selection	102.262

4.8. Using an alternative dataset

In order to address the problem of a difference in approach in generating the datasets and using the classifiers for solving the problems, we have used a different approach of generating the dataset for learning the models. The previous datasets were build using the solver with a fixed combination of heuristics per problem, that performed best on that specific problem. The solver in which we used the classifier however, classifies the conflicts and constraints per step. Therefor, we have tried to generate a dataset that is generated with using the best combination of heuristics per step instead of per problem. We expected the dataset to result in a better classifier for conflict selection and conflict resolution, since the classifier is trained for selecting the optimal conflict and constraint per step, instead of a fixed combination of heuristics

In order to generate this dataset, we needed to know which combination of heuristics per step result in the best makespan per problem. However, it was not possible to brute force every combination of heuristics for every step in the solver, since this are way too many combinations. Therefor, we have chosen to select a random combination of conflict selection heuristic and conflict resolution heuristic per step in the solver and to save the sequence of heuristics that are chosen per problem, combined with the resulting makespan. Since the sequence is also saved, it was possible to rerun the sequences that resulted in the best makespan in order to generate the dataset. After running the solver with random heuristics, we had an extensive overview of different sequences with the best makespan per problem.

After reviewing these best makespans per problem, we noticed that there are problems that have a lot of different sequences of heuristics that result in the optimal makespan, while other problems have only a few, or even one, sequence that result in the lowest makespan. This is the case since some problems are easier to solve than other problems. In order to avoid overfitting the classifier to certain problems that have a lot of samples in the dataset, we have used oversampling and undersampling of the instances per problem, so that every problem has equal importance in the dataset.

4.8.1. Oversampling and undersampling the dataset

For oversampling, we have used SMOTE again. For undersampling however, we have used a method called Cluster Centroids. Cluster Centroids is a method that uses a K-means clustering algorithm[1]. Let N_t and N_i be the number of samples for respectively the minority and majority classes, so $N_i > N_t$, and let $\mu_k, k = 1, ..., N_t$ be the cluster centroids of the majority class, that are obtained by applying the K-means algorithm. These cluster centroids replace the samples of the majority class. So the number of samples of the majority class after undersampling is the same as the number of samples of the minority class. This approach has advantages over randomly selection samples from the majority class. For example, the selection of samples which are similar to each other and have similar classification difficulties is avoided.

4.8.2. Results

After applying oversampling and undersampling in order to even out the amount of samples per problem, the classifiers for both conflict selection and conflict resolution are trained. After evaluating the classifiers, we noticed that the dataset was harder to classify than the previous dataset. The amount of false positives and false negatives was higher than before and the area under the curve of the ROC curve was also lower, as can be seen in Figure 4.15, Figure 4.16, Figure 4.17 and Figure 4.18.

We have also used the two learned classifiers, one for conflict selection and one for conflict resolution, in the solver as replacement of the heuristics. The average makespan when using the alternative dataset was 73.062, even higher than when using the original datasets, as can be seen in Table 4.3.

We think this is the case because we cannot retrieve all optimal sequences of the problems, since brute forcing all sequences is not possible. Therefor, only a subset of the optimal solutions can be found. It is hard to find the rules for the best case decisions that the heuristics make for all problems.

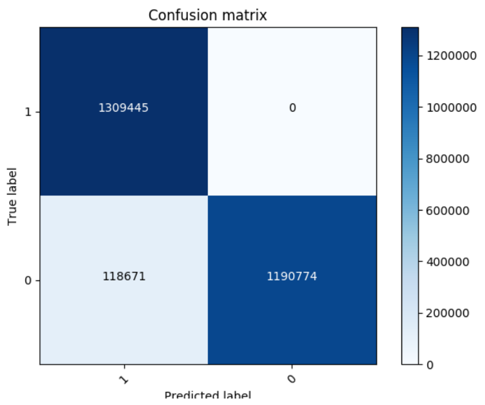

Figure 4.15: Confusion matrix of XGBoost classifier for conflict selection with the alternative dataset

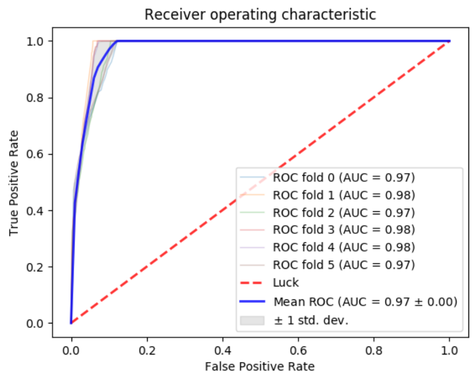

Figure 4.16: ROC curve of XGBoost classifier for conflict selection with the alternative dataset

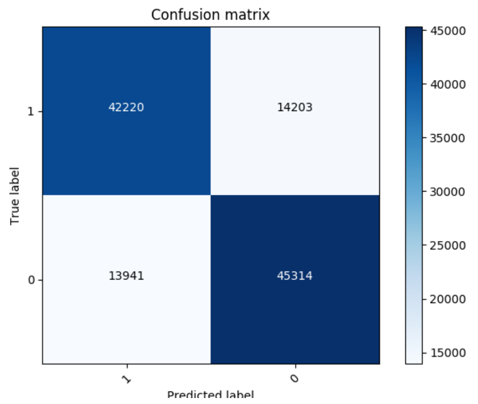

Figure 4.17: Confusion matrix of XGBoost classifier for conflict resolution with the alternative dataset

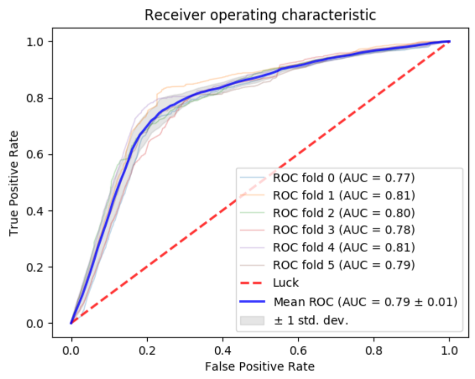

Figure 4.18: ROC curve of XGBoost classifier for conflict resolution with the alternative dataset

4.9. Monte Carlo Simulation

The use of an alternative dataset did not improve the quality of the solutions. The average makespan of the solutions of the benchmark dataset was even higher than when we used the first dataset. However, the machine learning model can also be used in an other way than as just a classifier. The classifier is able to return a probability of the likeliness of belonging to class 1, for each conflict for both conflict selection and conflict resolution. We have used these probabilities for Monte Carlo simulation to further improve the solutions of the precedence constraint posting algorithm.

4.9.1. Introduction to Monte Carlo techniques

The essence of Monte Carlo simulation is the generation of random objects or processes using a computer[17]. These objects could arise naturally, for example as part of the modelling of a real-life

Table 4.3: The average makespans with all combinations of heuristics including the classifier with the alternative datasets

Conflict selection heuristic	**Constraint selection heuristic**	**Average makespan**
ω_{res}	Max-Slack	71.870
Min-Slack	Max-Slack	71.670
ω_{res}	$flex_I$	72.308
Min-Slack	$flex_I$	73.047
ω_{res}	$RM1$	71.833
Min-Slack	$RM1$	71.662
ω_{res}	RB	71.943
Min-Slack	RB	72.054
ω_{res}	Makespan	85.845
Min-Slack	Makespan	85.147
ML conflict selection	ML constraint selection	71.850
ML conflict selection alternative dataset	ML constraint selection alternative dataset	73.062

system. However, the random objects are often artificially introduced in order to solver purely deterministic problems. With these artificial introduced problems, Monte Carlo simulation simply involves random sampling from certain probability distributions, such as we have encountered with our classifier. The general idea of Monte Carlo techniques is to repeat the experiment many times in order to obtain many quantities of interest.

Optimisation is a typical use case of Monte Carlo simulation. In many applications, complicated objective functions are deterministic and the randomness is introduced artificially to more efficiently search the domain of the objective function. In our case, the choices of conflict selection and conflict resolution are introduced artificially.

There are multiple reasons why Monte Carlo techniques are useful[17]. First of all, Monte Carlo algorithms are easy and efficient to use. The implementations tend to be highly scalable. Monte Carlo algorithms are very easy to parallelise too, since the various repetitions of the experiment can be run independently. This allows the repetitions to run parallel, which significantly reduces the computation time. Furthermore, the randomness of the Monte Carlo simulation is also a strength. It is not only essential for the simulation of random systems, but it is also a great benefit for deterministic numerical computation. For example, with randomised optimisation, the randomness can cause the algorithm to naturally escape local optima, which results in a better exploration of the search space.

We have applied Monte Carlo simulation in our precedence constraint posting solver, Solve and Robustify. In order to test if the probabilities that our classifier result in a better performance of the Monte Carlo simulation than when using the heuristics, we have implemented a Monte Carlo simulation for both the heuristics and our classifier, which are explained in the next sections. We expect the solutions to be better than when not using Monte Carlo simulation, since the solver looks ahead to see which decisions are better to make in 5 steps. We have chosen 5 steps, since the average amount of precedence constraint posting steps in our problem set is around 10 steps per problem. This is both for using Monte Carlo simulation with the machine learning classifier as when using random heuristics. The most important motivation for this experiment is to check wether the classifier makes better decisions for the random walk in the Monte Carlo simulation.

The idea behind the Monte Carlo solvers is to simulate the behaviour of the solver and to select and resolve conflicts based on the result of that simulation. For our experiment we have chosen to take 100 samples per choice with a simulation depth of 5 and then choose the conflict or constraint that results in the lowest makespan after these 5 steps. This means that for each selection moment, 100 parallel simulations are run that simulate 5 steps of the solver. With using only 5 steps, the runtime of the algorithm still reasonably manageable. The result of these simulations are problems that have a makespan. Eventually, the conflict or constraint that is chosen in the first step, that results in the lowest makespan after 5 steps, is chosen as the next conflict or constraint. The exact algorithms are explained in the following sections.

Algorithm 3 Monte Carlo solver with random heuristics

```
Input: A problem $P$, number of Monte Carlo samples $n$, Monte Carlo depth $m$
Output: A schedule $S$
while $true$ do
    $S$ ← CalculateESTS($P$)
    $conflictSet$ ← FindResourceConflicts($S$)
    if $conflictSet = \emptyset$ then
        return $S$
    end if
    if Unsolvable($conflictSet$) then
        return $failure$
    end if
    for 1 to $n$ do
        $temporal\_conflict$ ← SelectConflictRandomHeuristic($conflictSet$)
        $temporal\_problem$ ← SelectMonteCarloConflictHeuristic($P$, $temporal\_conflict$, $n$)
    end for
    $conflict$ ← $temporal\_conflict$ with lowest makespan of associated $temporal\_problem$
    for 1 to $n$ do
        $temporal\_constraint$ ← SelectConstraintRandomHeuristic($conflict$)
        $temporal\_problem$ ← SelectMonteCarloConstraintHeuristic($P$, $temporal\_constraint$, $n$)
    end for
    $constraint$ ← $temporal\_constraint$ with lowest makespan of associated $temporal\_problem$
    PostPrecedenceConstraint($P$, $constraint$)
end while
```

Algorithm 4 Monte Carlo conflict selection with random heuristics

```
Input: A problem $P$, a conflict $conflict$, a depth $m$
Output: A problem $P\_result$
$temporal\_conflict$ ← SelectConstraintRandomHeuristic($conflict$)
PostPrecedenceConstraint($P$, $constraint$)
if $m$ - 1 is 0 then
    return $P$
else
    $S$ ← CalculateESTS($P$)
    $conflictSet$ ← FindResourceConflicts($S$)
    if $conflictSet = \emptyset$ then
        return $S$
    end if
    if Unsolvable($conflictSet$) then
        return $failure$
    end if
    $temporal\_conflict$ ← SelectConflictRandomHeuristic($conflictSet$)
    return SelectMonteCarloConflictHeuristic($P$, $temporal\_conflict$, $m - 1$)
end if
```

4.9.2. Random heuristics

When using the Monte Carlo solver with random heuristics, each time a conflict selection is done, one of the two possible conflict selection heuristics is taken to choose the conflict. For conflict resolution, one of the five possible conflict resolution heuristics is taken to choose the constraint. The basis of the Monte Carlo solver is the same as the ESTA algorithm, Algorithm 1. The differences are the parts of conflict selection and precedence constraint selection. For both conflict selection and precedence constraint selection, a number of n samples are executed. In each sample, a number of m next steps of the solver are executed, which results in a scheduling problem. For each sample, the makespan of the resulting problem is calculated and the conflict or constraint of the sample with the lowest makespan is chosen as the next conflict or constraint. This can be seen in Algorithm 3, where Algorithm 4 and Algorithm 5 perform the simulations steps for conflict selection and constraint selection respectively.

Algorithm 5 Monte Carlo constraint selection with random heuristics

```
Input: A problem $P$, a constraint $constraint$, a depth $m$
Output: A problem $P\_result$
PostPrecedenceConstraint($P$, $constraint$)
if $m$ - 1 is 0 then
    return $P$
else
    $S \leftarrow$ CalculateESTS($P$)
    $conflictSet \leftarrow$ FindResourceConflicts($S$)
    if $conflictSet = \emptyset$ then
        return $S$
    end if
    if Unsolvable($conflictSet$) then
        return $failure$
    end if
    $temporal\_conflict \leftarrow$ SelectConflictRandomHeuristic($conflictSet$)
    $temporal\_constraint \leftarrow$ SelectConstraintRandomHeuristic($temporal\_conflict$)
    return SelectMonteCarloConstraintHeuristic($P$, $temporal\_constraint$, $m - 1$)
end if
```

After using the random heuristics Monte Carlo solver on the 480 PSPLIB benchmark test sets, the resulting average makespan is 68.991. This is indeed better than the average makespan using only heuristics, which can be found in Table 4.1.

Algorithm 6 Monte Carlo solver with machine learning classifier

```
Input: A problem $P$, number of Monte Carlo samples $n$, Monte Carlo depth $m$
Output: A schedule $S$
while $true$ do
    $S \leftarrow$ CalculateESTS($P$)
    $conflictSet \leftarrow$ FindResourceConflicts($S$)
    if $conflictSet = \emptyset$ then
        return $S$
    end if
    if Unsolvable($conflictSet$) then
        return $failure$
    end if
    $conflictProbabilities \leftarrow$ ClassifyConflicts($conflictSet$)
    for 1 to $n$ do
        $temporal\_conflict \leftarrow$ SampleConflict($conflictSet$, $conflictProbabilities$)
        $temporal\_problem \leftarrow$ SelectMonteCarloConflictHeuristic($P$, $temporal\_conflict$, $n$)
    end for
    $conflict \leftarrow temporal\_conflict$ with lowest makespan of associated $temporal\_problem$
    $constraintProbabilities \leftarrow$ ClassifyConstraints($conflict$)
    for 1 to $n$ do
        $temporal\_constraint \leftarrow$ SampleConstraint($conflict$, $constraintProbabilities$)
        $temporal\_problem \leftarrow$ SelectMonteCarloConstraintHeuristic($P$, $temporal\_constraint$, $n$)
    end for
    $constraint \leftarrow temporal\_constraint$ with lowest makespan of associated $temporal\_problem$
    PostPrecedenceConstraint($P$, $constraint$)
end while
```

4.9.3. Machine learning classifier

We have also implemented the Monte Carlo solver with use of the machine learning classifiers that we have learned in Section 4.5, instead of using random heuristics. The selection of conflicts and constraints deviates slightly from the selection for the random heuristics. The process of Monte Carlo simulation, with 100 samples and 5 steps per sample is the same. The difference lays within the random selection of the conflicts and constraints. Instead of using a random heuristic for the selection, the classifiers for conflict selection and conflict resolution, that are explained in Section 4.5, are used. The algorithm can be seen in Algorithm 6.

For conflict selection, the classifier returns per conflict a probability of belonging to class 1, or in other words the probability for being chosen by the best cases of the heuristics. Every time a random conflict needs to be chosen, a conflict is sampled from the conflict set, with a probability based on the probabilities of belonging to class 1. Therefor, the conflict with the highest probabilities of belonging to class 1 have the highest chance to be selected every step. The algorithm for selecting conflicts can be seen in Algorithm 7.

For conflict resolution, there are two possible constraints that can be chosen. The classifier for conflict resolution returns a probability for each constraint to be chosen by the best cases of the heuristics, just as with conflict selection. In every step, a constraint is chosen based on these probabilities and therefor, the constraint with the highest probabilities of being chosen by the best cases of the heuristics has the highest chance of being selected every step. The algorithm for selecting conflicts can be seen in Algorithm 8.

Algorithm 7 Monte Carlo conflict selection with machine learning classifier

```
1: **Input:** A problem $P$, a conflict $conflict$, a depth $m$
2: **Output:** A problem $P\_result$
3: $constraintProbabilities \leftarrow$ ClassifyConstraints($conflict$)
4: $temporal\_constraint \leftarrow$ SampleConstraint($conflict$, $constraintProbabilities$)
5: PostPrecedenceConstraint($P$, $constraint$)
6: **if** $m$ - 1 is 0 **then**
7:    **return** $P$
8: **else**
9:    $S \leftarrow$ CalculateESTS($P$)
10:   $conflictSet \leftarrow$ FindResourceConflicts($S$)
11:   **if** $conflictSet = \emptyset$ **then**
12:      **return** $S$
13:   **end if**
14:   **if** Unsolvable($conflictSet$) **then**
15:      **return** $failure$
16:   **end if**
17:   $conflictProbabilities \leftarrow$ ClassifyConflicts($conflictSet$)
18:   $temporal\_conflict \leftarrow$ SampleConflict($conflictSet$, $conflictProbabilities$)
19:   **return** SelectMonteCarloConflictHeuristic($P$, $temporal\_conflict$, $m - 1$)
20: **end if**
```

After using the machine learning Monte Carlo solver on the 480 PSPLIB benchmark test sets, the resulting average makespan is 68.818750. This is indeed better than using only heuristics, as can be seen in Table 4.1. Furthermore, it is also lower than the average makespan of using the heuristics Monte Carlo solver, which was 68.991667.

4.9.4. Conclusion

As stated earlier, the average makespan after solving the 480 PSPLIB benchmark test set problems was 68.818 with the machine learning Monte Carlo solver and 68.991 for the heuristics Monte Carlo solver, see Figure 4.19. The distribution of the makespans is more or less the same, as can be seen in Figure 4.20.

The average makespan of the Monte Carlo solver when using the machine learning classifiers is slightly lower than when using random heuristics, while the solver without Monte Carlo simulation with the machine learning classifier performs worse than the best combination of heuristics. This means

Algorithm 8 Monte Carlo constraint selection with machine learning classifier

```
**Input:** A problem $P$, a constraint $constraint$, a depth $m$
**Output:** A problem $P\_result$
PostPrecedenceConstraint($P$, $constraint$)
**if** $m$ - 1 is 0 **then**
    **return** $P$
**else**
    $S \leftarrow$ CalculateESTS($P$)
    $conflictSet \leftarrow$ FindResourceConflicts($S$)
    **if** $conflictSet = \emptyset$ **then**
        **return** $S$
    **end if**
    **if** Unsolvable($conflictSet$) **then**
        **return** $failure$
    **end if**
    $conflictProbabilities \leftarrow$ ClassifyConflicts($conflictSet$)
    $temporal\_conflict \leftarrow$ SampleConflict($conflictSet$, $conflictProbabilities$)
    $constraintProbabilities \leftarrow$ ClassifyConstraints($temporal\_conflict$)
    $temporal\_constraint \leftarrow$ SampleConstraint($conflict$, $constraintProbabilities$)
    **return** SelectMonteCarloConstraintHeuristic($P$, $temporal\_constraint$, $m-1$)
**end if**
```

that the machine learning classifier is worse in selecting the best conflict and constraint, but it is better in selecting the set of best conflicts and constraints. The set of best conflicts and constraints are chosen more often, which results in better makespans after five steps of Monte Carlo simulation, which eventually results in better results for the Monte Carlo solver with machine learning classifier.

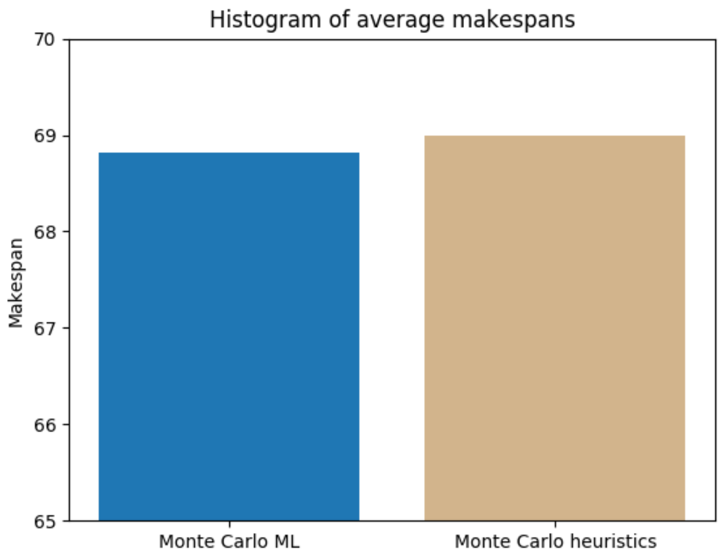

Figure 4.19: The average makespans after solving the 480 PSPLIB benchmark test set problems with Monte Carlo solvers

In Figure 4.21, the makespan of the Monte Carlo solver with random heuristics is plotted against the makespan of the Monte Carlo solver with machine learning classifiers. As can be seen, for some problems the Monte Carlo solver with random heuristics performs better and for other problems the Monte Carlo solver with machine learning classifiers performs better. Out of the 480 problems, the Monte Carlo solver with machine learning classifiers performs better on 136 problems and the Monte Carlo solver with random heuristics perform better on 99 problems. In order to test whether the decrease in

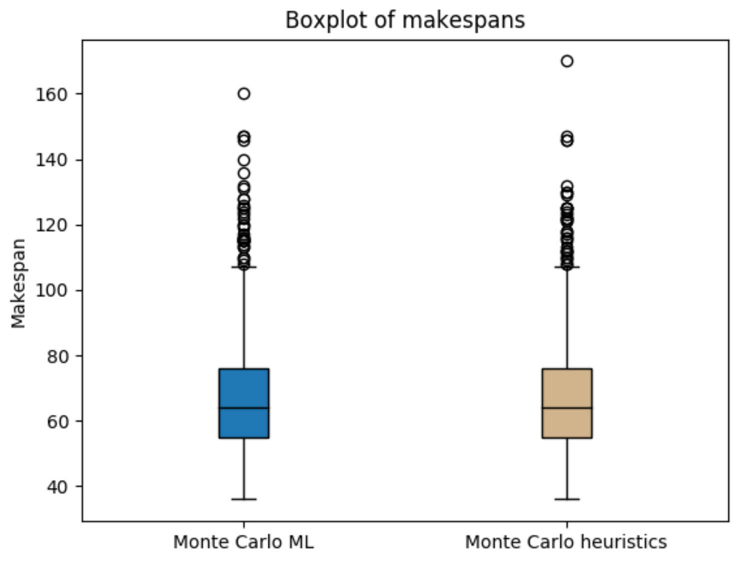

Figure 4.20: The boxplot after solving the 480 PSPLIB benchmark test set problems with Monte Carlo solvers

makespan is a statistically significant decrease, we have used a statistical hypothesis test.

The differences between the two makespan scores is not normally distributed. In Figure 4.22, the Q-Q plot of the differences in makespan is shown. Since the data is not aligned on the red line, the difference in makespan is not distributed normally. Therefor, we could not use the Student's t-test for checking statistical significance. Therefor, we have used the Wilcoxon singed-ranked test in order to verify whether the decrease in makespan was statistically significant. This is statistical hypothesis test for comparing two related samples when the population cannot be assumed to be normally distributed[21]. In this test, the hypothesis that is being tested is:

- **Null hypothesis:** The difference between the pairs follows a symmetric distribution around zero
- **Alternative hypothesis:** The difference between the pairs does not follow a symmetric distribution around zero

If the resulting p-value of the test is less than 0.05, the null hypothesis can be rejected. However, the limitation of the Wilcoxon signed-rank test is that it discards samples with a difference of 0. Since our results have 245 out of 480 results with a difference of 0, we have used the modification of the Wilcoxon test proposed by Pratt[26]. This modification incorporates the zero differences. The resulting p-value after running the Wilcoxon test is 1.7127512595031528e-12, which is lower than 0.05. Therefor, we can conclude that the average makespan when using random heuristics was higher compared to the average makespan when using machine learning classifiers. There was a statistically significant decrease in makespan.

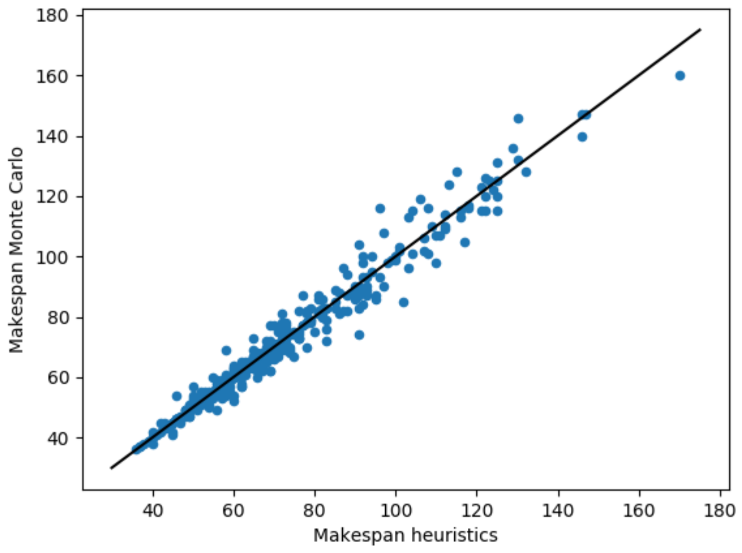

Figure 4.21: The scatter plot of resulting makespan per problem after solving the 480 PSPLIB benchmark test set problems with Monte Carlo solvers

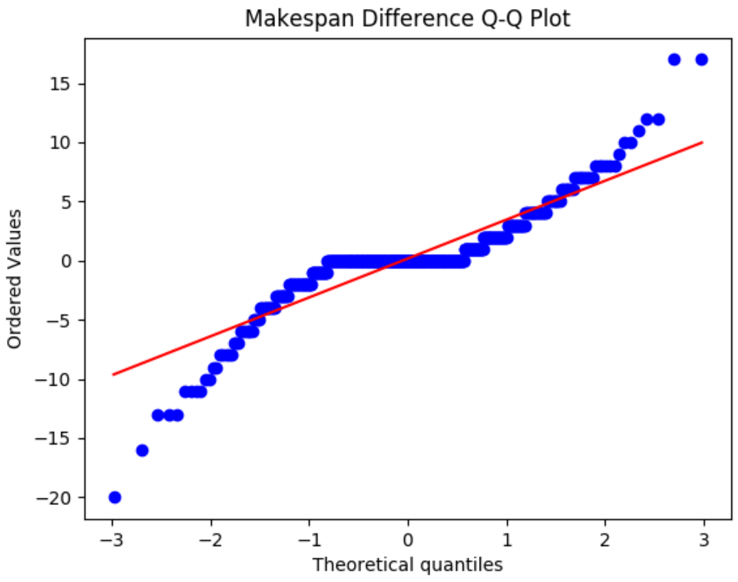

Figure 4.22: Q-Q Plot of difference in makespan for the Monte Carlo solvers

5 Conclusion and future work

Maintenance scheduling is a difficult problem when dealing with both temporal constraints and resource constraints. Several kind of tasks are present, such as recurring check-ups, additional unexpected repairs and small refurbishments. All these tasks have strict deadlines that need to be met. There are currently several methods for solving these kind of problems, but since it is a hard problem there are no exact solvers that run in a reasonable amount of time yet. The solvers that are currently available make use of heuristics, since they are in general much faster in solving problems. However, heuristic solvers often do not find optimal solutions for difficult problems. Therefor, we have researched the possibility of replacing the heuristics by machine learning classifiers that make better decisions and therefor result in better performance of the solver.

In this work we first described the challenges faced by maintenance scheduling with both resource constraints and temporal constraints. We studied the well defined scheduling model named Resource Constrained Project Scheduling Problem, together with a technique that simplifies resource constrained problems to problems containing only temporal constraints, named Precedence Constraint Posting. Furthermore, we studied several machine learning techniques and the advantages and disadvantages of these several techniques. We formulated detailed research questions and in this work we focused on answering these questions. In the next section we will look at the results of the research, based on these research questions.

5.1. Research

The goal for this work was to improve the performance of the Precedence Constraint Posting algorithm, Solve and Robustify, by replacing the used heuristics by machine learning classifiers. Using several datasets and machine learning techniques, we proposed to improve the quality of the resulting maintenance schedules.

5.1.1. Research questions

In Chapter 1, we proposed three research questions. We have answered these research questions in Chapter 4. The first question we have answered is:

Research Question 1: Which properties of tasks and/or task pairs have an influence on selecting and resolving conflict pairs?

We have eventually chosen to use the same input data for our datasets as the data that is input for the heuristics. Each heuristic uses its own property of the conflicts and tasks in order to make an educated guess for making choices. The different heuristics can make different choices regarding conflict selection and conflict resolution and therefor may result in a different makespan for the same problem. The features used by the heuristics have different problems on which they perform best. The goal of the machine learning classifiers is to learn the best practices of each heuristics in order to make choices based on the heuristic that works best for that particular problem.

When replacing the heuristics for machine learning classifiers, two classifiers are needed. One classifier for conflict selection and one classifier for conflict resolution. We have used two different approaches for generating the datasets. For conflict selection, the input data first had to be scaled and oversampled in order to be able to handle the values that are relative to the other values in a conflict set.

First of all, we have generated the datasets using a fixed combination of heuristics for conflict selection and conflict resolution that perform best per problem. After these combinations are determined for each problem, the solver is executed with these combinations and the choices that are made by the heuristics are used as instances for the datasets.

Furthermore, we have used an alternative method for generating the datasets that not uses a fixed combination of heuristics per problem, but uses a different combination of heuristics that perform best per step in the algorithm. We have done this in order to overcome the problem where the solver classifies the conflicts and constraints per step instead of per problem. We have selected random combinations of heuristics per step in each problem and executed the solver multiple times. The sequences of heuristics were saved and the sequences that resulted in the lowest makespan were run again for the generation of the datasets.

Next, we focused on the selection of the machine learning methods that are most effective on the datasets. The second research question we have answered in this work is:

Research Question 2: Which machine learning method(s) is/are the most effective method(s) for solving RCPSP instances?

In order to select the most effective machine learning methods, we have learned classifiers on a slightly easier problem. We have first imitated each heuristic separately using machine learning classifiers. This is done in order to determine which machine learning methods generate the best results when the heuristics data is used as input. We have run these experiments separately for conflict selection and conflict resolution. In Chapter 3, we already decide to focus only on Support Vector Machines, Decision Trees, Random Forest and XGBoost.

For conflict selection, all heuristics could be imitated perfectly after the scaling and oversampling was applied to the dataset. Every machine learning technique was able to imitate the heuristics.

For conflict resolution, not every machine learning technique was able to imitate the heuristics perfectly. Only Support Vector Machines and XGBoost were able to imitate the heuristics perfectly. Therefor, we selected Support Vector Machines and XGBoost as the most effective methods for classifying the conflicts for our solver.

Lastly, we have tried to improve the solutions of the Solve and Robustify algorithm by replacing the heuristics by machine learning classifiers. The last research question we have answered in this work is:

Research Question 3: Can a machine learning algorithm improve the choices of selecting and resolving conflict pairs?

In order to answer the last research question, we have learned the classifiers on the datasets. We first used the first datasets, with the fixed combination of heuristics, to train the classifiers that replaced the heuristics in the solver. When solving the problems with the solver that uses these classifiers, the solver performed better than most combinations of heuristics, but there were still combinations of heuristics that performed better than the classifiers in general. So the classifiers were not able to learn all best cases of the heuristics.

In order to test if problems with more tasks result in a better dataset in which the more difficult choices could be learned, we have also learned classifiers based on datasets that are generated based on the j60 PSPLIB dataset, which has 60 instead of 30 tasks per problem. The results were similar to the experiment with the j30 test set. The solver performed better than most combinations of heuristics, but there were still combinations of heuristics that performed better than the classifiers in general.

Secondly, we have learned the classifiers based on alternative datasets, in which we tried to obtain optimal schedules with dynamic heuristic selection per step. We have done this in order to overcome the problem of difference in approach in generating the datasets and using the classifiers for solving the problems. We expected the solver with the alternative dataset to perform better, since the classifier is trained for selecting the optimal conflict and constraint per step, instead of a fixed combination of heuristics. However, the average makespan was even higher than when using the first datasets. This was the case because not all optimal sequences of heuristics could be retrieved, because brute forcing all sequences was not feasible. It is hard to find the rules for the best case decisions that the heuristics make for all problems when not all optimal solutions are known.

Furthermore, we have used Monte Carlo simulation to further improve the solutions of the Solve and Robustify algorithm. The classifiers for both conflict selection and conflict resolution are able to return probabilities of the likeliness of belonging to a certain class, so we have used these probabilities for Monte Carlo simulation.

The idea behind the Monte Carlo solver is to simulate the behaviour of the solver and select and resolve conflicts based on the results of that simulation. In our experiment, we have chosen to take 100 samples per choice, with a simulation depth of 5 steps. We have run the solver with choosing a random heuristic where every heuristic has an equal probability of being selected in each step. Furthermore, we have run the solver with choosing a conflict or constraint based on the probabilities that are returned by the classifiers in each step.

The average makespan after solving the 480 PSPLIB benchmark test set problems was 68.991 when using the random heuristics and 68.818 when using the machine learning classifiers. We have also showed that there was a statistically significant decrease in makespan when using the machine learning classifiers in the Monte Carlo solver opposed to the random heuristics. This means that the machine learning classifiers are better in selecting the probability of best conflicts and constraints. The conflicts and constraints are chosen more often by the Monte Carlo solver in each step, which eventually leads to better results for the Monte Carlo solver with the machine learning classifiers.

5.2. Future work

We have eventually only improved the results of RCPSP instances slightly, after using Monte Carlo simulation. In this section, we highlight some possibilities to further improve the quality of the solutions.

We only looked at using the features that are used by the heuristics for the generation of our dataset and training of the classifiers. Extra features could be used in order to add more context of the problem. More information about the state of the problem at the moment of choosing conflict and constraint could result in an improvement of the choices that are made. However, choosing the right features can be just as hard as solving the problem itself. Feature selection requires a deep knowledge of the problem domain.

Examples of features that can be used in order to give more context to the machine learning model are for example the release time, the deadline and the duration of tasks. However, these features need to be preprocessed in order to use them for classifying. Many machine learning methods need features to have the same scale in order to process them. For example, in order to be able to compare duration times of tasks, these durations need to be scaled based on the average duration of all tasks. The release time and deadline need to be converted to a specific interval in order to only look at the relative release time and deadline between tasks, instead of the real value of the features.

Furthermore, at this moment, we do not know the optimal choices the Precedence Constraint posting algorithm needs to make in order to result in the optimal makespan. The optimal makespan of each problem in the j30 PSPLIB test set is available, but the choices that need to be made in the PCP solver that result in these makespans are not available. We approximate the optimal choices by using the best performing combination of heuristics per problem, but this does not always result in the optimal makespan. Since the optimal choices are not known, it is hard for the machine learning algorithm to learn the optimal choices, since some samples in the dataset are not optimal. If it is possible to obtain these optimal choices, the classifier would perform better, since all samples would be based on optimal choices.

In Chapter 4, we briefly discussed a dynamic heuristic switching approach[9]. This approach also looks at the encountered subproblem at each subproblem for making decisions. However, this approach selects the heuristic that is associated with the nearest cluster of the subproblem, and that specific heuristic makes the selection for that step. This is contrary to our approach, where the machine learning model makes the selection instead the model selecting a heuristic that makes the selection. In future work, it is possible to combine these two approaches for a more effective and robust solver.

Our last suggestion is to use a regression algorithm instead of a classification algorithm. In classification, the labels of samples are categorial and unordered, while in regression the labels are continuous or ordered values. In our current solution, we take the probability of a sample for belonging to a certain class and the sample with the highest probability is chosen. This could be converted to a regression problem, in which the sample with the highest label value is chosen. This could result in a better performing machine learning model. However, in order to convert the current solution to a regression model, many changes need to be made. The process of generating the dataset needs to be changed completely, since a score needs to be calculated for each sample in the dataset. The better the conflict or constraint is, the higher the score of the associated sample in the dataset has to be. Therefor, a method for scoring each conflict and constraint has to be developed, that can be used as value of the label of the sample.

Bibliography

[1] Hakan Altınçay and Cem Ergün. Clustering based under-sampling for improving speaker verification decisions using adaboost. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, pages 698–706. Springer, 2004.

[2] Amedeo Cesta, Angelo Oddi, and Stephen F Smith. Profile-based algorithms to solve multiple capacitated metric scheduling problems. In *AIPS*, pages 214–223.

[3] Amedeo Cesta, Angelo Oddi, and Stephen F Smith. A constraint-based method for project scheduling with time windows. *Journal of Heuristics*, 8(1):109–136, 2002. ISSN 1381-1231.

[4] Amedeo Cesta, Angelo Oddi, Nicola Policella, and Stephen F Smith. *A Precedence Constraint Posting Approach*, pages 113–133. Springer, 2015.

[5] Nitesh V Chawla. Data mining for imbalanced datasets: An overview. In *Data mining and knowledge discovery handbook*, pages 875–886. Springer, 2009.

[6] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[7] Hédi Chtourou and Mohamed Haouari. A two-stage-priority-rule-based algorithm for robust resource-constrained project scheduling. *Computers & industrial engineering*, 55(1):183–194, 2008.

[8] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial intelligence*, 49(1-3):61–95, 1991. ISSN 0004-3702.

[9] Giovanni Di Liberto, Serdar Kadioglu, Kevin Leo, and Yuri Malitsky. Dash: Dynamic approach for switching heuristics. *European Journal of Operational Research*, 248(3):943–953, 2016.

[10] Leon Endhoven, Tomas Klos, and Cees Witteveen. Maximum flexibility and optimal decoupling in task scheduling problems. In *Proceedings of the The 2012 IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technology-Volume 02*, pages 33–37. IEEE Computer Society. ISBN 0769548806.

[11] RP Evers. Algorithms for scheduling of train maintenance. 2011.

[12] James A Hanley and Barbara J McNeil. A method of comparing the areas under receiver operating characteristic curves derived from the same cases. *Radiology*, 148(3):839–843, 1983.

[13] Sönke Hartmann and Dirk Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of operational research*, 207(1):1–14, 2010. ISSN 0377-2217.

[14] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[15] Rainer Kolisch and Arno Sprecher. Psplib-a project scheduling problem library: Or software-orsep operations research software exchange program. *European journal of operational research*, 96 (1):205–216, 1997.

[16] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques, 2007.

[17] Dirk P Kroese, Tim Brereton, Thomas Taimre, and Zdravko I Botev. Why the monte carlo method is so important today. *Wiley Interdisciplinary Reviews: Computational Statistics*, 6(6):386–392, 2014.

[18] Philippe Labone and Malik Ghallab. Planning with sharable resource constraints. In *Proceedings of the 14th international joint conference on Artificial intelligence*, volume 2, pages 1643–1649.

[19] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3): 18–22, 2002.

[20] Michele Lombardi and Michela Milano. A precedence constraint posting approach for the rcpsp with time lags and variable durations. In *International Conference on Principles and Practice of Constraint Programming*, pages 569–583. Springer.

[21] Richard Lowry. Concepts and applications of inferential statistics. 2014.

[22] Léon Robert Planken. Algorithms for simple temporal reasoning. 2013.

[23] Nicola Policella, Angelo Oddi, Stephen F Smith, and Amedeo Cesta. Generating robust partial order schedules. In *International Conference on Principles and Practice of Constraint Programming*, pages 496–511. Springer.

[24] Nicola Policella, Amedeo Cesta, Angelo Oddi, and Stephen F Smith. From precedence constraint posting to partial order schedules. *Ai Communications*, 20(3):163–180, 2007. ISSN 0921-7126.

[25] Nicola Policella, Amedeo Cesta, Angelo Oddi, and Stephen F Smith. Solve-and-robustify. *Journal of Scheduling*, 12(3):299, 2009.

[26] John W Pratt. Remarks on zeros and ties in the wilcoxon signed rank procedures. *Journal of the American Statistical Association*, 54(287):655–667, 1959.

[27] A Alan B Pritsker, Lawrence J Waiters, and Philip M Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. *Management science*, 16(1):93–108, 1969. ISSN 0025-1909.

[28] John R Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976.

[29] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006. ISBN 0080463800.

[30] Stephen F Smith and Cheng-Chung Cheng. Slack-based heuristics for constraint satisfaction scheduling. In *AAAI*, pages 139–144.

[31] JJ Staats. Improving pcp algorithms using flexibility metrics: Creating flexible schedules for technical maintenance. 2014.

[32] Michel Wilson, Cees Witteveen, Tomas Klos, and Bob Huisman. Enhancing flexibility and robustness in multi-agent task scheduling. In *Proceedings OPTMAS workshop*, 2013.

[33] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.

[34] Huiting Zheng, Jiabin Yuan, and Long Chen. Short-term load forecasting using emd-lstm neural networks with a xgboost algorithm for feature importance evaluation. *Energies*, 10(8):1168, 2017.