Multi-Depot Vehicle Scheduling Problem with Coupling Trips and Time Windows

Bachelor Thesis

written by

YUAN, Zhi

Matr. Nr.: 1174618

Department of Mathematics

Darmstadt University of Technology

Siemens AG – Munich

Table of Contents

1 Introduction

This work presented here is based on my internship during the summer 2005 by Siemens AG in Munich. Our task was to develop a software to help our customer, a middle-large size logistics company, to schedule their locomotives to their weekly timetabled freight trains. The goal of the software is to minimize the operational cost of the locomotive deployments, i.e. to use a minimal number of locomotives to cover all the planned trains, and secondarily minimize the total driving time. There are many constraints to consider, among which there are different types of locomotives with different pulling power and speed. Each train will be pulled out only when sufficient pulling power is ensured. In our models we have taken into account an interesting variant that every train can be pulled by one or two locomotives of the same type, which is called a "coupling trip feature".

We have formulated this locomotive scheduling problem into a multicommodity flow network, and then model it as an Integer Linear Programming problem, and solve it using commercial ILP solver ILOG OPL Studio. Various models have been considered. One interesting variant is, unlike normal Multi-Depot Vehicle Routing Problem (MDVRP) such that each vehicle must return to its own depot where they started, depots with vehicles of the same type can exchange vehicles as long as the number of vehicles in each depot remain unchanged. It looks at the first glance more complicated, but the computational effort for solving the ILP of this model is far easier than applying the strict depot returning rule. Our experiment shows that the strict depot returning version requires about 10 times more computation time than the exchangeable depot version, to solve the same problem instance to optimality. The reason is that in the exchangeable depot model, each vehicle type will be viewed as a distinct commodity in the multicommodity flow network, while in the model applying strict depot returning rule, each depot must be considered as a commodity. Normally the ratio of the number of vehicle types and the number of depots is about 1:5, the exchangeable version greatly reduce the size of the multicommodity network. Another interesting variant is to bring in a flexible time window to each train instead of a fixed starting time. As we will see, the introduction of time windows brings significant computational difficulty for the ILP solver. A normal instance with 50000 variables and 20000 constraints needs only usually a few minutes to be solved to optimality; considering introduction of reasonable time windows, after 24 hours' computation, the ILP solver still reports a gap of about 15%.

To tackle this computational difficulty the ILP solver is confronted, we have tried another approach – heuristic algorithms. We first implemented a greedy construction heuristic tailored for this Multi-Depot Vehicle Scheduling Problem with Time Window and Coupling Trips (MDVSPTWCT). But it does not perform very well in practice, because in a highly constrained problem like this one, it is sometimes hard to distinguish an immediate best solution component during the construction process, a

simple evaluation function may become unreliable and even misleading. Therefore we have proposed two simple but robust and powerful construction strategies: PGrid and RGS. PGrid introduces several heuristic factors into a linear evaluation function, relevant or irrelevant, parameterizes it with different weight on different heuristic information, and finds the best parameter combination by grid search. RGS (Randomized Greedy Search) brings a weighted perturbation into each construction step so that solution components within a certain quality range will be given a probability to be chosen, the better the component quality measured by a certain evaluation function is, the higher its selection probability is. We can even hybridize PGrid and RGS into one process, by applying PGrid first to determine an amount of elite parameter combinations, followed subsequently by starting RGS to explore the search space measured by these elite evaluation functions extensively. Experiment shows this hybrid of PGrid and RGS totally outperforms the classical metaheuristic GRASP (Greedy Randomized Adaptive Search Procedure).

In solving the MDVSPTWCT problem, our heuristic method is competitive with the exact method under the support of the commercial ILP solver OPL Studio. In the fixed timetable model, with the help of OPL Studio, we are able to solve the given problem instances to optimality in several minutes in the exchangeable depot version, and a pair of hours in the strict depot returning version. Our heuristic algorithm solves the problem within a few minutes with a less than 5% gap to optimality. In the flexible time window case, OPL Studio becomes almost helpless. For a normal-size instance, the first feasible solution emerged only after 3 hours with very poor quality. After keeping the OPL running for 24 hours, there is still about 15% gap. A fine-tuned hybrid of PGrid and RGS usually solves the problem within 5 minutes with a solution at least as good as the one returned by OPL Studio after running 24 hours.

Our experiment results show a great potential in saving operational cost comparing the current manual schedule. In the fixed timetable model, the optimal schedule can save over 20% operational cost comparing with the current hand-made schedule by our customer. If a reasonable time window (such as six hours' flexibility during a week's schedule) is allowed, the company can expect more than 40% cost saving in comparison with the current manual schedule. We can also see the power of the introduction of time windows, which brings at least 30% cost reduction comparing the fixed timetable scheduling, and convinces us that it is worth the great computation effort.

The remainder of the article is organized as follows. In Chapter 2, we give an introduction to some mathematical background. Chapter 3 provides a detailed description of our project, and how we model it into a multicommodity flow network. Chapter 4 describes the mathematical ILP model to solve the MDVSPTWCT, and some preprocessing techniques will be discussed in detail. Chapter 5 gives a gentle introduction to our heuristic algorithms, and presents how our heuristic algorithms can be applied to our project MDVSPTWCT. We discuss in Chapter 6 how our test

instances are built up, and look into some technical details of each instance. All our test results will be listed in Chapter 7.

2 Mathematical Background

This chapter presents some mathematical prerequisites that might be needed for understanding the contents of the work. For terminology we stick mainly to Ahuja et. al. (1993), Schrijver (1986), Nemhauser and Wolsey (1999). The rest of the chapter is organized as follows, Section 2.1 gives a brief introduction to the mathematical model of integer linear programming; Section 2.2 gives a quick guide to some basic models of network flows.

2.1 Integer Linear Programming

Modeling and solving integer linear programming problem lies at the heart of discrete optimization. Various problems in science, engineering, economics and society can be modeled into integer linear programming (ILP) problems.

Given an $m \times n$ matrix A, a m-vector b, and a n-vector c. An ILP is a system of the following form:

$$\begin{aligned} \text{minimize} \quad & c^T x \\ \text{subject to} \quad & Ax = b \\ & \underline{x} \le x \le \overline{x} \\ & x \in \mathbb{Z}^n \end{aligned} \tag{2.1}$$

Solving ILP is in general NP-hard. The exact algorithm for solving this problem in general follows the branch-and-bound paradigm, see Schrijver (1986) or Nemhauser and Wolsey (1999) for more details.

2.2 Network Flows

2.2.1 Directed Graphs and Networks

A *directed graph* (or *digraph*) $G = (N, A)$ consists of a finite nonempty node set N and a finite arc set A whose elements are ordered pairs of distinct nodes. A *directed network* is a directed graph whose nodes and/or arcs have associated numerical values (typically costs with vector c, capacities with vector u, and/or supplies and demands with vector b).

2.2.2 Minimum Cost Flows and Single Commodity Flows

The minimum cost flow model is the most fundamental of all network flow problems. We first present an ILP formulation of the minimum cost flow problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{(i,j)\in A} c_{ij} x_{ij} \\ \text{subject to} \quad & \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b_i \quad \forall i \in N \\ & 0 \le x_{ij} \le u_{ij} \quad \forall (i,j) \in A \\ & x \in \mathbb{Z}^{|A|} \end{aligned} \tag{2.2}$$

Note that the well-known shortest path problem and maximum flow problem are both special cases of minimum cost flow problem. For shortest path problem, one can simply set the source node to have of one unit supply and the sink node one unit demand, with all other nodes as transshipment nodes; for maximum flow problem, one can set the supply/demand on all nodes to be 0 and the costs on all arcs to be 0, and add a virtual arc from sink to source with cost -1.

Minimum cost flow problem can be solved in polynomial time using e.g. successive shortest path algorithm. But in practice, a pseudo polynomial-time network simplex algorithm solves the problem quite efficiently, see Ahuja et. al. (1993), or Löbel (1997) for more details.

2.2.3 Multicommodity Minimum Cost Flows

In many application contexts, including our MDVSP project, there are more than one commodities sharing the same network, and especially sharing common facilities. In this case the network will have several copies, and each layer belongs to a commodity. A bundle constraint will tie the different commodities together.

Let x_{ij}^k denote the flow of commodity k on arc (i, j), and let x^k and c^k denote the flow vector and per unit cost vector for commodity k. Using this notation we can formulate the multicommodity flow problem as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{k\in K} c^k x^k \\ \text{subject to} \quad & \sum_{\{j:(i,j)\in A\}} x_{ij}^k - \sum_{\{j:(j,i)\in A\}} x_{ji}^k = b_i^k \quad \forall i \in N, k \in K \\ & 0 \le x_{ij}^k \le u_{ij}^k \quad \forall (i,j) \in A, k \in K \\ & \sum_{k\in K} x_{ij}^k \le u_{ij} \quad \forall (i,j) \in A \\ & x \in \mathbb{Z}^{|A|\times|K|} \end{aligned} \tag{2.3}$$

Solving multicommodity flow problem is in general NP-hard, some state-of-the-art computation efforts have been presented in Löbel (1997).

3 Project Description

3.1 General Information and Some Terminologies

The aim of our project is to help our potential customer, one of the middle-large size logistics companies in Europe, develop a software to assist them in assigning locomotives to their timetabled shipments. Before we start, we will first have a close look over some details of the problem setting:

- The shipments (or trains): what to be transported are mainly industrial bulk goods. And the companies who require or supply them usually order a whole freight train, consisting of several wagons of different total weight. Every week the logistic company will come up with a complete list of trains for the coming week, it is our task to assign a fleet of locomotives to pull all these trains. For each train, a fixed start and a specific destination station is given, and the weight of each train is also known in advance. Note that the weight of the train is important since that decides which locomotive combinations will be possible for it.

- Timetable of the trains: also given is a timetable, with the arrival deadlines of all trains on it. Some of the shipments are strict about punctuality, i.e. they have to arrive exactly at the time given by the timetable; while the other shipments are more flexible, their arrival time can vary within a range of time, usually several hours, around the given deadline. The duration of each shipment is also contained in the timetable, depending on different types of locomotives, the duration may also be different.

- Locomotives: the logistic company owns about 50 locomotives of several manufacturers. However, the differences between them are minor, so they are grouped into 2 classes of similar locomotives. The main differences among the classes are the driving power of the engine (the service weight), and the speed. It is clear that a locomotive can be assigned to a train if it has sufficient driving power; if it does not, it is also possible to combine 2 locomotives of the same class to pull the train, if that suffices. It is called a “twin combination” for the latter case. The locomotives owned by the company are located in different garages of different cities. In order to keep the workload of each garage stable from week to week, it is required that, the number of locomotives of the same class should remain unchanged in each garage, after a week’s service plan. Note that the locomotives are deployed from their garages at the beginning of the week, after serving all the shipments, they do not have to return to their own garages, the garages with locomotives of the same class can exchange with each other. But sometimes in the busy weeks, the company may need to lease some locomotives,

in such case the leased locomotives will have to return to the exact depot where they start. In this article, both cases will be considered in detail.

- Objectives: the main objective is to reduce operating expenses, that is, to use as few locomotives as possible to pull all trains and, on a subordinate level, to schedule the locomotives in such a ways that the total trip length, including delivery trips and unloaded trips, is as short as possible.

A currently similar project of "Locomotive Scheduling" has been carried on in our research group, details can be found in Fügenschuh et. al. (2006).

3.2 Problem Model

First we will translate the above real world problem into mathematical languages. This problem can be described as a Multiple-Depot Vehicle Scheduling Problem (MDVSP). Our terminology follows Löbel (1997) to a large extent.

In the following we will transform the above real world problem into a network flow model.

In the network we are going to depict, we use N to denote the set of all nodes. We first classify the all the nodes into 2 types:

- The city nodes (or station nodes or garage nodes), denoted as N_{city}. Each of these nodes has a specific geographic location.
- The virtual nodes, which include a source node n_s and a sink node n_t.

Before we further distinguish the set of city nodes N_{city}, we first give several definitions. A garage (or station) is a location where vehicles are parked and serviced. The term fleet denotes the set of vehicles (e.g. locomotives in our project) of our transportation company. The fleet is distributed among different garages, and it is known which vehicle types and how many vehicles of each type are stationed at each garage. A depot is defined as an abstract garage that possesses a nonempty set of vehicles that need not be distinguished for the scheduling processes. We use D to denote the set of all depots. For each depot $d \in D$, we associate a start node d^+ and an end node d^- where its vehicles begin and terminate their weekly schedules. Let

$$N^+_{depot} := \{d^+ \mid d \in D\} \quad \text{and} \quad N^-_{depot} := \{d^- \mid d \in D\}$$

denote the set of all such start and end depot nodes, respectively. We can view a depot

as a set of homogeneous vehicles that are stationed at the same location. In order to derive a set of depots from a set of garages, we can simply partition the set of vehicles of each garage into several subsets of vehicles of the same type, and then each subset of homogeneous vehicles from the same garage depicts a depot.

A shipment (or a task or a train) is a planned trip that needs to be accomplished by the vehicles. In our project, it represents the loaded trains to be transported. We denote the set of all shipments by S. For each shipment $s \in S$, it has a specific origin and destination station, denoted by s^+ and s^-, respectively. Let

$$N^+_{shipment} := \{s^+ \mid s \in S\} \quad \text{and} \quad N^-_{shipment} := \{s^- \mid s \in S\}$$

denote the set of all origin and destination shipment nodes, respectively.

All in all, the following set relationships hold:

$$N = \{n_s\} \dot{\cup} \{n_t\} \dot{\cup} N_{city} \quad \text{and} \quad N_{city} = N^+_{depot} \dot{\cup} N^-_{depot} \dot{\cup} N^+_{shipment} \dot{\cup} N^-_{shipment}$$

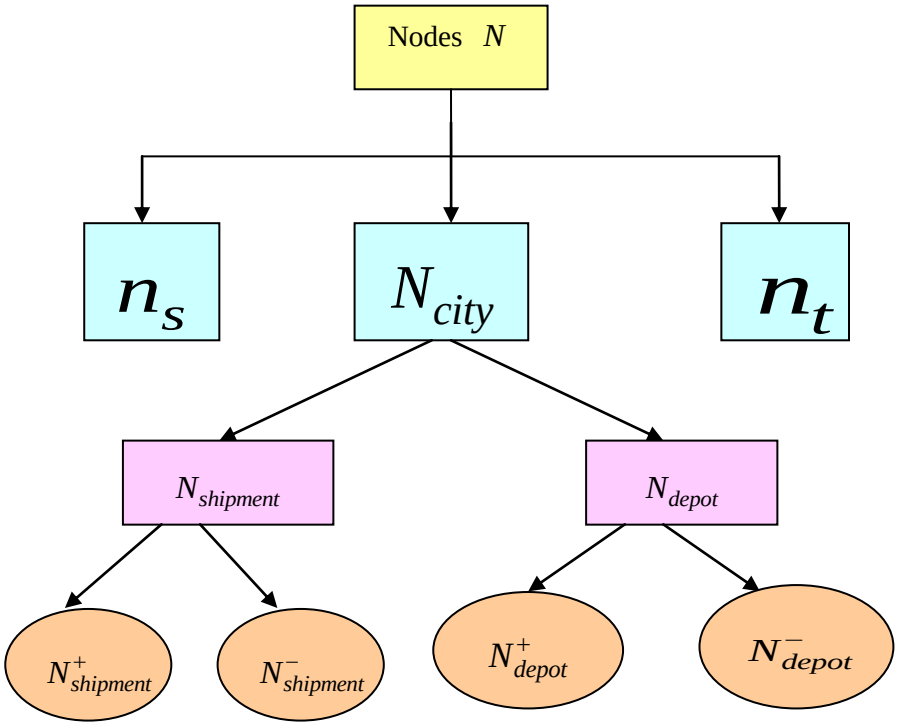


Figure 3.1: hierarchy of nodes

Now we start to introduce the arcs that connect all the nodes described above in the network. First we denote all the arcs in the network to be A, and all arcs in this network are directed.

Note that a multicommodity flow model is a multi-layered network, i.e. there is a copy (layer) of the network facilities (like arcs) for each commodity, so that it

guarantees each commodity flow can only run within its own layer, but flows of different commodities are still sharing the same network resources, like arc capacity etc. We denote *C* the set of all commodities.

In many cases every depot is considered as a commodity, so that we make sure the vehicle from one depot will return to its own depot after it services all its scheduled trips. But we will also consider another scenario, where each depot can exchange vehicles of the same type with other depots, as long as the number of vehicles remain unchanged before and after all trips are served. We will show in next section (?) that, in this case we can consider each vehicle type as a commodity. In practice the number of different vehicle types is far less than the number of depots, so this model is relatively easier to solve and will also result in less operational cost.

We denote T as the set of different vehicle types. In the following, if it is not specified, we refer to $C = T$, and $k \in C$ as the vehicle type *k*.

In the sequel we will list all types of arcs in our network:

- Delivery Arc, which starts from origin shipment node s^+ and ends at the destination shipment node s^-.

$$A_k^{delivery} := \{(s^+, s^-) \mid s \in S, k \in C\}$$

- Unloaded Arc A_k^{ul}. A vehicle is either active, i.e. serving a shipment on a delivery arc, or unloaded, i.e. driving by itself without serving any tasks. There are 3 types of unloaded arcs in our network:
  - Start Arc (or pull-out trip), which connects a starting depot node d^+ and a starting task node s^+.

$$A_k^{start} := \{(d^+, s^+) \mid s \in S, d \in D, k \in C\}$$

  - End Arc (or pull-in trip), which connects an end task node s^- and an end depot node. d^-.

$$A_k^{end} := \{(s^-, d^-) \mid s \in S, d \in D, k \in C\}$$

  - Deadhead Arc (or Inter-task arcs), which connects an end task node s^- and a start task node s^+.

$$A_k^{dh} := \{(s^-, s'^+) \mid s, s' \in S \text{ and } s \neq s', k \in C\}$$

- In-Service Arc, which connects a source node N_s and a start depot node d^+.

$$A_k^{is} := \{(N_s, d^+) \mid d \in D, k \in C\}$$

- Out-of-Service Arc, which connects a end depot node d^- and a sink node N_t.

$$A_k^{os} := \{(d^-, N_s) \mid d \in D, k \in C\}$$

To sum up,

$$A = A_k^{is} \dot{\bigcup} A_k^{os} \dot{\bigcup} A_k^{delivery} \dot{\bigcup} A_k^{ul} \quad \text{and} \quad A_k^{ul} = A_k^{start} \dot{\bigcup} A_k^{end} \dot{\bigcup} A_k^{it}$$

The hierarchy of arcs is shown in the figure below:

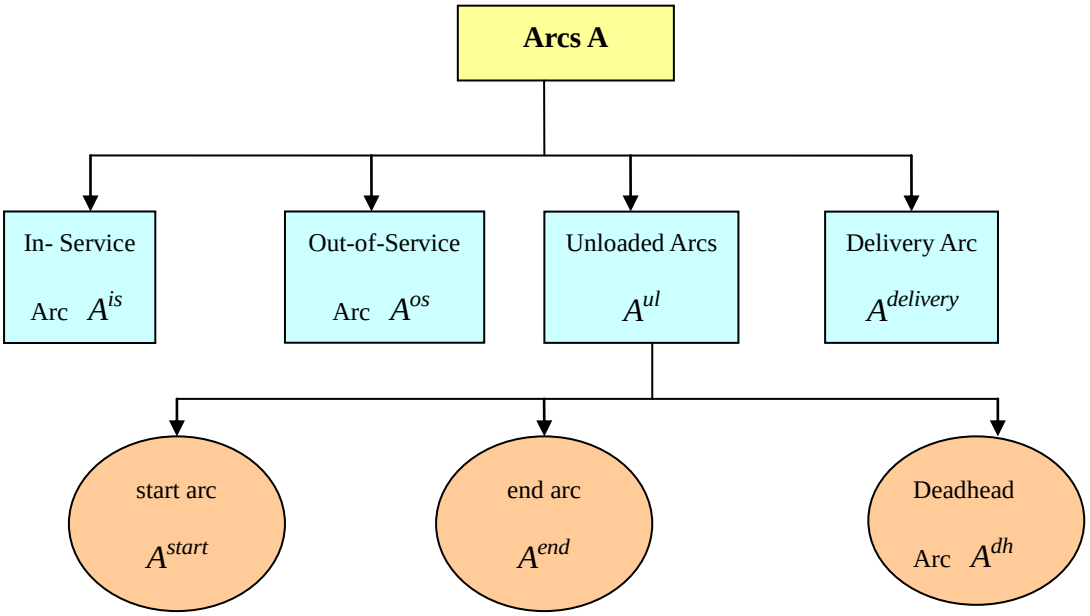


Figure 3.2: hierarchy of arcs

Having defined all the nodes and arcs of our network separately, now we will be happy to present a panorama of our multicommodity flow network for the MDVSP as follows:

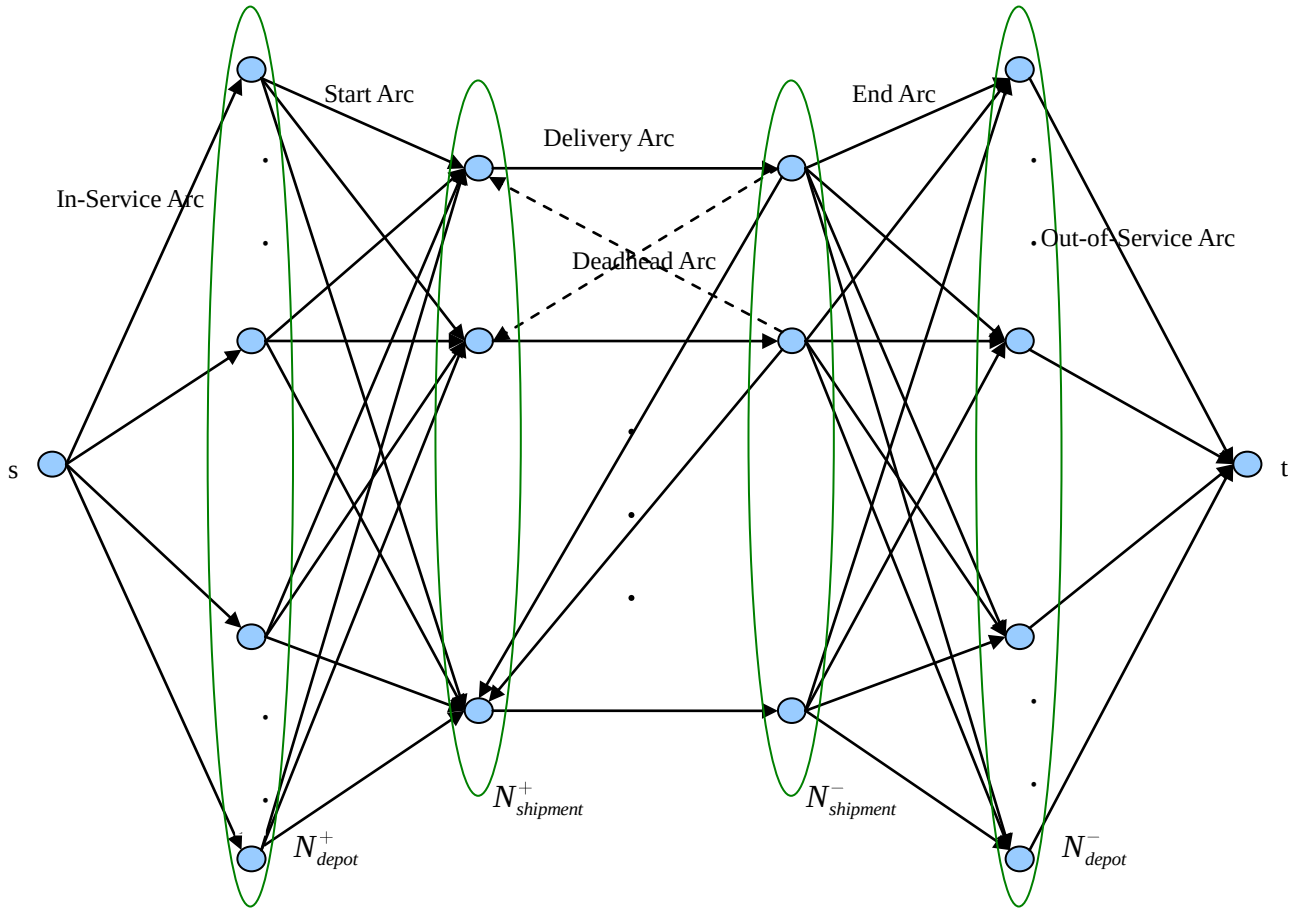


Figure 3.3: panorama of the network flow model for MDVSP

Note that not all deadhead trips are shown in the panorama above, since some of the infeasible deadhead trips are already excluded. Some of the deadhead arcs above are drawn with dashed lines, that is because for the version of MDVSP with time windows, these arcs' feasibility will depend on the choice of the starting time of the associated shipments.

3.3 Computational Complexity of MDVSPTWCT

Löbel (1997) has given a thorough and detailed proof that MDVSP is NP-hard and its associated decision problem is NP-complete. The MDVSPTWCT problem is NP-hard, since it contains MDVSP as a sub-problem.

4 Mathematical Models

We introduce a mathematical model that is based on Multicommodity flow formulation. The multicommodity network is built in the last section. And we will formulate the model as an integer linear programming problem.

In section 4.1 we introduce some parameters we use to describe the multicommodity network, which are useful for the construction of the ILP model. In the following three sections, ILP models for three different versions of the problem, sorted in increasing computational effort, will be explained in detail. Section 4.2 introduces the ILP model for MDVSP with coupling trips, fixed timetable and exchangeable depot; section 4.3 presents MDVSP with coupling trips, fixed timetable and strict depot; section 4.4 refers to MDVSP with coupling trips, time windows and exchangeable depot. The last section 4.5, we will focus on some topics in preprocessing.

4.1 Parameters of the multicommodity network model

The following parameters are important for the network model presented in Figure 3.3:

- c_i^k --- the cost attached to each arc $a_i^k \in A$
  - For In-Service Arc $a_i^k \in A_k^{is}$ which connects the source node and $d_{k'}^l \in D$, a depot of vehicle type k' and located in city l, c_i^k is 0 if $k \neq k'$, and is the cost of deploying a vehicle of type k if $k = k'$. Normally this cost is set to a very high value, since having as few as vehicles is our primary goal in this scheduling task.
  - For Out-of-Service Arc $a_i^k \in A_k^{os}$, the cost c_i^k is always 0.
  - For Delivery Arc or Unloaded Arc $a_i^k \in A_k^{delivery} \bigcup A_k^{ul}$, the cost is set to be the driving time (or distance) of transporting the shipment multiplied by a unit cost.

- u_i^k --- the upper bound of each arc $a_i^k \in A$.
  - For In-Service Arc or Out-of-Service Arc $a_i^k \in A_k^{is} \bigcup A_k^{os}$ which connects the

source or sink node and a depot node $d^l_{k'} \in D$, a depot of vehicle type k' and located in city l, u^k_i is 0 if $k \neq k'$, since a depot can only have one type of vehicle. Otherwise when $k = k'$, u^k_i is the number of vehicles that this depot possesses, so that this depot can deploy no more vehicles than this upper bound.

- For Delivery Arc or Unloaded Arc $a^k_i \in A^{delivery}_k \bigcup A^{ul}_k$, the upper bound u^k_i is

$$u^k_i = \begin{cases} 1 \text{ if arc } a^k_i \text{ is admissible} \\ 0 \text{ elsewise} \end{cases}$$

- we will define an arc a^k_i to be admissible in 2 cases: 1) if arc a^k_i is a delivery arc, then it should be capable to be carried out by vehicle of type k (either one locomotive alone or using a "twin combination"); 2) otherwise if arc a^k_i is an unloaded arc, then any delivery arc that is adjacent to this arc a^k_i should be capable to be carried out by vehicle of type k.

Note that we do not introduce any parameter as lower bound for the arcs, since the lower bounds are by default set to 0 for all arcs. Hence by setting the upper bound u^k_i of some arc a^k_i to be 0, we have simply removed this inadmissible arc from our network. We can further simplify the model by removing all inadmissible arcs A_{inadm} from A by setting:

$$A = A \setminus A_{inadm}$$

- $M_{N_{city} \times A}$ --- the node-arc incidence matrix for all city nodes $n \in N_{city}$. It is for formulating flow conservation constraint (see next section). Let $n = |N_{city}|$ be the number of elements in the set N_{city}, and let $m = |A|$. $M_{N_{city} \times A}$ is an $n \times m$ matrix. Each column of the matrix corresponds to an arc, e.g. (i, j), then this column has a $+1$ in the ith row, a -1 in the jth row. Note that the flow conservation constraint only holds for city nodes N_{city}, and does not hold for the source node n_s and sink node n_t in our network, so we exclude the source and

sink node from our incidence matrix.

1. f_i^k --- the magnitude of flow that should run on delivery arc or unloaded arc $a_i^k \in A_k^{delivery} \bigcup A_k^{ul}$.

   - For delivery arc $a_i^k \in A_k^{delivery}$, this parameter f_i^k describes the number of vehicles of type k that is required to carry out this shipment i.
   - For unloaded arc $a_i^k \in A_k^{ul}$, this parameter f_i^k describes the number of vehicles of type k that is required to carry out any one of the adjacent shipment(s).

   With the help of this parameter f_i^k, we can decide whether a specific shipment needs a "twin combination", i.e. it needs 2 locomotives to pull the train. In our project, we do not accept f_i^k greater than 2, in that case the upper bound parameter u_i^k will be set to 0. But in fact our model can be generalized to accept f_i^k greater than 2, i.e. several vehicles serving the same shipment simultaneously.

4.2 MDVSP with fixed timetable and exchangeable depot

To begin with, we take the starting and arrival times for the trains as they were given from the timetable that the logistic company has precomputed. Exchangeable depot here means depots, which are located at different garages but have the same type of vehicles, can exchange vehicles with each other when a week's plan is finished, as long as each of them still has the same number of vehicles as before.

We first introduce the decision variables of our model.

- x_i^k --- represents the flow on each arc $a_i^k \in A$. It is a nonnegative integer. Note that here $k \in T$ stands for a commodity, and it decides which layer of the network this arc lies in. In the exchangeable depot model, we refer to $C = T$, the set of different vehicle types.
- e_i^k --- is a binary variable on each delivery arc or unloaded arc $a_i^k \in A_k^{delivery} \bigcup A_k^{ul}$.

It equals 1 if and only if the arc a_i^k is loaded with some flow, equals 0 when no flow passes the arc a_i^k. In other words,

$$e_i^k = \begin{cases} 0 & \text{if } x_i^k > 0 \\ 1 & \text{if } x_i^k = 0 \end{cases}$$

Note that the e_i^k only decides whether there is flow on an arc, the magnitude of the flow is decided jointly with flow magnitude parameter f_i^k.

Now we start to formulate the problem.

1. the bundle constraint (is also called "flow conditions").

$$\sum_{k \in C} e_i^k = 1, \quad \forall i \in A_{delivery}, \tag{4.1}$$

$$\sum_{k \in C} e_i^k \leq 1, \quad \forall i \in A_{unloaded} \tag{4.2}$$

The equality (4.1) says, in this multicommodity layered network, each delivery arc should have flows on one and only one layer, in other words, each shipment should be serviced by one and only one vehicle type. For unloaded arcs in the network, the inequality (4.2) ensures that no more than one commodity can service each unloaded arc.

2. Flow upper bound constraints

$$e_i^k \leq u_i^k, \quad \forall i \in A_{delivery} \bigcup A_{unloaded} \text{ and } \forall k \in T \tag{4.3}$$

Note that the flow upper bound parameter u_i^k for delivery arcs and unloaded arcs $a_i \in A_{delivery} \bigcup A_{unloaded}$ is binary-valued, deciding whether an arc a_i is admissible under commodity *k*. Inadmissible arcs will be directly eliminated by this constraint.

3. flow magnitude constraints

$$x_i^k = e_i^k f_i^k, \quad \forall i \in A_{delivery} \text{ and } \forall k \in T \tag{4.4}$$

$$x_i^k \leq e_i^k f_i^k, \quad \forall i \in A_{unloaded} \text{ and } \forall k \in T \tag{4.5}$$

Recall that in our problem each shipment can be transported by more than one vehicles of the same type, the flow magnitude parameter f_i^k is precomputed to indicate how many vehicles of type *k* is required to service shipment s_i.

Constraint (4.4) ensures sufficient flows on each delivery arc, and constraint (4.5) restricts flows on each unloaded arc to a certain magnitude.

4. Handling In-Service Arcs and Out-of-Service Arcs (IO Arcs)

$$x_i^k \le u_i^k, \quad \forall i \in A_{is} \bigcup A_{os} \text{ and } \forall k \in T \tag{4.6}$$

$$x^k_{(n_s, d_i^+)} = x^k_{(d_i^-, n_s)}, \quad \forall d_i \in D \text{ and } \forall k \in T \tag{4.7}$$

The flow upper bound parameter u_i^k in IO Arcs indicates how many vehicles are available in the associated depot. So inequality (4.6) ensures no more than this number of vehicles may be deployed. Recall that each depot should have the same number of vehicles before and after a week's services. And the constraint (4.7) guarantees that the number of vehicles that left the depot equals the number of vehicles that come back after their services.

5. Flow conservation constraint

$$M_{N_{city} \times A} \bullet x^k = 0 \quad \forall k \in T \tag{4.8}$$

We formulate our flow conservation constraint with the help of the node-arc incidence matrix $M_{N_{city} \times A}$, so that the total inflow of each city node equals its total outflow. In literature, such nodes that neither hold nor generate any flows are also referred to as transshipment nodes.

6. The objective function is to minimize

$$\sum_{i \in A, k \in C} c_i^k \bullet x_i^k \tag{4.9}$$

Our goal is to reduce the operational cost. For this reason, to use as few as possible vehicles is of the first priority, hence we have addressed a relative high cost on each In-Service Arc $a_i \in A_{is}$, where a vehicle is deployed. The cost of the delivery arcs and unloaded arcs depends on the length or driving time of the arc, and the vehicle type.

To sum up, we have the following integer optimization problem:

$$\begin{aligned} &\min \quad (4.9) \\ &\text{subject to} \quad (4.1, \ldots, 4.8) \\ &x \in \mathbb{Z}^{|A| \times |V|}, \; e \in \{0,1\}^{|A| \times |V|} \end{aligned} \tag{4.10}$$

4.3 MDVSP with fixed timetable and strict depot

The second model we are presenting is very similar to the one introduced in the last section, except that every vehicle must return to its own depot after all its services, i.e. no vehicles can be exchanged among depots. It seems at the first glance easier to solve, since it is just a special case of the model with exchangeable depot, but as we will see, the new restriction makes the size of the network expand, causing computational difficulty.

Let us recall that a depot is uniquely determined by its location and vehicle type. Let $d^{k,l} \in D$ denote a depot that possesses vehicle type $k \in T$ and is located at garage $l \in G_k$, where G_k denotes the set of garages that possess vehicles of type *k*. Recall that the model of the exchangeable depot version takes every vehicle type as a commodity, i.e. every vehicle type has one layer of the network, and vehicles of this type only run on this layer. In the model of strict depot, in order to ensure each deployed vehicle to return to its own depot, we provide each depot one layer of the network, and vehicles of one depot can only run on its own layer. So instead of $|T|$ commodities in the exchangeable depot model, the number of commodities in the strict depot version is in total

$$\sum_{k \in V} k \cdot |G_k|$$

Our network has expanded to one more dimension.

Correspondingly, our decision variables will also grow one more dimension to control the network:

- $x_i^{k,l}$ --- The flow on arc $a_i \in A$, in layer $k \in T, l \in G_k$.
- e_i^k --- The binary variables deciding whether flow exists on arc $a_i \in A$, for a vehicle type $k \in T$.

And all the parameters *c, u, f,* stay at their original dimension $|A| \times |T|$, because the cost, flow upper bound and flow magnitude only depends on vehicle type and arc-specific characteristics, and different depots have no influence on them.

The Integer Linear Programming formulation is as follows:

1. For bundle constraints we take exactly the previous formulation (4.1), (4.2), meaning flows of one and only one vehicle type are allowed to run on each

delivery arc and unloaded arc. But note that vehicles of the same type but coming from different depots are also allowed to pull a train at the same time.

2. For flow upper bound constraint we simply take the formulation (4.3).

3. The flow magnitude constraints look a little different, we reformulate it as follows:

$$\sum_{l \in G_k} x_i^{k,l} = e_i^k f_i^k, \quad \forall i \in A_{delivery} \text{ and } \forall k \in T \tag{4.11}$$

$$\sum_{l \in G_k} x_i^{k,l} \leq e_i^k f_i^k, \quad \forall i \in A_{unloaded} \text{ and } \forall k \in T \tag{4.12}$$

Sometimes more than one vehicles of the same type are required to serve a shipment, but these vehicles do not need to be from the same depot.

4. In handling IO arcs, we apply the following instead:

$$x_i^{k,l} \leq u_i^{k,l}, \quad \forall i \in A_{is}, k \in T, l \in G_k, \text{ and } i = (n_s, d^{k,l}), d^{k,l} \in D \tag{4.13}$$

$$x_i^{k,l} = 0, \quad \forall i \in A_{is}, k \in T, l \in G_k, \text{ and } i \neq (n_s, d^{k,l}), d^{k,l} \in D \tag{4.14}$$

$$x_{(n_s, d_i^+)}^{k,l} = x_{(d_i^-, n_s)}^{k,l}, \quad \forall d_i \in D \text{ and } \forall k \in T, \ \forall l \in G_k \tag{4.15}$$

So the each depot only gets flows from the source in the right layer, and (4.15) ensures vehicles that start from each depot end in their own depot.

5. The flow conservation constraint stays more or less the same, to ensure flow conservation holds for each layer:

$$M_{N_{city} \times A} \bullet x^{k,l} = 0 \quad \forall k \in T \text{ and } \forall l \in G_k \tag{4.16}$$

6. The objective function:

$$\sum_{i \in A, k \in C} (c_i^k \bullet \sum_{l \in G_k} x_i^{k,l}) \tag{4.17}$$

Again to sum up, we have the following ILP formulation for strict depot problem:

$$\begin{aligned} &\min \quad (4.17) \\ &\text{subject to} \quad ((4.1), (4.2), (4.3)) \\ &\quad (4.11, \ldots, 4.16) \\ &x \in \mathbb{Z}^{|A| \times |D|}, \ e \in \{0,1\}^{|A| \times |T|} \end{aligned} \tag{4.18}$$

4.4 MDVSP with flexible time window and exchangeable depot

We modify the above model for vehicle scheduling for the case where the starting times of the shipments are allowed to be changed within a given interval. The corresponding model is called multi-depot vehicle scheduling problem with time windows (MDVSPTW). In comparison with the MDVSP, the MDVSPTW model is more complicated, for two reasons: firstly many previously inadmissible deadhead trips by the timetable, i.e. with the fixed starting time, have now become feasible due to their starting time flexibility; secondly, to ensure every deadhead trip to be admissible becomes a computationally hard task.

We first formulate the MDVPTW into ILP model. Here we will only consider the time window constraints over the exchangeable depot MDVSP model (4.10).

Additional decision variable:

- t_i --- the starting time for each shipment $i \in S$.

We first introduce bounds on the starting time of the shipments. Assume each shipment i with a related time interval $\left[\underline{t_i}, \overline{t_i}\right]$, in which the starting time must lie:

$$\underline{t_i} \le t_i \le \overline{t_i} \quad \forall i \in S \tag{4.19}$$

If shipment *i* and *j* are serviced consecutively by the same vehicle, i.e. the node s_i^- and s_j^+ are connected, then the corresponding starting times must satisfy:

$$t_j - \delta_{(i,j)}^k - t_i \ge M \bullet (e_{(i,j)}^k - 1), \quad \forall i, j \in S, \text{ and } \forall k \in T \tag{4.20}$$

Where M is a sufficiently large constant, (i, j) denotes the deadhead arc from shipment *i* to *j*, and $\delta_{(i,j)}^k$ denotes the total delivery trip and deadhead trip duration, that is,

$$\delta_{(i,j)}^k = \delta_{i,k}^{dlv} + \delta_{(i,j),k}^{dh} \tag{4.21}$$

Where

- $\delta_{i,k}^{dlv}$ --- denotes the delivery trip duration, i.e. the time the vehicle of type *k* need to service the shipment *i*.
- $\delta_{(i,j),k}^{dh}$ --- denotes the time for deadheading from the end of shipment *i* to the start

of the shipment *j*.

This constraint (4.20) ensures, if shipment *i* and *j* are connected, i.e. when $e^k_{(i,j)} = 1$, the left hand side of the inequality should greater than or equal to 0, making the deadheading feasible; or else when $e^k_{(i,j)} = 0$, the big M constant ensure there is no restriction between t_i and t_j, i.e. the inequality holds trivially. Note that experiments show that we should not give M an arbitrary large value, due to the fact that the larger the M is, the harder the computational effort will be, especially in achieving a tight lower bound relaxation. We suggest a reasonable value of M to be $12960 = 1440 \times 9$, where 1440 refers to the number of minutes per day. The our scheduling period is weekly based, i.e. the largest possible difference between the starting times of 2 shipment is at most 7 days, and we assume no shipment will take longer than one day and deadheading duration is bounded to one day, which sums up to 9 days in total.

To sum up, the ILP model of the MDVSPTW is as follows:

$$\begin{aligned} &\min \quad (4.9) \\ &\text{subject to} \quad (4.1, \ldots, 4.8) \\ &(4.19), (4.20) \\ &x \in \mathbb{Z}^{|A|\times|T|},\ e \in \{0,1\}^{|A|\times|T|} \\ &t \in \mathbb{Z}^{|S|} \end{aligned} \tag{4.22}$$

4.5 Preprocessing

In this section we discuss several techniques that we have applied in the preprocessing phase before we pass the mathematical model to ILOG OPL Studio. It includes reducing variables and constraints, and simplifying the OPL data file.

4.5.1 Reducing Big-M constraints

Before actually solving the model by branch-and-cut techniques, it is desirable to have a compact formulation with as few constraints and variables as possible. We are also aware that the linear programming based branch-and-cut approach usually suffers from constraints having big-M-terms, such as (4.20) in our model. In the following we seek possibilities to reduce redundant big-M constraints and decrease the value of M.

We observe that constraint (4.20) is attached with every deadhead trip, whereas many of them are redundant. Let $i, j \in S$ be shipments, and $a^k_{(i,j)}$ denotes the deadhead trip from shipment *i* to *j* with vehicle type *k*. We can partition the set of deadhead arcs A^k_{dh} into 3 cases, i.e.

$$A^k_{dh} = A^k_{always} \dot{\bigcup} A^k_{dilemma} \dot{\bigcup} A^k_{never}$$

Where

- $A^k_{always} := \{a^k_{(i,j)} \mid \bar{t}_i + \delta^k_{(i,j)} \leq \underline{t}_j\}$ -- denotes those deadhead arcs that are always feasible despite the choice of t_i and t_j.
- $A^k_{never} := \{a^k_{(i,j)} \mid \underline{t}_i + \delta^k_{(i,j)} > \bar{t}_j\}$ -- denotes those deadhead arcs that are always infeasible despite the choice of t_i and t_j.
- $A^k_{dilemma} := \{a^k_{(i,j)} \mid \bar{t}_i + \delta^k_{(i,j)} > \underline{t}_j \wedge \underline{t}_i + \delta^k_{(i,j)} \leq \bar{t}_j\}$ -- denotes those deadhead arcs whose feasibility depends on the choice of t_i and t_j.

A graphical interpretation of the three classes of deadhead trips is presented in Figure 4.1 as follows:

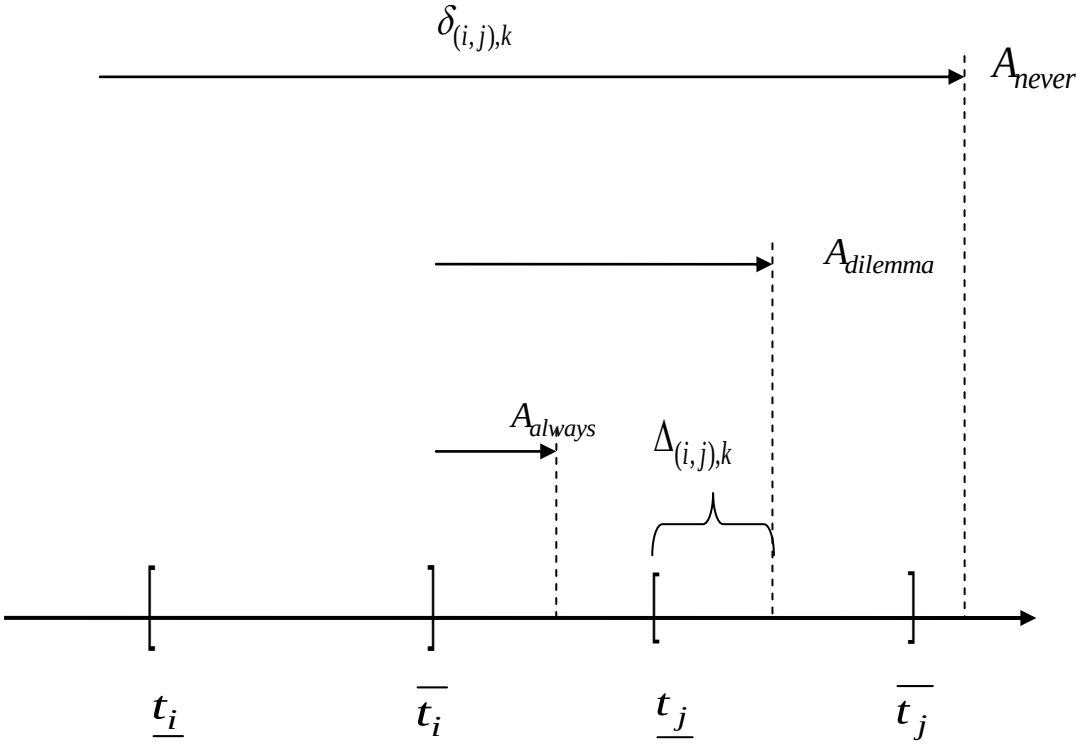


Figure 4.1: classification of deadhead trips.

For the deadhead arcs in A^k_{always} and A^k_{never}, it is clear that the constraint (4.20) is redundant, since for $(i, j) \in A^k_{always}$ the constraint (4.20) always holds, and for $(i, j) \in A^k_{never}$, the left side of (4.20) is negative, the inequality holds if and only if $e^k_{(i,j)} = 0$, i.e. this arc can never be taken. So we can even set all the parameters $u^k_{(i,j)}$ to 0 for $(i, j) \in A^k_{never}$, so that variables $e^k_{(i,j)}$ and $x^k_{(i,j)}$ will be fixed to 0 at the beginning.

Now only the arcs in $A^k_{dilemma}$ remain to appear in constraint (4.20). This is already a great reduction in the number of big-M constraints. And we also notice an interesting fact that the larger the sizes of our time windows are, the more deadhead arcs will join the set of $A^k_{dilemma}$. If we restrict the time window to a fixed value, the set $A^k_{dilemma}$ will decrease to an empty set.

Can we do it even better? The answer is positive. We define a parameter representing the time closeness between adjacent shipments as follows:

$$\Delta_{(i,j)}^{k} = \bar{t}_i + \delta_{(i,j)}^{k} - \underline{t}_j,\ \forall a_{(i,j)}^{k} \in A_{dilemma}^{k},\ \text{ and } \forall k \in T$$

where the parameter $\Delta_{(i,j)}^{k}$ represents the *interval closeness* of the two shipments i and j with vehicle type k. In the sequel we will estimate the bound of $\Delta_{(i,j)}^{k}$.

Let $\Delta_{(i,j)}^{k}$ be the interval closeness for a dilemma arc $a_{(i,j)}^{k} \in A_{dilemma}$, it follows from definition of dilemma arcs $A_{dilemma}$ that,

$$\bar{t}_i + \delta_{(i,j)}^{k} > \underline{t}_j \tag{4.23}$$

$$\underline{t}_i + \delta_{(i,j)}^{k} \le \bar{t}_j \tag{4.24}$$

which implies $\Delta_{(i,j)}^{k} = \bar{t}_i + \delta_{(i,j)}^{k} - \underline{t}_j \overset{(4.23)}{>} 0$,

and $\Delta_{(i,j)}^{k} = \bar{t}_i + \delta_{(i,j)}^{k} - \underline{t}_j \overset{(4.24)}{\le} \bar{t}_i - \underline{t}_i + \bar{t}_j - \underline{t}_j$.

To sum up, $\Delta_{(i,j)}^{k}$ has a bound:

$$0 < \Delta_{(i,j)}^{k} \le \bar{t}_i - \underline{t}_i + \bar{t}_j - \underline{t}_j \tag{4.25}$$

In other word, The interval closeness $\Delta_{(i,j)}^{k}$ for $a_{(i,j)}^{k} \in A_{dilemma}$ between two shipments i and j with vehicle type $k \in T$, is non-negative and bounded upwards by the sum of time window sizes of shipment i and j.

Then the constraint (4.20) can be reformulated as follows:

$$\bar{t}_i - t_i + t_j - \underline{t}_j - \Delta_{(i,j)}^{k} \ge M \bullet (e_{(i,j)}^{k} - 1),\ \forall a_{(i,j)}^{k} \in A_{dilemma},\ \text{ and } \forall k \in T$$

When $e_{(i,j)}^{k} = 0$, we should guarantee:

$$M \ge -\bar{t}_i + t_i - t_j + \underline{t}_j + \Delta_{(i,j)}^{k} \quad \forall a_{(i,j)}^{k} \in A_{dilemma},\ \forall \underline{t}_i \le t_i \le \bar{t}_i, \underline{t}_j \le t_j \le \bar{t}_j$$

Let t_i take its maximum $\bar{t}_i$ and t_j take its minimum $\underline{t}_j$, we conclude:

$$\mathrm{M} \ge \Delta_{(i,j)}^{k}, \quad \forall a_{(i,j)}^{k} \in A_{dilemma}$$

In this way we can greatly reduce the value of M,

$$M \overset{(4.25)}{\geq} \overline{t_i} - \underline{t_i} + \overline{t_j} - \underline{t_j},\ \forall a^k_{(i,j)} \in A_{dilemma}$$

i.e. the smallest value of M becomes dependent on the sum of the time window sizes of the adjacent shipments. For simplification reason, we can set $M = 720 \ll 12960$, as we know the shipments of all our test cases have a common time window size as 360.

4.5.2 Handling flow conservation constraint

In implementing the flow conservation constraint (4.8), we have applied the node-node adjacency list instead of the node-arc incidence matrix. Assume our network has *n* nodes and *m* arcs. In an adjacency list representation for a digraph, each node *i* is associated with an inflow list and an outflow list, node *j* is on the *i*th inflow list if there is an arc *(j, i)*; similarly node j is on the *i*th outflow list if the network contains an arc *(i, j)*. The incidence matrix is an $n \times m$ matrix, and it needs to store every entry, which takes $\Theta(mn)$ storage places. But note that in every column of the matrix, only 2 entries ($+1$ and -1 entry) are useful for our computation. In comparison with the node-arc incidence matrix, the adjacency list stores only the useful arc information, and it is not hard to deduce that the total amount of information stored in an adjacency list is $\Theta(m)$, which results in a noticeable saving in storage place and computation time in preprocessing.

5 Primal Heuristics

In general, NP-hard combinatorial problems like MDVSPTW with Coupling Trips, are solved in two approaches, exact algorithms and heuristic algorithms. The exact methods, e.g. Branch-and-Bound strategy for solving general integer linear programming problems, usually search the solution space in a systematic way, such that either a provable optimal solution is found, or we can prove theoretically at most how far our current best solution from the optimal solution is. Such exact methods usually have in worst case exponential complexity with respect to the problem instance size, which in practice causes a strong rise in computation time when the problem size increases. On the other side, although without guarantee of finding optimal solutions, heuristic methods (or heuristics) usually acquire good or even near-optimal solutions within a relatively low computational cost. Other than that, heuristics can also serve as a preprocessing step for providing a good initial feasible solution for a Branch-and-Bound procedure, which in practice yields a noticeable speed up. In this chapter, we will introduce two simple but robust construction strategies, PGrid and Randomized Greedy Search (RGS), for solving our MDVSPTW with Coupling Trips problem as well as general NP-hard problems.

5.1 A specification of MDVSPTW with Coupling Trips

In Chapter 4 we have modeled the problem of MDVSPTW with Coupling Trips as multicommodity flow network. Here our heuristics aim at solving MDVSP-CT with time windows and strict depot, i.e. the problem with flexible starting time windows and each vehicle must return to its own depot. Note that the heuristics can also be applied to problems with fixed timetables, which is equivalent to having a time window of a single value. Besides, a feasible solution of the strict depot version is also a special feasible solution for exchangeable depot version. So our heuristics solve a general problem that can provide feasible solutions to all the mathematical models introduced in Section 4.2 – 4.4.

As we know, in formulating a network flow problem, we can model it in two equivalent approaches: We can either define flows on arcs (each arc has an amount of flow x_i^k and the flow conservation holds); or we can define flows on paths and cycles.

First we claim that the multi-commodity network described in (4.22) does not contain any cycles, if each delivery arc has a length greater than 0, i.e. every shipment will take a non-trivial time to finish. We will try to give an informal proof by contradiction.

We assume that there exists a cycle C in the network, then C does not contain source node, sink node and any depot nodes, since they have either zero inflow or zero outflow. What if C contains only shipment nodes with delivery arcs and inter-task arcs? No problem, constraint (4.20) guarantees the total length of all arcs of the cycle adds up to 0, which is impossible because we assume the length of each delivery arc to be non-zero. So there exists no cycle in the network and we can decompose the network with flows on arcs to a network with flows on paths.

Furthermore, as the Flow Decomposition Theorem tells us, every directed path with positive flow connects a deficit node to an excess node. In our network described in (4.22) the only deficit node is the source node, and the only excess node is the sink node, which implies every path must start from the source node and end at the sink node. Moreover, a path can have flow of at most 2 units, since every delivery arc allows only no more than 2 units of flow (recall that at most 2 locomotive can service a train).

We will consider every path connects the source to the sink, and has only one unit of flow. Note that this is no loss of generality, if a path with flow of 2 units exists, we can easily decompose it further to two paths of one unit flow. And it also has a practical meaning, each path corresponds to a schedule of one single locomotive.

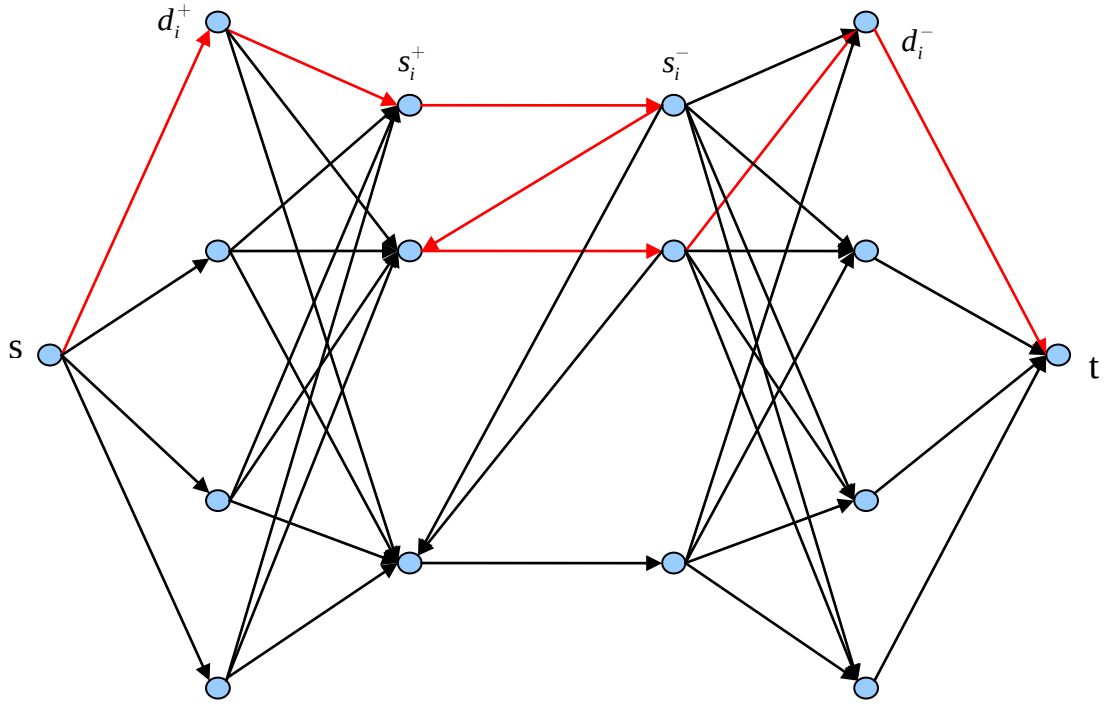


Figure 5.1: a path flow in our MDVSP network flow model, marked in red

Let P be the set of all possible paths in the network, and 2^P denotes the power set of P, i.e. the set of all subsets of P, and this is also the *search space* in our problem. A *feasible candidate solution* of MDVSPTW with Coupling Trips can be viewed as a set of paths, $P_s \in 2^P$ that *covers* each delivery arc exactly once. We clarify the term

that a set of paths *covers* a delivery arc $i \in A_{delivery}$ if and only if the arc flow x_i^k, which is accumulated by the paths passing this arc i, satisfies the bundle constraint (4.1) and the flow magnitude constraint (4.4), in other words, the shipment i is taken by sufficient vehicles of only one certain type. Note that in our project, there exists two types of vehicles (locomotives), namely, Type I locomotives are a class of weaker but faster locomotives, and Type II locomotives belong to the stronger but slower class. We can simplify our problems by differentiate the shipments (the trains in this case) into two types, Type I trains are the easier tasks that can be pulled by one locomotive of either class; while Type II trains are the heavier ones that can be either pulled by a single Type II locomotive or a couple of Type I locomotives, which corresponds to a "coupling feature".

Our network flow model shown in Figure 3.3 can actually be further simplified. Note that each origin shipment node s_i^+ has only one outflow node, i.e. the destination shipment node s_i^-; it holds for the inverse case as well, to be explicit, each destination s_i^- has only one inflow node s_i^+. In this case, we can actually merge these two nodes and the delivery arc connecting them into a single node, which represents a shipment. Furthermore, following constraint (4.7), the inflow of each start depot node d_i^+ equals the outflow of each end depot node d_i^-. Therefore we can even ignore the source node and the sink node, and then merge the both depot nodes d_i^+ and d_i^- into one single depot node. After the merging process explained above, a contracted network is shown in figure 5.2.
l

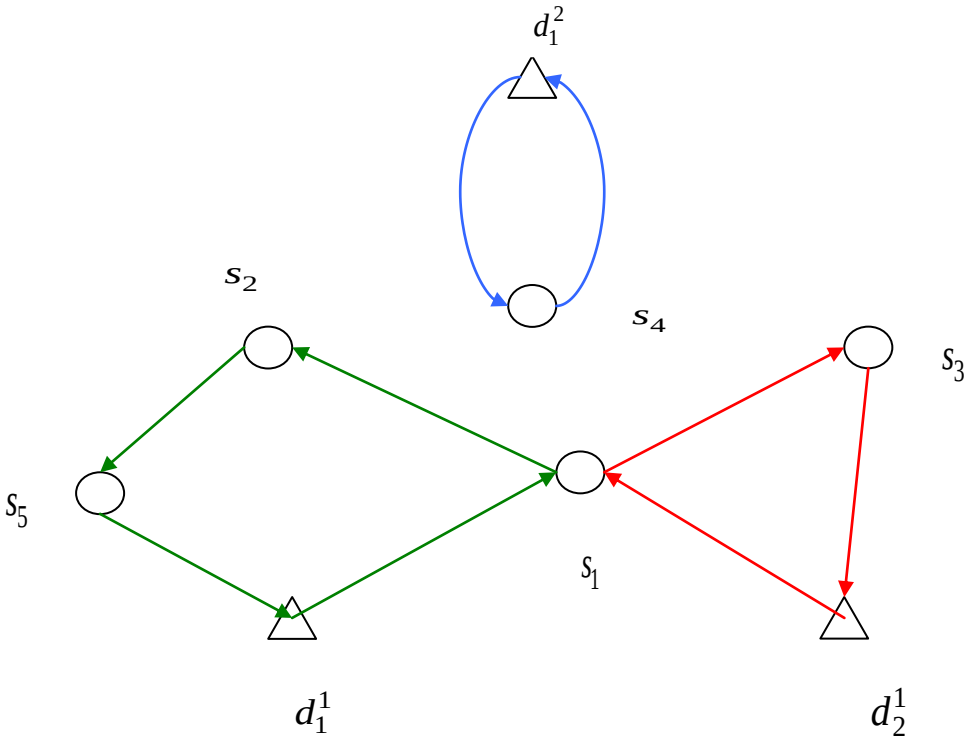


Figure 5.2: a contracted graph model of MDVSP with Coupling Trips. Note that the shipment s_1 in the graph is taken by two Type-I locomotives simultaneously.

Note that the contracted graph model shown above is consistent with the model of Vehicle Routing Problem (VRP), we recommend the interested readers to acquire further details in Toth & Vigo (2002).

5.2 A Greedy Constructive Search Paradigm for MDVSPTWCT

Before we introduce our metaheuristics PGrid and RGS, which requires running the greedy construction algorithm iteratively, we first start with introducing a greedy algorithm framework for our problem MDVSPTW with Coupling Trips.

5.2.1 The Big Picture of the Greedy Construction Search

As stated in Hoos & Stützle (2004), basically all computational approaches for solving hard combinatorial problems can be characterized as search algorithms. The fundamental idea behind the search approach is to iteratively generate and evaluate *candidate solutions*. Generally, the evaluation of candidate solutions much depends on the given problem, and is often straightforward to implement so as to have it done in polynomial time. The fundamental differences between search algorithms are in the way in which candidate solutions are generated. Normally they can be classified into two types, *constructive* or *local search* methods.

Constructive search methods (or *construction heuristics*) build a solution to a combinatorial optimization problem from scratch in an incremental way. Step by step and without backtracking, they add solution components until a complete solution is generated. Often, greedy construction heuristics are used which at each construction step add a solution component with maximum myopic benefit as estimated by a heuristic function.

Here first we will outline a greedy construction framework for solving MDVSP, details are explained in depth thereafter:

greedy(N)
Input: the multicommodity network N

Output: a set of paths P_s

(1) Initialize Depot Set D and Shipment Set S from N, $P_s := \varnothing$

(2) **while** (S is not empty)

(3) Select the best depot $d \in D$ to start the path p

(4) Select a best first shipment $s_{first} \in S$ and add to p

(5) **while** (path is not full and S is not empty)

(6) Select a best next shipment $s_{next} \in S$ and add to p

(7) **end while**

(8) $P_s := P_s \cup \{p\}$

(9) Update

(10) **end while**

(11) **return** P_s

Box 1: A greedy construction search paradigm for MDVSP with

coupling trips and time windows.

The construction of a feasible solution is composed of two parts, in a macro-scale sense we generate paths one by one to cover the previously uncovered shipments until all shipments are served; subsidiarily in a micro-scale sense, we build a path by incrementally appending unserved shipment to the end of the current path in the hope of covering as many shipments as possible, while at a subsidiary level, minimizing the driving time of unloaded trips (especially the deadhead trips).

In the sequel we will take a closer look at each step of the greedy construction search algorithm shown in the box above.

5.2.2 Initialize the Search Graph

Line (1) initializes the graph that the search is based on. The depot set D and the shipment set S is built up. These two sets contain all the solution components for our construction search, and during the search, empty depots and covered shipments will be removed from the respective set, so that only the available components will appear in our search phase. The goal of our construction search is to cover all the shipments, and the search process ends when S becomes empty. By some algorithms introduced in later sections that iteratively call the greedy construction search, the initialization should also restore some attributes of each components that is modified during the last search phase (e.g. the starting time interval of each shipment) to its original state, so that we can restart the search immediately without reading the input data again.

5.2.3 Select the Best Depot to Start

In line (3) we will select a best depot to start a path. Recall that a depot is a set of homogeneous vehicles, therefore the difference between the depots is the vehicle type and the depot geographical location. Since we assume that all locomotives are of the same starting cost, Type I and Type II locomotives are different in their capacity and speed. In this case, it is difficult to define a criterion for the goodness of a depot. Our selection phase is divided into two steps:

Determine which locomotive type to start. We denote v_1, v_2 the speed of Type I and Type II locomotives, and κ_1, κ_2 the capacity of Type I and Type II locomotives, respectively. Recall that Type I locomotives are weaker but faster than Type II locomotives, we have

$$\kappa_1 < \kappa_2, \quad v_1 > v_2$$

Note that there exist two types of shipments distinguished by their weight. The Type I locomotives are always favorable, if there are only the Type I shipments, i.e. the easier shipments, left in the shipment set S, since in that case all the easier shipments can be pulled by a single locomotive of both types, and Type I locomotives are of higher velocity. Similarly we can also assume if there is only Type II locomotives (the heavier locomotives) left in the shipment set S, Type II locomotives are always favorable, since these shipments require a combination of two Type I locomotives for enough pulling power, and the speed advantage of Type I locomotives is far from recovering its capacity deficiency. In general when Type I and Type II shipments both exist in S, to determine which vehicle type to start will be more subtle, neither will have dominant advantage. We set σ_1, σ_2 to be binary indicators as follows:

$$\sigma_1 = \begin{cases} 1, & \text{if there exists Type I shipments} \\ 0, & \text{elsewise} \end{cases}, \text{ and}$$

$$\sigma_2 = \begin{cases} 1, & \text{if there exists Type II shipments} \\ 0, & \text{elsewise} \end{cases}$$

And we define a linear scoring function for each vehicle type $score_{vt}$ as follows:

$$score_{vt}(i) = v_i \cdot \sigma_1 + \lambda \cdot \kappa_i \cdot \sigma_2, \quad \lambda \in \mathbb{R} \ \wedge \ \lambda \in [0, +\infty], \ i \in T \tag{5.1}$$

where λ is a non-negative parameter called *capacity weight*. As can be seen from above, if $\sigma_1 = 0$, i.e. only Type II shipments left in S, the velocity factor v_i has no influence on the scoring function, the locomotive type with largest capacity (in our case Type II locomotive) will stand out; if $\sigma_2 = 0$, i.e. the shipment set S contains only Type I shipments, then the capacity factor κ_i will lose all its influence, the locomotive type with higher speed will achieve better score. If $\sigma_1 = \sigma_2 = 1$, the *capacity weight* λ will come into play. Its value will decide the influence of the capacity factor over the velocity factor. It is easy to see that if we set λ to 0, only the velocity factor decides the score; if λ is set to $+\infty$, the capacity factor will dominate the velocity factor. We select the best vehicle type $t_{best} \in T$ according to the scoring function:

$$t_{best} = \arg\max_{t \in T} \{score_{vt}(t)\} \tag{5.2}$$

Determine depot of which location should be selected after the vehicle type is chosen. Normally the location can be selected randomly. But the selection process can be

further optimized by combine the depot location selection together with the selection of the first shipment. Normally we will select the earliest possible shipment in S to be the first shipment in the path, let $s_{earliest} \in S$ be it, and we use $l(s^{+}_{earliest})$ denote the location of the starting station of the earliest shipment, $d(l_1, l_2)$ denote the geographical distance between two locations $l_1, l_2 \in L$. Then we define the scoring function of each depot location with respect to a given locomotive type to be as follows:

$$score_{dl}(l) = d(l, l(s^{+}_{earliest})),\ l \in L \tag{5.3}$$

That means, the depot with location nearest to the potential first shipment should stand out. Putting it into formula, the best depot location $l_{best} \in L$ with respect to a vehicle type t_{best} is selected according to the given scoring function:

$$l_{best} = \arg\min_{l \in L} \{score_{vt}(l) : l \in L \wedge d^{l}_{t_{best}} \in D\} \tag{5.4}$$

Summing up, after the vehicle type t_{best} and the depot location l_{best} are determined, we have found our best depot $d^{l_{best}}_{t_{best}} \in D$ to start service. The method *selectDepot()* is shown in Box 2.

selectDepot(D)

Input: the depot set D, (optional) the location of the earliest possible shipment $l(s^{+}_{earliest})$

Output: the best depot $d_{best} \in D$

(1) **if** (D is empty)
(2) print error message("Sorry, out of locomotives!")
(3) exit
(4) **end if**
(5) **foreach** ($t \in T$)
(6) score t using (5.1)
(7) **end foreach**
(8) select the best locomotive type t_{best} using (5.2)
(9) **foreach** ($l \in L \wedge d^{l}_{t_{best}} \in D$)
(10) score l using (5.3)
(11) **end foreach**
(12) select the best depot location l_{best} using (5.4)
(13) $d_{best} := d^{l_{best}}_{t_{best}}$
(14) $|d_{best}|--$
(15) **if** ($|d_{best}|$ == 0)
(16) $D := D \setminus \{d_{best}\}$
(17) **end if**
(18) **return** d_{best}

Box 2: greedy method to select the best depot.

5.2.4 Select the Best First Shipment

Now we are back to the line (4) of Box 1. Our locomotive is started from depot $d \in D$, selecting the first shipment. The selection procedure will be different from the way we select the next shipments thereafter, so we distinguish it out. As already introduced above, the shipment which can start the earliest should be chosen. We formulize the scoring function for selecting the first shipment as follows:

$$score_{first}(i) = \underline{t_i},\ i \in S \tag{5.5}$$

And the best first shipment stands out when it has the minimum lower bound of time

interval:

$$s_{first} = \arg\min_{s \in S}\{score_{first}(s)\} \tag{5.6}$$

The *selectFirstShipment()* is shown in Box 3. Note that we also keep an global variable $t_{current}$, denoting the current time of the path. After the first shipment s_{first} is selected, $t_{current}$ is set to the earliest arrival time of the first shipment s_{first} at its destination station.

Everytime a shipment is selected, we will check whether this shipment is fully covered. If so, it is removed from the shipment set S; if not, we select the earliest possible starting time as the start time of the shipment, and shrink its starting time interval to a fixed time point at its start time, so that another locomotive of the same type can pick up this shipment at the same start time.

selectFirstShipment(S, k)
Input: the shipment set S, the locomotive type of the path k

Output: the best first shipment $s_{first} \in S$

(1) **foreach** ($s \in S$)
(2) score s using (5.5)
(3) **end foreach**
(4) select the best first shipment s_{first} using (5.6)
(5) **if** (s_{first} is covered)
(6) $S := S \setminus \{s_{first}\}$
(7) **else**
(8) $\overline{t_{s_{first}}} := \underline{t_{s_{first}}}$
(9) **end if**
(10) $t_{s_{first}} := \underline{t_{s_{first}}}$
(11) $t_{current} := \underline{t_{s_{first}}} + \delta^{dlv}_{s_{first}}$
(12) **return** s_{first}

Box 3: greedy method to select the best first shipment.

5.2.5 Select the Best Next Shipment

After the first shipment is selected, how to select the shipments that follow is essential for the solution quality. In the sequel we will look into the line (6) of Box 1, the method *selectNextShipment()*, in more details.

We select the shipments one by one and add it the current path, so as to
1) cover as many shipments as possible, so that the fewest possible shipments will be left for later constructions;
2) at a subsidiary level, minimize the driving time of unloaded trips (especially the deadhead trips).

In general, these two goals are not in great conflict with each other, but they will bring in a subtle difference to the construction strategy. The second goal will indicate that a geographical nearest neighborhood criterion should be applied when selecting the next node to be added. But a simple thought will show that this criterion is undesirable. Let's imagine that the geographical nearest neighbor is timetabled at the end of the week, then our locomotive will become very "lazy". It drives to visit its geographic nearest neighbor, and stop there idle for a whole week only to wait for this nearest neighbor to start. The first goal however, will suggest that the temporal nearest neighbor should stand out, i.e. a shipment with earliest possible starting time should be better, so that the sum of the deadhead driving time and the idle time of waiting for the next shipment to start should be minimized. Nevertheless, the geographical distance between the destination station of the previous shipment and the origin station of the next shipment can be viewed as a reasonable factor to be considered, when one decide to parameterize the greedy algorithm.

Now we will put what we have discussed above into formula. Suppose we have arrived at the destination station of shipment $i \in S$, with current time $t_{current}$, and we wish to select the best next shipment $j \in S$. Let $k \in T$ denote the locomotive type of the current path.

First we need to determine whether a connection to a candidate shipment is feasible. A shipment $j \in S$ is feasible to be appended after a shipment $i \in S$ in a path of locomotive type $k \in T$ under current time $t_{current}$, if and only if the following inequalities holds:

$$u^k_{(i,j)} > 0 \tag{5.7}$$

$$t_{current} + \delta^{dh}_{(i,j),k} \le \overline{t}_j \tag{5.8}$$

The inequality (5.7) says the deadhead trip connecting shipment i and j with locomotive type k is applicable. Recall from Chapter 4 that $\delta^{dh}_{(i,j),k}$ denotes the driving time of the deadhead trip from shipment i to shipment j with vehicle type *k*. The inequality (5.8) says the vehicle stands now in the end station of shipment i at $t_{current}$, if the shipment j is to be taken, the vehicle drives $\delta^{dh}_{(i,j),k}$ time to arrive at the starting station of shipment j. This connection is feasible if and only if the arrival time is no later than the latest possible starting time of shipment j, i.e. $\overline{t}_j$, the upper bound of the starting time interval of shipment j.

After the feasibility of shipment j is confirmed, how do we score it to measure its goodness? In this section, we will first define the scoring function in a simple way, following the temporal nearest neighborhood criterion. In the next sections we will look into some more complex, parameterize or randomly perturbed, scoring functions. Before we define the scoring function, we first evaluate the earliest possible starting time of shipment j following shipment i with path type k and current time $t_{current}$:

$$t_j \mid (i,k,t_{current}) = \max\{(t_{current} + \delta^{dh}_{(i,j),k}), \underline{t}_i\}, \quad i \in S,\ k \in T,\ t_{current} \in \mathbb{R}^+ \qquad (5.9)$$

The equation above shows, after the vehicle drives along the deadhead trip to the starting station of shipment j, if the arrival time falls within the starting time interval of j, i.e. $\left[\underline{t}_i, \overline{t}_i\right]$, then we can start shipment j immediately; if the arrival time falls before the lower bound of starting time interval of j, i.e. $\underline{t}_i$, the vehicle has to wait until the earliest time that the shipment j can start, i.e. $\underline{t}_i$.

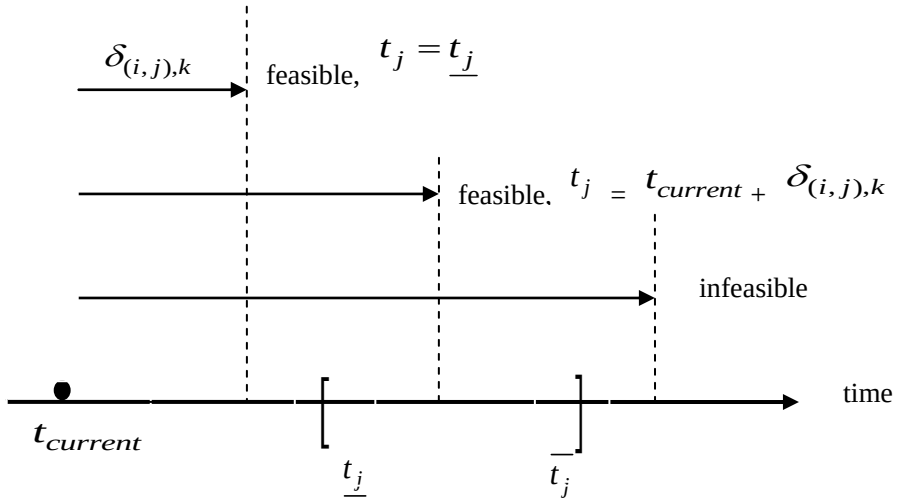


Figure 5.3: an overview of possible starting time of shipment j

A scoring function to measure the temporal distance of shipment j is given as follows:

$$score_{next}(j \mid (i,k,t_{current})) = t_j \mid (i,k,t_{current}) - t_{current},\ i \in S,\ k \in T,\ t_{current} \in \mathbb{R}^+ \quad (5.10)$$

We are expecting the temporal nearest neighbor to stand out, which means it should have the minimum score:

$$s_{next} = \arg\min_{s \in S} \{score_{next}(s \mid (i,k,t_{current})) \mid s \in S \wedge s \text{ satisfies (5.7), (5.8)}\} \quad (5.11)$$

The method *selectNextShipment* is shown in Box 4.

selectNextShipment(S, i, k, $t_{current}$)

Input: the shipment set S, the previous shipment $i \in S$, the locomotive type of the path k, current time $t_{current}$

Output: the best first shipment $s_{next} \in S$

(1) **foreach** ($s \in S$)
(2) **if** (s satisfies (5.7), (5.8))
(3) score s using (5.10)
(4) **end if**
(5) **end foreach**
(6) select the best next shipment s_{next} using (5.11)
(7) **if** (s_{next} is covered)
(8) $S := S \setminus \{s_{next}\}$
(9) **else**
(10) $\overline{t_{s_{next}}} := \underline{t_{s_{next}}}$
(11) **end if**
(12) determine $t_{s_{next}}$ using (5.9)
(13) $t_{current} := t_{s_{next}} + \delta^{dlv}_{s_{next}}$
(14) **return** s_{next}

Box 4: greedy method to select the best next shipment.

5.2.6 Computational Complexity of the Greedy Construction search

In the following we will make a brief evaluation about the computational complexity of our greedy constructive search introduced in Section 5.2 above. We first need some parameters to describe the instance size.

- $n = |S|$ denotes the number of shipments;
- $k = |D|$ denotes the number of depots;
- m denotes the total number of vehicles, and m can be computed as follows:

$$m = \sum_{i=1}^{k} |\, d_i \,|, \quad d_i \in D \tag{5.12}$$

where $|\, d_i \,|$ denotes the number vehicles contained in depot d_i.

The outer loop in line (2) of Box 1 is called every time when a path is built. We can bound the number of paths $|\, P_s \,|$ as follows:

$$|\, P_s \,| \leq \min\{n, m\} \tag{5.13}$$

That is because the number of paths can exceed neither the number of vehicles nor the number of shipments. For every path we will select a best depot at the beginning following Box 2, scoring each depot takes constant time, and we will at most score for each depot once, so the depot selections take at most $O(\min\{n, m\} \bullet k)$ time.

Each shipment can be selected at most twice by two different paths, and every time we are selecting a shipment, no matter as the first shipment or next shipment, each available shipment will be scored in constant time (see Box 3 and Box 4), and at most n shipments are to be scored. So the shipment selections take at most $O(2n \cdot n) = O(n^2)$ time.

All in all, it requires $O(\min\{n, m\} \bullet k + n^2)$ time for a greedy construction.

The memory space required by the greedy algorithm is exactly the memory space needed for storing the multicommodity flow network, which is mentioned in Chapter 3 to be $O(kn + n^2)$.

5.3 How to improve the greedy construction search

As can be seen from above, a greedy construction search algorithm builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Some classical greedy algorithms, despite their natural myopic behavior, are able to find optimal solution for some combinatorial optimization problems, like Dijkstra's algorithm for finding a shortest path between two nodes in a graph with non-negative arc weights, and Prim's algorithm for solving minimum spanning tree problem, etc. However, for hard combinatorial problems, i.e. the provable NP-hard problems, like TSP or in our case MDVSPTWCT, no greedy-type algorithms are known to guarantee the optimal solution. Nevertheless, greedy constructive algorithms are still an attractive and effective problem solving strategy,

and a careful tuning of the algorithm is essential for the solution quality. In the sequel we will introduce a few strategies to improve the greedy construction algorithm.

In this section we will first give a strategy for handling the problem specific difficulty of coupling trips in subsection 5.3.1, and describe the general guideline for tuning a constructive greedy search algorithm in 5.3.2.

5.3.1 Handling Coupling Trip Feature

As we will see in Chapter 7, the first naïve version of the greedy algorithm introduced in the last section does not perform well. One of the main problems is handling the coupling trip feature. Our experiment shows that for the heavy shipments, i.e. the shipments that requires a twin combination of Type I locomotives or a single Type II locomotive, we should put strong guidance to encourage them to be pulled by Type II locomotives, and meanwhile avoid Type I locomotives to serve the heavy ones. There are 3 reasons for doing so,

1) the trip with one Type II locomotive will definitely consume less energy than the sum of two Type I locomotives taking the same trip;
2) once a heavy shipment is taken by a Type I locomotive, this shipment will become unlikely to be selected by a second Type I locomotive, since its starting time is fixed by the first locomotive;
3) if Type I locomotives are run out, but some heavy shipments are still taken by just one Type I locomotive, the partial solution will become infeasible, since in this case the remaining Type II locomotives are helpless.

Unfortunately the second argument above makes the infeasibility disaster described in the third argument occur often.

To overcome the coupling trip difficulty, we propose the following 2 steps:

1) increase the probability that Type II locomotive depots should be selected at the early construction phase, to avoid the third catastrophe. This can be easily achieved by enlarging the parameter *capacity weight* λ used in equation (5.1).
2) introduce a new parameter *double penalty* ω to stimulate Type II locomotives to pull heavy shipments, as well as Type I locomotives to stay away from them. The scoring function of selecting next shipment is modified as follows:

$$score'_{next}(j\,|\,(i,k)) := \begin{cases} \dfrac{score_{next}(j\,|\,(i,k))}{\omega}, & \text{if } k = 2 \ \wedge\ s_j \text{ heavy} \\ score_{next}(j\,|\,(i,k)) \cdot \omega, & \text{if } k = 1 \ \wedge\ s_j \text{ heavy and not taken} \\ \dfrac{score_{next}(j\,|\,(i,k))}{\omega}, & \text{if } k = 1 \ \wedge\ s_j \text{ heavy and partly taken} \\ score_{next}(j\,|\,(i,k)), & \text{else} \end{cases} \tag{5.14}$$

where $s_j \in S$, $k \in T$, and the *double penalty* $\omega > 1$. The equation (5.14) says, a heavy shipment should be favored for a Type II locomotive, and should be punished by a Type I locomotive, with exception that if the heavy shipment is partly taken, i.e. it is already taken by one Type I locomotive, then it should be favored in a second Type I locomotive's selection phase.

Note that a reasonable *double penalty* ω should be greater than 1, but its value should not be too large, otherwise some "lazy Type II locomotives" will occur.

5.3.2 Intensity versus Diversity

A common strategy to improve a constructive search algorithm is to follow with a local search phase. But the prerequisite to perform a local search is to define an effective neighborhood relation for a solution space, which is not always easy. We will discuss it in more details in section 5.7.

The algorithms we describe in the following sections 5.4 – 5.6, are based on iteratively running the greedy construction search algorithm without local search. As is shown in Chapter 7, iterative greedy construction search algorithms yield very good solutions for MDVSPTWCT, when they are carefully tuned. One general principle for tuning such kind of construction heuristics is to balance between intensity and diversity of the searchable space of the given problem instance.

- **Searchable Space:** is a subset of the whole solution space that is reachable by the iterative greedy construction search, and every element of it has a hitting probability. Note that the searchable space is usually not complete, the setting of different parameter combinations will affect its structure.
- **Intensity:** the searchable space becomes intensive, when it is in a concentrative manner. The space focuses on relatively less elements, and some good elements are biased with higher hitting probability, by strongly guiding the construction phase via the greedy evaluation function.
- **Diversity:** the term diversity means, to explore the solution space in a reasonably extensive manner, to cover more elements and to prevent the search process from stagnation, i.e. to make sure our search area won't be restricted to some region without high quality solutions in it.

An extreme example of intensity is to perform the greedy construction search in a "static" way, such that in each partial solution construction step there is always a unique best solution component to be selected and added. In such case, every greedy construction will return exactly the same solution. One possible diversification is, at each partial solution construction step, we store the set of the solution components that achieve the equally best evaluated value, and select one element from the set

uniformly, i.e. solution components that share the same best score have an equal chance to stand out. In this case we for example can rewrite equation (5.11) as follows:

$$S_{\min} = \arg\min_{s \in S} \{score_{next}(s \mid (i,k,t_{current})) \mid s \in S \wedge s \text{ satisfies (5.7), (5.8)}\} \quad (5.15)$$

And

$$s_{next} \in S_{\min} \text{ uniformly randomly drawn} \quad (5.16)$$

In general, admitting random selection of only best solution components with equally good scoring value still generates only limited number of different candidate solutions. In order to increase diversity of the searchable space, Hart and Schogan (1987) proposed *semi-greedy* heuristics, which later become the construction part of the well-known metaheuristic GRASP (Greedy Randomized Adaptive Search Procedures, see Feo and Resend (1989); (1995); Hoos and Stützle (2004)).

In contrast to standard greedy construction search methods, *semi-greedy* does not necessarily select the best scored solution component. Instead, it generates in each construction step a *restricted candidate list* (RCL), and then selects one element out of the RCL randomly according to a uniform distribution. In *semi-greedy* there are two different ways to define RCL:

i. **cardinality restriction**: only the *k* best ranked solution components are included in the RCL;
ii. **value restriction**: let *S* be the set of feasible solution component, and $s \in S$. Let $g(s)$ denote the greedy scoring value of *s*. We further define $g_{\min} = \min\{g(s) \mid s \in S\}$ and $g_{\max} = \max\{g(s) \mid s \in S\}$. Then the solution component *s* is admitted into the RCL, if $g(s) \le g_{\min} + \alpha(g_{\max} - g_{\min})$.

k and α used above are parameters of the algorithm. Clearly, the smaller *k* or α, the more intensive the selection will be.

Semi-greedy brings in considerable diversity to the standard greedy construction search, but as we will see in chapter 7, this strategy does not yield enough satisfactory solutions, since it does not give bias to the elements in the RCL according to their quality, which makes the searchable space too extensive, prevent the best solutions from standing out. In section 5.5, we will discuss another diversification strategy Randomized Greedy Search (RGS), which restricts the number of candidate solution components as well as biases the selection according to the quality of the solution component measured by a given scoring function. GRASP improves *semi-greedy* with a subsidiary local search, and all local search issues will be discussed in section 5.7.

5.4 Parameterize the greedy algorithm with grid search (PGrid)

The first strategy is to parameterize a greedy algorithm. It is well known that the scoring function (or heuristic evaluation function) plays an essential role in the success of a greedy algorithm. But how to define a good scoring function is quite problem-specific, and sometimes even instance-specific. Most of the classic greedy algorithms use a rather straightforward heuristic evaluation function, since the immediate benefit of a solution component is rather obvious, e.g. nearest neighborhood heuristic for TSP, which always selects the next node with the minimum distance from the previous selected node; or the heuristic evaluation function has been proved to be optimal, e.g. Dijkstra's algorithm and Prim's algorithm that use a label to measure the goodness of each remaining nodes (see Nemhauser and Wolsey (1999)). But we notice that in a highly constrained combinatorial optimization problem like MDVSPTWCT, if the algorithm is only greedy for a single objective, then it is usually fragile, sometimes even misleading. To make the greedy approach more robust, we introduce a parameterized greedy approach called PGrid. We first introduce more criteria for our greedy goal, and try to parameterize them and find the best parameter combination.

The strategy of using a parameterized scoring function to improve a greedy algorithm is given a term in Fügenschuh (2005) as *parameterized greedy heuristic* (or *pgreedy* for short). As mentioned in Fügenschuh (2005), the following 3 characteristics are essential for a pgreedy heuristic:

1. introduce more than one criteria into the greedy scoring function;
2. parameterize these criteria in a linear scoring function;
3. use a parameter tuning technique to find the best parameter combination.

In our project we have modified the guideline described above a little bit. During the greedy construction phase we have three steps, select depot, select first node, and select next node. And therefore instead of one greedy scoring function we have in total three scoring function for each step (see section 5.2). Besides, non-linear scoring function is also allowed. We reformulate the pgreedy framework as follows:

1'. Introduce more than one criteria in more than one scoring functions;
2'. Parameterize these criteria in not only linear scoring function, but also non-linear parameterizations are allowed, e.g. the parameter *double penalty* ω introduced in section 5.3.1;
3. Use a parameter tuning technique to find the best parameter combination.

Finding new criteria for the scoring function is in general more an art than a science.

There are two types of criteria to be selected:

- Reasonable criteria: like the capacity and velocity of a vehicle are important factors to be considered, when selecting a vehicle to start; or like the temporal and geographic distance of the next node from the previous node when selecting next node.
- Perturbation criteria: in selecting next node, we can add some more "non-sense" criteria to the scoring function, such as the duration of the next node, the time window size of the next node and even the index of the next node. How to apply these perturbation criteria are according to the trade-off of intensity and diversity of the given problem instance. When the searchable space becomes too intensive, too restricted, we can introduce some perturbation criteria to bring more distraction to broaden the searchable space.

Since there are more than one scoring functions and parameterization is possibly not linear, we cannot bound the parameter space within a unit ball as in Fügenschuh 2005. However, we can still bound each parameter manually, within a reasonable range.

In order to find the best parameter combination, our first experiment for the parameter tuning is to apply a grid search. We divide the parameter cube into regular grid, and to try out each parameter combination at each vertex of the grid.

The hybrid of pgreedy and grid search is simple but robust and powerful in practice, as shown in chapter 7. But this algorithm still has some drawback, especially tuning parameters with grid search is relatively a naïve approach. For further development, we recommend to tune the parameter using techniques from machine learning aspect, like "F-race" mentioned in Stützle et. al. 2002, or techniques from global optimization, like Improving Hit and Run mentioned in Fügenschuh 2005.

5.5 Randomized Greedy Search (RGS)

Another strategy to bring diversity into the standard greedy construction search algorithm is to multiply the greedy scoring function with a perturbation factor, which is drawn from some probability distribution. We suggest the term *Randomized greedy Search* (RGS) for such procedures.

Similar to the semi-greedy heuristic, it is not necessary that only the best-scored solution components are able to be selected. But the better the score, the higher the chance that the solution component will picked up.

Let S be the set of feasible solution component, and $s \in S$. Let $g(s)$ denote the

greedy scoring value of *s*. The we set $g_{\max} = \max\{g(s) \mid s \in S\}$, and $g_{\min} = \min\{g(s) \mid s \in S\}$. We further define $h(s) = g(s) - g_{\max} \le 0$ to be a "normalized" greedy heuristic scoring value of *s*. We multiply the normalized greedy score $h(s)$ with a perturbation *p*, and acquire the perturbed normalized greedy score $h_p(s) = h(s) \bullet p$, where p is positive real number randomly generated from some probability distribution.

We have applied two different probability distributions for *p* in our experiments:

- Uniform distribution within the value interval [*l*, *u*], e.g. [800, 1000]. If we set $\alpha := \frac{u-l}{u}$, it becomes similar with semi-greedy heuristic using RCL with value restriction and quality parameter α, solution component *s* might be possible to enter the candidate list, if $g(s) \le g_{\min} + \alpha(g_{\max} - g_{\min})$. But unlike semi-greedy, every element inside the candidate list has a biased probability to be selected. It is easy to see that, the better the heuristic score, the higher the probability that the solution will be chosen.

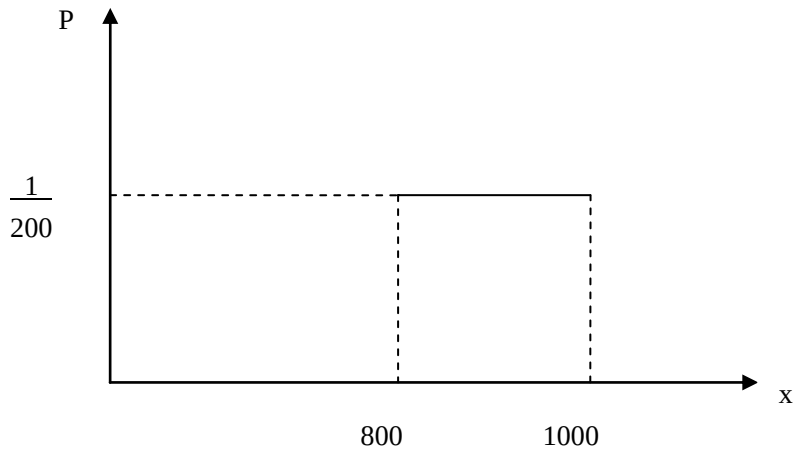


Figure 5.4: probability density function of uniform distribution in [800, 1000].

- Normal distribution with $N(\mu, \sigma)$, e.g. *N*(1000, 50). With *p* drawn from this distribution, theoretically every feasible solution component will be possible to be chosen, since normal distribution spreads from $-\infty$ to $+\infty$. For the distributed

values are relatively centralized, solution component with better score will have bigger advantage than in the uniform distribution. That is to say, comparing with the uniform distribution, better solution components are even more likely to be chosen, while all other solution components still retain its possibility to be selected. In practice the perturbation parameter *p* drawn from normal distribution usually performs better than the one generated from uniform distribution.

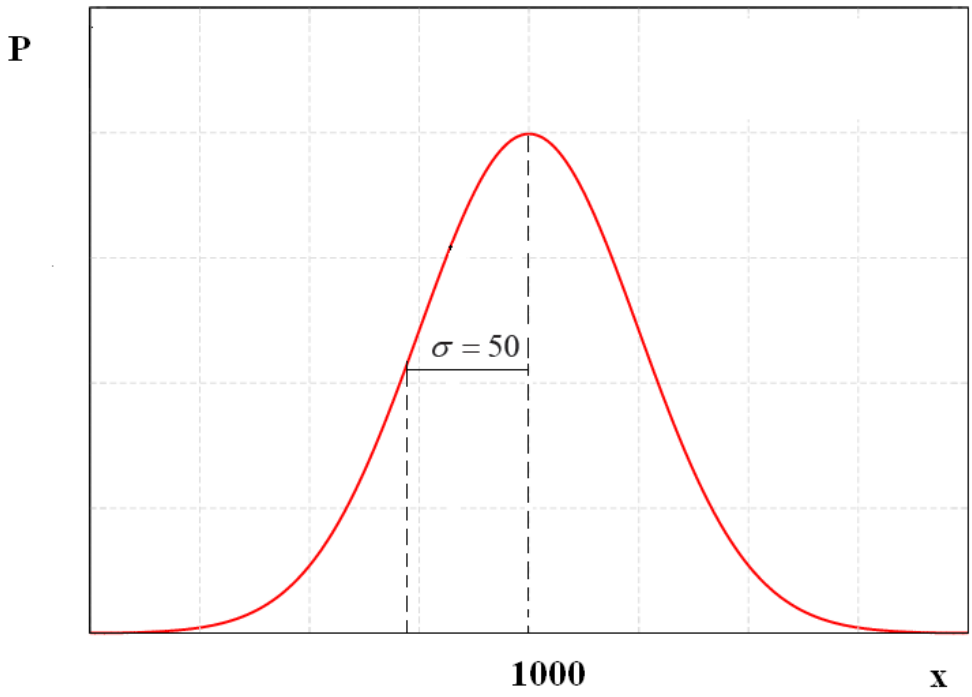


Figure 5.5: probability density function of normal distribution $N(1000, 50)$.

Note that the larger the perturbation interval size in uniform distribution, or the larger the standard deviation σ in normal distribution is, the more diverse our searchable space will be, i.e. the bigger the searchable space we might reach, and the more divergent the search will be performed, which means it might take a longer time to reach a very good solution, but the probability to hit a perfect value is increased.

5.6 The Hybrid of PGrid and RGS: PRGS

There are two ways of hybridizing PGrid and RGS:

i. As we have seen in section 5.5, the algorithm RGS starts with some scoring function, and tries to play some perturbation on it so as to explore it more extensively. Usually the scoring function plays also an essential role in a

successful implementation of RGS. Therefore we can first apply PGrid to find a good parameter combination, and then follow by an extensive exploration of this scoring function with RGS. This usually performs better than applying RGS to the standard greedy scoring function.

ii. The second way to hybridize PGrid and RGS is to combine them together from the beginning. For every parameter combination at each grid vertex we spend some more iterations with a perturbation from RGS. Note that PGrid and RGS are both diversification mechanisms, the hybrid of PGrid and RGS in such a way will considerably increase the diversity of the searchable space, and it does not necessarily improve the solution quality. It really depends on the trade-off of intensity and diversity of the searchable space, and should be only applied when the searchable space is too concentrative, and needs to bring in distraction.

5.7 Local search and further topics

A common strategy to improve a constructive search algorithm is to follow with a local search mechanism, so that the solution can be improved by finding its local optimum in the search space specified by some neighborhood relation. To define a effective neighborhood relation for a highly constrained combinatorial problem MDVSPTWCT or for its simplified model Vehicle Routing Problem (VRP) is not at all easy. Some researchers have mentioned to apply a 2-exchange neighborhood adopted from TSP, but it only saved minor costs in driving time. Some more effective neighborhood relations for VRP are introduced in details in Kindervater and Savelsbergh 1997. Another difficulty in performing local search for such problems are the implementation of the constraint propagation, an example is given in Fuegenschuh 2005.

Apart from parameter tuning techniques mentioned in section 5.4 for pgreedy, F-race and IHR, there are other promising directions we can further improve our construction heuristic solution in the future. One worth mentioning is a simple and effective metaheuristic algorithm called Iterated Greedy (IG), which destroy part of the solution and try to reconstruct it again, see Ruiz and Stuetzle 2005. It is to expect, a hybrid of pgreedy and IG will be powerful in practice. Also Dorigo and Stuetzle 2004 has reported that a variance of Ant Colony Optimization (ACO) algorithm, named ACS, has yield state-of-the-art results for Vehicle Routing Problem.

6 Input Data

In this chapter we will have a close look at the 3 test instances of our problem. We start from section 6.1 by introducing a very tiny hand-made instance. Section 6.2 will give details of the 2 real-world instances.

6.1 A Tiny instance

For demonstration use we have artificially constructed a tiny instance under the name "Tiny".

In this instance we have included the 4 major cities in Germany: Berlin, Frankfurt (am Main), Hamburg and Munich. The time that the deadhead trip needs to travel between each 2 cities is listed below (the following data is retrieved and approximated from the database of German Railway, unit in hours. The first figure is the time required for Type I locomotive, the figure after the slash represents the deadhead time for Type II locomotive.):

	Berlin	Frankfurt	Hamburg	Munich
Berlin	--	4 / 5	2 / 3	7 / 8
Frankfurt	4 / 5	--	4 / 5	4 / 5
Hamburg	2 / 3	4 / 5	--	6 / 7
Munich	7 / 8	4 / 5	6 / 7	--

As can be seen from above, Type I locomotive always drives at a higher speed than Type II locomotive, but Type II locomotive are more powerful than its Type I alternative.

Note that in the table above, the driving time of the deadhead trips are symmetric, i.e. the driving time from city *i* to city *j* equals the driving time from city *j* to city *i*. It is so constructed in this instance for convenience, but in general, it does not have to be.

The following table shows the depot information. Recall that a depot owns a set of homogenous vehicles (in our case locomotives), at a specific location. We show below how many locomotives each depot owns:

	Type I Locomotive	Type II Locomotive
Berlin	1	0
Frankfurt	1	1
Hamburg	2	1
Munich	2	0

So there are in total 11 locomotives, located in all the 4 cities, 2 locomotives out of 11 belong to the stronger but slower Type II.

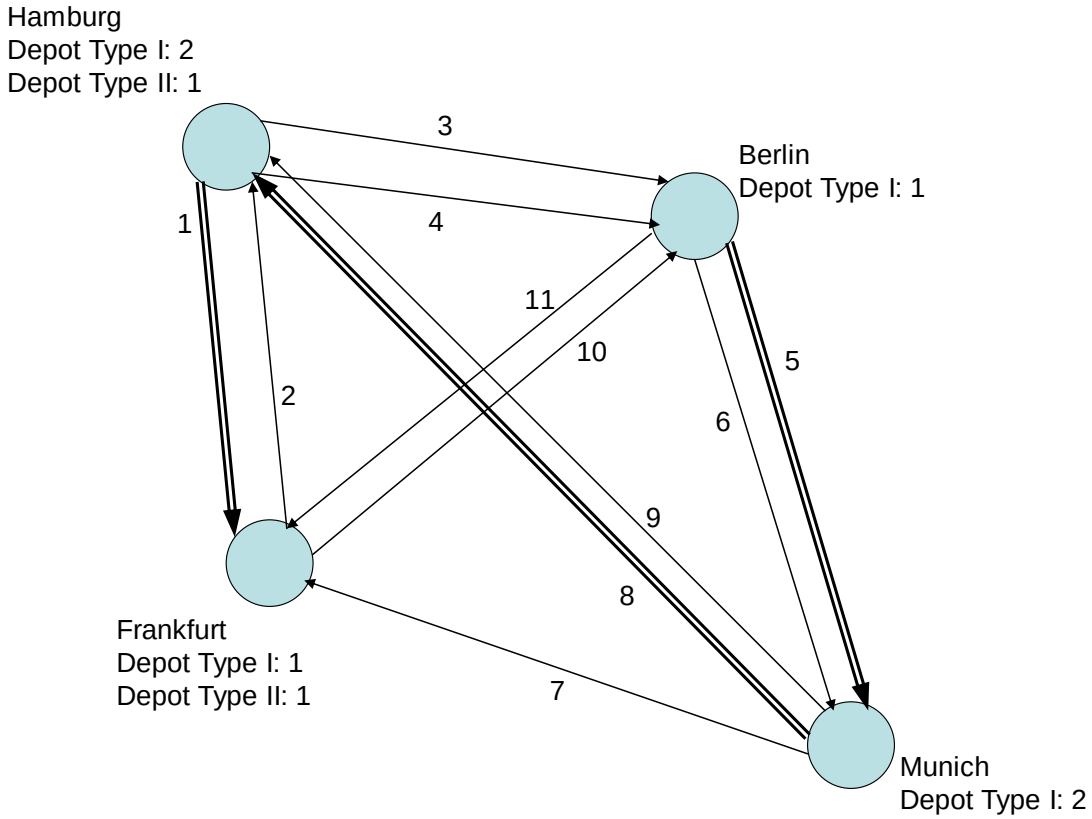
In the sequel we will introduce the list of our shipments, and a timetable attached with them. In the timetable, the arrival time of each shipment is fixed, named deadline of each shipment, but the starting time of each shipment is depending on the speed of the locomotive, i.e. the locomotive type, so we have for each shipment a separate starting time for each locomotive type, the one on top is for Type I locomotive, and the one below is for Type II. The scheduling period is 48 hours, starting from 0 to 48, with time unit as hour:

index	origin	Starting time	Arrival time	Destination	Weight
1	Hamburg	8 5	14	Frankfurt	2
2	Frankfurt	5 3	14	Hamburg	1
3	Hamburg	15 13	22	Berlin	1
4	Hamburg	6 4	14	Berlin	1
5	Berlin	17 14	26	Munich	2
6	Berlin	27 25	35	Munich	1
7	Munich	32 29	39	Frankfurt	1
8	Munich	36 35	48	Hamburg	2
9	Munich	33 30	44	Hamburg	1
10	Frankfurt	15 14	25	Berlin	1
11	Berlin	34 31	42	Frankfurt	1

For simplicity, the weight of the shipment is set to 1, if it can be accomplished by a single Type I locomotive or a single Type II locomotive; the weight of the shipment is set to 2, if it can only be pulled by a single Type II locomotive or a twin combination of Type I locomotive, i.e. 2 Type I locomotives will serviced the shipment together.

The shipments described above are also shown in Figure 7.1, with the index number of each shipment on the shipment arc. Note that in Figure 7.1, we use the double weighted arrows to represent the heavy shipments, which can be pulled alone by Type II locomotive, or by a twin combination of 2 Type I locomotives.

If we are considering solving a model with time windows, a time window size of 6 hours is introduced for each shipment.



6.2 Real-World Data

From a large amount of customer raw data, we have extracted 2 test instances, under the name KW47 and KW48, referring to the disposition schedules the logistics company made in the 47th and the 48th calendar week, respectively. What makes things complicated is, not all data we need is given from the customer, which means, some details will need to be simulated.

All we have as our raw data is a list of disposition timetable. The timetable is made on a weekly base, at the end of each week, a timetable for the next week should be

announced. The timetable contains details of each shipment, their starting city, their destination city, the starting time of the train, the arrival time, etc. But according to this timetable, some of the trips are taken by Type I locomotive, the faster one, and some of them are taken Type II locomotives, but only one trip duration for each trip is known, denoted as δ_i^{raw}. Therefore we assume all the trip durations given are for Type I locomotive. And we keep the arrival time of each shipment fixed, and recompute the trip duration for Type II locomotive by multiplying the timetabled trip duration by a locomotive speed ratio λ^{dlv}.

$$\delta_{i,1}^{dlv} = \delta_i^{raw}, \quad \delta_{i,2}^{dlv} = \lambda^{dlv} \cdot \delta_i^{raw}$$

We decide to take the value of λ to be 1.2, which means in general, the Type I locomotives are 20% faster than the Type II locomotives. Thus in reality the shipment transportation duration is always less or equal to our estimated trip duration.

Another unknown detail from the timetable is the weight of each shipment. As we know, the shipments are classified into 2 classes by weight, namely the normal and heavy shipments. The normal shipments can be pulled by one locomotive of any type, while the heavy shipments have to be pulled by two Type I locomotives or one Type II locomotive. As we learn that there are only minor heavy shipments, we have set the heavy shipment percentage η_{heavy} to be 25%, and choose 25% of shipments randomly as heavy ones.

Since we also have no information about our deadhead trip durations $\delta_{(i,j),k}^{dh}$, we have consulted the database of German Railway (Deutsche Bahn) for passenger trip durations. We first extract all the cities that ever appear in the disposition timetable, about 40 of them, and check the German Railway database for passenger trip durations of any two cities, denoted by $\delta_{(i,j)}^{passenger}$. And we set the deadhead trip duration as follows:

$$\delta_{(i,j),1}^{dh} = \delta_i^{passenger}, \quad \delta_{(i,j),2}^{dh} = \lambda^{dh} \cdot \delta_{(i,j)}^{passenger}, \text{ where } \lambda^{dh} = 1.2$$

We are doing so under 2 assumptions, firstly the logistic company shares more or less the same tracks with German Railway passenger trips; secondly, we assume that a single locomotive can drive at least as fast as pulling a whole passenger train. So in reality the deadhead trip duration should be less or equal to our estimated time.

The information about depots and locomotives is also missing. But we are given the information that there are about 60% Type I locomotives and 40% of Type II. We have assumed for the two test instances in total 27 locomotives are in service, and they are uniformly distributed in 10 to 12 randomly selected locations.

In solving the problem with time windows, we will introduce a time window of 6 hours, i.e. 360 in minutes, around the arrival time given from the timetable for each shipment.

$$\bar{t}_i - \underline{t}_i = 360 \quad \forall i \in S$$

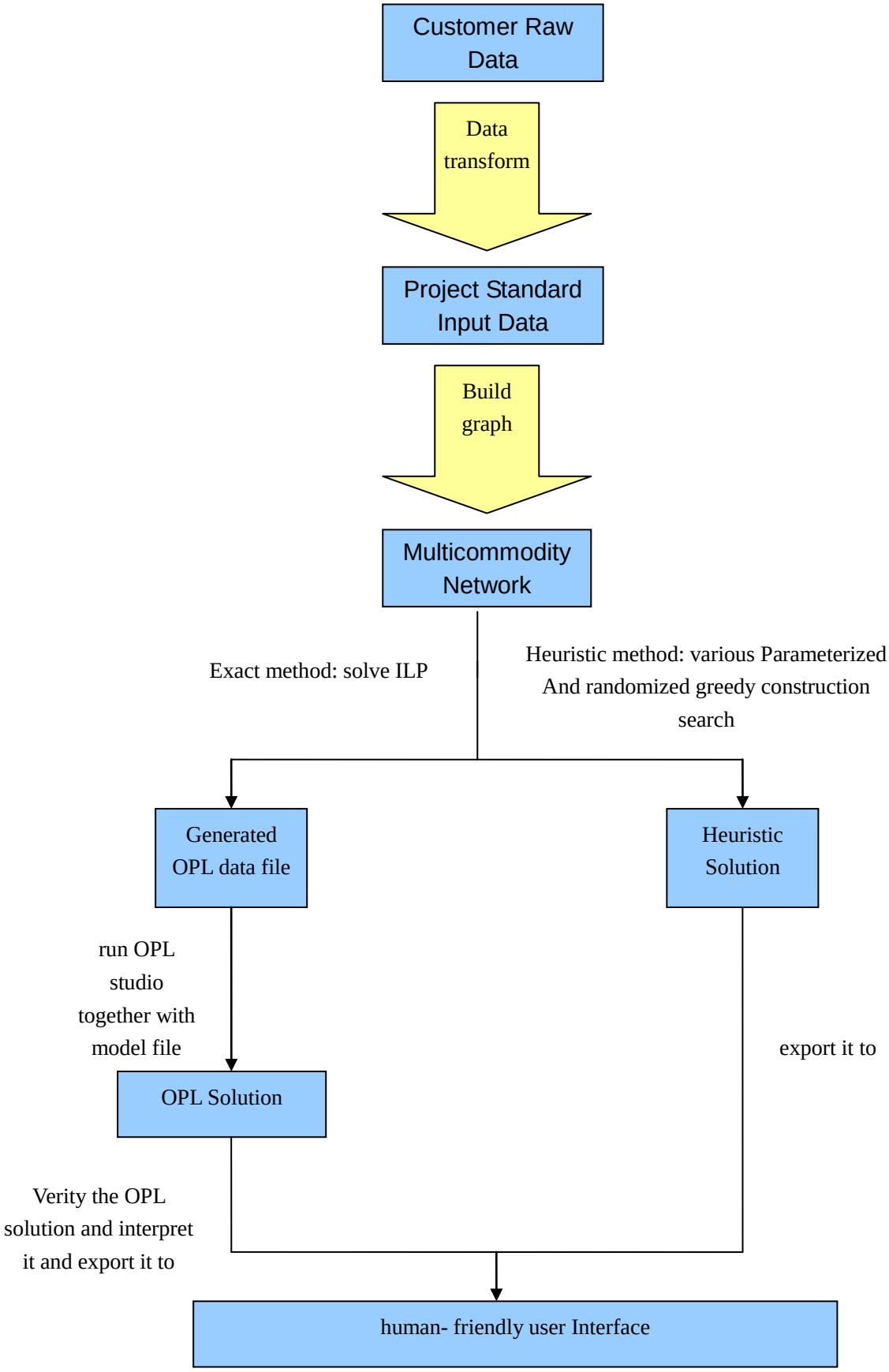
To sum up, the technical details of all the 3 test instances are listed below, we denote S the set of shipments, D the set of depots, T the set of locomotive types, N the set of all nodes of the multicommodity network, A the set of arcs, $|A|$ the number of arcs within one layer of the network, TW size stands for the size of the time window

	$\|S\|$	$\|D\|$	$\|T\|$	$\|N\|$	$\|A\|$	TW size (if considered)
Tiny	11	6	2	36	197	6
KW47	172	13	2	372	18299	360
KW48	178	11	2	380	18814	360

7 Computational Results

7.1 Solution Process of our software Locomotive Scheduler

The software is designed in an object-oriented manner, implemented using Java programming language. Our *Locomotive Scheduler* adopt two approaches to assign locomotives to the shipments, the exact methods with the help of a commercial ILP solver ILOG OPL Studio, and the self-developed heuristic methods implemented in Java.


Customer Raw Data
Data transform
Project Standard Input Data
Build graph
Multicommodity Network
Exact method: solve ILP
Heuristic method: various Parameterized And randomized greedy construction search
Generated OPL data file
Heuristic Solution
run OPL studio together with model file
export it to
OPL Solution
Verity the OPL solution and interpret it and export it to
human- friendly user Interface

7.2 Solving the ILP model with OPL

The ILP model we describe in Chapter 4 is solved with the commercial ILP solver ILOG OPL Studio 4.2. OPL is a modeling language for mathematical programming and combinatorial optimization problems, and is currently the most wide-spread one in industrial practice.

An OPL project is composed of two types of files, the model file (.mod) and the data file (.dat). The model file is the essential part of the OPL project and is instance-independent. It serves as a general mathematical programming model for solving a specific combinatorial optimization problem, which in our case refers to the MDVSP with coupling trips and (optionally) time windows. The OPL data file provides data that specifies an instance of the problem defined by the associated model file, and it is formatted following a strict rule that is given by the model file. A brief user guide to the OPL language can be found by Martin (2002).

In the solution process of our software *Locomotive Scheduler*, the OPL input data file is generated and placed at the corresponding OPL Studio workspace directory, if correctly configured. After OPL Studio yields a solution, either partial solution or final solution, it will be read into our software, verified, interpreted and transformed into human-readable format and displayed on our user interface.

7.3 Test Environment

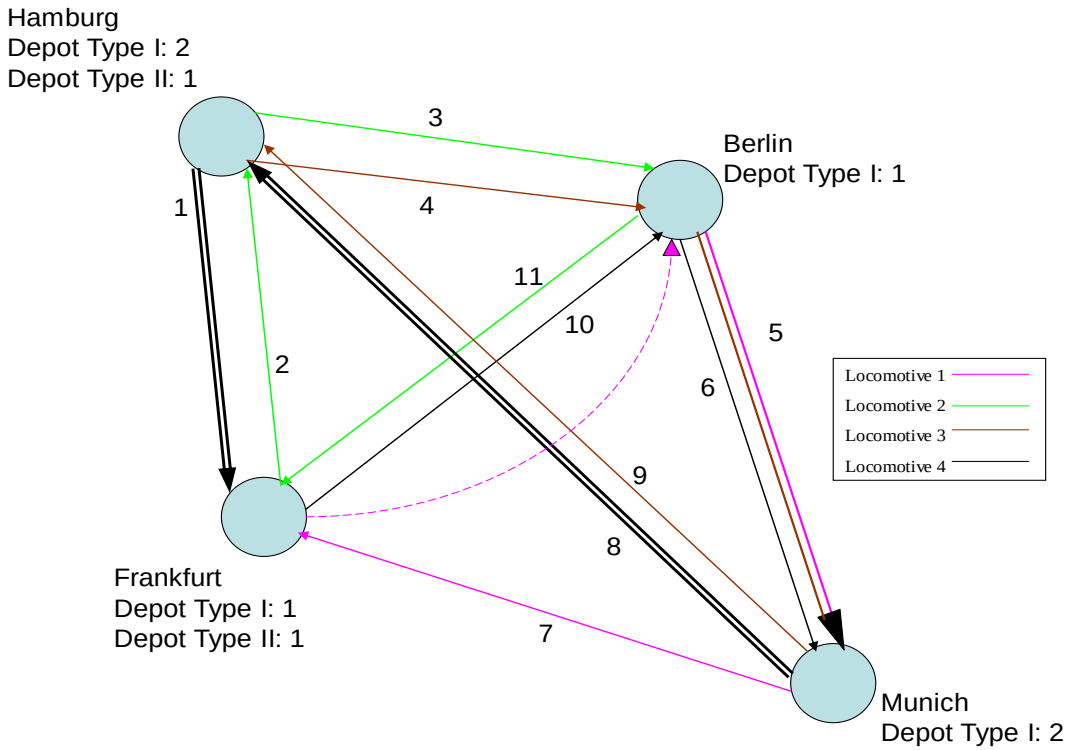
For efficiently solving the ILP problem, we used ILOG OPL Studio 4.2. The software *Locomotive Scheduler* is written in Java programming language, and is compiled and tested using JDK 1.5.0_06, on a personal computer running Windows XP as operating system, with hardware support from AMD AthlonXP 1800+ (1.5GHz) and 256MB DDR RAM.

7.4 Solving the Tiny instance

Here in the following shows a table of locomotive schedule for our hand-made instance “Tiny” in the version of strict depot and fixed timetable. This solution is computed by OPL Studio, and interpreted by our software. For computing the objective value, we assume that starting one locomotive costs 1000 units, and driving one unit time costs 1 unit.

	Shipment index	Origin	Start time	Destination	Arrival time
Locomotive 1 (type 1 , depot location Berlin):					
Shipment	5	Berlin	17	Munich	26
Shipment	7	Munich	32	Frankfurt	39
Deadhead		Frankfurt	39	Berlin	43
Locomotive 2 (type 1 , depot location Frankfurt):					
Shipment	2	Frankfurt	5	Hamburg	14
Shipment	3	Hamburg	15	Berlin	22
Shipment	11	Berlin	34	Frankfurt	42
Locomotive 3 (type 1 , depot location Hamburg):					
Shipment	4	Hamburg	6	Berlin	14
Shipment	5	Berlin	17	Munich	26
Shipment	9	Munich	33	Hamburg	44
Locomotive 4 (type 2 , depot location Hamburg):					
Shipment	1	Hamburg	5	Frankfurt	14
Shipment	10	Frankfurt	14	Berlin	25
Shipment	6	Berlin	25	Munich	35
Shipment	8	Munich	35	Hamburg	48
#Locomotives: 4		Total driving time: 115		Cost: 4115	

Table 7.1: Optimal locomotive schedule for our hand-made instance “Tiny” in the version of exchangeable depot and fixed timetable.



7.5 Solving real-world instances using OPL

In Chapter 6 we have described in depth how our real-world instances KW47 and KW48 are extracted from the vast customer data files. Each instance corresponds to a shipment timetable with a manual scheduled locomotive disposition plan of one calendar week. We first try to generate locomotive schedules for these two instances under three different mathematical models with the help of OPL Studio. The three different mathematical models are introduced in Section 4.2 – 4.4, namely MDVSPCT with fixed timetable and exchangeable depot, MDVSPCT with fixed timetable and strict depot, and MDVSPCT with flexible time windows and exchangeable depot.

7.5.1 Results for MDVSPCT with fixed timetable and exchangeable depot

At first we take the starting time of the shipments to be fixed as given from the customer's timetable, and assume every depot is allowed to exchange locomotives of the same type. The both instances can be solved within a reasonable amount of time to optimality. Assume there are n shipments and t locomotive types, it will lead to an integer linear programming problem with around n^2t many variables and n^2t many constraints, which in our instances KW47 and KW48 amounts to approximately 50000 to 100000 variables and constraints. But the computational effort does not seem to grow as the instance size grows, e.g. the instance KW47 has fewer shipments but requires dramatically longer computation time (about 41 minutes) than KW48 (1 minute). The number of shipments and the number of locomotive types are shown in column 2 and 3, respectively. And the computation time is listed in the 4^{th} column. The actual amount of locomotives of different types that is needed to serve all shipments is shown in column 5, the sum of these is given in column 6. We estimate the cost for the whole locomotive schedule as the sum of locomotive starting cost and the total driving cost. The starting cost per locomotive is estimated as 1000000 units, and driving the locomotive one minute, either when pulling a train or driving empty on a deadhead trip, is amount to 1 unit cost. The total cost of each instance is given in the last column, and the total driving time is shown in column 7 separately.

Instance name	Shipments	Locomotive types	OPL Computation Time (sec)	Locomotives	Σ	Total driving time	Cost
KW47	172	2	2507	8 / 12	20	82879	20082879
KW48	178	2	60	10 / 11	21	86049	21086049

Table 7.2 Optimal solutions and computational details for MDVSPCT with fixed timetable and exchangeable depot

There is one thing worth noticing, although it takes OPL 41 minutes to solve the instance KW47 to optimality, the first solution 2.0085e+007 (see Table 7.3) actually came at about one and half minutes, with near-optimal quality. Note that according to the best LP relaxation node found at that time as 1.9585e+007, we could already conclude that the minimum number of locomotives is already found to be 20. For industrial purpose this solution is already acceptable. The rest of computation time (about 40 more minutes) is devoted to tightening the lower bound to prove optimality.

Elapsed time (sec)	Best integer solution	Best Node	Gap
90	2.0085e+007	1.9585e+007	2.49%
971	2.0083e+007	1.9585e+007	2.48%
2507	2.0083e+007	2.0083e+007	0%

Table 7.3 Cplex logs for acquired partial solutions for KW47 according to elapsed time.

7.5.2 Results for MDVSPCT with fixed timetable and strict depot

In the next experiment we apply the strict depot returning rule instead of the exchangeable depot rule shown in the last section, i.e. now every locomotive must return to its own depot after a week's disposition plan. The mathematical model for MDVSPCT with fixed timetable and strict depot is presented in Section 4.3. Note that this strict depot model is just a special case of the exchangeable depot, which means if we denote $S_{exchange}$ the set of all feasible solutions of the exchangeable depot model, and S_{strict} the set of solutions of the strict depot model, the following relation holds:

$$S_{strict} \subseteq S_{exchange}$$

So as one can expect, in general the optimal solution of the exchangeable depot model will have a slightly better objective value than the strict depot model. Moreover, the

strict depot model will require noticeably longer computation time than the exchangeable depot model. the computational details are listed below:

Instance name	Shipments	depots	OPL Computation Time (hour:minute)	Locomotives	Σ	Total driving time	Cost
KW47	172	13	6h : 30min	9 / 11	20	84363	20084363
KW48	178	11	0h : 23min	11 / 10	21	86479	21086479

Table 7.4 Optimal solutions and computational details for MDVSPCT with fixed timetable and strict depot

Comparing the optimal objective value with the one shown in Table 7.2, they have the same number of locomotives in both strict and exchangeable models, because no time constraints have been imposed on the pull-in trips and pull-out trips, so that different depot returning rules have no influence on the number of locomotives; but the strict depot model requires more driving time than the exchangeable depot model, because the latter can make use of the flexible depot returning rule to further reduce the driving time spent on the pull-out trips.

The noticeable difference between the two models is the computation time. In comparison with Table 7.2, the strict depot model consumes approximately 10 times longer computation time than the exchangeable depot model, to solve the problem to optimality. Especially in the instance KW47, the first solution came after running at least 2 hours. To see how this time difference occurs, we denote k the number of depots (shown in column 3 of Table 7.4), n the number of shipments and t the number of locomotive types as used in the last section. The integer linear programming model of the strict depot version will have about kn^2 variables and tn^2 constraints. Comparing with the exchangeable model, they have more or less the same number of constraints, but different number of variables. Since $t:k \doteq 1:5$, the number of variables is about 5 times more than the exchangeable version, which in our instances KW47 and KW48 amounts to as many as about 200000 to 300000 variables. The OPL computation time increases dramatically as the size of the network expands.

7.5.3 Results for MDVSPCT with flexible time window and exchangeable depot

As is shown above, although the strict depot model causes some computational difficulty, the both previous two models with fixed timetable are able to be solved to

optimality in a reasonable time by OPL Studio. But the introduction of flexible time windows brings a real computational challenge. After running OPL Studio for 24 consecutive hours on the time window model with a modest-size instance KW48, the best solution found still has an about 15% gap with the current best lower bound. The detailed computational results are shown below:

Instance name	Time window size (hours)	OPL Computation Time (hour : min)	Locomotives	Σ	Total driving time	Cost	Gap
KW47	6	6 : 30	5 / 10	15	84713	15084713	19.87%
KW48	6	24 : 00	7 / 8	15	87387	15087387	13.21%

Table 7.5 Computational results for MDVSPCT with flexible time windows and exchangeable depot

An interesting observation is, although the time window version has greatly reduced the number of vehicles (at least 25% to 30% reduction), it does not bring in an expansion in the driving distances. The total driving time is still more or less the same as the fixed timetable version.

In the sequel we take the instance KW48 for example, to see the found solution quality at different computation time milestones.

Elapsed time (hour : min)	Best integer solution	Best Node	Gap
0 : 15	2.1093e+007	1.3089e+007	37.95%
0 : 30	1.8100e+007	1.3089e+007	27.68%
1 : 00	1.8099e+007	1.3093e+007	27.66%
3 : 00	1.7096e+007	1.3093e+007	23.41%
5 : 00	1.6093e+007	1.3094e+007	18.64%
10 : 00	1.5094e+007	1.3094e+007	13.25%
24 : 00	1.5087e+007	1.3094e+007	13.21%

Table 7.6 Cplex logs for acquired partial solutions for KW48 with time windows and exchangeable depot according to elapsed time.

It is normally not practical to leave the machine running for one week in order to compute a weekly schedule. In general, a good and reliable locomotive disposition plan is required to be delivered within one hour's computation. In this case, OPL Studio won't be an ideal choice. We must find other ways to overcome the difficulty. Our effort is mainly devoted to developing a robust and efficient primal heuristic.

7.6 Solving MDVSPTWCT using heuristics

In the section, the computational results obtained by our self-developed heuristics will be presented.

7.6.1 Greedy Construction Search

First we have a look at the results computed by a naïve greedy construction search:

Instance name	Time window size (sec)	Computation Time (sec)	#Locomotives	Total driving time	Cost
Tiny	0	0.01	5	154	5154[1]
KW47	0	0.07	25	125982	25125982
KW48	0	0.08	24	127679	24127679
KW47	360	0.07	19	119820	19119820
KW48	360	0.08	19	119452	19119452

Table 7.7 Computational results for MDVSPTSCT with naïve greedy construction search

The greedy construction search algorithm runs really fast for solving our problem. For the real-world instances KW 47 and KW 48, it takes less than 0.1 second for one run. Unlike solving this problem with mathematical programming, the computation time of the greedy heuristic stays almost the same despite the introduction of time windows. In the sequel we will try to run the greedy algorithm iteratively, in order to find the best parameter combinations, or to discover the best solution under randomized mutation.

7.6.2 Iterative Greedy Construction Search Algorithms for Tiny

In the sequel we will display the results for instance Tiny acquired by 4 different metaheuristics, with their implementation details listed below:

- Semi-Greedy: the construction phase of GRASP (see Hoos, Stützle 2004), with a restricted candidate list of size 3; starting depot totally randomized; 20000 rounds of greedy construction search.
- PGrid: parameters *Double Penalty*, *Duration Weight*, *Index Weight* are applied and rastered into grids to be searched; starting depot totally randomized; 20000

[1] For simplicity, we assume every locomotive costs 1000 units for the instance Tiny.

rounds of greedy construction search.

- RGS: Iterative Randomized Greedy, with perturbation randomly drawn for a uniformly distributed interval [600, 1200]; *Double penalty* is fixed to be 1; starting depot totally randomized; 20000 rounds of greedy construction search.
- PRGS: hybrid of PGrid and RGS, parameterize *Double Penalty*, with perturbation randomly drawn for a uniformly distributed interval [800, 1200]; starting depot totally randomized; 20000 rounds of greedy construction search.

Their results are listed in the table below:

Algorithm name	#Greedy iterations	Computation Time (sec)	#Locomotives	Total driving time	Cost
Semi-greedy	20000	80	4	123	4123
PGrid	20000	80	4	125	4125
RGS	20000	80	4	117	4117
PRGS	20000	80	4	115	4115[2]

Table 7.7 Iterative greedy search algorithms for Tiny

The PRIG algorithm finds the optimal solution for Tiny. The best value for the parameter *Double Penalty* is found to be 1.7.

7.6.3 Iterative Greedy Construction Search Algorithms for Real-World Instances with Fixed Timetable

The 4 iterative greedy construction search algorithms mentioned above are subsequently applied to our real-world instances KW47 and KW48 with fixed timetables. The 4 algorithms all share some properties in common: calling the greedy construction search iteratively; maximize the value of *Capacity Weight* (see equation (5.1), section 5.2.3) in favor of Type II depots in starting depot selection; combine depot location selection together with the first shipment selection; every algorithm has a computation time of less than 5 minutes (300 seconds), for KW47, each algorithm makes 3000 calls of greedy search, and for KW48, 3500 iterations for each algorithm. The other implementation details are listed below:

- Semi-greedy: our experiment shows, as the instance size grows, semi-greedy gives rather poor performance. Its best solution is acquired when the size of the restricted candidate list is set to 1, which is amount to a naive greedy construction search. When the size of the restricted candidate list is set to greater than 1, we cannot even get a feasible solution, since it has simply run out of all our

[2] The proved optimal objective value for Tiny

locomotives[3]. It shows this algorithm has made its search space too extensive, but lack of necessary bias that guides the search process in a more concentrative way.

The following algorithms successfully find good solutions to the problem:

- PGrid: we first raster and search the two most important and "reasonable" heuristic parameters *Double Penalty* and *Geo Weight* in grids, it yields good solutions (22 locomotives forKW47 and 23 for KW48), but not satisfied enough, since our search space is too concentrative and limited, lack of distraction (or variance). So we must bring in some new perturbations. Our first approach is to introduce some "unreasonable" heuristic factors like the time interval size, next shipment duration and next shipment index, therefore 3 more weight parameters are introduced to the linear scoring function. We are doing so only to bring new perturbation into the problem, so that the amount of our search exploration will be extensively enlarged. This procedure has greatly improved the solution quality for such a problem that is lacking in distraction.
- RGS: after PGrid finds a good parameter combination, we intensively exploit the parameter combination, with a perturbation drawn from some probability distribution. We have experimented with uniform distribution and Gaussian distribution (i.e. normal distribution, see section 5.5), at last Gaussian distribution seems to excel others. Our perturbation value is drawn from a Gaussian distribution of $N(1000,100)$, i.e. a normal distribution with mean value 1000 and standard deviation 100.
- PRGS: for the hybrid of PGrid and RGS, we search the parameter space with *Double Penalty* and *Geo Weight* in grids, and meanwhile applied a perturbation value drawn from a Gaussian distribution $N(1000,50)$, and enlarge the number of iterations for each parameter combination. The best parameter combination for KW47 is found to be

$$\lambda_{DP} = 1.4,\ \lambda_{Geo} = 0.9$$

and the best parameter combination for KW48 is found to be:

$$\lambda_{DP} = 1.5,\ \lambda_{Geo} = 0.9$$

The final computational results are listed in the following table in details:

[3] We set an upper bound to the number of locomotives to 27.

Algorithm name	#Greedy iterations	Computation Time (sec)	#Locomotives	Total driving time	Cost	Optimal cost	Root gap
KW47							
Semi-greedy	3000	250	> 27[4]	-	-	20084363	-
PGrid	3000	250	21	106949	21106949	20084363	5.09%
RGS	3000	250	21	106390	21106390	20084363	5.09%
PRGS	3000	250	21	100094	21100094	20084363	5.06%
KW48							
Semi-greedy	3500	300	> 27	-	-	21086479	-
PGrid	3500	300	22	117817	22117817	21086479	4.89%
RGS	3500	300	22	112065	22112065	21086479	4.86%
PRGS	3500	300	22	107054	22107054	21086479	4.84%

Table 7.8 Iterative greedy search algorithms for KW47 and KW48

7.6.4 Iterative Greedy Construction Search Algorithms for Real-World Instances with Time Windows

Recall that the MDVSPCT problem with time windows causes great computational difficulty for our ILP solver OPL Studio. But with our heuristic, a good solution can be delivered within 5 minutes, with comparable quality of the solutions acquired by running OPL Studio for 24 hours. The results of only 3 algorithms are listed, PGrid, RGS and PRGS, Semi-greedy is eliminated because of its bad performance.

- PGrid: the introduction of time windows brings in a lot more possible component combinations, which makes the searchable space much more dense and extensive. In such circumstance we take only reasonable greedy criteria into account, namely temporal distance and geographic distance, and 2 parameters are rastered into grid, the *Double Penalty* and *Geo Weight*.
- RGS: due to the diverse nature of the problem with time windows, we have to narrow our perturbation distribution to a smaller range. The best perturbation is found to be *N(1000, 25)*. Note that our RGS is run based on the best parameter combination found in the former PGrid procedure.
- PRGS: we keep the parameter *Double Penalty* and *Geo Weight* in grid like in PGrid, and add perturbation to our grid search by a multiplier generated from normal distribution *N(1000, 25)*.

[4] We assume there are at most 27 locomotives, any algorithms that exceed this number are considered unreliable.

Algorithm name	Greedy iterations	Computation Time (sec)	Locomotives	Total driving time	Cost	Root gap	OPL best solution	OPL Root gap
KW47								
PGrid	3500	300	15	104057	15104057	19.97%	15084713	19.87%
RGS	3500	300	15	102260	15102260	19.96%	15084713	19.87%
PRGS	3500	300	15	102838	15102838	19.97%	15084713	19.87%
KW48								
PGrid	3500	300	16	101680	16101680	18.68%	15087387	13.21%
RGS	3500	300	15	106845	15106845	13.32%	15087387	13.21%
PRGS	3500	300	15	106653	15106653	13.32%	15087387	13.21%

Table 7.9 Iterative greedy search algorithms for KW47 and KW48

7.7 Comparisons of Results

In Section 7.7.1 we show the potential cost saving after an optimization procedure. Section 7.7.2 shows that an introduction of time windows will bring further noticeable saving. And Section 7.7.3 proves the efficiency of our heuristic algorithm in solving MDVSPTWCT.

7.7.1 The Power of Optimization

Comparing with hand-made locomotive schedules currently being used by our customer, our optimized schedule results show a great potential in saving operational costs.

Due to various commercial security reasons, we are not yet able to access the complete locomotive disposition plan, and therefore the total driving distance of the whole customer manual schedule is unknown yet. However, the number of deployed locomotives is clearly given. As is shown in the manual disposition plan, currently the instance KW47 needs in total 26 locomotives, and KW48 needs 27. Note that normally the total operational cost of a transportation plan is proportional to the number of vehicles in use. Besides, as observed in section 7.5.3, the driving distances differ only slightly, when the number of deployed vehicles changes.

Experiment shows that our optimization results with the fixed timetable model bring in at least 20% saving in operational cost comparing the manual schedules.

Instance	#vehicles before optimization	#vehicles after optimization[5]	#saved vehicles	Saving in percentage (%)
KW47	26	20	6	23.1%
KW48	27	21	6	22.2%

Table 7.10 Result comparison before and after optimization for KW47 and KW48

7.7.2 The Power of Time Window

The introduction of a reasonable time window, in our case we allow each shipment to have 6 hours flexibility for its arrival time during a week's plan, brings a great reduction in operational costs.

Unfortunately the optimal results of MDVSPCT with time windows for instances KW47 and KW48 are not found yet, as mentioned in section 7.5.3 and 7.6.4, so we use the currently best known computational results instead.

As is shown below, despite that the solution of MDVSPCTTW is still not yet optimal, the current results already show an over 25% cost saving in comparison with the optimal results with fixed timetable, and an over 40% saving comparing with the original manual schedule.

Instance	#vehicles before optimization	#vehicles after optimization without time window[5]	#vehicles after optimization with time window[6]	Saving comparing with manual result	Saving comparing with optimized result with fixed timetable
KW47	26	20	15	42.3%	25.0%
KW48	27	21	15	44.4%	28.6%

Table 7.11 Result comparison before and after introduction of time windows for KW47 and KW48

7.7.3 The Power of Heuristics

Experiment shows, in solving MDVSPCT problem, our heuristic method PRGS is competitive with the exact method under the support of the commercial ILP solver OPL Studio.

In the fixed timetable model, with the help of OPL Studio, we are able to solve the given problem instances KW47 and KW48 to optimality from a few minutes to a few hours in the strict depot returning version. Our heuristic algorithm solves the problem within 5 minutes with around 5% gap to optimality.

[5] The optimal result of the MDVSPCT model with fixed timetable, see section 7.5.1 and 7.5.2
[6] The currently best known optimization results of MDVSPCT model with time windows, see section 7.5.3.

In the flexible time window case KW47+TW and KW48+TW, OPL Studio encounters difficulties. For a normal-size instance, a reasonably good solution only comes up after keeping the OPL running for several hours. A fine-tuned hybrid of PGrid and RGS usually solves the problem within 5 minutes with a solution at least as good as the one returned by OPL Studio after running 24 hours.

Instance	Objective value by heuristics	Computation time for heuristic (hour : min)	Objective value by ILP Solver	Computation time for ILP Solver (hour : min)	Solution gap between heuristics and ILP Solver
KW47	21100094	0:05	20084363	6 : 30	5.06%
KW48	22107054	0:05	21086479	0 : 23	4.84%
KW47+TW	15102260	0:05	15084713	6 : 30	0.12%
KW48+TW	15106653	0:05	15087387	24 : 00	0.13%

Table 7.12 Result comparison between OPL and heuristic in computation time and solution quality for KW47 and KW48

Bibliography

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin (1993), Network Flows: Theory, Algorithms, and Applications. Prentice Hall, New Jersey.

[2] A. Schrijver (1986), Theory of Linear and Integer Programming. Wiley Interscience, John Wiley & Sons Inc.

[3] G. Nemhauser, L. Wolsey (1999), Integer and Combinatorial Optimization. Wiley Interscience, John Wiley & Sons Inc.

[4] A. Löbel (1997), Optimal Vehicle Scheduling in Public Transit. PhD Thesis. Shaker Verlag. Online available at: ftp://ftp.zib.de/pub/zib-publications/books/Loebel.disser.ps

[5] A. Fügenschuh, H. Homfeld, A. Huck, A. Martin, Z. Yuan (2006): Locomotive and Wagon Scheduling in Freight Transport. Submitted to Transportation Science.

[6] P. Toth, D. Vigo (2002), The Vehicle Routing Problem. SIAM Monographs on Discrete Mathematics and Applications, SIAM.

[7] H. Hoos, T. Stützle (2004), Stochastic Local Search - Foundations and Applications. Morgen Kaufman.

[8] J.P. Hart and A.W. Shogan (1987), Semi-greedy Heuristics: An Empirical Study. Operations Research Letters, 6:107–114, 1987.

[9] T. A. Feo and M.G. C. Resende (1989), A probabilistic heuristic for a computationally difficult set covering problem. Operations Research Letters, 8:67–71, 1989.

[10] T.A. Feo and M.G. C. Resende (1995), Greedy Randomized Adaptive Search Procedures. J. of Global Optimization, 6:109–133, 1995.

[11] A. Fügenschuh (2005), The Integrated Optimization of School Starting Times and Public Transport. PhD Thesis. Logos Verlag Berlin.

[12] A. Fügenschuh (2005), Parametrized Greedy Heuristics in Theory and Practice. Volume 3636/2005 of Lecture Notes in Computer Science, pages 21-31, Springer Berlin.

[13] M. Birattari, T. Stützle, L. Paquete, K. Varrentrapp (2002), A Racing Algorithm For Configuring Metaheuristics. In W. B. Langdon, E. Cantu-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 11--18, Morgan Kaufmann Publishers, San Francisco, CA, USA, 2002.

[14] G.A.P. Kindervater, M.W.P. Savelsbergh (1997), Vehicle Routing: Handling Edge Exchanges. E.H.L. Aarts, J.K. Lenstra (eds.). Local Search in Combinatorial Optimization, Wiley, Chichester, 337-360.

[15] R. Ruiz and T. Stützle (2005), A Simple and Effective Iterated Greedy Algorithm for the Flowshop Scheduling Problem. European Journal of Operational

Research, 177(3):2033-2049, 2007.
[16] M. Dorigo and T. Stützle (2004), Ant Colony Optimization, MIT Press, Cambridge, MA, USA.
[17] R. Martin (2002), A Short Introduction to OPL - Optimization Programming Language. Online available under: http://zuseex.algo.informatik.tu-darmstadt.de/lehre/2002ws/algomod/AlgMod_OPL.pdf
[18] Wikipedia. Online available at URL http://www.wikipedia.org