Heuristic methods for routing and scheduling

by
Waldemar Kocjan

May 2001

Abstract

A locomotive assignment is one of the subproblems in railway scheduling domain. In this report present general mathematical model of this specific subproblem and describe how methods known from other problems domain like traveling salesman problem, operation research and constraint programming can be used to solve it. We concentrate especially on method known as *k–interchange* for traveling salesman problem with time windows and give an outline how it can be adopted to locomotive assignment problem. Further, while turning from one trip to another a locomotive must often be reallocated from one station to another. This can be performed in two ways. A locomotive can be driven from one place to another not performing any specific trip and exclusively using track resource, i.e. performs so called deadhead transport, or can be attached to any other transport and passively drown to another station, i.e. perform so called passive transport. Because the cost of passive transport is much lower then cost of a deadhead it is advantageous to, if possible, replace any deadhead by passive transport. In this report we describe a method of converting deadheads into passive transports, describe conversion algorithm, its implementation and report computational result of the algorithm. Finally, we give directions for future research in locomotive planning problem domain.

SICS Technical Report T2001:17
ISSN 1100-3154
ISRN:SICS-T–2001/17-SE

Keywords Train planning, Vehicle routing and scheduling, Local search, Constraint programming

Acknowledgement

The research reported in this paper is a part of the TUFF project founded by Swedish State Railway (SJ) and Green Cargo AB, and ACOOR project founded by Swedish government agencies of NUTEK and VINNOVA.

The author wish to thank Per Kreuger and Martin Aronsson of the DSPS group at SICS for the comments and all help regarding this work, Per Kreuger and Peter Funk of the Department of Computer Engineering at Mälardalen University for supervising this work.

Contents

Chapter 1

Introduction

1.1 Railway planning problem

Since the dawn of industrial revolution railway traffic has played an important roll in development of steel and coal industry. With time railway also became a means of long distance cargo and passenger transportation. Despite hard competition from car and air traffic railroad remains an important factor in world transportation system. Today railroad is considered as more safe, environmentally more advantageous and cheaper than many other kinds of transports.

Planning problems have always been of great interest to railroad companies. Increasing railroad traffic have imposed need for better distribution of resources like locomotives and tracks and yielded improved safety. On the other hand, increasing competition between operators and competition from road and air traffic have given railroad planning problems an economical dimension, it creates a demand on performing transportation tasks in the most economic and energy efficient way.

The railway planning problem belongs to class of more complex and hard to model problems. Scholtz in [Sch00] and authors of [Bus97] determine four subdomains of the railway planning problem in following way:

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

Train scheduling is the core of the railway scheduling problem. It is scheduling of the trains with periodical and non periodical departure, fixing the time schedule and assuring adequate resource allocation for scheduled trips, like track allocation, locomotive allocation and so on. Except resource limit train scheduling is also the subject to some safety restriction like e.g. keeping necessary distance between two trains, exclusive allocation of specific tracks and others.

Rolling stock scheduling concerns allocating locomotives and assigning them to specific trains. This problem is the main subject of this rapport.

Personnel scheduling is the assignment of drivers, service staff and train personnel. In this scheduling domain one can recognize some subdomains. For example service personell responsible for maintaining locomotives is subject to other constraints than locomotive drivers or train staff. Scheduling personell is also subject to legal restrictions as well as union agreements which has to be taken into consideration.

Rescheduling is operative decision making about adjusting the schedule to a new situation caused by for example delay of a train or damage of track network. A planner must often make very fast decision about how the train is going to be rescheduled and this time limit make impossible to perform complete re–optimization of the schedule.

1.2 TUFF project

TUFF, an acronym for **T**åg**u**tveckling **f**ör **f**ramtiden (eng. Train Planning for the Future), is a project run in cooperation between Statens Järnvägar,(eng. Swedish Railway, abbrev. SJ) / Green Cargo AB and The Decision Support for Planning and Scheduling group of the Intelligent System Laboratory at the Swedish Institute of Computer Science (abbrev. SICS). The main purpose of the TUFF project is to investigate in which way information technology and computer science can make the train planning process more effective. The problem domain is specially defined for freight transport planning and is funded by SJ/Green Cargo.

Research of the problem resulted up to now among others in **tuff–3**, a prototype computer program for train planning. The program uses constraint programming technology and is implemented in Mozart and SICStus Prolog.

The **TUFF** architecture is agent based and consist of number of independent agents, like the train scheduler and vehicle routers, cooperating through a coordinator agent as in figure 1.1. In addition to this, the program has a graphical user interface (GUI) which enables communication between the TUFF system and the user. The GUI is connected to the coordinator agent so a user can communicate with the coordinator through it. Requests expressed by the user are propagated through the coordinator to the agents involved in specific operations. The feedback from a specific agent is returned to the coordinator using predefined protocol. Necessary data resources are stored in a network database (Nets in figure **??**), describing the railroad net, and so called Tripsets, which are a specification of the trains.

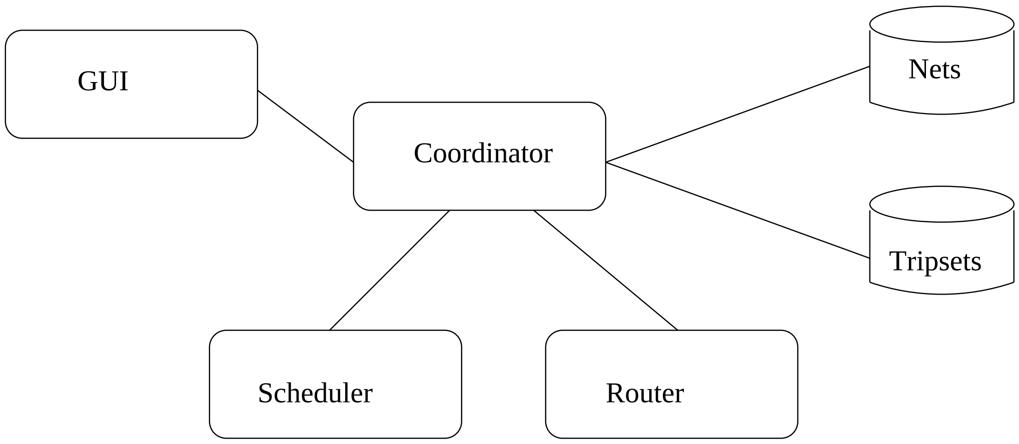

Figure 1.1: TUFF system architecture

The GUI makes it possible to define parameters for the plan. It is also used to display scheduling results in form of a Gantt diagram and track allocation diagrams. The figure 1.2 gives an example of a solution for a locomotive assignemnet problem displayed in the form of a Gantt diagram, where Y–axis represents resources, i.e. the locomotives in the assignement and X–axis is a time axis. The rectangles in the diagram represent tasks/trips performed by a given locomotive during given time period.

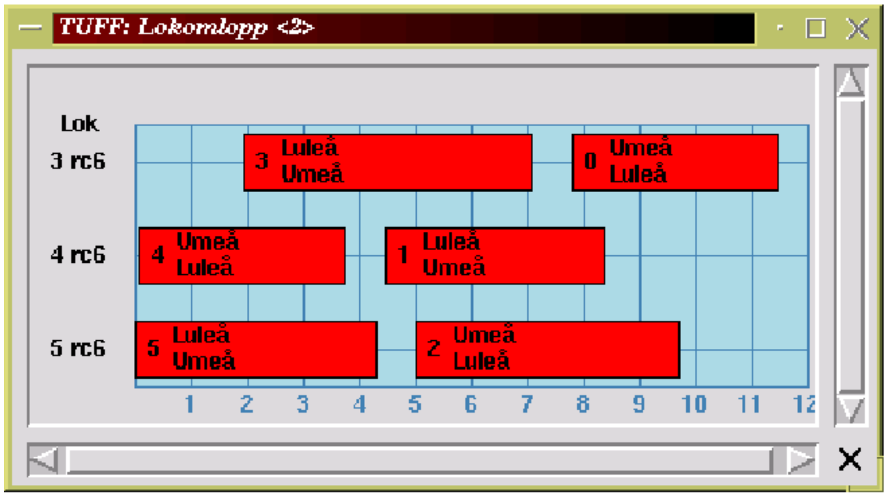

Figure 1.2: Locomotive allocation vizualized as a Gantt diagram

1.3 Goals

The goal of this report is to present a general mathematical model for locomotive assignment problem, refer some optimization methods used to solve problems in similar domains and investigate if those methods can be adopted to the locomotive assignment problem domain.

Work reported here was done during six months at the Swedish Institute of Computer Science.

1.4 Structure of this report

This report describes some chosen methods of dealing with the planning problems in startegic decision making domain, especially locomotive assignment problem. The report is based on research at Swedish Insitute of Computer Science in Kista, Sweden.

In chapter 2 a mathematical foundation and present a general model for locomotive assignment problem is described.

Chapter 3 presents some approaches and methods known in operations research, computer science and other disciplines which we consider useful in combination with our approach.

Next chapter, 4, describes the general optimization algorithm for minimizing assignment cost. The algorithm takes as an input a set of locomotive and the trips assigned to them generated by **TUFF** system in its newest version. It is necessary to stress that the presented algorithm is only an outline of the method. There is too many question marks which must be solved before this part can be implemented and integrated with the system.

In chapter 5 concentrates on the special subproblem in locomotive assignment problem which deals with a conversion from so called deadhead transports, i.e. transports where an locomotive is driven from one station to another without servicing any trip, to passive transports, i.e. a transport where locomotive is reallocated from one station to another using a transport which serves some trip. It describes how this mechanism can be used to minimize costs of locomotive assignments, present an algorithm to solve this, describe our implementation of the algorithm and present computational results of our implementation.

The last chapter, 6 describes direction in which research on locomotive assignment problem could be proceeded.

Chapter 2

Problem description and mathematical foundation

In this chapter we are going to give an overview of the locomotive assignment problem and describe a mathematical model of the problem.

2.1 Basic definitions

In this section we will give some basic definitions needed later for creating a mathematical model for the locomotive assignment problem.

Definition 2.1.1 (Railway network) *Railway network G_t is an undirected multigraph $G_t = \{S, E\}$ where S is a set of vertices representing stations of the network $S = \{s_1, s_2 \ldots, s_n\}$ and E is set of edges $E = \{e_1, e_2, \ldots, e_n\}$ representing track segments between stations.*

We use a multigraph to represent the railway network because there can be more than one track segment connecting s_i with s_j, which is often the case in real world.

Every track segment has its length and a capacity which is a maximum velocity on the track and weight. Every station has its capacity,which represents e.g. maximum number of trains allowed at the station at the same time.

In the rest of this report we use the term track graph as a synonym of railway network.

Trains traversing given railway networks are modeled as trips. Generally, trips are represented by unique identifiers Id, the route r which a given trip follows, a vehicle resource requirement, like locomotive type, location resource requirement and speed parameters used to determine traversal duration [Kre01].

Further, we define task and its synonym term transport in following way:

Definition 2.1.2 (Task/Transport) *A task/transport is a traversal of a track by individual trip.*

The result of assigning values to the time points and durations associated with every transport is a schedule.

Given those basic definitions we can define planning problem in following way:

Definition 2.1.3 (Planning problem) *Given set of trips $P = \{p_1, p_2, \ldots, p_n\}$ and the trackgraph G_t find a schedule for all trips $p \in P$.*

There is some restriction which must be taken into consideration when solving planning problem. One of most important is a exclusivity of track allocation. This constraint can be expressed in following way

Definition 2.1.4 (Exclusivity of track allocation constraint) *Given segment of the track $t_i \in G_t$ can be used by one and only one trip $p_i \in T$ in given time θ_i.*

The other constraints is a safety distance constraint also called headway constraint.

Definition 2.1.5 (Headway constraint) *For two trips p_i and p_j traveling in the same direction and using the same track, if trip p_i starts at time t_i then p_j can start earliest at time $t_j = t_i + c$ where c is a minimum time constant.*

Given those basic definitions we will define in next section locomotive assignment problem.

2.2 Locomotive assignment problem

We define locomotive assignment problem in following way:

Definition 2.2.1 (Locomotive assignment) *Given track graph G_t with a set of stations S and set of track segments E such that $G_t = \{S, E\}$, a set of trips $P = \{p_1, p_2, \ldots, p_n\}$ and a set of locomotives L of suitable types $\mathcal{T}$ assign locomotive of suitable type to every trip in such way that assignment fulfill time and space continuity constraints.*

Time and space continuity constraints used in 2.2.1 are defined as follows:

Definition 2.2.2 (Time and space continuity constraints) *Let $l_i \in L$ be a locomotive of type $\tau_i \in \mathcal{T}$ and $p_i \in P$ a trip departing from station $s_i \in S$ at the time dep_i, l_i can be assigned to p_i if it is at s_i at the time dep_i.*

Assigning locomotive to the trips can be seen as a operation of mapping elements of set L onto elements of set P. To a locomotive $l_i \in L$ one or more $p \in P$ can be assigned . In any case l_i needs to turn from p_i to p_{i+1}. The turn is an operation of docking locomotive l_i from the transport p_i to the transport p_j . Because of the time and space continuity constraints it is sometimes necessary to reallocate l_i from station s_i to s_j, both $\{s_i, s_j\} \in S$. The reallocation must be performed in such a way that the time continuity constraints holds, i.e. we must assure that if we turn l_i from p_i to p_j where $p_i = \{s_{st(i)}, s_{end(i)}, dep_i, trav_i\}$, $s_{st(i)}, s_{end(i)} \in S$ denoting start and end station for p_i respectively, dep_i departure time for p_i and $trav_i$ traversal time or duration of p_i, and $p_j = \{s_{st(j)}, s_{end(j)}, dep_j, trav_j\}$, then if we use $d_{i,j}$ for the turn time from i to j,

$$dep_i + trav_i + d_{i,j} = dep_j$$

and reallocation itself must start and end in time interval between $arr_i = dep_i + trav_i$ and dep_j. In other words the time used for possible reallocation of a locomotive is $0 \leq d_{i,j} \leq dep_j - arr_i$. For difference between turn time and reallocation time we will generally use the term *waiting time*.

A reallocation of the locomotive from one station two another can be performed in two ways. A locomotive can be driven from s_i to s_j without performing any task, or it can be, if possible, attached to a transport performed from s_i to s_j if such attachment is allowed. We will call the first type of reallocation a *deadhead transport* or shortly a *deadhead*, whereas we will use term *passive transport* for the other one.

There is a large difference in the cost between these two ways of reallocating a locomotive. We simplify here the cost relation between transport servicing trips, passive transports and deadheads in following way: if the cost for trip t_i between stations s_1 and s_2 is equal to c then a passive transport performed on exactly the same track in exactly the same conditions between s_1 and s_2 costs $2 * c$ and a deadhead costs equals $4 * c$.

It is easy to see that a great deal of cost optimization can be performed by replacing deadheads by passive transports or eliminating deadheads in favor of transports servicing trips. We should also mention that cost for setting one locomotive into operation is very high so even if some deadheads may be very expensive it is more advantageous to use a deadhead to turn from one trip to another rather than using new locomotive to service given trip.

Generally, locomotive assignment problem should be optimized with respect to following parameters:

- minimum number of used locomotives
- minimum cost of turns

where minimum cost of turns can be constrained by

- minimum total deadhead cost
- minimum total cost for passive transports

Note that we omit the cost for waiting time here. We assume that cost for waiting time is constant and minimizing waiting time for one turn of locomotive l_i must increase the waiting time for some other turn for l_i. With such assumptions minimizing cost for waiting time locally for p_i will not minimize the global cost for l_i's waiting time nor will it have any influence on the overall cost for using l_i. In reality the cost function for waiting time is more complex.

2.3 Mathematical model

Given definitions from the previous sections we can now define our mathematical model for locomotive assignment problem.

This model follows a general model given in [Dro97] for locomotive assignment. Some small differences between our model and the one presented in [Dro97] will be explained in section 2.4.

Let $\mathcal{I}$ denote the set of tasks (trips), $\mathcal{K}$ the possible turns between the given trips and $\mathcal{T}$ the set of locomotive types. If $i \in \mathcal{I}$ and $j \in \mathcal{I}$ then $(i, j) \in \mathcal{K}$ if and only if at least one locomotive type can do task j after task i without any intermediate tasks.

The subsets $\mathcal{I}' \subset \mathcal{I}$ and $\mathcal{K}' \subset \mathcal{K}$ denote the special cases of turns and tasks which are crossing chosen timeline which we will call a *period border*. This part of the objective function counts the number of locomotives used for servicing all trips.The required number of locomotives for specific trip i is denoted as r_i and maximal number of passive locomotives as pt_i. For detailed discussion of this special cases of transports and formalism for cyclic time see [Kre] and [Aro01].

The optimization model contains some decision variables: $x^t_{(ij)}$, $(ij) \in \mathcal{K}$, $t \in \mathcal{T}$ denotes the number of locomotive of type t connecting from task i to j. The number of active locomotives of type t on task i is denoted y^t_i and number of passive locomotives as z^t_i. All these variables has a cost $c^t_{(ij)}$ associated with them, which is a cost of connecting one locomotive of type t from the task i to j. Further constants d^t_i and e^t_i are the costs of running locomotive of type t on task i actively and passively respective. Cost of using locomotive of type t is denoted as g_t.

With those definitions the optimization problem can be stated as follows:

$$\min \sum_{t \in \mathcal{T}} [\sum_{(ij) \in \mathcal{K}} c^t_{(ij)} x^t_{(ij)} + \sum_{i \in \mathcal{I}} (d^t_i y^t_i + e^t_i z^t_i) + g_t (\sum_{(ij) \in \mathcal{K}'} x^t_{(ij)} + \sum_{i \in \mathcal{I}'} y^t_i + z^t_i)]$$

s.t.

$$(\mathbf{1}) \quad \sum_{t \in \mathcal{T}} y_i^t = r_i \qquad \forall i \in \mathcal{I}$$

$$(\mathbf{2}) \quad \sum_{t \in \tau} z_i^t \leq pt_i \qquad \forall i \in \mathcal{I}$$

$$(\mathbf{3}) \quad y_i^t + z_i^t = \sum_{j|(ji) \in \mathcal{K}} x_{(ji)}^t = \sum_{j|(ij) \in \mathcal{K}} x_{(ij)}^t \qquad \forall i \in \mathcal{I}, \quad \forall t \in \mathcal{T}$$

$$(\mathbf{4}) \quad x_{(ij)},\ y_i^t \ and\ z_i^t\ are\ non-negative\ integers$$

The first term of objective function counts the costs of connections used in the solution. The second and third terms count the costs of active and passive transports,respectively. The last term summarize total number of locomotives used in the solution and multiply them by by the constant cost of given locomotive type.

Constraint (1) states that number of active locomotives assigned to the task i must be exactly the number of locomotives needed i.e. if the some task demands 2 active locomotives instead of 1 then 2 active locomotives must be used. In practice number of active locomotives used on the task vary between 1 and 2.

Constraint (2) limits number of passive transports. It states that number of passive locomotives on the task i must be lower or equal maximum number of passive locomotives allowed for given In practice it is often determined by type of transport and type of used active locomotive.

Constraint (3) requires that number of locomotives of type t connecting to the task i is identical with number locomotives actually assigned to the task. It is also called conservation constrain because it maintain basic property of task (?). This number must be the number of the locomotives connecting from the task.

Finally, last constraint states that solution of the problem must be non–negative and must contains property of integer solution.

2.4 Representation

It is commonly known that a visualization of given problem and its solution has great influence on the use of methods for solving it. The way in which a problem and its solution are visualized influences created model of a problem and the set of tools available for given model.

When thinking about a train schedule its natural to visualize it in form of a Gantt diagram, where axes of the diagram represent time and single locomotives. A train schedule is then represented as a rectangle which stretches from the departure time at the start station for a trip to an arrival time at the end station. Such train/transport is ordered to locomotive servicing represented trip.

Optimizing such model reminds of a problem of non–overlapping polytop fitting i n–dimensional space, here limited to rectangle fitting in 2–dimensional plane.

The idea of global geometrical constraint for rectangle fitting in the plane was originally developed by CHIP system developers as `diff2` constraint [Sim95]. There was some work done with adopting this ideas into the train scheduling problem domain in [Sch00], but the model still suffer of an inefficient propagation.

In our model we are going to represent a schedule by a set of locomotive circuits. A locomotive circuit is a weighted, directed, finite graph $G_L = \{V, E\}$ where vertices V represents tasks performed by a locomotive and edges E represents turns between the tasks. A locomotive task occurs when a locomotive serve any trip p. A turn is a reallocating a locomotive L from trip p_i to trip p_{i+1}.

One locomotive circuit can be served by several locomotives.Although in reality there exists several types of locomotives we assume that the type of the locomotive used in one circuit must be uniform. The set of the circuits can be served by several different types of locomotives.

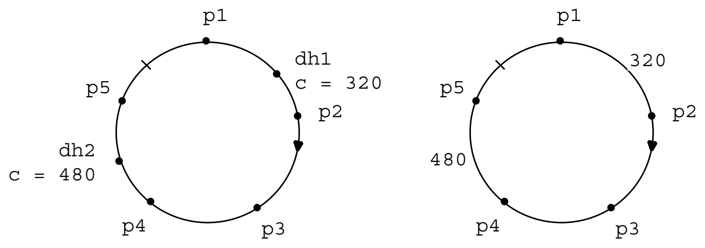

Figure 2.1: The representation of the locomotive circuit. The left figure shows representation used in [Dro97]. The arrow represents direction of the turns. Orthogonal line indicates period frame for the circuit. Note that in this representation deadheads are explicitely represented as trips(vertices). The right figure shows representation of the locomotive circuit used in our model. Cost of the deadhead c is included in the cost of the turn between two trips.

In [Dro97] all the deadheads are explicit represented as separate trips / transports. The consequence of such representation is that a locomotive may not be reallocated from one station to another while turning between two trips. If such reallocation is necessary than a new transport has to be created. The optimization process is then to remove all those special transports, if such removal is possible.

In such model every special transport is represented as a vertex of a locomotive circuit. In contrary, in our representation is such special transport counted into cost for a turn between two trips. This allows straightforward implementation of the optimization methods known from e.g. traveling salesman problem domain, especially methods based on exchanging edges with high cost with those with lower cost (see chapter 3). Figure 2.1 illustrates difference between both representations.

Chapter 3

Methods

3.1 Local search for standard traveling salesman problem

The standard traveling saleman problem (TSP) can be defined as follow:

Definition 3.1.1 (Standard TSP) *Given finite set V of vertices and distance $t_{i,j}$ for each pair of vertices $i, j \in V$ find a tour with minimal total length, where tour is a closed path that visits each vertex exactly once.[Sav85]*

There are additional assumption made for instance of TSP known as standard TSP. The distance matrix in standard TSP is symmetric and it satisfies triangle inequality.

We will describe below local optimization methods for standard TSP known as $k - interchange$. The k–interchange method is a substitution of k–links(arcs) of the tour with another set of k links. This method was first introduced by Croes for $k = 2$ in [Cro58] and Lin for $k = 3$ in [Lin65].

A tour is said to be $k - optimal$ if its not possible to obtain of a tour of shorter length by replacing k of its links by another set of k links. There is infinite number of possible k which can be taken into consideration, nevertheless the computing effort raises very rapidly with k so the most common cases are $k = 2$ and $k = 3$. We will try to illustrate here those methods and explain in following sections how the idea of *k–interchange* can be used in TSP with time window constraints and later how this method can be incorporated in optimizing locomotive circuits.

Given a TSP tour, *k–2 interchange* tries to substitute two links $(i, i + 1)$ and $(j, j + 1)$ with two other links, in our case (i, j) and $(i + 1, j + 1)$, as illustrated in figure 3.1. Let c be a cost of the path between two nodes. If condition

$$c_{i,i+1} + c_{j,j+1} > c_{i,j} + c_{i+1,j+1} \tag{3.1}$$

holds, then *k–2* results in local tour improvement.

The total number of possible 2–interchanges is equal the number of subsets of two links from the set of n links, i.e. $\binom{n-1}{2}$, which implies time complexity of $O(n^2)$.

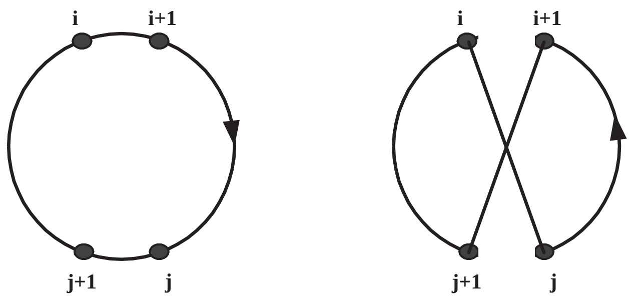

Figure 3.1: 2–interchange

Notice that 2–interchange reverse path $(i+1, \ldots, j)$. The effort of checking if 2–interchange results in any local improvement is reduced here by the fact that distance matrix for such standard problem is symmetric.

The idea can be easily extended to case of $k = 3$. But in case of $k = 3$ it is a triplet of links which are replaced. Figure 3.2 shows one of eight possible 3–interchanges. Because computation complexity of verifying 3–interchange

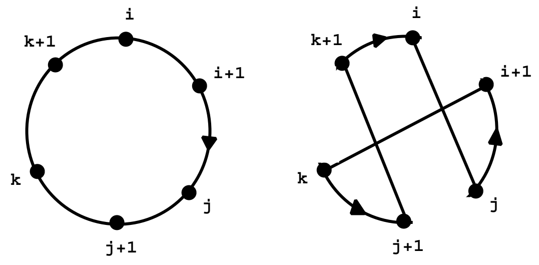

Figure 3.2: One of possible $k - 3$ interchanges

raises rapidly if the number of vertices increases there was made some effort to improve 3–interchange by different means. A new way of dealing with high complexity was given in [Or,76]. In proposed method only a subset of possible 3–interchanges is taken into consideration. This procedure apply interchanges which would result in one, two or tree consecutive vertices being inserted between two other vertices. Figure 3.3 show how Or–interchange is carried out. In this example consecutive vertices $i, i+1, i+3$ are reallocated and inserted between j and $j+1$. One of the advantages of Or method over 3–interchange is its computational complexity. While verifying 3–optimality yields $O(n^3)$, time complexity for Or method is $O(n^2)$.

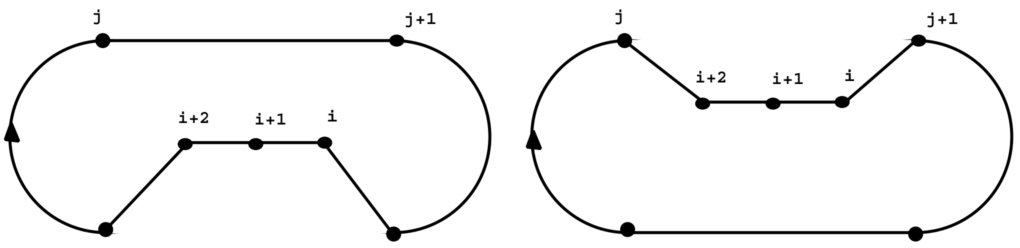

Figure 3.3: An Or–interchange

The idea of *k–interchange* for standard TSP was succesfully adopted for domain of asymetric TSP's by [Kan80]. The main difference between method described in [Kan80] and *k–interchange* method for symmetric TSP is that in optimizing an asymmetric problem no tour segment can be reversed. They are instead reordered if and only if there is an interchange which results in lowering the cost of the tour, i.e. if equation 3.1 holds.

This method was also succesfully extended for iterated search by [Mar92]. For more informtion on this method and review of other optimization methods for asymmetric TSP see [Cir].

3.2 Optimizing TSP with time windows using k–interchange

Local search for standard traveling salesman problem with the time windows (TSPTW) extends standard TSP by introducing for every vertex i a *time window* , denoted $[e_i, l_i]$ where e_i is a earliest possible time of service at i and l_i is a latest possible service time at i.

When dealing with TSPTW authors make usually following assumptions. A service time at any vertex is equal to 0, latest possible service time at i is a strict constraint, which, if violated, make tour unfeasible. Earliest service time constraint at i is not a strict constraint, arriving at i earlier then e_i does not lead to infeasibility, but merely introduces waiting time at i. We are going to give our interpretation of those constraints in section 4.3 when describing using *k–interchange* for optimizing locomotive circuits.

Optimizing TSPTW strongly depend on the definition of the objective function. We are going to use here definitions from [Sav85]. The rest of this section presents algorithm explicitly described in [Sav85] if not stated otherwise.

[Sav85] states that local improvement of the tour is both feasible and profitable if and only if following conditions are satisfied:

1. the time spent on actual traveling is minimized
2. the completion time of the tour is minimized

The first objective can be formalized as

$$min \left\{ \sum_{i=1}^{N-1} t_{i,i+1} + t_{N,1} \right\} \tag{3.2}$$

where $t_{i,i+1}$ is a traveling time between two vertices and $t_{N,1}$ is a finishing tour from vertex N to the vertex which starts whole tour.

With this objective a k–interchange operation is feasible and profitable if and only if a cost of the travel, i.e. actual travel time is reduced, which follows from condition for interchange in equation 3.1

$$t_{i,j} + t_{i+1,j+1} < t_{i,i+1} + t_{j,j+1} \tag{3.3}$$

and when the new tour is feasible.

Feasibility of a new tour are expressed in following way. Let i, j, k be a vertices of the tour, D_i maximum departure time at i, W_i the waiting time at i, $t_{i,j}$ traversal time between i and j and l_i the latest departure time at i. The new tour is feasible if and only if

$$i < k \leq j : D_k^{new} = D_i + t_{i,j} + \sum_{p=k+1}^{j} (W_p^{new} + t_{p-1,p}) \leq l_k \tag{3.4}$$

and

$$\begin{split} j < k \leq N : D_k^{new} = D_i + t_{i,j} + \sum_{p=i+2}^{j} (W_p^{new} + t_{p-1,p}) \\ + W_{i+1}^{new} + t_{i+1,j+1} + \sum_{p=j+1}^{k-1} (W_p^{new} + t_{p,p+1}) \leq l_k \end{split} \tag{3.5}$$

Equation 3.4 states that new departure time at k, which is equal to departure time at i plus traversal time between i and j, and sum of all new waiting times on all newly inserted vertices between i and j and traversal time between them must be lower or equal latest departure time at k. Equation 3.5 states this for k being second link of k–interchange.

The second objective of the problem is minimizing completion time of the tour. From the assumption about no service time at vertices follows that minimizing cost/length of the tour shifts departure time at the vertices. Thus the second objective can be expressed as follows:

$$minD_N + t_{N,1} \tag{3.6}$$

which states that aim of optimization is to decrease departure time and traveling time from last visited vertex to vertex there tour starts. Objective implies decreasing arrival time at $j+1$

$$A_{j+1}^{new} < A_{j+1} \tag{3.7}$$

and carrying out whole or part of the gain to the vertex which finishes tour

$$j+1 \leq k \leq N : D_k > e_k \tag{3.8}$$

Also reversed path of the tour must be feasible

$$j < k \leq j : D_i + t_{i,j} + \sum_{p=k+1}^{j} W_p^{new} + t_{p-1,p}) \leq l_k \tag{3.9}$$

If any of constraints 3.7, 3.8 or 3.9 is violated then tour is not considered to be feasible.

The main problem with carrying out such optimization is time complexity connected with feasibility check for all the vertices of the new path. Straightforward implementation implies time complexity $O(N)$ for each vertex. This implies that verification of 2–optimality will run in $O(N^3)$. To decrease this time [Sav85] propose lexicographic search strategy reducing feasibility check to $O(1)$ for every 2–interchange. Using this strategy one choses the links $(i, i+1)$ in order they appear in the current tour starting with (1,2). First a link $(i, i+1)$ is fixed. Then links $(j, j+1)$ equal to $(i+2, i+3)$ are chosen followed by $(i+3, i+4), \ldots, (N-1, N)$. This specific order implies that after considering all possible interchanges for link $(i, i+1)$ one can use gathered information to compute length and check feasibility for the path from $(i+1, \ldots, j)$.

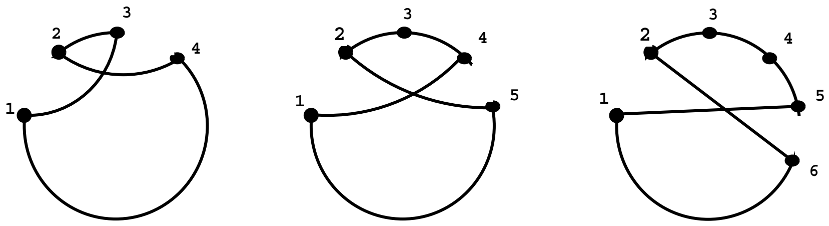

Figure 3.4: The lexicographic search strategy

Performing interchange on link $(j, j+1)$ the path $(j-1, \ldots, i)$ of previously considered interchange is expanded by link $(j, j-1)$, which usually results in change of the departure time at $j-1$ and possibly on all the other vertices on the path $(j-1, \ldots, i+1)$. If we define

$$SHIFT^{(j,j+1)} = D_j^{(j,j+1)} + t_{j,j-1} - D_{j-1}^{(j-1,j)} \tag{3.10}$$

and $PFS^{(j,j+1)}$, possible forward shift in time of the departure time at j causing no violation of the time window constraints along the path $(j, \dots, i+1)$ as

$$PFS^{(j,j+1)} = min_{i+1 \le k \le j} \left\{ l_k - (D_j^{(j,j+1)} + \sum_{p=k}^{j-1} t_{p,p+1}) \right\} \tag{3.11}$$

then expanding path $(j-1, \dots, i+1)$ with the link $(j-1, j)$ is feasible if and only if

$$SHIFT^{(j,j+1)} \le PFS^{(j-1,j)} \tag{3.12}$$

For prove of this theorem see [Sav85].

Although this method is applied on TSPTW with symmetric distance matrix the lexicographic search strategy can be also applied on TSPTW for asymmetric problems.

3.3 Constraint programming

Constraint programming (CP) is a software technology which lately was successfully applied in solving many combinatorial problems like traveling salesman problem or job shop scheduling and other scheduling problems. In this section we will give short presentation of CP.

One can describe constraint using informal language as a relation between several unknowns, i.e. variables, where every variable is taking value in given domain ([Bar]). The idea of constraint programming is thus to solve problems by stating constraints (condition, properties) which must be satisfied by solution. This leads to idea formulated as Constraint Satisfaction Problem (CSP).

A CSP consist of tree parts:

1. A finite set of variables
2. A domain associated with each variable
3. A set of constraints restricting the value that any variable can take

Solving CSP is assigning value for each variable in such a way that no constraint (condition) is violated and value of variable lies within its domain.

Formally, the domain of variable is defined as follows:

Definition 3.3.1 (Domain) *The domain of the variable is the set of values that may be assigned to the variable. D_x denotes the domain for variable x.*

Assigning value to each variable is called a labeling and assignment itself a label.

Definition 3.3.2 (Label) *A label is a pair of variable and the value. It represents the assignment of that value to the variable. A label which assigns the value v to the variable x is denoted $< x, v >$. This assignment is meaningful if v is in the domain of x.*

To one value can be assigned one or more values. Such assignment is called compound label.

Definition 3.3.3 (Compaund label) *A compound label is the simultaneous assignment of values to a set of variables. The compound label of assigning $v_1, v_2, \ldots, v_n$ to $x_1, x_2, \ldots, x_n$ is denoted by $(< x_1, v_1 >, < x_2, v_2 >, \ldots, < x_n, v_n >$.*

Given those definitions we can define a constraint in following way:

Definition 3.3.4 (Constraint) *A constraint on a set of variables is conceptually a set of compound labels for the variable in the problem. A constraint on the set of variables S is denoted C_s.*

Finally, we define Constraint Satisfaction Problem as:

Definition 3.3.5 (Constraint Satisfaction Problem) *A CSP is a triple (Z, D, C) where Z is a finite set of variables $\{x_1, x_2, \ldots, x_n\}$, D is a function which maps every object in Z into a set of objects of arbitrary type and C is a finite set of constraints on a subset of variables in Z.*

Solving of CSP is carried out by two methods: *problem reduction* and *search*. The idea of problem reduction is to make the problem smaller by reducing domain of variables. Reduction is made by constraint propagation. Those are basically two types of constraint propagation: *domain propagation* and *interval propagation*. To see the difference between those two consider following example:

Assume that there exists a CSP with variables X and Y and domains $D_x = \{1, 2, \ldots, 10\}$ and $D_y = \{1, 2, \ldots, 7\}$. The constraint of variables is $C_{xy} : 2X = Y$. Using domain propagation narrows domain as much as possible so the domains will become

$$D_x = \{1, 2, 3\}$$

and

$$D_y = \{2, 4, 6\}$$

, whereas interval propagation will limit only bounds of domain and produce

$$D_x = \{1, 2, 3\}$$

and

$$D_y = \{2, 4, 6\}$$

A propagation itself is an incomplete method, most often it must be combined with some search strategy. There is several search strategies which can be applied to solve CSP. Backtracking is one of the most simple but frequently used techniques.

Basic idea of backtracking is to try to assign a value to the variable. The generic backtracking algorithm works in following way:

Algorithm 1 Simple backtracking

```
1: repeat
2:   choose a variable in CSP
3:   repeat
4:     choose value in domain of this variable. Check if it satisfies con-
       straints.
5:   until value is found or there is no more values which can be assigned
6:   if value found then
7:     go to 1
8:   else
9:     go to last assigned variable and change its value. Go to 2.
10:  end if
11: until solution is found or all combination of labels have been examined
    and failed
```

While searching for a solution of CSP it is important what kind of search strategy is used while looking for variable to be labeled. There is some search strategies which could be used to find such variable and to label it with specific value. A naive search strategy chooses the first variable in arbitrary ordering of variables and label it with lowest possible value in domain. This method is implemented in SICStus prolog as labeling option called leftmost.

Another approach is trying to find a variable which is most likely to fail. This method aims at recognizing propagation dead–ends as soon as possible and thereby reduce computation costs. The easiest way to determine which variable is most likely to fail is to determine the size of variables domains another to test which of variable with smallest domain has most constraints suspended on it. Both strategies are implemented in SICStus as ff respective ffc [SIC00].

Combining labeling with backtracking reduces necessary number of backtracks and limit a search space of the problem.

3.4 Optimizing: branch–and–bound algorithm

There is several methods for optimizing TSPTW and derived problems in terms of minimizing cost of solution. The usual method for optimizing combinatorial problems is branch–and–bound algorithm. Note that the predicate `minimize/2` in `clpfd` solver implemented in SICStus prolog are implemented as branch–and–bound[SIC00].

The way in which generic branch–and–bound algorithm works can be describe in following way. Let S be a set of feasible solutions to integer programming problem $\mathcal{L}$. The algorithm divides feasible set S into a set of subsets $\{S^i : 1 = 1, \ldots, k\}$ and then solves the problem over each of the subsets. The division is frequently done recursively as shown in tree in figure 3.5.

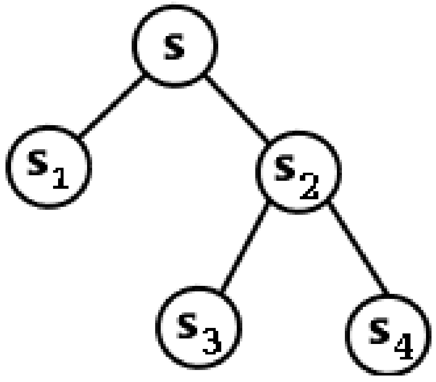

Figure 3.5: Branching feasible set S into subproblems S_1 and S_2. S_2 is also partitioned into S_3 and S_4.

The subproblems are created by adding linear constraints to the problem, after solving linear programming relaxation of the problem and computing lower bound for it. The obvious way to do it is to take $S = S^1 \cup S^2$ with $S^1 = S \cap \{x \in R^n_+ : bx \leq d_0\}$ and $S^2 = S \cap \{x \in R^n_+ : dx \geq d_0 + 1\}$, where $(d, d_0) \in Z^{n+1}$

Carried to the extreme, division can be viewed as total enumeration of the elements in S.Suppose that S has been divided into subsets $\{S^1, \ldots, S^k\}$. If we can establish that no further division of S^i is necessary, then enumeration tree can be pruned at the node corresponding to S^i.

After partitioning of feasible set, assuming that we have an algorithm which computes a lower bound $b(S_i)$ to the optimal cost for subproblem S_i, where

$$b(S_i) \leq \min_{x \in S_i} c'x$$

During evaluation process some upper bound U corresponding to cost of certain feasible solutions will be set and maintained during the further evaluation. If the lower bound $b(S_i)$ corresponding to particular subproblem S_i satisfies $b(S_i) \geq U$ then the best feasible solution so far is an optimal solution for S_i and subproblem S_i does not to be considered further. If this condition is fulfilled for all subproblems then optimal solution for S is found and algorithm terminates.

Algorithm 2 Generic branch–and–bound

```
1: Select an active subproblem $S_i$
2: if $S_i$ is infeasible then
3:     delete $S_i$
4: else
5:     compute $b(S_i)$
6:     if $b(S_i) \geq U$ then
7:         delete $S_i$
8:     else
9:         obtain optimal solution for $S_i$ or break it into subproblems and add
           them to list of active subproblems
10:    end if
11: end if
```

As we can see branch–and–bound is very general optimization method. Its performance greatly depends on strategies used while choosing and branching active problem as well as on way of computing lower bound. While implementing this algorithm in constraint programming languages its usual to combine algorithm with some labeling strategies. More about how this strategies can cooperate during optimization process see 5.6.7

3.5 Related work

Traveling salesman problem is a problem with one of the largest bibliography. We can refer here to [Low85] for problem history and bibliography.

Although insertion heuristics for TSP is outside the scoop of this report we need to mention worst case analysis of insertion heuristics in [Ros77] and the work of Christofides presented in [Chr76].

The *k–interchange* as an improvement method for TSP was originally introduced in [Cro58] for $k = 2$ and [Lin65] for $k = 3$. Since then there was several papers written on this subject, which examine different issues related to application of this method. It is worth to mention generalization of this method in [Lin73] and report of worst–case behavior in [Pap78]. The effective implementation of this method was described in [Hal00].

The extensive description of methods related to asymmetric TSP was given in [Cir]. The referred paper confirm among others the results of TSP heuristic described in [Zha93]. It also gives a review of *k–interchange* methods applied on asymmetric TSP. The paper also report results of *k–interchange* adopted to asymmetric TSP by Kanellakis and Papadimitriou and presented in [Kan80], as well as the results of iterated local search described in [Mar92] and the local search and *k–interchange* particular improved by dynamic programming approach (see [Sim96] for local search in general and [Glo96] for case of *4–interchange*).

The traveling salesman problem and related vehicle routing and scheduling problems with the time window constraints were presented in the work of Salomon [Sal83], [Sal87] and [Des92]. The worst–case performance for different heuristics was referred in [Sal86].

The idea presented in [Sav85] was later developed in [Kin85].

From the literature about constraint programming we need to mention [Tsa93] and [Stu98] as well as introduction to CP in [Bar].

Using of constraint programming for solving scheduling problems and TSP's with time windows was a subject of [Pes98],[Cas97],[Kil00] and others.

Finally, the branch–and–bound algorithm is described in most of the books about linear programming and optimization. Using branch–and–bound to solve TSP and scheduling problems is referred in, among others, [Cir] and [Sig00]. We also need to mention [Har80], where different search strategies using branch–and–bound algorithm are described.

Chapter 4

Four steps optimization

4.1 Introduction

In this chapter we are going to present an outline of the method for optimizing a locomotive assignment problem. A starting point of this algorithm is a input from existing **TUFF** system. The **TUFF** system generates an initial feasible solution which is a set of locomotive circuits.

The ambition of this algorithm is to keep basic properties of initial solution like maximum number of locomotives used and an improved cost of initial solution. There is two possible strategies which can be used while optimizing a set of locomotive circuits. All circuits can be merged into a 'grand tour', then the *k–interchange* is applied on single cyclic linear graph serviced by a number of locomotives or it can be applied on set of disjointed graphs.

Both strategies have their benefits and drawbacks. Merging of all circuits in one grand tour makes very hard to maintain the basic features of the original set of circuit which is overall the cost for all circuits and the number of used locomotives. On the other hand it gives a benefit of applying lexicographic search strategy during optimization process and makes application of *k–interchange* more straightforward.

Performing interchange operation on an originally disjoint set of circuits maintain the original features of initial solution but introduce new 'free parameters' during optimization process, i.e. the strategy of choosing which circuit from original set is going to be chosen for searching for proper interchange.

Consequently, depending on the chosen strategy in algorithm, i.e. depending on if it is going to be applied on grand tour or if its going to be applied on a set of disjointed circuits, there are two possible way of implementing presented algorithm. In this chapter we are going to show how those operations can be used depending on the chosen strategy.

Algorithm 3 The 'grand tour' optimization

```
Require: initial solution: set of feasible locomotive circuits $G_L$
Require: trackgraph $G_t$
1: Merge circuits into the grand tour
2: Optimize using *k–interchange*
3: Disjoint 'the grand tour' into set of separate circuits if the sum of cost
   for disjointed circuits is lower then overall cost for 'the grand tour'.
4: Convert deadheads into passive transports
```

Algorithm 4 Optimizing a set of disjoint graphs

```
Require: initial solution: set of feasible locomotive circuits $G_L$
Require: trackgraph $G_t$
1: for all locomotive circuits $lc \in G_L$ do
2:     if a sum of cost for $lc_i \in G_L$ and $lc_j \in G_L$ is higher then cost for
       merged $lc_i$ and $lc_j$ then
3:         merge $lc_i$ and $lc_j$
4:     end if
5: end for
6: Optimize using *k–interchange* on disjoint graphs
7: for all Resulting circuits do
8:     if a cost for circuits can be reduced by disjoint operation then
9:         disjoint circuit
10:    end if
11: end for
12: Convert deadheads into passive transports
```

4.2 Merging graphs

Depending on the chosen approach the merging operation can be defined in two possible way. If we choose to apply the merge operation as a way of dealing with basic difference between traveling salesman problem and locomotive assignment problem, i.e. to treat multi–actor problem, which is the nature of locomotive assignment problem, as single–actor problem, the TSP.

Let $G_L = \{g_1, \ldots, g_n\}$ be a set of n locomotive circuits, and $g_i = \{p_{i_1}, \ldots, p_{i_n}\}$ and $g_j = \{p_{j_1}, \ldots, p_{j_n}\}$ two locomotive circuits in G_L, where $p_{i_i}, p_{i_{i+1}}$ and $p_{j_i}, p_{j_{i+1}}$ are the arbitrary consecutive trips in lc_i respective g_j.

Given those definitions the merging for a 'grand tour' can be defined as follows:

Definition 4.2.1 (Merging for a grand tour) *Given a set of* **n** *locomotive circuits with the uniform cycle time merge them into one graph by generating turns between an arbitrary $p_{1_{i+1}}$ of some graph g_1 and an arbitrary p_{1_i} of some graph g_2 and deleting turn between p_{i_i} of every $g_i \in G_L$ and $p_{i_{i+1}}$. Close graph by generating turn between $p_{n_{i+1}}$ and p_{1_i}.*

Merging circuits into a grand tour removes an earlier time specification for the trips in the merged graph. If the *cycleTime* is the total time necessary to service all trips in a given circuit and perform all the turns between all the trips in the circuit then a new time specification is adjusted by adding to the original time the index i of the merged graph $g_i \in G_L$ multiply with the *cycleTime*. All the time specification for the new merged graph relates to the time specification of arbitrary p_{1_i}. This operation of recomputing time specification for all the trips in the merged graph is performed mainly to maintain precedence relation between the trips inside the new merged circuit and does not have any practical impact on the optimization process.

In most of the cases such an operation will result in a circuit with overall cost greater then overall cost of original set of circuits, because it will be necessary to introduce deadheads transport between the arbitrary trips in two merged circuits.We would like to assume that any deadhead which is introduced by merging graphs will be removed or its cost will be reduced during further optimization, but proving this assumption we will leave for future work.

On the other hand we can choose a merging strategy which will minimize the necessity of introducing new deadhead transports. Merging graphs can be implemented using an insertion heuristics which minimizes number of the necessary deadheads. The strategy in such approach is to look at all links where one original period frame starts and next period frame begin and calculate the cost of inserting given circuit into circuit already merged. The cost of such insertion is computed by calculating the cost for deadheads which will be generated if the insertion would take place and additional period frames which would have to be generated to perform such insertion. The cost for the additional period frames is a cost of additional locomotives which would have to be used to service such circuit. All possible insertion are sorted by their cost and cheapest one is chosen.

There is some other possible strategies which could be designed having in mind step 2 and 3 of algorithm. As we know that for example step 3 will split merged graph on to expensive links we can maintain during merging some expensive deadheads on the turns where period frame start and choose to perform *k–interchange* in phase 3 only on links not marked as start of period frame, or perform it in way which assure that splitting operation will be performed on them.

Yet another strategy is to maintain array of circuits which were merged using different insertion approaches and having different overall cost. Such strategy allows to use optimization techniques known from research in field of genetic algorithms. With well defined heuristics for mutation and cross over between different individuals/merged graphs in population array combined with *k–interchange* method could, despite probably high time complexity, results in interesting solutions

In any case, this part of algorithm demands further research.

In the other approach, when we choose to run the algorithm on the set of disjointed graphs, merging operation can be seen as optimization step.

Definition 4.2.2 (Merging as an optimization step) *Given the set of locomotive circuits $G_L = \{g_1, \dots, g_n\}$ if there exists two such circuits $g_i, g_j \in G_L$ with arbitrary consecutive turns $p_{i_i}, p_{i_{i+1}} \in g_i$ and $p_{j_i}, p_{j_{i+1}} \in g_j$, that the cost of turns $p_{i_i}, p_{i_{i+1}}$ plus $p_{j_i}, p_{j_{i+1}}$ is higher then $p_{i_{i+1}}, p_{j_i}$ plus $p_{j_{i+1}}, p_{i_i}$, the number of locomotives which could service such circle is equal to or lower then the sum of the locomotives servicing g_i and g_j and the departure time windows for all the trips in circuits g_i and g_j will not be violated, merge both circuits by generating turns $p_{i_{i+1}}, p_{j_i}$ and $p_{j_{i+1}}, p_{i_i}$ and delete turns $p_{i_i}, p_{i_{i+1}}$ and $p_{j_i}, p_{j_{i+1}}$.*

The basic mechanism of this operation is the same as in *k–interchange*. If there exists two links which can be replaced by two other links such that equation 3.1 holds and all time constraints are satisfied then merging operation can be performed. Moreover, let $n(g)$ be a number of locomotives servicing circuit g, if equation

$$n(g_{merged}) \leq n(g_i) + n(g_j) \tag{4.1}$$

holds then merging graph operation will result in total cost improvement.

The merging operation can be performed in recursive way. New merged circuits can be merged with another circuit if the conditions described above are fulfilled.

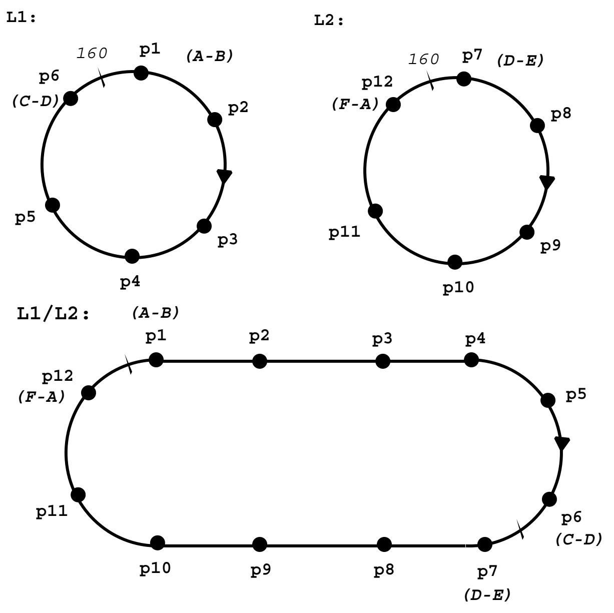

Figure 4.1: An example of merging operation as an optimization step. The circuits for locomotives *L1* and *L2* are merged and the turns $p6 - p1$ and $p12 - p7$ are deleted and replaced by turns with lower cost. The new circuit is serviced by two locomotives. Cursive fonts indicates start and end station for given trip.

The feasibility check while merging can be performed in two steps. The first step is checking for the trivial case. If

$$\begin{aligned} depTime(p_{i_i}) + duration(p_{i_i}) + turnTime(p_{i_i}, p_{j_{i+1}}) \\ \leq depTime(pj_{i+1}) + ct \end{aligned} \tag{4.2}$$

and

$$\begin{aligned} depTime(p_{j_i}) + duration(p_{j_i}) + turnTime(p_{j_i}, p_{j_{i+1}}) \\ \leq depTime(p_{j_{i+1}}) mod ct \end{aligned} \tag{4.3}$$

where

$depTime(p)$	is the departure time of the trip p
$turnTime(p_i, p_j)$	is the time necessary for the turn from turn p_i to p_j inclusive time for necessary deadheads
ct	is the original cycle time

then interchange is feasible and there is no need to adjust departure time windows for all trips. Otherwise checking if interchange is feasible must be performed iterative for all trips in merged circle.

4.3 Local optimization

In this section we are going to present variant of *k–interchange* which can be applied as local optimization method for improving locomotive assignment.

generally, the operation of local optimization can be defined as follows:

Definition 4.3.1 *Improve the locomotive assignment by exchanging turns* $(i,\ldots,i+1)$ *and* $(j-1,j)$ *by an array of consecutive turns* $(k,k+1,\ldots,k+n)$ *if the cost for the relinked tour is lower than the tour cost before relinking.*

Condition for exchange is the same as expressed in [Sav85] and referred in equation 3.1. Nevertheless, instead of applying arbitrary *2–interchange* we allow here to exchange one link by array of links i.e. one link can be interchange with one or more consecutive links. This method can be seen as a variant of Or–interchange with this difference that it does not limit number of consecutive links to 3 but allow any number of consecutive links if and only if sum of the costs connected with those links is lower then sum of costs for deleted links.

$$c_{i,i+1} + c_{j,j-1} > \sum_{i+1 \le k \le j-2} c_{k,k+1} \tag{4.4}$$

Local improvement of the tour can be performed if and only if it reduces total cost of the tour and is feasible with respect to time. Recall from 2.4 that a locomotive circuit is a closed, directed, weighted graph with set of nodes representing trips and edges representing turns between trips. Recall also that circuit represents a trips which are going to be performed periodically, i.e. after accomplishing whole tour a new tour is started. The new tour has the same attributes as departure times and other like previous tour. That means that local improvement of the tour will always result in diminishing the cost if the total time of accomplishing new tour expressed in terms of number of original period frames in the tour is the same like number of period frames before interchange operation was applied.

If exchange of links results in a new tour such that if a turn of time length greater or equal original period frame and costs which can be distributed over neighbor period frames then such frame can be removed from the tour. This means in practice that if original number of locomotives servicing given tour was l_{old} then number of locomotives servicing new tour $l_{new} = l_{old} - 1$ which has great influence on overall cost of the tour.

On the other hand, while checking for the feasibility of interchange we need to check that number of locomotives used for servicing given tour will not increase. This information will be maintained using defined defined in section 3.2 terms $SHIFT$ and PFS. If condition 3.12 does not hold then new period frame must be introduced and the new number of locomotives is higher then original.

The optimization method described above can be also used with the other approach where all operations are applied on set of disjointed graphs. In this case an array of links which can be inserted between two vertices can consist of an array of consecutive links from the same or some other circuit. When some exchange was performed we must also check if any of the inserted links can be exchanged with some other links from other graphs in set. If the order in which graphs were optimized is arbitrary then we must also check if some links in newly optimized circuit can be exchanged with some links in other graphs. The idea of applying *k–interchange* on set of disjointed circuits using arbitrary order is illustrated in Appendix A.

Although this operation seems to find *k–optimal* solution it has a time complexity $O(n^4)$ where n is a number of turns in the set of the locomotive circuits. This complexity includes check for time feasibility for exchange.

4.4 Disjointing graphs

K–interchange applied on the 'grand tour' results in the situation where all the locomotives are used to run all specified trips. Even if such solution is *k–optimal* from the point of view of the 'grand tour' it can still be optimized by removing some unnecessary expensive links. Recall, that we deal here with the situation where servicing the tour is distributed over several 'agents'/'travelers'. Having this is mind we can divide the *k–optimized* grand tour into the set of separate circuits so that instead of servicing all the trips in the tour, an actor services only a subset of the all trips. This operation is formally equivalent with disjointing a grand tour into the set of separate circuits.

To perform disjoint operation we will use the same *k–interchange* method as described in [Cro58] and [Lin73]. In this case we use *2–interchange* where original links with given cost are replaced by two other links with lower cost, but instead of keeping unity of original circuit we transform it in two separate circuits. This idea is illustrated in figure 4.2. This operation can

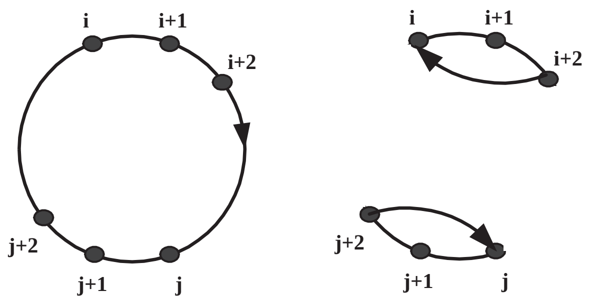

Figure 4.2: Disjointing graphs using *k–interchange*

be performed independently of chosen approach. In any case if operation is applied on a graph in a way where time feasibility for both separated graphs is not violated and equation 3.1 holds then disjointing operation then it follows that sum of used locomotives in both disjointed circuits is lower or equal then number of locomotives in circuit before disjoint and this operation is a optimization step.

4.5 Converting deadheads to passive transports

The last operation in our four step algorithm is to diminish overall cost for the set of circuits by converting possible deadheads into passive transports. We have chosen this operation to be a final step of the algorithm for following reasons. The first reason is based on assumption that the overall gain from performing *k–interchange* operation is higher than the gain achived from converting the deadheads to the passive transports. The deadhead to passive conversion is seen here as a additional operation reducing the overall cost for the set of locomotive circuits, not as an opearation which find the overall cost which is optimal. On the other hand, performing conversion to passive transport in the same time *k–interchange* is performed could open new possibility of relinking locomotive circuits which could result in opening new possibility for cost optimization. Although this possibility is quite fascinating we leave it outside a scoop of this report as possible area for future works.

Another reason for performing this conversion at the end of algorithm is an impact it has at internal representation of the circuits. As we mentioned in chapter 2 we represent set of locomotive circuits as a set of finite, directed weighted graphs with nodes representing trips and edges representing turns between trips for a specified locomotives. We described also that both nodes and edges of circuits have some sets of attributes connected to them. When conversion between deadheads and passive transports is performed initial attributes of graph loses its meaning. A turn between the consecutive trips p_i and p_{i+1} formerly represented by an edge in circuit will be converted into combination: deadhead to trip used for conversion, passive transport, deadhead to start station of p_{i+1}, or using graph representation: an edge, followed by vertex, followed by another edge. This means that cost of turn between p_i and p_{i+1} is distributed on three different elements. This would demand re–defining in which way optimization using interchange operation should be performed and would mean additional constraints which imply additional computing effort.

In the next chapter we are going to explain how to perform deadhead to passive transport conversion and describe implementation of this operation.

Chapter 5

Converting deadheads to passive transports

5.1 Background

Trying to find locomotive circuits with the optimal or near optimal costs often results in circuits with necessary so called deadhead transports i.e. transports where some locomotive is reallocated from one place to another without doing any specific trip.

In real life there is two ways in which such reallocation can be performed. One, most obvious is to drive a locomotive from one station to another, the other is to attach a locomotive to the train performing an active trip and let it follow this trip. This of course assumes that there is such trip to which specific deadhead can be attached. This type of reallocating locomotive, there a locomotive passively follow active trip is called passive transport (see definitions in section 5.2).

There is several advantages of passive transports over deadheads. One of the main advantages is that passive transport is generally 50% cheaper then deadhead which can make significant difference when a locomotive is allocated over long distance. Another significant advantage of passive transport over deadhead is that it does not demand exclusive allocation of track resource. Passive transport uses track resource allocated exclusively for the active trip to which it is attached.

In this chapter we will try to design an algorithm which convert deadheads to passive transports. We will refer to problem domain as pt–problem and to algorithm as pt–conversion algorithm. In the next section we give mathematical formulation of the problem with consideration to real–world constraints.

5.2 Mathematical foundation

Pt–problem can be defined as follows:

Definition 5.2.1 (Pt–problem) *Given the trackgraph G_t and the set of locomotive circuits G_L with their tripsets minimize cost for the set by converting deadheads to passive transports.*

We define terms 'deadhead' and 'passive transport' used in 5.2.1 in following way:

Definition 5.2.2 (Deadhead transport) *A deadhead is reallocating a locomotive from one station to another with exclusive allocation of track resource and without servicing any specific trip.*

Definition 5.2.3 (Passive transport) *A passive transport is a reallocating a locomotive from one station to another by attaching it to the transport servicing a specific trip.*

Given those basic definitions we can define conversion from the deadhead to passive transport. We will call it for deadhead to passive conversion.

Definition 5.2.4 (Deadhead to passive conversion) *Let $\mathcal{L}_i$ be a locomotive with the circuit $Circ_{\mathcal{L}_i}$ and consecutive trips $t_i, t_{i+1} \in Circ_{\mathcal{L}_i}$, with a deadhead between t_i and t_{i+1}. Then let $\mathcal{L}_j$ be a locomotive with the circuit $Circ_{\mathcal{L}_j}$ and a trip $t_j \in Circ_{\mathcal{L}_j}$. If $\mathcal{L}_i$ can be reallocated from end station of t_i to start station of t_j, attached to transport serving t_j and then reallocated from end station of t_j to start station of t_{i+1} such operation is called pt–conversion.*

The utility of deadhead to passive conversion can be formalized in terms of a cost function. Let c_d be a deadhead cost and c_{pt} cost for the passive transport. Conversion of any deadhead i with cost c_{d_i} into passive transport can be described as

$$c_{d_i} \Rightarrow c_{d_j} + c_{pt} + c_{d_k} \tag{5.1}$$

there

$$c_{d_j}, c_{d_k} \geq 0$$

Note that in our definition conversion can only be performed by other active transport. We do not consider in our definition using one deadhead to piggy–back another deadhead as deadhead to passive conversion. We leave this problem for future consideration.

Deadhead to passive conversion can be carried out until cheapest conversion is found, although, if applied consecutively on the set of deadheads it will result in locally cheapest conversion. Fixing a conversion for a deadhead will, in most cases, have influence on departure time windows for other

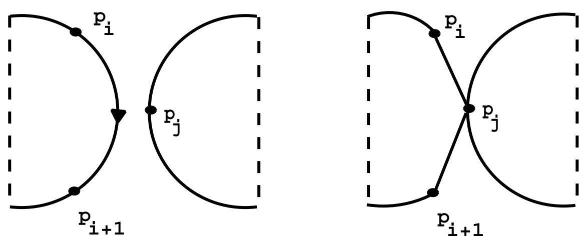

Figure 5.1: Deadhead in turn from p_i and p_{i+1} is converted to passive transport using a trip p_j.

trips in the locomotive circuit containing given deadhead. Moreover, it will influence time window for the active trip used in conversion and all trips in its locomotive circuit. For detailed discussion of those issues see section 5.5.

We will call such result for local pt–optimality which we define in following way:

Definition 5.2.5 (Local pt–optimality) *A set of locomotive circuits is said to be locally pt–optimal if and only if there is not any deadhead transport which can be converted to passive transport minimizing overall cost for set of circuits.*

Ideally, the deadhead to passive conversion would be performed in the way which would minimize overall cost for a set of locomotive circuits. It means that while performing conversion we would not choose a cheapest alternative for a deadhead we are going to fix but compare every possible conversion with all the other possible conversions for every deadhead. Carrying out optimization in such way will rapidly increase the computational cost of the conversion (see section 5.3).

The conversions optimality is measured with respect to an initial set of circuits. The deadhead to passive conversion does not minimize overall cost for a set of circuits, it may still be a possibility to minimize cost of the initial solution or result of the conversion by eliminating e.g. some deadheads.

The deadhead to passive conversion in turn try to convert all the possible deadheads even those which could be removed in the earlier optimization step, before conversion.

The deadhead to passive conversion is not reversible, i.e. if some deadheads were converted to the passive transports, then there is no possibility to eliminate converted deadhead by replacing it by any active transport or replacing any remaining deadhead transport by an active one. This is due to the change of the topology of the graph representing the locomotive circuit containing converted deadhead (see section 4.5).

While performing deadhead to passive conversion we must take into consideration some of the real world constraints.

1. A deadhead can be converted into a passive transport if a part or a whole of deadhead path covers a whole path of some active transport. In other words, a deadhead can be attached and detached from some transport only on the start respective end station of an active transport servicing given trip.

2. A deadhead can use one and only one active transport. In real life deadhead to passive conversion demanding more then one active trip is extremely rarely performed. This is because service time connected with detaching locomotive from one train, reallocating it on another track and attaching to another active trip is to expensive and demands to much time comparatively to the cost of the deadhead.

3. A deadhead does not initially have to be in active transports start station. One deadhead can be converted to a combined deadhead – passive transport.

4. A deadhead does not have to have same end station like active transport. A deadhead can be converted to a combined passive transport – deadhead.

5. Condition 3 and 4. A deadhead can be converted to a combined deadhead – passive transport – deadhead.

6. The conversions 3,4 and 5 may be performed if and only if they do not violate time windows for any trip in any of the locomotive circuits.

7. The conversions 3,4 and 5 may be performed if and only if the overall cost for servicing the set of locomotive circuits after deadhead to passive conversion is lower then initial cost for servicing the set of locomotive circuits.

8. One active transports can not have more then one locomotive following passively. Some of the trips does not allow passive transports.

Given those basic definition and assumptions we can formalize our algorithm in form of objective function. Let $\mathcal{K}$ denote turn between two trips, $x_{(ij)}$ number of locomotives turning from one trip to another and $c_{(ij)}$ a cost connected with performing such turn, then:

$$min \sum_{(i,j)\in\mathcal{K}} c_{(ij)}x_{(ij)} \tag{5.2}$$

Given an assumption of a constant cost for a waiting time (see section 2.2) a cost of turn can be expanded according to definition 5.1 as:

$$c_{(ij)} = d_{(ij)} + pt_{(ij)} \tag{5.3}$$

where $d_{(ij)}$ is a sum of the costs for performing deadhead transports while turning from i to j and $pt_{(ij)}$ is a cost of a passive transport in a turn (ij). According to this expansion the objective function can be rewritten to:

$$min \sum_{(i,j)\in\mathcal{K}} (d_{(ij)} + pt_{(ij)})x_{(ij)} \tag{5.4}$$

5.3 Conversion algorithm

In this section we are going to present a deadhead to passive conversion algorithm. The aim of this algorithm is to reduce overall cost for a set of locomotive circuits according to objective function given in equation 5.4.

Algorithm 5 Deadhead to passive conversion

```
Require: track graph $G_t = \{Station, Track\}$
Require: set of feasible locomotive circuits $G_{Lc}$
 1: for all deadheads $d \in G_{Lc}$ do
 2:    compute time window $TW_d$ (section 5.3.1).
 3:    create a set of stations $SS$ such that $s \in SS \rightarrow c_{d_{start,s}} + c_{pt_{s,next}} < c_d$,
       where $c_{d_{start,s}}$ is a cost for new deadhead to the station $s$, $c_{pt_{s,next}}$ is a
       cost for passive transport from $s$ to the station where next trip starts
       and $c_d$ is a current cost of deadhead (section 5.3.2).
 4:    create list of transports $CT$ which can be used for conversion of $d$ to
       passive transport such that
       $t \in CT \rightarrow startStation_t \in SS \wedge depTime_t < depTime_{nextTrip} \wedge$
       $c_{d_{start,s}} + c_{pt} + c_{d_{end\_pt,next}} < c_d$ (section 5.3.3).
 5:    for all candidates $ct \in CT$ do
 6:       compute cost for using $ct$ to convert $d$ into a passive transport.
 7:    end for
 8:    sort all $ct \in CT$ according to their cost in ascending order (section
       5.4).
 9: end for
10: sort all deadheads according to the size of their candidate list $CT$ in
    ascending order (section 5.4).
11: reduce the overall cost for $G_{lc}$ by finding a feasible conversion of all
    $d \in G_{lc}$ into the passive transports. Start a search choosing the best
    possible conversion for a deadhead $d$ with lowest number of candidates
    $ct \in CT_d$ (section 5.4).
```

This algorithm is designed to find a feasible deadhead to passive transport conversions for a set of deadheads. Computing a time window for a given deadhead is performed in constant time $O(1)$. The set of stations for a given deadhead demands checking for every station $s \in S$ relation between cost of a deadhead and distance to s. If $|S|$ is a number of stations in S then this step performs in $O(|S|)$. Performing next step requires checking for every trip if it starts and ends in a station $s_n \in SS$ and if has its departure time within time window for a deadhead. If $|P|$ is a number of all trips then this step performs in time $O(|P|)$. All those steps are performed for every deadhead i.e. it performs in time $O(|D|)$ where $|D|$ is a number of deadheads $d \in G_{lc}$.

Reducing of the overall cost is implemented using branch–and–bound algorithm. The reduction can be carried out to the point where the overall costs for the set of circuits is minimimal. Below we discuss the execution time complexity for such reduction where we use naive search strategy, i.e. where the candidates in candidate list for every deadhead are not sorted according to their costs and the set of deadheads is not ordered according to the number of candidates in their candidate list.

Let $|D|$ be a number of all deadheads, c_j a list of candidates for given deadhead $d_j \in D$ and

$$M = |\bigcup_{j=1}^{|D|} c_j|$$

then in worst case algorithm must search over $\binom{|D|}{M}$ times which gives time complexity $O(2^{|D|*M})$.

If the goal of conversion is defined as reducing the overall cost for the set of circuits by finding any feasible set of conversions then the conversion can be carried out in a lexicographic order which yields a polynomial worst execution time complexity $O(|D| * M)$.

The strategy of ordering the candidate transports which can be used for converting given deadhead and ordering the set of deadheads according to the number of such candidate transports in their candidate lists, described in steps 8 and 10 of the algorithm 5 and this of choosing the best possible conversion for a given deadhead gives possibility of finding a locally optimal conversion cheaper than a solution found using a naive search strategy.

5.3.1 Computing time window for deadhead

A deadhead d can start from its departure station $s_{d_{start}}$ earliest at the time equal the time of earliest arrival of previous trip at $s_{d_{start}}$. The latest possible departure time can be computed by substracting from the departure time for next trip, which starts at station s_{next} a time which is necessary for traversing distance from $s_{d_{start}}$ to s_{next}. If the departure time is expressed by departure time window then the latest possible departure from $s_{d_{start}}$ is defined as latest possible departure from s_{next} minus time necessary for traversing distance from $s_{d_{start}}$ to s_{next}.

This step will limit the search space for trips which can be used in converting a deadhead to a passive transport to those trips which starts and ends in a deadheads time window.

5.3.2 Creating set of stations

The aim of this step is to create set of stations which can be taken into consideration when searching for trips which can be used for converting a deadhead into a passive transport. This poses a restriction on the search space by defining stations where such a trip must start.

From the general cost relation passive transport – deadhead we can define a set of trip–start stations S_0 as subset of all stations such that cost for deadhead to $s \in S_0$ plus cost of passive transport from s to the station where next trip starts is lower then cost for initial deadhead:

$$s \in S_0 \Rightarrow c_{d_{start,s}} + c_{pt_{s,next}} < c_{d_{start,next}} \tag{5.5}$$

5.3.3 Creating transport candidate list

The aim of this step is for every deadhead $d \in G_L$ with the time window TW_d and the set of candidate stations S create list of active trips $CandList$ which may be used to convert deadhead d to passive transport.

Creating candidate list for deadhead i is subject to following constraints:

$$\begin{aligned} at \in CandList \rightarrow & \\ & at_{start} \in S \wedge \\ & at_{end} \in S \wedge \\ & c_{d_j} + c_{pt} + c_{d_k} < c_{d_i} \wedge \\ & depTime_{at} \in TW_d \end{aligned} \tag{5.6}$$

there at_{start} and at_{end} are start respective end stations for active trip, C is a cost of transport, $depTime_{at}$ is departure time for the trip and TW_{dh} is departure time window for the deadhead.

The first two constraints in this conjunction state that for every active trip in the deadheads list of possible candidates for use in conversion it must start and end in one of the stations in the station candidate set. As we mentioned in 5.3.2 set of candidate stations S is created in such way that cost of deadhead from original deadheads start station to $s \in S$ plus cost of the passive transport which goes from s using shortest path to station there next trip starts is lower than the cost of the original deadhead. This equation can be rewritten as a cost of a deadhead to $s \notin S$ and then a passive transport from s to the start station for the next trip must be greater then the cost of the original deadhead. From this follows that if the end station of the passive transport is not an element of the set of the candidate station then the cost of the deadhead from the end station of the passive transport to the start station of the next trip is greater than the cost of the original deadhead and the active trip which such features is not a candidate for conversion.

Additional constraint $c_{d_j} + c_{pt} + c_{d_k} < c_{d_i}$ limits further the list of possible candidates which can be used in conversion. It states that if the cost of the deadhead to the start station of an active trip together with the cost of the passive transport from to its end station and the deadhead from the end station to the start station for the next trip must be lower than the cost of the original deadhead.

The last part of equation states that the active trip which is used in converting a deadhead to a passive transport must have a departure time within a time window for a given deadhead.

5.4 Searching for conversion

5.4.1 Background

Dealing with scheduling optimization generally imposes on any optimization algorithm the necessity to deal with the time feasibility. If any task t_i put the exclusive lock on the resource/machine in the time when it is performed and t_i may not be preempted then next task t_{i+1} can not be started until t_i is finished and an exclusive lock on a resource/machine ceases.

Parallel with the passive transport problem is apparent here: if the locomotive is reallocated from one station to another then it can start next trip. To start any trip a locomotive must obviously fulfill constraints of being available at the station where a trip is going to start at the departure time of the trip.

When the timetable for the locomotive circuits is fixed then the feasibility check with respect to time is simple. If we know that next departure is going to be done in time T then start time for the deadhead + time necessary for passive transport must be earlier/lower then departure time for the next trip. We know also that if the conversion demands an initial deadhead from the start station of the original deadhead to the station where an active trip used for conversion starts then start time for such deadhead + traversal time must be lower or equal to the departure time of the active trip[1].If any of those rules are violated then timetable for the trips is violated and such conversion is unfeasible with respect to time.

On the other hand, if the departure time for active trip and departure time for the next trip are not violated then, out of transitivity rule, departure

[1]Actually, departure time for deadhead is always defined as time window

times for the trips after active trip and departure times for trips after next trip will not be violated. Such features of a schedule with fixed timetable make possible to use local search methods when searching for feasible conversion of a deadhead to a passive transport. The only time feasibility check we need to do is performed on defined deadhead/candidate passive transport.

When it comes to schedules where departure times are defined in terms of time windows using local search methods seems much more difficult. Checking a feasibility of conversion with respect to the time must be defined in terms of time interval. Because size of the departure time windows for the single trips in circuits differs then it is not enough to check if conversion performed on one deadhead does not violate departure time window for the trip which follows deadhead. Even if simple conversion may be feasible with respect to time it may limit departure time window for the next trip in such way that it would be impossible to some of the trips which follows them. We must also take into consideration that one locomotive circuit can have more then one passive transports. Performing local time feasibility check and accepting one of the conversions may cause other candidate conversions unfeasible. On the other hand, using any active transport in conversion may limit its departure time window in the way that it violates feasibility of the active trip circuit or the feasibility of other circuits which interact with active trip circuit in converting some other deadhead.

In any case performing deadhead to passive conversion to minimize overall cost for a set of locomotive circuits impose on the algorithm necessity to know global relations between trips, deadheads and circuits, otherwise it carries a risk of finding only locally optimal solution.

5.4.2 Converting schedule with fixed timetable

The aim of this section is to present an algorithm for deadhead to passive transport conversion for the fixed timetable. The aim of the algorithm is to perform such conversion in globally optimal way. Note that this is a method alternative to this described in [Dro97].

Through the previous steps of conversion algorithm there was created set of candidate deadheads $D = \{d_1, d_2, \ldots, d_n\}$ where every deadhead $d_i \in D$ has a list of candidate transports which could be used in conversion. Deadhead d_i is constraint with the time window in which it must be reallocated from end station of the previous trip $s_{end(p)}$ to the station where the next trip of locomotive with d_i is going to service $s_{st(p+1)}$. This time window in case of fixed time table has a lower bound equal arrival time at at $s_{end(p)}$ and upper bound is the time when $p+1$ starts minus time necessary to traverse distance between $s_{end(p)}$ and $s_{st(p+1)}$. We assume here that velocity at which such allocation occur is constant. Further, we assume that velocity of any kind of transport i.e. active transport servicing trip, deadhead transport and passive transport is equal. The traversal time used in computing upper bound of time window of d_i is a traversal time from $s_{end(p)}$ to $s_{st(p+1)}$ using shortest path between those two stations.

Because of the constant velocity assumption and shortest path assumption we know that if there exists such active transport p_i which starts at the station $s_{end(p)}$ and ends at the station $s_{st(p+1)}$ but outside the range of time window interval for d_i then such conversion will be unfeasible because it will violate departure time for $p+1$.

Fixed time table imply that checking feasibility for departure time constraint can be performed locally. It implies that if pt–conversion of d_i is feasible with respect to departure time constraint for $p+1$ then feasibility for all circuits is maintain. On the other hand, if departure time of $p+1$ is violated then whole conversion is unfeasible.

Given these assumptions we can perform our conversion locally. Nevertheless, because of our pt–optimality definition we can not perform our conversion on only one deadhead at the time. The reason for this is as follow. Assume that there exists a deadhead $d_i \in D$ with a list of candidate transport which could be used in pt–conversion $C_{d_i} = \{c_1, c_2, \ldots, c_n\}$. Further, assume that there exists another deadhead $d_j \in D$ with the candidate list $C_{d_j} = \{c_1, c_2, \ldots, c_n\}$, and that some $c_i \in C_{d_i}$ is equal some $c_i \in C_{d_j}$. In other words we have a situation where two deadheads can use the same active transport for pt–conversion, which violates constraints stating that one active transport can be used for conversion of one and only one deadhead to passive transport. If we try to convert one deadhead at the time, then we can not be sure that our conversion will be pt–optimal.

In our algorithm we use a simple divide and conquer method to divide search space and limit searching for conversion only to those deadheads which are absolutely necessary to take into consideration.

Choosing deadheads which will be in the set of deadheads for local conversion has to fulfill following condition.

Let D be a set of all deadheads and $d_i \in D$ where d_i has a candidate list C_{d_i}. Then let $L \subseteq D$, where L denotes candidates for local conversion and $d_j \in L$ be a candidate deadhead for local conversion with the candidate list C_{d_j}.

$$\forall d_i \in D, \forall d_j \in L \; (\; \exists c \; (c \in C_{d_i} \land c \in C_{d_j} \rightarrow d_i \in L)) \tag{5.7}$$

Deadheads which undergone local pt–conversion are removed from the set of candidates for conversion

$$D_{new} = D - L. \tag{5.8}$$

and

$$\forall d_i \in L, \exists c_i \in C_{d_i} \rightarrow \forall d_j \in D_{new}, \exists c_j \in C_{d_j} \land c_i \neq c_j \tag{5.9}$$

In other words there not exists any deadhead $d_j \in D_{new}$ which have in its candidate list a candidate used by any d_i chosen for local conversion.

The set of deadheads for local conversion is then passed to constraint solver and optimized using minimizing constraint and branch–and–bound search. More about constraint solver in section 5.6

5.5 Conversion with departure time windows

As we mentioned in 5.4.2 deadhead to passive transport conversion performed on timetable with fixed departure time can be performed locally. Subdividing of search space for optimization follow the rule 5.7. This rule will not apply to timetable where departure time is defined in terms of time window. The reason for it is that constraining time window for departure for p_{i+1} will have an influence on the departure time window for p_{i+2} and all consecutive trips up to some p_n with unchanged departure time window. It will also effect a time window of an active trip used to convert given deadhead to a passive transport and time windows with precedence constraint.

Moreover there exists a possibility that some of the trips in the circuit containing converted deadhead will later be used as a candidate for conversion for some other deadhead, say d_k, in another circuit, say C_{L_k}. In this case checking for feasibility while converting d_i would constrain time windows for $d_k \in C_{L_k}$ and all the trips in C_{L_k}. This dependency would in turn make it necessary for checking feasibility of conversion with all other circuits with wich C_{L_k} interact.

Although it is possible to limit such conversion space into some more or less local subspaces which would include all the circuits which interact during the conversion we assume that such method on one hand would increase computational complexity, on the other hand it is often the case that such multiple bindings between different circuits would result in a subspace equal a total search space. The experience with goods example supports this assumption (see chapter 5.7).

Therefore we decided to implement in case of timetable with time windows a global conversion which operates on whole conversion search space.

5.6 Implementation

5.6.1 General issues

In this section we are going to describe our implementation of deadhead to passive transport conversion algorithm for the case where departure times for the trips specified by time windows. We have chosen to implement this algorithm using SICStus prolog version 3.8.4. The code should run also on later versions of SICStus.

Algorithm takes as an input initial solution from the trip scheduler, which is of form

```
solution(Solution)
```

where `Solution` is a list of locomotive turns of form

```
[LocomotiveId,[ListOfTurns]]
```

and `ListOfTurns` is a list of locomotive turns performed by the locomotive `LocomotiveId`. Each turn in `ListOfTurns` has the format

```
[TurnFromTrip,TurnToTrip,Cost]
```

where `TurnFromTrip` and `TurnToTrip` are trip id's occurring in trip specification.

The list of turns for the different locomotives does not have to have an uniform length. The time for accomplishing any circuit can be different and can even span over several cycle times which means that such circuit is in practice serviced by several locomotives . In such cases the `LocomotiveId` represents in fact more than one locomotive (see section **??**).

Specific trips are given using the predicate

```
trip(TripId,StartStation,EndStation,DepartureTW,DepTime)
```

where `TripId` is a unique trip id. Specifying `TripId` follows general rules in **TUFF** where every trip has a unique number. The trips are numbered as a serie of integers $\{1, 2, \dots, i, i+1, \dots, n\}$. This implementation of the conversion algorithm is heavily dependent on the fact that trips are numbered in increasing order. `StartStation` is a station where the trip starts and `EndStation` is a station where the trip ends. If $G_t = \{S, E\}$ both `StartStation` and `EndStation` must be in S. `DepartureTW` specifies departure time window parameters defined as `DepTWStart..DepTWEnd`, i.e. its lower and upper bound. `DepTime` specifies the fixed departure time used in `Solution`[2].

As we can notice we do not specify arrival time at end station nor the path (route) of the trip in our input into algorithm. This simplification is due to assumption that a locomotive attached on an active trip must follow this active trip from its start to the end station. We assume in our implementation that trip p_i from s_{st} to s_e can be performed using the shortest path from s_{st} to s_e. How the shortest path between two station is computed see 5.6.3 which also specifies how traversal costs between stations are measured.

The second input parameter to the algorithm is a trackgraph $G_t = \{S, E\}$ where S is a set of stations/vertices and E are edges between $s_i, s_j \in S$, i.e. single tracks between stations. G_t is specified as database of tracks using predicate

[2]In fact this parameter is unnecessary and will be soon removed. In **TUFF** system departure time window used in trips specifies departure interval in terms of `Solution` and used 'fixed departure time' is set to the lower bound of departure time window, i.e. `DepTWStart`

```
track(FromStation,ToStation,Cost)
```

The edges between stations are not symmetrical, i.e. the cost of traversing edge from s_i to s_j is not the same as the cost of traversing edge from s_j to s_i. Assuming that all locomotives are of uniform type we do not maintain information about the cost for traversing a given edge for different locomotive types. The cost for traversing an edge between to stations is in our model a function of the traversal time. As we assume that all the locomotives used in the solution are of uniform type the cost used here is equal to the time necessary for traversing the given edge.

We also need to mention that despite different length the time resolution can be arbitrary but it must be uniform in the whole problem for trips, turns and costs for traversing distance between two stations.

In our implementation we choose to read all these parameters from a separate prolog file.

5.6.2 Cyclic time

As mentioned in 5.6.1 we deal here with a set of trips performed periodically in a given time period. We also mentioned in the section above that the length of the circuits contained in a given set does not have to have to be uniform but can span over one or more cycle times. It means in practice that a locomotive with given `LocomotiveId` which has its `ListOfTurns` spanning over given cycle time *ct* represents in reality more then one actual locomotive. The number of actual locomotives x_l which given `LocomotiveId` represents is given by

$$x_l = \begin{cases} lt/ct & \text{if lt mod ct} = 0 \\ lt/ct + 1 & \text{if lt mod ct} > 0 \end{cases} \tag{5.10}$$

where lt is a arrival time at the station specified as destination of last trip in the `ListOfTurns` and / is an integer division.

The the cycle time is stored using the predicate

```
cycleTime(CT)
```

where `CT` is the given cycle time in the resolution used in trips, turns and cost for traversing the edges of the trackgraph.

This possibility of creating locomotive circuits that can span over several cyclic time periods influence the computation of for example a list of candidate transports for a given deadhead. The time window for given deadhead d_i expressed by $TW_{d_i} = [tw\_start_{d_i}, tw\_end_{d_i}]$ is actually defined by $TW_{d_i} = [tw\_start_{d_i} modCT, tw\_end_{d_i} modCT]$ where CT is the cycle time and $tw\_start_{d_i}$ and $tw\_end_{d_i}$ are the lower and the upper bounds of the time window for deadhead d_i.

5.6.3 Creating the distance matrix

Computing a deadhead to passive transport conversion demands a matrix with the distances between the stations in which the given deadhead is reallocated. This matrix is computed from the given track graph G_t. To compute matrix we use Dijkstra's algorithm run all–to–all.

Recall from the previous section that we deal here with a graph where distances between vertices are not symmetrical, i.e. $distance_{s_i,s_j} \neq distance_{s_j,s_i}$, thus the algorithm run in time complexity $O(|S|^3)$ where $|S|$ is a number of vertices/stations in the graph.

Although right now computation of the distance matrix is done in run–time it is much more advantageous to use the already existing distance matrix, created earlier off–line. This would remarkable diminish the run–time of the algorithm.

Created distance matrix is stored in the prolog database using the directive

```
assertz(distance(From,To,Distance))
```

and is accessed with

```
distance(From,To,Distance)
```

The distance database is used during the whole conversion and flushed when conversion is finished using directive

```
abolish(distance,3)
```

5.6.4 Computing time windows for the deadheads

As we mentioned in 2.2 the cost of a turn between two trips consist basically of the cost for the waiting time and the cost for the deadhead from end station of the first trip to start station of the next trip. Assuming that the cost for the waiting time is linear then it can not be optimized: decreasing the waiting time in one turn must increase the waiting time for some other turn in the same circuit. Therefore the waiting time parameter may be completely omitted.

Because the cost of the waiting time can not be optimize we omit it in our specification of the cost for the given turn. Thus if any turn has a cost greater then zero it means that such turn containes a deadhead. This information is read from the initial solution, given as an input to the conversion algorithm.

Knowing that it is necessary to perform deadhead transports between trip p_i and p_{i+1} we compute the time window for this deadhead by picking information about p_i and p_{i+1} from the trip database and information about the traversal time of p_i and the traversal time from the end station of p_i to the start station of p_{i+1}. The deadhead's time window is computed as described in 5.3.1. This computation is implemented in the procedure

```
makeDHTW(FromTurn,ToTurn,MaxTW,ActualTW)
```

5.6.5 Creating the set of candidate stations

The set of candidate stations is created using the predicate

```
createStationCircle(Turn,StationCircle)
```

where `Turn` is a list `[FromTrip,ToTrip,Cost]`, i.e. the same format as `Solution`'s `TurnList`. After computing the start station of the deadhead a circle including all stations that can be searched for the transports which can be used for conversion is computed. Computing set of such stations follows the rules in equation 5.5. In the set of candidate stations are included all stations lying in the cost range

$$dt * 4 + df * 2 < Cost$$

where *dt* is the cost,'distance' from the start station of a deadhead to the given station and *df* is the cost,'distance' from the station to the start station of the next trip following the deadhead. In other words, we assume that if there is an active transport which can be used in the conversion and is going directly from the computed station to the station where next trip starts it will be cheaper to reallocate the given locomotive to this station as a deadhead and then use an active transport from this station to transport the locomotive. Computing the set of candidate stations is performed using `distance/3` database.

5.6.6 Creating the list of candidate transports

Given a deadhead, its departure time window and the set of candidate stations, a list of active transports which may be used in the conversion of this deadhead is computed according to equation 5.6. Computing the candidate transport list is done with the predicate

```
makeCandidateList(  CurrentStation,Destination,ActualTW,
                    AllStations,StationCircle,CandidateList)
```

where

`CurrentStation`	is the station where deadhead starts
`Destination`	is the start station for trip following deadhead
`ActualTW`	is the departure time window for the deadhead
`AllStations`	is the set of candidate stations. This list is used for checking if the end station of any candidate trip lies in cost range.
`StationCircle`	is the set of candidate stations and is initially equal the `AllStations`. This list

	is parsed while searching for trips starting in $s \in StationCircle$
`CandidateList`	list of trips that can be used during the conversion.

Predicate uses

```
mod_compare(TWTStart,TWTEnd,EarliestTime,LatestTime)
```

where

`TWTStart`	is the lower bound for the candidate transport departure time window
`TWTEnd`	is the upper bound for the candidate transport departure time window
`EarliestTime`	is the lower bound of the departure time window for given deadhead
`LatestTime`	is the upper bound of the departure time window for the given deadhead

which checks if the departure time window for the given candidate transport lies in the range of the deadhead's departure time window using modulo operation. The `mod_compare/4` take into consideration cyclic time.

5.6.7 Final conversion

Given all deadheads with their departure time windows and list of candidate transports the search for conversion is performed. Our search implements search for conversion where the departure time for the trips is specified as the time windows. Search is performed globally on all deadheads (see section 5.4). This part of the program is implemented using the constraint solver for finite domains (`clpfd`) included in SICStus prolog. Search for conversion is called with predicate

```
run_search(List,Result)
```

where `List` is a list of deadheads with the following format:

```
[LocomotiveId,TurnFrom,TurnTo,Cost,TimeWindow,CandidateList]
```

where

`LocomotiveId`	is the unique number for the locomotive performing the given deadhead
`TurnFrom`	is the trip id the locomotive is turning from
`TurnTo`	is the trip id the locomotive is turning to
`Cost`	is the initial cost of the deadhead
`TimeWindow`	is the deadhead's departure time window from the end station of `TurnFrom`. `TimeWindow` is specified as the interval

	`EarlierstDepartureTime..LatestDepartureTime`
`CandidateList`	is a list of active transport which can be used for converting deadhead into passive transport.

and `Result` is the converted list of deadhead.

As we mention in section 5.4 there is some constraints which should be fulfilled while searching for a proper conversion. All these constraints are posted before the actual search starts.

Feasibility of a conversion with respect to time data is performed in two steps. First by posting constraints and data essential for such control, second, by performing actual feasibility check during the search for a conversion.

To properly perform the feasibility check for conversion we post constraints for all the trips and circuits, which contains data about their departure time windows, travel times and actual number of locomotives used in the circuits. To do this we need to expand the information contained in `solution(Solution)` with the actual number of locomotives used in every locomotive circuit in `Solution`. This information is obtained by the predicate

```
nof_engines(Circuits,OrigEngines)
```

where `Circuits` is the initial solution containing all circuits (equivalent with `Solution`) and `OrigEngines` is a list containing the number of locomotives used in each circuit. Then all trips and circuits are posted in

```
init_circuits(Circuits,DepDoms)
```

which in turn calls

```
post_trips(FirstTrip,LastTrip,DepDoms)
```

where

`FirstTrip`	is the unique number of the first trip
`LastTrip`	is the unique number of the last trip
`DepDoms`	is the list of posted departure domains for all trips

This implementation is heavily dependent on the standard way of assigning unique ids to the trips , where trips ids are serie of consecutive integers $\{1, \ldots, i, i+1, \ldots, n\}$.

The list `DepDoms` is created when the departure time interval for every trip is accessed by the call to `trip/5` database. Then the departure times are converted using the uniform cycle time by the modulo operation. Each departure time domain is then posted as

```
Dep in (DepStart..DepEnd)
```

where `DepStart` and `DepEnd` are the lower and upper bound of the departure time window for the given trip after applying the modulo operation as described above, or as a disjunctive domain

```
Dep in ((0..DepEnd)\/(DepStart..CT))
```

if the trip crosses the specified cycle time.

All the circuits are posted with constraint that the actual number of locomotives used in converted circuit may not be higher then the actual number of locomotives in the original circuit. The time windows constraints posted here will be necessary to check if conversion of the deadhead is feasible with respect to the time. These constraints will be later used by `fd_contained`.

The deadhead's domain is posted using predicate

```
create_domain(List,CurrentDomain,DHId,Domain)
```

where

`List`	is the list of deadheads which are to be converted
`CurrentDomain`	is the domain of variables already posted (used for tail recursion)
`DHId`	is the unique id for original deadhead
`Domain`	is the complete domain of posted variables. The return variable for tail recursive `create_domain/4`.

The `Domain` is a list of posted deadheads. The value of every variable in `Domain` consist of the list of the active transports which can be used to convert the given deadhead into the passive transport, which are represented by id's of the trips they service, and the special value which is the deadheads temporary generated id. The value of variable is expressed as a disjunction of the terms. The unique id of the deadhead garantee termination of the program: if none of the active transports can be used to convert the deadhead into the passive transport then the variable representing the given deadhead takes the value of the deadheads id.

Every candidate transport for a deadhead has a cost associated with it. This cost is posted with a relation list created for every deadhead. To create this relation the predicate `relation(?X,+MAPLIST,?Y)` from SICStus prolog is used, where `X` and `Y` are integers or domain variables and `MAPLIST` is a list of `INTEGER-CONSTANTRANGE` pairs, where an integer key occurs uniquely (see [SIC00]). Every deadhead has a map list which contains pairs `Candidate-Cost` where `Candidate` is an active transport which can be used for converting the deadhead to passive transport or the original deadhead and the `Cost` is a cost for converting the transport using this `Candidate`. This relation is posted inside the predicate `sum_cost/6`.

`sum_cost/6` is also used to post some other constraints which will be used for checking time feasibility of the conversion.

Search strategy

Converting the set of deadheads into the passive transports demands a very high comutational costs. . Finding globally optimal solution for a problem with many deadheads having large candidate lists can take unproportionally long time. Because of that we decided to implement search for finding a solution with cost lower or t worst equal initial solution to the initial solution.

To perform the search we use the generic SICStus predicate `labeling(:OPTIONS, +VARIABLES)` where

`OPTIONS`	is a list of search options
`VARIABLES`	is a list of domain variables or integers

Because we do not do a complete search for an optimal solution it is important to find one as profitable as possible. To do this we decided to use a greedy strategy while searching for a solution, i.e. we label a domain variable with the candidate with lowest conversion cost thus maximizing the gain. Choosing of value which is going to be assigned to on variable is done with the option `value(enumerator)` (see SICStus manual [SIC00]), where `enumerator` is a predicate called as `enumerator(X,REST,BB)`

`X`	is the domain variable
`REST`	is the list of variables which needs labeling, except `X`
`BB`	is a bound called by the auxiliary process `apply_bound` to ensure that branch–and–bound works correctly

In our implementation `enumerate/3` finds cheapest candidate transport by reading `costs/1` attributes of domain variable posted earlier in `sum_cost/6`,which equals the list used in `relation/3` described earlier.

`labeling/2` supports several options for choosing domain variable strategy. In our implementation we choose to use the first fail strategy (`ff` option in SICStus). First fail chooses those variables with the smallest domain. If there exists several variables with the same domain size the leftmost variable is chosen.

The strategy of choosing variable can be improved by using the option `variable(SEL)`, where `SEL` is a predicate to select the next variable. In this way we could find the best possible domain to be label in given phase of labeling. We leave this option to be implemented in the future.

Checking for time feasibility

While searching for a solution `labeling/2` checks if a given fulfill time feasibility constraint. This check is performed using earlier posted constraints for the trips and circuits time windows as well as constraints posted in `sum_cost/6`. The departure time window constraints for the given candidate are accessed using predicate

```
element(H1, [DHDom|DepDoms], CandDepDom)
```

where

`H1`	is a given candidate for conversion
`DepDoms`	is the departure time windows for all trips
`DHDom`	is the departure time window for the original deadhead
`CandDepDom`	is the departure time window for the choosen candidate

The departure time window for the candidate is then checked using `fd_contained/3` against the deadhead's time window. `fd_contained` checks if the given conversion will result in a transport which can start inside deadhead's time window and will be finished before next trip will start.

5.7 Computational results

The results presented here comes from our implementation of deadhead to passive transport conversion algorithm using SICStus prolog described in 5.6. Test was run on Digital HiNote with 266 MHz processor.

Specification for trips, tracks and solution was filtered out of original output from **TUFF** system and stored in appropriate files. Referring computational result we are going to concentrate on the last part of algorithm where we search for solution because it is the part which has greatest execution time complexity.

We start with the file specifying 68 trips and solution distributed on 4 locomotive circuits using totally 15 locomotives. The first part of algorithm finds 28 different deadheads which are possible to convert to passive transports. The number of candidates trips which can be used for conversion vary from 1 to 30. First we try to minimize conversion running algorithm using `minimize/2` with `labeling/2` search function using first fail strategy. We use greedy approach to bound branched domain space.

A time to perform those 28 deadheads is 19726 minutes, i.e.328 hours and 46 minutes which multiplied with 4 which is a cost for a deadhead per time entity gives a cost of 78904.

The first solution is found very quickly, it takes less then 1 second. Its costs is 37280 which divided by constant cost for a deadhead gives 9320 minutes ,i.e. 155 hours and 20 minutes. This is a cost improvement equal 173 hours and 46 minutes. In this solution 19 deadheads were replaced by passive transports and 9 kept their original deadheads.

Algorithm find next solutions but the time of finding better then previous solution is growing. After 52 minutes solution with the cost 34064 which is 8516 minutes, i.e. 141 hours and 56 minutes, which is 13 hours and 24 minutes better then first solution found. In this solution there is, like in first one, 19 deadheads replaced by passive transports and 9 which kept original deadheads. Because program has not terminated during long time we decided to finish it.

The main conclusion of running algorithm with minimizing function on this specification is that even if there are considerable gains which comes out running branch–and–bound algorithm on this specification it is not comparable with computational cost. The computational cost which follows is a result of general complexity of the problem.

Its worthwhile to notice that all 28 deadheads depend strongly on each other so dividing search space according method described in 5.4.2 results in all deadheads clustered together.

The next test specification we used is one with 211 trips and initial solution distributed on 4 locomotive circuits serviced by 114 locomotives. Algorithm finds 29 deadheads which can be converted to passive transports. The total cost of those deadheads is 9376, which gives 2344 minutes in the time cost. In this case the first conversion found is optimal. Time used to find solution is less then 1 second. The cost after conversion is 8290, which gives 2072 minutes. Conversion results in total cost improvement equal 272 minutes, i.e. 4 hours and 32 minutes. In this solution there is only 14 deadheads which were converted. 15 of original deadheads were preserved.

Finding the optimal conversion in this example is due to the configuration of the problem. First, the cost of the deadheads in this example seems to be much smaller then the deadhead cost in the previous one. Also the span between the cheapest and the most expensive conversion. The total number of the candidates in candidate list for every deadhead is 397 in previous example and only 213 in this one, which is quite big difference if we recall exponential time complexity for minimizing function (see 5.3). Note that when we run the last example we need only 4 backtracks to find an optimal solution in the last example and 45 just find a first feasible solution in the first one.

The greedy heuristics described in section 5.3 also contributes to speeding up the execution of the program. The same optimization of the second example run without this heuristic demands longer time to find an optimal solution. Although a solution is found within reasonable time.

Without ordering candidates the algorithm performs 37 backtracks and makes 98213 resumptions comparatively to 4 backtracks and 19006 resumption when greedy heuristics is used.

Chapter 6

Conclusions and future work

The railway scheduling and routing is a very complex problem. The mathematical model, especially the new representation of the locomotive assignment where the cost for the deadhead is the part of the cost for a turn, is a one of the contributions presented in this work.

Although it is of little practical interest to find globally optimal deadhead to passive transport conversion solution to given conversion experiments and test show that there is a lot to gain running conversion algorithm longer then to first feasible solution. Right now the gain of such optimization is not comparable with regarded computation time. To improve it its necessary to find better conversion heuristics which limits conversion search space and speed up computation. One of such heuristics can be defined in strategy of choosing most profitable domain variable for labeling.

Another interesting problem for future works is to find a new representation of the problem which would allow to search for possible deadhead conversions while optimizing tours by exchanging links representing tours in circuits. It is possible that such approach would open new possibility for relinking circuits which would diminish overall cost of solution.

The four step optimization algorithm presented in chapter 4 is only an outline of a method which demands a lot of improvement to function properly. E.g. it is necessary to define strategies and approaches which could be used in optimization steps. It is open question if there is for example the method which would guarantee that all deadheads inserted during merging circuits will be removed during further step of optimization or which guarantee that overall cost for given set of circuits after performing algorithm will be equal or lower then original cost then such method should be defined.

Notice also that four step algorithm is heavily bounded to *k–interchange* method. It is proved that this method has exponential execution time complexity and is very expensive when it comes to computational costs. *K–interchange* is not scalable method either. Execution time grows with number of specified trips. It would be interesting to look at some other algorithms known from traveling salesman problem domain and see if they can be extended on locomotive assignment problem with time windows. From this domain Zhang algorithm referred in [Cir], even if its not a local optimization method, seems specially interesting. Even other methods like e.g. genetic algorithm and SAT should be investigated.

Despite this, we consider the contributions made in this work in the field of applying *k–interchange* to reduce the cost for the locomotive assignment as quite valuable. Especially the method of disjointing graphs using *k–interchange* seems to be important for the problem and even in the other assignments domains.

Bibliography

[Aro01] Aronsson, M., Kreuger, P., Lindblom, S. Cyclic time scheduling constraints. Unpublished, February 2001.

[Bar] Bartak, R. On–line guide to constraint programming. http://kti.ms.mff.cuni.cz/~bartak/constraints.

[Bus97] Bussieck, M.R., Winter, T, Zimmerman, U.T. Discrete optimization in public rail transport. *Mathematical Programming*, 79:415–444, 1997.

[Cas97] Caseau, Y., Laburthe, F. Solving small tsps with constraints. *Proc. of the 14th Int. Conf. on Logic Programming*, 1997.

[Chr76] Chrsitofides, N. Worst–case analysis of a new heuristic for the travelling salesman problem. Technical report, Graduate School of Industrial Administration,Cornegie Mellon University, 1976.

[Cir] Cirasella, J., Johnson, D.S., McGeoch, A., Zhang, W. The asymmetric traveling salesman problem: Algorithms, instance generators, and tests.

[Cro58] Croes, A. A method for solving traveling salesman problems. *Operational Research*, 5, 1958.

[Des92] Desrochers, M., Desrosiers, J., Salomon, M. A new optimization algorithm for the vehicle routing problem with time windows. *Oper. Res.*, 40:342 – 354, 1992.

[Dro97] Drott,J.,Hasselber,E.,Kohl,N.,Kremer,M. A planning system for locomotive scheduling. Technical report, Carmen Systems AB, 1997.

[Glo96] Glover, F. Finding a best traveling salesman 4–opt move in the same time as a best 2–opt. *J. Heuristics*, 2, 1996.

[Hal00] Halsgaun, K. An effective implementation of the lin–kerninghan traveling salesman problem. *Eur.J. of Oper.Res.*, 126:106–130, 2000.

[Har80] Haralick, R.M., Elliott, G.L. Increasing tree search efficency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[Kan80] Kanellakis, P.C., Papadimitriou, C.H. Local search for aymmetric traveling salesman problem. *Oper.Res.*, 28, 1980.

[Kil00] Kilby, P., Prosser, P., Show, P. A comparison of traditional and constraint–based heuristic methods on vehicle routing problems with side constraints. *Constraints*, 5(4):389–414, 2000.

[Kin85] Lenstra J.K. Savelsbergh M. Kindervater, G. Sequential and parallel local serach for the time-constrained traveling salesman problem. Technical report, Erasmus University, Department of Computer Science, 1985.

[Kre] P. Kreuger. Total order of repetitive tasks with bounded total circuit time. Unpublished.

[Kre01] Kreuger, P., Carlsson, M., Sjöland, T., Åström, E. Sequence dependent task extensions for trip scheduling. Technical report, Swedish Institute of Computer Science, 2001.

[Lin65] Lin, S. Computer solutions to traveling salesman problem. *Bell System Tech Journal*, 44, 1965.

[Lin73] Lin,S. ,Kerningham,B.W. An effective heuristic algorithm for traveling salesman problem. *Operational Research*, 21, 1973.

[Low85] Lowler, E.L, Lenstra, J.K., Rinnooy Kan, A.H.G, Shmoys, D. *The Traveling Salesman Problem.* Wiley, 1985.

[Mar92] Martin, O., Otto, S.W., Felten, E.W. Large–step markov chains for the tsp incorporating local search heuristics. *Operation Res. Lett.*, 11:219–224, 1992.

[Or,76] Or, I. *Traveling Salesman–type Combinatorial problems and Their Relation to the Logistic of Blood Banking.* PhD thesis, Dept. of Industrial Engineering and Management Sciences, Northwestern University, 1976.

[Pap78] Papadimitriou, C.H., Steiglitz, K. Some examples of difficult traveling salesman problems. *Oper.Res.*, 26, 1978.

[Pes98] Pesant, G., Gendreau, M., Potvin, J.–Y., Rousseau, J.–M. An exact constraint programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32:170–186, 1998.

[Ros77] Rosenkrantz, D.J., Stearns, R.E., Lewis II, P.M. An analysis of several heuristics of traveling salesman problem. *SIAM Journal of Computing*, 6:563 – 581, 1977.

[Sal83] Salomon, M. *Vehicle routing and scheduling with time window constraints: Models and algorithms.* PhD thesis, University of Pennsylvania, 1983.

[Sal86] Salomon, M. On the worst-case performace of some heuristics for the vehicle routing and scheduling problem with time window constraints. *Networks*, 16:161–174, 1986.

[Sal87] Salomon, M. Algorithms for vehicle routing and scheduling problem with time window constraints. *Op. Res.*, 35:254–265, 1987.

[Sav85] Savelsbergh, M.W.P. Local search in routing problems with time windows. *Annuals of Operations Research*, 4(6):265–305, 1985.

[Sch00] Scholtz, V. Knowledge-based locomotive planning for the swedish railway. Technical Report 05, Swedish Institute of Computer Science, 2000.

[SIC00] Sicstus prolog 3.8.4 manual, May 2000.

[Sig00] Sigurd, M., Pisinger, D., Sig, M. The pickup and delivery problem with time windows and precedences. Technical report, University of Copenhagen, 2000.

[Sim95] Simonis, H. The use of exclusion constraints to handle location continuity conditions. Technical report, Cosytec SA, 1995.

[Sim96] Simonetti, N., Balas, E. Implementation of linear time algorithm for certain generalized traveling salesman problems. In *Integer Programming and Combinatorial Optimization: Proc. 5th Int. IPCO Conference*, pages 316–329. Springer–Verlag, 1996.

[Stu98] *Programming with Constraints: An Introduction.* The MIT press, 1998.

[Tsa93] Tsang, E. *Foundation of Constraint Satisfaction.* Academic Press, 1993.

[Zha93] Zhang, W. Truncated branch–and–bound: A case study on the asymmetric tsp. In *Proc. of AAAI 1993 Spring Symposium on AI and NP–Hard Problems*, pages 160–166, 1993.

Appendix A

Interchange on set of circuits

Given a track graph in figure A.1 and trip specification of table A.1 there was generated a locomotive assignement for 4 locomotives.

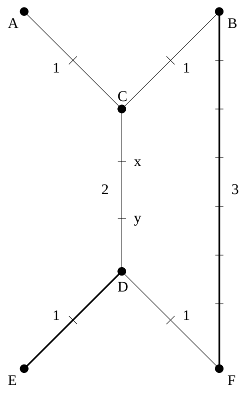

Figure A.1: A railway network

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 A.1: Trip specification

$$\begin{tikzcd}
\textit{L1:} & p1 \arrow[r, bend left] & p3 \arrow[l, bend left, "4"] & \textit{L2:} & p2 \arrow[r, bend left, "2"] & p5 \arrow[l, bend left] \\
\textit{L3:} & p4 \arrow[r, bend left] & p6 \arrow[l, bend left, "1"] & \textit{L4:} & p7 \arrow[loop left, "4"] &
\end{tikzcd}$$

Figure A.2: Initial solution. Cost of the turn is mesured in time of necessary deadhead.

$$\begin{tikzcd}
\textit{L1:} & p1 \arrow[r] & p3 \arrow[d] & \textit{L2} & p2 \arrow[loop left, "4"] \\
 & & p5 \arrow[ul] & &
\end{tikzcd}$$

Figure A.3: The first intechange between *L1* and *L2*,$c_{old} = 6 > c_{new} = 4$

$$\begin{tikzcd}
\textit{L1:} & p1 \arrow[r] & p3 \arrow[d] & \textit{L2:} & p2 \arrow[loop left, "4"] \\
 & p7 \arrow[u, "4"] & p5 \arrow[l, "2"] & & \\
\textit{L3:} & p4 \arrow[r, bend left] & p6 \arrow[l, bend left, "1"] & &
\end{tikzcd}$$

Figure A.4: All locomotive circuits after inserting trips $p7$ between $p5$ and $p1$ in ciruit for *L1*. Although cost for new links alone is higher then cost for old links, this interchange diminishes number of necessary locomotives, which diminishes overall cost for whole set of circuits

$$\begin{tikzcd}
p1 \arrow[r] & p3 \arrow[d] \\
p7 \arrow[u, dotted, "4"] \arrow[d, "3"] & p5 \arrow[l, dotted, "2"'] \arrow[ul, "0"] \\
p4 \arrow[r, bend left] & p6 \arrow[l, dotted, "1"] \arrow[ul, "1"]
\end{tikzcd}$$

Figure A.5: Possible interchange between *L3* and *L1*. Dotted lines indicates old links.

$$\begin{tikzcd}
\textit{L1:} & p1 \arrow[r] & p3 \arrow[d] & \textit{L2:} & p2 \arrow[loop left, "4"] \\
& & p5 \arrow[ul] & & \\
\textit{L3:} & p4 \arrow[r] & p6 \arrow[dl, "1"] & & \\
& p7 \arrow[u, "3"] & & &
\end{tikzcd}$$

Figure A.6: Final solution