EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science

MASTER'S THESIS

Automatic scheduling with variable constraints

Design and development of
a generic assignment and scheduling algorithm

by
M.J. Hofman

Supervisors: Prof.Dr. E.H.L. Aarts
Ing. A. v. Wezel

Eindhoven, November 2004

Abstract

In this thesis we study and implement a method for the automatic creation of work schedules as part of a larger project mainly within the context of the Dutch public transportation process. Until now the process of assigning, called matching in technical terms, is performed by hand. Preferences submitted by the employees are hereby taken into account, i.e., the final work schedule should lie as close to the preferences as possible. Furthermore all drivers should be treated as equal as possible with respect to deviations from their preferences. In general, work schedules are also subject to certain constraints imposed by for example the government. As these constraints can change frequently we develop a method to create and use them in a variable way. This is done by using a context free language to define the constraints while using compiler techniques to evaluate them.
We use a constraint satisfaction approach algorithm for the reason that it has a good perspective of solving the problem at hand while at the same time it's concept is generic enough to be able to solve other related problems as well with only minor adjustments.
We show that the developed algorithm performs better than the manual method with respect to almost all defined requirements.

Acknowledgements

This master's thesis concludes my studies at the Department of Mathematics and Computer Science at the Eindhoven University of Technology. The research presented in this thesis was performed at the company of Ovitech in Eindhoven as part of a larger project concerning work schedules in general. In short Ovitech is a small company concerned with optimizing processes in certain parts of the public transportation sector. The abbreviation Ovitech stands for "Openbaar Vervoer Informatie Technology" which is Dutch for "Public Transport Information Technology".

First of all I want to thank Marijn Pessers for presenting me with the opportunity to perform my graduation project at Ovitech. I especially want to thank Alex van Wezel for his support, patience and advice, not only with respect to my work. Furthermore I thank the other employees from Ovitech with whom I had a lot of interaction during my time there, Harun Nabibaks, André Hebben and Bas van de Langenberg, for their readiness to answer my questions at any time and for the relaxed work atmosphere they created.
I would also like to thank my supervisor Emile Aarts for the advice and support he gave me.
And last but not least I thank my parents for the support they gave me and without whom I would not have been able to perform my study in the first place.

Melchior Hofman

Eindhoven, November 2004

Contents

ABSTRACT **2**

ACKNOWLEDGEMENTS **3**

1 INTRODUCTION **6**

1.1 BACKGROUND 6
1.1.1 Dutch Public Transportation 6
1.1.2 Motivation and objectives 7
1.2 OUTLINE OF REPORT 8

2 PROBLEM DESCRIPTION **9**

2.1 INFORMAL INTRODUCTION TO THE PRECARP MATCHING PROBLEM 9
2.1.1 Assumptions and boundaries 10
2.2 FORMAL INTRODUCTION TO THE PRECARP MATCHING PROBLEM 11
2.2.1 Input 11
2.2.2 Decision variables 12
2.2.3 Constraints 12
2.2.4 Problem definition 13

3 VARIABLE CONSTRAINTS **15**

3.1 CONSTRAINT CREATION 15
3.1.1 Context Free Grammar 15
3.1.2 Attribute Grammar 17
3.1.3 Semantics and conflicts 17
3.1.4 Constraint types 18
3.1.5 Storage 19
3.2 CONSTRAINT EVALUATION 19

4 SOLUTION APPROACH **21**

4.1 ANALYSIS OF THE PROBLEM 21
4.1.1 Computational complexity 22
4.1.2 Motivation 24
4.2 CONSTRAINT SATISFACTION 25
4.2.1 The Constraint Satisfaction Problem 25
4.2.2 Solving the Constraint Satisfaction Problem 26
4.3 SOLUTION: THE CORE ALGORITHM 28
4.3.1 Consistency checking 29
4.3.2 Variable and value selection 29
4.3.3 Dead end recovery 31
4.3.4 Pseudo code 32

5 RESULTS **40**

5.1 NNC1: A REAL-LIFE INSTANCE 40
5.1.1 Representation 40
5.1.2 Computational results 42
5.1.3 Manual matching 44
5.2 PARAMETER SETTINGS 44
5.3 THEORETICAL TEST CASES 45
5.3.1 A boundary case 45
5.3.2 'Tight' instances 46
5.4 SUMMARY 47
5.4.1 Manual versus automatic matching 47
5.4.2 Other cases 49
5.4.3 Parameter value influence 49

6 CONCLUSIONS ... 50

6.1 IMPROVEMENTS AND FUTURE RESEARCH ... 50

6.1.1 Improvements ... 50

6.1.2 Future research ... 51

6.2 GENERALIZATION OF THE SOLUTION ... 52

6.2.1 The TP-problem ... 52

6.2.2 Mapping the PCMP to the TP-problem ... 53

6.3 EVALUATION AND RECOMMENDATIONS ... 55

6.3.1 Evaluation ... 55

6.3.2 Recommendations ... 55

BIBLIOGRAPHY ... 56

SYMBOL INDEX ... 57

APPENDICES ... 59

APPENDIX A ... 60

APPENDIX B ... 63

1 Introduction

This report is part of the result of my graduation project and is concerned with the problem of constructing time-tables. More precisely the automatic construction of schedules which have to meet certain requirements. The problem is placed within the context of the Dutch public transportation process.
This chapter is organized as follows. In Section 1.1 the context of the problem is sketched in order to make the reader acquainted with the background against which the project was performed
In Section 1.2 the global outline of the report is discussed, highlighting the main themes.

1.1 Background

1.1.1 Dutch Public Transportation

As Ovitech operates in the public transportation sector the graduation project was also set in this context. To introduce the reader to this report we first sketch the Dutch public transportation sector.
In figure 1.1, which is taken from Van Wezel [2002], the complete process model is shown. The process is circular starting with “Strategy and Policy”-information leading to a clearly defined strategy for the public transportation company. Using this strategy target markets are identified after which marketing is performed; the management acts on the opportunities of the market. Then the company has to deal with competition from other public transportation companies operating in the same market for contracts which are awarded by the local governments of the different regions of the Netherlands.
When a contract has been awarded to the company it has to start planning, i.e., allocating its resources in order to fulfill the production need defined in the contract. The production need is sometimes completed with information measured during a previous iteration of the information cycle.
After planning the actual production is started during which measurements take place in order to optimize the production-costs ratio. The results of the measurements are then analyzed and if necessary the strategy is adapted and the circle starts a new iteration.

Figure 1.1 *Dutch Public Transportation, process model*

We now make the phases which are important to this report more concrete.
Mainly it concerns the “Planning” and “Execution and measuring” phases. In the planning phase the available resources of the company are allocated. For example, by contract the production need for n different regions $r_1,..,r_n$ has been identified as *y* travelers at time *t* which need to be transported to *m* other regions $r_1,..,r_m$ by bus. Now the transportation company has to draw up a line-network, assign busses to lines and assign personnel to the busses. Furthermore it has to determine the frequency a certain stop is visited.
In planning the above the company has to obey certain rules. Some of which are defined in the awarded contract and others which are stated in the collective agreement of the employees. An example of a rule defined in an awarded contract can be “between 8 AM and 9 AM the total capacity of seating-places on the line Leeuwarden – Sneek should be at least 200 in both directions”. A rule from the collective agreement may be “at least 11 hours of rest should be taken between two consecutive labor periods”. When the “Planning” phase is completed the phase “Execution and Measuring” is started in which the actual production is performed. Furthermore a number of things are measured, like the amount of people transported during a certain time interval per line.

1.1.2 Motivation and objectives

As mentioned in the previous Section, the “Planning” phase can be partitioned in a number of sequential sub-phases, i.e., drawing up a line-network, assigning busses to lines and assigning personnel to busses. The idea is to design and develop an algorithm to assist in the execution of sub-phase 3, the assignment of personnel

to busses or trains. In this sub-phase personnel concerns the drivers of the trains and/or busses of a particular location. In Chapter 2 the assignment problem of this phase is described in more detail.
When the developed algorithm for sub-phase 3 is sufficiently effective the intention is to generalize it in such a way that it can handle problems from the other sub-phases as well. The decision to start with sub-phase 3 originates from Ovitech because it has an earlier deadline. Therefore some design choices are made which should speed up development and efficiency of the algorithm towards the customer problem but on the other hand could restrict the generalization in a later stadium.

As mentioned before the algorithm alone will only provide part of the desired functionality. Besides the development of an algorithm and specifying a generalization the graduation assignment consists of the design and development of an *automated variable rule checking module* which could be used to check the rules mentioned before. Therefore the module must have the capability to check a variety of non-predefined rules hence the word *variable*. The design is described in Chapter 3.

1.2 Outline of report

This report discusses the construction of time-tables in the practical context of public transportation and is organized as follows.
After making the reader acquainted with the background of the project in Section 1.1 we first give an informal description of the problem at hand in Section 2.1. Then, in Section 2.2 this problem is modeled in a mathematical way.
The solution to the problem has to satisfy a number of constraints. The creation thereof is discussed in Section 3.1 and the way of checking the time-table with the created constraints is described in Section 3.2.
Analyzing the model from Section 2.2 in Section 4.1 we arrive at a solution method in Section 4.2. We describe this method in detail in Section 4.3.
The solution method is then put to the test to some practical problems in Section 5.1 as well as a few theoretical ones (Section 5.3). We investigate the range of the different tuning parameters and the influence on the performance of the described solution method in Section 5.2. Section 5.4 gives an overview of the results together with a comparison between results of the manual construction and the ones we achieved with the automatic construction.
In the last Chapter, 6, some remarks for the improvement of the developed solution and a hint of how the solution can be extended and used to solve other related problems is given. An evaluation of the developed product and some accompanying recommendations conclude this report.

Appendices
At the end of this report a symbol-index table is included to help the reader with the used mathematical notation. The contents of both the appendices A and B, respectively the Constraint Attribute Grammar and the Constraint Storage Database, are explained in Chapter 3.

2 Problem description

In Section 1.1 we described the context of the problem. In this chapter we discuss the problem itself.
The outline of this chapter is as follows. In Section 2.1 the problem is presented in words defining the boundaries. Also the assumptions made are stated. In Section 2.2 a formal definition is given in terms of a mathematical model.

2.1 Informal introduction to the PreCarp Matching Problem

At a Dutch public transportation company it is customary for the drivers of the trains and busses to submit their own personal preference schedule. These contain the working times at particular days during a certain predefined period. This is done per branch-location of the company, for example the branch-location of Amsterdam, Maastricht or Eindhoven.
The creation of such preference schedules is done with a stand-alone application. The schedule consists of so-called block types which are assigned to days. A block type defines the start and the end of a working period along with some additional information. Each day can at most have one block assigned.
In the preference schedule a driver can also give extra preference to block type-day combinations with a certain predefined maximum per combination as well as per schedule. Table 2.1 shows an example of such a schedule. A single cell of this schedule denotes a day of a week. The text in a cell denotes a block type and the number between brackets stands for the amount of extra preference given to that block type-day combination. Note that also empty cells can occur in the preference schedule. In such a case the assumption is that the driver does not care which block type he or she gets assigned to for that particular day. It might also be the case that the contract of the driver states that this driver does not work for the company at that day.
Furthermore drivers can only select those block types that are actually needed per day.

Table 2.1 *Example of a part of a preference schedule*

	Sunday	**Monday**	**Tuesday**	**Wednesday**	**Thursday**	**Friday**	**Saturday**
Week 15	P	B	B	FREE (10)		B	M
Week 16	N	FREE (10)	C			G	N
Week 17	FREE (20)	LEAVE (20)	LEAVE (20)	LEAVE (20)		B	M
Week 18	L	B	B (15)				N

When the drivers have completed filling in their final preference schedule they send it to the branch-location they belong to. At this point the schedules fulfill all the posed rules. At each location a number of planners collect all the schedules and try to put together a so-called scenario with the aid of an application called PreCarp.
A scenario is a collection of possibly modified preference schedules which taken together fulfill the production need of a branch-location. The production need is stated in terms of block types and the amount of them needed per day. A graphical representation of a possible production need for a period of 2 weeks is shown in table 2.2.

Table 2.2 *Example of the production need for a location*

	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Week 15	P, 2 M, 4 L, 8 N, 3	B, 13 G, 3 O, 5 P, 7	B, 13 G, 3 O, 5 P, 7	B, 13 G, 3 O, 5 P, 7	B, 13 G, 3 O, 5 P, 7	B, 13 G, 3 O, 5 P, 7	L, 12 K, 4 M, 10 N, 13
Week 16	K, 2 P, 4 L, 8 N, 3	B, 13 G, 3 O, 5 P, 7	B, 5 G, 1 O, 3 P, 4 L, 2	B, 5 G, 1 O, 3 P, 4 L, 2	B, 5 G, 1 O, 3 P, 4 L, 2	B, 5 G, 1 O, 3 P, 4 L, 2	L, 12 K, 4 M, 10 N, 13

We shall refer to the problem of creating a scenario as the PreCarp Matching Problem (PCMP). We sketch this problem in more detail and in doing so arrive at the actual problem description.
When receiving a lot of different preference schedules the planners probably need to modify these schedules in order to meet the production need. In the ideal case that the production need is met nothing has to be done but in practice this situation will probably never be encountered. But by modifying the schedules one can create schedules which conflict with the posed rules. Besides one also has to take into account the extra preference given to some cells. Furthermore all drivers should retain an equal percentage of their preference schedule.
Creating a scenario by hand with all of the above elements accounted for has been proven to take a lot of time. Creation times in the order of several weeks are no exception. Also the final results have been bad with respect to the drivers being treated equally. This was one of the reasons to request automation of this process. Also, multiple different scenario's to choose from were requested.

Another closely related problem is the creation and checking of rules which are not predefined. With a tool for handling this problem planners but also drivers could create their own rule-sets and adapt them to changing collective agreements and different driver contracts. This tool was mentioned in the introduction as the automated variable rule checking (CC) module.
The concept of rules will be renamed to constraints for the remainder of this report.

Designing the language for the constraints, designing and developing the CC-module as well as an algorithm to solve the PCMP are part of the problem description. The complete description also encompasses a short description of how to generalize the developed algorithm. The idea of this part is to be able to handle other (public transport) related problems as well.

2.1.1 Assumptions and boundaries

In our timetable construction problem some decisions are already given or left out of the model. We assume that:

- for each day the production need is known;
- we look at a single branch-location and a single matching period;
- for each driver the block types he or she can be assigned to are known per day;
- the production need has to be met every day;
- the output needs to consist of multiple, preferably different scenario's or they should be different with each run of the algorithm;
- a driver belongs to only one branch-location per period;
- the program should deliver an answer within a period of at most two hours.

2.2 Formal introduction to the PreCarp Matching Problem

In the previous Section we described the concepts and their relationships of the PCMP. In this Section we define the problem in a formal way.
Before we give the definition of an instance of the PCMP we define the elements of which it consists. In the remainder we use italics to distinguish functions from other concepts.

2.2.1 Input

First we define the input to the problem.

Definition 2.1 (Production need input). The four-tuple pN = ⟨ D, PB, *tb*, *cb* ⟩ defines the production need, where

- D denotes a finite set of days which make up the period;
- PB denotes a finite set of all possible block types available to the branch-location;
- *tb* : D → $\mathcal{P}$(PB), where *tb*(d) denotes the available block types for a day d ∈ D;
- *cb* : PB × D → ℕ, where *cb*(pb, d) denotes the cardinality of block type pb ∈ PB at day d ∈ D;
- *prev* : D → D, where *prev*(d) denotes the day which is regarded as 'yesterday' from the perspective of day d ∈ D. In the remainder we sometimes use indices to denote this function, i.e., $prev(d_i) := d_{i-1}$;
- *next* : D → D, where *next*(d) denotes the day which is regarded as 'tomorrow' from the perspective of day d ∈ D. In the remainder we sometimes use indices to denote this function, i.e., $next(d_i) := d_{i+1}$.

□

Definition 2.2 (Block input). The five-tuple Bi = ⟨ PB, *bbt, bet*, *bw, bbw* ⟩ defines the block input, where

- PB denotes a finite set of all possible block types available to the branch-location;
- *bbt* : PB → ℕ, where *bbt*(pb) denotes the start-time of block type pb ∈ PB;
- *bet* : PB → ℕ, where *bet*(pb) denotes the end-time of block type pb ∈ PB;
- *bbt*(pb) ≤ *bet*(pb), ∀pb ∈ PB;
- *working period* : PB → ℕ, where *working period*(pb) := *bet*(pb) – *bbt*(pb) ;
- *bw* : PB → $\mathscr{B}$, where *bw*(pb) denotes the fact whether block type pb ∈ PB is considered a so-called work block;
- *bbw* : PB → $\mathbb{N}^+$, where *bbw*(pb) denotes the so-called bonus value of block type pb ∈ PB.

□

Definition 2.3 (Driver input). The four-tuple Di = ⟨ CH, D, *pref, expref*, *ab* ⟩ defines the driver input, where

- CH denotes a finite set of drivers belonging to the branch-location;
- D denotes a finite set of days which make up the period;
- *pref* : CH × D → PB, where *pref*(ch, d) denotes the preference block type chosen by driver ch ∈ CH for day d ∈ D;
- *expref* : CH × D → ℕ, where *expref*(ch, d) denotes the possible extra preference driver ch ∈ CH has assigned to day d ∈ D. If no extra preference has been assigned a default value of 1 is used, i.e., *expref*(ch, d) := 1 ;

- *ab* : $CH \times D \rightarrow \mathcal{P}(PB)$, the ability-function, where *ab*(ch, d) denotes the possible block types a driver ch ∈ CH can be assigned to at day d ∈ D.

□

2.2.2 Decision variables

In this Section we show that a solution to the PCMP consists of two assignments: the block assignment and the cost assignment.
Not all block assignments to drivers are relevant. In order to properly define the block assignment we first define the assignments which are relevant.

Definition 2.4 (Relevant assignment).
Let P be the set of all relevant (block, driver, day) combinations, that is,

$$P := \{ (pb, ch, d) \in PB \times CH \times D \mid pb \in (tb(d) \cup ab(ch, d)) \}$$

In this definition 'relevant' means that the block to assign has to be part of the production need for that day. Also the block the driver is assigned to has to be part of the driver's abilities for that day.

□

Definition 2.5 (Block assignment). The assignment of blocks to drivers is a mapping

$m : (PB \times CH \times D) \rightarrow T,$ with $T \subseteq P$

furthermore we define the assignment function as follows:

as : D × CH → PB, where *as*(d, ch) gives the block type driver ch ∈ CH is assigned to for day d ∈ D.

Members of set T are called matchings.

□

Definition 2.6 (Cost assignment). The cost assignment contains assignments of costs to drivers. In order to meet the requirement of equally dividing the deviations from the preference schedule between drivers we introduce the concept of penalties. These penalties are then incurred whenever a preference block type-day combination is changed. The costs are proportional to the penalties incurred. Also costs are assigned when a so-called soft-constraint (see Section 2.2.3) is violated for a particular matching. Therefore the cost assignment is a function:

$c : T \rightarrow \mathbb{R}^+$, where *c*((pb, ch, d)) denotes the total costs associated with the assignment of block pb ∈ PB to driver ch ∈ CH for day d ∈ D.

□

2.2.3 Constraints

Not all solutions are acceptable. As mentioned before the constraints posed in the collective agreement amongst others have to be met for every driver. Also a driver's contract can contain certain restrictions with respect to working times and/or periods. But not all constraints have to be met per se. To make this separation more clear we distinguish between hard constraints which have to be met for all solutions and soft constraints which may be violated at some cost.

Definition 2.7 (Constraint types). We define the following sets:

- HC as the set of hard constraints applicable to all drivers;
- HC_{ch} as the set of hard constraints applicable to driver ch $\in$ CH only;
- SC as the set of soft constraints applicable to all drivers;
- SC_{ch} as the set of soft constraints applicable to driver ch $\in$ CH only;

furthermore we have the following relations, $\forall ch \in CH$:

- $A := HC \bigcup SC$
- $A_{ch} := HC_{ch} \bigcup SC_{ch}$
- $TC := A \bigcup A_{ch}$,
- $HC \bigcap HC_{ch} = Ø$
- $SC \bigcap SC_{ch} = Ø$
- $HC \bigcap SC = Ø$
- $HC_{ch} \bigcap SC_{ch} = Ø$

and, as a result:

- $A \bigcap A_{ch} = Ø$

□

A constraint tc $\in$ TC always concerns the question: "can a particular block type-day combination be made on this day for this driver with respect to the other block type-day combinations already assigned?"
To express this we define the predicate *Satisfies*.

Definition 2.8 (Satisfiability). The predicate *Satisfies* is defined as:

- *Satisfies*(C, pb, ch, d): ⇔ "the assignment of driver ch $\in$ CH on day d $\in$ D to block pb $\in$ (*tb*(d) $\bigcup$ *ab*(cd, d)) satisfies all constraints in set C, with $C \subseteq (HC \bigcup SC \bigcup HC_{ch} \bigcup SC_{ch})$"

□

2.2.4 Problem definition

In this Section we formally define an instance of the PCMP and a PCMP-solution. Also we look at the initial feasibility of the problem.

Definition 2.9 (PCMP-instance). An instance of the PCMP is a four-tuple PCMP = ⟨ pN, Bi, Di, Co ⟩ with

- pN the production need input, see definition 2.1;
- Bi the block input, see definition 2.2;
- Di the driver input, see definition 2.3;
- $Co \subseteq TC$ the set of relevant constraints for this instance.

□

Definition 2.10 (PCMP-solution). A solution of a PCMP-instance, called a scenario, consists of a block assignment *m*, see definition 2.5, for every driver ch ∈ CH for every day d ∈ D, such that:

1. for every day at least the production need is fulfilled, i.e.,
   $\forall_{d \in D} : \forall_{pb \in tb(d)} : (\#_{ch \in CH} : (m(pb, ch, d) \in T)) \geq cb(pb, d)$;
2. for every day each driver has exactly one block type assigned to him/her, i.e.,
   $\forall_{d \in D} : \forall_{ch \in CH} : (\exists_{pb \in PB} : m(pb, ch, d) \in T \wedge (\forall_{pb2 \in PB \wedge pb2 \neq pb} : m(pb2, ch, d) \notin T))$;
3. a scenario satisfies all constraints in the set Co, i.e.,
   $\forall_{d \in D} : \forall_{ch \in CH} : \mathit{Satisfies}(Co, as(d, ch), ch, d)$;
4. all the drivers are treated as equal as possible within a certain margin, i.e.,
   $\forall_{ch \in CH} : L \leq (\sum_{d \in D} c((as(d, ch), ch, d))) \leq U$, with $L, U \in \mathbb{N}$.

We refer to sub solutions which satisfy only part of the requirements of the PCMP-solution as PCMP($x_1,..,x_n$), where $x_i : 1 \leq i \leq n$, with n = 4, denotes a requirement which has been fulfilled in the solution.

□

Informally the last requirement states that the cumulative penalty for every driver should lie on or between the lower bound L and the upper bound U. The closer U – L is to zero the more equal the drivers have been 'treated'.

However some instances cannot be solved even with an empty constraint set Co. This can happen when the number of blocks needed from a particular type on a certain day is larger than the number of drivers with an ability for that block type at that day. The existence of this kind of situation can be determined from the input data and is therefore called the *initial* feasibility detection. We perform this check before starting the matching algorithm itself because otherwise it will possibly run for a very long time and still return no answer.

Definition 2.11 (Initial feasibility). An instance of the PCMP is initially feasible if and only if for every day the summed cardinality of every block type needed per day is less than or equal to the number of different drivers whom have these blocks as (part of) their ability. In essence this means that a particular branch-location should have enough drivers with the right skills to meet its production need every day. The predicate *InitFeasible* is defined as

- *InitFeasible* : $\Leftrightarrow \forall_{d \in D} : \forall_{Y \in \mathcal{P}(tb(d))} : \sum_{q \in y \in Y} cb(q, d) \leq |\{ i \mid i \in CH \wedge q \in ab(i, d) \}|$

□

3 Variable Constraints

In this chapter we discuss several aspects related to the constraints. Section 3.1 describes the creation of the constraints from the definition of the grammar used to express them to the model of the database used to store them.
Section 3.2 is concerned with the checking procedure of the created constraints. As the constraints are not predefined this checking is performed with a separate module built on compiler techniques.

3.1 Constraint Creation

As mentioned before constraints are used to verify whether a matching for a particular driver can be made with respect to other matchings made earlier for this driver. Also, constraints can be created that apply only to one matching. An example of a constraint with respect to more than one matching is the collective agreement requirements of a resting period of eleven consecutive hours between two consecutive workblock types.
As all regulations can change from time to time it is desirable to be able to create new constraints and delete or update old ones. Because such an ability was also conform the requirements of developing a general module it was further developed and implemented. In this Section we describe the design of the constraint syntax and the way of storing the constraints efficiently in a modular fashion.

3.1.1 Context Free Grammar

To express the constraints a language is needed. We define the constraint language (CL) by a context free grammar (CFG) because of the large expression power. Also the fact that the constraints need to be conditional played a part, i.e., conditional constraints have a similarity with if-statements in programming languages and these are mostly defined by a context free grammar.
Before we present the used grammar we first give some definitions of the used notions.

Definition 3.1 (Grammar). A grammar G is defined as a four-tuple, $G = \langle V, T, S, P \rangle$, where

- V denotes a finite set of objects called variables;
- T denotes a finite set of objects called terminal symbols;
- $S \in V$ is a special symbol called the start variable;
- P denotes a finite set of productions.
- $V \cap T = \emptyset$
- $V \neq \emptyset$
- $T \neq \emptyset$

□

Definition 3.2 (Context Free Grammar). A grammar $G = \langle V, T, S, P \rangle$ is said to be context-free if all productions in P have the form

$A \rightarrow x$, where $A \in V$ and $x \in (V \cup T)^*$.

□

So, in a sentence x of the grammar the substitution of the variable on the left of a production can be done independently of the symbols in the rest of the sentence or context for that matter.

We use the Extended Backus-Naur Form (EBNF) [Linz, 1997] to specify the productions of the CL. The EBNF consists of a collection of productions that describe the formation of strings or sentences in the language. Each production consists of a non-terminal symbol and an EBNF expression separated by a 'is

defined to be' sign and terminated with a period defining the non-terminal. The strings that can be generated consist of terminals. An EBNF expression is composed of zero or more terminal symbols and non-terminal symbols. All the strings in the language can only be generated from a non-terminal, called a start symbol. Formally: $L(G) = \{ x \in T^* \mid S \Rightarrow^* x \}$. Here =>* indicates that an unspecified number of steps (including zero) can be taken to derive x from S. T* stands for the set of all possible orderings of elements in T.
Before we give the definition of the constraint language we first list the special symbols used in table 3.1.

Table 3.1 *Special non-terminals in EBNF*

Symbol	Meaning
:=	is defined to be
|	alternatively
.	end of production
{ X }	0 or more instances of X
"xyz"	the terminal symbol xyz

Definition 3.3 (Constraint Language). The Constraint Language (CL) is defined by the following CFG, with

- V = { CONSTRAINT, STAT, EXPR, EX_1, EX_2, EX_3, EX_4, EX_5, EX_6, EX_7, EX_8, DENOT, VAR, DATE, SET, INTERVAL, SPAN, PREDEF, OP_1, OP_2, OP_3, OP_4, OP_5, OP_6, OP_7 };
- T = { if, then, (,), /, {, }, ;, COUNT, IN, RUST, ID };
- S = CONSTRAINT;
- P with the following elements and $1 \le i < 7$:

CONSTRAINT := STAT "~".

STAT := "if" EXPR "then" EXPR.

EXPR := EX_1.
EX_i := EX_{i+1} { OP_i EX_{i+1} }.
EX_7 := OP_7 EX_7 | EX_8.
EX_8 := DENOT | VAR | "(" EX_1 ")" | PREDEF.

DENOT := NUMBER | STRING | DATE.

VAR := ID.

DATE := "/" NUMBER "/" NUMBER "/" NUMBER.

SET := "{" DENOT { ";" DENOT } "}".

INTERVAL := "[" NUMBER ";" NUMBER "]" | "[" DATE ";" DATE "]".

SPAN := "/" NUMBER.

PREDEF := IN DENOT SET | IN VAR SET |
COUNT INTERVAL EXPR | COUNT SPAN EXPR |
RUST INTERVAL | RUST SPAN.

OP_1 := "==".
OP_2 := "or".
OP_3 := "and".
OP_4 := "<" | "<=" | ">=" | ">" | "=".
OP_5 := "+" | "-".
OP_6 := "*".
OP_7 := "-" | "not".

□

3.1.2 Attribute Grammar

Of course constraints have to be well typed. The AND-operator for example is a Boolean operator and cannot work on integer values. This kind of limitations are expressed by means of an attribute grammar.

Definition 3.4 (Attribute Grammar). An attribute grammar (AG) is a six-tuple AG = $\langle$ N, DOM, $\sum$, P, RC, S $\rangle$ where

- N denotes a finite set of objects called non-terminals;
- DOM = { dom_A | A $\in$ N }, where each dom_A is a set. dom_A will be called the attribute domain associated with non-terminal A;
- $\sum$ denotes a finite set of objects called terminals;
- P denotes a finite subset of N × (N $\bigcup \sum$)*, an element of P will be called a production rule scheme;
- RC = { C_p | p $\in$ P }, where p is a production rule scheme $A_0 \rightarrow x_0A_1x_1...A_nx_n$ ($x_i \in \sum^*$, $A_i \in$ N), and C_p is a function: $C_p : dom_{A0} \times dom_{A1} \times ... \times dom_{An} \rightarrow \mathscr{B}$
  C_p is called the rule condition associated with production rule p;
- S $\in$ N, S is called the start symbol.

□

For convenience and clarity we will use the notation used by Ten Eikelder, Van Geldrop, Hemerik & Zwaan [1999] to express the attribute grammar. In essence this means that when writing the set of non-terminals we also give the corresponding attribute domains, i.e., we write for instance {A(dom_A), B(dom_B), ...}, if A, B, ... are non-terminals with attribute domains dom_A, dom_B, ... respectively.
Also a production rule scheme

$A_0 \rightarrow x_0A_1x_1...A_nx_n$

and the corresponding rule condition

$C_p : dom_{A0} \times dom_{A1} \times ... \times dom_{An} \rightarrow \mathscr{B}$

will be written as the 'guarded production rule'

$A_0(v_0) \rightarrow x_0A_1(v_1)x_1...A_n(v_n)x_n$,
$C_p(v_0,v_1,v_2) \equiv$ true

Before the dummy variables v_x we put either a '+' or a '-' denoting the fact that the variable is either synthesized by the (non)-terminal or inherited by the (non)-terminal.

In Appendix A we give a constraint attribute grammar (CAG) for the constraints with the CFG of definition 3.3 as its underlying CFG.

Definition 3.5 (Constraint).
A constraint cs is a string derived from the start symbol CONSTRAINT. This string cs acts as a function *cs*.

$cs : T \rightarrow \mathscr{B}$. A matching t $\in$ T satisfies a constraint cs if $cs(t) \equiv$ true.

□

3.1.3 Semantics and conflicts

For the larger part the semantics of the grammar is trivial. In cases where the semantics are not considered to be trivial we describe them in table 3.2 in an informal way.

Table 3.2 *Non-trivial semantics of the CAG*

Non-terminal	Terminal	Value	Semantics
DATE	NUMBER	v1	**D**ay of the month: DD
DATE	NUMBER	v2	**M**onth of year: MM
DATE	NUMBER	v3	**Y**ear: YYYY
INTERVAL	NUMBER	v1	day-type of the week: [0;6] with 0 = Sunday
INTERVAL	NUMBER	v2	day-type of the week: [0;6] with 0 = Sunday
SPAN	NUMBER	v	number of days: $\mathbb{N}^+$
PREDEF	RUST	v_1	total consecutive resting time in minutes over days defined in INTERVAL or SPAN
PREDEF	COUNT	v_1	the number of times expression EXPR is true at the days defined in INTERVAL or SPAN
PREDEF	IN	v_1	indication whether the value v_2 is an element of SET

As can be seen all constraints are conditional constraints. But by simply substituting the denotation 'true' for the first 'EXPR' also unconditional constraints can be created.
Furthermore we use pre-declared identifiers which means that the elements in the Nametypebag are predefined.
To introduce the concept of conflicts we first look at some examples of different constraints.
As mentioned before constraints can restrict the number of different feasible matchings. For example suppose a certain driver can only work for three days in a week. Then assuming the driver's identification number is 10, (chauffeurID, inttype) is an element of Nametypebag and 'inzet' is used to denote whether the block is a work block this would be expressed as:

c1. if chauffeurID = 10 then (COUNT [0;6] inzet) =< 3 ~

Or, a constraint for all drivers which states that at least eleven hours of rest should be taken between two consecutive work blocks could look like:

c2. if true then RUST /2 => 11*60 ~

Now to see when conflicts can occur we look at the constraints c3 and c4 defining the possible block types for a certain day or day-type respectively.

c3. if datum = /24/12/2004 then IN blok { B; D; E } ~

c4. if dag = 1 then IN blok { A; C } ~

Constraint c3 says that at 24-12-2004 only block types B, D and E are valid while constraint c4 says that on days of type 1 (Monday) only block types A and C are valid. In the case that 24-12-2004 is a Monday the constraints have a conflict.
For now we do not guard against possible conflicts. It is up to the user to make constraints to do not create a conflict in the sets in which they are used.
In Section 6.1 we define the notion of conflicts in a formal way and propose a way of handling them.

3.1.4 Constraint types

Constraints can be of only two types: hard or soft. The distinction is made to give the user a tool to indicate preference with respect to more than just a single matching; the soft constraints.
Hard constraints should always be satisfied whereas soft constraints may be violated at some predefined cost. This cost is the same for all soft constraints and will be denoted by 'softPen'.

3.1.5 Storage

The constraints are stored in a database which we designed specifically for this purpose. The idea was to store the constraints in a modular fashion more or less derived from the syntax of the constraint language.
A model of the database is shown in Appendix B. In this database all possible newly created constraints can be stored in an efficient manner as components incident in more than one constraint need only be stored once. Furthermore updating old constraints is efficient as the constraints don't need to be parsed first to identify the different components which would be the case if the constraints were stored as single text strings. These components can be identified by their fieldname in combination with the table name. Of course special care has to be taken when creating or modifying constraints which incorporate any shared components. Exactly how this is to be done falls beyond the scope of this thesis.

3.2 Constraint Evaluation

The evaluation of the created constraints is performed by the CC-module. This evaluation is done by means of parsing the constraints and evaluating the variables on-the-fly. Parsing is a technique for explaining sentences through its grammatical derivation describing its structure [Linz, 1997].
To parse sentences from a CFG in linear time in the length of the sentence we need the CFG to be deterministic. The CFG of our constraint language, defined in definition 3.3 is deterministic. This is indicated by the fact that in any step of the derivation we can determine the next derivation-step by only looking only at the next input-symbol.
According to Linz [1997] such a grammar is called an LL1 grammar. This kind of grammar can be parsed efficiently by LL-parsers. In the abbreviation LL the first L denotes the fact that the input is scanned from the left to the right. The second L states that left-most derivations are constructed. LL-parsers allow for linear-time parsing.
The 1 denotes the one-symbol lookahead which is used to choose the right production rule at anytime during parsing. Formally: the one-symbol lookahead-sets of the alternatives of each non-terminal are mutually disjunctive.
Furthermore the CAG from Appendix A is one-pass. Informally this means that the input symbols need only be traversed once. Definition 3.6 formally defines this condition.

Definition 3.6 (One-pass condition).
For each production rule scheme

$$A{<}{-}i_0,{+}s_0{>} \quad \rightarrow x_0 A_1{<}{-}i_1,{+}s_1{>}\ldots A_n{<}{-}i_n,{+}s_n{>}x_n$$
$$s_0, i_1,\ldots,i_n \ : \ C(i_0,\ldots,i_n,s_0,\ldots,s_n)$$

and for all $k : 1 \leq k \leq n$, i_k depends on $i_0, s_1,\ldots,s_{k-1}$ only.

□

These conditions make that the evaluation procedure of the constraints will run as efficiently as possible [9] with respect to the needed functionality.

The language is parsed with a recursive descent parser, following the manner used in Ten Eikelder, Van Geldrop, Hemerik & Zwaan [1999]. In essence we create a procedure for every non-terminal with a procedure body in the form of an if..fi-statement with one alternative per production rule for that non-terminal.
As the one-pass condition holds for the production rule scheme we can both perform the verification of context conditions and the evaluation of the constraint while parsing it, i.e., 'on-the-fly'.
Evaluation boils down to assigning the right values to the variables in the constraint and then evaluating the operators with the relevant operands in the predetermined order of priority. The priorities can be derived from the CFG. Operators with a high suffix number, for example the 'not'-operator which has suffix number 7 has the highest priority indicating that it binds its operands tightest. When two or more operators have the same priority the precedence is left-to-right.

In figure 3.1 we give an example of a somewhat simplified parse tree for the evaluation of constraint c2.
The oval shapes denote the non-terminals from the CFG from definition 3.3. The diamond shapes are used to denote the terminals from the CFG.
The simplification consists of omitting the parts of the tree which occur between the EX non-terminals and the terminals.
When traversing the tree in a left-to-right fashion while remembering the encountered terminals one can reconstruct the original constraint. As can be seen at the depth of occurrence in the tree the 'at least' operator has a higher precedence than the 'times' operator. Furthermore, in this case, the 'times' operator has to evaluate its operands before the 'at least' operator can evaluate its operands. Note that this is taken care of by the parser which traverses the tree from left to right. In doing so it thus encounters first the terminals 11 and 60 then evaluates the 'times' operand after which it has to evaluate the 'at least' operand.

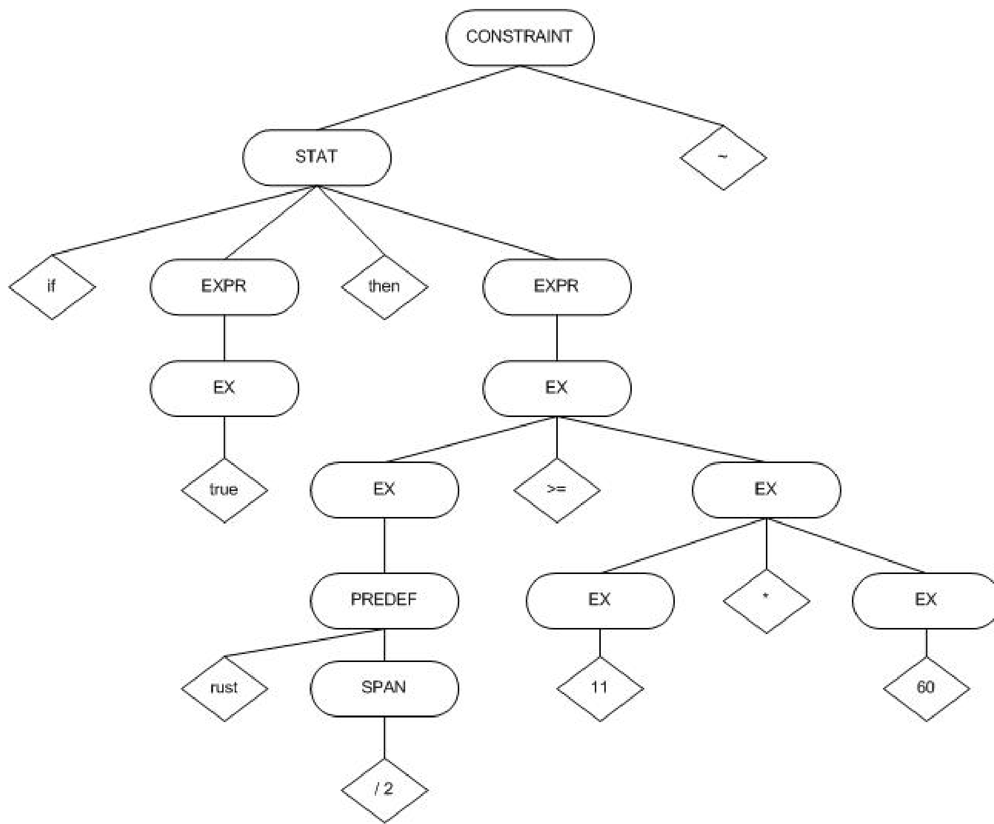

Figure 3.1 *Simplified parse tree of constraint c1*

4 Solution Approach

In Chapter 2 we presented a model for the problem at hand. This model is the starting point for selecting a solution method.
In this chapter we present the chosen solution method. In Section 4.1 the model is analyzed in order to make a motivated choice between several methods which seem applicable to problems of the presented form. Section 4.2 discusses the chosen solution method, a constraint satisfaction approach (CSA) in its general form. In Section 4.3 we describe the workings of the CSA-algorithm we implemented.

4.1 Analysis of the problem

To guide the search for a suitable algorithm to solve a PCMP-instance (definition 2.9) we first analyze the problem in more detail.
Looking at the PCMP-solution (definition 2.10) we observe that the core of the solution consists of a block assignment *m* (definition 2.5). This function *m* maps a block type to a driver for a certain day. In essence this can be seen as the problem of finding a maximum matching in a bipartite graph G = <B, D, E> with vertex sets B and D and edge set $E \subseteq B \times T$. Each day d corresponds to a new graph G_d. The set B then consists of the block types which have to be assigned on that day d given by the function *tb* (definition 2.1). When multiple blocks of the same type, say pb, have to be a assigned, i.e. *cb*(pb,d) > 1, we enumerate the individual blocks with x: $0 \leq x <$ *cb*(pb,d). The set T consists of all drivers available to the branch-location under consideration.

The only requirement which needs to be satisfied for this problem to be feasible is that $|B| \leq |T|$ for every G_d. But in our case not all drivers can be assigned to all block types. This restriction is indicated by the ability function *ab* (definition 2.3). Now the problem can still be solved by a maximum bipartite matching algorithm [Blokhuis, 1998] but the feasibility requirement is more complicated (recall: 'initial feasibility' from definition 2.11). Unfortunately the restrictions imposed can go further. Up till now all days $d \in D$, or graphs G_d were considered independent. But with the variable constraints defined in chapter 3 (definition 3.5) it is possible to influence the availability of drivers on a day $d_i \in D$ depending on the matchings made on days $d_j \in D$, $j \neq i$, for that driver. For example when for a driver a certain resting period needs to be taken between two consecutive work blocks on days d_i and d_{i+1} assigning a certain block to day d_i impacts the block types which can be chosen on d_{i+1} for that driver. So the graphs G_d are now dependent and detecting infeasibility can only be done dynamically; while running the solution algorithm. Figure 4.1 shows a possible graph G_d.

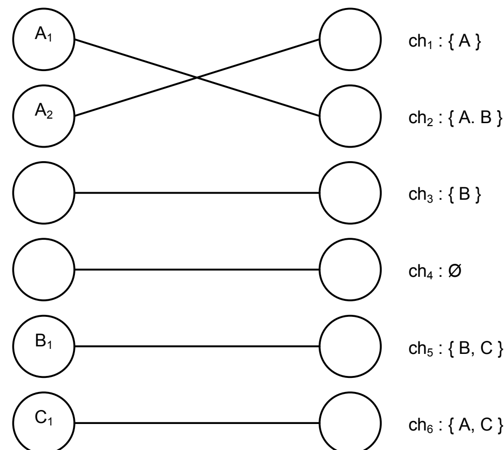

Figure 4.1 *Representation of a graph G_d*

The used notation is interpreted as follows. The nodes on the left are elements of the set B:
X_i denotes block type X with sequence number i. Nodes without indentation are called 'dummyblocktypes' and can be seen as idle-blocks. These are needed whenever |B| < |T|.
The objects or nodes on the right are elements of set T:
Y_j : Z denotes driver with index j with abilities $z \in Z$ for day $d \in D$.
Note that a driver need not have abilities for every day $d \in D$.
Now, the interdependency between days is best expressed as:
let $M \subseteq D$ be the subset of days that have already been matched, $ch \in CH$ and $pd \in PB$, then block assignment m(pb, ch, d) with $d \notin M$ can in the worst case depend on m(pb, ch, d), $\forall_{d \in M}$.

The described problem looks very much like a constraint satisfaction problem (CSP) in which variables have to be assigned a value from their domain such that all constraints are satisfied. The constraints then pose limits to the combinations of variables and values. For a formal definition of CSP we refer to definition 4.5 in Section 4.2

4.1.1 Computational complexity

In this Section we show that the PCMP is NP-complete by giving a reduction from the SAT-problem, for which it is known to be NP-complete [Garey & Johnson, 1979]. We first give a short definition of the SAT-problem.

Definition 4.1 (Satisfiability Problem). The satisfiability problem, or SAT problem for short is defined as follows. We consider Boolean formulae in conjunctive normal form (CNF), that is formulae of the type:

$$\phi(x_1,...,x_n) = \bigwedge_{k=1}^{m} \left(\bigvee_{i \in P_k} x_i \bigvee_{i \in N_k} \overline{x}_i \right)$$

where $x_1,...,x_n$ are Boolean variables, $\overline{x}_1,...,\overline{x}_n$ are their complements, and $P_1,...,P_m$, $N_1,..., N_m$ are subsets of $\{1,...,n\}$ satisfying $P_k \cap N_k = \emptyset$ for $k \in [m] = \{1,...,m\}$. Each of the terms $C_k = \bigvee_{i \in P_k} x_i \bigvee_{i \in N_k} \overline{x}_i$ for $k \in [m]$ is called a clause of ϕ, and the quantity $| P_k \cup N_k |$ is called the degree of the clause. The degree of ϕ is the maximum degree of its clauses. The size of ϕ, denoted $|\phi|$ is the sum of the degree of its clauses. The satisfiability problem for ϕ is to determine whether or not ϕ is satisfiable, i.e., whether or not there is an assignment $(a_1,...,a_n) \in \{0,1\}^n$ to the variables such that:

$$\phi(a_1,...,a_n) = 1$$

□

The satisfiability problem is well-known to be NP-complete, even when the input is restricted to the class of degree 3 CNFs. It is solvable in polynomial (linear) time in $|\phi|$ when ϕ is restricted to belong to the class of degree 2, or quadratic CNFs and when ϕ is restricted to the class of Horn CNFs, i.e., to CNFs which satisfy $|P_k| \leq 1$ for all $k \in [m]$.

Theorem 4.1 The PCMP is NP-complete.
Proof: the decision variant of PCMP is in NP as for a given scenario s all constraints can be checked in polynomial time (Section 3.2). PCMP is NP-complete as $SAT \propto PCMP$, i.e., SAT reduces in polynomial time to the PCMP.

In short we show that each block type assignment can be represented as a clause consisting of a variable per driver indicating whether the driver has been assigned to that block type. Extra clauses are added to enforce only one driver per block. These are created by means of the XOR-function.

Let:

- n = | CH |, the number of different drivers, $n \in \mathbb{N}^+$;
- $\forall_{d \in D \wedge pb \in tb(d) \wedge 0 < i \leq cb(pb,d)}$ the string st = d ++ pb_i denote a variable ;
- V the set of all possible strings st ;
- st.day := d ;
- st.block := pb ;
- st.nr := index ;
- *ast* : (D × CH) → (PB × ℕ), where *ast*(d,ch) = (pb, index) denotes the fact that ch is the index-th driver to have been assigned to block type pb on day d ;

Furthermore, if $\exists_{d \in D}$: $n > \sum_{pb \in tb(d)} cb(pb, d)$ we add dummy block types to this day's production need. Each driver is able to have this block type assigned. Formally, with $\forall_{d \in D}$:

let: dummytype ∈ PB

$g(d) := n - \sum_{pb \in tb(d)} cb(pb, d)$

then $g(d) > 0 : tb(d) := tb(d) \bigcup \text{dummytype} \wedge cb(\text{dummytype}, d) = g(d)$

Now the reduction is as follows:
Let:

- U be a set of Boolean variables, U = {u_1,...,u_n}, with |U| = |CH| = n;
- C be a collection of clauses: C = { c_1, ..., c_p, c_{p+1}, ..., c_m }, with $p = \sum_{d \in D} \sum_{pb \in tb(d)} cb(pb, d)$;

the clauses are constructed in the following way:
for every day d ∈ D we introduce z clauses $c_{d,1}$,...,$c_{d,z}$ with $z = \sum_{pb \in tb(d)} cb(pb, d)$.
Each clause is associated with a different string st ∈ V ; $c_{q,j} \sim st$, with
st.day = q, for $1 \leq j \leq z$
Now each clause $c_{q,j}$ consists of n literals $u_{q,j,i}$ numbered consecutively with i : $1 \leq i \leq n$
Each literal $u_{q,j,i}$ denotes whether $ast(q, ch_i) = (pb, j)$ is true. Here, q = st.day, pb = st.block and j = st.nr.

We now have that a clause represents a certain block which has to be assigned to a driver on a day. A literal $u_{q,j,i}$ is said to be true if the block is assigned to the corresponding driver i.

An example of a clause is the following, where
[x = y] = 1 if x = y
[x = y] = 0 if x ≠ y
using Knuth's notation. Furthermore, let $c_{q,j,i} = u_{q,j,1} \vee ... \vee u_{q,j,n}$ and q = 5 and $c_{5,j} \sim 5E_3$ then
$u_{5,j,i} = [\, ast(5, ch_i) = (E, 3)\,]$

If a clause is true, at least one driver is matched to the corresponding block. We now need to add extra clauses c_{p+1},...,c_m to express the fact that a driver can only have one block assigned to him per day.
So for every day we have n different *variables*, i.e., assigned drivers, which need to be pair-wise disjunct.
This boils down to |D|*n Boolean expressions denoting the exclusive or, XOR, ⊗ , of $u_{q,j,i}$ with $1 \leq j \leq z$, and $1 \leq i \leq n$.
Formally:

$$\forall d \in D : \forall i : (\otimes j : u_{q,j,i})$$

Each of these Boolean expression can be easily transformed in their corresponding CNF [Lewis & Papadimitriou, 1981] yielding the desired clauses. At this point a satisfying truth assignment for C corresponds with a solution for PCMP(1) and vice versa. For clarity's sake we also show how this can be extended to a solution for PCMP(1,2). Note that this extra step is not necessary to prove NP-completeness of the PCMP as a whole.

Another restriction is expressed in the PCMP through the ability function (definition 2.3). The reduction to the SAT-problem corresponds to deleting the *proper* literals from the *proper* clauses. We illustrate this with an example before giving a formal definition of the *proper* literals and clauses. With $ch_1 \in CH$ and $d_1 \in D$, suppose $ab(ch_1, d_1) = \{ A, B, C, dummytype \}$ then the driver with index 1 can only be assigned to block types A, B or C on the day with index 1. So an assignment to any other block type "doesn't count". Therefore the literals corresponding with driver 1 on day 1 from the clauses corresponding with all block types other than A, B, C or dummytpe should be deleted from the XOR-expression for day 1 prior to transforming the expression in CNF.
Formally:
let $\forall ch_i \in CH \wedge d \in D : ab(ch_i, d) = L_{i,d}$
then $W_{i,d} = (tb(d) \setminus L_{i,d})$, gives the set of block types driver i cannot be assigned to on day d.
Then from the clauses $c_{q,j}$ with $c_{q,j} \sim qw_x$, for $w = W_{i,d}$ and $q = d$ and $1 \le x \le cb(w, d)$, locate the literals $u_{q,j,i}$ and delete those literals from the XOR-expressions corresponding to day d and driver i.

Now a satisfying truth assignment for C corresponds with a solution for PCMP(1,2) and vice-versa.
All the reduction steps can be performed in polynomial time and as PCMP(1,2) ⊆ PCMP this concludes the proof that the PCMP is NP-complete.

□

4.1.2 Motivation

As we showed in the previous section the PCMP is NP-complete in the worst case. This means that it cannot be solved in polynomial time in the size of the problem with a deterministic algorithm; at least not with a known one. Algorithms for NP-complete optimization problems can be divided in two classes: optimization and approximation algorithms. Optimization algorithms, while acknowledging the worst-case super polynomial time complexity, seek to obtain as much improvement on the running time as possible, over straightforward explicit enumeration [Coonen, Leiserson & Rivest, 1991]. Approximation algorithms drop the guarantee of finding an optimal solution and try to find good solutions within a reasonable amount of time.
For many NP-complete optimization problems one has to resort to the use of approximation algorithms, as the sizes of the instances that need to be solved are far beyond the horizon of what is optimally solvable in a reasonable amount of time. Approximation algorithms aim at finding good approximate solutions in a reasonable amount of time. Often, there is a trade-off between the time one is willing to spend and the quality of the resulting solution. We name two classes of approximation algorithms: constructive and local search algorithms.
A constructive algorithm generates a solution by iteratively extending a partial solution until a full solution is obtained. Naturally, the kind of extensions one allows is problem dependent and determines, together with the order in which the extensions are made, the quality of the found solution.
A local search algorithm, like a genetic algorithm as used in Van den Daele [2002], Ellingsen & Penaloza [2003], Marques & Morgado [1997], Sastry & Goldberg [2004] and Tijms [2003] presupposes the presence of a neighborhood structure which specifies for each solution a set of neighboring solutions. The basic concept of local search algorithms is stepwise improvement of a solution by exploring neighborhoods. The most straightforward local search algorithm, known as iterative improvement, starts by constructing an initial solution and continually tries to find a better solution in the neighborhood of the current solution. If such a better solution is found, it replaces the current solution [Van Iersel, 1994; Nuijten, 1994].

We shall use a constructive algorithms for two reasons.

- First of all this resembles the way humans, the planners, try to solve the problem. By talking with them valuable information was gained about the problem solving process which would be best simulated by a constructive algorithm.
- Second, the number of times the constraints have to be evaluated can be restricted by half with respect to a local search algorithm. This is due to the fact that constraints in most cases need to be, be it sometimes only partially, evaluated over multiple consecutive days. But this only needs to be done over days which have been matched before. With a local search algorithm all days are always matched, so if a matching is changed, all constraints have to be evaluated over a certain number of consecutive days per constraint in *both* directions. If with a constructive algorithm we match the days in a consecutive order from the beginning of the period to the end or vice versa we need only evaluate the constraints in one direction.

So we chose a constructive approximation algorithm for solving the PCMP. Also the method needs to be generic to a certain degree in order to be able to make the algorithm suitable to solve other problems as well in the future.
The method chosen is known as the constraints satisfaction approach or CSA for short. We describe this method in the next Section.

4.2 Constraint Satisfaction

In this Section we give a formal definition of the constraint satisfaction approach. Also we discuss the most important techniques that are used in solution methods for the constraint satisfaction problem, CSP. Then in the next Section we present a CSA-solution for the PCMP.

4.2.1 The Constraint Satisfaction Problem

An instance of the CSP involves a set of variables, a domain for each of the variables specifying the values to which it may be assigned, and a set of constraints on the variables. The constraints define which combinations of domain values are allowed and which are not. The problem is to find an assignment of values to variables such that all constraints are satisfied simultaneously. Every value assignment that satisfies all the constraints is called a solution. Before giving a definition of the CSP we formally define domains, constraints and assignments.

Definition 4.2 (Domain). A domain is a finite set of objects called values. The size of a domain Do is q*|Do|, where q is the maximum number of bits needed to encode any value in the domain.

□

Definition 4.3 (Constraint). Given is a collection of domains $Do = \{Do_1,..., Do_n\}$.
A constraint $co : Do_1 \times ... \times Do_n \rightarrow \mathscr{B}$ on Do defines which combinations of values from the domains satisfy the constraint. The set of all constraints is denoted by C_{Do}.

□

A constraint *co* on $Do = \{ Do_1,..., Do_n\}$ is called polynomially computable, if for all $a \in Do_1 \times ... \times Do_n$ *co*(a) can be computed within a time bounded by a polynomial in the sum of the sizes of the domains.
Only polynomially computable constraints are considered.

Definition 4.4 (Assignment). Given are a set of variables $V = \{v_1,...,v_n\}$ and for each variable $v_i \in V$ a domain specifying the values to which it may be assigned.

An assignment $a \in Do(v_1) \times ... \times Do(v_n)$ specifies for each variable $v_i \in V$ a value in its domain, i.e., $a(v_i)$ denotes the value of variable v_i. The set of all assignment of variables in V over Do is denoted by A.

□

Definition 4.5 (CSP). An instance of the Constraint Satisfaction Problem is a triple CSP = ⟨ V, Do, C ⟩, where

- $V = \{v_1,...,v_n\}$ is a set of variables;
- $Do = \{ Do(v_1),..., Do(v_n) \}$ is a collection of domains, such that each variable $v_i \in V$ is given a domain $Do(v_i)$ for which the size is polynomial in n;
- $C = \{co_1,...,co_q\}$, $C \subseteq C_{Do}$ is a set of polynomially computable constraints on Do, such that q is polynomial in n.

The question is to find an assignment $a \in Do(v_1) \times ... \times Do(v_n)$ such that $\forall i : 1 \leq i \leq q : co_i(a)$.

Such an assignment is called a solution.

□

The size of an instance of the CSP is characterized by the number of variables, i.e., each instance of the CSP can be encoded in size bounded by a polynomial in n, where n is the number of variables. This follows from the size of the domains being polynomial in n, the number of constraints being polynomial in n, and each constraint being polynomially computable, which implies that there is an encoding that is polynomial in the sum of the sizes of the domain, which in turn is polynomial in n. Nevertheless the CSP is NP-complete.

Theorem 4.2 The CSP is NP-complete.
Proof. The CSP is in NP, as for a given assignment *a* all constraints can be checked in polynomial time. A straightforward polynomial time transformation from the Satisfiability problem as defined in Lewis & Papadimitriou [1981] concludes the proof.

□

4.2.2 Solving the Constraint Satisfaction Problem

Most approaches to solve the CSP are based on tree search algorithms. In turn these are usually based on a depth-first backtrack search procedure in which a solution is constructed by instantiating one variable after another. Besides tree search approaches there are also other solution methods proposed for the CSP [Tsang, 1996]. Here, we present a framework of constraint satisfaction tree search algorithms. With respect to this framework we first introduce some definitions.

Definition 4.6 (Search State). A search state of an instance I = ⟨ X, D, C ⟩ of the CSP is a pair ⟨ ∏, δ ⟩, where $\prod \subseteq C(D)$ is a set of posted constraints and for each variable $x \in X$, $\delta(x) \subseteq D(x)$ gives the current domain.

□

Definition 4.7 (Search Tree). A search tree of an instance I of the CSP is a minimal connected directed acyclic graph ⟨ V, E ⟩, where V is a set of search states of I.

□

In every search state the current domain for a variable represents the values of each variable that can still lead to a solution. Furthermore, a search state records which constraints are posted, thus recording which

decisions led to this search state. The node set of a search tree consists of a set of search states. Going from one node to another is done by posting an extra constraint. At any node in the search tree a limited number of constraints may be posted, defining the edges of the tree. Initially, i.e., in the root of the search tree, the search state is equal to $\langle \emptyset, D \rangle$, representing that no constraints are posted and the current domains are equal to D(x). Each time a constraint is posted, inconsistent values of the unassigned variables are removed. Let $\sigma = \langle \prod, \delta \rangle$ be a search state. A value $v \in D(x)$ for a variable x is called inconsistent if no solution exists that includes the assignment of v in addition to the assignments made so far. The process of removing inconsistent values is usually called consistency checking. Once a search state is found such that $|\delta(x)| = 1$, for all $x \in X$, a solution is found and the instance is solved. If a search state is reached where the current domain of any variable is empty, i.e., there exists an $x \in X$ such that $\delta(x) = \emptyset$, we say that a dead end occurs. If a dead end is reached it is proven that no solution exists that satisfies all the original constraints together with the posted constraints. Then the tree search algorithm has to backtrack, i.e., undo certain decisions and try alternatives for them. The search typically stops if a solution is found or if all paths in the search tree have been tried without success. In the latter case, the instance is said to be infeasible.

We focus on algorithms where decisions are the selection of one value for a variable. Selecting a value $v \in \delta(x)$ for a variable x can be seen as posting a unary constraint co_x such that $co_x(v') \Leftrightarrow v' = v$, for all $v' \in D(x)$. Only those variables $x \in X$ are regarded with $|\delta(x)| > 1$. In general, constraint satisfaction tree search algorithms can be described by the following framework. The three basic components of tree search algorithms are consistency checking, variable and value selection and dead end recovery.
We discuss these components only briefly. The framework in which these components are placed is shown in figure 4.2.

```
while not solved and not infeasible do
        "check consistency"
        if "a dead end is detected" then
                "backtrack"
        else
                "select variable"
                "select value for variable"
        endif
endwhile
```

Figure 4.2 *Framework of constraint satisfaction tree search algorithms*

Consistency checking
Each time a variable is assigned a value, inconsistent values of the unassigned variables are removed. A value $v \in \delta(x)$ for a variable x is called inconsistent if no solution exists that includes the assignment of v to x in addition to the assignments made so far. Important forms of consistency are node, arc and path consistency. Here we only address node and arc consistency. Node consistency refers to the consistency of unary constraints. A constraint c_x is node consistent if $c_x(v)$ holds for all values $v \in \delta(x)$. Node consistency is easily achieved by deleting all values that do not satisfy the unary constraints. It suffices to do this only at the beginning of the search. Using unary constraints can therefore be eliminated by including appropriate redefinitions of the domains, i.e., by introducing new domains $D'(x) = \{ v \mid c_x(v) \wedge v \in D(x)\}$.

Definition 4.8 (Node consistency). Let x be a variable with current domain $\delta(x)$. Furthermore, let c_x be a constraint defined on x. Then, $\delta(x)$ is said to be node consistent for c_x if and only if $\forall_{v \in \delta(x)}: c_x(v)$

□

Arc consistency refers to the consistency of binary constraints. If a value v of a variable x is inconsistent with all values v' of variable x', i.e., $\forall_{v' \in \delta(x')}: \neg c_{x\,x'}(v,v')$, then no solution exists that includes the assignment of v to x in addition to the assignments made so far, and thus v can be eliminated from δ(x). If, however, for all values v ∈ δ(x) at least one value in δ(x') is consistent with it, $c_{x\,x'}$ is called arc consistent.

Definition 4.9 (Arc consistency). Let x and x' be variables with current domains δ(x) and δ(x') respectively and let $c_{x\,x'}$ be the constraints defined on x and x', then $c_{x\,x'}$ is said to be arc consistent with respect to δ(x) and δ(x') if and only if $\forall_{v \in \delta(x)}: \exists_{v' \in \delta(x')}\, c_{x\,x'}(v, v')$

□

When a constraint $c_{x\,x'}$ is arc consistent with respect to δ(x) and δ(x') the constraint $c_{x'\,x}$ is not necessarily arc consistent too.

Variable and value selection

The selection of a next variable and its value is done by a variable and value selection heuristic, respectively. In its simplest form both the variables and their values are chosen in a predefined order. More sophisticated variable selection strategies try to select in a dynamic way a variable which constrains the rest of the variables maximally. The idea of these *most constrained* variable selection strategies is that one starts with critical variables, i.e., the variables that are difficult to instantiate. A critical variable is one that is expected to cause backtracking, namely one who's remaining possible values are expected to conflict with the remaining possible values of other variables. Furthermore, so-called *least constraining* value selection strategies are used that select a value leaving as many values as possible open for the remaining uninstantiated variables. A least constraining value is one that is expected to participate in many solutions to the CSP.

Dead end recovery

If a dead end is reached it is derived that no solution exists that satisfies all original constraints and all posted constraints. As a result at least one posted constraint should be undone and an alternative constraint should be tried. In its most simple form this is done by way of chronological backtracking, which consists of undoing the last constraint and trying another constraint. Chronological backtracking, however, often results in exhaustively searching a subtree in which no solution can be found. The main problem of escaping from a dead end is to decide which posted constraints to undo. Besides chronological backtracking, we mention back jumping and dependency directed backtracking. Back jumping [Willemen, 2002], too undoes the last posted constraints, but it also undoes those constraints that did not directly contribute to the dead end, up to the last posted constraint that did contribute to the dead end. This decreases the number of backtrack steps in the tree search. Dependency directed backtracking [Tsang, 1996] incorporates back jumping and constraint recording that records the reasons for the dead end in the form of new constraints, so that the same conflicts will not arise again in the later search.

4.3 Solution: the core algorithm

In this Section we first model the PCMP as a CSP. Then we describe the CSA-algorithm with which we try to solve the modeled instance.

Definition 4.5 states that an instance of the CSP consists of three elements; a set of variables V, a set of domains Do and a set of constraints C. Recalling the definition of a PCMP-instance (definition 2.9) and a PCMP-solution (definition 2.10) we see that a block assignment (definition 2.5) has to be made on every day of the period. Such an assignment assigns a block to a driver. Therefore we take the blocks as the variables and the drivers as the domains. But only the blocks from the production need are relevant. To distinguish between block types on more than one day and of which multiple ones are needed on one day we number them in the following way, conform the identification used in Section 4.1.

A $v \in V$ is uniquely identified through a day $d \in D$, a block type $pb \in tb(d)$ and a sequence number $i : 1 \leq i \leq cb(pd,d)$. For example: $d_0 \in Do$, with $d_0 = 5$, pb = A and $cb(A,5) = 3$, then a $v \in V$ can be $v = 5A_1$ or $v = 5A_2$ or $v = 5A_3$.
The constraint-set C is equal to the constraint-set Co from definition 2.9.

Formally:

- Do = CH;
- C = Co;
- $\forall_{d \in D,\, pb \in tb(d),\, i \in cb(pb,d)} : v \in V$, with $v := d{+}{+}pb_i$

In the following Sections we will discuss the application of the three basic components of constraint satisfaction tree search algorithms in our solution method of the PCMP.

4.3.1 Consistency checking

Because of the expression power needed in the constraints we cannot easily convert them to unary and binary constraints. As we have seen before in Section 3.1.3, c1, constraints can work over several variables at once, the so-called multi-variable constraints. Also they can be conditional and/or stated inversely.
An example of an inversely stated conditional constraint is the following constraint:

c6. if chauffeurID = 10 then (COUNT /3 inzet) =< 2 ~

which states that driver with ID 10 may not work three days in a row. It is called an inversely stated constraint because it is concerned with a domain element rather than a variable element.
Due to time limitations we haven't pursued the issue of converting inversely stated constraints or converting multi-variable constraints into several binary ones.
Therefore we have only incorporated node consistency checks. As all variables have, in essence, the same domain we perform the node consistency checks dynamically to reduce the bookkeeping overhead of each variable and its domain, which would be incurred if the check would be performed once, before the start of the algorithm. The dynamic check is performed with the use of the driver's ability function *ab*.

4.3.2 Variable and value selection

For variable selection we use a random selection rule. This makes it possible for the algorithm to produce different scenarios each time it is run and complies with the request of producing multiple different scenarios. However, the randomization is restricted such that variables are chosen per consecutive day. So if the algorithm starts at $d_i \in D$ then variables are chosen from $tb(d_i)$ until all variables of that day are matched, i.e., $\forall_{pb \in tb(d)} : cb(pb,d) = 0$. Only then day *next*(d) or *prev*(d) is considered depending whether we start at the first day of the period or the last respectively. In Section 6.1.1 we briefly discuss a possible most-constrained variable selection rule. For value selection we have three different options: RWS, IRWS, BestSelectMin and BestSelectMax. Here the strategy with respect to the fourth requirement of a PCMP-solution comes in: treating all drivers as equal as possible. As can be seen, an equal treatment boils down to the drivers having an almost equal amount of penalty at the time a solution is found. Before describing the value selection rules we first give a description of how the penalty system works.

The Penalty System
Penalties are incurred whenever a matching is made which doesn’t correspond to the preference of the driver and whenever a soft constraint is violated (see also definition 2.6).

In the latter case the penalty is always the same but in the former it isn't. The reason behind this decision is to make it possible to treat drivers equally. We illustrate this with an example and then give a formula for the cost assignment function *c*. Note that all used functions have been defined in Chapter 2.

Let a driver $ch_1 \in CH$, a day $d_1 \in D$ and A, B, C $\in tb(d_1)$.

Suppose $pref(ch_1,d_1) = A$, $bbt(A) = 480$ and $bet(A) = 1020$.
Furthermore $as(d_1,ch_1) = B$, $bbt(B) = 700$ and $bet(B) = 1500$.
Also $bbt(C) = 1500$ and $bet(C) = 1900$.

Then from a driver's point of view it is better to have block type B assigned than type C. This is because the working period from block types A and B overlap each other. To express this we assign the *full* so-called change penalty whenever a preference block type and an assigned one have no overlap in time. Otherwise the change penalty is only assigned to a certain extent.

Definition 4.10 (Change Penalty). The change penalty is defined as the bonus value of the preference block type times the extra preference a driver has given:

$$cp : CH \times D \rightarrow \mathbb{N}, \text{ where } cp(ch,d) := expref(ch,d) * bbw(as(d,ch))$$

□

The bonus is a way to indicate the difficulty a planner has to assign this block type to a driver. Usually block types with a high bonus value are not preferred much.

Definition 4.11 (Extent). $\forall_{d \in D,\, ch \in CH} : pref(ch,d) = x \wedge as(d,ch) = y$, the extent in which a change penalty is incurred is given by the function:

$$ex : PB \times PB \rightarrow \mathbb{R}, \text{ with}$$

$$num := \min(bet(x),bet(y)) - \max(bbt(x),bbt(y))$$
$$denom := (bet(y) - bbt(y))$$

then

$$ex(y,x) := 0 \quad \text{if } \neg bw(y)$$

$$ex(y,x) := k \quad \text{if } bw(y)$$

with

$$k := 1 \quad \text{if } \max(bbt(x),bbt(y)) < \min(bet(x),bet(y)) \text{, i.e., no overlap}$$
$$k := 1 - \frac{num}{denom} \quad \text{otherwise}$$

□

Also when drivers have no ability whatsoever for a day the average penalty incurred by the drivers from the previous matched day is assigned to keep the total equal. This also happens when drivers have given no preference to a day. Section 5.3.1 explores the consequences of this choice.

The cost assignment function *c* (definition 2.6) then is defined as:

$$c((pb,ch,d)) := cp(ch,d) * ex(pref(ch,d), pb) \quad \text{if Satisfies}(SC \cup SC_{ch}, pb, ch, d)$$

$$c((pb,ch,d)) := cp(ch,d) * ex(pref(ch,d), pb) + softPen \quad \text{if } \neg \text{Satisfies}(SC \cup SC_{ch}, pb, ch, d)$$

where $softPen \in \mathbb{N}$.

Pattern recognition
The above way of partially assigning penalty to matchings can be considered as a mild form of pattern recognition with respect to time. A question was to try and include another form of pattern recognition which could recognize patterns of preference block types over a number of days and develop a way to reduce the amount of match deviations from the preference in the days of these patterns.
A clear example of the reason for this is a vacation period. Such a period typically consists of several consecutive days with a certain block type as preference. To let deviations from these patterns weigh more than others we increase the change penalty per day in the pattern for the driver with a certain percentage. The patterns themselves can be defined by the user.

Now we return to the matter of value selection for which we have defined the following rules:

- **RWS**, roulette wheel selection. Following a technique used frequently in genetic algorithms [Kennedy & Eberhart, 2001], this rule selects a driver which has a relative large total penalty from a pool CHP denoting the drivers that can be selected. Each driver ch $\in$ CHP has a total penalty, which is the sum of penalties incurred from the already matched days for that driver: $cump(\text{ch}) := \sum_{\text{d} \in \text{D}} c(as(\text{d},\text{ch}), \text{ch}, \text{d})$.
  This number is used to create the roulette wheel. The larger the number the larger the part of the wheel for the corresponding driver. The *size* of the wheel is defined as $\text{size}_{\text{RWS}} := \sum_{\text{ch} \in \text{CHP}} cump(\text{ch})$
  After creating the wheel a random number x : $0 \leq x \leq size$ is generated. The driver corresponding to the part of the wheel x falls in is chosen.

- **IRWS**, inverse roulette wheel selection. Selects a driver which has a relative low total penalty. The numbers used to form the parts of the wheel are different from the ones defined in RWS in the following way:
  let $max := \uparrow_{\text{ch} \in \text{CHP}} cump(\text{ch})$, then the number used to create the wheel for a ch $\in$ CH is defined as:
  $icump(\text{ch}) := max * 1.1 - cump(\text{ch})$
  The random number x is then generated in the interval $[0.. \sum_{\text{ch} \in \text{CHP}} icump(\text{ch})]$

- **BestSelectMin**. Selects a driver with the absolute least amount of total penalty. When more than one driver has this least amount one is randomly selected from them.

- **BestSelectMax**. Selects a driver with the absolute most amount of total penalty. When more than one driver has this most amount one is randomly selected from them.

Besides these selection rules we make use of bottleneck lists and the tabu-search technique, both of which will be described in the following section. We explain why these selection rules help us in keeping the penalty within margins in Section 4.3.4.

4.3.3 Dead end recovery

A dead end occurs when the current partial solution cannot be extended by assigning an unassigned block to unassigned drivers. As a result at least one posted constraint should be undone and an alternative constraint should be added. In our case of assigning values to variables, this means that at least one assignment has to be replaced by another; in its most simple form undoing the last assignment and trying another assignment, i.e., chronological backtracking. We can also restart from scratch and try to construct a different solution by means of the randomized variable and value selection rules. We use chronological backtracking in trying to solve the instance. However, if this doesn't seem to help, i.e., a maximum number of backtracks have occurred, we restart the entire day first using a bottleneck list to select the variables from. This bottleneck list is constructed per day. If a dead end is reached, the variable under consideration is placed in the bottleneck list together with a number denoting the depth in the search tree (definition 4.7) for this day at which this dead end occurred. This number is used to select variables from the list when restarting a day. Variables are

selected in order of decreasing number, following Willemen [2002]. If restarting a day a number of times doesn't work we backtrack entire days, again till a certain predefined maximum is reached. Then if this maximum is reached the entire scenario is restarted from scratch. Because of the randomized variable and, in case of RWS and IRWS, randomized value selection, the chance of exploring a different part of the solution space is large. Restarting the entire scenario is also bounded by a maximum.
The following table (4.1) introduces abbreviations for the bounds mentioned above. These abbreviations will be used in the pseudo-code in the following Section.

Table 4.1 *Parameters of the algorithm*

Concept	Abbreviation
max. # backtrack times per day	maxbtd
max. # restart times of a day	maxrtd
max. # backtrack over days in a scenario	maxbts
max. # restart times of a scenario	maxrts
max. # tabuiterations per variable	maxtabu

Tabu search is used to efficiently guide the search for a value.
When a variable is selected we use one of the value selection rules to select a value for that variable. When this variable-value combination, a matching, violates a hard constraint tabusearch is used to try to select another value which doesn't violate any of the hard constraints. In essence this works as follows. Whenever a value from the variable's domain cannot be assigned it is placed on a list, the tabulist. Such a tabulist is constructed per day. Values from this list are not considered again for this variable. So in fact we reduce the domain of the variable. The tabulist is cleared whenever backtracking occurs. Not all values in the domain of a variable are considered; the tabu search terminates either when a feasible, non hard constraint violating, value has been found or when a maximum number of iterations, maxtabu (table 4.1) is reached.

4.3.4 Pseudo code

We describe the main procedures of our solution approach by means of pseudo code. Comments in the pseudo code are preceded by // marks. Most procedures change the current solution, the set of matchings T. Therefore the variable representing the current solution is considered global together with the parameters from table 4.1. The initial value of a global variable is denoted by a star, e.g., T* is the initial value of T.
The begin and end of *while* loops and *if-then* selection statements are marked by increased and decreased indentation respectively.
In words the constructive algorithm works as follows. First we determine whether the input can be transformed into a feasible solution by means of node-consistency checking with respect to the ability function *ab* (definition 2.3). This is expressed by the predicate *initFeasible* (definition 2.11). Then we try to assign values to variables of the production need per day, looping through the days in a consecutive way.
Assigning the drivers (values) to the production need (variables) is done by looking at the preference (*pref*) of the drivers. Drivers who have a preference corresponding with a block type from the production need are assigned first. This is done in order of decreasing total penalty, according to selection-rule RWS or BestSelect. Then, in the frequently occurring case that not all variables have been assigned a value we 'forcefully' match values to the remaining variables. Forcefully, because the matching is not in accordance with the preference. This is done in order of increasing total penalty while also taking into account the change penalty which would be incurred if the matching is to be definite.
If the previous two steps, matching drivers according to their preference and forcefully matching drivers, are successful then all variables have been assigned a value and the production need is fulfilled.
So at that moment the first requirement of a solution for the PCMP (definition 2.10) is met. In order to meet the second requirement we perform a procedure called PostMatching. In this procedure, which is executed per day, the remaining unassigned values ch for that day $d \in D$, i.e., $\{ ch \in CH \mid as(d,ch) = \emptyset \}$ are assigned

to dummy block types b. These dummy block types denote that the assigned driver is idle (*bw*(b)) for that day. Of course these assignments should not violate any of the hard constraints.
Note that drivers can have selected idle block types as their preference and as idle block types, naturally, are no part of the production need these drivers are not considered in the first step of the constructive algorithm.

In consecutive order we denote the steps of the algorithm as:

1. matchPreference
2. forceMatchBlock
3. postMatching

Let

prv() : T → T, where *prv*((a,b,c)) returns the chronological previous matching if one exists;
first() : T → PB, where *first*((a,b,c)) = a;
second() : T → CH, where *second*((a,b,c)) = b;
last() : T → D, where *last*((a,b,c)) = c;

The main body of the algorithm is described in figure 4.4. All sub-procedures which are written in italics are also described in pseudo code further on in a top-down approach. To make the code more readable the penalty/cost assignments are omitted. A short description of each procedure is given directly below the corresponding figures.
In essence we use the following framework, which is slightly different from the one shown in figure 4.2 because of reasons explained earlier.

```
if not infeasible then
    while not solved do
        "select variable"
        "select value for variable"
        "check consistency"
        if "a dead end is detected" then
            "undo value assignment"
            "backtrack"
        else
            "make value assignment permanent"
        endif
    endwhile
endif
```

Figure 4.3 *Framework of constraint satisfaction tree search algorithms*

The pseudo code is shown in a different syntax than the one used in figures 4.2 and 4.3 to emphasize the difference in detail. The pseudo code is a more detailed description of the framework defined in figure 4.3.

```
TRY TO MATCH ALL DAYS( set D )

day := | D |;
currentbts := 0;
currentrts := 0;
set CH;

if (initFeasible) then
    while currentrts ≤ maxrts and day > -1 do
        ok := assign( day );
        ok := ok and postMatching( day );
        if (not (ok) ) then
            // backtrack an entire day if possible, else restart
            if currentbts ≥ maxbts then
                // backtracking impossible, restart entire scenario
                day := | D |;
                forall d∈ D do reset( d );
                currentrts := currentrts + 1;
                currentbts := 0;
            else
                // backtrack one day
                if day < | D | then
                    reset( day );
                    day := day + 1;
                else
                    reset( day );
                currentbts := currentbts + 1;
        else
            // day matched succesfully
            if "substantial progress" then
                currentbts := 0;
            day := day – 1;
```

Figure 4.4 *Main procedure of the algorithm*

This procedure tries to match all the days in the period by means of the sub-procedures *assign* and *postMatching* which are defined in figure 4.5 and 4.9 respectively. The number of backtracktimes over days is reset only if some 'substantial progress' is made. By this we mean that the number of different, matched, days between two successive backtrack-day actions should be more than maxbts. Other options are explored in Section 5.2. Note that we traverse the days in the period in decreasing order.

```
ASSIGN ( d ∈ D )

currentrtd := 0;
currentbtd := 0;
reset( d );
matchPreference( d );
bottleneckList := [ ];
depth := -1;
stop := false;
restarted := false;

repeat
        depth := depth + 1;
        block := selectBlock( d, bottleneckList, restarted );
        restarted := false;
        ok := forceMatchBlock( block, d );
        if (ok) then
                // the block has been matched succesfully to a driver
                cb( block, d) := cb( block, d) – 1;
        else
                // unable to match the block, place it in the bottlenecklist
                bottleneckList := bottleneckList ++ (block, depth);
                // backtrack blocks if possible, else restart day if possible
                if (currentbtd ≥ maxbtd) then
                        // backtracking impossible, try restarting entire day if possible
                        if currentrtd ≥ maxrtd then
                                // abort day
                                stop := true;
                                currentrtd := 0;
                                reset( d );
                        else
                                // restart day
                                currentrtd := currentrtd + 1;
                                restarted := true;
                                reset( d );
                                matchPreference( d );
                else
                        // backtrack
                        currentbtd := currentbtd + 1;
                        // undo last matching
                        t := prv( T );
                        T := T \ t;
until ( forall pb ∈ tb( d ) : cb(pb, d) = 0 ) or stop
return not stop;
```

Figure 4.5 *Procedure 'Assign'*

This procedure tries to match one day d by means of the sub-procedures *matchPreference* and *forceMatchBlock* which are defined in figure 4.6 and 4.7 respectively. The sub-procedure *selectBlock* randomly selects a block type from the production need for this day, or uses the bottleneck list if a restart has occurred.

```
MATCHPREFERENCE( d ∈ D)

noMoreBlocks := false;

repeat
        block := selectBlock( day, [ ], false );
        if (block ≠ Ø) then
                // a block has been found
                candidates := Ø;
                forall ch ∈ CH ∧ t ∈ T: second(t) ≠ ch do
                        // select from drivers which are not already matched
                        if (block ∈ ab(ch, d) and
                            block ∈ pref(ch, d) and
                            Satisfies( {HC ⋃ HC_ch}, block, ch, d) ) then
                                candidates := candidates ⋃ { ch };
                                // assign as many drivers from candidates as possible to blocks of this type
                                members := "select at most cb(block, d) drivers from the candidates,
                                            according to selection rule RWS or BestSelectMax";
                                forall chm ∈ members do
                                        T := T ⋃ m(block, chm, d);
                                        cb(block, d) := cb(block, d) – 1;
        else
                noMoreBlocks := true;
until ( forall pb ∈ tb(d): cb(pb, d)=0 ) or noMoreBlocks;
```

Figure 4.6 *Procedure 'MatchPreference'*

This procedure tries to match as much block types from the production need to drivers conform their preference. Here, *selectBlock* randomly selects a block type from the production need. The value selection rule RWS or BestSelectMax is used to select drivers who already have a (relative) large amount of penalty. Recall that assigning conform preference does not increase the penalty of a driver.

```
FORCEMATCHBLOCK (block ∈ tb(d), d ∈ D)

result := false;
tabulist := [ ];
ch := selectDriver( block, d, tabulist );
if ch ≠ Ø then
        // a driver with the right ability has been found
        // node consistency check is performed in selectDriver
        if (Satisfies( {HC, HC_ch}, block, ch, d)) then
                // assign
                result := true;
                T := T ⋃ m(block, ch, d);
        else
                // this driver assignment would violate one or more hard constraints, search another driver
                chn := tsearch( ch, block, d );
                if chn ≠ Ø then
                        // suitable driver found
                        result := true;
                        T := T ⋃ m(block, chn, d);
return result;
```

Figure 4.7 *Procedure 'ForceMatchBlock'*

This procedure tries to match a driver to a block type using selection rule IRWS or BestSelectMin to select a driver with a relative low amount of penalty. As a result the penalty of that driver will increase. *tsearch* (defined in figure 4.8) is used to guide the search for a driver.

```
TSEARCH(ch ∈ CH, pb ∈ tb(d), d ∈ D)

tabuIterations := 0;
result :=  Ø;
tabulist := [ ch ];
stop := false;
end := false;

while (not stop) and (not end) and (tabuIterations < maxtabu) do
        chn := selectDriver(pb,d,tabulist);
        if chn = Ø then
                stop := true;
        else
                if (Satisfies ( {HC, HC_chn}, b, chn, d)) then
                        end := true;
                        result := chn;
                else
                        //add this driver to the tabulist
                        tabulist := tabulist ++ [ chn ];
                        tabuIterations := tabuIterations + 1;
return result;
```

Figure 4.8 *Procedure 'tSearch'*

This procedure implements the Tabu Search mechanism.

```
POSTMATCHING( d∈D)

result := true;

forall ch∈CH : t∈T and second(t) ≠ ch do
        if pref(ch,d) ∉ tb(d) then
                if Satisfies( {HC, HC_ch}, pref(ch,d), ch, d) ) then
                        // non-work block preferred and ok
                        T := T ∪ m(pref(ch,d), ch, d);
                else
                        // non-work block preferred and not ok
                        result := false;
        else
                // workblock preferred but none available, try matching with non-work block
                stop := false;
                repeat
                        block := "select non-work block that hasn't been selected before for this driver"
                        if block ≠ Ø then
                                if Satisfies( {HC, HC_ch}, block, ch, d) ) then
                                        T := T ∪ m(block, ch, d);
                                        stop :=  true;
                        else
                                stop := true;
                                result := false;
                until stop;
return result;
```

Figure 4.9 *Procedure 'PostMatching'*

This procedure tries to match unassigned drivers for day d with non-work blocks, i.e., with blocks b, where $\neg$ *bw*(b).

```
RESET (d∈D)

forall block ∈tb(d) and ch∈CH  do T: = T \ (block, ch, d);
forall block ∈tb(d) do cb(block, d) :=cb(block, d)*;
```

Figure 4.10 *Procedure 'Reset'*

This procedure undoes any matchings made for day d and resets the possibly modified cardinalities for day d.

```
SELECTBLOCK(d∈D, list bottleneckList, boolean restarted)
// use list only if the day d is restarted which is indicated by the boolean

block := Ø;
if (restarted) then
        // use the list, which has at least one element
        block := "select block with highest value from the bottleneckList";
        bottleneckList := bottleneckList – block;
else
        // don't use the list
        block := "randomly select a block b∈tb(d) with cb(b,d)>0";
return block;
```

Figure 4.11 *Procedure 'selectBlock'*

This procedure is used to select a new variable.

```
SELECTDRIVER(pb∈ tb(d), d∈D, list tabulist)

result := "select a driver ch∈CH out of the collection of unassigned drivers for this day, d, minus the
          drivers ch∈CH in the tabulist with pb∈ab(ch, d) according to the selection rule IRWS or
          BestSelectMin. If this isn't possible return Ø";
return result;
```

Figure 4.12 *Procedure 'selectDriver'*

This procedure is used to select a new value using the selection rule IRWS or BestSelectMin.

In all cases where a choice can be made between the rules IRWS or BestSelectMin and RWS or BestSelectMax, we used the 'BestSelect' version for reasons explained in Section 5.4.3.

5 Results

In this chapter we report on the experiments carried out with the algorithm presented in Section 4.3.
First in Section 5.1 we present a real-life problem instance. In Section 5.2 the results of the solution of the instance are presented and compared with the results of a manual solution. In Section 5.3 we analyze some test instances and Section 5.4 discusses various possible parameter settings for the algorithm. Finally Section 5.5 reflects on the results of the experiments.

5.1 NNC1: a real-life instance

In this Section we present an instance of the PCMP taken from real-life. For privacy reasons names and indices with respect to persons and places have been altered. The instance we treat shall be called NNC1 in the remainder and is considered a relatively large real-life instance of the PCMP which is mainly determined by the number of drivers |CH|, but also the number of the constraints |Co| and the length of the period |D| have some influence. Unfortunately some test data has not been available in time. Whenever this is the case we elaborate our choice of the used simulated data.

5.1.1 Representation

As defined in definition 2.9 an instance of the PCMP consists of the production need input, the block input, the driver input and a set of constraints. Below we describe each of these elements for the NNC1-instance.

Production need input
The number of days in the period |D|, is 203. So this matching is performed for 29 weeks.
The possible block types available to this branch-location during this period are given by the set PB and denoted by strings.
PB := { A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, VERL, VRIJ }, totaling to 18 different block types.
For easy reference we will not give the results of the functions *tb* and *cb* for each day. It suffices to mention that on an average weekday 6 different block types with a total cardinality of approximately 25 are needed and on an average day in the weekend 3 different block types with a total cardinality of approximately 10 are needed.
Furthermore blocks of the type VRIJ and VERL are never needed, i.e.,

$cb(\text{ VRIJ, d }) = 0$ and $cb(\text{ VERL, d }) = 0, \forall d \in D$.

Block input
The set PB is equal to the one defined at the production need input. Furthermore we have that:

$\forall pb \in PB \wedge \neg\, bw(\text{ pb }) : \textit{working period}(pb) = 0$;

$\forall pb \in PB \wedge bw(\text{ pb }) : bbw(pb) = x$, with $x \in \mathbb{N}^+$;

$\forall pb \in PB \wedge \neg\, bw(\text{ pb }) : bbw(pb) = 2x$;

The value of x and the decision to keep the bonus value of all work blocks equal and to double the bonus for a non-work block with respect to a work block was made because no real-life data was available and the idea was to let the drivers work on their preferred days as much as possible. Therefore not granting a non-work block was considered a worse move than not granting a work block.

Only VRIJ is considered a non-work block, i.e., $bw(b) \Leftrightarrow b = \text{VRIJ}$;

We omit the start and end times of each individual block type for the sake of clarity. It suffices to say that all block types have different working periods, together covering an entire day. These working periods can overlap each other.

Driver input
The number of drivers available to this branch-location during this period is given by |CH| and is equal to 34. D is the same set as used at the production need input. We do not give the results of the functions *pref* and *ab*. Instead we mention that the ability for each driver is equal to PB. The ability for each driver-day combination unfortunately was not available. In order to test the algorithm properly we let each driver have all the possible block types as ability. Formally:

$ab(ch, d) = PB, \forall pb \in PB \wedge d \in D$.

Furthermore, preference has been entered by each of the 34 drivers.
The extra preference *expref* has been set to 1 for every driver on every day because this information was unavailable also. As stated in definition 2.3 $expref(ch, d) = 1$ if a driver $ch \in CH$ hasn't entered any extra preference for day $d \in D$.

Constraint set
We use eight constraints which are all elements of the set HC. We give a textual representation $ct_1,\ldots,ct_8$ and a representation in the CL from definition 3.3, $c_1,\ldots,c_8$, of each constraint below.

ct_1 : Between two successive work blocks at least 11 consecutive hours of rest need to be taken.
c_1 : if true then rust / 2 >= 11 * 60 ~

ct_2 : During a period of 7 days at least one period of 36 consecutive hours of rest need to be taken. When this is not the case, then during a period of 9 days at least one period of 60 consecutive hours of rest need to be taken.
c_2 : if not (rust / 7 >= 36 * 60) then rust / 9 >= 60 * 60 ~

ct_3 : If during three consecutive days a block type G,O,L,F,P,A or K is used then either the fourth day needs to be matched to a block type G,O,L,F,P,A or K, or 48 hours of consecutive rest need to be taken.
c_3 : if (count / 3 (in blok { G;O;L;F;P;A;K })) = 3 then
((count / 4 (in blok { G;O;L;F;P;A;K })) = 4) or ((rust / 5) >= 48 * 60) ~

ct_4 : During one week, from Sunday till Saturday, at most 6 work blocks may occur.
c_4 : if true then (count [0;6] inzet) <= 6 ~

ct_5 : If the driver belongs to a so-called *bus*-location then no more than 6 work blocks may occur during a period of 7 days.
c_5 : if vesttype = bus then (count /7 inzet) <= 6 ~

ct_6 : If the driver belongs to a so-called *spoor*-location then no more than 7 work blocks may occur during a period of 8 days.
c_6 : if vesttype = spoor then (count /8 inzet) <= 7 ~

ct_7 : During a period of 13 weeks at most 29 occurrences of a block type from the set { G, O, L, F, P, A, K } may be present.
c_7 : if in blok { G;O;L;F;P;A;K } then (count /91 (in blok { G;O;L;F;P;A;K })) <= 29 ~

ct_8 : During a period of 13 weeks at most 9 work blocks may occur on Sundays in the period.
c_8 : if (dag = 0 and inzet) then (count /91 (dag = 0 and inzet)) <= 9 ~

Furthermore we use $bbw(pb) = 6, \forall_{pb \in PB}$. Note that the absolute value of the *bbw* function result is not relevant for the computational results as it is equal for every block type in both the manual and the automatic matching.

5.1.2 Computational results

We evaluate our algorithm on the ability to solve the instance providing it is feasible and the amount of time needed to do so. Besides we also take into account some concepts defined by Ovitech to evaluate the quality of a solution. We describe these concepts and what their ideal state should be below.

All concepts are defined per driver.

Definition 5.1 (Evaluation Concepts). The following concepts are used in the evaluation of the algorithm.

1. **Total change penalty (TCP)** := "the sum of the costs (def. 2.6) incurred by a driver over the complete period". Formally, $\forall ch \in CH$:

   $$\sum_{t \in T \wedge second(t) = ch} c((first(t), ch, last(t)))$$

   Ideally this number should be low and equally distributed amongst the drivers.

2. **Block matches (BM)** := "the number of days on which a certain work block type has been assigned while another work block type was preferred". Formally, $\forall ch \in CH$:

   $$\sum_{t \in T} [(pref(ch, last(t)) \neq first(t)) \wedge bw(pref(ch, last(t))) \wedge bw(first(t))]$$

   So, whenever a matching isn't conform the preference and both the preference block type and the assigned block type are work blocks, we say a block match has occurred.
   Ideally this number should be low and equally distributed amongst the drivers.

3. **Day matches (DM)** := "the number of days on which either a work block type has been assigned while a non-work block type was preferred (+1) or vice-versa (-1)". Formally, $\forall ch \in CH$:

   $$\sum_{t \in T} [(pref(ch, last(t)) \neq first(t)) \wedge \neg bw(pref(ch, last(t))) \wedge bw(first(t))] -$$
   $$\sum_{t \in T} [(pref(ch, last(t)) \neq first(t)) \wedge bw(pref(ch, last(t))) \wedge \neg bw(first(t))]$$

   Ideally this number should be zero for every driver, assuming that every driver has preferred an equal amount of non-work blocks.

4. **Deviation from the average total change penalty in terms of percentage (DEV)**.
   Ideally this number should be close to zero and equally distributed amongst the drivers.
   For this last requirement we make use of a margin of 10% and evaluate what percentage of the drivers fall within this margin concerning their total change penalty. The result is shown as "% in margin". This margin is conform requirement 4 of the PCMP-solution (definition 2.10).

5. **Similarity with the preference roster in terms of percentage (SIM)**.
   Ideally this number should be close to 100 and equally distributed amongst the drivers.

□

We run the algorithm for a number of times with fixed parameters. We choose 10 as the number of runs because this seems to be reasonable considering the results of Willemen [2002] and the fact that we have used his method as a framework. Also we do not intend to perform extensive statistical measurements on the results. They are only intended to give an idea of the workings of the developed method. The results of the

last run of this instance is depicted in figure 5.1. In the remainder the values on the vertical-axis denote different units, depending on the evaluation concept under consideration. The averages of the runs are shown in table 5.1, together with the standard deviation which is shown between parentheses beneath the average value. The parameter values which are used are shown in table 5.2. The last two columns show the running time of the algorithm. For the NNC1-instance-runs in table 5.1 all ten were successful in solving the instance.

Test Environment
All runs were performed on a Windows 2000 SP 4 – PC with an Intel Pentium 4, 2.8 GHz, processor and 512 MB internal memory. The algorithm's executable was obtained by the Java developing environment of NetBeans 3.6. We assume that the users, the planners in our case, use PC's with a similar configuration.

Table 5.1 *Results of ten runs of the algorithm on the NNC1-instance*

	TCP	**BM**	**DM**	**DEV**	**SIM**	**% in margin**	**time (min)**	**(secs)**
Average	***129.72***	***16.69***	***-6.91***	***2.18***	***80.47***	***100***	***0.29***	***17.2***
Deviation	***2.31***	***0.23***	***0.11***	***0.24***	***0.18***	***0***	***0.12***	***7.38***

Table 5.2 *Parameter settings (low)*

Parameter	**Value**
Value-Selection-Rule Used	BestSelect
maxtabu	10
maxbtd	10
maxrtd	10
maxbts	10
maxrts	10

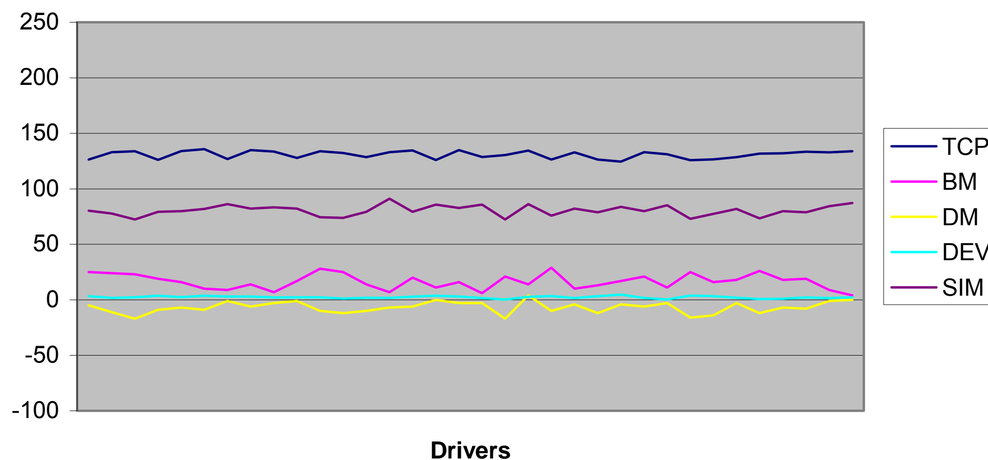

Figure 5.1 *Results of run number 10 from table 5.1 on the NNC1-instance*

5.1.3 Manual matching

In this Section we give the results of a manual matching of the NNC1-instance performed by planners. In Section 5.3 we draw some conclusions from the comparison between the automatic matching (figure 5.1) and the manual matching depicted in figure 5.2.

Table 5.3 *Results of a manual matching of the NNC1-instance*

RUN NR	TCP	BM	DM	DEV	SIM	% in margin	time (min)	(days)
1	149.27	25.44	-0.15	47.73	78.85	14.71	43920	30.5

For a manual matching the parameters are of course irrelevant. The graph of the only manual run is shown in figure 5.2 on the same scale as the one for the automatic matching from figure 5.1.

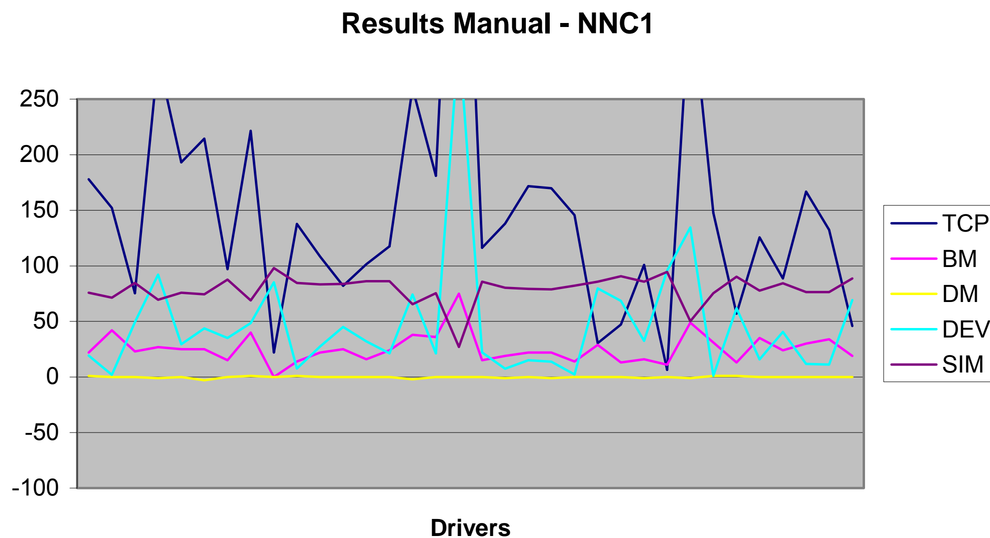

Figure 5.2 *Results of a manual matching of the NNC1-instance*

5.2 Parameter Settings

In the previous Section we kept the parameter settings constant at the values from table 5.2. In this section we discuss the expected influences of changing them. Then, in the following Section, we experiment with these settings on some test instances in order to try and fine-tune the algorithm.
We distinguish three kinds of parameters. First, the parameters from table 4.1 with the exception of *maxtabu* together give an upper bound on the number of iterations of the algorithm. In the remainder we shall refer to these kind of parameters as upper bound parameters. Second, we have the value selection rules RWS, IRWS, BestSelectMin and BestSelectMax, which we will refer to as the selection parameters. Third and last there is the parameter maxtabu that deals with maintaining a feasible schedule in each step. We shall call this the tabulist parameter.

Expectations

We expect that increasing the value of the upper bound parameters leads to a higher average running time of the algorithm while possibly increasing the number of instances solved. Moreover, if we are able to solve an instance for certain values of the upper bounds, we expect to solve it for higher values as well. For the selection parameters the effect on the running time is not clear but we expect the quality of the solution with respect to an equal spreading of the values of the evaluation concepts (definition 5.1) to improve when using the BestSelect rules as they are less random. Note that solutions of a single instance can still differ because of the random variable selection rule.
Finally, increasing the tabulist parameter value will probably increase the total running time in the same way as it did for the upper bound parameters.

Backtracking and bottleneck identification
As we always use both backtracking and bottleneck identification we don't introduce any parameters for turning these features on or off. We can however change the moment in which the backtracking counter over days (currentbts in figure 4.4) is to be reset. In figure 4.4 we mentioned the notion of *substantial progress*. This notion is introduced to keep the algorithm from looping indefinitely. Then, three options are possible in general. First, the one we used, namely keeping more than *maxbts* days distance between two successive resets. This implies that at least one day progress has been made whenever a reset occurs.
Second, we can relax the previous option by resetting the parameter when a day is matched that hasn't been matched before. Finally, we can choose never to reset the parameter, effectively setting the upper bound on the number of backtrack steps. Choosing this last option implies that days which are difficult to match early in the process, can restrict the degrees of freedom for the days which are being matched later on. This is the case when a large amount of backtracking is needed in the beginning of the process.

5.3 Theoretical Test Cases

As we only have one real-life instance we create some instances ourselves. In doing so we explore a boundary case (Section 5.3.1) as well as some random generated ones (Section 5.3.2). Furthermore we 'tighten' the NNC1-instance by increasing the production need to investigate the region where the algorithm shows a significant decrease in speed and where backtracking starts to occur.

5.3.1 A boundary case

In this Section we explore a boundary instance, namely the NNC1-instance in which no preference rosters have been entered, i.e., $\forall ch \in CH \wedge d \in D : pref(ch, d) = \varnothing$, called NNC2. Having no preference implies that no change penalty can be incurred (see also Section 4.3.2) and as a result the value selection rules are completely non-deterministic. Therefore it isn't possible to treat all drivers equal with respect to the amount of free time (non-work blocks). This would normally be guided by the penalty mechanism; assigning the full extent if a non-work block is preferred while a workblock is assigned, and no penalty whatsoever if a non-work block is assigned while a workblock is preferred. With this we try to grant the requested free time as much as possible. Implicitly we assume that drivers have requested a reasonable, and between them relatively equal amount of free time. This assumption is justified because the constraints enforce a certain minimum amount of free time. If a driver has requested more free time than others, at first this will be granted but eventually the total penalty of this driver will be substantially lower and he/she will get work blocks assigned and as a result other drivers will have less chance of being assigned a workblock.
But in this boundary case it can be that one driver gets assigned far more free time than another. This of course is undesirable. In Section 6.1.1 we propose a solution for this problem.

So we expect to see lines 1,2,3 and 4 lie exactly at zero and line 5 at 100 because no 'changes' have occurred as no preference has been altered. Implicitly we look at days without preference as days with a joker stating that it doesn't matter which block type is assigned. Furthermore, because we don't restrict the drivers abilities we expect the algorithm to finish in more or less the same time as it needed for the NNC1-instance.

In table 5.4 the results of five runs are shown. We omit the graph for the solution of this instance because most lines overlap each other. We used the parameter settings from table 5.2.

Table 5.4 *Results of five runs of the algorithm on the NNC2-instance*

	TCP	BM	DM	DEV	SIM	% in margin	time (min)
Average	***0***	***0***	***0***	***0***	***100***	***100***	***0.41***
Deviation	***0***	***0***	***0***	***0***	***0***	***0***	***0.17***

5.3.2 'Tight' instances

In this Section we present the results of some increasingly tight instances. As PCMP-instances are not easily put together because of the amount of input needed, we restrict the number of instances to 2. Furthermore, by 'tight' we mean an increasing cardinality with respect to the block types of the production need. This is done so we can use the preference rosters of the NNC1-instance and get useful results which we can compare. We try to indicate a point in which the instances become hard to solve in terms of running time. Then a statement can be made about the average-case running-time with respect to the production need and the number of non-restricted drivers, i.e., drivers $ch \in CH$ with $ab(ch, d) = tb(d)$, $\forall d \in D$.

The results of the runs are shown in table 5.8 and 5.9. If the instance is not solved within the bounds posed by the parameters only the time needed to reach this conclusion is shown, the other cells are left blank. We then try to increase certain parameters in order to solve that specific instance within a 'reasonable' amount of time. In our case reasonable has been stated as approximately two hours. Besides the low parameter set (table 5.2) we use the parameter sets from the tables 5.5 (medium 1), 5.6 (medium 2) and 5.7 (high).

Table 5.5 *Parameter settings (medium 1)*

Parameter	Value
Value-Selection-Rule Used	BestSelect
maxtabu	20
maxbtd	100
maxrtd	10
maxbts	100
maxrts	10

Table 5.6 *Parameter settings (medium 2)*

Parameter	Value
Value-Selection-Rule Used	BestSelect
maxtabu	25
maxbtd	50
maxrtd	15
maxbts	50
maxrts	15

Table 5.7 *Parameter settings (high)*

Parameter	Value
Value-Selection-Rule Used	BestSelect

maxtabu	30
maxbtd	100
maxrtd	25
maxbts	100
Maxrts	25

The original NNC1-instance had a total cardinality of 7 work block types on an average Sunday, 12 on a Saturday and 25 on a weekday. With each new instance we increase this number with 1 on every day. The block type which we increase shall be determined randomly. We number the instances according to the increase with respect to the NNC1-instance in the following way: an increase of 1 is indicated as NNC11, an increase of 2 as NNC12 and so on. Note that an increase of 10 or more leads to an infeasible solution because there are only 34 drivers available (see also Section 5.1.1 and definition 2.11, initial feasibility). We run an instance five times with the same set of parameters. If not all five runs were successful we run the instance again for five or three times, depending on the total running time, with another parameter set, effectively increasing the number of search states (definition 4.6). Furthermore, if all five runs were successful, we also run the algorithm again a number of times with increased values of the parameters in order to investigate the effect on the efficiency in terms of speed and quality of the evaluation concepts.

Table 5.8 *Results of twenty runs of the algorithm on the NNC11-instance*

	Instance	**Parameter setting**	**TCP**	**BM**	**DM**	**DEV**	**SIM**	**% in margin**	**time (min)**	**success**
Average	NNC11	***low***	***180.20***	***18.25***	***-1.86***	***1.79***	***78.28***	***100***	***1.02***	***5 out of 5***
Deviation			***1.28***	***0.28***	***0.15***	***0.24***	***0.18***	***0***	***0.48***	
Average	NNC11	***medium 1***	***179.37***	***18.39***	***-1.93***	***1.95***	***78.25***	***100***	***1.57***	***5 out of 5***
Deviation			***2.62***	***0.23***	***0.05***	***0.26***	***0.20***	***0***	***1.17***	
Average	NNC11	***medium 2***	***180.83***	***18.40***	***-1.81***	***2.05***	***78.17***	***100***	***1.55***	***5 out of 5***
Deviation			***5.06***	***0.13***	***0.08***	***0.18***	***0.41***	***0***	***1.64***	
Average	NNC11	***high***	***179.27***	***18.55***	***-1.87***	***1.92***	***78.20***	***100***	***0.95***	***5 out of 5***
Deviation			***3.13***	***0.18***	***0.03***	***0.22***	***0.32***	***0***	***1.37***	

Table 5.9 *Results of fourteen runs of the algorithm on the NNC12-instance*

	Instance	**Parameter setting**	**TCP**	**BM**	**DM**	**DEV**	**SIM**	**% in margin**	**time (min)**	**success**
Average	NNC12	***low***							***1.73***	***0 out of 5***
Deviation									***0.14***	
Average	NNC12	***medium 1***							***18.18***	***0 out of 3***
Deviation									***0.39***	
Average	NNC12	***medium 2***							***17.78***	***0 out of 3***
Deviation									***0.41***	
Average	NNC12	***high***	237.13	20.24	3.12	2.06	75.60	100	***63.55***	***1 out of 3***
Deviation			0	0	0	0	0	0	***32.97***	

5.4 Summary

In this Section we summarize the results from the preceding cases and try to explain the differences.

5.4.1 Manual versus automatic matching

In this Section we compare the results of the manual matching from Section 5.1.3, figure 5.2 versus the automatic one from Section 5.1.2. The most obvious difference lies in the smoothness of the lines of the two

graphs. The graph from figure 5.2 shows extreme differences between drivers on all measurements except the DM-line.

Solution Quality

One of the objectives of the planners is to keep the DM-count as close to zero as possible, while at the same time keeping the BM-count as low as possible. This implies that the planners try to grant the same amount of idle days a driver has preference for. This idea works as long as each driver has preferred roughly the same amount of idle days. This is enforced by some additional constraints. The algorithm doesn't guide the solution forming process with the explicit BM and DM parameters but instead tries to accomplish the same effect, i.e., a low BM and close-to-zero DM by means of the change penalty system (definitions 4.10 and 4.11). As this system both incorporates the BM and DM concepts per driver the requirement of treating all drivers equal is successfully met. The DEV line in figure 5.1 is close to zero for all drivers indicating little difference between them concerning the average TCP value. Also the SIM line is relatively smooth with respect to the manual matching. This indicates that for all drivers the similarity to their respective preference roster is relatively equal.

With *c* and *as* are the functions defined at definition 2.6 and 2.5 respectively, the values for the variables L and U of the PCMP-solution (definition 2.10, sub 4) were imposed as:

$$L = \frac{\sum_{ch \in CH} \sum_{d \in D} c(as(d,ch),ch,d)}{|CH|} - \left(0.1 * \frac{\sum_{ch \in CH} \sum_{d \in D} c(as(d,ch),ch,d)}{|CH|} \right)$$

$$U = \frac{\sum_{ch \in CH} \sum_{d \in D} c(as(d,ch),ch,d)}{|CH|} + \left(0.1 * \frac{\sum_{ch \in CH} \sum_{d \in D} c(as(d,ch),ch,d)}{|CH|} \right)$$

Meaning that the total penalty for a driver should lie within a 10% margin of the average total penalty for all drivers. Ideally this should be the case for all drivers. However it was also satisfiable if at least 20% of the drivers were within this margin. As can be seen from table 5.3 the manual matching does not accomplish this while the one generated with our algorithm does.

Speed

If we measure the two methods in terms of efficiency, i.e., creation speed and effectiveness, we get the following. The manual result was obtained by two full time employees (fte) in a month. The automatic result can obtained by one fte in at most two hours. So, in terms of creation speed our algorithm is to be preferred. When looking at the effectiveness of the two methods we see that the automatic one treats the drivers a lot more equal than the manual one.

Derived Results

Now we briefly return to the matter of the smoother DM-line from the manual matching as opposed to the more erratic one from the automatic matching.

A negative DM-value of an instance indicates that on average drivers have been assigned more idle days than they preferred. This can only occur when there is no need for them to work. The reason the DM-line of the manual matching is closer to zero lies in the fact that the planners have assigned more blocks than necessary to the drivers, according to the production need. This is done in order to create a buffer which for example can be used in case drivers get ill. These so-called optional reserve blocks are not assigned by our algorithm as they are not a part of the production need. The results of the instances NNC11 and NNC12 give an indication of the performance of the algorithm when the production need is increased. However in these cases the reserve blocks are no longer optional so comparing the performance of the algorithm and the manual method with respect to the DM-line is not straightforward. Nevertheless we try to get an idea of the workings of the algorithm by means of the following observation:

The average number of extra blocks assigned per day by our algorithm is abbreviated as ABL

Table 5.10 *Average DM-value per instance*

Instance	ABL	DM-value
Automatic NNC1	0	-6.91
Automatic NNC11	1	-1.87
Manual NNC1	*1.34*	-0.15
Automatic NNC12	2	3.12

Note the increase of the DM-value by approximately 5. On these basis we expect the ABL-value of the manual NNC1 instance to lie around 1.34 indicating that on average 1.34 extra cardinality is used per period day. With this method one can determine whether a branch-location has a possible overcapacity, i.e., when drivers get assigned too much idle block types the DM-concept is negative and there is overcapacity. As solutions can be found relatively quick for the NNC11 instance, in which there were no optional blocks, we expect that for optional blocks the algorithm will provide an answer in a time period of at most 5 minutes. However the procedure of assigning optional blocks falls beyond the scope of this thesis.

5.4.2 Other cases

Looking at the results of the other, test, cases we can conclude that using the algorithm is preferred above any manual matching in any case, if only because of the creation time. Furthermore, when the results turn out badly for particular instances planners can still use the matching of the algorithm as a starting point.
Because the algorithm only returns matchings which satisfy all the hard constraints. In the worst case the planners have to wait for about an hour and a half before receiving a notification of failure (see table 5.9). This is well within the stated bound of approximately two hours.

5.4.3 Parameter value influence

Before jumping to any conclusions about the observed influence of the value of the parameters we should mention that the amount of presented test data in not enough to draw any hard scientific conclusions. However we can see certain trends. Besides, a starting point for fine-tuning the parameters has been established, see also Section 6.1.1.
Recalling the expectations from Section 5.2 we see that the test data indeed seems to behave as expected, i.e., an increase in the upper bound parameters together with the tabulist parameter leads to an increase in running time. However, with respect to only increasing the upper bound parameter or only increasing the tabulist parameter we can not say very much.
Also the data does not make it clear which of the two, backtracking or restarting has the larger influence on the running time. Our guess would be that the restarting parameters, i.e., maxrtd and maxrts, have the larger influence because these concern iterations of loops on higher levels.
Note that we have not used the random value selection rules, i.e., IRWS and RWS. This is done because it became clear in a series of pre-test runs that the runs with the BestSelect-rules produced structural better results on all defined concepts (definition 5.1) and especially on the requirement of treating the drivers as equal as possible. The reason lies probably in the fact that the BestSelect-rules use less randomization.

6 Conclusions

In this chapter we conclude the discussion of the problem and the algorithm implemented to solve it.
Section 6.1 discusses some efficiency improving measures for the algorithm as well as the CC-module. In Section 6.2 we look at a possible way to generalize the used algorithm and CC-module in order to tackle other slightly similar problems in the same public transport context. Furthermore, an example of such a generalization is presented for a real-life problem.
To conclude this report in Section 6.3 we evaluate on the developed modules and state some recommendations for the use of them.

6.1 Improvements and Future Research

In this Section we treat a number of performance increasing measures and some points which could be researched further in order to make the algorithm more efficient.

6.1.1 Improvements

In this Section we mention some measures which will improve the efficiency of the algorithm but which we have not been able to implement due to time limitations. It should be noted that an increased efficiency in terms of speed is not always necessary. For the NNC1-instance for example, the algorithm is quick enough. But in light of other, more 'tight' instances and the desired generalization we described some possible improvements to our algorithm.

Tuning the parameters
For optimal performance the upper bound parameters should be tuned. In any case these should not be set too low because then the risk of not finding a solution while one possibly exists is large. On the other hand setting them very high can be dangerous in situations where no solution exists. From the test results of Chapter 5 we can infer reasonable settings for the upper bound parameters and for the tabulist parameter. Setting this parameter at a high value can result in extremely large running times while still not returning a feasible solution. On the other hand, setting this parameter too low can result in ignoring parts of the solution space which contain feasible solutions. It should also be noted that for different instances of the problem different parameter settings may be necessary. For example, for the NNC12-instance even the high parameter setting looks inadequate, but for the NNC1-instance the low setting is already good enough in producing solutions in an acceptable time period.

Variable Selection Rule
In a CSA-method it is common to use a most constrained variable selection rule together with a least constraining value selection rule in order to keep the number of feasible future matches maximal in each step. In essence we use a weighted value selection rule. Having a high or low weight or penalty in our case, corresponds to being either most or least constrained. For the variables we used a random selection rule in some cases together with a bottleneck list with an order-sensitive selection method. A most constrained variable selection rule can be used to try to improve the running time of the algorithm for feasible instances. The bonus value (definition 2.2) of a block type pb $\in$ PB, *bbw*(pb), should then be used to indicate the restrictedness of a variable. The higher the value the more restricted the variable. The bonus value is set by the planners to indicate the non-desiredness of the block type for drivers this method should work. Because through experience planners known which block types are difficult to match and which are not.

Apply consistency checking
In our tree search algorithm all constraints are checked after carrying out a driver assignment. In this way, we can only prove inconsistency for the driver we assigned in the last step. By applying (node) consistency checking, we may be able to prove inconsistency for other unassigned drivers as well. With the additional computational effort we can detect dead ends earlier and therefore reduce the number of unnecessary backtracks of the tree search algorithm [Willemen, 2002].

Empty preference rosters
In Section 5.3.1. we observed that using completely empty preference rosters poses a problem with respect to equally dividing work between drivers. We also saw that when using penalty the mechanism will eventually provide for equal treatment of the drivers. Therefore the idea is to use the bonus-value (*bbw* in definition 2.2) of the assigned block as penalty for the drivers.

6.1.2 Future research

Beside the improvements mentioned in the previous Section there are some other concepts which could make the algorithm more efficient. We mention three such concepts for which we did not have time to research in depth.

Arc consistency check
Currently we check constraints only after an assignment has been made. The improvement mentioned before is with respect to node consistency (definition 4.8). In the same way arc consistency checks (definition 4.9) could be applied to further reduce the number of backtrack steps at the cost of some additional computational overhead [Barták, 2001; Reith, 2001]. The question then arises whether the expected decrease in running time outweighs the increase in overhead in the average case. Furthermore, to use arc consistency checks we need to convert all multi-variable constraints to binary ones (see Section 4.3.1)

Detect conflicting constraints
Here we propose a solution for the problem of conflicting constraints from Section 3.1.3.
Constraints cs1 and cs2 are said to be in conflict if and only if:
$\exists t \in T : (cs1(t) \Rightarrow \neg cs2(t)) \wedge (cs2(t) \Rightarrow \neg cs1(t))$
If conflicting hard constraints occur the algorithm will never find a feasible solution. So it seems a good idea to automatically detect such conflicting constraints before running the algorithm. The straightforward, brute force, way to do this, i.e., by evaluating every possible pair of constraints over all possible matchings, would probably take far too much time. Therefore we should research the question whether conflicting constraints can be identified just by their semantics. The example from Section 3.1.3 and the theory from Reniers [2000] indicates that such an approach might be possible.

Use backjumping
To escape from dead ends we used backtracking. In each step we backtracked at most one matching or day. This can lead to slow running times of the algorithm in the worst case. Therefore we can choose to backtrack multiple matchings and/or days in one step. This technique is called backjumping for obvious reasons [33]. The question which should be researched further is how far to backjump and at what stage in the process, i.e., after several unsuccessful backtracking attempts, or always in which case the need for backtracking is eliminated. A rule of thumb as to how far to jump back can be inferred from the constraints. A frequently occurring span-value or interval length might be useful as a bound. We tested an implementation of the algorithm with a backjump parameter of 7 on the NNC12 instance, i.e., each time a day could not be matched and had to be restarted, we jumped back an entire week and restarted from there. So we only applied backjumping on days. The results of the test are shown in table 6.1 and the graph corresponding to the fifth run is depicted in figure 6.1. Note that the algorithm is now able to solve the instance fairly quickly for all runs. We only performed the test with the ‘high’ parameter set (see table 5.7).

Table 6.1 *Results of five runs of the algorithm using backjumping on the NNC12-instance*

	TCP	BM	DM	DEV	SIM	% in margin	time (min)	success
Average	***243.28***	***20.02***	***3.10***	***1.90***	***74.98***	***100***	***5.13***	***5 out of 5***
Deviation	***1.51***	***0.33***	***0.09***	***0.29***	***0.16***	***0***	***3.52***	

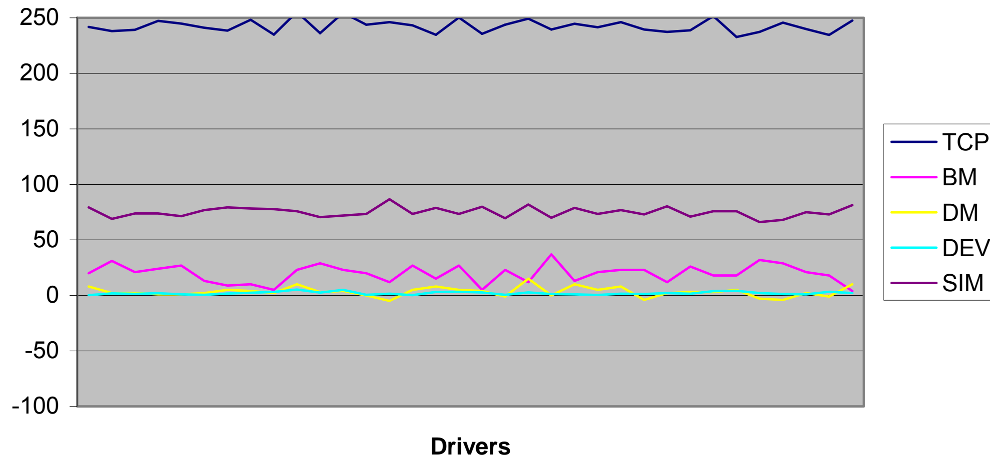

Figure 6.1 *Result of run number 5 of the algorithm using backjumping on the NNC12-instance*

6.2 Generalization of the solution

The developed algorithm is now fairly specific for the PCMP. The idea was to be able to generalize the algorithm in such a way that also other planning problems from the field of public transportation sector could be handled. In this Section we show how the algorithm can be used to solve such a problem which we shall call train planning (TP). We also indicate what needs to be changed in order to make the algorithm suitable to solve TP. Before proposing a mapping from the concepts of the PCMP to these of the TP-problem we give an informal description of the TP-problem.

6.2.1 The TP-problem

For a railroad network used in the public transportation sector we are asked to determine how much material at least is needed in order to meet the production need. The railroad network consists of stations and a number of connections between them. The production need is defined as the number of people outbound on a specific connection of a station on a specific time, for all stations, connections and time-units. One time-unit is equal to one minute.
The material that can be used consists of train components. Each component has a fixed capacity, a fixed cost and some variable costs per driven distance. Furthermore some components can be coupled to other components with a maximum of three components. Also components can only be coupled or decoupled on some specific stations which have a shunting-yard. The cost of the solution, defined in terms of total fixed cost plus total variable costs of the used components should be minimal.

In addition we also need to take into account that:
- in the ideal case the number of people for an outbound connection should be less than the total capacity of the train assigned to that connection for a timeslot but under-capacity within a certain margin is also accepted, i.e., if the capacity is short by a few people it isn't economical to assign an extra component to increase capacity;
- components should be at a shunting-yard during certain timeslots, for example at night.

Given the following:
- the railroad network, consisting of nodes and connections;
- component-types together with their maximum capacity;
- per component-type the fixed cost and the variable cost per kilometer;
- a list defining how much of which components-types can be coupled, together with the time this coupling would take;
- per station the timeslot in which trains can be (de)coupled if this is possible at that particular station;
- for every connection the distance in kilometers. A connection has the same length in both its directions;
- the number of people using a connection in a given direction at certain times defined in a time-table;
- a number of constraints indicating for example the total amount of seating places which must be available on a connection during a particular timeslot, or the relative amount of standing places to seating places which is acceptable on a connection during a certain timeslot.

The question is to calculate the total amount of component-types needed to transport all the people and to fulfill the constraints. Also the total amount of kilometers driven per component and with all this information calculate the total cost price.

6.2.2 Mapping the PCMP to the TP-problem

Using the concepts from the PCMP and the TP-problem the correlation shown in table 6.2 can be observed.
A difference with the PCMP lies in the extra dimension per day, i.e., the timeslots. Also no preference input is provided. The extra dimension can be eliminated by simply adding an indication to the timeslots identifying the days. Furthermore, as we have seen in Section 5.3.1 having a preference input is not a critical requirement. The core of the algorithm now minimizes the penalty per driver using the value selection rules. Following the correlation this would result in minimizing the cost per component where in fact the total cost of all components should be minimized. Note that also the penalty need not be divided equally amongst the components. In fact it is desirable to use as few components as possible as much as possible because the fixed costs of a component always far outweigh the variable cost during the depreciation period. The way to accomplish this is not directly clear but it might be that no changes need to be made whatsoever, i.e., the dynamics of the system will prefer assigning already used components over assigning 'new' ones because the latter choice shall increase the cost more than the former.
Unfortunately the CC-module cannot be used in it's current form without a number of changes to the underlying constraint language. On the other hand no variable constraints are *required* thus the CC-module need not be used at all for this problem.

Table 6.2 *Correlation between PCMP and TP-concepts*

PCMP	**TP**
driver	(train) component
block	connection, with a direction (station~station) := *con*
day	collection of timeslots
production need: - per day: block type & number	production need: - per day - per timeslot: *con* & number
abilities: - per driver - per day: possible blocks	abilities: - per component - per day - per timeslot: possible *con*'s
constraints: - possible blocks on days per driver: depends on other days in context of one or more days	constraints: - possible *con*'s on days and timeslots per component: depends on other days and timeslots in context of two successive days
penalty system: - deviation from preference - violation of soft constraints	penalty system: - too much capacity - variable costs per kilometer - fixed costs per component
criteria: - fulfill production need - minimize penalty per driver - divide penalty equally over drivers - satisfy constraints	criteria: - fulfill production need - minimize total costs of all used components - satisfy constraints

6.3 Evaluation and Recommendations

To conclude this thesis we state some recommendations for the use of the developed algorithm and CC-module.

6.3.1 Evaluation

Concerning the end result of the project we can see that the objectives have been met, i.e., an algorithm has been developed which produces the desired results. Furthermore the method incorporated by the algorithm was chosen in such a way that it should be relatively easy to solve other related problems as well provided an appropriate mapping is made.

6.3.2 Recommendations

Concerning the developed product we recommend it to be further generalized in order for Ovitech to have it as one or two ready-to-use off-the-shelf modules. Also it is recommended to be incorporated in (public transport) companies which make use of duty-rosters, to save on both time and money.

Bibliography

Barták, R. [2001], Theory and practice of constraint propagation, in: *Proceedings of the 3rd workshop on constraint programming in decision and control*, Poland.

Coonen, T.H., C.E. Leiserson and R.L. Rivest [1991], *Introduction to algorithms*, MIT Press, USA.

Daele, P. van den [2002], *Automatische planning van lesroosters*, Eindwerk, Hogeschool Gent, Belgium, preprint.

Blokhuis, A. [1998], *Discrete Optimization, Grafentheorie en discrete optimalisering*, Dictaat 2F540, Eindhoven University of Technology, Eindhoven, preprint.

Eikelder, H. ten, R. van Geldrop, C. Hemerik and G. Zwaan [1999], *Syllabus bij het college Compilers (2M220)*, Eindhoven University of Technology, Eindhoven, preprint.

Ellingsen, K. and M. Penaloza [2003], *A Genetic Algorithm approach for finding a good course schedule*, Technical Report, South Dakota School of Mines and Technology, USA.

Garey, M.R. and D.S. Johnson [1979], *Computers and intractability: a guide to the theory of NP-completeness*, Freeman and Company, New York.

Iersel, H.C.P.M. van [1994], *Time and resource constrained scheduling: a case study in production planning*, Technical Report, Eindhoven University of Technology, Eindhoven, preprint.

Kennedy, J. and R.C. Eberhart [2001], *Swarm Intelligence*, Morgan Kaufmann, USA.

Lewis, H.R. and C.H. Papadimitriou [1981], *Elements of the theory of computation*, Prentice-Hall, New Jersey.

Linz, P. [1997], *An introduction to formal languages and automata*, Jones and Bartlett Publishers, London.

Marques, A.and F, Morgado [1997], *Application of a genetic algorithm to a scheduling assignment problem*, Technical Report, CISUC, Coimbra, preprint.

Nuijten, W.P.M. [1994], *Time and resource constrained scheduling: a constraint satisfaction approach*, Ph.D. thesis, Ponsen & Looijen, Wageningen.

Reith, S. [2001], *Generalized satisfiability problems*, Ph.D. thesis, Julius Maximilians Unversität Würzburg, Würzburg.

Reniers, M.A. [2000], *Programming Logic, Course Notes 2R860*, Eindhoven University of Technology, Eindhoven.

Sastry, K. and D.E. Goldberg [2004], *Designing competent mutation operators via probabilistic model building of neighborhoods*, University of Illinois, Urbana.

Tsang, E. [1996], *Foundations of constraint satisfaction*, Academic Press Limited, London.

Tijms, A. [2003], *Attacking the schedule problem with a genetic algorithm*, Technical Report, University of Leiden, Leiden.

Wezel, A. van [2002], *Concept Plan van Aanpak: Kwantiteiten huis*, Ovitech Research Report E25.10.2-01 Concept PvA Kwantiteitenhuis, Eindhoven, preprint.

Willemen, R.J. [2002], *School timetable construction: algorithms and complexity*, Ph.D. thesis, Eindhoven University of Technology, Eindhoven.

Symbol Index

Basic sets

$\mathbb{N}$	denotes the set of nonnegative integers {0,1,2,...}
$\mathbb{Z}$	denotes the set of integers {...,-1,0,1,...}
$\mathbb{R}^{+}$	denotes the set of positive real numbers {0,...}
$\mathscr{B}$	denotes the set of Booleans {true, false}

Definitions

$a := b$	*a* is by definition equal to *b*
$a :\Leftrightarrow b$	*a* is by definition equivalent to *b*
$S \propto P$	S reduces in polynomial time to P

Operations on sets

$A \bigcup B$	denotes the union of sets A and B
$A \bigcap B$	denotes the intersection of sets A and B
$A \times B$	denotes the Cartesian product of the sets A and B
$\mathcal{P}(G)$	denotes the powerset of the set G
$A \subseteq B$	A is a subset of B
$a \in B$	a is an element of B
$a \notin B$	a is not an element of B
$\| A \|$	denotes the number of elements in A
$A \setminus B$	elements of A that are not in B

Quantified expressions

$\forall_A : B$	true if all elements in A make predicate B true
$\exists_A : B$	true if at least on element in A makes predicate B true
$\#_A : B$	the number of elements in A which make predicate B true
$\uparrow_A : F$	the element of A which gives the largest value for function F
$\otimes_a : B_a$	the *exclusive-or* of all elements in B with number *a*

Logical operators

$a \equiv b$	*a* is by definition equivalent to *b*
$a \wedge b$	denotes the *logical-and* of *a* and *b*
$a \vee b$	denotes the *logical-or* of *a* and *b*
$a \otimes b$	denotes the *exclusive-or* of *a* and *b*
$\neg a$	denotes the *complement* of *a*
$\overline{a}$	denotes the *complement* of *a*

[*a* = *b*]	returns 1 if *a* equals *b* and 0 otherwise
a ⇔ *b*	bi-implication of *a* and *b*

Other symbols

a ++ *b*	denotes the concatenation of *a* and *b*
X*	denotes zero or more occurrences of X
Ø	denotes the empty set
[ ]	denotes the empty list
#	"the number of"

Functions

max (*a*, *b*)	returns the maximum of *a* and *b*
min (*a*, *b*)	returns the minimum of *a* and *b*

Appendices

Appendix A - **Constraint Attribute Grammar**

Appendix B - **Constraint Storage Database**

Appendix A

Constraint Attribute Grammar

The constraint attribute grammar (CAG) is given by:

- Non-terminals with attribute directions, with $1 \leq i \leq 8$:

CONSTRAINT<-Nametypebag>
STAT <-Nametypebag>
EXPR <-Nametypebag, +Type>
EX_i <-Nametypebag, +Type>
DENOT <+Type, +Value>
VAR <-Nametypebag, +Type, +Name>
DATE <+Type, +Value>
SET <+Type, +Value>
INTERVAL <+Type, +Value>
SPAN <+Type, +Value>
PREDEF <-Nametypebag, +Type>
OP<+Optype>

- Attribute domains:

Name = { sv | sv $\in$ String}

Type = {inttype, booltype, stringtype, datetype, settype(t), numinterval, dateinterval, timetype}
With t = {numtype, stringtype, datetype}

Optype = {equivales, orop, andop, lessthan, atmost, morethan, atleast, equals, plus, minus, times, negop }

Value = <u>sum</u> (iv: $\mathbb{Z}$, bv: Bool, sv: String)

Nametypebag = <u>bag</u> <u>of</u> (*Name* x *Type*)

Doptypes : <u>bag</u> <u>of</u> (Optype x Type x Type x Type) =
[(equivales, booltype, booltype, booltype),
(orop, booltype, booltype, booltype),
(andop, booltype, booltype, booltype),
(lessthan, inttype, inttype, booltype),
(lessthan, datetype, datetype, booltype),
(lessthan, timetype, timetype, booltype),
(atmost, inttype, inttype, booltype),
(atmost, datetype, datetype, booltype),
(atmost, timetype, timetype, booltype),
(morethan, inttype, inttype, booltype),
(morethan, datetype, datetype, booltype),
(morethan, timetype, timetype, booltype),
(atleast, inttype, inttype, booltype),
(atleast, datetype, datetype, booltype),
(atleast, timetype, timetype, booltype),
(equals, inttype, inttype, booltype),
(equals, stringtype, stringtype, booltype),
(equals, datetype, datetype, booltype),
(equals, timetype, timetype, booltype),
(plus, inttype, inttype, inttype),
(minus, inttype, inttype, inttype),
(times, inttype, inttype, inttype)]

Moptypes: bag of (Optype x Type x Type) =
[(negop, booltype, booltype),
  (minus, inttype, inttype)]

- Terminals and startsymbol as with CFG.

- Guarded production rules, with the exception of the OP-non-terminals as these can be easily derived from the *Doptypes* and *Moptypes* attribute domains. Underscored symbols are terminals. $\{ X \}^{b}_{j=a}$ stands for zero or more instances of X, each instance is uniquely identified by the symbol j, ranging from *a* up to and inclusive *b*:

CONSTRAINT <-d> → STAT <-d> ~

STAT <-d> → if EXPR<-d, +t> then EXPR <-d, +t>
t = booltype

EXPR <-d, +t> → EX_1 <-d, +t>

EX_i <-d, $+t_0$> → EX_{i+1} <-d, $+t_1$> { OP_i <$+os_j$> EX_{i+1} <-d, $+t_j$>$\}^n_{j=2}$
$1 \le i < 7$
$r_1,..,r_n$: $r_1 = t_1$
$\wedge$ ($\forall_j : 2 \le j \le n$: (os_j, r_{j-1}, t_j, r_j) in *Doptypes*)
$t_0 : t_0 = r_n$

EX_7 <-d, $+t_0$> → OP_7 <+os> EX_7 <-d, $+t_1$>
t_0 : (os, t_1, t_0) in *Moptypes*
→ EX_8 <-d, +t>

EX_8 <-d, +t> → DENOT<+t, +v>
→ VAR <-d, +t, +n>
→ (EX_1 <-d, +t>)
→ PREDEF <-d, +t>

DENOT <+t,+v> → NUMBER <+t,+v>
→ STRING <+s>
t : t = stringtype
v : v = sv(s)
→ DATE <+t, +v>

VAR <-d, +t, +n> → ID <+n>
t : <n, t> in d

DATE <$+t_0$, +v> → / NUMBER <+t, $+v_1$>/ NUMBER <+t, $+v_2$> /
NUMBER<+t, $+v_3$>
t : t_0 = datetype
v : v = dv(v_1,v_2,v_3)

SET <$+t_0$, $+v_0$> → { DENOT <+t, $+v_1$> { ; DENOT <+t, $+v_j$> $\}^n_{j=2}$ }
t : t_0 = settype(t)
v : v_0 = (v_1, .. , v_n)

INTERVAL <$+t_0$, +v> → [NUMBER <+t, $+v_1$>; NUMBER <+t, $+v_2$>]
t : t_0 = numinterval
v : v = (v_1 , v_2)
→ [DATE <+t, $+v_1$> ; DATE <+t, $+v_2$>]
t : t_0 = dateinterval
v : v = (v_1 , v_2)

SPAN <+t, +v> → / NUMBER <+t, +v>

PREDEF <-d, +t_0, +v> → IN<+v_1> DENOT<+t, +v_2> SET <+t, +v_0>
t : t_0 = booltype
v : v = v_1

→ IN<+v_1> VAR<-d, +t, +v_2> SET <+t, +v_0>
t : t_0 = booltype
v : v = v_1

→ COUNT<+v_1> INTERVAL <+t, +v_0> EXPR<-d, +t>
t : t_0 = inttype
v : v = v_1

→ COUNT<+v_1> SPAN <+t, +v_0> EXPR<-d, +t>
t : t_0 = inttype
v : v = v_1

→ RUST<+v_1> INTERVAL <+t, +v_0>
t : t_0 = timetype
v : v = v_1

→ RUST<+v_1> SPAN <+t, +v_0>
t : t_0 = timetype
v : v = v_1

□

Appendix B

Constraint Storage Database

This model is in fact part of a larger database hence the 'dangling relation' originating from the table tbl_rule_recRule. The expression method used to depict the model is that of MS Visio 2002. The model is easily read as an Entity Relationship (ER)-diagram as follows.
Each table can be seen as an entity with a name specified in the gray section of each table. The bold printed name denotes the primary key of the entity while the other, non-bold names are the remaining attributes of the entity. Regarding the relations: an arrow from table A to B denotes a one-to-many relationship from table B to A, therefore the primary key of table B is added to the attributes of table A as a so-called foreign key.