SICS Technical Report T2000:05 ISSN: 1100-3154
SICS-T--2000/05-SE

Knowledge-Based Locomotive Planning for the Swedish Railway

by

Volker Scholtz

Swedish Institute of Computer Science
Box 1263, S-16429 Kista, SWEDEN

Knowledge-based Locomotive Planning for the Swedish Railway

Volker Scholz

December 1998

Prolog

This work was done during my half-year stay at SICS (Swedish Institute of Computer Science) in Stockholm. I want to thank my colleagues Per Danielsson, Per Kreuger, Thomas Sjöland and Emil Åström for their interest and their comments regarding my work. Thanks to Jan Olsson, the head of the COL department for the offer to do a M.Sc. thesis in his group in connection with the TUFF project.

My supervisor Per Kreuger was always interested in discussing new ideas, although he was busy with other projects. He encouraged me to continue the work also in difficult situations through his positive way of thinking. I remember him as a good friend.

I also want to thank Mats Carlsson from the ISL department for proof-reading parts of the work. Thanks to Prof. Dr. Volker Claus for his role as examiner, his interest in my work and the time he spent on my problems. Dipl. Inform. Friedhelm Buchholz supervised the work in the final phase in Stuttgart and gave some valuable comments.

I really appreciated the familiar and liberal atmosphere at SICS. I was warmly welcomed as a foreigner and got some taste of international research. It was really a good time at the end of my days of study.

Sjung om studentens lyckliga dag,
Låtom om oss fröjdas i ungdomens vår!
Än klappar hjertat med friska slag,
Och den ljusnande framtid är vår.

Prins Gustafs Studentsång

Contents

Chapter 1

Introduction

1.1 Railway Planning Problems

With the advent of the industrial revolution, railway traffic played an important role in the coal and steel industry. It is still an important factor in the transportation system of modern countries in the information age. Passenger transport could gain new customers by high-speed trains like the ICE in Germany or the X2000 in Sweden, whereas freight transport meets hard competition from the road carriers. The competition becomes also harder due to the decay of monopolistic structures and deregulation.

Planning problems in railway traffic where always of great interest to railway companies due to economical reasons. They need computer-aided tools for the most efficient use of their resources. Transport planning has also an ecological dimension, the great amount of transport tasks should be performed in the most energy efficient way.

In the planning of rail traffic one has to deal with the following problems [BWZ97]:

- train scheduling
- rolling stock scheduling
- personnel scheduling
- rescheduling

We will describe the different subproblems briefly [BWZ97].

Train scheduling is the determination of the departure and arrival times of the trains at the stations. Passenger trains have periodic departure times

whereas schedules for freight transport are adjusted to customer demand. There are safety restrictions (two trains must keep a certain distance) and capacity restrictions, e.g. the number of platforms at stations. The allocation of tracks to trains must be planned.

Rolling stock scheduling deals with the assignment of locomotives and cars to trains. They are often stored in depots and must be maintained after certain time intervals. In this work, we will look at the assignment of locomotives to trains.

Personell scheduling is the assignment of engine-drivers and accompanying staff to trains. Legal restrictions like working times, breaks etc. have to be taken into account.

Rescheduling is the generation of adjusted schedules after delays of trains, in case of heavy traffic or technical failure. A dispatcher must be able to recompute parts of the schedule on-line so that a fast reaction to events in the network is possible. A complete reoptimization is in this real-time situation often not possible, and local improvement heuristics are used.

Before these planning tasks can be addressed, some strategic decisions have to be made. The structure of the network, routes and lines have to be determined. Although the network is a result of its historical development, decisions about new tracks have to be taken. The railway network can be divided into different subsystems: long-distance trains and regionally operating trains. Another strategic decision is the number of engines and cars that shall be used in the long-term.

Todays railway companies often follow a top-down approach in their planning tasks [BWZ97]. The planning of the network and routes is followed by train scheduling, and train scheduling followed by rolling stock and personnel planning. This leads to subproblems of manageable size. Additionally, the different planning horizons of the subtasks are reflected in this approach. One begins with long-term strategical planning, followed by tactical and operational planning. This problem decomposition has also disadvantages. It is only possible to optimize the solutions for the different isolated problems. It is difficult to obtain an optimum solution for the whole problem.

1.2 The Swedish Railway

Swedish State Railways SJ [1] operates passenger and freight trains in Sweden. Fig. 1.1 shows the Swedish railway network for freight transport. Large parts of the network consist of single tracks due to the large distances. These

[1] Statens Järnvägar [SJ]

can only be occupied by one train at a time and are used in both directions. Track capacity is therefore a special issue in the Swedish network.

In the Swedish freight transport system, there are long-distance trains between 11 cities. [2] These are the centers of so-called production areas. The transport of the cargo from a production area to its center is done by regional trains, so-called terminal trains.

1.3 The TUFF Project

The TUFF [3] project is a cooperation between the Complex Operations Laboratory at SICS [4] and SJ. Its aim is to investigate how information technology can make the train planning process for freight transport at SJ more efficient and is funded by SJ. In the TUFF 2 project, a scheduler for the allocation of tracks to trains has been developed. We will give a formal description of this planning task in Chapter 3. The scheduler uses constraint programming technology and is implemented in the constraint language Oz 2. One research goal is to study the feasibility of constraint programming techniques in the railway scheduling domain. Constraint programming has already been successfully applied to real-sized problems in scheduling and transport planning [Sim96].

The program has a graphical user interface with windows for the railway network and the plan parameters (see Fig. 1.1, 1.2). Partial plans for certain routes in the net can be visualized in so-called train sequence diagrams (see Fig. 1.3), where the x-axis represents time and the y-axis the intermediate locations between a station pair. These time-space diagrams are used by railway companies for an illustration of the timetable. The whole plan can be inspected by several such diagrams by choosing the stations pairs in the map.

An interesting feature of this planning system is the possibility to extend an existing timetable by an additional plan specification for new trains so that the planner can generate an extended plan. If this process is iterated, one can extend the timetable in every step by a moderate number of new trains. With this technique, planning problems with several hundred trains can be handled.

The generated plans are not optimum solutions with respect to parameters like the total plan time and the waiting times for the trains, but feasible

[2] Borlänge, Gävle, Göteborg, Helsingborg, Luleå, Malmö, Nässjö, Stockholm, Sundsvall, Umeå and Örebro [SJ 98]

[3] **T**ågplane**u**tveckling **f**ör **f**ramtiden, Train Planning for the Future

[4] Swedish Institute of Computer Science [SIC]

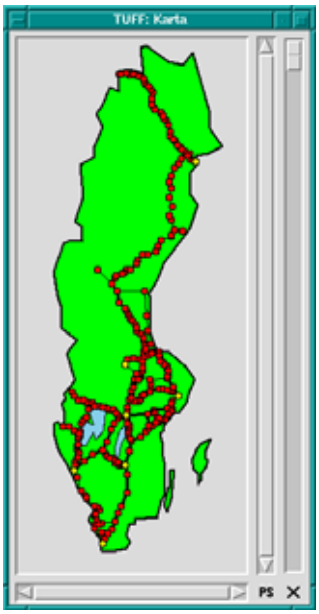

Figure 1.1: SJs freight transport network

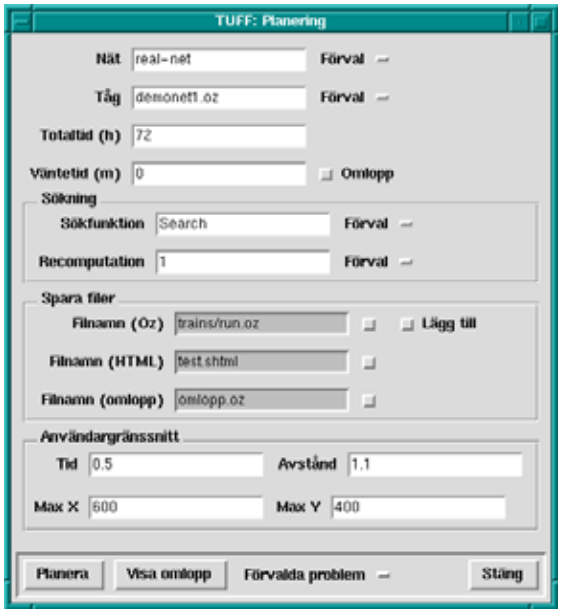

Figure 1.2: The window for the plan parameters

plans which respect upper bounds for these parameters can be generated quickly [KCO+97].

The purpose of this work is to extend the TUFF system by a planner for the assignment of locomotives to transports. In the common hierarchical approach, the track allocation problem is solved in a first step and leads to a timetable with fixed departure times. For this timetable, the locomotive assignment problem is solved.

The goal of this work is to integrate the track allocation and locomotive assignment tasks. We hope to generate better solutions by coordinating these mutually dependent tasks. Constraint Programming provides the possiblity to add new constraints to the ones which already exist for track allocation.

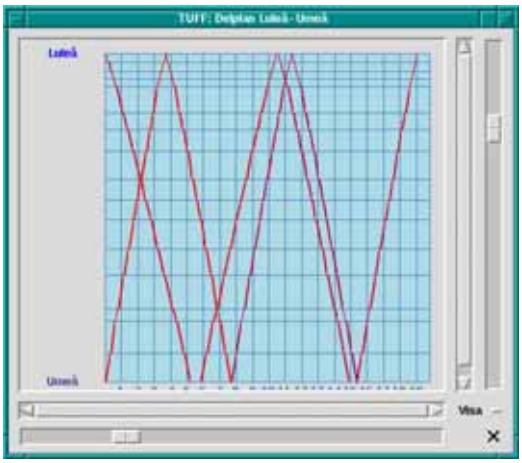

Figure 1.3: A train sequence diagram

1.4 Locomotive Assignment

The objective of the locomotive assignment problem is to assign locomotives to trains as cost-efficient as possible.

We define a *trip* as a transport on which is not possible to change the locomotive. The number of weekly trips at SJ is more than 3000 for freight traffic as well as passenger traffic [DHKK97]. A trip requires at least one locomotive, sometimes two and can be run by more locomotives than are actually needed. The required locomotives are called *active locomotives* while the extra locomotives are *passive*. It can be useful to move a locomotive to the start location of a new trip by running it as a passive locomotive in a trip. One can also move a locomotive by running it without any cars.

This is called a *passive transport* and it is more expensive than a passive locomotive. We call the trip order which a locomotive has to serve a *route*.

Different trips require different locomotive types (different speeds, horsepower etc.). Every trip is compatible with a subset of locomotive types. The locomotive types rc1–rc6 at SJ are hierarchically ordered, where a locomotive of type rc6 can serve any trip. A locomotive of a certain type in this hierarchy can run any trip in the same or a lower class. Additionally, there are the classes X2000, Radio and Diesel. Trips which require one of these classes require one particular locomotive type, i.e. the assignment problem decomposes by the locomotive type and can be solved for each type separately.

At the start and end location of a trip a certain amount of time is needed to change the cars between two trips which are served by the same locomotive. This process is called *docking* which needs a certain amount of *turn time*.

There are several factors for the cost of a locomotive schedule, we order them after their importance:

- the number of locomotives
- the amount of passive transport kilometers
- the waiting times of the locomotives

The number of locomotives is the largest cost factor. There is a fixed cost for using a locomotive in a schedule. Additionally, SJ pays a fee to the track operator (Banverket) depending on the number of locomotives used. Unused locomotives can sometimes be rented to other railway companies.

Passive transports are not directly productive and require locomotive and staff resources. They consume also track capacity, so that they should be avoided. Obviously, a smaller number of locomotives leads to an increase in passive transport and a balance between these two cost factors must be achieved. We will define the cost function in Chapter 3.

If there are restrictions on the departure times of the trips, waiting times for the locomotives can occur. The time a locomotive has to wait until it can run the next transport is not productive and should also be minimized.

1.5 Overview

In the next chapter, we will give some basic mathematical definitions. Chapter 3 contains a formal definition of the whole planning problem, i.e. the

track allocation and locomotive assignment problem. After we have defined our problem, we look at several approaches to vehicle routing problems in the literature in Chapter 4. Chapter 5 provides a short introduction to constraint programming. In Chapter 6, we present a constraint model for the locomotive assignment problem which can be added to the TUFF system. This constraint model uses a new geometric constraint which is not available in Oz 2 and a propagation algorithm for this constraint is presented in Chapter 7. We discuss several heuristics for the generation of locomotive routes in Chapter 8. Chapter 9 describes the implementation part of this work. Chapter 10 contains experiments on problem sets for the Swedish network and the last chapter gives an outlook on further problems.

Chapter 2

Mathematical Foundations

2.1 Basic Definitions

$\mathbf{N}$ denotes the set of the non-zero natural numbers and $\mathbf{N}_0$ contains also the number zero. $\mathbf{Z}$ denotes the set of integers and $\mathbf{R}$ the set of reals. $\mathbf{R}^+$ denotes the positive real numbers including zero. We use a special notation for the set of all subsets of a certain size. Let $A \neq \emptyset$, $|A| = n$ and $0 \leq k \leq n$:

$$\binom{A}{k} = \{S \subseteq A || S| = k\}$$

2.2 Graphs

A graph $G = (V, E)$ is given by a set of nodes V and a set of edges E. It is directed if $E \subseteq V \times V$ and undirected if $E \subseteq \binom{V}{2}$. Our graphs contain no loops, i.e. $\forall v \in V : (v, v) \notin E$ in the directed case and $\forall v \in V : \{v\} \notin E$ in the undirected case.

A multigraph $G = (V, E, \omega)$ contains additionally a function ω that maps every edge to its two incident nodes. If ω is not injective, the graph contains multiple edges between nodes. The multigraph is directed if $\omega : E \to V \times V$ and undirected if $\omega : E \to \binom{V}{2}$. We use loop-free multigraphs, i.e. $\forall e \in E \quad \forall v \in V : \omega(e) \neq (v, v)$ in the directed case and $\forall e \in E \quad \forall v \in V : \omega(e) \neq \{v\}$ in the undirected case.

2.3 Network Flows

We introduce two network flow minimization problems in this section. The notation is taken from [AMO93].

2.3.1 The Minimum Cost Flow Problem

Let $G = (N, A)$ denote a directed graph over which we want to send units of flow. Every arc $(i, j) \in A$ has a maximum capacity $u_{ij} \in \mathbf{R}^+$ and an associated cost $c_{ij} \in \mathbf{R}^+$. [1] For every node $i \in N$, the number $b(i) \in \mathbf{R}$ represents its supply/demand function or divergence. The amount of flow in the network is constant: $\sum_{i=1}^{n} b(i) = 0$. Nodes with $b(i) > 0$ are supply nodes, nodes with $b(i) < 0$ are demand nodes and nodes with $b(i) = 0$ are transshipment nodes. The goal is to determine the flow $x_{ij} \in \mathbf{R}^+$ on every arc $(i, j) \in A$ which fulfills the following conditions:

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{2.1}$$

$$\sum_{\{j|(i,j)\in A\}} x_{ij} - \sum_{\{j|(j,i)\in A\}} x_{ji} = b(i) \quad \forall i \in N \tag{2.2}$$

$$0 \leq x_{ij} \leq u_{ij} \quad \forall (i, j) \in A \tag{2.3}$$

Equation 2.1 defines the cost function that we want to minimize. The flow on every arc is weighted with its associated cost. The flow conservation condition is described in Eqn. 2.2, i.e. the amount of out-flow minus the amount of in-flow of a node is equal to its divergence $b(i)$. Eqn. 2.3 limits the amount of flow on every arc to a maximum value.

Today, the most efficient method for solving this problem is a simplex algorithm which is tailored to the network problem [AMO93], [Loe98]. The network simplex algorithm runs theoretically in pseudo-polynomial time [AMO93]. Experiments have shown that on the average, one can assume a low-order polynomial in the number of arcs and nodes [Loe98].

2.3.2 The Multicommodity Minimum Cost Flow Problem

We are given a directed graph $G = (N, A)$, a cost $c_{ij} \in \mathbf{R}^+$ and a maximum capacity $u_{ij} \in \mathbf{R}^+$ for every arc $(i, j) \in A$. We want to represent the flow of K different commodities $1, \ldots, K$ in the network, i.e. the flow units have different type. Let x_{ij}^k denote the flow of commodity $k \in \{1, \ldots, K\}$ on arc $(i, j) \in A$ and $c_{ij}^k \in \mathbf{R}^+$ the associated cost. The vectors $\mathbf{x}^k = (x_{ij}^k)_{(i,j)\in A}$ and $\mathbf{c}^k = (c_{ij}^k)_{(i,j)\in A}$ denote the flow vector and cost vector of commodity k.

[1] We assume that the flow cost varies linearly with the amount of flow over an arc.

$b^k(i)$ denotes the divergence of the node $i \in N$ for the commodity k. The amount of flow of every commodity is constant:

$$\sum_{i=1}^{n} b^k(i) = 0 \quad \forall k \in \{1, \ldots, K\}$$

The multicommodity flow problem can be formulated as follows:

$$\min \sum_{k=1}^{K} \mathbf{c}^k \cdot \mathbf{x}^k \tag{2.4}$$

$$\sum_{\{j|(i,j)\in A\}} x_{ij}^k - \sum_{\{j|(j,i)\in A\}} x_{ji}^k = b^k(i) \quad \forall i \in N \quad \forall k \in \{1, \ldots, K\} \tag{2.5}$$

$$\sum_{k=1}^{K} x_{ij}^k \leq u_{ij} \quad \forall (i,j) \in A \tag{2.6}$$

$$0 \leq x_{ij}^k \leq u_{ij}^k \quad \forall (i,j) \in A \quad \forall k \in \{1, \ldots, K\} \tag{2.7}$$

Eqn. 2.4 defines the cost function. We compute the flow cost for every commodity by computing the dot product $\mathbf{c}^k \cdot \mathbf{x}^k$ and sum over the commodities. The next equation describes the flow conservation condition for each commodity. Eqn. 2.6 states a bundle constraint for each arc, i.e. the sum of flows of all commodities on an arc $(i,j) \in A$ must respect the upper bound u_{ij}. The individual flows of the commodities are restricted by their own upper bounds (Eqn. 2.7).

This problem can be solved by Lagrangean relaxation methods [AMO93]. The run times are not polynomial in general and can degenerate into exponential in the worst-case.

Chapter 3

Problem Definition

We begin this chapter with a formal description of the model for track allocation in the TUFF system [KCO+97]. After that, we extend this by a definition of the locomotive assignment problem. We conclude the chapter with an example in order to explain the definitions.

3.1 Track Allocation

The track allocation problem can be summarized as follows: schedule a set of train trips with predetermined paths over a railway network, where

- paths consist of track sequences and a track connects nodes where trains can meet and overtake.
- the trains must keep a certain safety distance.
- there is a maximum number of waiting trains at each node.
- schedules should respect bounds on the total time required to execute the schedule and on waiting times of the trains.

We begin with a definition of the railway network:

Definition 1 (Network) *The railway network is given by an undirected multigraph* $G_1 = (V_1, E_1, \omega)$ *where* V_1 *is the set of* locations, $E_1 = \{e_1, \ldots, e_\alpha\}$ *the set of* tracks *and* $\omega : E_1 \to \binom{V_1}{2}$.

The capacity function $\sigma : V_1 \to \mathbf{N}$ *denotes the capacity of a location, i.e. the maximum number of trains that can be there at the same time. The function* $v : E_1 \to \mathbf{R}^+$ *defines a maximum velocity per track,* $\delta : E_1 \to \mathbf{R}^+$ *is the*

track length and $s' : E_1 \to \{0, 1\}$ *labels a track as double* (0) *or single* (1). *A double track can be used simultaneously by two trains running in opposite directions, a single track cannot.*

The set of routes $R = \{r_1, \ldots, r_\beta\}$ *defines the paths of the trains in the network. Given two locations* u, $v \in V_1$, *there is exactly one path which leads from* u *to* v. *A route* r_i *is given by its start location* $S(r_i) = u$, *its end location* $E(r_i) = v$ *and* $\epsilon = \epsilon(r_i)$ *tracks which are traversed by the route:* $r_i = (e_{i_1}, \ldots, e_{i_\epsilon})$ *where* $|\omega(e_{i_j}) \cap \omega(e_{i_{j+1}})| = 1 \quad \forall j \in \{1, \ldots, \epsilon - 1\}$ *(connectedness) and* $S(r_i) \in \omega(e_{i_1})$, $E(r_i) \in \omega(e_{i_\epsilon})$. *The track order induces a start location* $S(e_{i_j}, r_i)$ *and an end location* $E(e_{i_j}, r_i)$ *for each track that is contained in the route* ($1 \leq j \leq \epsilon$), *depending on the direction in which it is traversed.*

We need a multigraph representation of the network, because two locations can be connected by several tracks (e.g. by a single and a double track). The set of locations contains stations but also intermediate points in the network where trains can meet. The capacity function for the locations is necessary because they have a limited number of parallel tracks, this limits the number of trains that can wait simultaneously at a location. Most locations in the Swedish network have two parallel tracks.

The properties of a track are its maximum velocity, its length and the attribute single or double. A single track corresponds to one track in reality, it can't be used simultaneously by two trains running in opposite directions, but by two trains with the same direction and a safety distance. It is used for trains traveling in both directions. A double track corresponds to two tracks in the physical network, one for each direction. Trains of opposite direction can meet at a double track.

The set of routes R is an input to the planning system. The problem of generating suitable transport paths in the network is not addressed in this work. We assume that the paths are already known. Our routes are simple paths, i.e. a track can't be traversed several times by a route. Transports with loops must be decomposed into several single paths.

The railway network used in the TUFF system consists of 264 locations, 336 tracks and 126 example routes.

Definition 2 (Planning problem) *The input for the planning of track allocation is a set of* trips $P = \{p_1, \ldots, p_n\}$. *A trip is given by a start location* $S(p_i)$, *an end location* $E(p_i)$ *and a route* $r(p_i) \in R$ *connecting them.* $\nu : P \to \mathbf{R}^+$ *defines the maximum speed of a trip. Every trip has a time window* $[\tau_{min}(p_i), \tau_{max}(p_i)]$ *for the departure time of the trip where* $\tau_{min}(p_i)$, $\tau_{max}(p_i) \in \mathbf{N}_0$ *and* $\tau_{min} \leq \tau_{max}$. *The time window* $[w_{min}(p_i, v), w_{max}(p_i, v)]$ *denotes the minimum and maximum waiting time for each location* $v \in V_1$

which is visited by trip p_i. For every location $v \in V_1$, we have additionally a window $[w_{min}(v), w_{max}(v)]$ for a location dependent minimum and maximum waiting time.

As an additional input, some parameters for the computed timetable must be given:

- *$[s_{min}, s_{max}]$ is a global time window for the departure time of all trains.*
- *the location slack factor μ*

We call the difference between the actual and the minimum waiting time at a location slack. *The* location slack factor *μ with $0 \leq \mu \leq 1$ is the ratio of slack at a location and the time for the following track traversal. The slack is bounded by μ for all locations and trips in the problem.*

The input to the planning problem is a set of trips (or trains). [1] Each trip has a maximum velocity which depends on the route and the train weight. The freight trains in the Swedish network have velocities between 80 and 120 km/h [SJ 98]. There is also a time window for the departure time of a trip which is given by customer demands. The time points are represented as natural numbers, i.e. we use a discrete time scale (minutes). A trip can require certain waiting times at intermediate locations on its route in order to allow docking operations. This can be specified by time windows for the minimum and maximum waiting time at these locations. Every location has also an individual time window for the waiting times.

A specification of a planning problem contains also parameters for the generated timetable. There is a time window for the whole plan defining the planning horizon. The waiting times of the trains at the stations are a quality measure for the generated timetable and ensure reasonable travel times for the trains. One can specify an upper bound for the slack at locations, i.e. the amount of waiting time which exceeds the minimum waiting time.

Definition 3 (Timetable) *We have again n trips $P = \{p_1, \ldots, p_n\}$. A trip consists of a sequence of track traversals, so-called* tasks. *Let $t_{i,j}$ denote the traversal of track $e_i \in E_1$ by the trip p_j with the route $r(p_j) = (e_{j_1}, \ldots, e_{j_\epsilon}) \in R$ with $\epsilon = \epsilon(r(p_j))$. Let $I(p_j) = \{j_1, \ldots, j_\epsilon\}$ denote the set of the indices of the edges traversed by trip p_j. $w_{i,j} \in \mathbf{N}_0$ denotes the waiting time of the train in the start location $S(e_i, p_j)$ of the track and $s_{i,j} \in \mathbf{N}_0$*

[1] We look at the freight transports without a determined schedule. Passenger and freight transports with fixed departure times could also be included into our model (with a departure time window of zero length), but we won't consider them further.

its departure time at this location. $d_{i,j} \in \mathbf{N}_0$ *is the duration of the track traversal which is given by*

$$d_{i,j} = \lceil \frac{\delta(e_i)}{\min\{\nu(p_j), v(e_i)\}} \rceil$$

$h_{i,j} \in \mathbf{N}_0$ *is the headway that must be respected between trains which travel in the same direction on this track. The headway is a constant fraction of the duration of the track traversal:* $h_{i,j} = \gamma d_{i,j}$ *with* $\gamma = 0.15$*. The departure time* $s(p_j)$ *of a trip* p_j *is the departure time on its first track:* $s(p_j) = s_{j_1,j}$*.*

The computed timetable consists of all start times $s_{i,j}$ *and waiting times* $w_{i,j}$ *and must respect the following constraints:*

- *time bounds:*
  - $\forall j \in \{1, \ldots, n\}$: $s(p_j) \in [s_{min}, s_{max}]$
    (time window for the whole plan)
  - $\forall j \in \{1, \ldots, n\} \quad \forall i \in I(p_j)$:
    $w_{i,j} \in [w_{min}(p_j, v), w_{max}(p_j, v)] \cap [w_{min}(v), w_{max}(v)]$
    where $v = S(e_i, p_j)$*.*
    (minimum, maximum waiting time per location)
  - $\forall j \in \{1, \ldots, n\} \quad \forall i \in I(p_j)$:
    $w_{i,j} - \max\{w_{min}(p_j, v), w_{min}(v)\} \leq \mu d_{i,j}$
    where $v = S(e_i, p_j)$*.*
    (maximum slack per trip and location)
- *trip constraints:*
  - $\forall j \in \{1, \ldots, n\} : s(p_j) \in [\tau_{min}(p_j), \tau_{max}(p_j)]$
    (time window for each trip)
  - $\forall j \in \{1, \ldots, n\} \quad \forall k \in \{1, \ldots, \epsilon - 1\}$:
    $s_{j_{k+1},j} = s_{j_k,j} + d_{j_k,j} + w_{j_{k+1},j}$
    where $I(p_j) = \{j_1, \ldots, j_\epsilon\}$*.*
    (the tracks are traversed in the order given by the routes)
- *location constraints:*
  Let $\Pi : P \times \mathbf{N}_0 \longrightarrow V_1 \cup E_1$ *denote the function that maps a trip* $p \in P$ *and a time point* $t \in \mathbf{N}_0$ *to the location or track where the corresponding train is located at time* t*. We can formulate the location constraints as:*

$$\forall v \in V_1 \quad \forall t \in \mathbf{N}_0 : |\{p \in P | \Pi(p, t) = v\}| \leq \sigma(v)$$

- *track constraints: We must distinguish two cases:*

- *the trips p_k and p_l traverse the single track e_i in* opposite *directions*
  ($k < l \wedge i \in I(p_k) \cap I(p_l) \wedge S(e_i, p_k) \neq S(e_i, p_l) \wedge s'(e_i) = 1$):

  $$(s_{i,k} + d_{i,k} \leq s_{i,l}) \vee (s_{i,l} + d_{i,l} \leq s_{i,k})$$

  There is no such restriction for opposite directions and double tracks.
- *the trips p_k and p_l traverse the track e_i in the* same *direction*
  ($k < l \wedge i \in I(p_k) \cap I(p_l) \wedge S(e_i, p_k) = S(e_i, p_l)$):

  $$\begin{aligned} & (s_{i,k} + \max\{h_{i,k}, h_{i,k} + d_{i,k} - d_{i,l}\} \leq s_{i,l}) \\ \vee \quad & (s_{i,l} + \max\{h_{i,l}, h_{i,l} + d_{i,l} - d_{i,k}\} \leq s_{i,k}) \end{aligned}$$

For the computation of the track traversal time $d_{i,j}$, we have to consider the maximum velocity specifications for the trip and the track. The other input parameter is the track length. We get identical traversal times for both traversal directions. This is a simplification because we can have different traversal times in reality if the track has a certain slope.

The headway $h_{i,j}$ ensures that trains traveling in the same direction have a certain safety distance. This is approximated by a constant fraction of the traversal time for the track. In reality, the headway corresponds to the distance of two signals on the track. A track is divided by its signals into several segments. One could also model all these track segments in the network model, but this would lead to a significantly larger net. Thus, the headway abstraction was introduced.

The constraints for the time bounds include the plan time window and the waiting times. For the waiting time at locations, we must consider the trip and location dependent time windows and take their intersection. The amount of slack, i.e. the amount of waiting time which exceeds the minimum waiting time is bounded for every location by the location slack factor μ. The trip constraints include the time window for the departure time of each trip and the traversal of the tracks in the route specific order. The location constraints state that there's a maximum number of waiting trains at every location and every time instance.

The track constraints ensure that trains with opposite directions can't use a single track simultaneously. They must traverse it sequentially, the two possible orders can be expressed as a disjunction. This is illustrated in Fig. 3.1, where task $t_{i,k}$ is completed before task $t_{i,l}$ on the single track e_i. We illustrate this in a time-space diagram. Task $t_{i,k}$ starts at time $s_{i,k}$ at its start location $S(e_i, p_k)$ and arrives after the track traversal time $d_{i,k}$ in its

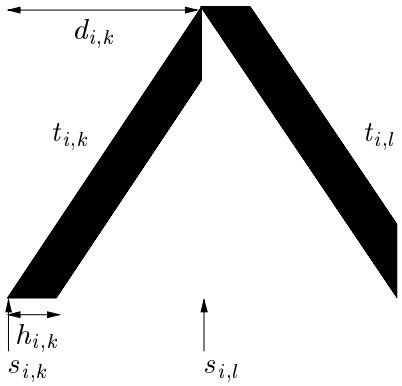

Figure 3.1: Trip p_k traverses track e_i before trip p_l (adapted from [KCO+97])

end location $E(e_i, p_k)$. Due to the headway condition, another trip traveling on track e_i in the same direction must wait the headway time $h_{i,k}$ until it can start. This is shown by the left dark area, it is a *safety area* for task $t_{i,k}$. Task $t_{i,l}$ can start at time $s_{i,l} = s_{i,k} + d_{i,k}$ in the opposite direction and has its own safety area.

Trains which travel in the same direction on a track must respect a certain headway time in the start and end location. This is shown in Fig. 3.2. Observe that the safety areas from Fig. 3.1 have been extended by white triangles, because the headway condition must be respected in the start *and* end location by trains with identical direction. If the first trip p_k is slower than the second trip p_l, we get the headway $h_{i,k}$ (left figure). If p_k is faster than p_l, the second trip must wait longer so that the headway is not violated in the end location. We get the headway $h_{i,k} + d_{i,k} - d_{i,l}$ (right figure). We can combine these two cases into the expression $\max\{h_{i,k}, h_{i,k} + d_{i,k} - d_{i,l}\}$.

3.2 Locomotive Assignment

In the last section, we have described the track allocation problem. For more detailed information about its implementation in the TUFF system, the reader may refer to [KCO+97]. The existing planner in the TUFF system shall be extended by the planning of locomotive assignment which we describe in the following definition.

Definition 4 (Locomotive assignment) *Let $V_2 \subseteq V_1$ denote the start and end locations of the routes R in the network that we have defined (we only work on a subset of all possible routes). We call a location $v \in V_2$* turn location. *We define the directed graph $G_2 = (V_2, E_2)$ with $E_2 = V_2 \times V_2$.*

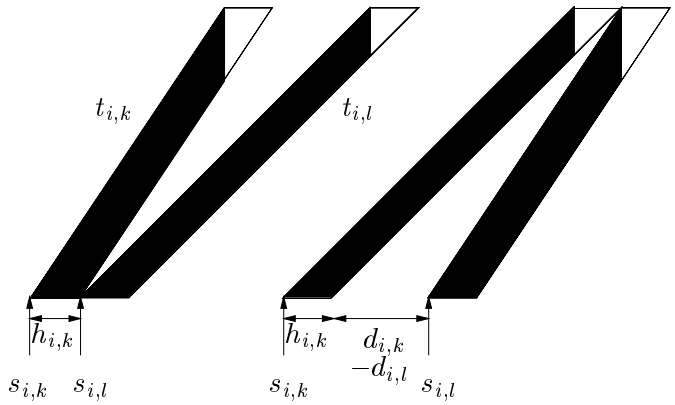

Figure 3.2: Headway $\max\{h_{i,k}, h_{i,k} + d_{i,k} - d_{i,l}\}$ for two trips with different speeds (adapted from [KCO+97])

$\delta_2 : E_2 \to \mathbf{R}^+$ *defines an estimated travel time for a passive transport between a turn location pair.*

Given a set of trips P*, a specification* $l : P \to 2^L$ *for the suitable locomotive types for each trip where* L *denotes the set of locomotive types and a set of locomotives* $M = \{l_1, \ldots, l_m\}$ *with types* $\psi : M \to L$*, we are looking for an assignment* $\phi : P \to M$ *of the trips to the locomotives that respects the locomotive types:*

$$\forall p \in P : \psi(\phi(p)) \in l(p)$$

The assignment ϕ *must fulfill the location continuity constraint. Let* $P = \{p_1, \ldots, p_n\}$ *and* $s(p_i) = s_{i_1,i}$ *is the departure time of trip* p_i *which traverses* $\epsilon = \epsilon(r(p_i))$ *tracks.* $d(p_i)$ *is the traversal time of trip* p_i*:* [2]

$$d(p_i) = d_{i_1,i} + \sum_{k=2}^{\epsilon} (d_{i_k,i} + w_{i_k,i})$$

Let $d(p_i, p_j) = \delta_2(E(e_{i_\epsilon}, p_i), S(e_{j_1}, p_j))$ *denote the travel time for a passive transport connecting the trips* p_i *and* p_j *and* Θ *the turn time which is needed for the docking of cars between transports:*

$$\begin{array}{rll} & \forall p_i, p_j \in P, i < j : & \\ \phi(p_i) = \phi(p_j) \quad \Longrightarrow & & (s(p_i) + d(p_i) + d(p_i, p_j) + \Theta \leq s(p_j)) \\ & \vee & (s(p_j) + d(p_j) + d(p_j, p_i) + \Theta \leq s(p_i)) \end{array}$$

$\Sigma : M \to V_2$ *defines the start locations of the locomotives.* ϕ *must allow a passive transport from the start location of a locomotive to the start location*

[2] We assume that there's no waiting time in the start location $S(p_i)$ of trip p_i.

of its first trip if these are different. Let $p' = \arg\min_{\{p \in P | \phi(p) = l'\}} s(p)$ *be the first trip that is served by the locomotive* $l' \in M$*:*

$$\forall l' \in M : \Sigma(l') \neq S(p') \Longrightarrow \min_{\{p \in P | \phi(p) = l'\}} s(p) \geq \delta_2(\Sigma(l'), S(p'))$$

ϕ *shall be optimized with respect to the following parameters:*

- *minimum number of used locomotives*
- *minimum total passive transport time for the minimum locomotive number*
- *minimum amount of waiting time for the locomotives*

We call the subset of locations where trips can start and end *turn locations*. The travel times δ_2 for the passive transports are calculated from the route length between the turn location pair in the network and a constant velocity. This is an estimation since we can't be sure that the route is free. Every trip can be served by a subset of locomotive types so that we must consider the locomotive types in the assignment ϕ. The locomotive type requirement of a train depends on the weight of its cars and the route on which it must travel.

The location continuity constraint ensures that the locomotives have enough time to get to the start location of the next trip. There must be enough time for a necessary passive transport and for the turn time. If a locomotive has another start location than the start location of its first trip, we get a passive transport at the beginning of the locomotive route. We do not formulate an explicit turn time at the beginning of a locomotive route. This turn time can be included into the trip departure time window.

The cost function for the locomotive schedule gives priority to the number of used locomotives, followed by the amount of passive transport time and the locomotive waiting times. This is a common cost function also for other vehicle routing problems and it reflects the high cost of locomotives.

Observe that we have restricted the locomotive assignment problem to the case where every trips needs exactly one active locomotive. We don't consider additional passive locomotives or several active locomotives as described in Section 1.4. We needn't take into account maintenance intervals for the locomotives, as we assume that a locomotive can be replaced at any time in the schedule by another one, if maintenance is necessary. Thus, we compute the vehicle route for a virtual locomotive which can be replaced by several physical locomotives.

We do not address the issue of personnel planning and its interaction with the track allocation and locomotive assignment problems. As far as we

know, these three problems have always been handled separately and the integration of two of them is a first step.

3.3 Example

As an example for the combined track allocation and locomotive assignment problem we look at the railway network in Fig. 3.3.

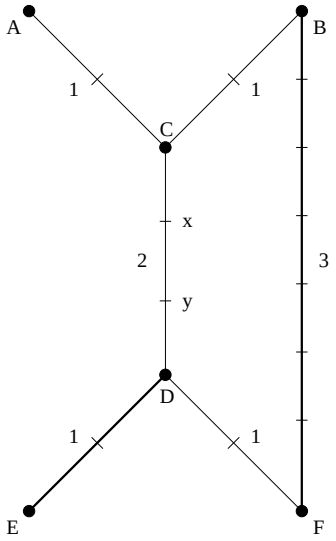

Figure 3.3: A small railway network

We have six locations A–F where trips can start and end (turn locations), these are connected by single tracks (thin lines) and double tracks (thick lines, B–F and D–E). Every line segment in the figure corresponds to a track that connects points where trains can meet and overtake. The minimum travel times between the turn locations are given in hours. We'll make the following assumptions (simplifications):

- all transports can be handled by the same locomotive type, all locomotives have the same type.
- all trains have the same speed, the travel times are given in Fig. 3.3.
- trains traveling in the same direction on a single track must respect a headway of 30 minutes.
- the turn time between two transports is 30 minutes.
- we ignore the maximum number of waiting trains at a station.

We assume that the trains always take the route with the shortest travel time. The trip specification is given in Table 3.1.

trip	start, end location	departure time window
p_1	$A - E$	8–10
p_2	$A - E$	8–10
p_3	$E - F$	12–15
p_4	$F - B$	8–10
p_5	$F - A$	12–18
p_6	$B - D$	12–15
p_7	$B - E$	8–24

Table 3.1: Trip specification with departure time windows

In Fig. 3.4, alternative schedules are shown, depending on how many locomotives are available. In the first Gantt diagram, five locomotives l_1–l_5 are available, their start positions are given in Table 3.2.

locomotive	start location
l_1	A
l_2	A
l_3	F
l_4	C
l_5	F

Table 3.2: Locomotive start locations

Locomotive l_1 begins with trip p_1 and can continue in E with trip p_3 after a turn time of half an hour (the time window for p_3 has been reached). l_2 performs the trip p_2 parallel to p_1 and must respect a headway of 30 minutes after p_1. l_3 begins with trip p_4 and must wait in B until the time window for p_6 is reached. It continues then with trip p_6 which has to stop in x (see Fig. 3.3) when trip p_5 with opposite direction on locomotive l_5 has reached y. p_6 can be continued after p_5 has passed the track between y and x. l_4 is located in C in the beginning so that a passive transport PT to B, the start location of p_7 is necessary. p_7 starts after the passive transport. Observe that p_7 doesn't get in conflict with p_3, because the path D–E consists of double tracks and can be used simultaneously by two trains with opposite directions. l_5 performs trip p_5 according to its time window.

In the second Gantt diagram we want to get rid of the locomotive l_5. p_5 can be scheduled on l_2 after a passive transport from E to F and the turn time. Observe that the passive transport can't start immediately because it

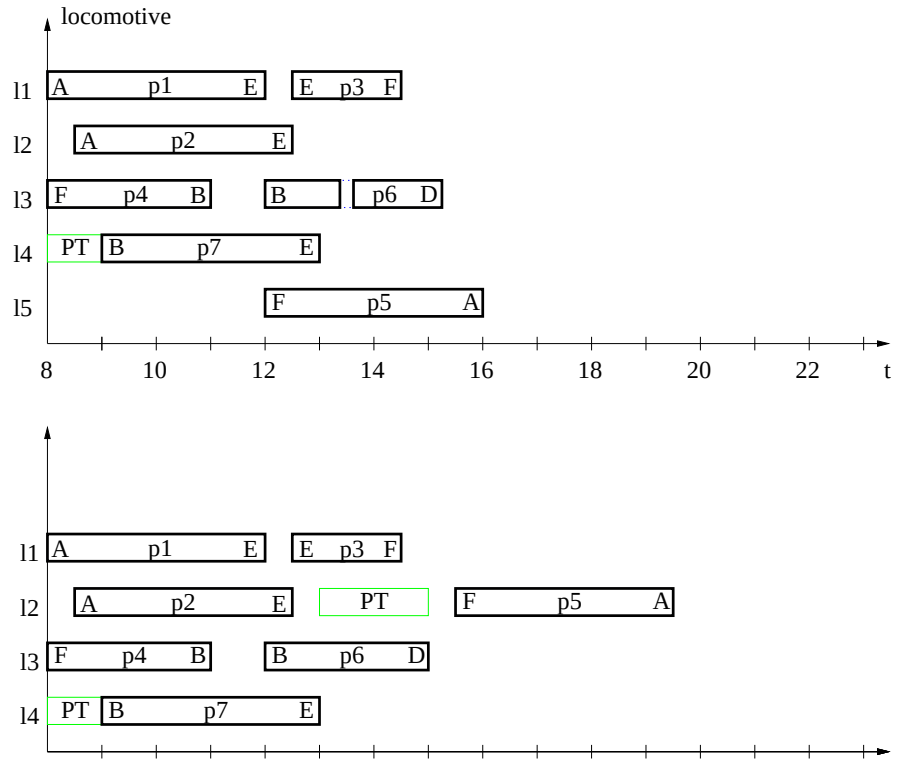

Figure 3.4: Solutions with five and four locomotives

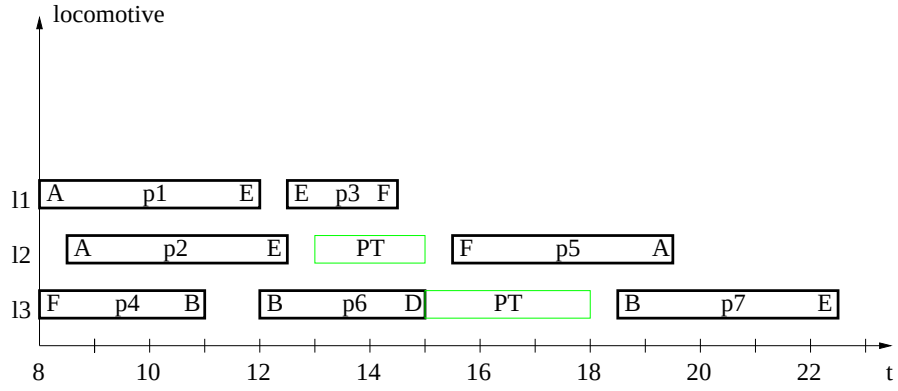

Figure 3.5: Solution with three locomotives

must respect a headway with regard to p_3 on l_1. This means that the track allocation problem has not only to be solved for the normal trips, but also for the passive transports.

In the third Gantt diagram (Fig. 3.5) three locomotives are sufficient if we eliminate l_4. We schedule p_7 on l_3 after a passive transport from D to B. A smaller locomotive number is not possible because the three trips p_1, p_2 and p_4 start in the earliest time window (8–10) and can't be done sequentially. The amount of passive transport is not optimal because the passive transport on l_2 could be replaced by the trip p_3.

Chapter 4

Related Work

We describe in this chapter several vehicle routing problems which can be found in the literature and the techniques which are used to solve them. We conclude the chapter with a comparison between these problems and our locomotive assignment problem.

4.1 Routing and Scheduling of Vehicles

Assad et al. classify in [ABBG83] vehicle problems into *scheduling* and *routing* problems. A *route* is an ordered sequence of locations visited by a vehicle[1], whereas a *schedule* also contains the associated arrival and departure times. If the times are known in advance, the problem is a *scheduling* problem. If there are no restrictions on the arrival and departure times, we have a *routing* problem. The definition of our locomotive problem contains time windows for the departure times, it is therefore a combined routing and scheduling problem. We describe in the following section a scheduling problem.

4.2 The Multiple Depot Vehicle Scheduling Problem

A well studied scheduling problem in public transport is the Multiple Depot Vehicle Scheduling Problem (MDVSP) [Loe98], which has been applied to bus and locomotive scheduling. Given a set of timetabled trips, one wants to assign a fleet of vehicles to them. The vehicles are stationed at several depots

[1]Observe that this has nothing to do with our definition of routes in the network in Def. 1.

and are supposed to return to their own depot after operation. They should be used as efficiently as possible. We will describe the problem formulation in [Loe98] in the following.

The vehicles are grouped by their type and their spatial location into depots, $\mathcal{D}$ denotes the set of depots. A *depot* $d \in \mathcal{D}$ contains all vehicles of the same type stationed at the same location. We associate a start point d^+ and an end point d^- of the vehicles with every depot $d \in \mathcal{D}$. Let $\mathcal{T}$ denote the set of timetabled trips, i.e. every trip $t \in \mathcal{T}$ has a specified departure time s_t at its first stop t^- and an arrival time e_t at its last stop t^+. The terms first and last stop come from bus scheduling where a bus has several stops on its trip. Different trips have different vehicle demands so that the vehicle types must be considered. The vehicle demand of trip t is given by its depot group $G(t) \subseteq \mathcal{D}$.

The timetabled trips are linked by passive trips. *Pull-out* trips connect a start point d^+ with a first stop t^-, *pull-in* trips a last stop t^+ with an end point d^- and *dead-head* trips a last stop p^+ with a following first stop q^-. Two timetabled trips p,$q \in \mathcal{T}$ can be linked by a dead-head trip iff $e_p + \Delta_{p,q} \leq s_q$ where $\Delta_{p,q}$ denotes the travel time from p^+ to q^-.

The objective is to minimize the vehicle number and the operational costs. This can be formulated as a multicommodity flow model. We define the sets

$$A^{t-trip} := \{(t^-, t^+) | t \in \mathcal{T}\}$$
$$A^{pull-out} := \{(d^+, t^-) | t \in \mathcal{T}, d \in \mathcal{D}\}$$
$$A^{pull-in} := \{(t^+, d^-) | t \in \mathcal{T}, d \in \mathcal{D}\}$$
$$A^{d-trip} := \{(p^+, q^-) | p, q \in \mathcal{T} \wedge G(p) \cap G(q) \neq \emptyset \wedge e_p + \Delta_{p,q} \leq s_q\}$$

for timetabled, pull-out, pull-in and dead-head trips. Every trip is represented by an arc in our network. We introduce an arc for a dead-head trip between two timetabled trips p, $q \in \mathcal{T}$ if they can be served by a common depot and if there is enough time for the vehicle to get from the last stop of p to the first stop of q. Our network has the node set

$$N = \{d^+, d^- | d \in \mathcal{D}\} \cup \{t^+, t^- | \in \mathcal{T}\}$$

and the arc set

$$A = A^{t-trip} \cup A^{pull-out} \cup A^{pull-in} \cup A^{d-trip} \cup \{(d^-, d^+) | d \in \mathcal{D}\}.$$

Fig. 4.1 shows an example for two depots d_1, d_2 and four trips a, b, c and d. We distinguish the two depots by different colours (black and hatched). Normal arcs can be used by vehicles of depot d_1, dotted arcs by vehicles

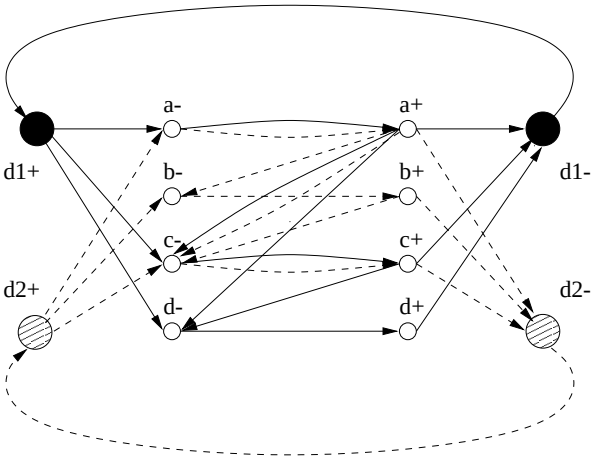

Figure 4.1: An example with two depots and four trips (adapted from [Loe98])

of depot d_2. Observe that we use multiple arcs for the purpose of better illustration, but our multicommodity flow network contains actually single arcs with depot-dependent capacities. The trips a and c can be served by both depots whereas b and d must be served by their own depot. The deadhead arcs result from a here not specified timetable and indicate the common depots of the connected trips. If we look at Fig. 4.1, we can imagine how a vehicle of depot d flows from its start point d^+ over several start/stop pairs t^-, t^+ to its end point d^- and back to the start point.

Let x_a^d denote the flow of depot d on the arc $a \in A$. We have to specify the cost c_a^d for each arc $a \in A$. We set $c_a^d = 0$ if $a \in A^{t-trip} \cup \{(d^-, d^+) | d \in \mathcal{D}\}$. The timetabled trips and the backward arcs don't produce costs. If $a \in A^{pull-in} \cup A^{d-trip}$, c_a^d represents the operational cost of the passive trip. For $a \in A^{pull-out}$, c_a^d is the operational cost of a pull-out trip plus the capital cost for a new vehicle. The capital cost is set to a large constant $M > 0$ which is bigger than the operational costs of any solution. The minimization of the flow costs leads then to a solution with the minimum vehicle number.

We can now give the multicommodity flow formulation (compare Section 2.3.2):

$$\min \sum_{d \in \mathcal{D}} \mathbf{c}^d \cdot \mathbf{x}^d \tag{4.1}$$

$$\sum_{\{j|(i,j) \in A\}} x_{(i,j)}^d - \sum_{\{j|(j,i) \in A\}} x_{(j,i)}^d = 0 \quad \forall i \in N \quad \forall d \in \mathcal{D} \tag{4.2}$$

$$\sum_{d \in G(t)} x_a^d = 1 \quad \forall t \in \mathcal{T} \quad \forall a \in A^{t-trip} \tag{4.3}$$

$$0 \leq x_a^d \leq 1 \quad \forall d \in \mathcal{D} \quad \forall a \in A^{t-trip} \cup A^{pull-out} \cup A^{pull-in} \cup A^{d-trip} \tag{4.4}$$

$$\lambda_d \leq x_a^d \leq \kappa_d \quad \forall d \in \mathcal{D} \quad \forall a \in \{(d^-, d^+) | d \in \mathcal{D}\} \tag{4.5}$$

Equation 4.1 defines the cost function we want to minimize, i.e. the total amount of flow. Eqn. 4.2 describes the flow conservation condition for each depot. Eqn. 4.3 states that every timetabled trip is served by exactly one vehicle. The last two equations define the arc capacities. We set all arc capacities to one except the capacities for the backward arcs (d^-, d^+). These serve to model the depot capacities. λ_d and κ_d define lower and upper bounds for the number of used vehicles of depot d.

Given this network flow model, Löbel applies in [Loe98] Lagrangean relaxation techniques to solve the MDVSP for large problem instances. Problem sizes up to 49 depots and 25000 timetabled trips can be handled. It is known that the MDVSP is NP-hard for several depots [Loe98], whereas the single depot case can be solved by a simplex method in polynomial time. These solution methods are used for the MDVSP in public transport [Loe98].

4.3 The Vehicle Routing and Scheduling Problem with Time Windows

We will introduce another vehicle routing problem that is encountered in the distribution of goods. Given a set of customers requiring service, a fleet of vehicles, stationed at a central depot, shall be routed so that each customer is visited once. The vehicles have a maximum capacity for the delivered goods which must not be exceeded and are supposed to return to the depot after service. The objective is to determine a set of minimum-cost vehicle routes, i.e. the minimum number of routes with the minimum travel distance. This is the so-called vehicle routing problem (VRP) [ABBG83].

In the vehicle routing and scheduling problem with time windows (VRSPTW), the customers can additionally specify a time window for the delivery time, i.e. an earliest and latest delivery time [Sol87]. Due to the time windows, a vehicle can arrive to early at a customer location and has to wait until the earliest delivery time has been reached. These waiting times are an additional cost factor.

Let $C = \{1, \ldots, n\}$ denote the set of costumers. The service at costumer $i \in C$ takes s_i units of time and begins at time b_i. The time window for

service is given by an earliest time e_i and a latest time l_i, i.e. $e_i \leq b_i \leq l_i$. The travel time between two costumers i and j is denoted by t_{ij}, their distance by d_{ij}. The distance matrix $D = (d_{ij})$ is symmetric and satisfies the triangle equality and the travel times t_{ij} are proportional to the distances. If a vehicle coming from customer i arrives too early in j, it has to wait until the time window for service is reached: $b_j = \max\{e_j, b_i + s_i + t_{ij}\}$.

Solomon suggests in [Sol87] the following cost factors, which determine a lexicographical ordering between the solutions (i.e. the factors are hierarchically ordered, beginning with the most important one):

1. number of vehicles
2. total schedule time
3. total travel distance
4. total waiting time

We will describe an insertion-type heuristic for tour building which performed best compared to other heuristics on a heterogenous problem set [Sol87]. The heuristic builds sequentially one vehicle route after the other. A route is initialized with a seed costumer, e.g. the farthest unrouted customer or the unrouted customer with the earliest deadline for service. At every iteration, a new unrouted customer u is inserted between two customers i and j in the current partial route. A new route is started if no more customers with feasible insertions (i.e. without violation of time windows) can be found. Let $(i_0, \ldots, i_m)$ denote the current route where $i_0 = i_m = 0$ represent the depot. For every unrouted customer u, its best feasible insertion place is computed: we determine its predecessor $i(u)$ and successor $j(u)$ with

$$c_1(i(u), u, j(u)) = \min_{1 \leq p \leq m} c_1(i_{p-1}, u, i_p)$$

$c_1(i, u, j)$ defines the cost of an insertion of costumer u between two costumers i and j and will be defined later. Next, the best costumer u^* to be inserted in the route is selected by

$$c_2(i(u^*), u^*, j(u^*)) = \max_{u \in U} c_2(i(u), u, j(u))$$

where U denotes the set of unrouted and feasible customers. Costumer u^* is then inserted between $i(u^*)$ and $j(u^*)$.

The cost function $c_1(i, u, j)$ consists of two components c_{11}, c_{12}:

$$c_1(i, u, j) = \alpha_1 c_{11}(i, u, j) + \alpha_2 c_{12}(i, u, j), \quad \alpha_1, \alpha_2 \geq 0, \quad \alpha_1 + \alpha_2 = 1$$

$c_{11}(i,u,j)$ is the additional distance the vehicle must travel when u is inserted between i and j:

$$c_{11}(i,u,j) = d_{iu} + d_{uj} - \mu d_{ij}, \quad \mu \geq 0$$

$c_{12}(i,u,j)$ is the extra time which is required to visit customer u on the current route:

$$c_{12}(i,u,j) = b_{j_u} - b_j$$

where b_{j_u} is the new time for service at customer j when u is inserted. $c_2(i,u,j)$ is the benefit of serving customer u between i and j compared to service on a direct route:

$$c_2(i,u,j) = \lambda d_{0u} - c_1(i,u,j), \quad \lambda \geq 0$$

Solomon develops in [Sol87] a set of benchmark problems with different characterics (e.g. the distribution of customer locations, time windows) and tests several heuristics on this problem set. The insertion heuristic behaved very stable across different problem variants and often obtained the optimal or near-optimal solutions.

Since the VRP is NP-hard [LK81], the VRSPTW is NP-hard by reduction. A worst-case analysis in [Sol86] shows that the worst-case performance ratio r_n of the insertion heuristic [2] in terms of the travel distance and the number of vehicles grows linearly with the number of costumers, i.e. $r_n \in \Omega(n)$. This shows that the VRSPTW is a notoriously difficult problem.

Potvin et al. suggest in [PR93] a parallel route building algorithm instead of a sequential route building algorithm which builds one route at a time. Experiments have shown that the last routes often have poor quality in the sequential approach. In the parallel approach, the initial number of vehicles is determined by the sequential route building algorithm. The generated routes are also used for selecting appropriate seed customers. The insertion cost $c_1(i,u,j)$ of costumer u is computed for all insertion positions in all routes. $c_2(i,u,j)$ is different from the sequential route building algorithm as it is a regret measure for costumer u. A costumer receives a large regret measure if there is a large gap between its minimum insertion cost and the next best insertions. Costumers with large regrets are inserted first since their number of good insertion positions is small.

Computational experiments in [PR93] show that the parallel approach is better than the sequential approach for some problem variants where the costumer locations are randomly distributed and not clustered.

[2] the quotient of the cost of the heuristic solution and the cost of the optimal solution

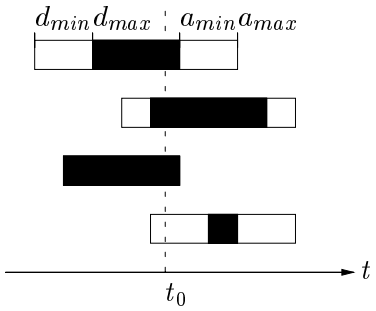

Figure 4.2: Four trips and their kernel times (black)

4.4 Discussion

We will now compare the two described vehicle routing problems with our locomotive assignment problem.

The network flow model in Section 4.2 can't handle time windows for the trips, the departure times must already have been determined. This information is needed in order to determine which timetabled trips can be connected by a passive trip. A previously fixed timetable leads to suboptimal solutions, because flexible departure times could be used to generate better vehicle schedules.

There have been approaches which try to move the departure times of a fixed timetable in order to obtain better vehicle schedules. Bokinge et al. [BH80] shift the departure time of every trip so that the maximum number of required vehicles over all time instances is minimized. This decision is possible because the trips have only a small flexibility and a kernel time interval, defined by the latest departure and earliest arrival time, is known for every trip. Fig. 4.2 shows four trips with their earliest departure time d_{min}, their latest arrival time a_{max} and the black kernel time interval $[d_{max}, a_{min}]$. At time instance t_0, at least three vehicles are required.

Every trip is shifted in a way which minimizes the vehicle demand at every time instance, i.e. the number of kernel time intervals which include a certain time point. The network flow problem is then solved for the modified departure times and some postprocessing has to be done because the vehicle schedules become scattered through the trip shifting.

This approach was developed for the case where a given timetable shall be changed by a small amount, e.g. in bus scheduling where it is more convenient for the passengers to have a timetable with periodic departure times. These departure times may only be changed by small amounts.

The approach works only if the trip kernels are known. The trip kernel

can't be determined for short trips with a large flexibility. We won't consider this approach further because in our problem, we needn't modify an old timetable. We can build a new timetable which allows good vehicle routes. This way seems to be more direct than changing a given timetable by small amounts. Additionally, network flow algorithms would be difficult to integrate into the constraint-based TUFF environment.

But the kernel time interval idea could be used to get a lower bound for the required number of vehicles: we determine the maximum of required vehicles over all time points by considering the kernel intervals. This is a lower bound for the number of vehicles for our schedule. It is only a lower bound because we do not consider the start locations of the vehicles in this estimation.

The VRSPTW from Section 4.3 includes time windows in the problem specification and can be mapped to our locomotive assignment problem. If we identify customers with trips and the journey between two costumers with a passive transport connecting two trips, we can see the correspondence between the two problems. The time window for the begin of service of a costumer corresponds to the time window for the departure time of a trip.

Despite the similarities, there are also differences between the two problems. The locomotives have different start locations and needn't return to them after operation whereas the vehicles in the VRSPTW all start and end at the same depot. There are no vehicle capacities either, so that a locomotive can perform an arbitrary amount of trips [3]. But the routing component is similar in both problems.

We presume that our locomotive assignment problem is NP-hard because the VRSPTW is also NP-hard. Thus, we have to use heuristics in order to solve the problem. The heuristics for the VRSPTW should also be helpful for our locomotive assignment problem and we will suggest several just heuristics in Chapter 8.

[3] we needn't bother about maintenance intervals, we discussed this in Section 3.2.

Chapter 5

Constraint Programming

5.1 Introduction

This chapter provides a short introduction to constraint programming and defines its most important concepts. We will begin with an informal definition of the problems encountered in constraint programming and explain it by means of an example [Tsa93]:

Definition 5 *A Constraint Satisfaction Problem is a problem composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can simultaneously take. The task is to assign a value to each variable satisfying all the constraints.*

A well-known example of this kind of problems are puzzles like the following [Sch98]:

Definition 6 (MONEY) *Find distinct digits for the letters in the equation*

$$SEND + MORE = MONEY$$

such that $S, M \neq 0$ *and the equation is satisfied. What is a solution of this problem?*

This problem can be formulated by the following numerical constraints:

$$S, M \neq 0 \tag{5.1}$$

$$D, E, M, N, O, R, S \text{ pairwise distinct} \tag{5.2}$$

$$\begin{aligned} & 10^3 \cdot S + 10^2 \cdot E + 10 \cdot N + D && (5.3)\\ + \quad & 10^3 \cdot M + 10^2 \cdot O + 10 \cdot R + E && (5.4)\\ = \quad & 10^4 \cdot M + 10^3 \cdot O + 10^2 \cdot N + 10 \cdot E + Y && (5.5) \end{aligned}$$

We give now a formal definition of constraint satisfaction problems [Tsa93]:

Definition 7 (CSP) *A* Constraint Satisfaction Problem *is given by a triple* (X, D, C) *with*

- $X = \{x_1, x_2, \ldots, x_n\}$, *a finite set of* variables.
- $D = \{D_1, D_2, \ldots, D_n\}$, *a finite set of* domains. D_i *is the domain of the variable* x_i $(1 \leq i \leq n)$.
- $C = \{C_1, \ldots, C_m\}$ *is a finite set of* constraints. *The constraint* $C_i(i_1, \ldots, i_k)$ *over the variables* $x_{i_1}, \ldots, x_{i_k}$ *is the set of valid* labels $C_i(i_1, \ldots, i_k) \subseteq D_{i_1} \times \cdots \times D_{i_k}$.

A label is the assignment of all variables in a CSP to a value. We will omit the index of a constraint if it is not necessary. In the following we will only deal with *finite domain* constraint problems where the domains are finite sets of nonnegative integers:

Definition 8 (Finite Domain CSP) *A* finite domain *CSP is a CSP with* $D_i \subseteq \{0, \ldots, FD_{sup}\}$ $\forall i \in \{1, \ldots, n\}$ *where* $FD_{sup} \in \mathbf{N}$ *is the largest finite domain integer value.*

We call the variables occuring in a finite domain CSP *finite domain variables* (FD-variables):

Definition 9 (FD-variable) *The domain* D *of a* finite domain variable x *is a subset of the nonnegative integers:* $D \subseteq \{0, \ldots, FD_{sup}\}$. *We use the interval notation* $D = [a, b]$ *for a domain of the form* $D = \{a, a+1, \ldots, b\}$.

We restrict our CSPs to binary CSPs i.e. they contain only constraints $C(i, j)$ $(1 \leq i < j \leq n)$ where two variables are involved. This is no severe restriction as a general CSP can always be transformed into a binary CSP [Tsa93]. We can define a constraint graph for binary CSPs which represents the dependencies between the variables:

Definition 10 (Constraint Graph) *The* Constraint Graph *of a binary CSP* (X, D, C) *is a directed graph* $G = (X, E)$ *with*

$$E = \{(x_i, x_j) | \exists c \in C : (c = C(i,j) \vee c = C(j,i)) \wedge x_i, x_j \in X\}$$

Every constraint $C(i,j)$ *is represented by two arcs* (x_i, x_j), $(x_j, x_i) \in E$.

Although an undirected graph would be sufficient to illustrate the variable dependencies, the formulation as a directed graph is more suited for an algorithm that we will describe in Section 5.4.

5.2 Solving of Constraint Problems

A naive and inefficient strategy to solve a CSP is the *generate-and-test* paradigm [Bru95]. As the search space is finite in a finite domain CSP, we can generate all labellings of the variables and test which of them satisfy the imposed constraints. Since the search space has the size of the cartesian product of the domains, this strategy works only for small problems.

The *backtracking* paradigm is an improvement of this strategy. Instead of checking the constraints after all variables have been instantiated, we assign values to the variables sequentially and perform a consistency check after every assignment. If a constraint is violated, we backtrack to the most recently instantiated variable and assign an alternative value to it. We formulate this algorithm in pseudocode [Bru95]:

procedure Backtrack(k, $[D_k, D_{k+1}, \ldots, D_n]$)
begin
  repeat
    select $w_k \in D_k$;
    $D_k := D_k - \{w_k\}$;
    $x_k := w_k$;
    boolean *consistent* $:= true$;
    for $i := 1$ **to** $k - 1$ **do**
      if $(w_i, w_k) \notin C(i,k)$ **then**
        consistent $:= false$;
      endif
    endfor
    if *consistent* **then**
      if $k < n$
        then Backtrack($k + 1$, $[D_{k+1}, \ldots, D_n]$);
        else print solution; **stop**;
      endif

```
    endif
  until D_k = ∅
end
```

The procedure Backtrack gets two parameters, the first defines the search depth, the second is the list of variables that have not been instantiated yet. Backtracking does an *a posteriori* pruning of the search space as it backtracks when a failure is detected. The search space is still quite large so that this strategy can only be applied to small problems.

Constraint programming introduces the notion of *constraint propagation* which means that the assignment or restriction of a variable is propagated through the constraint graph and reduces the domains of dependent variables. Constraint propagation during search reduces the search space *a priori* before a failure is detected. We will look at the *forward-checking* strategy as an example for the combination of backtracking and propagation [Bru95]. After a variable has been assigned, one removes all incompatible values from the domains of the not instantiated variables. We give the algorithm for forward-checking also in pseudocode [Bru95]:

```
procedure ForwardChecking(k, [D_k, D_{k+1}, ..., D_n]) % selection
begin
  repeat
    select w_k ∈ D_k;
    D_k := D_k − {w_k};
    x_k := w_k; % assignment
    boolean empty := false;
    for i := k + 1 to n do
      D'_i := ∅;
      forall w_i ∈ D_i do
        if (w_k, w_i) ∈ C(k, i) then
          D'_i := D'_i ∪ {w_i}; % propagation
        endif
      endfor
      if D'_i = ∅ then empty:=true endif;
    endfor
    if not empty then
      if k < n then ForwardChecking(k + 1, [D'_{k+1}, ..., D'_n])
          else print solution; stop;
      endif
    endif
  until D_k = ∅
end
```

The invariant can be formulated as follows: for every not instantiated variable there exists at least one value in its domain which is consistent with the labels of the variables that have already been assigned. Observe that we needn't check consistency with the already assigned variables as this is guaranteed by the invariant.

The algorithm runs through a *selection-assigment-propagation* cycle [Bru95]:

1. Selection: choose the next variable x_k which shall be instantiated (the variable ordering is implicitely given by the variable indices in the pseudocode above).
2. Assignment: assign the values in D_k successively to x_k (the value order is defined by the search strategy, see Section 5.3).
3. Propagation: restrict the domains of the not instantiated variables by propagation.

We can illustrate the search process by a search tree where the nodes represent not instantiated variables and the branches the different assignment alternatives. In every node of the search tree one has to make two choices:

- the next variable that shall be instantiated.
- the next value the variable shall be assigned to.

We will describe some heuristics for this choice in the next section.

5.3 Search Strategies

Variable ordering aims at moving failures to the upper levels of the search tree [Kum92]. A good variable ordering reduces the bushiness of the search tree. Examples for common variable ordering heuristics are:

- *first-fail*-principle: choose the variable with the smallest domain.
- choose the variable that participates in the highest number of constraints.

One hopes that these variables act as a bottleneck to the problem and their instantiation leads to an early pruning of the search space.

Value ordering tries to move solutions to the left of the tree so that they can be found quickly by depth-first-search. Examples for value orderings are:

- the minimum, maximum or middle value of a domain.
- splitting of domains: branching into the two alternatives $x \leq m$, $x > m$ where m denotes the middle value of the domain of x.

5.4 Constraint Propagation

Constraint propagation tries to identify redundant values in variable domains. We will start with a definition of pairwise consistency between the variables in a CSP and develop an algorithm that removes values from variable domains in order to achieve pairwise consistency. Recall from Def. 10 that the constraint graph $G = (X, E)$ contains two directed arcs between two dependent variables.

Definition 11 (Arc Consistency) *Arc $(x_i, x_j) \in E$ is arc-consistent iff $\forall a \in D_i \quad \exists b \in D_j : (a, b) \in C(i, j)$. A CSP is arc-consistent if all arcs are arc-consistent.*

We explain this definition by a graph-colouring example [Kum92]. In this problem, the nodes of a graph have to be coloured so that adjacent nodes have different colors. We assign to every node a finite domain variable which can take different color values. Fig. 5.1 shows an example graph and the initial variable domains. The arcs (V_3, V_2), (V_3, V_1), (V_1, V_2) and (V_2, V_1)

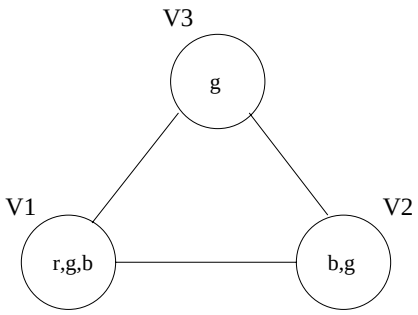

Figure 5.1: Graph-Coloring Problem with initial variable domains (r, g, b denote the colors red, blue, green).

are already arc-consistent. The arcs (V_2, V_3) and (V_1, V_3) are not consistent. If we want to make (V_2, V_3) arc-consistent, we have to remove the value g from the domain of V_2 (see Fig. 5.2). As the domain of V_2 has become smaller, we can't be sure that consistent arcs of the form (V_i, V_2) remain consistent. Some of the members of V_i might no longer be compatible with

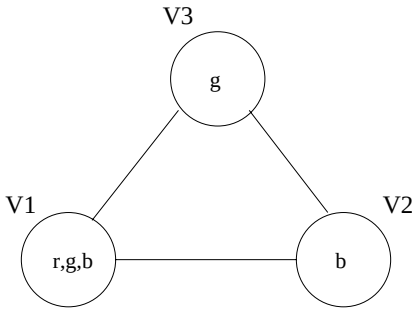

Figure 5.2: Domains after restriction of V_2

any remaining member in the domain of V_2. In this example (V_3, V_2) remains arc-consistent but (V_1, V_2) becomes inconsistent. We have to examine this arc again. By achieving arc-consistency for (V_1, V_2) and for (V_1, V_3) we arrive at the solution $V_1 = \{r\}$, $V_2 = \{b\}$ and $V_3 = \{g\}$.

As we have seen in the example, we need a procedure which achieves arc-consistency for the arc (x_i, x_j) by removing redundant values from the domain of x_i:

procedure boolean ReviseDomain(x_i,x_j,(X, D, C))
begin
  boolean $Delete := false$;
  forall $a \in D_i$
    if $\nexists b \in D_j : (a, b) \in C(i, j)$
      $D_i := D_i - \{a\}$;
      $Delete := true$;
    endif
  endfor
  return $Delete$;
end

The procedure ReviseDomain gets as input the arc (x_i, x_j) and the CSP (X, D, C). It returns *true* if a value from D_i has been removed. If d denotes the size of the largest variable domain, the time complexity of this procedure is $O(d^2)$, because the complexity of the **for**-loop and the **if**-statement are in $O(d)$.

To achieve arc-consistency for all arcs in the constraint graph, we have to consider all arcs and must examine all arcs again which are incident to an updated variable:

AC3-algorithm
input: CSP (X, D, C)
output: arc-consistent domains

begin
  $L := \{(x_i, x_j) | C_{i,j} \in C \vee C_{j,i} \in C\}$
  while $L \neq \emptyset$
    $L := L - \{(x_i, x_j)\}$;
    if ReviseDomain(x_i,x_j,(X, D, C))
    then $L := L \cup \{(x_k, x_i) | C_{k,i} \in C \wedge i \neq k \neq j\}$;
    endif
  endwhile
end.

We examine all arcs in the constraint graph (**while**-loop). If the variable x_i has been updated, we must add incident arcs of the form (x_k, x_i). The condition $k \neq i$ in the **then**-part of the **if**-statement is obvious, because our constraint graph contains no loops. We can also say $k \neq j$ because it is not necessary to add the arc (x_j, x_i) after the application of $ReviseDomain(x_i, x_j, (X, D, C))$. None of the elements deleted from the domain D_i provided support for any value in the domain D_j.

Let d denote the size of the biggest domain and e the number of arcs in the constraint graph. As every arc has to be examined at least once and the time complexity of $ReviseDomain$ is $O(d^2)$, a lower bound for the run time is $\Omega(ed^2)$.

An arc (x_k, x_i) is inserted into L when at least one value has been removed from the domain D_i. This can happen at most d times for every arc and a maximum of $O(ed)$ arcs can be added to L. Thus, the worst-case run time is $O(ed^3)$.

5.5 Constraint Programming Languages

Constraint programming languages were developed in order to free the user from the implementation of propagation and search algorithms [Bru95, Tsa93]. They provide built-in functions for commonly encountered constraints and search strategies. The paradigm of knowledge-based programming in artificial intelligence says that the problem knowledge and the solution techniques should be separated. This separation allows the modification and extension of existing programmes in an easy way. Constraint programming languages are well-suited for this approach, because they provide a natural separation between the constraints and the search strategy.

Most constraint programming languages are based on the logic programming paradigm. This is quite natural since logic programs operate on relations between objects. Languages like Prolog have built-in search strategies (depth-first-search). The variables are not typed and have terms as values. Prolog programmes for combinatorial problems use the generate-and-test strategy which provides no pruning of the search space.

Constraint logic programming introduces types for the variables (domains) and propagation techniques for constraints. The hope is that the pruning of the search space by constraint propagation leads to more efficient programs for combinatorial search problems.

One of the most popular constraint programming languages is CHIP (Constraint Handling in Prolog) [Bru95]. It can handle finite, boolean and rational domain variables. The basic search strategy is forward-checking combined with the first-fail-principle for variable selection. Applications include scheduling, geometrical layout and vehicle routing problems [Sim96]. CHIP introduced the notion of global constraints i.e. global conditions that are difficult to express by elementary binary constraints. It provides an efficient implementation of high–level global constraints with a strong propagation behaviour. We will describe an example for these constraints in Section 5.7.

In this work the constraint programming language Oz 2 was used. It allows higher-order functional and object-oriented programming. We will focus in the next section on the constraint part of Oz 2.

5.6 Constraint Programming in Oz

We follow in this section the exposition in [Wür97] and [Sch98]. The constraints in Oz operate on FD-variables, the domains are finite sets of non-negative integers. Constraints are terms of first order predicate logic with arithmetic equality.

The domains are stored in the *constraint store*. It contains *basic constraints* for each variable of the form $x = n$, $x = y$ or $x \in D$ where x and y denote finite domain variables, $n \in \mathbf{N}_0$ and D denotes a finite domain. The more expressive constraints which contain the problem formulation are stored in *propagators*, one for each non-basic constraint.

The propagators communicate via shared variables in the constraint store which serves as a blackboard (see Fig. 5.3). A propagator for a constraint C is a computational agent that tries to narrow the domains of the variables occuring in C. It performs constraint propagation.

A propagator is triggered whenever one of its variables is narrowed by another propagator or the search procedure. Propagation takes place until a

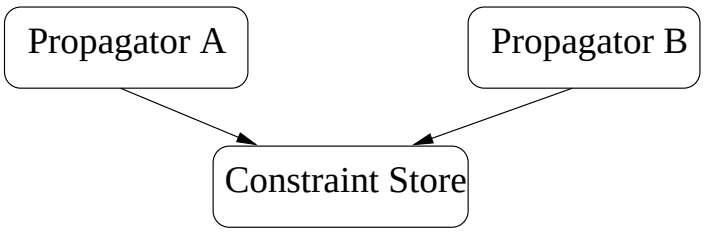

Figure 5.3: Propagators communicating via the constraint store

fixed point reached, i.e. none of the propagators can narrow its variables anymore.

The search procedure in Oz is called *distribution*. Given a CSP (X, D, C), we choose in every node of the search tree a constraint C' and branch into the two alternatives $C \wedge C'$ and $C \wedge \neg C'$. This is called the distribution of C with the constraint C' at a *choice point*. C' can be a basic constraint, e.g. variable assignment with the alternatives $x = n$ and $x \neq n$ or a non-basic constraint. Due to the equivalences

$$C = C \vee T = C \vee (\neg C' \wedge C') = (C \wedge C') \vee (C \wedge \neg C')$$

this search procedure is complete and generates all solution tuples of the CSP. After the branching into one of the two alternatives the propagation starts again until a fixed point is reached.

Oz contains predefined distribution strategies like first-fail but allows also user-defined distribution procedures. In Chapter 9, such a user-defined distribution will be described.

Oz can also be extended by additional propagators by implementing them in C++. We will describe in Chapter 9 the extension of Oz by a new propagator.

5.7 A High–Level Geometric Constraint

We define in this section a geometric constraint which describes a non-overlapping condition between rectangles in the plane. It is one example for a global constraint and also existent in the CHIP system. This so-called `diff2`-constraint will be used later in the constraint model for the locomotive problem.

The following definition is adapted from [BC94] from the higherdimensional to the two-dimensional case, the constraint for n dimensions is called `diffn`.

Given a set of n iso-oriented rectangles in the plane, the `diff2`-constraint ensures that the rectangles do not overlap:

Definition 12 (diff2) *The constraint* $diff2((x_1, \ldots, x_n), (y_1, \ldots, y_n), (w_1, \ldots, w_n), (h_1, \ldots, h_n))$ *with* $n > 1$ *holds iff the following conditions hold:*

- $\forall i \in \{1, \ldots, n\}$*:* x_i *and* y_i *are FD-variables for the bottom-left corners of the rectangles and* w_i*,* h_i *are FD-variables for the rectangle widths and heights.*
- $\forall i \in \{1, \ldots, n\}$*:* w_i*,* $h_i \neq 0$*.*
- $\forall i, j \in \{1 \ldots n\}$ *with* $i < j$*:*

$$\begin{aligned} & (x_i \geq x_j + w_j) \\ \vee \quad & (x_j \geq x_i + w_i) \\ \vee \quad & (y_i \geq y_j + h_j) \\ \vee \quad & (y_j \geq y_i + h_i) \end{aligned}$$

We illustrate the non-overlapping condition with an example where we assume fixed rectangle sizes (i.e. the domains of the w_i and h_i are singleton domains). Fig. 5.4 shows that two rectangles in the plane overlap iff both the projections on the x- and the y-axis overlap. They do not overlap if at least one of the conditions $(x_i \geq x_j + w_j) \vee (x_j \geq x_i + w_i)$ or $(y_i \geq y_j + h_j) \vee (y_j \geq y_i + h_i)$ holds. The first condition says that the x-projections of rectangle R_i and rectangle R_j do not overlap, the second states the same for the y-projections. We demand non-zero widths and heights of the rectangles in Def. 12 in order to avoid special cases like intervals and points.

The aim of a propagation algorithm for this constraint is to narrow the domains of the participating variables as far as possible. It should also detect the case where it is not possible to place the rectangles without any overlap and produce a failure. Such an algorithm will be presented in Chapter 7.

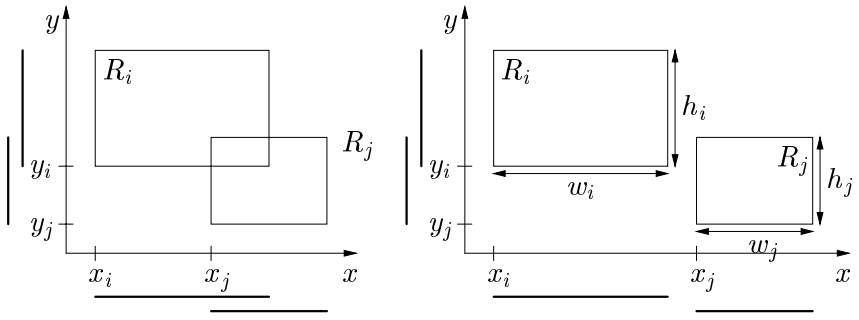

Figure 5.4: Rectangles R_i and R_j and their x- and y-projections

Chapter 6

Constraint Model for Locomotive Assignment

In this chapter, a constraint model from [Sim95b] is presented. It guarantees the location continuity constraint for the locomotives which we have introduced in Def. 4 in Section 3.2, i.e. the locomotives have always enough time to perform necessary passive transports between the trips. The locomotive trips are modeled as rectangles in a Gantt diagram. We use the geometric `diff2`-constraint to formulate non-overlap conditions.

6.1 The Exclusion Marker Model

Recall from Def. 4 that we are looking for an assignment $\phi : P \rightarrow M$ of trips to locomotives and suitable start times $s(p_i)$ for all trips $p_i \in P$ in the locomotive assignment problem. To simplify matters we will neglect the locomotive types in the beginning and assume that every trip can be handled by any locomotive. Different locomotive types will be introduced later.

We can illustrate the assignment in a Gantt diagram and model every trip p_i as a rectangle with width $d(p_i)$ for its duration and height one (Fig. 6.1).

We will use the FD-variables $s(p_i)$ for the start times and $r(p_i)$ for the resource (locomotive) of a trip $p_i \in P$ in our constraint formulation of the problem. We can set the initial domains to $s(p_i) = [0, FD_{sup}]$ and $r(p_i) = [0, m-1]$ where m denotes the numbers of locomotives. The set of locomotives is $M = \{0, \ldots, m-1\}$.

A locomotive can only handle one trip at a time i.e. the trips assigned to a locomotive may not overlap in time. We can express this by a geometric

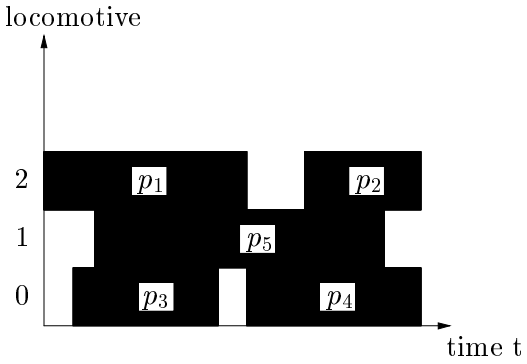

Figure 6.1: Gantt diagram

`diff2`-constraint (see Def. 12) which states that the trip rectangles may not overlap. Given the set of trips $P = \{p_1, \ldots, p_n\}$, we can formulate the following `diff2`-constraint:

$$\texttt{diff2}((s(p_1), \ldots, s(p_n)), (r(p_1), \ldots, r(p_n)), (d(p_1), \ldots, d(p_n)), (1, \ldots, 1)) \tag{6.1}$$

This is only a necessary condition because we also have to consider passive transports which introduce gaps between the trips on a locomotive. Recall that $\delta_2(v_1, v_2)$ denotes the travel time for a passive transport between the turn locations v_1, $v_2 \in V_2$ in our network. In order to simplify the model, we include the turn time Θ from Def. 4 into the trip durations $d(p_i)$.

Fig. 6.2 shows the markers that are introduced for every trip in order to allow passive transports. The figure contains four coordinate systems. The first coordinate system contains all trip rectangles (black) which must fulfill the `diff2`-constraint of Equation 6.1. We add an additional `diff2`-constraint for every turn location $v \in V_2$ that occurs in our planning problem. T denotes the set of all turn locations in our planning problem, i.e. all start and end locations in our trip set P.

The three following coordinate systems in Fig. 6.2 correspond to the `diff2`-constraints for the start location $S(p_i)$, the end location $E(p_i)$ of trip p_i and any other location $L \in T$ with $S(p_i) \neq L \neq E(p_i)$. The last coordinate system is representative for all locations L which are neither start nor end location of trip p_i.

Every coordinate system is divided into rows (horizontal dotted lines in Fig. 6.2) of height k, a parameter which will be explained later. Every row corresponds to a locomotive and we place markers (rectangles) into the row corresponding to the assigned locomotive $r(p_i)$. If the trip rectangle is placed at height $r(p_i)$ in the first coordinate system, we place our markers

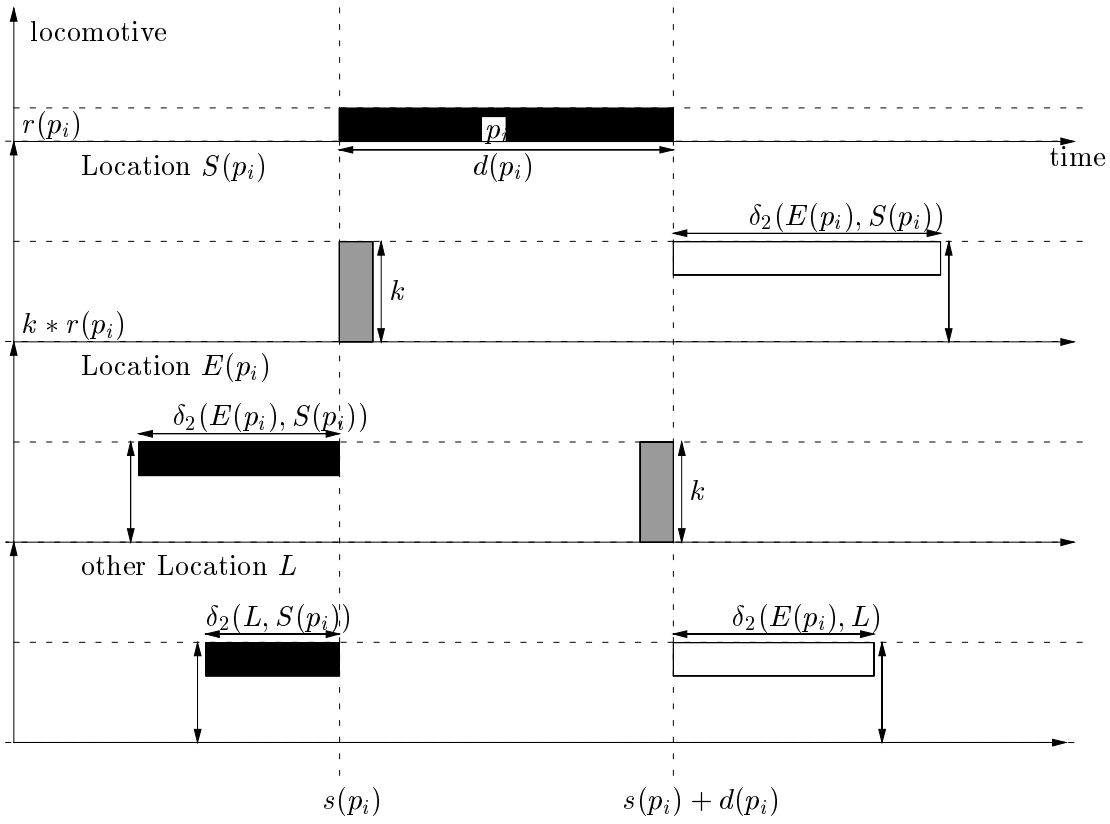

Figure 6.2: Start, end and exclusion markers for a trip

into the row with the lower bound $kr(p_i)$ and the upper bound $kr(p_i)+k-1$ in the following coordinate systems.

We will describe in the following all rectangle positions by the notation (x, y) for the position of its bottom-left corner where x and y are FD-variables. We place a start marker (grey) in the `diff2`-constraint of the start location $S(p_i)$ with position $(s(p_i), k \cdot r(p_i))$, width one and height k. An end marker (grey) is placed in the `diff2`-constraint of the end location $E(p_i)$ with position $(s(p_i) + d(p_i) - 1, k \cdot r(p_i))$, width one and height k.

To get the necessary gaps between the trips we introduce exclusion markers before and after the trip period. We call them before and after markers. An after marker (white) is placed in every location $L \neq E(p_i)$ at position $(s(p_i) + d(p_i), [k \cdot r(p_i), k \cdot r(p_i) + k - 1])$, width $\delta_2(E(p_i), L)$ and height one. These markers ensure that the locomotive has enough time to get from the end location $E(p_i)$ of trip p_i to a new location L for the next trip. The interval notation means that these markers can move vertically,

this is necessary in order to avoid overlaps between exclusion markers. Recall that the variables for the rectangle positions are FD-variables, i.e. their domains are subsets of the natural numbers.

We place symmetrically to the after markers before markers (black) in every location $L \neq S(p_i)$. They have the position $(s(p_i) - \delta_2(L, S(p_i)), [k \cdot r(p_i), k \cdot r(p_i) + k - 1])$, width $\delta_2(L, S(p_i))$ and height one. These markers ensure that the locomotive has enough time to get from a distant location to the start location $S(p_i)$ of trip p_i.

Exclusion markers prohibit start and end markers in certain time intervals before and after the trip period and guarantee location continuity, because exclusion markers may not overlap with start or end markers. The variable vertical position of the exclusion markers allows them to move so that overlaps between exclusion markers of different trips on the same locomotive can be avoided. We have introduced so far two markers per turn location for each trip, the total amount of markers is in $O(n \cdot |T|)$.

The exclusion markers we have introduced so far allow passive transports between two trips that are assigned to the same locomotive. We must also take into account passive transports from the start location of a locomotive to the start location of its first trip.

We extend the model from [Sim95b] by adding additional exclusion markers to the location `diff2`-constraints. Let $l \in M$ be a locomotive with start location $\Sigma(l)$. We put an exclusion marker in every location $L \neq \Sigma(l)$ in the row corresponding to locomotive l. This marker has the position $(0, [k \cdot l, k \cdot l + k - 1])$, width $\delta_2(\Sigma(l), L)$ and height one. This has to be done for all locomotives $l \in M$. We call these markers locomotive markers. We introduce $O(m \cdot |T|)$ locomotive markers and get $O((m+n)|T|)$ markers in total. In the `diff2`-constraint for the trip rectangles, we have $O(n)$ rectangles and in the location constraints $O(m + n)$ rectangles.

k is a parameter that must be chosen large enough so that overlaps between exclusion markers can always be avoided. This parameter is not further explained in [Sim95b]. We will show how to compute a minimum value for k from the distance function δ_2 and the trip durations $d(p_i)$. We assume that we know the minimum trip duration $d_{min} = \min_{p \in P} d_{min}(p)$ where $d_{min}(p)$ is the minimum trip time for trip p which can be computed from the minimum waiting times for a trip (compare Def. 4). We also know the maximum time for a passive transport: $\delta_{2,max} = \max_{v_1, v_2 \in V_2} \delta_2(v_1, v_2)$.

We look at the row that corresponds to one locomotive in a location `diff2`-constraint (Fig. 6.3). We will compute the maximum number of exclusion markers at the departure time $s(p_i)$ of a trip in a `diff2`-constraint for a location $L \neq S(p_i)$. Observe that the number of exclusion markers doesn't change between the discrete departure and arrival times, it is therefore suffi-

cient to look at these discrete time points. We start with the before markers

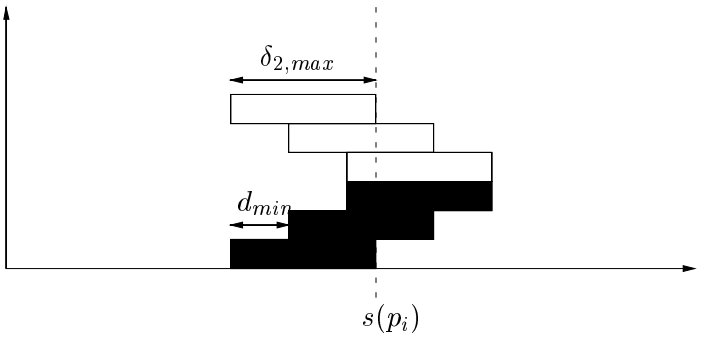

Figure 6.3: Before and after markers at departure time $s(p_i)$

(black) in Fig. 6.3. Trip p_i can have a before marker at time $s(p_i)$, this is the bottom rectangle in Fig. 6.3. The departure time of the next trip after p_i is at least $s(p_i) + d_{min}$, so that its before marker is also shifted by the amount d_{min}. The same argument holds for the next trip and so on. We get a maximum number of $1 + \lfloor \frac{\delta_{2,max}}{d_{min}} \rfloor$ before markers at time instance $s(p_i)$. If $\frac{\delta_{2,max}}{d_{min}}$ is an integer, we must also consider a before marker which exactly ends at time $s(p_i)$.

The maximum number of after markers (white) at time $s(p_i)$ is $1 + \lfloor \frac{\delta_{2,max}}{d_{min}} \rfloor$ with the same argument. We have to increase the lower bound for k by one for the locomotive start location markers and finally get:

$$k \geq 3 + \lfloor \frac{\delta_{2,max}}{d_{min}} \rfloor$$

The situation at the arrival times is analogous and doesn't change the lower bound for k.

We have omitted different locomotive types so far. We can introduce different locomotive types quite easily if we restrict the initial domains of the $r(p_i)$ appropriately: $r(p_i) = l(p_i) \subseteq M$ where $l(p_i)$ is the locomotive type requirement of trip p_i from Def. 4. Now trip p_i can only be assigned to locomotives with correct type $l(p_i)$.

6.2 Example

We show in Fig. 6.4 an assignment of five trips to two locomotives and the corresponding marker model. Although this is only a static view of the model (recall that the rectangle positions are FD-variables), it illustrates the

main idea. There are three turn locations A, B and C. The durations for a trip and a passive transport on the same route are identical. We have marked start markers with S, endmarkers with E and locomotive start markers with L. Before markers are black and after markers are white. Both locomotives have start position A. We get one passive transport between the trips C–A and B–A on the second locomotive.

The figure shows also a small mistake in the model, because gaps of width one are introduced between trips where the end and start locations are identical. The gaps of width one come from overlaps between after and end markers on the one hand and before and start markers on the other hand. We tried to solve the problem by separating the after and start markers of one location constraint into one `diff2`-constraint and the before and end markers into another, i.e. we have doubled the number of location `diff2`-constraints. This solves the problem but poses the additional question if we need before and end markers at all, because start and after markers are sufficient to get the gaps for the passive transports. We presume that they are necessary for a stronger propagation behaviour, especially if the predecessor of a trip is not determined at all times during search. As we would weaken this propagation by the separation into two constraints, we have decided to return to the old model. The mistake has no practical consequences as there must always be a considerable larger turn time between two trips in our locomotive problem.

6.3 Discussion of the Marker Model

Recall from Def. 4 that the location continuity constraint can be expressed by binary constraints between all pairs of trips (we express the implication in Def. 4 by a disjunction, replace ϕ by r and include the turn time Θ into the trip durations):

$$\begin{array}{lll} & \forall p_i, p_j \in P, i < j: & \\ r(p_i) \neq r(p_j) & \vee & (s(p_i) + d(p_i) + d(p_i, p_j) \leq s(p_j)) \\ & \vee & (s(p_j) + d(p_j) + d(p_j, p_i) \leq s(p_i)) \end{array}$$

Simonis discusses in [Sim95a] the disadvantages of this formulation as disjunctions. The number of binary constraints grows quadratically in the number of trips which generates a lot of overhead. Also the time to set up these constraints is in $O(n^2)$. The disjunctive formulation provides only little propagation because the domains for the start times $s(p)$ can only be narrowed if two trips have already been assigned to the same locomotive ($r(p_i) = r(p_j)$ and $|r(p_i)| = |r(p_j)| = 1$).

The formulation of the problem with `diff2`-constraints provides better propagation because it considers the twodimensional nature of the problem (time and locomotives). Propagation happens before a trip is actually assigned to a locomotive. For example, if the number of trips exceeds the number of available locomotives at a certain time instance, at least one start time interval must be reduced. This can be helpful if we have different locomotive types and a certain locomotive type is required by many trips. If the number of available locomotives of this type is too small compared to the number of trips that require it, the y-domain of some trips can be reduced.

We will illustrate this by an example (Fig. 6.5). Assume that the trips p_2 and p_3 have been constrained during search to the area which is indicated by the box. They are constrained to this area and have not yet been assigned to one of the two possible locomotives. Trip p_1 can move in an extended area which is indicated by the dotted outline. The `diff2`-constraint for all trip rectangles can detect that p_1 must be assigned to the top locomotive as the other two locomotives are already occupied by p_2 and p_3. Thus, the domain of $r(p_1)$ can be narrowed.

The time to set up the `diff2`-constraints is in $O(|T|n)$, this is an advantage compared to binary constraints if $|T| < n$. As the number of `diff2`-constraints grows linearly with the number of locations, this model should only be applied to situations where only a small number of locations is involved. The number of locations in a railway planning problem is usually small compared to the number of trips. This is not the case for other vehicle routing problems. In the vehicle routing problem from Section 4.3, we have as many locations as there are customers.

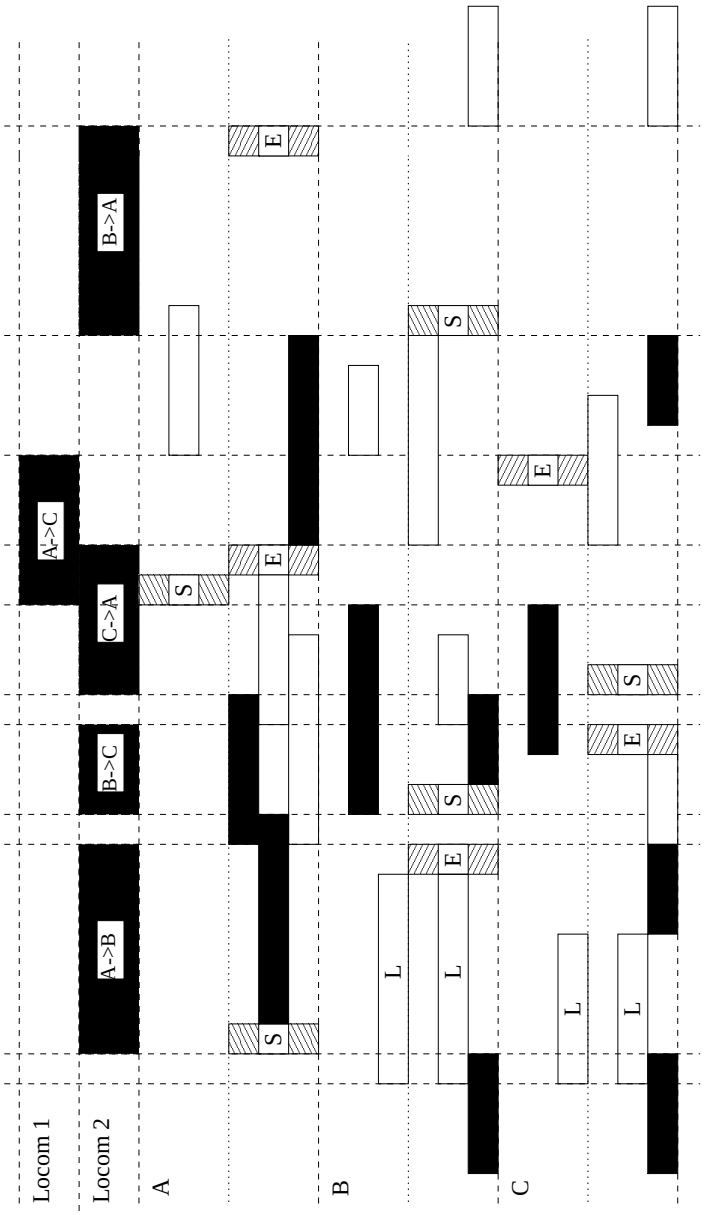

Figure 6.4: Example with five trips and two locomotives

Figure 6.5: Three trips and the area in which they can move

Chapter 7

Propagation Algorithm

7.1 Introduction

We will develop in this chapter a propagation algorithm which does pairwise reasoning for the rectangles in the `diff2`-constraint (see Def. 12). The purpose of this algorithm is to reduce the area in which a rectangle can move by considering the positions of the other rectangles. Our aim is a big reduction because a strong propagation reduces the search space. The algorithm must also detect the case when it is not possible to place the rectangles without overlap, in this case at least one rectangle domain becomes empty.

The algorithm was developed together with Anders Nordin [Nor]. The `diff2`-constraint is implemented in the CHIP system but the propagation algorithm has never been published. We want to give some solution ideas in this chapter. Related work can be found in [TAT91], where a dynamic algorithm for the sequential allocation of rectangles in the plane without overlap is discussed. Verdier et al. discuss in [dVT91] a constraint-based layout system which achieves arc-consistency for the non-overlap constraint. They work on more general objects, polygons with horizontal and vertical edges where the polygon points lie on a grid.

We will begin with the simpler case where all rectangles sizes are fixed (compare Def. 12), i.e. the widths and heights are FD-variables with a singleton domain and only the positions of the rectangles are undetermined. It can be easily shown that even the problem to decide if a set of movable rectangles can be placed without overlaps is NP-hard. The problem to reduce the areas in which the rectangles can move is then also NP-hard.

We reduce the NP-hard bin packing problem from [GJ79] to the decision problem if a set of movable rectangles can be placed without overlap:

Definition 13 (Bin Packing) *Given a finite set U of items, a size $s(u) \in \mathbf{Z}^+$ for each $u \in U$, an integer bin capacity $B > 0$ and an integer $K > 0$. Question: Is there a partition of U into disjoint sets $U_1, \ldots, U_K$ such that the sum of the sizes of the items in each U_i is B or less?*

In Fig. 7.1, a geometrical interpretation of the bin packing problem is shown. If we represent every item $u \in U$ by a rectangle of width $s(u)$ and height 1, the question is if all rectangles can be placed into a box of width B and height K. Every row in Fig. 7.1 corresponds to a bin in the bin packing problem. We can reduce the bin packing problem to the question if the $|U|$ rectangles which can all move in the same box of size BxK can be placed without overlap:

Bin packing problem has a solution $\Leftrightarrow$ Rectangles can be placed without overlap

Thus, we have shown that the rectangle problem is also NP-hard.

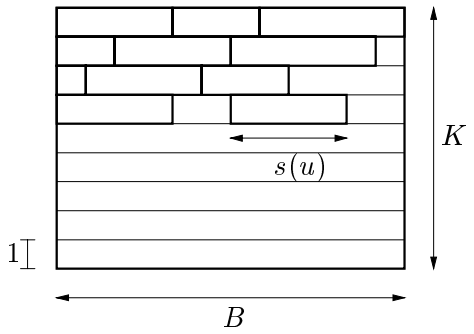

Figure 7.1: Geometrical interpretation of the bin packing problem

Recall from Fig. 5.4 that we defined the positions x_i and y_i of a rectangle R_i in the usual way: we worked on the grid $\mathbf{N}_0^2$ and labelled the grid lines with coordinates. In the following sections it will be more convenient to label the grid cells themselves with coordinates as shown in Fig. 7.2.

We give some basic definitions:

Definition 14 (Rectangular Area) *A rectangular area $A \subseteq \mathbf{N}_0^2$ is given by its left, right, bottom and top boundaries x_1, x_2, y_1, $y_2 \in \mathbf{N}_0$:*
$A = [x_1, x_2] \times [y_1, y_2]$ where $x_1 \leq x_2$ and $y_1 \leq y_2$.

The rectangle in Fig. 7.2 is given by $[x_i, x_i + w_i - 1] \times [y_i, y_i + h_i - 1]$ in our notation. We use the interval notation $[a, b]$ for the set of integers $\{a, a + 1, \ldots, b\}$.

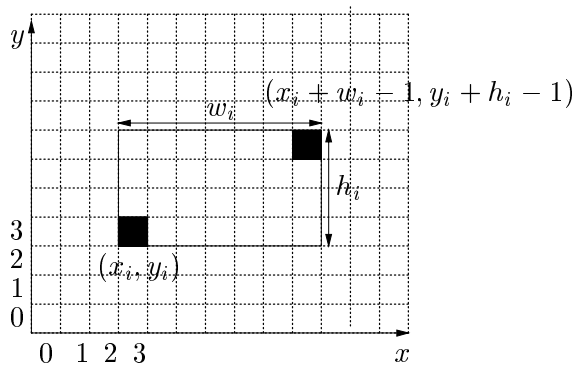

Figure 7.2: Grid N_0^2 and the new coordinate labelling

Definition 15 (Overlap) *Two rectangular areas A_1, A_2 overlap iff $A_1 \cap A_2 \neq \emptyset$.*

Definition 16 (Covering) *The rectangular area A_1 covers A_2 iff $A_2 \subseteq A_1$, i.e. A_1 includes A_2.*

Recall from Def. 12 that the FD-variables x and y constrain the bottom-left corner of rectangle R to a rectangular area (the grey region in Fig. 7.3)[1]. We call this region the *domain of the bottom-left corner* and define additionally the *domain of a rectangle D* as the area within which the rectangle must be included:

Definition 17 (Rectangle Domain) *The Domain D of a rectangle R with FD-variables $x \in [\underline{x}, \overline{x}]$, $y \in [\underline{y}, \overline{y}]$ for the position of the bottom-left corner, width w and height h is the area $D = [\underline{x}, \overline{x} + w - 1] \times [\underline{y}, \overline{y} + h - 1]$.*

We will now look at the conclusions we can draw from the domain and size of a rectangle.

Example 1 *Fig. 7.4 shows an example where we can compute a rectangular area K_1 which is necessarily covered by a rectangle R_1 with the domain D_1 in all its positions. Another rectangle R_2 with domain D_2 must not overlap K_1. This means that its domain D_2 can be reduced. Observe that D_2 doesn't remain rectangular if we remove the forbidden positions of R_2 so that we will introduce a new definition for rectangle domains in Section 7.3.*

The area K_1 can be regarded as a kernel region which is necessarily covered by R_1. We give a formal definition of this concept:

[1] i.e. x and y have intervals as initial domains. Domains with holes lead to several rectangular areas and this case will be explained later in the text.

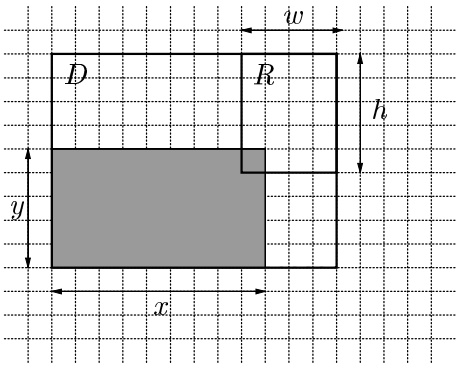

Figure 7.3: Rectangle R, rectangle domain D and the domain of its bottom-left corner

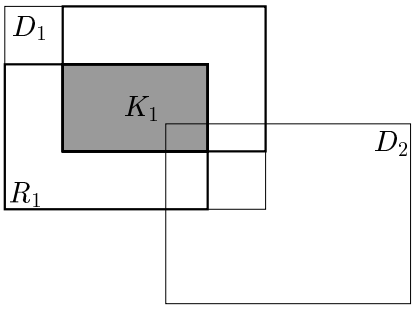

Figure 7.4: Reduction of the domain D_2

Definition 18 (Kernel) *The Kernel* $K(R, D) = [\alpha_1, \alpha_2] \times [\beta_1, \beta_2]$ *of a rectangle* R *with width* w *and height* h *and domain* $D = [x_1, x_2] \times [y_1, y_2]$ *is given by:* $\alpha_1 = \min\{x_1 + w - 1, x_2 - w + 1\}$, $\alpha_2 = \max\{x_1 + w - 1, x_2 - w + 1\}$, $\beta_1 = \min\{y_1 + h - 1, y_2 - h + 1\}$ *and* $\beta_2 = \max\{y_1 + h - 1, y_2 - h + 1\}$.

More informally, the kernel is obtained by putting the rectangle into the two extreme positions in the bottom-left and upper-right corner of its domain. The rectangular kernel is then given by the two corners of R which lie opposite to the domain corners in these positions.

7.2 Kernels

As the relative sizes of a rectangle and its domain change, the kernel must be interpreted differently. We will have a closer look at the different cases which can occur.

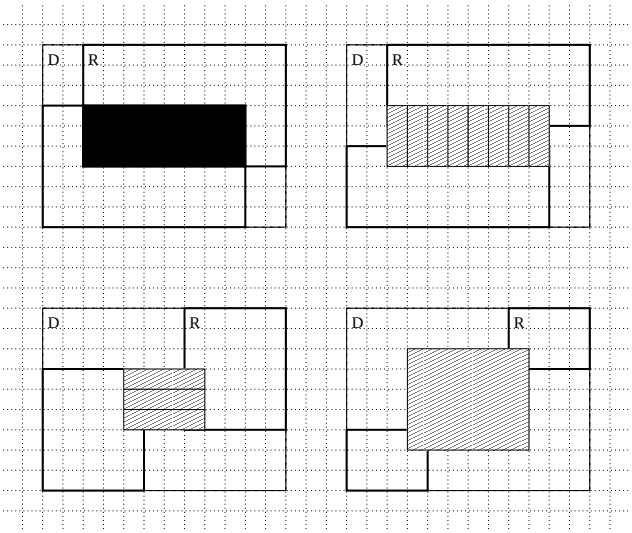

Figure 7.5: Four different cases for a rectangle R and its domain D

Figure 7.5 shows for every case the two extreme positions of a rectangle R in its domain D and the resulting kernels. We check if the x- and y-projections of the rectangle in these two positions overlap and get four different cases (we begin with the subimage in the top-left corner and proceed rowwise):

1. Overlap in x and y: the kernel K may not be *overlapped* by any other rectangle as explained in Example 1.

2. Overlap only in x: the kernel K can be divided into vertical strips of width one. Every strip must not be *covered* by another rectangle [Nor] because the x-projection of rectangle R covers the interval $[\alpha_1, \alpha_2]$ (see Def. 18) in all its positions. If a strip is entirely covered, R can't be placed in its domain D without overlap. In Fig. 7.5 we get eight vertical strips which must not be covered.

3. Overlap only in y: the kernel K can be divided into horizontal strips of height one. As in case 2, every strip must not be *covered* by another rectangle [Nor] because the y-projection of rectangle R covers the interval $[\beta_1, \beta_2]$ in all its positions. In Fig. 7.5 we have three horizontal strips that must not be covered.

4. No overlap: the kernel K must not be entirely *covered* by another rectangle [Nor]. If K is covered by another rectangle R', the projections of R and R' on both the x- and y-axis will overlap (it is not possible to move R enough outside in its domain).

We can now compute the forbidden areas for another rectangle R'. Figure 7.6 shows how to determine the forbidden positions for R'. In the first case the black area must not be overlapped whereas in the other three cases the dark shaded areas must not be covered. The figure shows for every case two extreme positions of R' in the forbidden area. The forbidden area D'' is indicated by a box. We will show in the next section how the forbidden

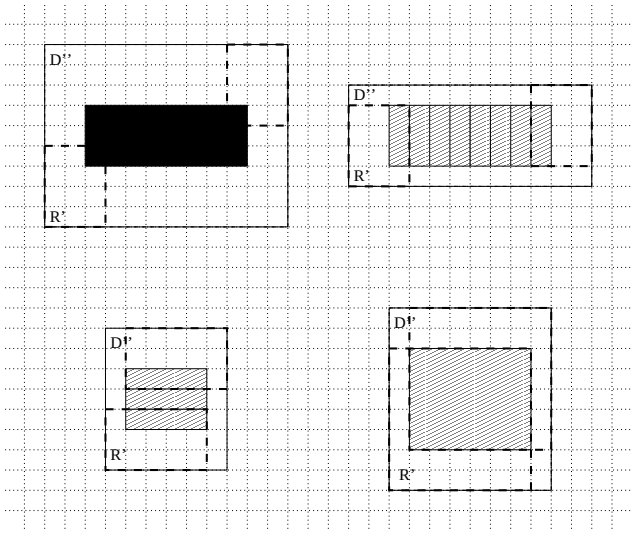

Figure 7.6: Forbidden positions for another rectangle R'

positions D'' can be removed from the domain D' of R'.

7.3 Domain Reduction

If we want to remove a rectangular part domain D'' from a domain D', it is easier to work on the correspondent domains of the bottom-left corner instead of the rectangle domains themselves. The direct correspondence between these two domain concepts was shown in Fig. 7.3.

Figure 7.7 shows a rectangle R' and its rectangle domain D'. In the beginning, the domain of its bottom-left corner is the grey area C' (left figure). We want to remove a rectangular area D'' from D'. D'' corresponds to a corner domain C'' (white rectangle in the interior of the grey area). We get a new domain of the bottom-left corner. It can be represented by a union of four rectangles (L, R, B and T in the right figure). The corresponding rectangle domains are indicated by boxes. There are also cases where we can get less than four rectangles (Fig. 7.8). If we have the fourth case in Fig. 7.8, a rectangle domain has become empty and we have detected an

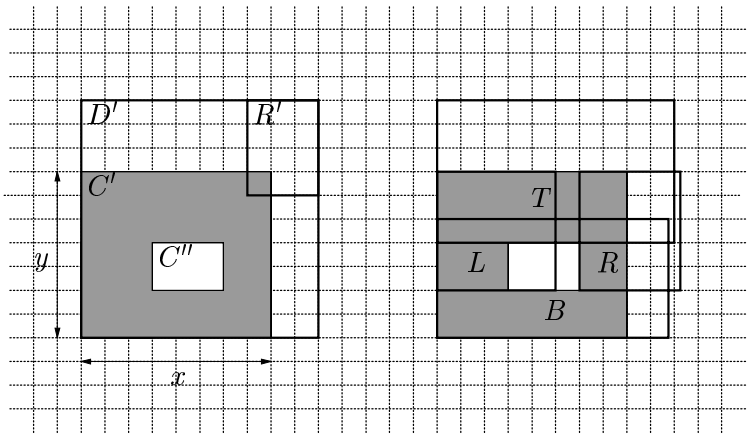

Figure 7.7: The removal of a rectangular part domain leads to four new part domains

overlap between rectangles. The `diff2`-constraint should produce a failure in this case.

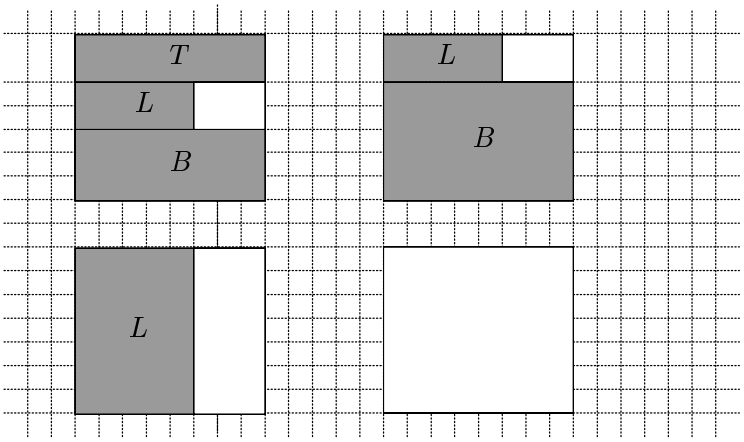

Figure 7.8: Some cases where we get less than four new part domains

We call the grey rectangles *part domains* of the corner and to these belong correspondent *part domains* of the rectangle (boxes). We will now return from the corner domains and work on rectangle domains again. The part domains of the rectangle build a set of domains:

Definition 19 (Domain Set of a Rectangle) *The domain set* $\mathcal{D} = \{D_1, \ldots, D_m\}$ *of a rectangle* R *consists of the part domains* $D_i = [x_1^i, x_2^i] \times$

$[y_1^i, y_2^i]$. *The union* $\bigcup_{i=1}^{m} D_i$ *describes the area the rectangle* R *must be included in.*

This definition subsumes Def. 17 for rectangle domains.

7.4 Several Part Domains and Forbidden Areas

We have explained how we can compute the forbidden area in the case when the domain set consists of one element. The kernel of this domain must be determined. We will now look at domain sets with several part domains. As we will see, the kernels of the part domains can be combined into a common kernel. [2]

Every part domain is rectangular so that we can compute its kernel. We sort the kernels into four different sets O, X, Y and C depending on the kernel type (compare Section 7.2). R denotes the considered rectangle and $\mathcal{D} = \{D_1, \ldots, D_m\}$ with $m > 1$ is its domain set:

Algorithm 1
input: Rectangle R and its domain set $\mathcal{D}$
output: sets of kernels O, X, Y and C

$O := X := Y := C := \emptyset$;
forall $D_i \in \mathcal{D}$ **do**
  $K := K(R, D_i)$;
  case $kerneltype(K)$ **of**
    $xy\_overlap\_type$: $O := O \cup \{K\}$;
    $x\_overlap\_type$: $X := X \cup \{K\}$;
    $y\_overlap\_type$: $Y := Y \cup \{K\}$;
    $no\_overlap\_type$: $C := C \cup \{K\}$;
  endcase
endfor

We compute for every part domain D_i the kernel $K(R, D_i)$ and decide which of the four cases from Section 7.2 occurs. If the domain set $\mathcal{D}$ contains m part domains, the complexity of this algorithm is $O(m)$. We can now distinguish two cases:

- Case 1: $X \cup Y \cup C = \emptyset$ (and $O \neq \emptyset$).

[2] All algorithms in this section are contributions from [Nor].

- Case 2: $X \cup Y \cup C \neq \emptyset$.

In case 1 we have only part domains with kernels that must not be overlapped, kernels of the O-type. Let $O = \{O_1, \ldots, O_\alpha\}$ denote the set of these kernels. We don't know in which part domain D_i the rectangle R will be placed as it can move freely in $\mathcal{D}$, but we know that the intersection

$$\mathcal{O} = \bigcap_{i=1}^{\alpha} O_i$$

must not be overlapped by any other rectangle R'. The computation of $\mathcal{O}$ can be done in time $O(m)$.

We call $\mathcal{O}$ the *non-overlappable area* of rectangle R in the following, as it must not be overlapped by any other rectangle. If $\mathcal{O} \neq \emptyset$, we can determine all forbidden positions of the other rectangle R' in the same way as shown in the first subimage of Fig. 7.6. Observe that $\mathcal{O}$ must be rectangular as it is an intersection of iso-oriented rectangles. The forbidden positions of R' must be removed from all part domains of R' in $O(m)$ time. After we have done that, we have finished the procedure for case 1.

If $\mathcal{O} = \emptyset$, we can't remove forbidden positions from another rectangle. In this case, we continue with the procedure for case 2 and try to compute *non-coverable* areas which we will explain in the following.

In case 2 we have $X \cup Y \cup C \neq \emptyset$, i.e. we have kernels that must not be covered. Kernels of the X-type contain vertical strips that must not be covered, kernels of the Y-type horizontal strips and kernels of the C-type must not be entirely covered. We will show how to compute a set $\mathcal{C} = \{C_1, ..., C_\beta\}$ of areas which must not be covered by any other rectangle. We will call $\mathcal{C}$ the *set of non-coverable areas*. We are obviously only interested in minimum areas C_i which must not be covered, i.e. these areas don't have a proper subset that must not be covered. If we enlarge a non-coverable area C_i, it can only be covered by a rectangle with bigger size and we can't remove as many forbidden positions from this rectangle.

If we go back to Fig. 7.5, we see that the following conditions must hold for any $C_i \in \mathcal{C}$:

- it must overlap all areas in O.
- it must cover at least one vertical strip of every area in X.
- it must cover at least one horizontal strip of every area in Y.
- it must cover all areas in C.

These four conditions must be fulfilled so that we can be sure that R can't be placed in its domain set $\mathcal{D}$ if C_i is entirely covered. If all areas in O are overlapped by C_i and C_i is covered by another rectangle, R can't be placed in the corresponding part domains. The situation for the other three kernel types is similar.

If we look at the projections of the two-dimensional areas onto the x- and y-axis, this can be formulated by the following relations between the resulting intervals:

- x-projection of C_i:
  - overlap all x-projections of the O-areas.
  - overlap all x-projections of the X-areas.
  - cover all x-projections of the Y-areas.
  - cover all x-projections of the C-areas.
- y-projection of C_i:
  - overlap all y-projections of the O-areas.
  - cover all y-projections of the X-areas.
  - overlap all y-projections of the Y-areas.
  - cover all y-projections of the C-areas.

We define covering and overlap of intervals in a similar way to the two-dimensional case (inclusion and intersection). Let A_x (A_y) denote the set of intervals which is obtained by projecting the rectangular areas in set A on the x-axis (y-axis). $I(A_c)$ is the intersection of this interval set ($c \in \{x, y\}$) and $C(A_c)$ the smallest interval that covers all members in the interval set. The set of non-coverable areas $\mathcal{C}$ can be computed by the following algorithm:

Algorithm 2
input: kernel sets O, X, Y and C
output: projections $\mathcal{C}_x$ and $\mathcal{C}_y$ of $\mathcal{C}$ and boolean flags $XSegm$, $YSegm$.

01**begin**
02**interval** $\mathcal{C}_x := \mathcal{C}_y := \emptyset$;
03**boolean** $XSegm := YSegm := false$;
04**interval** $temp := \emptyset$;
05
06% x-projection
07**if** $Y \cup C = \emptyset$ **then** $\mathcal{C}_x := I(O_x \cup X_x)$; **endif**;
08**if** $\mathcal{C}_x \neq \emptyset$

09**then** $XSegm := true$;
10**else**
11 **if** $Y \cup C \neq \emptyset$ **then** $temp := C(Y_x \cup C_x)$; **endif**
12 **if** $O \cup X \neq \emptyset$ **then** $temp$:=ExtendToOverlap($temp$,$O_x \cup X_x$); **endif**
13 $\mathcal{C}_x := temp$; % $XSegm = false$
14**endif**
15
16% y-projection
17**if** $X \cup C = \emptyset$ **then** $\mathcal{C}_y := I(O_y \cup Y_y)$; **endif**;
18**if** $\mathcal{C}_y \neq \emptyset$
19**then** $YSegm := true$;
20**else**
21 **if** $X \cup C \neq \emptyset$ **then** $temp := C(X_y \cup C_y)$; **endif**
22 **if** $O \cup Y \neq \emptyset$ **then** $temp$:=ExtendToOverlap($temp$,$O_y \cup Y_y$); **endif**
23 $\mathcal{C}_y := temp$; % $YSegm = false$
24**endif**
25**end.**

The output of this algorithm are the projections of $\mathcal{C}$ on the x- and y-axis. As will be explained later, $\mathcal{C}_x$ and $\mathcal{C}_y$ can consist of a single interval or a set of consecutive unit intervals. The output of the algorithm is actually not $\mathcal{C}_x$ and $\mathcal{C}_\dagger$ themselves, but $\mathcal{C}_x$ and $\mathcal{C}_y$ represent an interval and the flags $XSegm$ and $YSegm$ indicate if we have the single or unit interval case.

Algorithm 2 has complexity $O(m)$ because the operations I, C and ExtendToOverlap have complexity $O(m)$. The algorithm considers the projections of $\mathcal{C}$ on the x- and y-axis separately, we start with the x-projection.

The x-projections of the areas in O and X must be overlapped by a non-coverable area C_i, whereas the x-projections of the areas in Y and C must be covered. If $Y \cup C = \emptyset$ (line 7), we have only projections that must be overlapped. The projection of C_i must overlap the projections of all areas in O and X, thus we calculate the intersection $I(O_x \cup X_x)$ of these projections (line 7). If $I(O_x \cup X_x) = [a, b] \neq \emptyset$, we know that the projection of a C_i needs only to cover one of the unit intervals in $\{[a, a], [a+1, a+1], ..., [b, b]\}$. We write $[x, x]$ for a unit interval according to our labelling of grid cells (Section 7.1). We want to compute minimum areas C_i. If the projection of C_i shall overlap all projections of the areas in O and X, it is sufficient to overlap one unit interval in the intersection $I(O_x \cup X_x)$. We set $\mathcal{C}_x = [a, b]$ and set the boolean flag $XSegm$ which means that a region C_i has width one (line 9), it is a vertical strip.

If $\mathcal{C}_x = \emptyset$ (line 10), then either $I(O_x \cup X_x) = \emptyset$ or we didn't enter the `then`-part in line 7 because $Y \cup C \neq \emptyset$. If $Y \cup C \neq \emptyset$, we have also projections which must be covered. We compute an interval $temp := C(Y_x, C_x)$ that

covers all projections of the areas in Y and C (line 11). We use then the following procedure to extend *temp* to the smallest interval that overlaps all projections of O and X as well (line 12). The return value of this procedure is an interval, its first argument an interval and its second argument a set of intervals:

```
01interval procedure ExtendToOverlap([α,β],{[α1,β1],...,[αn,βn]})
02begin
03  if [α,β] = ∅ then
04    return [min_{1≤i≤n} βi, max_{1≤i≤n} αi];
05  else
06    for i := 1 to n do
07    if [α,β] ∩ [αi,βi] = ∅ then
08      if β < αi then
09        β := αi;
10      else
11        α := βi;
12      endif;
13    endif;
14    endfor;
15  return [α,β];
16  endif;
17end
```

The procedure has complexity $O(n)$. If the procedure gets an empty interval which shall be extended to overlap with all $[\alpha_i, \beta_i]$, it returns the interval which has as lower bound the minimum of the right interval ends and as upper bound the maximum of the left interval ends (line 4). This is the interval of minimum length which overlaps all $[\alpha_i, \beta_i]$. If $[\alpha, \beta] \neq \emptyset$ (line 5), we must extend it until it overlaps all $[\alpha_i, \beta_i]$. We go through all $[\alpha_i, \beta_i]$ (line 6) and check first if $[\alpha, \beta] \cap [\alpha_i, \beta_i] = \emptyset$ (line 7). If these two intervals already overlap, we needn't change $[\alpha, \beta]$. Otherwise we increase β if $[\alpha_i, \beta_i]$ lies on the right side of $[\alpha, \beta]$ (line 9). If $[\alpha_i, \beta_i]$ lies on the left side of $[\alpha, \beta]$, we must decrease α (line 11). We return $[\alpha, \beta]$ as the extended interval after we have completed the **for**-loop (line 15).

The procedure ExtendToOverlap can be entered with $temp = \emptyset$ if $Y \cup C = \emptyset$ and $I(O_x \cup X_x) = \emptyset$ in line 7 of algorithm 2. In this case, we don't enter the **then**-part in line 11 and *temp* remains empty. $I(O_x \cup X_x) = \emptyset$ means that the intersection of the x-projections of the O- and X-areas is empty. ExtendToOverlap returns then the smallest interval that overlaps all projections of O and X. Recall that we said in case 1 that we continue with case 2 if the non-overlappable area $\mathcal{O}$ was empty. In this case, we had $O \neq \emptyset$, $X = \emptyset$ and $I(O_x \cup X_x) = I(O_x) = \emptyset$ and ExtendToOverlap is

entered with $temp = \emptyset$, i.e. we compute the smallest interval which overlaps all x-projections of O. This leads also to a non-coverable area C_i, because if all O-areas are overlapped by a C_i the rectangle can't be placed in the corresponding part domains.

We return now to algorithm 2, line 13. We set $\mathcal{C}_x := temp$ and the boolean flag $XSegm$ remains unset i.e. a region C_i has the same width as $\mathcal{C}_x$ because the whole interval $\mathcal{C}_x$ must be covered by a C_i.

The computation of $\mathcal{C}_y$ and $Ysegm$ is symmetrical (lines 16-25). $\mathcal{C}$ is now given by its projections $\mathcal{C}_x$ and $\mathcal{C}_y$. The flags $Xsegm$ and $Ysegm$ indicate the form of the C_i:

- $Xsegm = true$: vertical strips C_i with width one.
- $Ysegm = true$: horizontal strips C_i with height one.
- $Xsegm = Ysegm = false$: one non-coverable area C_1.

This is illustrated in Fig. 7.9.

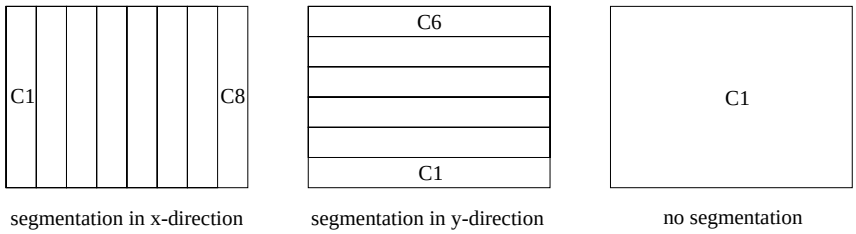

Figure 7.9: Possible configurations of set $\mathcal{C}$

Observe that the case $Xsegm = Ysegm = true$ can't occur as we would then have $Y \cup C = \emptyset$ (lines 7 and 9) and $X \cup C = \emptyset$ (lines 17 and 19). This means $X = Y = C = \emptyset$. We do not enter algorithm 2 if $O = \emptyset$ (then we would have no part domains at all), so that we will assume $O \neq \emptyset$ and $X = Y = C = \emptyset$. In this case we compute first a non-overlappable area $\mathcal{O}$ with the procedure for case 1. If $\mathcal{O} \neq \emptyset$ we do not enter algorithm 2. If $\mathcal{O} = \emptyset$ and $O \neq \emptyset$, the **if**-condition in line 11 is false (because $Y = C = \emptyset$), i.e. $temp$ remains unset. We enter ExtendToOverlap in line 12 with $temp = \emptyset$, i.e. we compute the smallest interval that overlaps all intervals in O_x. $XSegm$ remains false (line 13). The same result is obtained for the y-projection. We finally get one area C_1 which overlaps all kernels in O and $Xsegm = Ysegm = false$ in contradiction to our assumption.

Example 2 *Fig. 7.10 shows an example for the computation of $\mathcal{C}$. We have two kernels of the O-type, O_1 and O_2 and one kernel of the X-type. This*

means $X \cup O \neq \emptyset$ and $Y = C = \emptyset$, which is case 2 and we enter algorithm 2. The ***if****-condition in line 7 is true, because $Y \cup C = \emptyset$, we compute the intersection $I(O_x \cup X_x)$ and set $\mathcal{C}_x = I(O_x \cup X_x)$. This projection is divided into unit intervals, we set $XSegm = true$ (line 9). The projection $\mathcal{C}_x$ is shown in Fig. 7.10.*

We continue with the y-projection. The ***if****-condition in line 17 is false, because $X \cup C \neq \emptyset$. In line 21, we compute the interval $C(X_y \cup C_y)$ which covers the y-projection of the single X-kernel. $C(X_y \cup C_y) = X_y$ is shown in Fig. 7.10. The* ***if****-condition $O \cup Y \neq \emptyset$ in line 22 is true, we extend $temp = X_y$ so that it overlaps also the projections $O_{1,x}$ and $O_{2,x}$. We get $\mathcal{C}_y$ as shown in Fig. 7.10, it is not divided into unit intervals ($YSegm = false$, line 23). By combining $\mathcal{C}_x$ and $\mathcal{C}_y$, we get two areas C_1 and C_2 which must not be covered by another rectangle.*

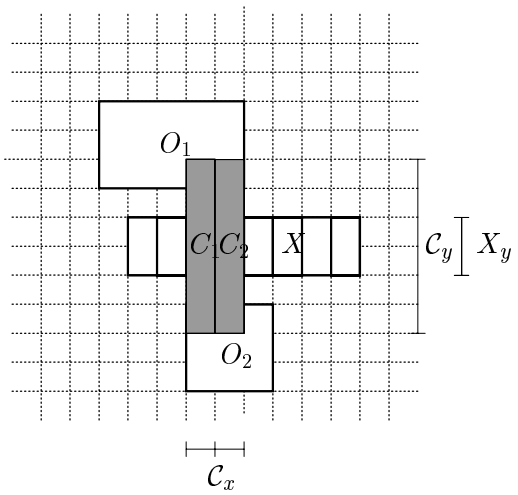

Figure 7.10: Example for a set of non-coverable areas $\mathcal{C}$

We have shown in this section how to compute a common kernel for a rectangle R with several part domains. Either we get a non-overlappable area $\mathcal{O}$ (case 1) or we get a set of non-coverable areas $\mathcal{C}$. As we have seen, both $\mathcal{O}$ and $\mathcal{C}$ are rectangular. Thus, we can compute the forbidden positions for another rectangle R' exactly in the same way as shown in Fig. 7.6. We remove these positions from the domain set $\mathcal{D}'$ as described in Section 7.3. All algorithms in this section have complexity $O(m)$ so that we can determine $\mathcal{O}$ and $\mathcal{C}$ in $O(m)$ time.

7.5 Pairwise Reasoning

We know how to reduce the domains of a rectangle pair. We can now give the overall algorithm which considers all rectangle pairs for a set $\mathcal{R} = \{R_1, ..., R_n\}$ of n rectangles. We use the AC3-algorithm which was introduced in Section 5.4:

Algorithm 3 (AC3)
input: set $\mathcal{R} = \{R_1, \ldots, R_n\}$ of rectangles and their domain sets $\mathcal{D}_1, \ldots, \mathcal{D}_n$
output: pairwise consistent rectangle domain sets

begin
  $L := \{(i,j) | i \neq j \wedge \mathcal{D}_i$ and $\mathcal{D}_j$ overlap$\}$;
  while $L \neq \emptyset$
    $L := L - \{(i,j)\}$;
    reduce $\mathcal{D}_i$ by determining the $\mathcal{O}$ or $\mathcal{C}$ areas of R_j;
    if $\mathcal{D}_i$ has changed
    then $L := L \cup \{(k,i) | j \neq k \neq i$ and $\mathcal{D}_k$ and $\mathcal{D}_i$ overlap$\}$;
    endif;
  endwhile;
end.

The set L contains the arcs in an abstract version of the constraint graph $G = (\{1, \ldots, n\}, L)$ where every node represents a rectangle. We collapse the variables x_i, y_i of rectangle R_i into one node because they are only influenced by variables of other rectangles.

We add the arc (i,j) to the constraint graph if the domainsets $\mathcal{D}_i$ and $\mathcal{D}_j$ overlap, otherwise the rectangles R_i and R_j can never overlap. As long as $L \neq \emptyset$, we remove an arc (i,j) from L and establish its consistency. We have to insert arcs into L again which could be affected by a smaller domain $\mathcal{D}_i$ because its $\mathcal{O}$- or $\mathcal{C}$-areas have been enlarged.

The time complexity for the determination of non-overlappable and non-coverable areas is $O(m)$, where m denotes the maximum number of part domains of a rectangle (compare Section 7.4). We must remove the forbidden positions from $O(m)$ part domains of $\mathcal{D}_i$, this can also be done in $O(m)$ time (the removal of forbidden positions in Section 7.3 is done in time $O(1)$). A lower bound for the AC3 algorithm is therefore $\Omega(mn^2)$ (compare Section 5.4), as we must examine every arc in the constraint graph at least once. An arc can be inserted d times into L, where d denotes the maximum domain size of a rectangle [3]. The worst-case run time is $O(dmn^2)$ (compare Section 5.4).

[3] i.e. the number of rectangle positions, $d = wh$ where w denotes the width and h the height of the corner domain.

d can be quite large and is not related to the other parameters n and m. It depends on the domain size of the variables for the rectangle positions which can vary from problem to problem. In the worst-case, d is a measure for the number of repeated arc inspections in the AC3-procedure.

m depends on d because the number of part domains grows with every removal of forbidden positions. Thus, m is difficult to estimate. We can show however that m can only grow by a constant in every removal step. This is illustrated in Fig. 7.11. Assume that we have four corner part domains

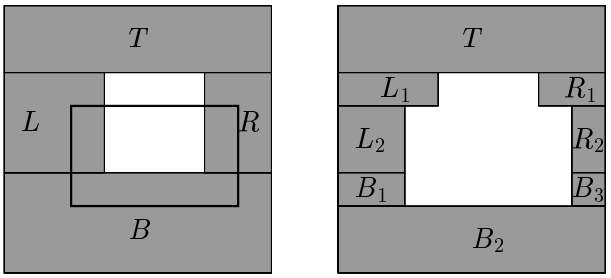

Figure 7.11: Growth of the number of part domains

B, L, R and T and that we want to remove the area in the thick box. T remains unchanged and the other three part domains are split because at least one corner of the box lies in these part domains. L is split into L_1, L_2, R in R_1, R_2 and B into B_1, B_2 and B_3. Observe also that it would be possible to merge L_2, B_1 and R_2, B_3 in order to reduce the number of part domains.

We remove always rectangular areas. Only part domains which contain a corner of the removed area are split (compare Fig. 7.7, 7.8). We look at the different cases in Fig. 7.7 and 7.8 and see that one corner can generate one new part domain:

- one corner inside of a part domain: two new part domains
- two corners inside: three new part domains
- four corners inside: four new part domains

In the worst-case, each of the four corners of the removed area generates a new part domain. We can get at most four new part domains in each removal step.

7.6 Examples

The pairwise reasoning will be illustrated by means of two examples.

Example 3 *We look at Fig. 7.12 where we have to rectangles R_1 and R_2 with sizes 6x6 and 2x2 which can both move in an area of 8x8. We show in this and the following example always rectangle domains (thick lines), not corner domains. We compute the kernel of R_1, it is of the O-type (grey). R_2 must not overlap O and we remove the positions of R_2 where R_2 overlaps O. We get four new part domains, strips of width two. We compute for these part domains the kernels and get two kernels of the X-type, two of the Y-type. Algorithm 2 computes the set of non-coverable areas $\mathcal{C}$. It consists of one area C_1 which covers at least one strip in each of the four kernels. R_1 covers C_1 in its middle position, we remove this position from the domain of R_1 and get four new part domains. Each part domain contains exactly one position of R_1. We compute the kernels of these part domains, they are all of the O-type because the part domains contain one rectangle position. We compute the intersection $\mathcal{O} = \bigcap_{i=1}^{4} O_i$ which is the same area as the kernel O in the beginning, i.e. we can't remove further positions from R_2.*

Example 4 *Fig. 7.13 shows a bigger example with four rectangles R_1, R_2, R_3 and R_4 of size 4x1, 1x4, 3x2 and 2x3. They can all move in the same area with size 5x4. It is obviously not possible to place all four rectangles in this area. Our algorithm starts with R_1 and computes its kernel. It is of the X-type and contains three vertical strips. These strips can be covered by R_2, so that we remove the three middle positions from R_2. We get two new part domains and compute their kernels (O_1 and O_2). The intersection $\mathcal{O} = O_1 \cap O_2$ is empty, we compute instead a set of non-coverable areas. Each non-coverable area C_i must overlap both O_1 and O_2, we get four non-coverable areas C_1–C_4. They can't be covered by one of the other rectangles so that we can't remove positions from other rectangles. We compute the kernel of R_3 and get one strip of the X-type. This strip can be covered by R_4. We remove the forbidden positions of R_4 and get two new part domains with the kernels O_1 and O_2. Their intersection $\mathcal{O} = O_1 \cap O_2$ is empty, we compute $\mathcal{C}$ and get two areas C_1 and C_2 (an area C_i must overlap both O_1 and O_2). These strips can be covered by R_1 and R_3. We remove the forbidden positions of R_3 and get two new part domains with size 3x4, D_1 and D_2. They have kernels of the X-type (three strips each) which are combined into one non-coverable area C by algorithm 2. It is identical to the old kernel X of R_3 so that we get no new information out of this. We remove also the forbidden positions of R_1 and get two new part domains with the kernels O_1 and O_2. $O_1 \cap O_2 = \emptyset$, thus we compute $\mathcal{C}$ and get three non-coverable*

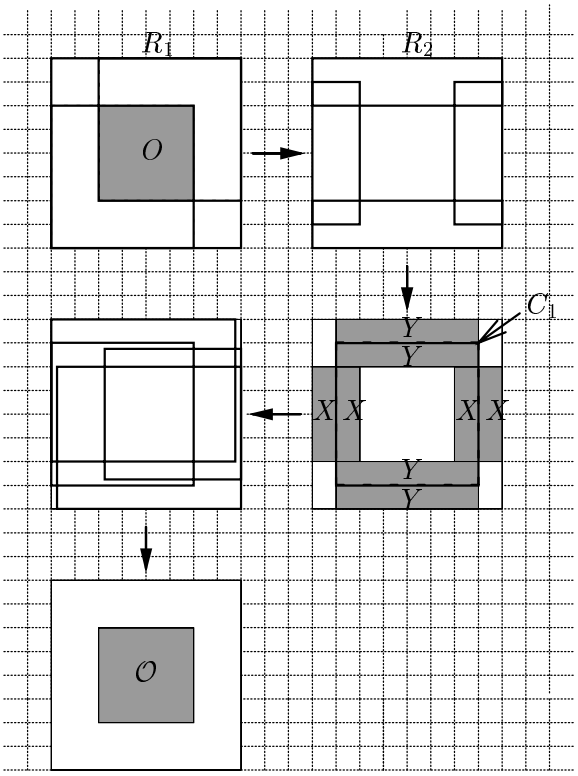

Figure 7.12: Example with two rectangles, sizes 6x6 and 2x2

areas C_1–C_3. They can't be covered by another rectangle and the propagation stops.

Our algorithm can't detect that it is not possible to place all four rectangle in the area, because it looks only at two rectangles at a time.

7.7 Stronger Consistency

The algorithm described in the preceeding sections does only pairwise reasoning for a rectangle set. For stronger consistency like k-consistency [Tsa93], we consider propagation based on rectangle area calculations. We can detect whether an area in the plane is too crowded by rectangles by calculating the sum of the areas of the rectangles which reside in this area. If the area is too crowded, we can possibly move some rectangles outside and reduce their domains. We have implemented a simple heuristic where we look at each of the rectangles separately and decide if its domain can be reduced: [4]

[4]Observe that this heuristic doesn't help in Example 4 either, but in other cases.

Algorithm 4
input: set $\mathcal{R} = \{R_1, \ldots, R_n\}$ of rectangles and their domain sets
$\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$
output: reduced rectangle domain sets

begin
forall $R_i \in \mathcal{R}$ **do**
B:=bounding box of $\{\mathcal{D} - \{\mathcal{D}_i\}\}$;
if $area(\mathcal{R}) > area(B)$ **then**
reduce the domain of R accordingly;
endfor
end.

A bounding box is here the smallest enclosing rectangle and $area()$ denotes the area sum of a rectangle set or a rectangle. For every rectangle R_i, we decide if the bounding box B of $\mathcal{D} - \{D_i\}$ becomes too crowded if we move R_i inside B. If yes, we can move R_i away from B and reduce the domain of R_i. The time complexity of this algorithm is $O(n^2)$, because every bounding box computation has complexity $O(n)$ if we store the bounding box of a rectangle domain set $\mathcal{D}_i$ separately.

Example 5 *In Fig. 7.14, we have three rectangles R_1, R_2 and R_3. The bounding box of the domains of R_1 and R_2 is B. There is an empty space of four unit squares left in B, the domain of R_3 is D_3. We can't move R_3 completely inside B, as there is not enough space for it. We remove the position of R_3 where R_3 is contained in B and get two new part domains for R_3.*

This is of course a simple heuristic, because all rectangle subsets of size $n - 1$ are considered. It would be better to consider only rectangle subsets in crowded areas, because they are good candidates for a domain reduction. It should be further investigated how to find crowded areas and the corresponding rectangle subsets. One idea for finding crowded areas could be to build the overlap graph of the rectangle domains. In this graph, we search after big cliques with a heuristic. We get rectangle subsets where the domains overlap heavily, i.e. crowded areas.

7.8 Variable Rectangle Sizes

All algorithms described so far can be easily adapted to the case with variable rectangle sizes if all calculations are based on the minimum rectangle size. The computation of kernels in Fig. 7.5 remains correct if we assume

minimum rectangle sizes, because all necessarly covered areas remain covered if the rectangle becomes bigger. This is illustrated in Fig. 7.15 for a rectangle with an O-type kernel. The kernel is enlarged for a bigger rectangle, but the old kernel area is still included in the new one. Thus, all other rectangles that overlapped the old kernel overlap also the new kernel. The situation for the other three kernel types is similar.

The calculation of forbidden positions in Fig. 7.6 remains also correct, as all forbidden positions for a minimum size rectangle remain forbidden for a bigger one. We can see this in Fig. 7.16 for an O-type kernel. The area of forbidden positions grows with the rectangle size, but the old area is still included in the new one. This argument holds also for the other three kernel types.

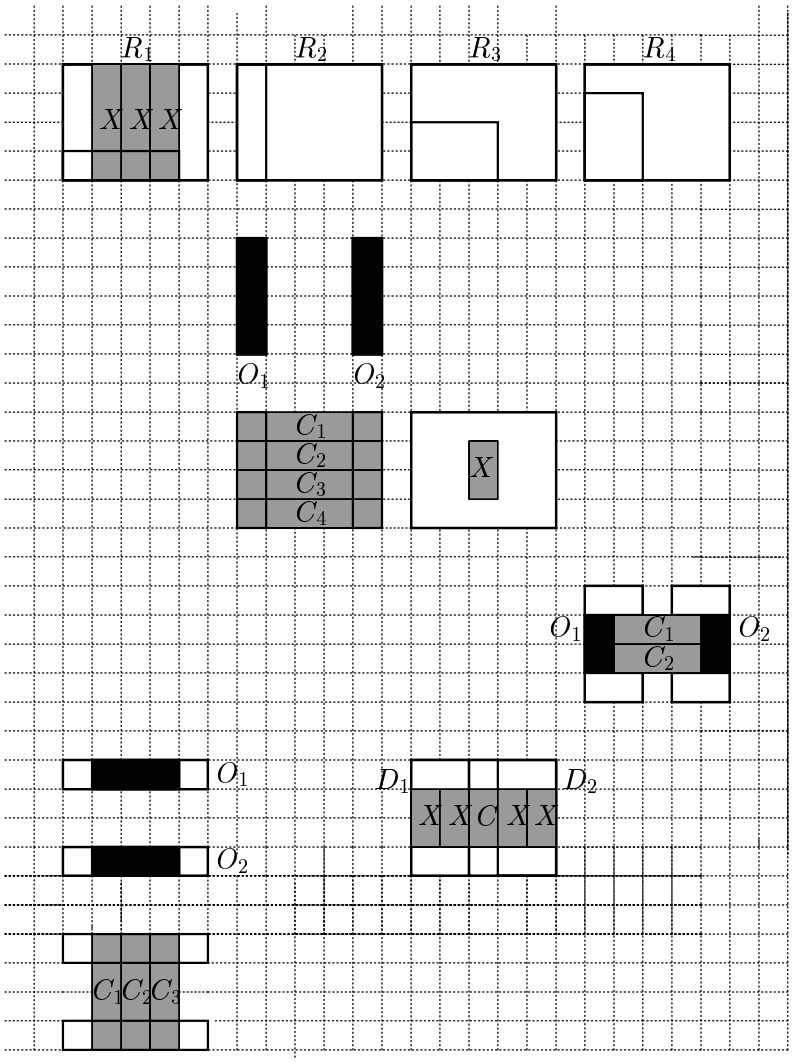

Figure 7.13: Example with four rectangles, sizes 4x1, 1x3, 3x2, 2x3

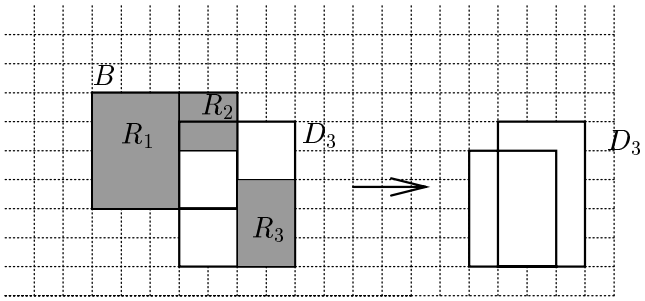

Figure 7.14: Three rectangles, bounding box B

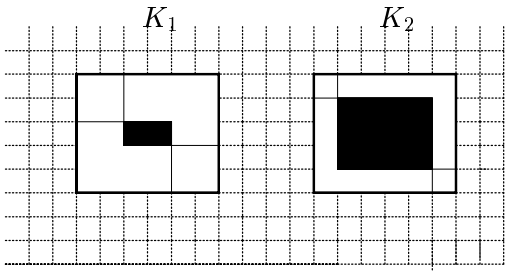

Figure 7.15: Change of the kernel size

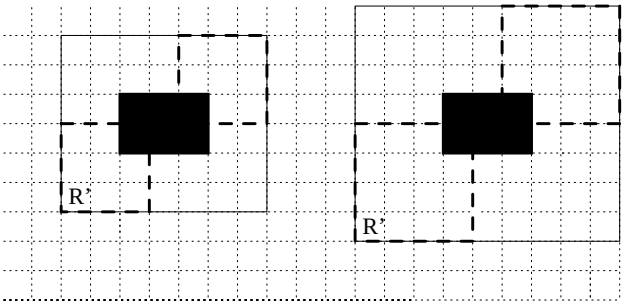

Figure 7.16: Change of the forbidden area

Chapter 8

Heuristics

We will describe in this chapter several heuristics which build locomotive routes. The constraint model from Chapter 6 ensures only feasible locomotive routes, it provides no optimization with respect to the number of locomotives and the amount of passive transport.

We need quite specialized search strategies for our locomotive assignment problem as TSP–like problems [1] haven't been solved satisfactory by constraint propagation yet. The propagation in former approaches is too weak for a reduction of the search space and only small problems can be solved to optimality [CL97]. Thus, we use route building heuristics which construct one solution. We walk down on one path of the search tree and do not explore the rest of the search tree.

We can't benefit from the main strength of constraint programming, constraint propagation. But the built-in mechanism of constraint programming languages to dynamically handle the constraints simplifies the programming task, because the various interactions of the constraints are implicit during program execution. It is not necessary to formulate the constraint checks in an algorithm, they are implicitely done by the constraint system.

8.1 Introduction

The main idea in our approach is the simultaneous planning of locomotive assignment and track allocation. The mutual exclusion condition for trains on tracks has an effect on the departure time windows of the trips. These in turn influence the vehicle routes, so that we should always take into account the constraints from the track allocation problem while doing locomotive assignment.

[1] Traveling Salesman Problem

This can be easily done in the TUFF system. The constraints for the track allocation problem are implemented in this system and can be used during the construction of the locomotive routes. We will describe the track allocation constraints of the TUFF system in Chapter 9. The constraints check implicitely if the constructed locomotive routes are possible and guide the route construction algorithm in case of failures.

We proceed in two steps in order to compute the timetable and the assignment of trips to locomotives for a certain problem. We start with the determination of the trip order for every locomotive. We call this a *locomotive plan*. In the locomotive plan, all passive transports are known but the departure times are still flexible because only the trip order is determined. We will describe several heuristics which construct locomotive plans in the following sections.

In the second step, we determine the departure times and solve the track allocation problem at a detailed level. We use a simple search heuristic which has been developed for track allocation in the TUFF system. It will be explained in Chapter 9.

Our cost function for the vehicle routes has two parts. Our primary objective is two minimize the number of used locomotives. For this minimum number of locomotives, the amount of passive transport time should be minimized. Our heuristics do not consider the waiting times of the locomotives as they are not as important as the other two cost factors. We won't address the problem of locomotive types in the beginning and explain later how the algorithms can be extended to different locomotive types.

8.2 Best Predecessor Heuristic

Let $M = \{l_1, \ldots, l_m\}$ denote the set of locomotives and $P = \{p_1, \ldots, p_n\}$ the set of trips. Trip p_i has the start location $S(p_i)$ and the end location $E(p_i)$. Its departure time window is $[\tau_{min}(p_i), \tau_{max}(p_i)]$. The locomotive l_j has the start position $\Sigma(l_j)$.

For every locomotive, we want to compute its trip order. We store the trip order for the locomotive l_j in a trip list $L_j = [p_{j_1}, \ldots, p_{j_\alpha}]$ with $\alpha = \alpha(j)$. The set of all trip lists determines the locomotive plan.

Best Predecessor Heuristic
input: trips $P = \{p_1, \ldots, p_n\}$
locomotives $M = \{l_1, \ldots, l_m\}$
output: trip order list $L_j = [p_{j_1}, \ldots, p_{j_\alpha}]$ for every locomotive l_j

```
01 begin
02   for j := 1 to m do
03     list L_j := nil;
04   endfor
05   list P := [p_{i_1}, ..., p_{i_n}] := sort(P, τ_max);
06   for k := 1 to n do
07     set C := ∅;
08     for j := 1 to m do
09       if p_{i_k} can be added to L_j
10         if L_j ≠ nil
11           then integer c_j := c(last(L_j), p_{i_k});
12           else integer c_j := c(Σ(l_j), p_{i_k}) + λ;
13         endif
14         C := C ∪ {c_j};
15       endif
16     endfor
17     list C' := [c_{k_1}, ..., c_{k_m}] := sort(C);
18     list A := [k_1, ..., k_m];
19     repeat
20       integer j := first(A);
21       A := tail(A);
22       L_j := append(L_j, p_{i_k});
23       boolean successful := false;
24       if constraints violated
25         then L_j := L_j − last(L_j);
26         else successful := true;
27       endif
28     until successful or A = nil;
29     if A = nil then print p_{i_k} not added; endif
30   endfor
31 end.
```

Assume that we we want to assign the trips to the locomotives in a certain order, we sort them e.g. after increasing latest departure time τ_{max} and begin with the earliest trips. We build up the locomotive plan from left to right. This sort step is done in line 5.

We go through the sorted trip list with the **for**-statement in line 6. A straightforward strategy is to add the current trip p_{i_k} at the end of the trip list where it fits best. We check first if the time window of trip p_{i_k} allows an addition to the list end L_j (line 9), i.e. if its latest departure time τ_{max} is greater than the current minimum route time. We compute then a cost

$$c(p_i, p_j) = \alpha_1 c_1(p_i, p_j) + \alpha_2 c_2(p_i, p_j), \quad \alpha_1, \alpha_2 \geq 0, \quad \alpha_1 + \alpha_2 = 1$$

for the connection of trip p_i with trip p_j. c_1 is the cost for a necessary passive transport

$$c_1(p_i, p_j) = \delta_2(E(p_i), S(p_j))$$

and $c_2(p_i, p_j)$ is the minimum waiting time of the locomotive before it can start with trip p_j after completing trip p_i:

$$c_2(i, j) = \tau_{min}(p_j) - (\tau_{min}(p_i) + d(p_i))$$

$d(p_i)$ is the duration of trip p_i and can be approximated by its minimum duration. Recall from Chapter 3 that the trips have variable waiting times at the locations, so that the trip duration can also vary. We assume that trip p_i starts at $\tau_{min}(p_i)$, so that we get a worst-case waiting time. δ_2 is the travel time for passive transports from Def. 4 in Chapter 3.

If we assign trip p_{i_k} to a new locomotive, we must take into account the start location of the locomotive. We define the cost

$$c(\Sigma(l_j), p_{i_k}) = \alpha_1 \delta_2(\Sigma(l_j), S(p_{i_k})) + \alpha_2 \tau_{min}(p_{i_k})$$

which includes the passive transport from the start location of the locomotive and the waiting time until the locomotive can start with trip p_{i_k}. We add a penalty $\lambda > 0$ to this cost (line 12) in order to force the heuristic to use few locomotives.

These connection costs are computed for all locomotives in the **for**-loop in line 8. The costs are sorted after increasing cost in line 17. In line 18, the list A contains the locomotives sorted after increasing connection cost. In the **repeat**-loop in line 19, we try to add trip p_{i_k} to a list end. We begin with the list end with the minimum connection cost and add p_{i_k} to it (line 22). The constraint system checks if the track allocation constraints are violated by this new trip (line 24). If yes, we remove the trip from this list end (line 25) and try the next best list end. The loop is terminated if we could add the trip or if we have tried all locomotives (line 28). Trips which can't be added at all are discarded and not scheduled (line 29). We continue the **for**-loop from line 6 until we have tried to add all trips. The output of the algorithm is a locomotive plan, given by the trip order lists L_j.

We will give an estimation of the worst-case run time of this algorithm. The time complexity for the sorting of the trips in line 5 is $O(n \log n)$. We have to perform the following tasks for every trip:

1. find predecessors for a trip, sorted after cost: $O(m \log m)$
2. check track and location constraints: $O(m^3 nt)$

The computation of the connection costs can be done in $O(m)$ time (line 8) and the sort operation for the predecessors has complexity $O(m \log m)$ (line 17).

We will now look a the complexity of the constraint checks. First, we check if the departure time of the added trip lies in its corresponding time window. This can be done in $O(1)$ time. If the time window is not violated, we continue with constraint checks for the track allocation problem.

We assume that a trip traverses an average number of t tracks. Assume that we have added the trip p to locomotive l_1. We must then check every traversed track in the added trip against collisions with track traversals of locomotive l_2. The same has to be done for the location constraints of $O(t)$ visited locations, i.e. we must check the maximum number of waiting trains. We go through the tracks of trip p in $O(t)$ time and check if there are collisions with l_2 [2]. The location constraints can be checked at the same time. Thus, the check between p and l_2 can be done in $O(t)$.

If there are collisions, we must shift all the departure times of the track traversals of the colliding trips on l_2 in the worst-case. We could also shift p but maybe this is not possible as its time window is too small. In the worst-case, not only the colliding track traversals but also all trips that follow afterwards on l_2 must be shifted. We get $O(nt)$ for the shifting in the worst-case. For all operations between l_1 and l_2, we get time $O(nt)$.

Trip p on l_1 and the $O(nt)$ shifted track traversals on l_2 must be checked against collisions with locomotive l_3. The number of tracks to be checked is $O(nt)$. In the worst case, $O(nt)$ track traversals on l_3 must be shifted. The check between l_1, l_2 and l_3 costs $O(nt)$.

This procedure is done $O(m)$ times in the worst-case until we have reached the locomotives $l_1, \ldots, l_{m-1}$ and l_m. Thus, the time for the constraint checks is $O(m^2nt)$ after the addition of a trip to a trip list.

A trip can be added m times in the worst case (**repeat**-loop in line 19), so that the total time for the constraint checks is $O(m^3nt)$.

For one iteration of the **for**-loop in line 6, we get time $O(m^3nt + m \log m)$. We have n iterations and must add the sort step in the beginning, so that the total time is $O(m^3n^2t + mn \log m + n \log n)$. We assume $t \in O(1)$ and $m \leq n$, because the number of locomotives is usually smaller than the number of trips. We finally get complexity $O(n^5)$.

[2] We assume that we have a data structure for the tracks that stores for every track references to all trips that traverse this track.

8.3 Nearest Neighbour Heuristic

Another heuristic uses the same cost function for a connection, but doesn't sort the trips beforehand. We choose instead the next trip as the trip which fits best to a certain list end, the nearest neighbour.

Nearest Neighbour Heuristic
input: trips $P = \{p_1, \ldots, p_n\}$
locomotives $M = \{l_1, \ldots, l_m\}$
output: trip order list $L_j = [p_{j_1}, \ldots, p_{j_\alpha}]$ with $\alpha = \alpha(j)$ for every locomotive l_j

```
01 begin
02   for j := 1 to m do
03     list L_j := nil;
04   endfor
05   repeat
06     integer t_j := current minimum route time of l_j;
07     if t_j = 0 then t_j := λ endif;
08     list M := [l_{j_1}, ..., l_{j_m}] := sort(M, t_j);
09     list N := [j_1, ..., j_m];
10     if l_{j_r}, ..., l_{j_s} are new locomotives
11       sort sublist [l_{j_r}, ..., l_{j_s}] after nearest neighbour in P;
12     endif;
13     repeat
14       integer j := first(N);
15       N := tail(N);
16       set C := ∅;
17       forall p_i ∈ P do
18         if L_j ≠ nil
19           then integer c_i := c(last(L_j), p_i);
20           else integer c_i := c(Σ(l_j), p_i);
21         endif
22         C := C ∪ {c_i};
23       endfor
24       list C' := [c_{j_1}, ..., c_{j_α}] := sort(C);
25       list A := [j_1, ..., j_α];
26       repeat
27         integer i := first(A);
28         A := tail(A);
29         L_j := append(L_j, p_i);
30         boolean successful := false;
31         if constraints violated
32           then L_j := L_j − last(L_j);
```

```
33              else P := P − {p_i}; successful := true;
34            endif
35          until successful or A = nil;
36        until successful or N = nil;
37        if N = nil then print P not added; stop; endif
38    until P = ∅;
39end.
```

We determine for all locomotives their current minimum route time t_j (line 6). If it is zero, we set it to a penalty $\lambda > 0$ in order to avoid the use of too many locomotives (line 7). We sort the locomotives after increasing route time (line 8), because we want to choose locomotives with few trips first in order to get equally long locomotive routes. N contains in line 9 the locomotive numbers, sorted after this order. If we have several new locomotives to choose from, we should take the locomotive which has the nearest neighbour in the trip set P. We sort the new locomotives after this criterion again in line 11.

We start with the locomotive l_j with minimum route time (line 14). If we can't add any trip to this locomotive, we must try the next best one. The **repeat**-loop in line 13 iterates over the locomotives.

We compute for all unscheduled trips in P the connection cost from Section 8.2 with the last trip in L_j (line 17) and sort them after increasing cost (line 24). The list A in line 25 contains the trips, sorted after increasing connection cost. In the following **repeat**-loop, we try to add a trip to the end of L_j. If no constraints are violated, we remove the trip from P (line 32). The loop is terminated if we could add a trip or there are no more trips to try.

If we couldn't add any trip at all, we try the next locomotive. If we can't add any trips to any locomotive, the algorithms terminates and prints out the remaining unscheduled trips.

The **repeat**-loop in line 5 has n iterations. We sort the locomotives after their travel time in time $O(m \log m)$ (line 8). The sort step for the new locomotives (line 11) takes time $O(mn + m \log m)$ in the worst-case: We must compute for m locomotives the nearest neighbour in P, this can be done in time $O(mn)$. The sort step takes $O(m \log m)$. We get a total time of $O(mn^2 + mn \log m)$ for the determination of the locomotive order.

The **repeat**-loop in line 13 has m iterations in the worst case. The computation of the connection cost in line 17 for the remaining trips can be done in time $O(n)$. They are sorted in time $O(n \log n)$ (line 24). We have a total time of $O(mn^2 \log n)$ for these operations.

The **repeat**-loop in line 26 has n iterations in the worst case. If we add a trip, we must do the necessary constraint checks. The complexity for this is $O(m^2nt)$ as in Section 8.2. We get complexity $O(m^3n^3t)$ for this loop.

Thus, the total time for the algorithm is $O(mn^2 + mn\log m + mn^2\log n + m^3n^3t)$. With the assumptions $t \in O(1)$ and $m \leq n$ we get complexity $O(n^6)$.

8.4 Insertion Heuristic

We can try to improve the best predecessor heuristic by not only considering the list ends for the addition of a trip, but also all insertion positions in the trip lists. We try to insert our current trip between two trips p_α and p_β in a trip list L_j, where the end location of p_α and the start location of p_β fits best to our current trip.

We give the insertion heuristic in pseudocode:

Insertion Heuristic
input: trips $P = \{p_1, \ldots, p_n\}$
locomotives $M = \{l_1, \ldots, l_m\}$
output: trip order list $L_j = [p_{j_1}, \ldots, p_{j_\alpha}]$ with $\alpha = \alpha(j)$ for every locomotive l_j

```
01 begin
02   for j := 1 to m do
03     list L_j := nil;
04   endfor
05   list P := [p_{i_1}, ..., p_{i_n}] := sort(P, τ_max);
06   for k := 1 to n do
07     set C := ∅;
08     forall insertion positions (p_α, p_β) in L_1, ..., L_m do
09       integer c_{j,α,β} := c(p_α, p_{i_k}, p_β);
10       C := C ∪ {c_{j,α,β}};
11     endfor
12     list C' := [c_{j_1,α_1,β_1}, ..., c_{j_γ,α_γ,β_γ}] := sort(C);
13     list A := [(j_1, α_1, β_1), ..., (j_γ, α_γ, β_γ)];
14     repeat
15       integer (j, α, β) := first(A);
16       A := tail(A);
17       L_j := insert(L_j, p_α, p_{i_k}, p_β);
18       boolean successful := false;
19       if constraints violated
20         then L_j := remove(L_j, p_{i_k});
```

21 **else** $successful := true$;
22 **endif**
23 **until** $successful$ **or** $A = nil$;
24 **if** $A = nil$ **then print** p_{i_k} not inserted; **endif**
25 **endfor**
26**end.**

We sort the trips into a list $P = [p_{i_1}, \ldots, p_{i_n}]$ after increasing latest departure time τ_{max} in line 5. We insert them in this order so that we build up the locomotive plan from left to right. This enables us to take into account the locomotive start positions for the first trips which are inserted and we insert always the most urgent trip. We try to avoid violations of departure time windows by inserting always the most urgent trip.

In the loop in line 6, we compute the insertion cost for the current trip p_{i_k} at all possible insertion positions in all current lists L_j. (p_α, p_β) denotes the position in between the trips p_α and p_β. The cost for inserting trip p_j between trip p_i and trip p_k is defined as

$$c(p_i, p_j, p_k) = \delta_2(E(p_i), S(p_j)) + \delta_2(E(p_j), S(p_k)) - \delta_2(E(p_i), S(p_k))$$

This is the additional amount of passive transport time after the insertion. In line 9, we store the corresponding locomotive l_j in the index j of $c_{j,\alpha,\beta}$ and the insertion position in α and β. Observe that $c(p_i, p_j, p_k)$ can also become negative if we can replace a passive transport by inserting trip p_j at this position.

We must also handle the cases when the trip is inserted in an empty list, or if it is inserted as the first trip or the last trip. These cases are not shown explicitely in the pseudocode in order to keep it simple. We add a penalty cost $\lambda > 0$ to the passive transport from the start location, if we use a new locomotive l_q with an empty trip list $L_q = nil$. This forces the heuristic to allocate a trip first to an already used locomotive and reduces the number of used locomotives:

$$c(l_q, p_j) = \delta_2(\Sigma(l_q), S(p_j)) + \lambda$$

If we insert a trip p_j as a first trip in a nonempty trip list L_q of a locomotive l_q with the current first trip p_k, we must consider the start location of the locomotive:

$$c(l_q, p_j, p_k) = \delta_2(\Sigma(l_q), S(p_j)) + \delta_2(E(p_j), S(p_k)) - \delta_2(\Sigma(l_q), S(p_k))$$

If we add a trip p_j after trip p_k at a list end, the cost is:

$$c(p_k, p_j) = \delta_2(E(p_k), S(p_j))$$

We sort the set C after increasing insertion cost in line 12. The list A contains the insertion positions which are represented by a triple (j, α, β), sorted after increasing cost. The **repeat**-loop in line 14 iterates over the insertion positions, we begin with the best insertion position. We insert trip p_{i_k} (line 17) and check if any constraints are violated. We repeat the insertions until an insertion was successful. If trip p_{i_k} can't be inserted at all, it is discarded and not scheduled (line 24).

We compute also for this algorithm the worst-case run time. The **for**-loop in line 6 has n iterations. The computation of the insertion cost for all insertion positions takes time $O(m+n)$, because we have $O(m+n)$ possible insertion positions. Thus, the complexity of the **forall**-loop in line 8 is $O(m+n)$. The insertion costs are sorted in time $O((m+n)\log(m+n))$ (line 12).

The **repeat**-loop in line 14 has $O(m+n)$ iterations in the worst-case, i.e. we must try all possible insertion positions until we can insert trip p_{i_k}.

The constraint checks after an insertion can be done in time $O(m^2nt)$: assume that we insert trip p_{i_k} on locomotive l_1. In the worst-case, the departure times of $O(nt)$ following trips must be shifted. This can be done in time $O(nt)$. This is different to the case where we add a trip at the end of a trip list, because we have no shifts in this case. The shifted trips must be checked against collisions with trips on locomotive l_2, this can be done in time $O(nt)$. We use the same argument as in Section 8.2 where we had to do the check for $O(nt)$ track traversals between l_1, l_2 and l_3. The only difference when we insert a trip is that we have already to perform $O(nt)$ operations between l_1 and l_2. We have to repeat this until we compare $l_1, \ldots, l_{m-1}$ and l_m, so that the total time for the constraint checks is $O(m^2nt)$.

Thus, the total time for the **repeat**-loop in line 14 is $O(m^2nt(m+n))$. We get for the total time $O(n(m+n)\log(m+n) + m^2n^2t(m+n))$. With the assumptions $t \in O(1)$ and $m \leq n$, we get $O(n^5)$.

8.5 Examples

We will compare the three route building heuristics by means of three examples. We will concentrate on the amount of passive transport time and choose our examples so that the locomotive number is fixed, because the penalty cost mechanism is the same for all three heuristics. We set the penalty parameter for the use of a new locomotive to zero: $\lambda = 0$.

Assume that we have two locomotives l_1, l_2 which have the start positions $\Sigma(l_1) = \Sigma(l_2) = A$ (see Fig. 8.1). We want to compute a locomotive plan for four trips $p_1 - p_4$, the departure time windows and estimated durations are given in Table 8.1.

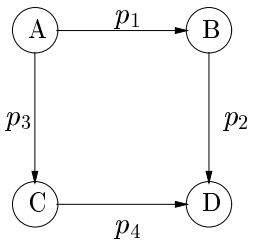

Figure 8.1: Example 1

trip	departure time window	duration
p_1	$[0, 4]$	1
p_2	$[0, 2]$	1
p_3	$[0, 4]$	1
p_4	$[0, 2]$	1

Table 8.1: Data for Example 1

Observe that we need two locomotives because one locomotive can't perform all trips in their respective time windows.

We begin the with the best predecessor heuristic. The trips are sorted after their last departure time and added to the plan in the order p_2, p_4, p_1, p_3 (without loss of generality). To simplify the discussion, we set $\alpha_1 = 1$ and $\alpha_2 = 0$ in our connection cost measure from Section 8.2, i.e. we don't consider waiting times of the locomotives. If we always choose the best connection, we can get the following trip lists:

$$L_1 = [p_2, p_1]$$

$$L_2 = [p_4, p_3]$$

This means that every locomotive has two passive transports, before and after its first trip. This is obviously not the optimal solution.

The nearest neighbour heuristic finds the optimal solution. We choose a nearest neighbour to the start position of locomotive l_1, for example p_1. The nearest neighbour for l_2 is p_3. We finally get:

$$L_1 = [p_1, p_2]$$

$$L_2 = [p_3, p_4]$$

The insertion heuristic sorts the trips after their latest departure time, we get the order p_2, p_4, p_1, p_3. p_2 is added to l_1, then p_4 to l_2. p_1 and p_3

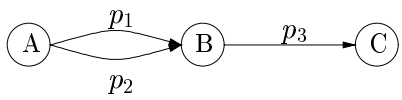

Figure 8.2: Example 2

are then inserted at the beginning of the trip lists. We finally get the same solution as with the nearest neighbour heuristic.

This example shows a deficiency of the best predecessor heuristic. We can only add trips at the list end. We must sort the trips after some criterion, it seems reasonable to sort them after their departure times. Trips with large time windows (p_1 and p_3) can't be inserted at the beginning of the trip list, because we add them late to the plan. This leads to a solution with many passive transports.

The second example shows a difference between the nearest neighbour and the insertion heuristic (Fig. 8.2). The trip durations are given in Table 8.2.

trip	duration
p_1	1
p_2	1
p_3	2

Table 8.2: Data for Example 2

We have now one locomotive l_1 with start location A. Assume that the departure time windows are quite large, i.e. they don't play an important role in this example. The nearest neighbour heuristic constructs for example the trip list $L_1 = [p_1, p_3, p_2]$, i.e. the locomotive must perform a passive transport between p_3 and p_2 of duration 3 from C to A.

Independent of the order in which the trips are inserted, the insertion heuristic finds the optimum. Assume that we have the order p_1, p_2, p_3. After p_1 is inserted, p_2 can be inserted before or after p_1. We insert p_2 at the beginning of the list. p_3 is finally added at the end so that we get the optimum $L_1 = [p_1, p_2, p_3]$ with a passive transport of duration 1 from B to A.

There are also examples where both the insertion and the nearest neighbour heuristic fail. In Figure 8.3, we have five trips with the data from Table 8.3.

We have two locomotives with the start positions $\Sigma(l_1) = A$ and $\Sigma(l_2) = C$. Two locomotives are needed in order to meet all time window restrictions.

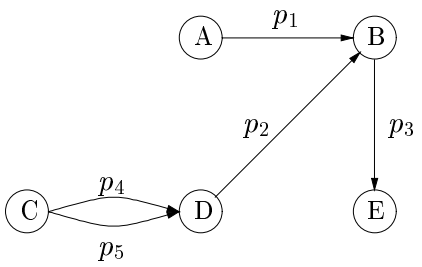

Figure 8.3: Example 3

trip	departure time window	duration
p_1	$[0, 2]$	1
p_2	$[2, 6]$	1
p_3	$[0, 7]$	6
p_4	$[0, 8]$	2
p_5	$[0, 9]$	2

Table 8.3: Data for Example 3

The optimum solution is obviously

$$L_1 = [p_1, p_3]$$

$$L_2 = [p_4, p_5, p_2]$$

with a passive transport of duration 2 between p_4 and p_5.

The insertion heuristic sorts the trips after their latest departure time, we get the order p_1, p_2, p_3, p_4, p_5. p_1 is inserted on l_1 because $\Sigma(l_1) = A$. p_2 is inserted after p_1 on l_1 because l_2 is in C, a passive transport of duration 2 would be necessary. On l_1, we need only a passive transport of duration 1. p_3 is inserted after p_2 and we get for l_1:

$$L_1 = [p_1, p_2, p_3]$$

p_4 is inserted in L_2 and p_5 after p_4. We get for l_2:

$$L_2 = [p_4, p_5]$$

This is not the optimum as we have passive transports between p_1, p_2 and p_4, p_5 of total duration 3. The same solution is obtained with the best predecessor heuristic.

The nearest neighbour heuristic starts for example with l_1 and assigns p_1 to it. Then, we assign p_4 to l_2 because l_2 is the locomotive with minimum route

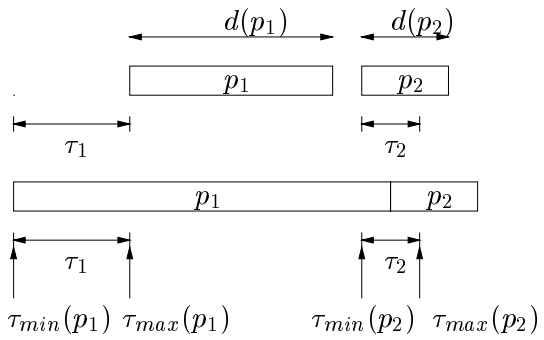

Figure 8.4: Time windows with $\tau_1 \geq \tau_2$

time. After that, l_1 is the locomotive with the smaller total route duration and we assign p_3 to it. We get for l_1:

$$L_1 = [p_1, p_3]$$

As l_1 has now a quite large total route time, we assign the remaining trips to l_2. We add first p_2 and then p_5. The result is:

$$L_2 = [p_4, p_2, p_5]$$

This solution is also not the optimum as we get a passive transport of duration 3 between the trips p_2 and p_5.

We can see that there are examples where all three heuristics fail. A discussion of the choice of one heuristic will follow in Section 8.9.

8.6 Matching Heuristic

As a preprocessing step to the route building algorithms above, we can try to build blocks of trips with good connections which can be used instead of single trips in our route building algorithms. This reduces the amount of trips n by a constant factor α with $0 < \alpha < 1$.

We show in Fig. 8.4 and 8.5, that the time window of a block is in general smaller than the smaller time window of the individual trips. Let $[\tau_{min}(p_i), \tau_{max}(p_i)]$ denote the time window of trip p_i and $\tau_i = \tau_{max}(p_i) - \tau_{min}(p_i)$ its window size.

We consider blocks of two trips p_1, p_2 and begin with the case $\tau_1 \geq \tau_2$. We look in this and the following cases always at worst-case departure times [3] of

[3] i.e. the departure times that lead to a minimum window size of the block.

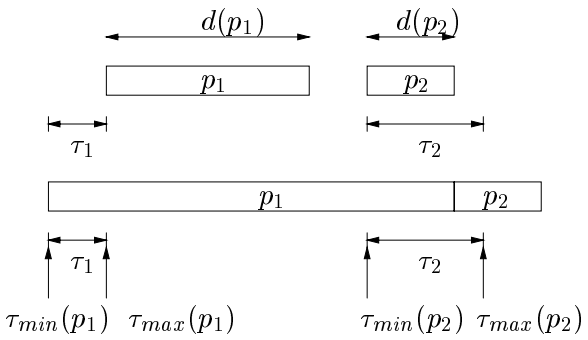

Figure 8.5: Time windows with $\tau_1 < \tau_2$

the trips because we don't know the departure times beforehand. In the first figure in Fig. 8.4, we have $\tau_{max}(p_1) + d(p_1) \leq \tau_{min}(p_2)$, this is the only case where the window size does not decrease because the window size of the block $[p_1, p_2]$ is $\tau = \tau_1$. In the second figure, when $\tau_{min}(p_1) + d(p_1) > \tau_{min}(p_2)$, the block window becomes smaller, its size is $\tau = \tau_{max}(p_2) - (\tau_{min}(p_1) + d(p_1)) < \tau_2$, because the flexibility of p_2 becomes smaller. Between these two extreme cases lie cases where the flexibility of p_2 is reduced only for a subset of p_1s departure times.

In Fig. 8.5, the case $\tau_1 < \tau_2$ is shown. In the upper figure, we have $\tau_{max}(p_1) + d(p_1) \leq \tau_{min}(p_2)$, the block window size is $\tau = \tau_1$. In the lower figure, $\tau_{min}(p_1) + d(p_1) > \tau_{min}(p_2)$ and the block window size is $\tau = \min\{\tau_1, \tau_{max}(p_2) - \tau_{min}(p_1) + d(p_1)\}$. Thus, τ can even become smaller than τ_1. As in Fig. 8.4, there are intermediate cases between these two extreme cases.

In the worst-case, the window size decreases with every combination of individual trips. This restricts the block size, because blocks with small windows are inflexible and are not so useful for our route building algorithms.

We suggest a matching heuristic which builds iteratively blocks of increasing size. Let P denote the set of trips. We build an undirected graph $G = (P, E)$. We add an edge $\{p_i, p_j\}$ to E, if $E(p_i) = S(p_j)$ or $E(p_j) = S(p_i)$ and if the trips p_i and p_j can be combined into a block. This depends on their time windows and durations. We can build the block $[p_i, p_j]$ if $\tau_{min}(p_i) + d(p_i) \leq \tau_{max}(p_j)$. We set also a threshold value $\epsilon > 0$ for the worst-case waiting time of the locomotive between the two trips, because we don't want to combine trips which lie far apart in time. If both orders $[p_i, p_j]$ and $[p_j, p_i]$ are possible, we choose the one with lower waiting time. The construction of this graph can be done in $O(n^2)$ time.

We can now compute a maximum cardinality matching with the algorithm from [MV80] in time $O(n^{2.5})$. It finds a maximum number of edges in G

which have not any nodes in common. We remove the blocks from P where the new time window size lies under a threshold $\delta > 0$. The remaining blocks and single trips are now regarded as new single trips with a new time window which can be computed in time $O(n)$.

In the next iteration, we repeat our graph construction in time $O(n^2)$. The procedure is repeated as long as there are blocks with appropriate time window size. If we have k iterations, the total time for the algorithm is $O(kn^{2.5})$.

8.7 Improvement Heuristic

After we have built locomotive routes with one of the heuristics above, we can try to improve the routes by a post-optimization procedure. We look at every trip which is preceeded or followed by a passive transport. These trips can possibly be inserted at a better position, an insertion of the other trips is not reasonable as this does not improve the amount of passive transport. We can always consider the insertion of the last trip in a route as this doesn't generate any new passive transport in the old route. Another idea is to insert trips on locomotives with few trips on locomotives with more trips so that the number of locomotives can be reduced.

Improvement Heuristic
input: trip order lists L_j
output: trip order lists L_j

01 **begin**
02 **list** $P := [p_1, \ldots, p_n]$;
03 **for** $i := 1$ **to** n **do**
04 $p_\gamma :=$ predecessor of p_j;
05 $p_\delta :=$ successor of p_j;
06 **set** $C := \emptyset$;
07 **forall** insertion positions (p_α, p_β) in $L_1, \ldots, L_m$ **do**
08 **integer** $c_{j,\alpha,\beta} := c(p_\gamma, p_i, p_\delta, p_\alpha, p_\beta)$;
09 **if** $c_{j,\alpha,\beta} > 0$ **then** $C := C \cup \{c_{j,\alpha,\beta}\}$;
10 **endfor**
11 **list** $C' := [c_{j_1,\alpha_1,\beta_1}, \ldots, c_{j_\epsilon,\alpha_\epsilon,\beta_\epsilon}] := sort(C)$;
12 **list** $A := [(j_1, \alpha_1, \beta_1), \ldots, (j_\epsilon, \alpha_\epsilon, \beta_\epsilon)]$;
13 **repeat**
14 **integer** $(j, \alpha, \beta) := first(A)$;
15 $A := tail(A)$;
16 remove p_i from old list;
17 $L_j := insert(L_j, p_i, p_\alpha, p_\beta)$;

```
18        boolean successful := false;
19        if constraints violated
20          then L_j := remove(L_j, p_i);
21          else successful := true;
22        endif
23      until successful or A = nil;
24    endfor
25end.
```

We look a each trip separately (line 3) and compute for every possible insertion position the benefit from this insertion. Let p_i denote the trip which we want to insert somewhere else, p_γ its predecessor and p_δ its successor at the current position. p_α is the predecessor at the insertion position, p_β the successor. The benefit of the insertion is (observe that we look now at a benefit and not a cost):

$$\begin{aligned} c(p_\gamma, p_i, p_\delta, p_\alpha, p_\beta) &= \delta_2(E(p_\alpha), S(p_\beta)) + \delta_2(E(p_\gamma), S(p_i) + \delta_2(E(p_i), S(p_\delta)) \\ &- \delta_2((E(p_\alpha), S(p_i)) + \delta_2(E(p_i), S(p_\beta))) - \delta_2(E(p_\gamma), S(p_\delta)) \end{aligned}$$

This definition can easily be adapted for the first and last positions in a list.

This benefit is computed for every potential insertion position in $O(n+m)$ time (line 7). All insertion positions with $c > 0$ give a benefit. We sort them in $O((n+m)\log(n+m))$ time after decreasing benefit (line 11) and try one insertion position after the other until no constraints are violated (**repeat**-loop in line 13). The time for the constraint checks is $O(m^2nt)$. This is done for $O(n+m)$ insertion positions in the worst-case. For one iteration of the **for**-loop in line 3, we get the time $O((n+m)(\log(n+m) + m^2nt))$.

We repeat this for n trips, so that the total run time is $O(n(n+m)(\log(n+m) + m^2nt))$ or $O(n^5)$.

If we iterate this algorithm as long as we can improve our solution (hill-climbing), we get a local search procedure. We start from an initial solution which was constructed by a route building algorithm and improve it until we reach a local optimum. Local search techniques have been successfully applied to traveling salesman and vehicle routing problems [RSL77], [ABBG83], [PR95], [Sav85]. This approach seems also to be reasonable for the locomotive assignment problem.

8.8 Locomotive Types and Passive Transports

We can include different locomotive types into our heuristics if we assign the trips only to locomotives with suitable types. This can be easily checked in the three route building heuristics and the improvement heuristic. The matching heuristic must be extended so that only trips with compatible locomotive types $l(p_i) \cap l(p_j) \neq \emptyset$ are connected by an edge (compare Def. 4). We can then do one matching step and build pairs of trips. If we want to combine two trip pairs (p_i, p_j) and (p_k, p_l) into one block, we must check if $(l(p_i) \cap l(p_j)) \cap (l(p_k) \cap l(p_l)) \neq \emptyset$ before we add an edge. A similar check must be done for the following iterations.

We haven't adressed the track allocation problem for the passive transports yet. As soon as we generate passive transports in our route construction procedure, we should add them to the track allocation problem as additional constraints in order to get early failures if there are no tracks for passive transports available. We can add passive transports immediately in the best predecessor and nearest neighbour heuristic because all generated passive transports are also in the final locomotive plan. In the insertion heuristic, passive transports can be replaced by trips and we would formulate too hard constraints if we would add the passive transport contraints immediately. Thus, we can only add them after the locomotive plan is completed. This is a disadvantage of the insertion heuristic.

After we have determined the locomotive plans and the trip order for every locomotive, we must determine the departure times of the trips at a detailed level. We will show in Chapter 9 how this can be done in the TUFF system.

8.9 Discussion

The examples suggest that we should choose the nearest neighbour or the insertion strategy for the construction of locomotive routes. The run times are similar. Experiments for the VRSPTW with similar heuristics in [Sol87] suggest also that both heuristics can construct near-optimum solutions and behave relatively robust.

When it comes to implementation, the nearest neighbour heuristic is difficult to implement in Oz because the search procedure in Oz requires a definite variable order on the current path in the search tree. In route construction, these variables are trip departure times. In the nearest neighbour heuristic, this order is not fixed as we postpone the addition of a trip if the constraints gave a failure. In the insertion heuristic, the trips are ordered after their departure times and we have a definite variable ordering. Thus, this heuristic is better suited for an implementation in Oz.

As we have mentioned before have local search techniques successfully been applied to routing problems. Local search methods are difficult to implement in constraint programming languages, because it is not possible to modify FD-variables arbitrarily. This would be necessary in order to insert trips at new positions. FD-variables can only be narrowed during search. Thus, the improvement heuristic is difficult to implement in Oz.

The probably easiest way to implement the matching heuristic and the matching algorithm on graphs is to use a conventional, imperative programming language. The output of such a program could be used as a preprocessed input for the TUFF system.

Chapter 9

Implementation

We describe in this chapter the implementation part of this work. We begin with a short description of the implementation of the TUFF system. Then we explain the new parts that have been added. The exclusion marker model from Chapter 6 has been implemented in Oz and the insertion heuristic was chosen from the heuristics in Chapter 8 as a search heuristic for the construction of locomotive routes. We conclude the chapter with a description of the `diff2`-propagator implementation in C++.

9.1 The TUFF system

The TUFF system was already presented briefly in Section 1.3. We give here a short overview over its implementation.

The input to a planning problem consists of a network file and a train file. The network file contains all the network data from Def. 1 in Section 3.1. For all locations $v \in V_1$, a maximum number of trains of waiting trains $\sigma(v)$ and location specific waiting times are defined. All tracks are given with their length, their maximum velocity and the attribute single or double track. The network file contains also the definition of the routes between the location pairs.

The train file contains the trips for a scheduling problem. Every trip p_i has the following attributes:

- Start location $S(p_i)$ and end location $E(p_i)$.
- Maximum speed $\nu(p_i)$.
- Departure time window $[\tau_{min}(p_i), \tau_{max}(p_i)]$.

One can additionally specify waiting times at intermediate locations in a route ($[w_{min}(p_i, v), w_{max}(p_i, v)]$ in Def. 2), but we don't use this feature in our experiments. The additional input for a planning problem is:

- Time window for the whole plan $[s_{min}, s_{max}]$
- Location slack factor μ

We have explained these parameters in Def. 2. The TUFF system computes with these inputs a timetable for all trains in the train file. The timetable can be visualized by train-sequence diagrams (Fig. 1.3 in Chapter 1).

The conditions for a timetable from Def. 3 are implemented as Oz finite domain constraints. The most important constraints are (Def. 3):

1. precedence constraints for the traversal of the tracks in a trip.
2. track constraints
3. location constraints

The precedence constraints are part of the trip constraints in Def. 3. The precedence and track constraints are implemented by constraints for arithmetic relations and correspond exactly to the formulation in Def. 3. The location constraints are implemented with the `cumulative`-constraint [HMSW98], a special constraint which has been developed for scheduling applications. It states that for all time instances, the resource usage does not exceed the available capacity. In our problem, the resources are locations and these have a maximum capacity of waiting trains. The other constraints of Def. 3, e.g. the time window $[s_{min}, s_{max}]$ for all departure times are represented by Oz FD-variables with corresponding domains.

The system uses a simple search heuristic for the determination of the timetable:

```
{FD.distribute generic(order:nbSusps value:splitMin) Deps}
```

`Deps` is the vector of all task departure times, i.e. track traversals. These variables are ordered after their number of suspensions, i.e. the number of constraints they take part in. Variables which are highly constrained are narrowed first during search in order to get early failures. In every choice point of the search tree, the search strategy branches into the two alternatives $x \leq m$ and $x > m$, where m denotes the middle value of the domain of x. The alternative where x is narrowed to the lower half of its domain is tried first. This leads to minimum departure times for the tasks.

If the planning problem has no solution, the constraint solving mechanism does not give us hints about the conflicting trips. The output of the system is just "no solution". The user must identify the conflicting trips manually. This is a considerable drawback of the system.

For the locomotive planning problem, the following input parameters have been added:

- Locomotives $M = \{l_1, \ldots, l_m\}$ with their start positions $\Sigma(l_i)$

It is possible to specify locomotive types in the train files but these are not used in our implementation (compare Section 9.3). The locomotive plans can be visualized in Gantt diagrams (see Chapter 10 for examples).

9.2 Exclusion Marker Model

The Oz code for the exclusion marker model can be found in Appendix B.1. We will point to the most important parts.

We post one `diff2`-constraint for all trip rectangles (line 29). Then, we go through all turn locations in our problem (line 36). For one turn location, we go through all trips (line 44) and add the markers which belong to the corresponding location `diff2`-constraint. We distinguish between the start location (line 49), the end location (line 64) and any other location (line 83). Depending on the location, we add start, end, after and before markers. After that, we add the markers for the start positions of the locomotives (line 106) if the current location is not a start location (line 110). We post the location `diff2`-constraint in line 126.

9.3 Insertion Heuristic

For the construction of suitable locomotive routes, the insertion heuristic from Chapter 8 was implemented as an Oz search mechanism, a so-called distribution. The code for this distribution can be found in Appendix B.2. The distribution mechanism in Oz can be specified by the following procedure [HMSW98]:

```
{FD.distribute generic(order: filter: value:)}
```

The feature `order` is a function which determines the variable order during search. We order the trips after their latest departure time (lines 9-11). Observe that the variable order is recomputed in every choice point, not

just once in the beginning. The latest departure times of the trips can change during search.

`filter` is a boolean function which defines the subset of variables which shall be distributed. We choose only variables where the resource (locomotive) hasn't been determined yet (domain size greater one, line 19). These are the unrouted trips.

`value` is the main procedure which determines the variable value alternatives in the choice point. We compute first the insertion cost for all insertion positions and then, we try them sequentially (line 143). The procedure `FindPos` computes the insertion cost for all insertion positions (line 27). The cost function from Section 8.4 was implemented, i.e. the insertion cost is the additional passive transport plus a penalty λ if a new locomotive is used.

We have tried to include the additional waiting time of the locomotive that occurs when the new trip is inserted into the cost function. It turned out that the weight for this cost factor must be much larger than the weight for the additional passive transport in order to give an effect on waiting times. The cost factor passive transports is more important than the waiting times so that we returned to the old cost function.

The insertion positions are sorted after increasing insertion cost (line 67). The procedure `TryPos` (line 73) inserts the current trip sequentially at these insertion positions until an insertion is successful. First, we assign the trip to its locomotive (line 80). We insert trip p_j by posting constraints on its departure time. p_i denotes the predecessor trip, p_k the successor (compare Def. 4):

$$s(p_i) + d(p_i) + d(p_i, p_j) + \Theta \leq s(p_j)$$

$$s(p_j) + d(p_j) + d(p_j, p_k) + \Theta \leq s(p_k)$$

These constraints are posted in lines 86 and 98. The turn time Θ does not occur in the code because it is included as a last waiting time into the trip durations $d(p_i)$ respectively $d(p_j)$. If there is no predecessor trip, the locomotive start location must be taken into account (line 90).

Observe that the trip durations $d(p_i)$ and $d(p_j)$ are FD-variables and are not determined during search, because they include waiting times at locations which are only known for a completely fixed timetable. The best we can do is to approximate these durations by their current upper bounds (the upper bounds of the corresponding FD-variables) so that we can be sure that the following trips do not start too early. As the trip durations vary only within a few percent, this is only a small mistake. But we loose of course solutions by this approximation.

By posting constraints on the departure time of the inserted trip, we do actually work that the exclusion marker model could do. The gaps for the passive transports could also be obtained by propagation in the `diff2`-constraint, but at a much higher cost. Thus, we post these constraints during search as additional constraints in order to reduce the variable domains early.

If the insertion was successful, we continue with the remaining unrouted trips (line 124). Otherwise, we try the next best insertion position (line 129). If all insertions have failed, we discard the trip and this is reported to the user (line 132). Finally, we determine the timetable by the search procedure from Section 9.1 (line 150).

If all trips could be inserted, we have now determined the complete timetable. The locomotive plan can be visualized as a Gantt diagram and the timetable can be inspected through train sequence diagrams.

If some trips where discarded because it was not possible to insert them, we must determine the timetable in a second run. As the track allocation constraints are built up for all trains which are contained in the train file, the not inserted trips are still represented as constraints. This means that we can't compute a solution in only one run. After the first run, the user must remove the trips which could not be inserted from the train file. The schedule can then be determined in a second run.

We have not implemented a differentiation between locomotive types, all trips can be handled by any locomotive. It shouldn't be difficult to extend the system in this way. As we have mentioned in Section 6.1, the initial domain for the FD-variable which represents the resource of a trip must be chosen appropriately. The search heuristic could be extended in a way that takes into account how specific the locomotive type requirement of certain trips is. Trips with very specific requirements could be inserted first in order to avoid later failures.

More important is the fact that we do not check the track constraints for the passive transports. We can't be sure that the passive transports can be performed, i.e. if the necessary tracks are available. We solve the track allocation problem only for the normal transports.

There are different alternatives to solve this problem. One can add additional constraints for the track usage of the passive transports after the locomotive plan has been determined. New FD-variables for the departure and waiting times of these passive transports must be introduced. This leads to solutions if the track capacity is not at its limit, but it doesn't help us if we can only detect that the passive transports are not possible.

A better approach is possibly to take the generated locomotive plan as an input to a second planning step. We fix the trip order for every locomotive

and add the generated passive transports to our trip set. For this extended trip set, we try to solve the track allocation problem with the old TUFF system. By fixing only the trip order for a locomotive, we give the trips their largest possible flexibility within their time windows. If the track allocation problem has no solution, we have do identify the conflicting trips and to change the locomotive plan.

9.4 Diff2 Propagator

The `diff2`-propagation algorithm from Chapter 7 was implemented as an Oz propagator according to the conventions of the Oz Constraint Propagator Interface (CPI) [MW97]. This interface allows the extension of the Oz language by additional constraint propagators which are implemented in C++. The C++-Code for the two main classes `Diff2Prop` and `DomainSet` of the propagator can be found in Appendix B.3.

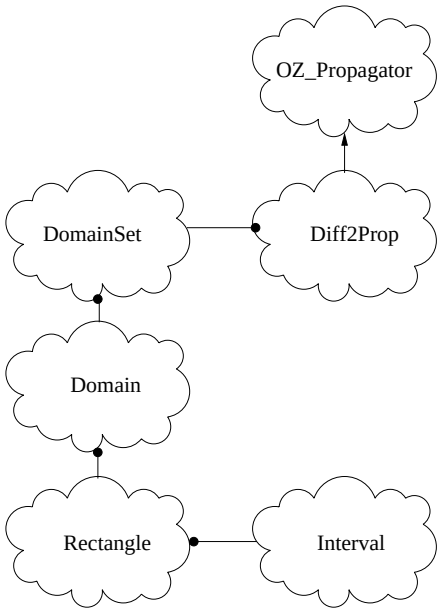

Figure 9.1: The most important classes of the `diff2`-propagator

Fig. 9.1 shows the most important C++-classes of the `diff2`- implementation. Arrows indicate inheritance relations, the other relations are has-a relations (class A has B as a member if the dot lies on As side).

`Diff2Prop` is the main propagator class and inherits from `OZ_Propagator`, a CPI class. It contains the method `Diff2Prop::propagate()` which starts

the AC3 propagation and the heuristic for stronger consistency from Section 7.7. The method `Diff2Prop::localPropagation()` contains the AC3–algorithm. In the first invocation of the propagator, the propagation is done for all overlapping rectangle pairs until the AC3–algorithm stops. In later invocations during search are only subsets of the rectangles considered, depending on which rectangles were changed. `Diff2Prop::globalPropagation()` contains the heuristic for rectangle area sums.

The part domains of each rectangle are stored in an instance of the class `DomainSet`. It contains a list of rectangular domains which are instances of the class `Domain`. Each `Domain` contains the domain itself, the corresponding kernel and an enumeration type indicating the type of the kernel. The class `DomainSet` contains also the non-overlappable area $\mathcal{O}$ and the non-coverable area $\mathcal{C}$ of a rectangle. They are updated by the method `DomainSet::updateForbiddenAreas` when the part domains have changed. This method contains all the algorithms from Section 7.4 for the computation of $\mathcal{O}$ and $\mathcal{C}$. The kernel computation for a rectangular domain from Def. 18 in Chapter 7 is done in the method `DomainSet::getKernel`.

For the domains we need two geometric base classes, the classes `Rectangle` and `Interval`. A rectangle is stored as a 4-tuple (rectangle sides). The class contains methods for inclusion, intersection, equality tests between rectangles and for the projections on the axes. `Interval` is the onedimensional analog of this class.

We have not implemented geometric data structures for the storage of the areas in which the rectangles can move, the domain sets. If we store for every domain set a bounding box in a geometric data structure, this would speed up the search after overlapping rectangle domains for a changed rectangle domain. We have found two data structures in the literature for the storage of rectangle sets, MX-CIF quadtrees [Sam89] and 4-D-trees [Ros85]. Both are static tree structures and make the insertion and deletion of rectangles difficult, the trees can become unbalanced [HNP+97]. Our rectangle domains must be updated constantly.

Another alternative is the grid file [OW93]. Grid files for points partition the plane into grid cells and store the points of one grid cell in a list. The search effort for a region query can be reduced by a constant factor with this technique, because only the lists of the cells which lie within the region must be scanned. Insertion and deletions of points can be easily done, although some effort must be spent on the splitting and merging of grid cells in order to keep the number of grid points per cell in a certain range. If we want to store rectangles which heavily overlap, it is difficult to find a mesh size where the rectangles can be separated into different grid cells, they will all lie on grid cell boundaries. We can extend the 2D gridfile to a 4D gridfile, because a rectangle can be represented by a 4-tuple (l, r, b, t) for the rectangle sides.

The memory requirement for a 4D gridfile can be enormous. If we partition our space into m intervals in each dimension, we get m^4 grid cells. For this reason, we haven't implemented this data structure either.

Chapter 10

Experiments

This chapter contains experiments with problem sets on the Swedish railway network. We begin with performance measurements with and without the exclusion marker model. After that, we look at out first example from Chapter 3 again. We conclude the chapter with a larger example.

10.1 Performance

All performance measurements are based on two differents train sets A and B where we have varied different parameters. The locomotive start locations and train files can be found in Appendix A.1 and A.2. Whenever we use fewer locomotives or trains in our experiments, we use the corresponding prefix of the locomotive and train set. Both examples where obtained by distributing manually the trips and locomotive start positions over the network. Train set A contains trains between 14 locations, the corresponding part of the network is shown in Fig. 10.1. Thin lines indicate single tracks and thick lines represent double tracks. The abbreviations for the city names are explained in Table 10.1.

Ch	Charlottenberg	No	Norrköping
Fa	Falköping	Sk	Skövde
Gö	Göteborg	St	Stockholm
Ha	Hallsberg	Tr	Trollhättan
Ka	Katrineholm, Karlstad	Up	Uppsala
Mj	Mjölby	Vä	Västerås
Nä	Nässjö		

Table 10.1: City names for Train Set A

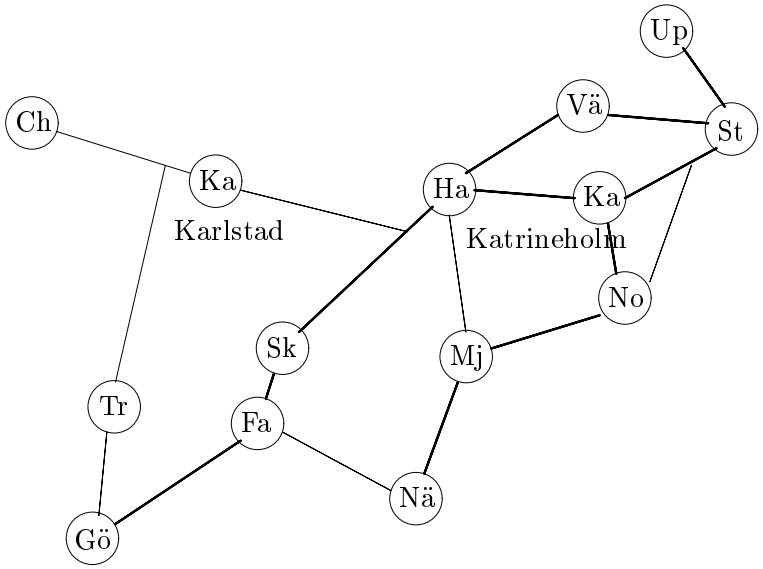

Figure 10.1: Network for Train Set A

Train set B contains long-distance trains between 11 locations, the network is shown in Fig. 10.2. A link between two cities is shown as a single track link if the majority of the tracks on this link are single tracks, otherwise it is a double track link. Table 10.2 contains the corresponding city names.

Gö	Göteborg	Ma	Malmö
Ha	Hallsberg	Sk	Skövde
He	Helsingborg	St	Stockholm
Ka	Karlstad	Um	Umeå
Lu	Lund, Luleå	Ös	Östersund

Table 10.2: City names for Train Set B

All performance measurements where done on a 248 MHz Sun Ultra Enterprise with 1 GB memory. We used Oz 2.0.4 and the C++-Compiler gcc 2.7.2.3.

The run times were measured with the tool Oz Panel. It provides a measurement of the run time, divided into different components like propagation, garbage collection of the Oz system etc. The sum of all these components was used as the run time. The memory consumption of the complete TUFF system was measured with the Unix program top.

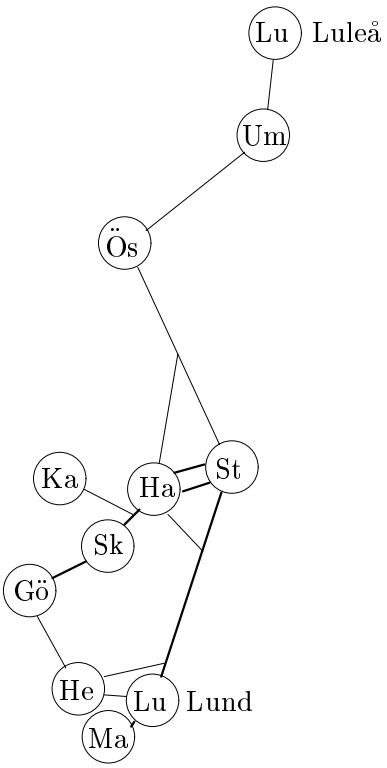

Figure 10.2: Network for Train Set B

10.1.1 Performance of the Exclusion Marker Model

The performance of the exclusion marker model including the `diff2`-constraint was tested on a problem including the first 50 trains from train set A. The departure time specifications from the train file in Appendix A.1 where ignored so that all trains have as departure time window the total schedule period.

The train set was stepwise reduced by 10 trains for the performance measurements. Table 10.3 shows the problem parameters. For the penalty parameter λ (see Section 8.4), we use the mean duration of a passive transport $\overline{\delta_2}$ which is estimated by the mean duration of the trips in the problem [1]. This parameter setting reduces the number of used locomotives quite well. We have chosen a location slack factor of $\mu = 0.05$, i.e. the slack of a trip is at most 5% of the whole trip duration. This is a reasonable upper bound for the slack in practical problems.

[1] The trip durations where calculated from the distance in the network and the passive transport velocity.

number of locations	14
total schedule time (hours)	24
train velocity (km/h)	120
velocity passive transports (km/h)	120
λ (min)	$\overline{\delta_2}$=110
location slack factor μ	0.05
departure time windows (hours)	24
turn time Θ (mins)	30

Table 10.3: Parameters for performance measurement

The results are shown in Table 10.4. The run times are the average of three runs.

number of trips n	10	20	30	40	50
number of locomotives m	3	6	9	12	15
number of used locomotives	3	6	6	10	12
time in s	1.9	7.7	21.5	34.7	69.8
memory in MB	30	47	150	255	352

Table 10.4: Results

The number of locomotives m was adapted to the number of trips n so that the ratio m/n is constant. The number of actually used locomotives is indicated in the third line.

As can be seen in Fig. 10.3, the run times seem to have an exponential behaviour. The run time for 60 trains could not be determined due to the high memory consumption. The high memory consumption is another problem, thus we can only handle small problems.

The run times can be explained by the inefficiency of the `diff2`–implementation. Tests have shown that the number of inspected arcs in the AC3–algorithm grows enormously with the number of rectangles in the constraint. As we have mentioned in Section 7.5, the number of arc inspections is related to the maximum number d of rectangle positions in the worst-case. d can be quite large in the marker model, its size is $d = kmt$ where k is the parameter in the marker model from Section 6.1, m denotes the number of locomotives and t is the size of the whole schedule period. t can make d quite large.

The memory consumption can be explained by the storage requirement for the part domains of the rectangles [2]. As the number of rectangles in the con-

[2] in addition to the memory requirement for the TUFF system, this will be explained in Section 10.1.2.

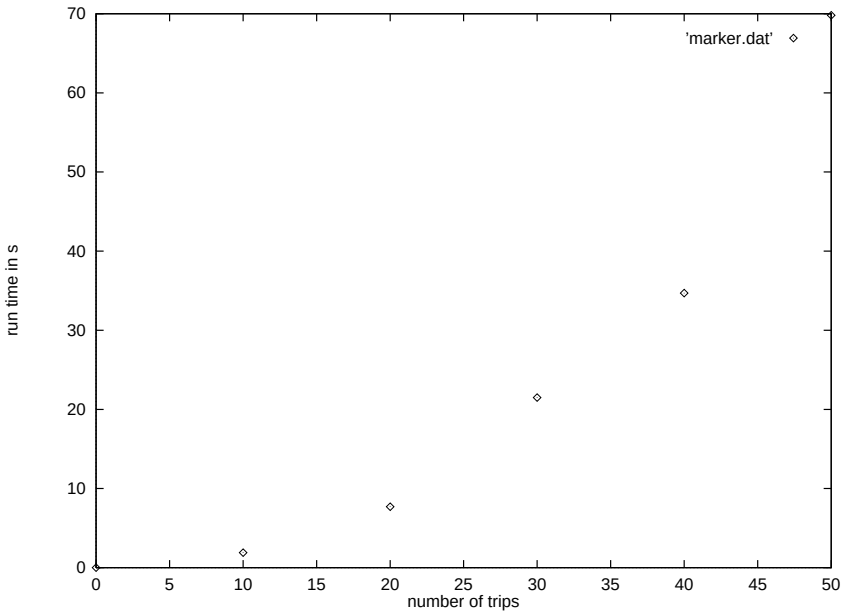

Figure 10.3: Performance with Marker Model

straint grows, we have to store more rectangle domains and these domains fall into more part domains during the propagation algorithm.

A remedy could be to limit the number of inspected arcs in the AC3–algorithm and to perform less propagation than is actually possible. The memory consumption could be limited by limiting the number of part domains for a rectangle domain. Thus, we represent a rectangle domain not at the most detailed level and loose some propagation. We should also try to merge rectangle part domains in order to reduce the number of part domains. Another idea is to simplify the computation of a common kernel for several part domains in Section 7.4. It should be investigated how much propagation is lost if the common kernel is based on the bounding box of the part domains. Such heuristics are necessary for this NP-hard problem.

Due to the inefficiency of the `diff2`–implementation, we won't use the `diff2`–constraint and the marker model in our further experiments. We have explained in Section 6.3 that the marker model can provide helpful propagation with an efficient `diff2`–implementation. The implementation of the insertion heuristic from Section 9.3 works also without the exclusion marker model.

If we go back to Table 10.4, we can see that the ratio of the number of used to the number of available locomotives is approximately constant. This is an expected behaviour and shows that the penalty parameter λ reduces the

number of used locomotives.

10.1.2 Performance without the Marker Model

The insertion heuristic was tested on the train sets A and B. The examples 1–3 are based on train set A:

- Example 1: train set A with large departure time windows
- Example 2: train set A with more restriced departure time windows
- Example 3: example 2 with a reduced locomotive number

By large time windows we mean that the windows size is equal to the whole schedule period. The time windows in Appendix A.1 for example 2 where generated by the following procedure. [3] The time window center c is uniformly distributed over the whole schedule period. The half window length w is normal distributed with a mean of 6 hours and a standard deviation of 2 hours. The time windows are then given by $[\max\{c-w,0\}, \min\{c+w,24\}]$. Due to an error in the generation of the window sizes, the time windows became a bit large but example 2 shows the qualitative effect of departure time windows.

The parameter settings for the other parameters are the same as in Table 10.3. The results for the examples 1–3 can be found in the Tables 10.5, 10.6 and 10.7. [4]

n	10	20	30	40	50	60	70	80	90	100
m	3	6	9	12	15	18	21	24	27	30
used locomotives	3	6	6	10	12	15	18	> 19	> 19	> 19
time in s	0.3	1.0	1.2	2.3	3.4	5.5	7.8	10.0	22.3	45.0
memory in MB	–	–	–	–	154	154	183	183	205	260

Table 10.5: Results for Example 1

In the examples 1 and 2, the locomotive number is larger than the actually needed number of locomotives. We can determine the locomotive plan in one run. In example 3 (Table 10.7), we have not enough locomotives and some trips must be discarded. As explained in Section 9.3, we determine in a

[3] The random numbers where generated with MATLAB.

[4] The locomotive plan window allows a maximum of 19 locomotives, larger locomotive numbers are indicated by > 19. For some small problems, the memory measurement of the top program was not reliable.

n	10	20	30	40	50	60	70	80	90	100
m	5	10	15	20	25	30	35	40	45	50
used locomotives	3	5	8	10	12	15	17	17	> 19	> 19
time in s	0.3	0.9	1.7	3.5	6.6	8.0	17.8	21.4	30.1	47.0
memory in MB	–	–	–	–	38	70	70	107	172	209

Table 10.6: Results for Example 2

n	10	20	30	40	50	60	70	80	90	100
m	1	3	4	6	7	9	10	12	13	15
used locomotives	1	3	4	6	7	9	10	12	13	15
phase I (in s)	0.3	2.3	6.5	12.3	26.7	48.1	75.9	88.6	166.2	203.2
n'	5	15	23	34	39	50	56	72	75	86
phase II (in s)	0.3	0.8	2.3	4.5	5.9	14.5	16.5	39.9	44.5	54.3

Table 10.7: Results for Example 3

first run the trips that can be scheduled (phase I). We get a new trip number n' and determine for these remaining trips the locomotive plan (phase II).

Fig. 10.4 shows the run times for example 1, 2 and phase I of example 3. We can see that the run times of example 2 are higher than those of example 1 due to the more constrained time windows. The insertion heuristic must search longer for a feasible insertion position and also the track allocation problem becomes more difficult. The curves are not smooth and this indicates that the trip number is not the only parameter which determines the difficulty of a problem.

The difficulty of the track allocation problem depends also on other input parameters [KCO+97]. Especially the distribution of the trips over the network is an important factor. If many trips traverse the same track, it is harder to find a schedule without conflicts.

Example 3 shows in phase I significantly higher run times than the other two examples. The run times show an approximately cubic behaviour. The insertion heuristic must try all insertion positions until it can decide that a trip cannot be scheduled. The run times in phase II are in the same range as those in examples 1 and 2, because we have enough locomotives in this case.

For the examples 2 and 3, we have also measured the run time when the track allocation constraints of the TUFF system are switched off [5]. We

[5] i.e. the track constraints, the precedence constraints and the location constraints as well as the final search procedure for the determination of the departure times.

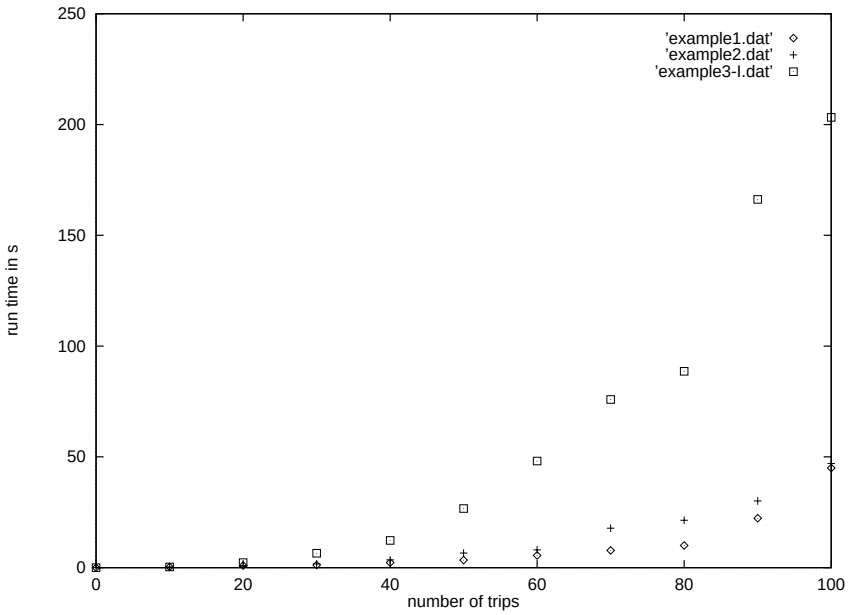

Figure 10.4: Performance for examples 1–3

don't get a solution in this case, but this should give us an estimation how the run time is distributed over the insertion heuristic part and the other constraints[6]. The results are shown in the Tables 10.8 and 10.9.

n	10	20	30	40	50	60	70	80	90	100
time in s	0.3	0.9	1.7	3.5	6.6	8.0	17.8	21.4	30.1	47.0
without constraints	0.2	0.4	0.7	0.7	1.8	3.2	3.4	6.4	5.3	9.2

Table 10.8: Results for Example 2 without TUFF constraints

n	10	20	30	40	50	60	70	80	90	100
phase I (in s)	0.3	2.3	6.5	12.3	26.7	48.1	75.9	88.6	166.2	203.2
without constraints	0.2	0.4	2.1	1.5	3.3	5.6	7.9	11.1	10.8	20.5

Table 10.9: Results for Example 3 without TUFF constraints

We can see in both cases that the overhead of the TUFF constraints dominates the total run time. This seems reasonable because the number of tracks is considerable larger than the number of trips in a problem, so that

[6]We can't isolate the insertion heuristic in another way, there are no profiling tools in Oz available.

the track allocation constraints consume the largest part of the run time. It can also be seen that the insertion heuristic has a higher share of the run time in example 3, because it must try all insertion positions for some trips.

The interpretation of the run times is difficult, because the run time behaviour of constraint systems is hard to analyze. We don't know the control flow in the program because the constraints interact dynamically. The measurements without TUFF constraints indicate that the run time is dominated by the track allocation constraints. This run time behaviour is not explained in [KCO+97]. We can see that the worst-case run time of $O(n^5)$ from Section 8.4 is not achieved in our examples and that the most difficult example has a run time of approximately $O(n^3)$.

We should also explain the high memory consumption of the TUFF system. This is a known deficiency of the TUFF system [KCO+97]. It comes from the fact that Oz creates in every choice point a copy of all FD-variables in the system. This is necessary for backtracking because the old variable values must be stored. The number of FD-variables in the TUFF system is quite large, all departure times for the track traversals and the waiting times at locations are represented by FD-variables. Due to the copying, the memory consumption grows linearly with the depth of the search tree.

This problem can be limited in Oz by *recomputation* [HMSW98]. This means that not in every choice point a copy of the variables is created, only in every n-th. If backtracking occurs, the old variable values must be recomputed from the nearest copy in the search tree. Thus, we trade space for time and get higher run times. We have used a recomputation distance of $n = 15$ for all examples in this Chapter. A bigger value for n doesn't reduce the memory consumption further.

The examples 4 and 5 are based on the train set B. Example 4 is an example with large departure time windows. The parameter settings are given in Table 10.10. In example 5 the number of locomotives was reduced so that some trips remain unscheduled.

number of locations	11
total schedule time (hours)	48
train velocity (km/h)	120
velocity passive transports	120
λ (min)	$\overline{\delta_2}$=350
location slack factor μ	0.05
departure time windows (hours)	48
turn time Θ (mins)	30

Table 10.10: Parameters for Example 4 and 5

n	10	20	30	40	50
m	3	6	9	12	15
used locomotives	3	5	7	9	11
time in s	1.1	2.8	8.9	18.6	36.5
without constraints (in s)	0.3	0.6	0.7	1.8	3.9
memory in MB	18	19	62	70	129

Table 10.11: Results for Example 4

n	10	20	30	40	50
m	2	2	4	5	7
used locomotives	2	2	4	5	7
phase I (in s)	0.4	8.3	24.2	70.5	155.7
n'	4	9	19	22	29
phase II (in s)	0.5	1.2	3.2	7.4	16.9

Table 10.12: Results for Example 5

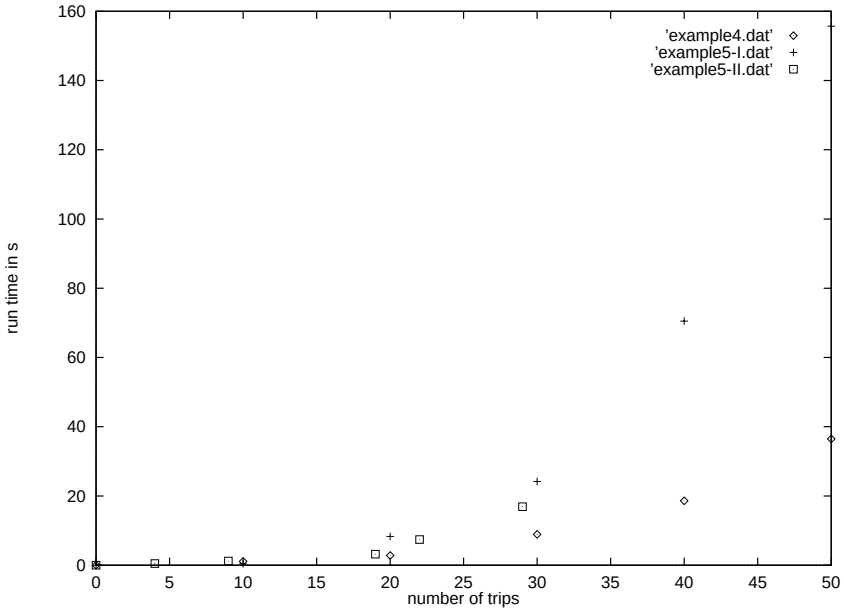

Figure 10.5: Performance for examples 4–5

Example 4 shows considerable higher run times than the examples 1 and 2 for the same number of trips. This can be explained by the higher difficulty of this example. One measure for the difficulty of the track allocation problem is the maximum track load, i.e. the maximum number of trains that traverse a certain track. For $n = 50$ trips, example 1 and 2 have a maximum track load of 11 trains whereas example 3 has a maximum track load of 22 trains (i.e. nearly the half of the 50 trains). This leads to higher run times.

The increase of the run times in example 5 comes again from the reduced locomotive number. They also seem to have a cubic behaviour, although the problems are not large enough in order to be sure. We show the locomotive plan for $n = 30$ trips in Fig. 10.6. Due to the long schedule period, the locomotive plan was split into two windows. In phase I, 11 trips are discarded and 19 trips remain. They are shown in Fig. 10.6. The locomotive start locations for this plan are given in Table 10.13. A passive transport has an approximately 20% shorter duration than a corresponding normal trip, because the trip durations include minimum waiting times at the locations, the passive transport durations do not.

l_1	Luleå	l_3	Umeå
l_2	Stockholm	l_4	Göteborg

Table 10.13: Locomotive start locations for Fig. 10.6

10.2 Small Example

After the performance measurements, we will now look at a smaller example in order to show how the TUFF system solves the track allocation problem. We can find the net from our first example from Section 3.3 in the Lake Vänern area (Fig. 10.7). The only difference is that this net contains only one double track path, the net in Section 3.3 contained two. The city names and the correspondent letters from Section 3.3 are given in Table 10.14. The

A	Ch	Charlottenberg
B	La	Laxå
C	Ki	Kil
D	Me	Mellerud
E	Ko	Kornsjö
F	Gö	Göteborg

Table 10.14: City names

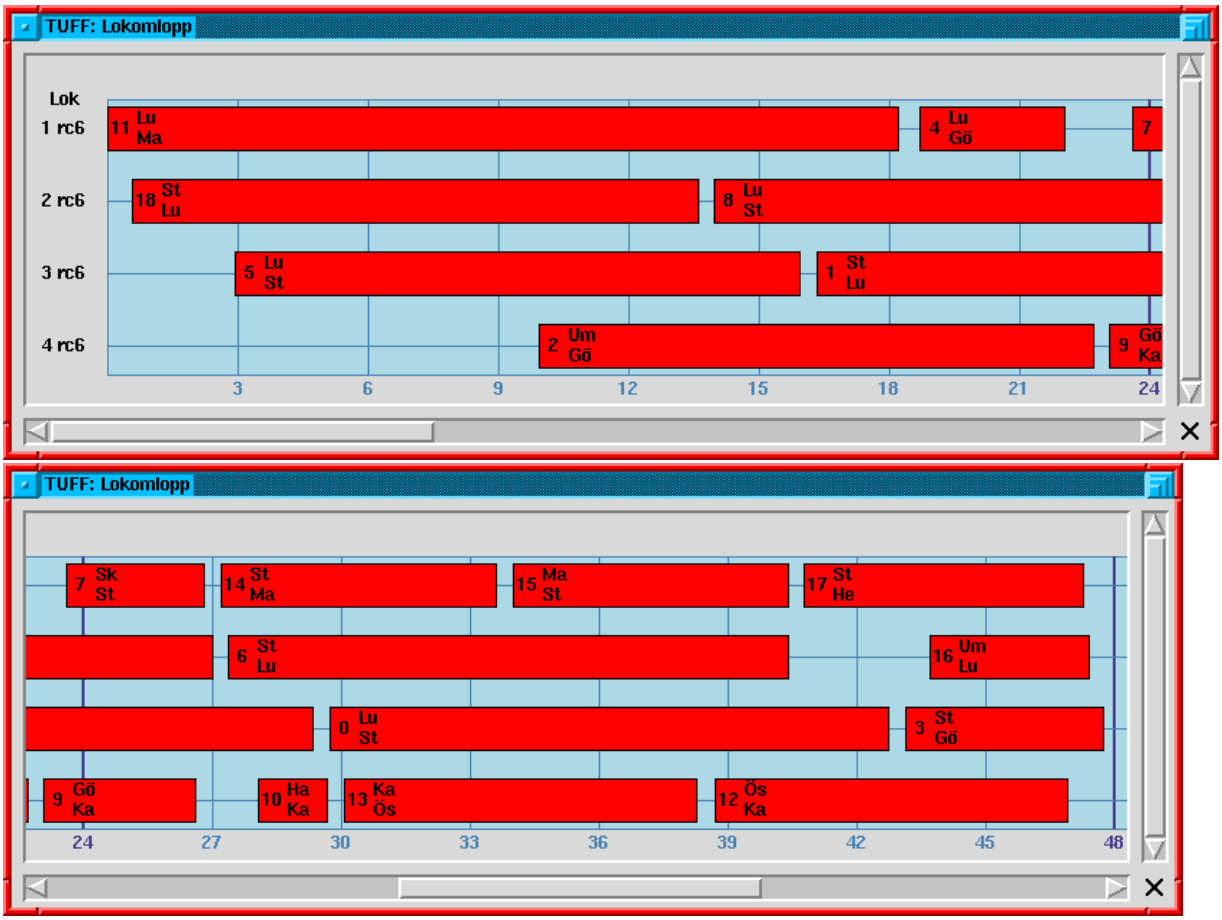

Figure 10.6: Locomotive plan for $n' = 19$ trips

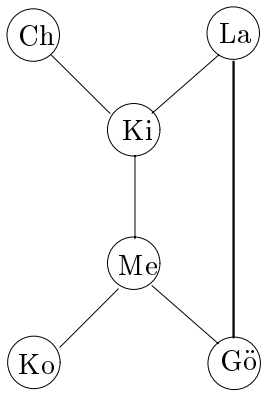

Figure 10.7: 6 cities in West Sweden

train file can be found in Appendix A.3. The departure time windows from the example in Section 3.3 were shifted by eight hours but have still their relative positions. The problem parameters are given in Table 10.15.

number of locations	6
total schedule time (hours)	24
train velocity (km/h)	120
velocity passive transports (km/h)	120
λ (min)	$\overline{\delta_2}$=124
location slack factor μ	0.05
departure time windows (hours)	24
turn time Θ (mins)	30

Table 10.15: Parameters for the small example

The start locations of the locomotives are given in Table 10.16 [7]. The routes are the same as in Section 3.3.

We look at the locomotive plan in Fig. 10.8. The minimum number of locomotives for this problem is obviously three, because three trips start in the earliest departure time interval. Our heuristic finds a solution with three locomotives and a passive transport between the trips p_5 and p_6. The other gaps come from the fact that the trips p_2 and p_5 must wait until their departure time window is reached. This solution is better than the ones which were presented in Section 3.3.

We show the timetable in three diagrams. The paths Gö–Me–Ki–Ch, Gö-La and La–Ki–Me–Ko cover the whole net. We begin with Gö–Ch in Fig. 10.9.

[7] The start locations of l_4 and l_5 where exchanged compared to Section 3.3.

l_1	Charlottenberg
l_2	Charlottenberg
l_3	Göteborg
l_4	Göteborg
l_5	Kil

Table 10.16: Start locations of the locomotives for the small example

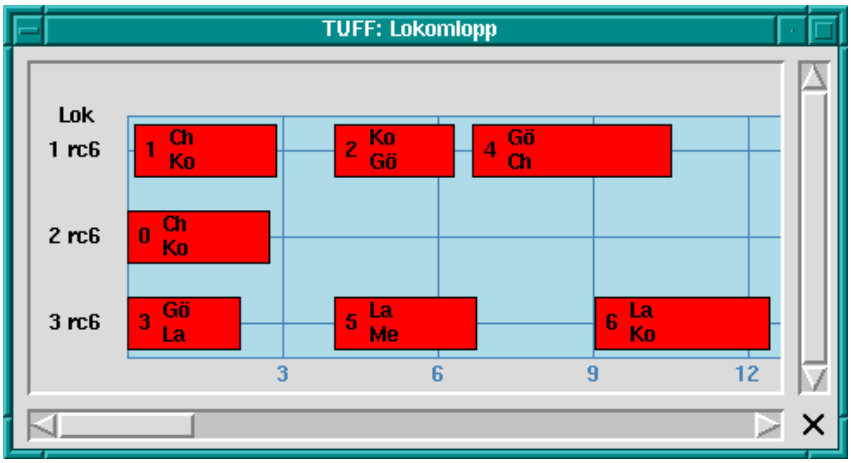

Figure 10.8: Locomotive plan for our small example

We can see the headway distance between the trips p_0 and p_1. This is the safety distance from Def. 3 that two trains must keep which run in the same direction. The trip numbers are also indicated in this figure, trip p_0 starts before trip p_1.

Fig. 10.10 contains only one train for the path Gö-La. Only trip p_3 uses this path. Fig. 10.11 shows the third path La–Ko, the trains between Kil and Mellerud are identical to those in Fig. 10.9.

In order to get a collision between trains of opposite directions, we decrease the departure time of trip p_4 and set its window to $[0, 2]$. Fig. 10.12 shows the new solution with four locomotives and one passive transport between the trips p_2 and p_6. An additional locomotive is needed because we have now four trips which start in the earliest departure time interval.

If we look now at the changed timetable for the path Göteborg–Charlottenberg in Fig. 10.13, we can see that the trips p_0, p_1 and p_4 must pass a certain track between Mellerud and Kil sequentially. The trip numbers where omitted in this figure because they were not readable. Trip p_0 starts before trip p_1 in Charlottenberg and trip p_4 starts in Göteborg. The conflict is solved

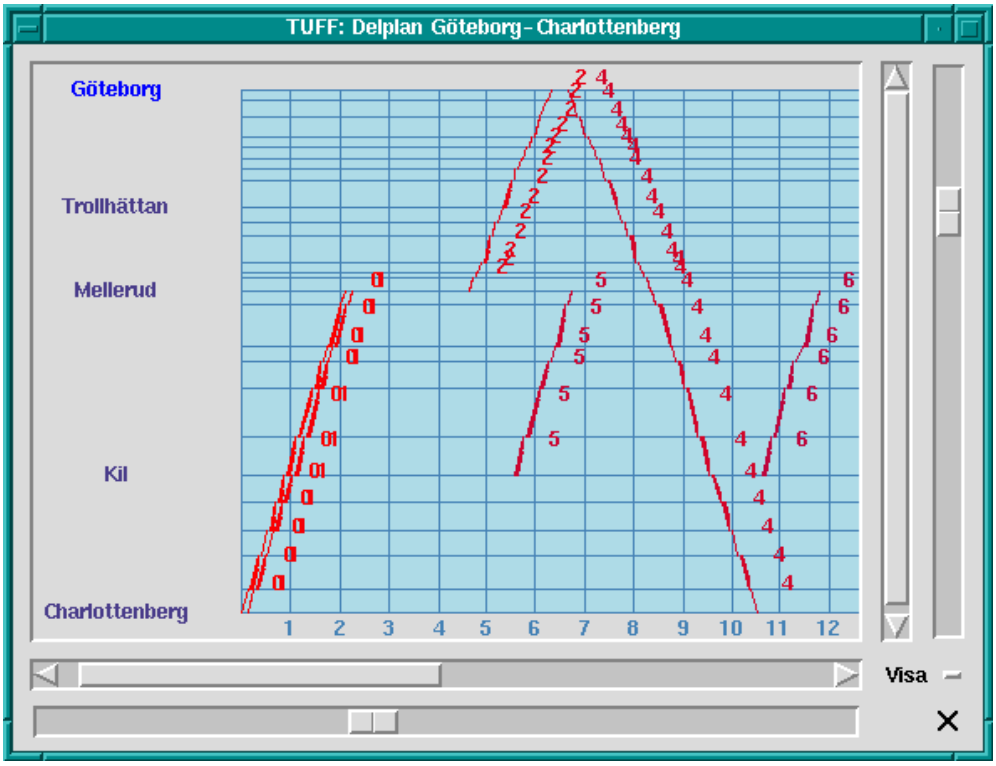

Figure 10.9: Timetable for Göteborg–Charlottenberg

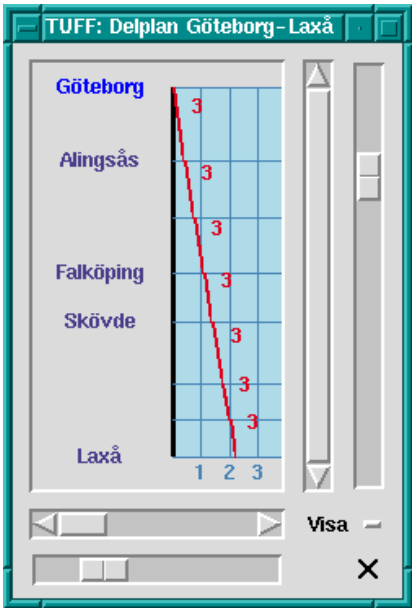

Figure 10.10: Timetable for Göteborg–Laxå

by the following order for the track traversals: p_0, p_4 and then p_1.

10.3 Larger Example

We will now look at some locomotive plans for a larger example. They are based on train set A from Section 10.1 with the network from Fig. 10.1.

The train file can be found in Appendix A.4. The trains have velocities between 80 and 120 km/h. The routes are given in Appendix A.1.2. The start locations of the locomotives are given in Table 10.17. We use the locomotives l_1 to l_{12} first and add the other locomotives later. The problem parameters are given in Table 10.18.

We will look at the effect of the penalty parameter λ in the following two locomotive plans. In Fig. 10.14, we have chosen $\lambda = \frac{1}{2}\overline{\delta_2} = 57$. The time scale is not visible in this screen shot, the vertical lines in Fig. 10.14 have a distance of three hours. Ten of the twelve available locomotives are used, this seems not to be the minimum number because the locomotive l_9 has only one trip. The heuristic finds good trip connections quite well, observe that the small gaps come from the turn time. The wider gaps come from passive transports, Fig. 10.14 contains 18 passive transports. Most of them have short durations (compared with the trip durations).

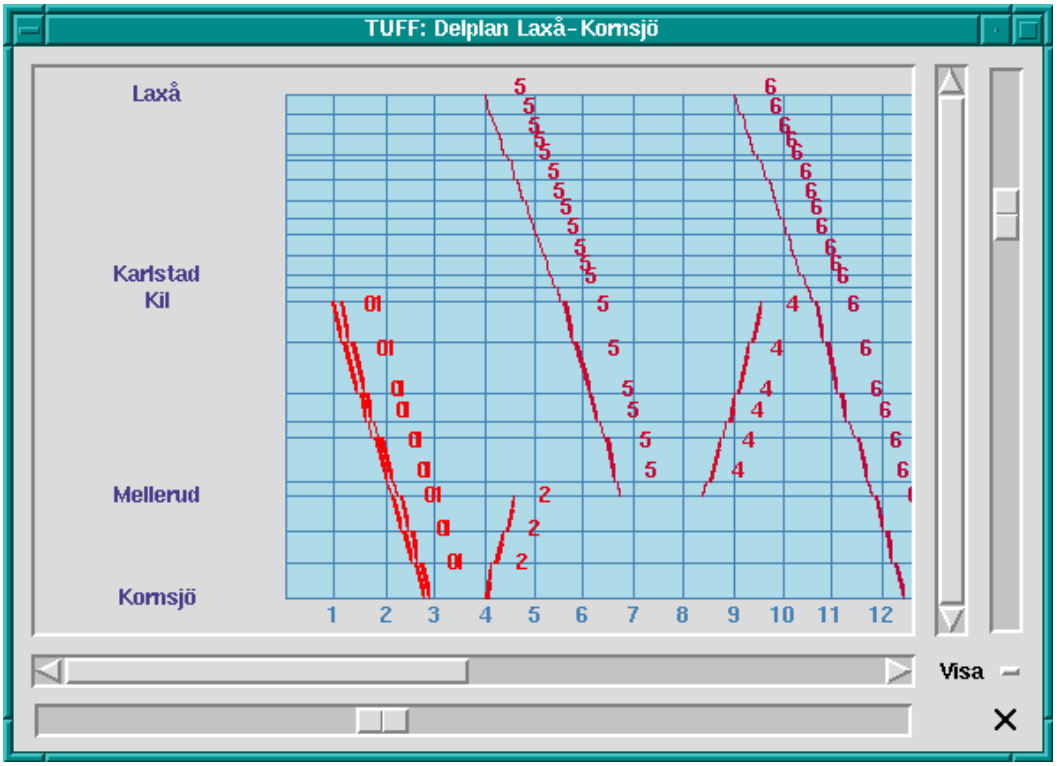

Figure 10.11: Timetable for Laxå–Kornsjö

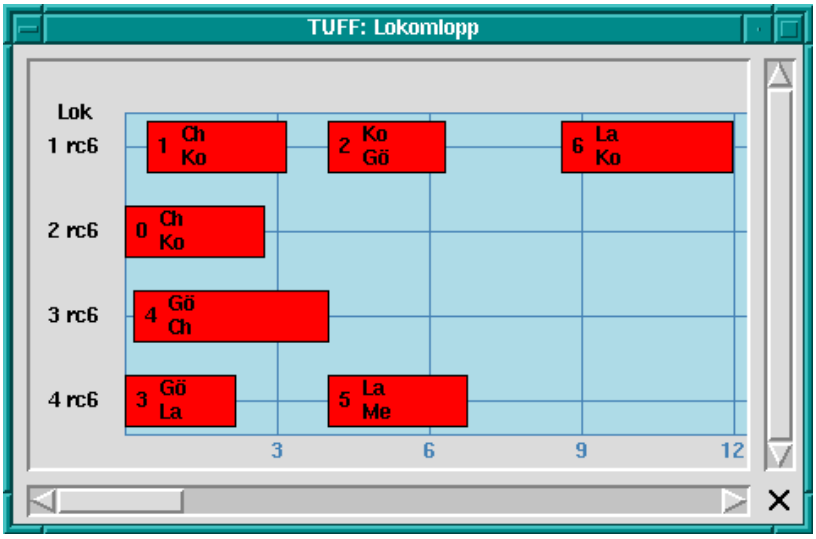

Figure 10.12: Locomotive plan with 4 locomotives

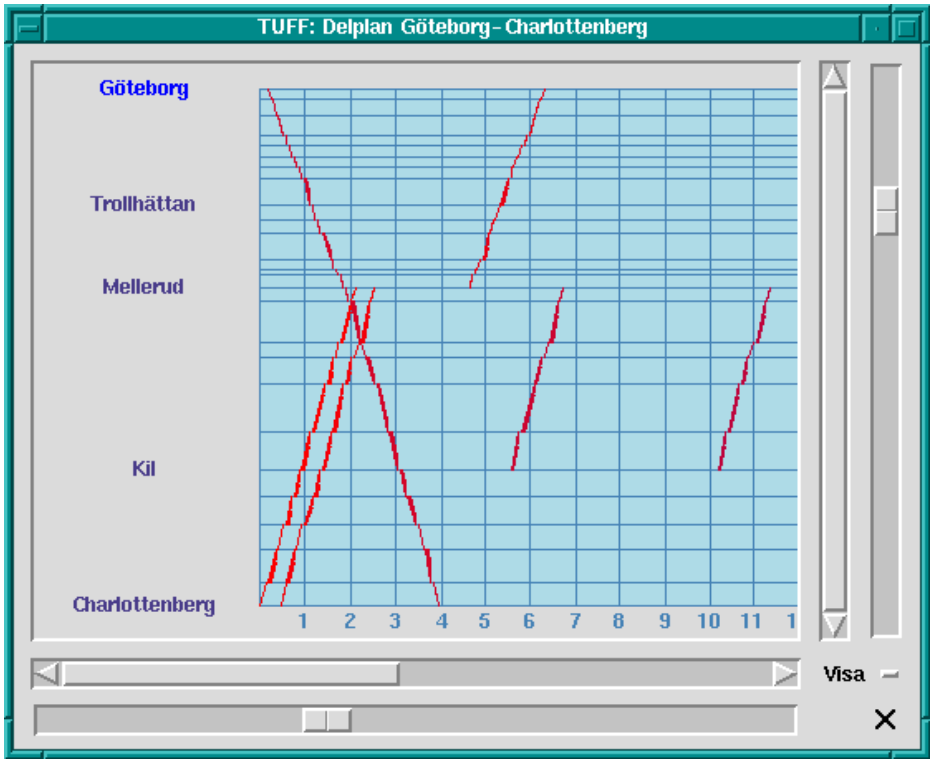

Figure 10.13: New timetable for Göteborg–Charlottenberg

l_1	Uppsala	l_{11}	Göteborg
l_2	Karlstad	l_{12}	Uppsala
l_3	Hallsberg	l_{13}	Skövde
l_4	Göteborg	l_{14}	Mjölby
l_5	Stockholm	l_{15}	Västerås
l_6	Charlottenberg	l_{16}	Uppsala
l_7	Nässjö	l_{17}	Karlstad
l_8	Hallsberg	l_{18}	Hallsberg
l_9	Norrköping	l_{19}	Göteborg
l_{10}	Stockholm	l_{20}	Stockholm

Table 10.17: Locomotive start locations for the larger example

number of locations	14
total schedule time (hours)	18
velocity passive transports (km/h)	120
location slack factor μ	0.05
departure time windows (hours)	18
$\overline{\delta_2}$ (mins)	114
turn time Θ (mins)	30

Table 10.18: Parameters for first example

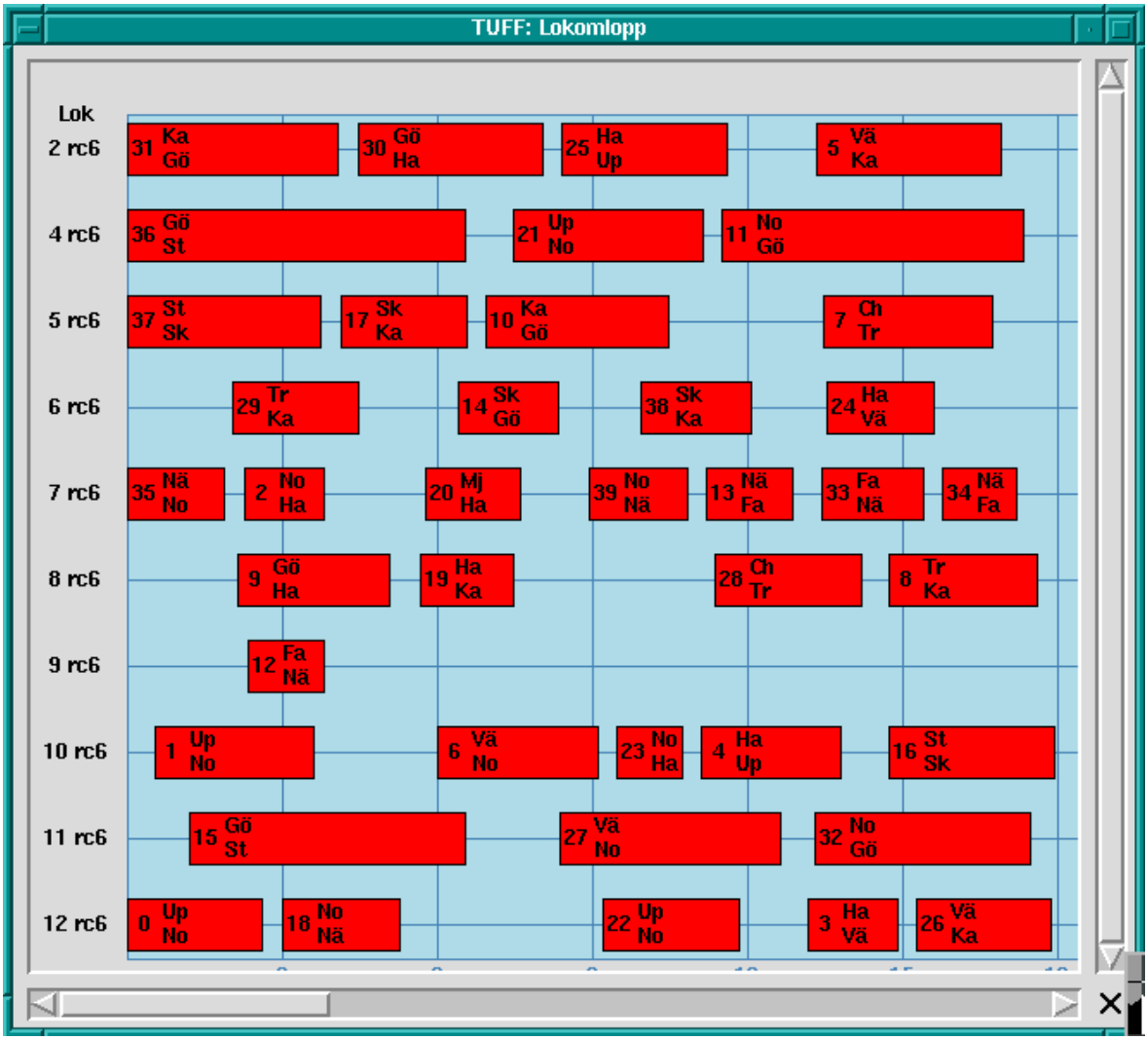

Figure 10.14: Locomotive plan for $\lambda = \frac{1}{2}\overline{\delta_2}$

Figure 10.15: Locomotive plan for $\lambda = \overline{\delta_2} = 113$

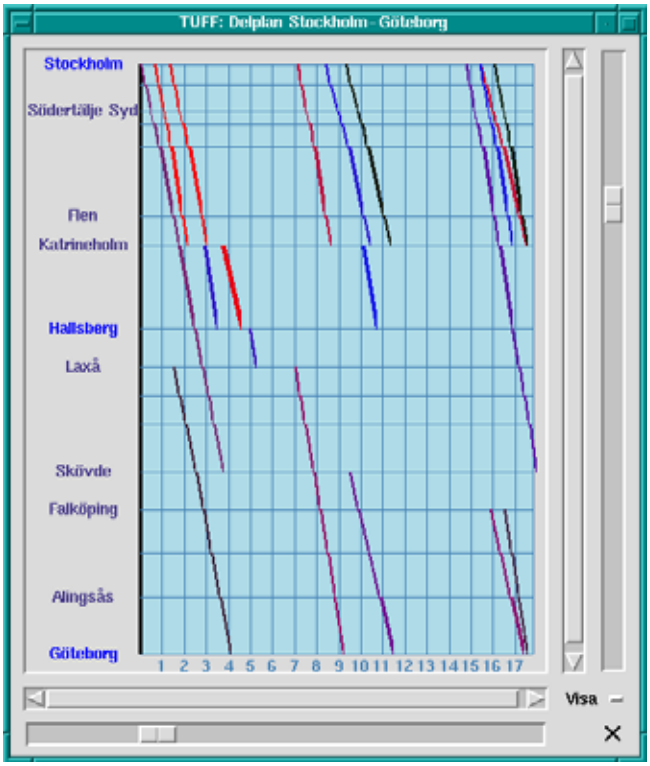

Figure 10.16: Timetable for $\lambda = \overline{\delta_2}$, direction Stockholm-Göteborg

In Fig. 10.15, we have increased λ to $\lambda = \overline{\delta_2}$. We need now only nine locomotives. This shows that λ decreases the number of used locomotives. A further increase of λ didn't lead to a smaller locomotive number in this example. We save also two passive transports in this example (16 passive transports), so that we should prefer this parameter setting for λ.

In Fig. 10.16, the timetable for the path St-Ka-Ha-Sk-Fa-Gö in Fig. 10.1 is shown. This is a path with double tracks, so that only trains of one direction are shown, because they can't be influenced by trains of the opposite direction. We can see the safety areas of the trains and that the track allocation conditions are fulfilled. For example, the two trains in the bottom-right corner of Fig. 10.16 correspond to the trips p_{11} and p_{32} in Fig. 10.15.

We add now time windows to this example. The modified train file can be found in Appendix A.5. The trains are ordered after their latest departure time in order to show the order in which the trips are inserted. This is actually not the exact insertion order, because the trips are sorted dynamically in every choice point (compare the remark in Section 9.3). The time windows have an average size of 4.7 hours and where set manually. The problem parameters are the same as in Table 10.18 and we use now all 20

locomotives from Table 10.17.

We will look again at locomotive plans for different λ–values. Fig. 10.17 shows the plan for $\lambda = \overline{\delta_2}$. 15 of the 20 available locomotives are used. l_2 and l_{14} have only a single trip so that it should be possible to reduce the locomotive number further. The number of passive transports is 12. This is not counterintuitive as we have now more locomotives than in the case without time windows. Thus, we can save passive transports. We can also see that considerable waiting times for the locomotives can occur. Take for example the trips p_5 and p_{28} on the locomotive l_{11}. The locomotive must wait several hours between these trips at the same location. Our heuristic can't detect this cost factor because it minimizes only the locomotive number and the amount of passive transports.

In Fig. 10.18, λ was set to $\lambda = 2\overline{\delta_2}$ and this saved one locomotive (14 locomotives). A further increase of λ didn't lead to a smaller locomotive number. We get 19 passive transports.

The penalty parameter mechanism is a weakness in this heuristic. The examples suggest that values in the neighbourhood of $\overline{\delta_2}$ lead to a minimum number of locomotives. However, this parameter must be adjusted manually from problem to problem in order to find the minimum locomotive number. As we have explained in Section 4.4, we can compute a lower bound for the locomotive number by considering the kernel time intervals of the trips. This is only a lower bound because the start locations of the locomotives are not taken into account. This estimation can give us a hint how far we are from the optimum locomotive number.

The heuristic is well-suited for the minimization of the passive transports, as can be seen in the examples. The generated passive transports are usually short compared to the trip durations. As we have mentioned before, the heuristic doesn't consider the waiting times of the locomotives at all. This is an additional weakness.

The improvement heuristic from Section 8.7 could probably be used to improve the plans further, especially the amount of passive transport. Also the locomotive number could be reduced in some cases by inserting lonely trips on locomotives with more trips. Such post-optimization procedures have been successfully applied to other vehicle routing problems [RSL77], [ABBG83], [PR95], [Sav85].

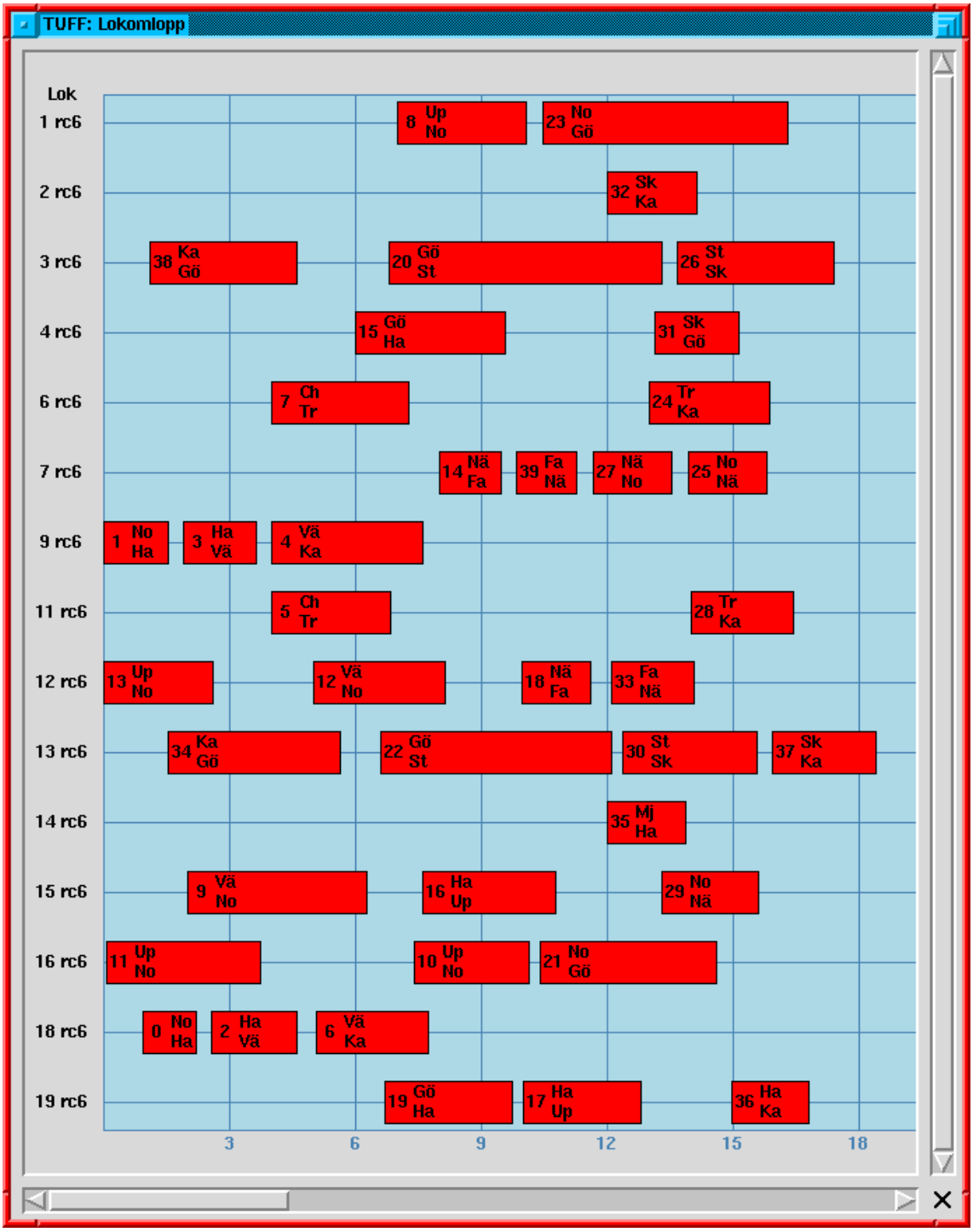

Figure 10.17: Locomotive plan for $\lambda = \overline{\delta_2}$

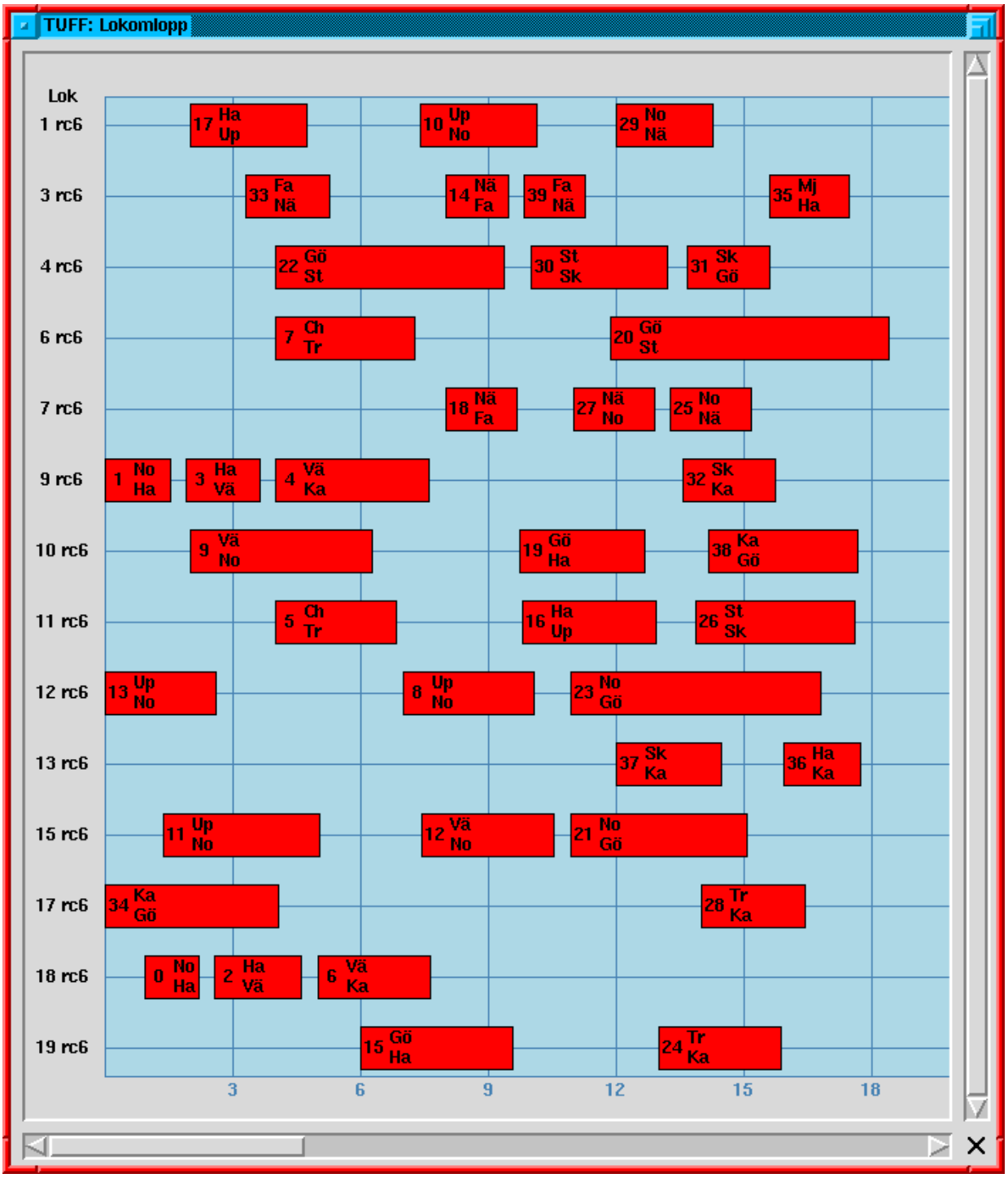

Figure 10.18: Locomotive plan for $\lambda = 2\overline{\delta_2}$

Chapter 11

Conclusions

The experiments show that we can solve problems of reasonable size with our approach. The run times are moderate but the memory consumption is a major problem if we want to handle bigger problems. Regarding the cost function, the heuristic considers the number of locomotives and the amount of passive transports, but not the waiting times. We do not obtain an optimal solution but the examples have shown that the generated solutions have a reasonable quality. Observe that we handle only the case where each transport requires one locomotive, we have mentioned in Section 1.4 that freight transports can also require several locomotives. The track allocation problem is solved for the normal transports, but the track allocation for the passive transports must be done in a second step.

The use of a specialized heuristic for route building was necessary because we have no powerful constraint formulation for the routing problem. The exclusion marker model ensures only feasible solutions and doesn't help for optimization, so that the search strategy becomes more important. Thus, we can only use the track allocation constraints as side constraints in our route building heuristic. We have not achieved the goal of constraint programming, a reduction of the search space by constraint propagation. Instead, we use a heuristic that generates a near-optimum solution.

An alternative to the exclusion marker model is the use of another high-level global constraint, the `cycle`-constraint in the CHIP system [BC94]. This constraint finds minimum-cost cycles in directed graphs. It can be used in transport problems if one uses a graph model which is similar to the models which are used in network flow approaches [Sim95b], [Sim96]. This constraint provides more propagation than the exclusion marker model. It has the disadvantage that dummy nodes for the passive transports must be introduced. The number of passive transports is not known beforehand and the number of potential passive transports can be quite large.

The extension of constraint programming languages by high-level constraints like the `diffn` and the `cycle`-constraint seems promising because their abstraction level is adapted to the problem itself. With these high-level constraints, passive transportation problems have been solved successfully with the CHIP system [SBCK95], [BKC94].

The goal to develop an efficient `diff2`-implementation for the exclusion marker model was too ambitious for this work. It would have required a much greater effort without contributing directly to the locomotive assignment problem. Although the `diff2`-implementation does not scale, the algorithm contains solution ideas that should be pursued further. Together with heuristics for area reasoning, an efficient implementation could be developed.

Regarding the behaviour of the TUFF system in case of over-constrained problems, an explanation component that shows the conflicting trips should be added. The output "no solution" is not satisfactory. Another idea could be to treat some constraints as soft, e.g. certain departure time windows in order to find any solution at all. "Soft" means that these constraints may be violated. Additionally, the time windows in freight transport are not that strict and one could introduce soft instead of hard time windows. We could add a delay cost if a transport starts after its time window. In this case, we could also delay trains until a locomotive becomes available.

In order to limit the problem size in practical problems, some decomposition into geographical regions should be tried. Starting from a plan for long-distance trains, one could generate plans for regional trains for different regions independently.

After the integration of track allocation and locomotive assignment, the integration with personnel and carriage planning remains to be done. For this purpose, it would be better to have a powerful constraint model instead of a specialized search heuristic. It is easier to combine the different constraints for the subproblems than to integrate several search heuristics.

In the TUFF 3 project at SICS, the coordination of track allocation and locomotive assignment shall be done by distributed planning agents. SJ is interested in a distributed planning mechanism because several people at different locations take part in the planning process. This is a totally different approach compared to this work, where one planner for both planning tasks has been developed. The main goal in this project is to develop suitable coordination mechanisms between different planners so that the quality of the overall solution can be improved. The planners can use constraint programming technology or conventional techniques from operations research (e.g. network flows). Maybe it's possible to use a route building heuristic from this work in a locomotive planner.

Bibliography

[ABBG83] A. Assad, M. Ball, L. Bodin, and B. Golden. Routing and scheduling of vehicles and crews. *Computers and Operations Research*, 10(2):63–211, 1983.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows*, pages 296–297 and 649–650. Prentice Hall, 1993.

[BC94] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[BH80] U. Bokinge and D. Hasselström. Improved vehicle scheduling in public transport through systematic changes in the time-table. *European Journal of Operational Research*, 5:388–395, 1980.

[BKC94] G. Baues, P. Kay, and P. Charlier. Constraint based resource allocation for airline crew management. In *ATTIS 94*, Paris, Apr 1994. http://www.cosytec.fr.

[Bru95] R. Bruns. *Wissensbasierte Genetische Algorithmen*. PhD thesis, Universität Oldenburg, 1995. Infix-Verlag.

[BWZ97] M.R. Bussieck, T. Winter, and U.T. Zimmermann. Discrete optimization in public rail transport. *Mathematical Programming*, 79:415–444, 1997.

[CL97] Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In Lee Naish, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*. MIT Press, 1997. http://www.ens.fr/~caseau/papers.html.

[DHKK97] J. Drott, E. Hasselberg, N. Kohl, and M. Kremer. A planning system for locomotive scheduling. Technical report, Carmen Systems AB, 1997. http://www.carmen.se.

[dVT91] F. du Verdier and J.P. Tsang. Un raisonnement spatial par propagation de contraintes. In *11ièmes Journées Internationales: Les Systèmes Experts et leurs Applications*, pages 297–314, Avignon, 1991.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability. A guide to the theory of NP-completeness.* W.H. Freeman, 1979.

[HMSW98] M. Henz, M. Müller, C. Schulte, and J. Würtz. *The Oz Standard Modules.* German Research Centre for Artifical Intelligence (DFKI), Postfach 15 11 50, D-66041 Saarbrücken, Germany, Feb 1998. http://www.ps.uni-sb.de/oz/documentation.

[HNP+97] H. Heid, D. Nicklas, A. Porrmann, T. Schäffer, and V. Scholz. Zwischenbericht der Projektgruppe Fahrgemeinschaften. Technical Report 10, Fakultät Informatik der Universität Stuttgart, Breitwiesenstrasse 20–22, D-70565 Stuttgart, Germany, 1997.

[KCO+97] P. Kreuger, M. Carlsson, J. Olsson, T. Sjöland, and E. Åström. The TUFF train scheduler – two trip scheduling on single track networks. In A. Davenport, editor, *Third International Conference on Principles and Practice of Constraint Programming, Workshop on Industrial Constraint-Directed Scheduling*, Schloss Hagenberg, Linz, Austria, 1997.

[Kum92] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, pages 32–44, Spring 1992.

[LK81] J. Lenstra and A. Rinnooy Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227, 1981.

[Loe98] A. Loebel. *Optimal Vehicle Scheduling in Public Transit.* PhD thesis, TU Berlin, 1998. Shaker-Verlag, Aachen.

[MV80] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 17–27, 1980.

[MW97] T. Müller and J. Würtz. *The Constraint Propagator Interface of DFKI Oz.* German Research Centre for Artifical Intelligence (DFKI), Dec 1997. http://www.ps.uni-sb.de/oz/documentation.

[Nor] A. Nordin. Personal Communication with Anders Nordin at SICS, May 1998.

[OW93] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*, pages 232–242. Reihe Informatik Band 70. BI-Wissenschaftsverlag, 1993.

[PR93] J.Y. Potvin and J.M. Rousseau. A parallel route building algorithm for the vehicle routing and scheduling problem with time windows. *European Journal of Operational Research*, 66:331–340, 1993.

[PR95] J.Y. Potvin and J.M. Rousseau. An exchange heuristic for routeing problems with time windows. *Journal of the Operational Research Society*, 46:1433–1446, 1995.

[Ros85] J. B. Rosenberg. Geographical data structures compared: A study of data structures supporting region queries. *IEEE Transactions on Computer-Aided Design*, 4(1):53–67, Jan 1985.

[RSL77] D. Rosenkrantz, R. Sterns, and P. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comp.*, 6:563–581, 1977.

[Sam89] H. Samet. *The Design and Analysis of Spatial Data Structures*, chapter 3, pages 200–213. Addison-Wesley, 1989.

[Sav85] M.W.P. Savelsbergh. Local search in routing problems with time windows. *Annals of Operations Research*, 4:285–305, 1985.

[SBCK95] H. Simonis, N. Beldiceanu, P. Charlier, and P. Kay. A model of TACT. Technical report, COSYTEC SA, Mar 1995. http://www.cosytec.fr.

[Sch98] C. Schulte. *Finite Domain Constraint Programming in Oz: A Tutorial.* German Research Centre for Artifical Intelligence (DFKI), Feb 1998. http://www.ps.uni-sb.de/oz/documentation.

[SIC] SICS. Swedish Institute of Computer Science. http://www.sics.se.

[Sim95a] H. Simonis. Modelling machine set-up time in CHIP. Technical report, COSYTEC SA, Mar 1995. http://www.cosytec.fr.

[Sim95b] H. Simonis. The use of exclusion constraints to handle location continuity conditions. Technical report, COSYTEC SA, Mar 1995. http://www.cosytec.fr.

[Sim96] H. Simonis. A problem classification scheme for finite domain constraint solving. In *CP 96 - Principles and Practice of Constraint Programming, Applications Workshop*, 1996.

[SJ] SJ. Statens Järnvägar. http://www.sj.se.

[SJ 98] SJ Godstransportdivision, SE-10550 Stockholm, Sweden. *Frakthandboken*, 1998.

[Sol86] M. M. Solomon. On the worst-case performance of some heuristics for the vehicle routing and scheduling problem with time window constraints. *Networks*, 16:161–174, 1986.

[Sol87] M. M. Solomon. Algorithms for the vehicle routing and scheduling problem with time window constraints. *Operations Research*, 35(2):254–265, March-April 1987.

[TAT91] T. Tokuyama, T. Asano, and S. Tsukiyama. A dynamic algorithm for placing rectangles without overlapping. *Journal of Information Processing*, 14(1):30–35, 1991.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction*, chapter 1. Academic Press, 1993.

[Wür97] J. Würtz. Constraint-based scheduling in Oz. In U. Zimmermann, U. Derigs, W. Gaul, R. Möhrig, and K.-P. Schuster, editors, *Operations Research Proceedings 1996*, pages 218–223. Springer-Verlag, Berlin, Heidelberg, New York, 1997. Selected Papers of the Symposium on Operations Research (SOR 96), Braunschweig, Germany, September 3–6, 1996.

Appendix A

Examples

A.1 Performance Example A

A.1.1 Locomotive Start Locations

```
locomStartV:s(1:'Uppsala' 2:'Karlstad' 3:'Hallsberg'
              4:'Goeteborg' 5:'Stockholm' 6:'Charlottenberg'
              7:'Naessjoe' 8:'Hallsberg' 9:'Norrkoeping'
              10:'Stockholm' 11:'Goeteborg' 12:'Uppsala'
              13:'Skoevde' 14:'Mjoelby' 15:'Vaesteraas'
              16:'Uppsala' 17:'Karlstad' 18:'Hallsberg'
              19:'Goeteborg' 20:'Stockholm' 21:'Charlottenberg'
              22:'Naessjoe' 23:'Hallsberg' 24:'Norrkoeping'
              25:'Stockholm' 26:'Goeteborg' 27:'Uppsala'
              28:'Skoevde' 29:'Mjoelby' 30:'Vaesteraas'
              31:'Uppsala' 32:'Karlstad' 33:'Hallsberg'
              34:'Goeteborg' 35:'Stockholm' 36:'Charlottenberg'
              37:'Naessjoe' 38:'Hallsberg' 39:'Norrkoeping'
              40:'Stockholm' 41:'Uppsala' 42:'Karlstad'
              43:'Hallsberg' 44:'Goeteborg' 45:'Stockholm'
              46:'Charlottenberg' 47:'Naessjoe' 48:'Hallsberg'
              49:'Norrkoeping' 50:'Stockholm' )
```

A.1.2 Routes

```
routeName:'Charlottenberg-Trollhaettan'
Ch-Tr
routeName:'Falkoeping-Naessjoe'
Fa-Nae
routeName:'Goeteborg-Hallsberg'
Goe-Fa-Sk-Ha
routeName:'Goeteborg-Stockholm'
Goe-Fa-Sk-Ha-Ka-St
routeName:'Hallsberg-Karlstad'
Ha-Ka
routeName:'Hallsberg-Uppsala'
Ha-Ka-St-Up
routeName:'Hallsberg-Vaesteraas'
Ha-Vae
routeName:'Karlstad-Goeteborg'
Ka-Sk-Fa-Goe
routeName:'Mjoelby-Hallsberg'
Mj-Ha
routeName:'Norrkoeping-Hallsberg'
No-Ka-Ha
routeName:'Norrkoeping-Goeteborg'
No-Mj-Nae-Fa-Goe
routeName:'Norrkoeping-Naessjoe'
No-Mj-Nae
routeName:'Skoevde-Goeteborg'
Sk-Fa-Goe
routeName:'Skoevde-Karlstad'
Sk-Ka
routeName:'Stockholm-Skoevde'
St-Ka-Ha-Sk
routeName:'Trollhaettan-Karlstad'
Tr-Ka
routeName:'Uppsala-Norrkoeping'
Up-St-No
routeName:'Vaesteraas-Katrineholm'
Vae-St-Ka
routeName:'Vaesteraas-Norrkoeping'
Vae-St-No
```

A.1.3 Train File

```
[%0
 addTrain(routeName:'Uppsala-Norrkoeping'
          typeOfEngine:rc6
```

```
        meanSpeed:120.0
        depTimeSpec:monday(0#13.5849 ))
%1
addTrain(routeName:'Uppsala-Norrkoeping'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(9.6051#15.8205))
%2
addTrain(routeName:'Norrkoeping-Hallsberg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(6.7361#18.5565))
%3
addTrain(routeName:'Hallsberg-Vaesteraas'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(1.1275#14.6277))
%4
addTrain(routeName:'Hallsberg-Uppsala'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0#9.4398))
%5
addTrain(routeName:'Vaesteraas-Katrineholm'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0.1926#7.7998))
%6
addTrain(routeName:'Vaesteraas-Norrkoeping'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(6.8708#16.8400))
%7
addTrain(routeName:'Charlottenberg-Trollhaettan'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0#8.3827))
%8
addTrain(routeName:'Trollhaettan-Karlstad'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(11.5099#20.7053))
%9
addTrain(routeName:'Goeteborg-Hallsberg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0#6.6093))
%10
addTrain(routeName:'Karlstad-Goeteborg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(6.9330#23.4336))
%11
addTrain(routeName:'Norrkoeping-Goeteborg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(8.6230#21.6954))
%12
addTrain(routeName:'Falkoeping-Naessjoe'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(3.4311#19.9217))
%13
addTrain(routeName:'Naessjoe-Falkoeping'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0#14.1581))
%14
addTrain(routeName:'Skoevde-Goeteborg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(10.1426#22.4429))
%15
addTrain(routeName:'Goeteborg-Stockholm'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0#8.6767))
%16
addTrain(routeName:'Stockholm-Skoevde'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(11.2244#20.1637))
%17
addTrain(routeName:'Skoevde-Karlstad'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(10.6966#24.0000))
%18
addTrain(routeName:'Norrkoeping-Naessjoe'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(17.0799#24.0000))
%19
addTrain(routeName:'Hallsberg-Karlstad'
```

```
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(14.3804#24.0000))
%20
addTrain(routeName:'Mjoelby-Hallsberg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0#7.6814))
%21
addTrain(routeName:'Uppsala-Norrkoeping'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(9.5999#23.0063))
%22
addTrain(routeName:'Uppsala-Norrkoeping'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(4.4799#13.9240))
%23
addTrain(routeName:'Norrkoeping-Hallsberg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(2.2605#17.7075))
%24
addTrain(routeName:'Hallsberg-Vaesteraas'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(18.5398#24.0000))
%25
addTrain(routeName:'Hallsberg-Uppsala'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0.6860#12.4040))
%26
addTrain(routeName:'Vaesteraas-Katrineholm'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(16.3637#24.0000))
%27
addTrain(routeName:'Vaesteraas-Norrkoeping'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(18.1988#24.0000))
%28
addTrain(routeName:'Charlottenberg-Trollhaettan'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(17.8257#24.0000))
%29
addTrain(routeName:'Trollhaettan-Karlstad'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0#8.7589))
%30
addTrain(routeName:'Goeteborg-Hallsberg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(10.9410#22.7161))
%31
addTrain(routeName:'Karlstad-Goeteborg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(3.3157#14.2206 ))
%32
addTrain(routeName:'Norrkoeping-Goeteborg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(4.1781#16.7697))
%33
addTrain(routeName:'Falkoeping-Naessjoe'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday( 0#8.2819))
%34
addTrain(routeName:'Naessjoe-Falkoeping'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday( 0#6.8799))
%35
addTrain(routeName:'Skoevde-Goeteborg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0#18.5963))
%36
addTrain(routeName:'Goeteborg-Stockholm'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(4.7397#15.2996))
%37
addTrain(routeName:'Stockholm-Skoevde'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(14.8207#24.0000))
%38
```

```
addTrain(routeName:'Skoevde-Karlstad'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0.5748#11.2831))
%39
addTrain(routeName:'Norrkoeping-Naessjoe'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(13.5101#23.2817))
%40
addTrain(routeName:'Mjoelby-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(7.2428#16.7911))
%41
addTrain(routeName:'Uppsala-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(6.3479#18.5841))
%42
addTrain(routeName:'Norrkoeping-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(10.7537#22.2114))
%43
addTrain(routeName:'Hallsberg-Vaesteraas'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(11.4994#24.0000))
%44
addTrain(routeName:'Charlottenberg-Trollhaettan'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(16.9452#24.0000))
%45
addTrain(routeName:'Trollhaettan-Karlstad'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(6.1390#16.7922))
%46
addTrain(routeName:'Goeteborg-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(8.1515#16.0656))
%47
addTrain(routeName:'Karlstad-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(3.6377#14.8011))
%48
addTrain(routeName:'Norrkoeping-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(10.3490#24.0000))
%49
addTrain(routeName:'Uppsala-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(10.8341#20.4368))
%50
addTrain(routeName:'Uppsala-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(10.8341#17.4368))
%51
addTrain(routeName:'Uppsala-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0#13.8113))
%52
addTrain(routeName:'Norrkoeping-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(12.3662#22.3215))
%53
addTrain(routeName:'Hallsberg-Vaesteraas'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0#12.5370))
%54
addTrain(routeName:'Hallsberg-Uppsala'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(5.7556#19.0264))
%55
addTrain(routeName:'Vaesteraas-Katrineholm'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0#13.77))
%56
addTrain(routeName:'Vaesteraas-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0#7.3579))
```

```
%57
addTrain(routeName:'Charlottenberg-Trollhaettan'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(16.6890#18.8714))
%58
addTrain(routeName:'Trollhaettan-Karlstad'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0#24.0000))
%59
addTrain(routeName:'Goeteborg-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(7.9401#12.0846))
%60
addTrain(routeName:'Karlstad-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0#24.0000))
%61
addTrain(routeName:'Norrkoeping-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0#15.7134))
%62
addTrain(routeName:'Falkoeping-Naessjoe'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(18.0595#24.0000))
%63
addTrain(routeName:'Naessjoe-Falkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0#9.0264))
%64
addTrain(routeName:'Skoevde-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(12.3389#24.0000))
%65
addTrain(routeName:'Goeteborg-Stockholm'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(14.3487#20.9832))
%66
addTrain(routeName:'Stockholm-Skoevde'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(6.4234#24.0000))
%67
addTrain(routeName:'Skoevde-Karlstad'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(7.5110#9.7338))
%68
addTrain(routeName:'Norrkoeping-Naessjoe'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(16.1805#24.0000))
%69
addTrain(routeName:'Hallsberg-Karlstad'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(6.0108#19.4171))
%70
addTrain(routeName:'Mjoelby-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0#7.3579))
%71
addTrain(routeName:'Uppsala-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(18.8714#24.0000))
%72
addTrain(routeName:'Uppsala-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0#7.9401))
%73
addTrain(routeName:'Norrkoeping-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(12.0846#24.0000))
%74
addTrain(routeName:'Hallsberg-Vaesteraas'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0#15.7134))
%75
addTrain(routeName:'Hallsberg-Uppsala'
         typeOfEngine:rc6
         meanSpeed:120.0
```

```
        depTimeSpec:monday(0#16.6890))
%76
addTrain(routeName:'Vaesteraas-Katrineholm'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(18.0595#24.0000))
%77
addTrain(routeName:'Vaesteraas-Norrkoeping'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0#9.0264))
%78
addTrain(routeName:'Charlottenberg-Trollhaettan'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(12.3389#24.0000))
%79
addTrain(routeName:'Trollhaettan-Karlstad'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(14.3487#20.9832))
%80
addTrain(routeName:'Goeteborg-Hallsberg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(6.4234#24.0000))
%81
addTrain(routeName:'Karlstad-Goeteborg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(7.5110#9.7338))
%82
addTrain(routeName:'Norrkoeping-Goeteborg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(16.1805#24.0000))
%83
addTrain(routeName:'Falkoeping-Naessjoe'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(6.0108#19.4171))
%84
addTrain(routeName:'Naessjoe-Falkoeping'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(2.8807#19.4127))
%85
addTrain(routeName:'Skoevde-Goeteborg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(8.3198# 21.8982))
%86
addTrain(routeName:'Goeteborg-Stockholm'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(14.2238#24.0000))
%87
addTrain(routeName:'Stockholm-Skoevde'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(16.2835#24.0000))
%88
addTrain(routeName:'Skoevde-Karlstad'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(10.3906#24.0000))
%89
addTrain(routeName:'Norrkoeping-Naessjoe'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(8.8556#10.9243))
%90
addTrain(routeName:'Mjoelby-Hallsberg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(9.6615#24.0000))
%91
addTrain(routeName:'Uppsala-Norrkoeping'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(11.8789#24.0000))
%92
addTrain(routeName:'Norrkoeping-Hallsberg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(12.3812#24.0000))
%93
addTrain(routeName:'Hallsberg-Vaesteraas'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(14.4822#24.0000))
%94
addTrain(routeName:'Naessjoe-Falkoeping'
        typeOfEngine:rc6
```

```
         meanSpeed:120.0
         depTimeSpec:monday(0#11.3218))
 %95
 addTrain(routeName:'Charlottenberg-Trollhaettan'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(2.5162#10.4111))
 %96
 addTrain(routeName:'Trollhaettan-Karlstad'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(8.4479#24.0000))
 %97
 addTrain(routeName:'Goeteborg-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(15.9658#24.0000))
 %98
 addTrain(routeName:'Karlstad-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(1.6625#18.2764))
 %99
 addTrain(routeName:'Norrkoeping-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(12.0193#24.0000))
]
```

A.2 Performance Example B

A.2.1 Locomotive Start Locations

```
locomStartV:s(1:'Luleaa' 2:'Stockholm' 3:'Umeaa' 4:'Goeteborg'
              5:'Malmoe' 6:'Oestersund' 7:'Lund' 8:'Karlstad'
              9:'Stockholm' 10:'Luleaa' 11:'Goeteborg' 12:'Malmoe'
              13:'Helsingborg' 14:'Stockholm' 15:'Skoevde')
```

A.2.2 Routes

```
routeName:'Goeteborg-Karlstad'
Goe-Sk-Ka
routeName:'Hallsberg-Karlstad'
Ha-Ka
routeName:'Helsingborg-Goeteborg'
He-Goe
routeName:'Luleaa-Malmoe'
Lu-Um-Oes-Ha-Lu-Ma
routeName:'Luleaa-Stockholm'
Lu-Um-Oes-St
routeName:'Luleaa-Umeaa'
Lu-Um
routeName:'Lund-Goeteborg'
Lu-He-Goe
routeName:'Skoevde-Stockholm'
Sk-Ha-St
routeName:'Stockholm-Goeteborg'
St-Ha-Sk-Goe
routeName:'Stockholm-Hallsberg'
St-Ha
routeName:'Stockholm-Helsingborg'
St-He
routeName:'Stockholm-Malmoe'
St-Lu-Ma
routeName:'Stockholm-Oestersund'
St-Oes
routeName:'Umeaa-Goeteborg'
Um-Oes-Ha-Sk-Goe
routeName:'Oestersund-Karlstad'
Oes-Ha-Ka
```

A.2.3 Train File

```
[%0
 addTrain(routeName:'Luleaa-Stockholm'
         typeOfEngine:rc6
         meanSpeed:120.0)
 %1
 addTrain(routeName:'Umeaa-Luleaa'
         typeOfEngine:rc6
         meanSpeed:120.0)
 %2
 addTrain(routeName:'Stockholm-Luleaa'
         typeOfEngine:rc6
         meanSpeed:120.0)
 %3
 addTrain(routeName:'Umeaa-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0)
```

```
%4
addTrain(routeName:'Stockholm-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0)
%5
addTrain(routeName:'Stockholm-Malmoe'
         typeOfEngine:rc6
         meanSpeed:120.0)
%6
addTrain(routeName:'Stockholm-Oestersund'
         typeOfEngine:rc6
         meanSpeed:120.0)
%7
addTrain(routeName:'Lund-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0)
%8
addTrain(routeName:'Oestersund-Stockholm'
         typeOfEngine:rc6
         meanSpeed:120.0)
%9
addTrain(routeName:'Luleaa-Umeaa'
         typeOfEngine:rc6
         meanSpeed:120.0)
%10
addTrain(routeName:'Stockholm-Malmoe'
         typeOfEngine:rc6
         meanSpeed:120.0)
%11
addTrain(routeName:'Luleaa-Stockholm'
         typeOfEngine:rc6
         meanSpeed:120.0)
%12
addTrain(routeName:'Stockholm-Luleaa'
         typeOfEngine:rc6
         meanSpeed:120.0)
%13
addTrain(routeName:'Skoevde-Stockholm'
         typeOfEngine:rc6
         meanSpeed:120.0)
%14
addTrain(routeName:'Luleaa-Stockholm'
         typeOfEngine:rc6
         meanSpeed:120.0)
%15
addTrain(routeName:'Goeteborg-Karlstad'
         typeOfEngine:rc6
         meanSpeed:120.0)
%16
addTrain(routeName:'Stockholm-Malmoe'
         typeOfEngine:rc6
         meanSpeed:120.0)
%17
addTrain(routeName:'Luleaa-Umeaa'
         typeOfEngine:rc6
         meanSpeed:120.0)
%18
addTrain(routeName:'Stockholm-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:120.0)
%19
addTrain(routeName:'Hallsberg-Karlstad'
         typeOfEngine:rc6
         meanSpeed:120.0)
%20
addTrain(routeName:'Luleaa-Malmoe'
         typeOfEngine:rc6
         meanSpeed:120.0)
%21
addTrain(routeName:'Umeaa-Luleaa'
         typeOfEngine:rc6
         meanSpeed:120.0)
%22
addTrain(routeName:'Umeaa-Luleaa'
         typeOfEngine:rc6
         meanSpeed:120.0)
%23
addTrain(routeName:'Oestersund-Karlstad'
         typeOfEngine:rc6
         meanSpeed:120.0)
%24
addTrain(routeName:'Karlstad-Oestersund'
         typeOfEngine:rc6
         meanSpeed:120.0)
%25
addTrain(routeName:'Stockholm-Malmoe'
         typeOfEngine:rc6
         meanSpeed:120.0)
%26
addTrain(routeName:'Malmoe-Stockholm'
         typeOfEngine:rc6
         meanSpeed:120.0)
%27
addTrain(routeName:'Umeaa-Luleaa'
```

```
        typeOfEngine:rc6
        meanSpeed:120.0)
%28
addTrain(routeName:'Stockholm-Helsingborg'
        typeOfEngine:rc6
        meanSpeed:120.0)
%29
addTrain(routeName:'Stockholm-Luleaa'
        typeOfEngine:rc6
        meanSpeed:120.0)
%30
addTrain(routeName:'Oestersund-Stockholm'
        typeOfEngine:rc6
        meanSpeed:120.0)
%31
addTrain(routeName:'Luleaa-Umeaa'
        typeOfEngine:rc6
        meanSpeed:120.0)
%32
addTrain(routeName:'Stockholm-Malmoe'
        typeOfEngine:rc6
        meanSpeed:120.0)
%33
addTrain(routeName:'Luleaa-Stockholm'
        typeOfEngine:rc6
        meanSpeed:120.0)
%34
addTrain(routeName:'Umeaa-Luleaa'
        typeOfEngine:rc6
        meanSpeed:120.0)
%35
addTrain(routeName:'Stockholm-Luleaa'
        typeOfEngine:rc6
        meanSpeed:120.0)
%36
addTrain(routeName:'Umeaa-Goeteborg'
        typeOfEngine:rc6
        meanSpeed:120.0)
%37
addTrain(routeName:'Helsingborg-Goeteborg'
        typeOfEngine:rc6
        meanSpeed:120.0)
%38
addTrain(routeName:'Stockholm-Malmoe'
        typeOfEngine:rc6
        meanSpeed:120.0)
%39
addTrain(routeName:'Stockholm-Oestersund'
        typeOfEngine:rc6
        meanSpeed:120.0)
%40
addTrain(routeName:'Oestersund-Karlstad'
        typeOfEngine:rc6
        meanSpeed:120.0)
%41
addTrain(routeName:'Karlstad-Oestersund'
        typeOfEngine:rc6
        meanSpeed:120.0)
%42
addTrain(routeName:'Stockholm-Malmoe'
        typeOfEngine:rc6
        meanSpeed:120.0)
%43
addTrain(routeName:'Malmoe-Stockholm'
        typeOfEngine:rc6
        meanSpeed:120.0)
%44
addTrain(routeName:'Umeaa-Luleaa'
        typeOfEngine:rc6
        meanSpeed:120.0)
%45
addTrain(routeName:'Stockholm-Malmoe'
        typeOfEngine:rc6
        meanSpeed:120.0)
%46
addTrain(routeName:'Luleaa-Stockholm'
        typeOfEngine:rc6
        meanSpeed:120.0)
%47
addTrain(routeName:'Stockholm-Luleaa'
        typeOfEngine:rc6
        meanSpeed:120.0)
%48
addTrain(routeName:'Skoevde-Stockholm'
        typeOfEngine:rc6
        meanSpeed:120.0)
%49
addTrain(routeName:'Luleaa-Stockholm'
        typeOfEngine:rc6
        meanSpeed:120.0)
]
```

A.3 Small Example

```
[
 %0
 addTrain(routeName:'Charlottenberg-Kornsjoe'
          typeOfEngine:rc6
          meanSpeed:120.0
          depTimeSpec:monday(0#2)
         )
 %1
 addTrain(routeName:'Charlottenberg-Kornsjoe'
          typeOfEngine:rc6
          meanSpeed:120.0
          depTimeSpec:monday(0#2)
         )
 %2
 addTrain(routeName:'Kornsjoe-Goeteborg'
          typeOfEngine:rc6
          meanSpeed:120.0
          depTimeSpec:monday(4#7)
         )
 %3
 addTrain(routeName:'Goeteborg-Laxaa'
          typeOfEngine:rc6
          meanSpeed:120.0
          depTimeSpec:monday(0#2)
         )
 %4
 addTrain(routeName:'Goeteborg-Charlottenberg'
          typeOfEngine:rc6
          meanSpeed:120.0
          depTimeSpec:monday(4#10)
         )
 %5
 addTrain(routeName:'Laxaa-Mellerud'
          typeOfEngine:rc6
          meanSpeed:120.0
          depTimeSpec:monday(4#7)
         )
 %6
 addTrain(routeName:'Laxaa-Kornsjoe'
          typeOfEngine:rc6
          meanSpeed:120.0
          depTimeSpec:monday(0#16)
         )
]
```

A.4 Larger Example

```
[
 %0
 addTrain(routeName:'Uppsala-Norrkoeping'
          typeOfEngine:rc6
          meanSpeed:120.0
         )
 %1
 addTrain(routeName:'Uppsala-Norrkoeping'
          typeOfEngine:rc6
          meanSpeed:100.0
         )
 %2
 addTrain(routeName:'Norrkoeping-Hallsberg'
          typeOfEngine:rc6
          meanSpeed:80.0
         )
 %3
 addTrain(routeName:'Hallsberg-Vaesteraas'
          typeOfEngine:rc6
          meanSpeed:100.0
         )
 %4
 addTrain(routeName:'Hallsberg-Uppsala'
          typeOfEngine:rc6
          meanSpeed:120.0
         )
 %5
 addTrain(routeName:'Vaesteraas-Katrineholm'
          typeOfEngine:rc6
          meanSpeed:80.0
         )
 %6
 addTrain(routeName:'Vaesteraas-Norrkoeping'
          typeOfEngine:rc6
          meanSpeed:120.0
         )
 %7
 addTrain(routeName:'Charlottenberg-Trollhaettan'
          typeOfEngine:rc6
          meanSpeed:100.0
         )
 %8
 addTrain(routeName:'Trollhaettan-Karlstad'
```

```
         typeOfEngine:rc6
         meanSpeed:80.0
        )
%9
addTrain(routeName:'Goeteborg-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:100.0
        )
%10
addTrain(routeName:'Karlstad-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0
        )
%11
addTrain(routeName:'Norrkoeping-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:80.0
        )
%12
addTrain(routeName:'Falkoeping-Naessjoe'
         typeOfEngine:rc6
         meanSpeed:120.0
        )
%13
addTrain(routeName:'Naessjoe-Falkoeping'
         typeOfEngine:rc6
         meanSpeed:100.0
        )
%14
addTrain(routeName:'Skoevde-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:80.0
        )
%15
addTrain(routeName:'Goeteborg-Stockholm'
         typeOfEngine:rc6
         meanSpeed:100.0
        )
%16
addTrain(routeName:'Stockholm-Skoevde'
         typeOfEngine:rc6
         meanSpeed:120.0
        )
%17
addTrain(routeName:'Skoevde-Karlstad'
         typeOfEngine:rc6
         meanSpeed:100.0
        )
%18
addTrain(routeName:'Norrkoeping-Naessjoe'
         typeOfEngine:rc6
         meanSpeed:80.0
        )
%19
addTrain(routeName:'Hallsberg-Karlstad'
         typeOfEngine:rc6
         meanSpeed:100.0
        )
%20
addTrain(routeName:'Mjoelby-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:120.0
        )
%21
addTrain(routeName:'Uppsala-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:80.0
        )
%22
addTrain(routeName:'Uppsala-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
        )
%23
addTrain(routeName:'Norrkoeping-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:100.0
        )
%24
addTrain(routeName:'Hallsberg-Vaesteraas'
         typeOfEngine:rc6
         meanSpeed:80.0
        )
%25
addTrain(routeName:'Hallsberg-Uppsala'
         typeOfEngine:rc6
         meanSpeed:100.0
        )
%26
addTrain(routeName:'Vaesteraas-Katrineholm'
         typeOfEngine:rc6
         meanSpeed:120.0
        )
%27
```

```
addTrain(routeName:'Vaesteraas-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:80.0
        )
%28
addTrain(routeName:'Charlottenberg-Trollhaettan'
         typeOfEngine:rc6
         meanSpeed:120.0
        )
%29
addTrain(routeName:'Trollhaettan-Karlstad'
         typeOfEngine:rc6
         meanSpeed:100.0
        )
%30
addTrain(routeName:'Goeteborg-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:80.0
        )
%31
addTrain(routeName:'Karlstad-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:100.0
        )
%32
addTrain(routeName:'Norrkoeping-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0
        )
%33
addTrain(routeName:'Falkoeping-Naessjoe'
         typeOfEngine:rc6
         meanSpeed:80.0
        )
%34
addTrain(routeName:'Naessjoe-Falkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
        )
%35
addTrain(routeName:'Naessjoe-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:100.0
        )
%36
addTrain(routeName:'Goeteborg-Stockholm'
         typeOfEngine:rc6
         meanSpeed:80.0
        )
%37
addTrain(routeName:'Stockholm-Skoevde'
         typeOfEngine:rc6
         meanSpeed:100.0
        )
%38
addTrain(routeName:'Skoevde-Karlstad'
         typeOfEngine:rc6
         meanSpeed:120.0
        )
%39
addTrain(routeName:'Norrkoeping-Naessjoe'
         typeOfEngine:rc6
         meanSpeed:100.0
        )
]
```

A.5 Larger Example with Time Windows

```
[
 %0
 addTrain(routeName:'Norrkoeping-Hallsberg'
          typeOfEngine:rc6
          meanSpeed:100.0
          depTimeSpec:monday(0#2)
         )
 %1
 addTrain(routeName:'Norrkoeping-Hallsberg'
          typeOfEngine:rc6
          meanSpeed:80.0
          depTimeSpec:monday(0#2)
         )
 %2
 addTrain(routeName:'Hallsberg-Vaesteraas'
          typeOfEngine:rc6
          meanSpeed:80.0
          depTimeSpec:monday(0#4)
         )
 %3
 addTrain(routeName:'Hallsberg-Vaesteraas'
          typeOfEngine:rc6
          meanSpeed:100.0
          depTimeSpec:monday(0#4)
```

```
        )
%4
addTrain(routeName:'Vaesteraas-Katrineholm'
         typeOfEngine:rc6
         meanSpeed:80.0
         depTimeSpec:monday(2#6)
        )
%5
addTrain(routeName:'Charlottenberg-Trollhaettan'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(4#6)
        )
%6
addTrain(routeName:'Vaesteraas-Katrineholm'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(2#6)
        )
%7
addTrain(routeName:'Charlottenberg-Trollhaettan'
         typeOfEngine:rc6
         meanSpeed:100.0
         depTimeSpec:monday(4#6)
        )
%8
addTrain(routeName:'Uppsala-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:100.0
         depTimeSpec:monday(7#7)
)

%9
addTrain(routeName:'Vaesteraas-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:80.0
         depTimeSpec:monday(2#8)
)

%10
addTrain(routeName:'Uppsala-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(7#8)
        )
%11
addTrain(routeName:'Uppsala-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:80.0
         depTimeSpec:monday(0#8)
        )
%12
addTrain(routeName:'Vaesteraas-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(4#8)
        )
%13
addTrain(routeName:'Uppsala-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(0#8)
        )
%14
addTrain(routeName:'Naessjoe-Falkoeping'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(8#10)
        )
%15
addTrain(routeName:'Goeteborg-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:80.0
         depTimeSpec:monday(6#10)
        )
%16
addTrain(routeName:'Hallsberg-Uppsala'
         typeOfEngine:rc6
         meanSpeed:100.0
         depTimeSpec:monday(2#10)
        )
%17
addTrain(routeName:'Hallsberg-Uppsala'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(2#10)
        )
%18
addTrain(routeName:'Naessjoe-Falkoeping'
         typeOfEngine:rc6
         meanSpeed:100.0
         depTimeSpec:monday(8#10)
        )
%19
```

```
addTrain(routeName:'Goeteborg-Hallsberg'
         typeOfEngine:rc6
         meanSpeed:100.0
         depTimeSpec:monday(6#10)
        )
%20
addTrain(routeName:'Goeteborg-Stockholm'
         typeOfEngine:rc6
         meanSpeed:80.0
         depTimeSpec:monday(6#12)
        )
%21
addTrain(routeName:'Norrkoeping-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(8#12)
        )
%22
addTrain(routeName:'Goeteborg-Stockholm'
         typeOfEngine:rc6
         meanSpeed:100.0
         depTimeSpec:monday(4#12)
        )
%23
addTrain(routeName:'Norrkoeping-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:80.0
         depTimeSpec:monday(8#12)
        )
%24
addTrain(routeName:'Trollhaettan-Karlstad'
         typeOfEngine:rc6
         meanSpeed:80.0
         depTimeSpec:monday(13#13)
        )
%25
addTrain(routeName:'Norrkoeping-Naessjoe'
         typeOfEngine:rc6
         meanSpeed:100.0
         depTimeSpec:monday(12#14)
        )
%26
addTrain(routeName:'Stockholm-Skoevde'
         typeOfEngine:rc6
         meanSpeed:100.0
         depTimeSpec:monday(10#14)
        )
%27
addTrain(routeName:'Naessjoe-Norrkoeping'
         typeOfEngine:rc6
         meanSpeed:100.0
         depTimeSpec:monday(10#14)
        )
%28
addTrain(routeName:'Trollhaettan-Karlstad'
         typeOfEngine:rc6
         meanSpeed:100.0
         depTimeSpec:monday(14#14)
        )
%29
addTrain(routeName:'Norrkoeping-Naessjoe'
         typeOfEngine:rc6
         meanSpeed:80.0
         depTimeSpec:monday(12#14)
        )
%30
addTrain(routeName:'Stockholm-Skoevde'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(10#14)
        )
%31
addTrain(routeName:'Skoevde-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:80.0
         depTimeSpec:monday(10#14)
        )
%32
addTrain(routeName:'Skoevde-Karlstad'
         typeOfEngine:rc6
         meanSpeed:120.0
         depTimeSpec:monday(12#16)
        )
%33
addTrain(routeName:'Falkoeping-Naessjoe'
         typeOfEngine:rc6
         meanSpeed:80.0
         depTimeSpec:monday(0#16)
        )
%34
addTrain(routeName:'Karlstad-Goeteborg'
         typeOfEngine:rc6
         meanSpeed:100.0
         depTimeSpec:monday(0#16)
```

```
    )
%35
addTrain(routeName:'Mjoelby-Hallsberg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(12#16)
        )
%36
addTrain(routeName:'Hallsberg-Karlstad'
        typeOfEngine:rc6
        meanSpeed:100.0
        depTimeSpec:monday(14#16)
        )
%37
addTrain(routeName:'Skoevde-Karlstad'
        typeOfEngine:rc6
        meanSpeed:100.0
        depTimeSpec:monday(12#16)
        )
%38
addTrain(routeName:'Karlstad-Goeteborg'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0#16)
        )
%39
addTrain(routeName:'Falkoeping-Naessjoe'
        typeOfEngine:rc6
        meanSpeed:120.0
        depTimeSpec:monday(0#16)
        )
]
```

Appendix B

Code

B.1 Exclusion Marker Model

```
% Exclusion Marker Model
fun {EngineDiff TaskRec TrainStartSlotsRec
     TurnLocV DistDict ResLimit Deps Wait}

   DurRec={Record.map TrainRec fun {$ Task} Task.tripTime end}
   StartRec=
   {Record.map TrainRec
    fun {$ Task}
       Deps.(TrainStartSlotsRec.(Task.id))
    end}
   Tasks={Arity TrainRec}
   ResRec={FD.record res Tasks 1#ResLimit}
   Resources={List.number 1 ResLimit 1}
   UseRec={FD.record res Tasks 1#1}
   PasDur={List.toRecord o  % Duration of passive transports
       {Map {Dictionary.entries DistDict}
        fun {$ O#Sub}
          O#
          {List.toRecord d
           {Map {Dictionary.entries Sub}
            fun {$ D#Dist}
               D#{self hours(Dist/PasVelocity time:$)}
            end}}
       end}}
   LocomStartV
in
   {self getLocomStartV(LocomStartV)}

   %% One global diff2 for the trip rectangles
   {Diff2 StartRec DurRec ResRec UseRec}

   %% Return
   StartRec#
   ResRec#
   %% and list of offsets from
   %% Simonis model for location continuity
   {FoldR TurnLocV
    fun {$ Loc RetL}
       Xdict={NewDictionary}             % Store of rectangle x origins
       Ldict={NewDictionary}             % Store of rectangle x sizes
       Ydict={NewDictionary}             % Store of rectangle y origins
       Hdict={NewDictionary}             % Store of rectangle y sizes
       Offs={FD.record offs Tasks 0#K-1} % Local excl. marker offsets
    in
    {ForAll Tasks
    proc {$ Task}
       Y={FD.decl}
    in
       {FD.sumAC [K] [ResRec.Task] ':' Y}
       case Loc==TaskRec.Task.orig then     % Loc is origin of task
          %% Determine position & size of start marker
          {Dictionary.put Xdict                      % Start
           {ToAtom 'S '#Task} StartRec.Task}
          {Dictionary.put Ldict {ToAtom 'S '#Task} 1}
          {Dictionary.put Ydict {ToAtom 'S '#Task} Y}
          {Dictionary.put Hdict {ToAtom 'S '#Task} K}
          %% Determine position & size of exlusion marker
          {Dictionary.put Xdict                      % After
           {ToAtom 'A '#Task} {FD.plus StartRec.Task DurRec.Task}}
          {Dictionary.put Ldict
           {ToAtom 'A '#Task} PasDur.(TaskRec.Task.dest).Loc}
          {Dictionary.put Ydict
           {ToAtom 'A '#Task} {FD.plus Y Offs.Task}}
          {Dictionary.put Hdict {ToAtom 'A '#Task} 1}
       elsecase Loc==TaskRec.Task.dest then
          %% Loc is destination of task
          %% Determine position & size of exlusion marker
          {Dictionary.put Xdict                      % Before
           {ToAtom 'B '#Task}
           {FD.minus StartRec.Task
            PasDur.Loc.(TaskRec.Task.orig)}}
```

```
071          {Dictionary.put Ldict
072           {ToAtom 'B '#Task} PasDur.Loc.(TaskRec.Task.orig)}
073          {Dictionary.put Ydict
074           {ToAtom 'B '#Task} {FD.plus Y Offs.Task}}
075          {Dictionary.put Hdict {ToAtom 'B '#Task} 1}
076          %% Determine position & size of end marker
077          {Dictionary.put Xdict                      % End
078           {ToAtom 'E '#Task}
079           {FD.plus StartRec.Task DurRec.Task-1}}
080          {Dictionary.put Ldict {ToAtom 'E '#Task} 1}
081          {Dictionary.put Ydict {ToAtom 'E '#Task} Y}
082          {Dictionary.put Hdict {ToAtom 'E '#Task} K}
083       else % Other location
084          %% Determine position & size of first exlusion marker
085          {Dictionary.put Xdict                      % Before
086           {ToAtom 'B '#Task}
087           {FD.minus StartRec.Task
088            PasDur.Loc.(TaskRec.Task.orig)}}
089          {Dictionary.put Ldict
090           {ToAtom 'B '#Task} PasDur.Loc.(TaskRec.Task.orig)}
091          {Dictionary.put Ydict
092           {ToAtom 'B '#Task} {FD.plus Y Offs.Task}}
093          {Dictionary.put Hdict {ToAtom 'B '#Task} 1}
094          %% Determine position & size of second exlusion marker
095          {Dictionary.put Xdict                      % After
096           {ToAtom 'A '#Task} {FD.plus StartRec.Task DurRec.Task}}
097          {Dictionary.put Ldict
098           {ToAtom 'A '#Task} PasDur.(TaskRec.Task.dest).Loc}
099          {Dictionary.put Ydict
100           {ToAtom 'A '#Task} {FD.plus Y Offs.Task}}
101          {Dictionary.put Hdict {ToAtom 'A '#Task} 1}
102       end
103    end}
104
105    % place markers for start position of locomotives
106    {ForAll Resources
107    proc {$ R}
108       Y=K*R
109    in    % Y=K*ri
110       case Loc==LocomStartV.R then
111          skip          % Loc is start location of resource
112       else
113          {Dictionary.put Xdict    % start trip marker
114           {ToAtom 'I '#R} 0}
115          {Dictionary.put Ydict
116           {ToAtom 'I '#R} {FD.plus Y {FD.int 0#K-1}}}
117          {Dictionary.put Ldict
118           {ToAtom 'I '#R} PasDur.(LocomStartV.R).Loc}
119          {Dictionary.put Hdict
120           {ToAtom 'I '#R} 1}
121       end
122    end
123    }
124
125    %% Post Diff2 for location continuity
126    {Diff2
127     {Dictionary.toRecord x Xdict}
128     {Dictionary.toRecord l Ldict}
129     {Dictionary.toRecord y Ydict}
130     {Dictionary.toRecord h Hdict}}
131
132    %% Return
133    locM(loc:Loc
134         x:{Dictionary.toRecord x Xdict}
135         l:{Dictionary.toRecord l Ldict}
136         y:{Dictionary.toRecord y Ydict}
137         h:{Dictionary.toRecord h Hdict}
138        )
139      |RetL
140   end nil}
141 end                          % EngineDiff
```

B.2 Insertion Heuristic

```
000 % insertion heuristic
001 {FD.distribute
002  generic(order:
003          fun {$ Task1 Task2}
004             case {Not {IsNumber Task1}} then
005                false
006             elsecase {Not {IsNumber Task2}} then
007                true
008             else
009                {FD.reflect.max (ResRecs.1).Task1}
010                =<
011                {FD.reflect.max (ResRecs.1).Task2}
012             end
013          end     % order
014          filter:
015          fun {$ Task}
016             case {Not {IsNumber Task}} then
017                true
```

```
018            else
019               {FD.reflect.size (ResRecs.2).Task} > 1
020            end
021         end     % filter
022         value:
023         proc {$ Task RosterLists|Ts Cont}
024
025           % find insert positions, sorted by costs
026           % returns [Res1#Pred1#Succ1#Cost1 Res2#Pred2#Succ2# ...]
027            proc {FindPos Task ?Pos}
028               RLList = {Record.toListInd RosterLists}
029
030               % returns [Res1#Pred1#Succ1#Cost1 ...]
031               % costi: cost for insertion after ti
032               proc {PassiveCosts Resource RL ?PC}
033                  case RL of T1|T2|Rest then % put Task between T1
034                     PC = [Resource#T1#T2    % and T2
035                      #(DistVv.(TrainRec.T1.dest).(TrainRec.Task.orig)
036                       +DistVv.(TrainRec.Task.dest).(TrainRec.T2.orig)
037                       -DistVv.(TrainRec.T1.dest).(TrainRec.T2.orig))
038                          ]
039                     |{PassiveCosts Resource T2|Rest}
040                  elseof [T] then % put Task after last task in RL
041                     PC = [Resource#T#nosucc
042                     #(DistVv.(TrainRec.T.dest).(TrainRec.Task.orig))]
043                  else
044                     PC = nil
045                  end
046               end   % PassiveCosts
047
048               CostVv=     % [Res1#Pred1#Succ1#Cost1 ...]
049               {Map RLList % for all insert positions
050                proc {$ Resource#RosterList ?CL}
051                   case RosterList of
052                      T1|Rest then % insert Task at beginning
053                      CL = [Resource#nopred#T1#
054                     (DistVv.(LocomStartV.Resource).(TrainRec.Task.orig)
055                     +DistVv.(TrainRec.Task.dest).(TrainRec.T1.orig)
056                     -DistVv.(LocomStartV.Resource).(TrainRec.T1.orig))
057                             ]
058                      |{PassiveCosts Resource RosterList}
059                  else % RosterList empty: insert as first task
060                     CL = [Resource#nopred#nosucc#
061                     (DistVv.(LocomStartV.Resource).(TrainRec.Task.orig)
062                            +Lambda)]   % add cost for new locomotive
063                  end
064                end}
065
066         in
067            Pos = {Map    % [Res1#Pred1#Succ1 ...] sorted after costs
068                   {Sort {Flatten CostVv}
069                    fun {$ _#_#_#C1 _#_#_#C2} C1 =< C2 end}
070                   fun {$ R#P#S#_} R#P#S end}
071         end   % FindPos
072
073         % inserts Task between Pred and Succ in PosList
074         proc {TryPos PosList}
075            NewRosterList
076         in
077            case PosList
078            of (Resource#Pred#Succ)|RestList then
079               choice
080                 % post constraints on resource usage and task order
081                  (ResRecs.2).Task =: Resource
082
083                  case {Not Pred==nopred} then
084                     Offset1 = {FD.reflect.max Dur.Pred}
085                     +DistVv.(TrainRec.Pred.dest).(TrainRec.Task.orig)
086                  in
087                     (ResRecs.1).Task
088                     >=: {FD.plus (ResRecs.1).Pred Offset1}
089                  else
090                    % passive transport from start location
091                     (ResRecs.1).Task
092                     >=: DistVv.(LocomStartV.Resource).(TrainRec.Task.orig)
093                  end
094
095                  case {Not Succ==nosucc} then
096                     Offset2 = {FD.reflect.max Dur.Task}
097                     +DistVv.(TrainRec.Task.dest).(TrainRec.Succ.orig)
098                  in
099                     (ResRecs.1).Succ
100                     >=: {FD.plus (ResRecs.1).Task Offset2}
101                  else
102                     skip
103                  end
104
105                  % update RosterLists
106                  case RosterLists.Resource==nil then
107                     NewRosterList =        % assign new locomotive
108                     {AdjoinAt RosterLists Resource
109                      [Task]}
110                  elsecase Pred==nopred then % insert as first
111                     NewRosterList =
                        {AdjoinAt RosterLists Resource
```

```
112                      Task|RosterLists.Resource}
113                   elsecase Succ==nosucc then % insert as last
114                      NewRosterList =
115                      {AdjoinAt RosterLists Resource
116                       {Append RosterLists.Resource [Task]}}
117                   else                % insert into a roster list
118                      NewRosterList =
119                      {AdjoinAt RosterLists Resource
120                       {InsertAfter Task Pred RosterLists.Resource}}
121                   end
122
123                   % recursion
124                   choice
125                      {Cont NewRosterList|Ts}
126                   end
127                [] % try next best insertion
128                   {TryPos RestList}
129                end
130             else  % all insertions tried: trip can't be inserted
131                {Browse 'insertion failed'#Task}
132                choice
133                   (ResRecs.2).Task =: 1
134                   {Cont RosterLists|Ts}
135                end
136             end
137          end   % TryPos
138       in
139          case {Not {IsNumber Task}} then
140             skip
141          else
142             {TryPos {FindPos Task}}
143          end
144       end   % value
145    )
146 RosterLists|Tasks2
147 }
148
149 {FD.distribute generic(order:nbSusps value:splitMin) Deps}
```

B.3 Diff2 Propagator

B.3.1 Diff2Prop.h

```
// filename: Diff2Prop.h
// author:   Volker Scholz

#ifndef DIFF2PROP_H
#define DIFF2PROP_H

#include "oz_cpi.hh"
#include "Rectangle.h"
#include "DomainSet.h"
#include "BigSet.h"
#include "Pair.h"
#include "Domain.h"

void fd_start(void) {;}          // Start point for debugger

typedef BigSet<Pair> PairSet;

class Diff2Prop : public OZ_Propagator
{
public:
  ~Diff2Prop();

  // constructor with Oz parameters
  Diff2Prop(OZ_Term d);

  // starts propagation
  virtual OZ_Return propagate(void);

  // returns C header function
  virtual OZ_CFun getHeaderFunc(void) const;

  // returns size of propagator object
  virtual size_t sizeOf(void);

  // returns list of propagator parameters
  virtual OZ_Term getParameters(void) const;

  // copies heap data structures
  virtual void updateHeapRefs(OZ_Boolean);

private:
  static OZ_CFun header;

  int _size;       // number of rectangles
  OZ_Term* _w;     // height, width of rectangles
  OZ_Term* _h;
  OZ_Term* _x;     // xpos, ypos of rectangles
  OZ_Term* _y;

  int* oldXSize;
  int* oldYSize;  // domain sizes after last propagation
  int* oldWSize;
  int* oldHSize;

  DomainSet** domainset; // array of pointers to rectangle domainsets
  PairSet* pset;         // sets of pairs for AC3, temporary
  bool firstInvocation;  // flag for number of invocations of ::propagate

  OZ_Return localPropagation(); // AC3-algorithm
  OZ_Return globalPropagation(); // area sum heuristic
  void getSubBoundingBox(int j, Rectangle& box);
  void getBoundingBox(int no, Rectangle& box);
      // get bounding box of rectangle domain number no
  int getSubSumAreas(int j);
};

#endif DIFF2PROP_H
```

B.3.2 Diff2Prop.cc

```
// filename: Diff2Prop.cc
// author:   Volker Scholz

#include "Diff2Prop.h"
#include "VectorExpect.h"
#include "Iterator_OZ_FDIntVar.h"
#include <stdio.h>
#include <stdlib.h>
#include <new.h>
#include <limits.h>

#define INIT_DOM_SET_SIZE 5

typedef DomainSet* DomainSetPtr;

OZ_Return Diff2Prop::propagate(void)
{
  OZ_FDIntVar x[_size], y[_size];    // FD variables for bottom left
  Iterator_OZ_FDIntVar X(_size, x);  // corners
  Iterator_OZ_FDIntVar Y(_size, y);  // Iterators for FD variables

  OZ_FDIntVar w[_size], h[_size];    // FD variables rectangle size
  Iterator_OZ_FDIntVar W(_size, w);  // Iterators for FD variables
  Iterator_OZ_FDIntVar H(_size, h);

  // update data structures according to new domains
  for(int i = 0; i < _size; i++)
    {
      x[i].read(_x[i]);
      y[i].read(_y[i]);
      w[i].read(_w[i]);
      h[i].read(_h[i]);

      // get domain boundaries
      int xmin = x[i]->getMinElem();
      int xmax = x[i]->getMaxElem();
      int ymin = y[i]->getMinElem();
      int ymax = y[i]->getMaxElem();
      int wmin = w[i]->getMinElem();
      int wmax = w[i]->getMaxElem();
      int hmin = h[i]->getMinElem();
      int hmax = h[i]->getMaxElem();

      // first invocation of ::propagate: insert one rectangular domain
      if (firstInvocation)
        {
          Domain d;

          d.dom =
            Rectangle(xmin, xmax+wmax-1, ymin, ymax+hmax-1);
```

```
        domainset[i]->Insert(d);
        oldXSize[i] = x[i]->getSize();
        oldYSize[i] = y[i]->getSize();
        oldWSize[i] = w[i]->getSize();
        oldHSize[i] = h[i]->getSize();
      }
    else  // later invocation: update domainsets for changed rectangles
      {
        if (x[i]->getSize() < oldXSize[i]
            || y[i]->getSize() < oldYSize[i]
            || w[i]->getSize() < oldWSize[i]
            || h[i]->getSize() < oldHSize[i])
          {
            Rectangle box =
              Rectangle(xmin, xmax+wmax-1, ymin, ymax+hmax-1);
            domainset[i]->setBoundingBox(box);
          }

        if (w[i]->getSize() < oldWSize[i]
            || h[i]->getSize() < oldHSize[i])
          {
            domainset[i]->setMinWidth(wmin);
            domainset[i]->setMaxWidth(wmax);
            domainset[i]->setMinHeight(hmin);
            domainset[i]->setMaxHeight(hmax);
            domainset[i]->removeTooSmallDomains();
          }
      }
  }

// if one of the updated domainsets is empty:
// corresponding rectangle can't be positioned, failure
for (int i = 0; i < _size; i++)
  if (domainset[i]->isEmpty()) goto failure;

// call AC3-propagation
OZ_Return r;
r = localPropagation();

// area sum heuristic
globalPropagation();

// update data structures after propagation
for (int i = 0; i < _size; i++)
  {
    // update domain sizes after propagation
    x[i].read(_x[i]);
    y[i].read(_y[i]);

    oldXSize[i] = x[i]->getSize();
    oldYSize[i] = y[i]->getSize();

    // update domains after propagation
    OZ_FiniteDomain aux_x(fd_empty);
    OZ_FiniteDomain aux_y(fd_empty);

    // empty domainset: failure
    if (domainset[i]->isEmpty()) goto failure;
    Rectangle box;
    domainset[i]->computeBoundingBox(box);

    int wmax = domainset[i]->getMaxWidth();
    int hmax = domainset[i]->getMaxHeight();
    aux_x.initRange(box.getXmin(), box.getXmax()-wmax+1);
    aux_y.initRange(box.getYmin(), box.getYmax()-hmax+1);

    *x[i] &= aux_x;
    *y[i] &= aux_y;
  }

 // toggle flag for first invocation
 if (firstInvocation)
   firstInvocation = false;

 // r = return value from AC3
 if (r==OZ_ENTAILED)
   {
     X.leave();
     Y.leave();
     W.leave();
     H.leave();
     return OZ_ENTAILED;
   }

 if (X.leave() | Y.leave() | W.leave() | H.leave())
   return OZ_SLEEP;
 else
   return OZ_ENTAILED;

 // failure
failure:
 X.fail();
 Y.fail();
 W.fail();
 H.fail();
 return OZ_FAILED;
}


OZ_Return Diff2Prop::localPropagation()
{
 // get new domain sizes
 OZ_FDIntVar x[_size], y[_size];
 OZ_FDIntVar w[_size], h[_size];

 for(int i = 0; i < _size; i++)
   {
     x[i].read(_x[i]);
     y[i].read(_y[i]);
     w[i].read(_w[i]);
     h[i].read(_h[i]);
   }

 // AC3
 pset->Clear();

 // insert all rectangle pairs into pset
 // where a forbidden area overlaps with a rectangle domain
 if (firstInvocation)  // consider all overlapping rectangle pairs
   {
     for (int i = 0; i < _size; i++)
       {
         bool x,y;
         domainset[i]->updateForbiddenAreas();
         Rectangle ri;
         domainset[i]->getNoOverlapArea(ri);
         if (ri.isEmpty())
           domainset[i]->getNoCoverAreas(ri, x, y);
```

```
        for (int j = 0; j < _size; j++)
          if (i!=j)
            {
              Domain d;
              domainset[j]->InitNext();
              domainset[j]->GetNextEntry(d);
                       // domainset has initially one entry
              Rectangle rj = d.dom;

              if (ri.intersects(rj))
                pset->Insert(Pair(j,i));
            }
      }
  }
else // consider overlaps of updated rectangles
  {
    for (int i = 0; i < _size; i++)
      {
        if (x[i]->getSize() < oldXSize[i]   // rectangle has changed
            || y[i]->getSize() < oldYSize[i]
            || w[i]->getSize() < oldWSize[i]
            || h[i]->getSize() < oldHSize[i])
          {
            bool x,y;
            if (w[i]->getSize() < oldWSize[i]
                || h[i]->getSize() < oldHSize[i])
              domainset[i]->updateKernels();

            domainset[i]->updateForbiddenAreas();
            Rectangle ri;
            domainset[i]->getNoOverlapArea(ri);

            if (ri.isEmpty())
              domainset[i]->getNoCoverAreas(ri, x, y);

            for (int j = 0; j < _size; j++)
              if (i!=j)
                {
                  Rectangle rj;
                  getBoundingBox(j,rj);

                  if (ri.intersects(rj))
                    pset->Insert(Pair(j,i));
                }
          }
      }
  }
// if no overlapping rectangle pairs exist: test if there
// are any overlaps between rectangle domains
// if not, the diff2 constraint is already fulfilled
if (pset->isEmpty())
  {
    bool inters = false;

    for (int i = 0; i < _size; i++)
      {
        Rectangle ri;
        getBoundingBox(i,ri);
        for (int j = i+1; j < _size; j++)
          {
            Rectangle rj;
            getBoundingBox(j,rj);

            if (ri.intersects(rj))
              {
                inters = true;
                goto end;
              }
          }
      }
  end:
    if (!inters)
      return OZ_ENTAILED;
  }

bool changed;
pset->InitNext();
Pair p;

while (pset->GetNextEntry(p))
  {
    int i = p.first;
    int j = p.second;
    changed = false;

    // remove non-overlappable areas
    Rectangle o;
    domainset[j]->getNoOverlapArea(o);
    int width = domainset[i]->getMinWidth();
    int height = domainset[i]->getMinHeight();

    if (!o.isEmpty())
      {
        Rectangle pos = Rectangle(o.getXmin()-width+1,
                                  o.getXmax()+width-1,
                                  o.getYmin()-height+1,
                                  o.getYmax()+height-1);
        bool changedNew = domainset[i]->removeDomain(pos);
        changed = changed || changedNew;
      }

    // remove non-coverable areas
    if (o.isEmpty())
      {
        bool Xsegmented, Ysegmented;
        Rectangle c;
        domainset[j]->getNoCoverAreas(c, Xsegmented, Ysegmented);

        if (!c.isEmpty())
          {
            Rectangle pos;
            bool removable = false;

            if (!Xsegmented && !Ysegmented
                && width >= c.getWidth() && height>=c.getHeight())
              {
                pos = Rectangle(c.getXmax()-width+1,
                                c.getXmin()+width-1,
                                c.getYmax()-height+1,
                                c.getYmin()+height-1);
                removable = true;
              }
            if (Xsegmented && height >= c.getHeight())
              {
                pos = Rectangle(c.getXmin()-width+1,
                                c.getXmax()+width-1,
                                c.getYmax()-height+1,
```

```
                                c.getYmin()+height-1);
                  removable = true;
                }

              if (Ysegmented && width >= c.getWidth())
                {
                  pos = Rectangle(c.getXmax()-width+1,
                                  c.getXmin()+width-1,
                                  c.getYmin()-height+1,
                                  c.getYmax()+height-1);
                  removable = true;
                }

              if (removable)
                {
                  bool changedNew = domainset[i]->removeDomain(pos);
                  changed = changed || changedNew;
                }
            }
        }

      // add new pairs if domain was reduced
      if (changed)
        {
          // update forbidden areas
          domainset[i]->updateForbiddenAreas();
          Rectangle ri;
          bool x,y;
          domainset[i]->getNoOverlapArea(ri);
          if (ri.isEmpty())
            domainset[i]->getNoCoverAreas(ri, x, y);

          for (int k = 0; k < _size; k++)
            {
              if (k!=i)
                {
                  Rectangle rk;
                  getBoundingBox(k, rk);
                  Pair p = Pair(k,i);
                  if (ri.intersects(rk) && !pset->isElement(p))
                    {
                      pset->Insert(p);
                    }
                }
            }
        }
    }
  return OZ_SLEEP;

}

OZ_Return Diff2Prop::globalPropagation()
{
  // consider subsets of size n-1
  for (int i = 0; i < _size; i++)
    {
      Rectangle box, ri;
      int areaSum;
      getSubBoundingBox(i, box);
      getBoundingBox(i, ri);
      areaSum = getSubSumAreas(i);
      int riWidth = domainset[i]->getMinWidth();
      int riHeight = domainset[i]->getMinHeight();
      int riArea = riWidth*riHeight;

      if (ri.intersects(box))
        {
          int areaDiff = areaSum+riArea-box.getArea();
          if (areaDiff>0)
          {
            int XOffset = riWidth-int(areaDiff
              /(riHeight<box.getHeight() ? riHeight : box.getHeight()))-1;
            int YOffset = riHeight-int(areaDiff
              /(riWidth<box.getWidth() ? riWidth : box.getWidth()))-1;

            Rectangle pos = Rectangle(box.getXmin()-XOffset,
                                      box.getXmax()+XOffset,
                                      box.getYmin()-YOffset,
                                      box.getYmax()+YOffset);
            bool changed = domainset[i]->removeDomain(pos);
          }
        }
    }

}

// get sum of areas without rectangle j
int Diff2Prop::getSubSumAreas(int j)
{
  int sum = 0;
  for (int i = 0; i < _size; i++)
    if (i!=j)
     sum += (domainset[i]->getMinWidth())*(domainset[i]->getMinHeight());

  return sum;
}

// get bounding box of subset without rectangle i
void Diff2Prop::getSubBoundingBox(int j, Rectangle& box)
{
  int xmin = INT_MAX;
  int ymin = INT_MAX;
  int xmax = INT_MIN;
  int ymax = INT_MIN;

  for(int i = 0; i < _size; i++)
    if (i!=j)
      {
        Rectangle b;
        getBoundingBox(i, b);

        xmin = (b.getXmin() < xmin) ? b.getXmin() : xmin;
        ymin = (b.getYmin() < ymin) ? b.getYmin() : ymin;
        xmax = (b.getXmax() > xmax) ? b.getXmax() : xmax;
        ymax = (b.getYmax() > ymax) ? b.getYmax() : ymax;
      }

  box.setCoords(xmin,xmax,ymin,ymax);
}


// utility function: get bounding box of a rectangle domain
void Diff2Prop::getBoundingBox(int no, Rectangle& box)
{
  OZ_FDIntVar x, y;
  OZ_FDIntVar w, h;
  x.read(_x[no]);
  y.read(_y[no]);
  w.read(_w[no]);
```

```
  h.read(_h[no]);

  int xmin = x->getMinElem();
  int xmax = x->getMaxElem();
  int ymin = y->getMinElem();
  int ymax = y->getMaxElem();
  int wmax = w->getMinElem();
  int hmax = h->getMaxElem();

  box.setCoords(xmin, xmax+wmax-1, ymin, ymax+hmax-1);
}
```

B.3.3 DomainSet.h

```
// filename: DomainSet.h
// author:   Volker Scholz

#ifndef DOMAINSET_H
#define DOMAINSET_H

#include "Domain.h"
#include "Rectangle.h"
#include "BigSet.h"
#include "OverlapType.h"
#include "oz_cpi.hh"

class DomainSet
{
public:
  ~DomainSet();

  // constructor: estimated number of entries, width, height,
  // domain set number
  DomainSet(int nEntriesApprox,
            int wmin, int wmax, int hmin, int hmax, int n);

  // insert new part domain, only .dom field is used
  // kernel and overlap type are automatically computed
  void Insert(Domain& d);

  // initialize iterator
  void InitNext();

  // get next entry (domain)
  bool GetNextEntry(Domain& d);

  // get rectangle width
  int getMinWidth();
  int getMaxWidth();

  // get rectangle height
  int getMinHeight();
  int getMaxHeight();

  // set rectangle width
  int setMinWidth(int wmin);
  int setMaxWidth(int wmax);

  // set rectangle height
  int setMinHeight(int hmin);
  int setMaxHeight(int hmax);

  // true iff domainset empty
  bool isEmpty();

  // return size
  int getSize();

  // get bounding box of part domains
  void computeBoundingBox(Rectangle& box);

  // set bounding box for part domains
  void setBoundingBox(const Rectangle& r);

  // get area O which must not be overlapped by another rectangle
  void getNoOverlapArea(Rectangle& o);

  // get areas C which must not be covered by another rectangle
  void getNoCoverAreas(Rectangle& c, bool& Xsegmented, bool& Ysegmented);

  // update non-overlap area, non-cover area
  void updateForbiddenAreas();

  // remove domain from all part domains
  bool removeDomain(const Rectangle& dom);

  // remove too small domains after increase of minimal rectangle size
  bool removeTooSmallDomains();

  // update kernels after change of rectangle size
  void updateKernels();

  // print all part domains
  void print();

  // update heap references
  void updateHeapRefs(OZ_Boolean duplicate);

  // redefine new operator
  void* operator new(size_t s);

  // redefine delete operator
  void operator delete(void* p, size_t s);

private:
  // rectangle size
  int minWidth, maxWidth;
  int minHeight, maxHeight;

  // domain set number
  int no;

  // compute kernel for rectangle and domain, return overlap type
  void getKernel(const Rectangle& domain,
                 Rectangle& kernel, OverlapType& o);

  // temp domain set
  BigSet<Domain>* temp;

  // domain set
  BigSet<Domain>* domainset;

  // non-overlap, non-cover area
  Rectangle o,c;
```

```
  // segmentation for non-cover area
  bool Xsegmented, Ysegmented;
};


#endif
```

B.3.4 DomainSet.cc

```
// filename: DomainSet.cc
// author:   Volker Scholz

#include "DomainSet.h"
#include <stdio.h>
#include <limits.h>

void DomainSet::updateForbiddenAreas()
{
  if (domainset->isEmpty())
    {
      o.setCoords(1,0,1,0);
      c.setCoords(1,0,1,0);
      return;
    }

  int sizeO, sizeX, sizeY, sizeC;
  sizeO = sizeX = sizeY = sizeC = 0;

  domainset->InitNext();

  // compute sizes of O, X, Y, C
  Domain d;
  while (domainset->GetNextEntry(d))
    {
      switch (d.ovrlap)
        {
        case XY_OVERLAP:
          sizeO++;
          break;
        case X_OVERLAP:
          sizeX++;
          break;
        case Y_OVERLAP:
          sizeY++;
          break;
        case NO_OVERLAP:
          sizeC++;
          break;
        }
    }

  // compute non-overlap area
  if (sizeX==0 && sizeY==0 && sizeC==0)
    {
      domainset->InitNext();
      Domain d;
      // init intersection with first kernel
      domainset->GetNextEntry(d);
      if (d.ovrlap == XY_OVERLAP)
        o = d.kernel;
      else
        {
          o.setCoords(1,0,1,0);
          goto cover;
        }

      // intersect all kernels with area
      while (domainset->GetNextEntry(d))
        {
          if (d.ovrlap == XY_OVERLAP && !o.isEmpty())
            o.intersection(d.kernel, o);
          else
            {
              o.setCoords(1,0,1,0);
              goto cover;
            }
        }
    }

  // if non-overlap area nonempty: we are done
  if (!o.isEmpty())
    {
      c.setCoords(1,0,1,0);
      return;
    }

cover:
  // compute non-cover area

  ////////////////////////////////////
  //         x-projection           /
  ////////////////////////////////////

  Interval Xproj = Interval(1,0); // empty interval

  Xsegmented = false;
  bool XprojEmpty = true;

  // intersection of O and X
  if (sizeY==0 && sizeC==0)
    {
      domainset->InitNext();
      while (domainset->GetNextEntry(d))
        {
          if (d.ovrlap==XY_OVERLAP || d.ovrlap==X_OVERLAP)
            {
              Interval kernelXproj;
              d.kernel.getXproj(kernelXproj);
              if (XprojEmpty)
                {
                  Xproj = kernelXproj;
                  XprojEmpty = false;
                }
              else
                Xproj.intersection(kernelXproj, Xproj);
            }
        }
    }


  if (!Xproj.isEmpty()) Xsegmented = true;
```

```
if (Xproj.isEmpty())           // otherwise we are finished with x
  {
    int XprojMin = INT_MAX;
    int XprojMax = INT_MIN;
    bool covered = false;

    // cover Y and C
    if (sizeY>0 || sizeC>0)
      {
        domainset->InitNext();
        while (domainset->GetNextEntry(d))
          {
            int kernelXmin = d.kernel.getXmin();
            int kernelXmax = d.kernel.getXmax();
            if (d.ovrlap==Y_OVERLAP || d.ovrlap==NO_OVERLAP)
              {
                XprojMin
                  = (kernelXmin < XprojMin) ? kernelXmin : XprojMin;
                XprojMax
                  = (kernelXmax > XprojMax) ? kernelXmax : XprojMax;
                covered = true;
              }
          }
        if (covered)
          Xproj = Interval(XprojMin, XprojMax);
      }

    int minXmax = INT_MAX;        // minimum of right interval points
    int maxXmin = INT_MIN;        // maximum of left interval points

    // extend to overlap of O and X
    if (sizeO>0 || sizeX>0)
      {
        domainset->InitNext();
        while (domainset->GetNextEntry(d))
          {
            int kernelXmin = d.kernel.getXmin();
            int kernelXmax = d.kernel.getXmax();
            Interval kernelXproj;
            d.kernel.getXproj(kernelXproj);

            if ((d.ovrlap==XY_OVERLAP || d.ovrlap==X_OVERLAP)
                && !Xproj.intersects(kernelXproj))
                  {
                    if (Xproj.isEmpty())
                      {
                        minXmax = kernelXmax < minXmax ? kernelXmax : minXmax;
                        maxXmin = kernelXmin > maxXmin ? kernelXmin : maxXmin;
                      }
                    else
                      {
                        if (Xproj.getMax() < kernelXmin)
                          Xproj.setMax(kernelXmin);
                        else        // kernel on right side of Xproj
                          Xproj.setMin(kernelXmax);
                      }
                  }
          }
        if (Xproj.isEmpty())
          Xproj.setCoord(minXmax,maxXmin);
      }
  }
```

```
////////////////////////////////////
//          y-projection          /
////////////////////////////////////

Interval Yproj = Interval(1,0);   // empty interval

Ysegmented = false;
bool YprojEmpty = true;

// intersection of O and Y
if (sizeX==0 && sizeC==0)
  {
    domainset->InitNext();
    while (domainset->GetNextEntry(d))
      {
        if (d.ovrlap==XY_OVERLAP || d.ovrlap==Y_OVERLAP)
          {
            Interval kernelYproj;
            d.kernel.getYproj(kernelYproj);

            if (YprojEmpty)
              {
                Yproj = kernelYproj;
                YprojEmpty = false;
              }
            else
              Yproj.intersection(kernelYproj, Yproj);
          }
      }
  }

if (!Yproj.isEmpty()) Ysegmented = true;

if (Yproj.isEmpty())           // otherwise we are finished with y
  {
    int YprojMin = INT_MAX;
    int YprojMax = INT_MIN;
    bool covered = false;

    // cover X and C
    if (sizeX>0 || sizeC>0)
      {
        domainset->InitNext();
        while (domainset->GetNextEntry(d))
          {
            int kernelYmin = d.kernel.getYmin();
            int kernelYmax = d.kernel.getYmax();
            if (d.ovrlap==X_OVERLAP || d.ovrlap==NO_OVERLAP)
              {
                YprojMin
                  = (kernelYmin < YprojMin) ? kernelYmin : YprojMin;
                YprojMax
                  = (kernelYmax > YprojMax) ? kernelYmax : YprojMax;
                covered = true;
              }
          }

        if (covered)
          Yproj = Interval(YprojMin, YprojMax);
      }

    int minYmax = INT_MAX;        // minimum of upper interval points
    int maxYmin = INT_MIN;        // maximum of lower interval points
```

```
    // (extend to) overlap of 0 and Y
    if (sizeO>0 || sizeY>0)
      {
        domainset->InitNext();
        while (domainset->GetNextEntry(d))
          {
            int kernelYmin = d.kernel.getYmin();
            int kernelYmax = d.kernel.getYmax();
            Interval kernelYproj;
            d.kernel.getYproj(kernelYproj);

            if ((d.ovrlap==XY_OVERLAP || d.ovrlap==Y_OVERLAP)
                && !Yproj.intersects(kernelYproj))
              {
                if (Yproj.isEmpty())
                  {
                    minYmax = (kernelYmax < minYmax) ? kernelYmax : minYmax;
                    maxYmin = (kernelYmin > maxYmin) ? kernelYmin : maxYmin;
                  }
                else
                  {
                    if (Yproj.getMax() < kernelYmin)
                      Yproj.setMax(kernelYmin);           // kernel on upper side
                    else                                  // of Yproj
                      Yproj.setMin(kernelYmax);
                  }
              }
          }
        if (Yproj.isEmpty())
          Yproj.setCoord(minYmax,maxYmin);
      }
  }

  ////////////////////////////////////
  // combine x- and y- projection     /
  ////////////////////////////////////

  c.setCoords(Xproj.getMin(), Xproj.getMax(),
              Yproj.getMin(), Yproj.getMax());

  return;
}

void DomainSet::getKernel(const Rectangle& domain,
                          Rectangle& kernel, OverlapType& o)
{
  int xmin, xmax, ymin, ymax;
  int xUpperRight = domain.getXmin()+getMinWidth()-1;   // pos 1
  int yUpperRight = domain.getYmin()+getMinHeight()-1;
  int xLowerLeft  = domain.getXmax()-getMinWidth()+1;
  int yLowerLeft  = domain.getYmax()-getMinHeight()+1;  // pos 2

  xmin = xUpperRight < xLowerLeft ? xUpperRight : xLowerLeft;
  xmax = xUpperRight > xLowerLeft ? xUpperRight : xLowerLeft;
  ymin = yUpperRight < yLowerLeft ? yUpperRight : yLowerLeft;
  ymax = yUpperRight > yLowerLeft ? yUpperRight : yLowerLeft;

  if (xUpperRight >= xLowerLeft && yUpperRight >= yLowerLeft)
    o = XY_OVERLAP;

  if (xUpperRight >= xLowerLeft && yUpperRight < yLowerLeft)
    o = X_OVERLAP;

  if (xUpperRight < xLowerLeft && yUpperRight >= yLowerLeft)
    o = Y_OVERLAP;

  if (xUpperRight < xLowerLeft && yUpperRight < yLowerLeft)
    o = NO_OVERLAP;

  kernel.setCoords(xmin,xmax,ymin,ymax);
  return;
}
```

B.3.5 Domain.h

```
// filename: Domain.h
// author:   Volker Scholz

#ifndef DOMAIN_H
#define DOMAIN_H

#include "Rectangle.h"
#include "OverlapType.h"
#include <stdio.h>

struct Domain
{
  Rectangle dom;
  Rectangle kernel;
  OverlapType ovrlap;

  void print()
    {
      printf("domain ");
      dom.print(); printf(" kernel: ");
      kernel.print(); printf(" ");

      switch(ovrlap)
        {
        case X_OVERLAP:
          printf("X_OVERLAP");
          break;
        case Y_OVERLAP:
          printf("Y_OVERLAP");
          break;
        case XY_OVERLAP:
          printf("XY_OVERLAP");
          break;
        case NO_OVERLAP:
          printf("NO_OVERLAP");
          break;
        }

      printf("\n");
    }
};

#endif
```

B.3.6 Interval.h

```
// filename: Interval.h
```

```
// author:   Volker Scholz

#ifndef INTERVAL_H
#define INTERVAL_H

class Interval
{
public:
  ~Interval() {}

  // constructor: left endpoint, right endpoint
  Interval(int min, int max);

  // constructor without parameters
  Interval() {}

  // get left endpoint
  int getMin() const;

  // get right endpoint
  int getMax() const;

  // set left endpoint
  void setMin(int min);

  // set right endpoint
  void setMax(int max);

  // set both endpoints
  void setCoord(int min, int max);

  // true iff empty interval (left > right)
  bool isEmpty() const;

  // true iff *this intersects i2
  bool intersects(const Interval& i2) const;

  // true iff *this covers i2
  bool includes(const Interval& i2) const;

  // true iff *this is equal to i2
  bool operator==(const Interval& i2) const;

  // true iff *this and i2 are different
  bool operator!=(const Interval& i2) const;

  // returns intersection of *this and i2
  void intersection(const Interval& i2, Interval& inters) const;

  // print left, right endpoint
  void print() const;

private:
  // left, right endpoint
  int min, max;

  // true iff interval is empty
  bool empty;
};

#endif INTERVAL_H
```

B.3.7 Rectangle.h

```
// filename: Rectangle.h
// author:   Volker Scholz

#ifndef RECTANGLE_H
#define RECTANGLE_H

#include "Interval.h"

class Rectangle
{
public:
  ~Rectangle() { }

  // constructor: left, right, bottom, top
  Rectangle(int xmin, int xmax, int ymin, int ymax);

  // constructor without parameters
  Rectangle() { empty = true; }

  // get left side
  int getXmin() const;

  // get right side
  int getXmax() const;

  // get bottom side
  int getYmin() const;

  // get top side
  int getYmax() const;

  // get rectangle width
  int getWidth() const;

  // get rectangle height
  int getHeight() const;

  // get rectangle area
  int getArea() const;

  // set all coordinates
  void setCoords(int xmin, int xmax, int ymin, int ymax);

  // set left side
  void setXmin(int xmin);

  // set right side
  void setXmax(int xmax);

  // set bottom side
  void setYmin(int ymin);

  // set top side
  void setYmax(int ymax);

  // get rectangles projection on x-axis
  void getXproj(Interval& xproj) const;

  // get rectangles projection on y-axis
  void getYproj(Interval& yproj) const;

  // true iff rectangle is empty
```

```
  bool isEmpty() const;

  // true iff *this intersects r2
  bool intersects(const Rectangle& r2) const;

  // true iff *this includes r2
  bool includes(const Rectangle& r2) const;

  // true iff *this includes point x,y
  bool includesPoint(int x, int y) const;

  // true iff *this is equal to r2
  bool operator==(const Rectangle& r2) const;

  // true iff *this is not equal to r2
  bool operator!=(const Rectangle& r2) const;

  // get intersection of *this and r2
  void intersection(const Rectangle& r2, Rectangle& inters) const;

  // remove domain dom from *this for given rectangle size
  // return four new part domains and true iff new domains were created
  bool removeDomain(const Rectangle& dom,
                    int width, int height,
                    Rectangle newRects[4]) const;

  // print left, right, bottom, top coordinate
  void print() const;

private:
  // coordinates
  int xmin, xmax, ymin, ymax;

  // true iff rectangle is empty
  bool empty;
};

#endif RECTANGLE_H
```