Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Bc. Michal Tuláček

Algorithms for automated logistics

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: prof. RNDr. Roman Barták, Ph.D.
Study programme: Computer science
Specialization: Theoretical Computer Science

Prague 2014

I wish to thank my supervisor, prof. Roman Barták, for his invaluable advice and help. I am grateful for the experimental data provided by the Sunnysoft company. Especially, my thanks go to Ing. David Šilhan and Ing. Martin Strouhal for their support and possibility to test the system in a real environment. I wish to thank my friend, a Googler RNDr. Ondřej Sýkora, for hints and advice with Google OR-tools. My greatest thank-you belongs to you, my sweet Martina. Without your help, support and understanding, I would never have finished this thesis.

Název práce: Algoritmy pro automatizovanou logistiku

Autor: Michal Tuláček

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: prof. RNDr. Roman Barták, Ph.D.

Abstrakt: Práce řeší existující problém optimálního plánování přeskladnění zboží mezi pobočkami maloobchodní společnosti. Cílem je navrhnout systém, který na základě objednávek od zákazníků a současných skladových dostupností zboží bude schopen navrhnout optimální plán. Na základě podrobné analýzy problému je v práci formalizován problém automatické logistiky. Po stručném přehledu existujících přístupů v oblasti řešení logistických problémů jsou pak navrženy metody řešení založené na programování s omezujícími podmínkami a smíšeného celočíselního programování. Obě metody jsou experimentálně navzájem porovnány, a to jak s ohledem na kvalitu nalezeného řešení, tak s ohledem na jejich výkonnost.

Klíčová slova: automatická logistika, plánování, optimalizace

Title: Algorithms for automated logistics

Author: Michal Tuláček

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: prof. RNDr. Roman Barták, Ph.D.

Abstract: This thesis deals with a real world problem of an optimal planning of transferring goods between stores of a certain retail company. The aim is to propose a system which, based on customer orders and current stock items availability, should be capable of devising an optimal plan. In this thesis, following a thorough analysis of the problem, there is a formalised problem of automated logistics. After a brief overview of existing approaches in the area of logistics problem solving, there are subsequently designed solution methods based on programming with limited conditions and mixed integer programming. Both methods are compared to each other experimentally, by taking into account the quality of the found solution and the methods' efficiency.

Keywords: automated, logistics, planning, optimization

Contents

Chapter 1

Introduction

In this thesis, we examine a real world problem of a retail company Sunnysoft. The company operates several stores around the Czech Republic together with an electronic shop. The customers can either order goods in the electronic shop to be delivered by an external carrier company (Czech Post, DPD etc.) or they can reserve the goods in one of the shops. However, the goods is distributed all around stores. Therefore, the company needs an efficient way to transfer the goods from a store where the goods is available to a store where the goods is demanded. The company itself does not handle the transfers. Instead, it uses the services of external delivery companies. Therefore, in practical case, the company is not limited by the capacity of transfer routes (the delivery company has to only be notified that, exceptionally, it has to use a truck instead of a regular van). However, a store can be overloaded if too many transfers handled in the store are planned.

Based on the analysis, we propose a formal description of the examined problem, an *Automated logistics problem* (ALP) and its variant *Automated logistics problem with low-priority demands* (ALP-lp) which prevents the overload of the stores.

For review, there are several approaches currently used in logistics, which could be used to solve ALP or ALP-lp. In particular, we present approaches based on a Vehicle routing problem, network flows, constraint programming and mixed integer programming.

To resolve Sunnysoft's problem, we propose three different techniques. In the first one, we relate the problem to a problem of finding a maximum flow through a network and propose an algorithm based on Ford-Fulkerson algorithm. In the second one, we formulate the problem as a constraint satisfaction problem. Finally, in the third technique, we formulate the problem as a mixed integer programming problem. We use state of the art solvers to solve the constraint satisfaction based problem and the mixed integer programming problem.

We perform benchmarking of the proposed algorithms. We perform a qualitative analysis of the solvers and compare the generated plans. We compare the methods to a naïve algorihm, as well. We focus on the overall duration of the transfers and on the amount of transfers through the stores. Each demand should be resolved as quickly as possible, because there are customers waiting for the goods. If the demands are resolved too late, the customers can either cancel the order and order the goods elsewhere or they will be displeased and will not buy the goods from the company in the future. The amount of transfers through the store should not exceed a particular limit. Otherwise, the store can overload and cease functioning.

Simultaneously, we perform a performance analysis and compare the techniques based on their running time while varying their parameters.

First, we examine the given problem in detail in chapter 2. Afterwards, we formalize an Automated logistics problem and its variant. In chapter 3 we review several models used in automated logistics and evaluate their ability to solve the problems. Next in chapter 4, we state algorithms which solve the defined problems. To evaluate the proposed models, we construct a simulation system and perform experiments on the proposed algorithms. Finally, the benchmarks and their results are described and discussed in chapter 5.

For a better reading experience, in the end of the thesis, there is a list of used abbreviation. The appendices contain more detailed information about the simulation system and about the techniques used to state the models for the solvers.

Chapter 2

Problem analysis

In this chapter, we discuss the problem outlined in the previous chapter. Our goal is to describe the process in such a way in order to be able to formulate a formalization out of it. The analysis is based on a real situation in the retail store company called Sunnysoft, which uses the ERP[1] software Helios.

2.1 Description of a problem

For a retail store company, it is important to have the goods distributed evenly across its stores. This requirement is met easily when a particular item is sold in large amounts. In this case, the item is usually available in high quantities and can be distributed among the company's stores. However, when the item gets sold rarely (or the distributor cannot deliver it in desired amounts), its stock supply cannot cover all of the stores sufficiently. In such a case, the goods should be stocked in a store where it gets sold the most. Simultaneously, the store should allow a quick relocation of the goods to another store if needed.

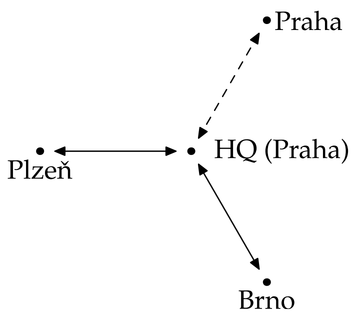

Figure 2.1.1. A situation in the retail store company Sunnysoft. The points represent stores and the lines show possible routes of the goods between the stores. The dashed line represents a route operated by a messenger and normal lines represent a route operated by a regular delivery company.

The stores do not need to be fully interconnected. As we can see on figure 2.1.1, there are no links between Brno and Plzeň. The company uses external delivery company which offers a special price for an often used delivery route. As there is a lower transfer rate between peripheral branches, it is possible to send the goods through other branches (in this particular case, through headquarters in Prague). Moreover, some branches which are close to each other

[1] Enterprise resource planning is a business management software – usually a suite of integrated applications – that a company can use to store and manage data from every stage of its business.

can be used by a messenger travelling between the branches more often (in our case it is between the branches located in Prague).

2.1.1 Process driven by documents

Processes related to goods handling are controlled by documents stored in the ERP system. There are several types of these documents. The relevant ones for the problem are the following: a *shipment order*, a *picking list*, and a *warehouse receipt*. The shipment order is a document ordering the store to deliver an item from a warehouse. The picking list is a document which proves that the item was delivered from the warehouse. Finally, the warehouse receipt is a document which proves that the item was put to the store. Documents can be related to one another. Typically, there is a following sequence *order → shipment order → picking list → invoice* in case the order is made by a customer and a different sequence *shipment order in store 1 → picking list in store 1 → warehouse receipt in store 2* in case the goods are transferred from one store to another. The picking lists and the warehouse receipts can be either unrealised or realised. The document is in effect (and is unchangeable) after its realisation. Such a document cannot be cancelled or deleted. If there is a need to revert the document, an opposite document has to be created and realised. For each item and each store, the ERP system stores a number representing the amount of pieces in the store. Let us call it *InStock*. The *InStock* gets updated by the realised picking lists and the realised warehouse receipts. Schematically, it can be expressed as follows:

$$InStock_{g,s} = \sum_{wr \in rWR_s} (\text{\# of items of } g \text{ on } wr) - \sum_{pl \in rPL_s} (\text{\# of items of } g \text{ on } pl),$$

where g is a particular commodity, s is a particular store, rWR_s is a set of realised warehouse receipts in store s and rPL_s is a set of realised picking lists in store s. The store operators are responsible for any differences created in the amounts of goods between *InStock* and the real amount. Therefore, we may assume that this number is valid and there is no need to prepare a backup plan in regards to a situation where there is a missing item.

As we discussed in the previous paragraph, the ERP contains an information about the amount of stocked items in the store. However, this number is not useable in most situations, for the goods can be unavailable even when it is physically present in the store. Some of the articles can be reserved for a customer, others could be planned for transfer to another store. Such a reservation is performed with shipment orders. We define a variable *Availability* as follows:

$$Availability_{g,s} = InStock_{g,s} - \sum_{so \in SO_s} (\text{\# of unhandled items of } g \text{ on } so),$$

where g and s have the same meaning as mentioned above and SO_s is a set of shipment orders in store s. As we can imagine, the availability can be negative, i.e. the demand for an article might be larger than the amount in that particular store and therefore, the article needs to be transferred from another store or ordered from the supplier.

This model has one disadvantage, which can lead to the irregularity described in section 2.2.1. Since the ERP system only stores the amount of goods in a store, the amount of available goods and the list of shipment orders, there is no clear relation between the reserved item and the shipment order. Sometimes, the shipment orders cannot be processed in the order of their creation. They could contain another article, which is not available. Therefore, the shipment orders are processed out of their order. Fortunately, there is a way to estimate whether we are able to at least partially process the shipment order – we must order the unhandled shipment orders and check, if there is at most ($\mathit{InStock} - 1$) items on the previous shipment orders.[2]

2.1.2 Balancing of goods and low-priority transfers

As we described in section 2.1.1, we can detect the missing goods in order to plan their transfer from another store. However, such a transfer is very often planned with small amounts of transferred goods. Therefore, we need some heuristics to estimate the need for a particular article in a given store and plan the transfer accordingly to this estimation. However, these transfers are different from transfers dealing with negative *Availability* (we call them "covering transfers", since they cover the missing items). The covering transfers must be processed as soon as possible for they indicate that there is a customer waiting for that particular item. The other transfers can be postponed (especially in the case of a store outage described in section 2.2.2) and are handled after the covering transfers are processed. Thus, we introduce a label "low-priority transfer" for a transfer, which does not necessarily need to be processed immediately.

2.1.3 The goods "on the way"

As described in section 2.1.1, the amount of goods in a particular store can be described using the variables *InStock* and *Availability*. Unfortunately, this does not cover all the situations. When an article is sent to another store, it is delivered from a default store but not stocked to its destination store, since there is a realised picking list, but no corresponding realised warehouse receipt. Such goods is marked for the ERP system as "in a void". Nevertheless, we can detect such goods by inspecting the unrealised warehouse receipts. If such a receipt is found and has a corresponding realised picking list, it means that there is goods on the way.

[2] In the real application the ERP has an extension column, where it is possible to find this characteristic – "can be processed", "can be processed partially" and "cannot be processed" – precomputed with the database triggers. The operators have the ERP preconfigured to show only the shipment orders, which can be at least partially processed.

It is important to keep track of the goods on the way, for the store can have a negative *Availability* but there might be a transfer planned, which could have been already handled by a certain previous action. Therefore we can introduce a variable *OnTheWay* with following definition:

$$OnTheWay_{g,s} = \sum_{wr \in utWR_s} (\text{\# of items of } g \text{ on } wr),$$

where the g and s have the same meaning as above and $utWR_s$ is a set of unrealised warehouse receipts in a store s with corresponding realised picking lists in some of the other stores.

It is useful to count with the planned transfer shipment orders. The amount of the goods in such orders is added to the *OnTheWay* variable. Therefore, we define a *OnTheWay** as follows:

$$OnTheWay^*_{g,s} = OnTheWay_{g,s} + \sum_{so \in tSO_s} (\text{\# of unhandled items of } g \text{ on } so),$$

where the g and s have the same meaning as above and tSO_s is a set of transfer shipment orders (i.e. shipment orders used to plan transfers between two particular stores) from other stores to store s. We exclude the transfer shipment orders for the low priority transfers described in section 2.1.2. Since these orders are low-priority, there is no guarantee, that the goods will be dispatched in a reasonable period of time.[3]

With this variable, we can slightly modify the previously defined *Availability* and define a variable *Availability**, which contains the current availability of a particular article in a store with an increase expected in a few days:

$$Availability^*_{g,s} = Availability_{g,s} + OnTheWay^*_{g,s}$$

There is no need to transfer the goods to the store, if the *Availability** is non-negative. However, we cannot use this variable while planning a transfer to another store, because the goods is not present physically.

2.1.4 Pick-up and delivery of items

The goods planned for a transfer are periodically sent to its destination branch. Although, this action cannot be performed continuously as the delivery company stops for a pick-up of the goods only once in a while. The transfers are therefore performed in batches that are planned on regular bases, which gives us the possibility of creating a schedule out of them. Moreover, for each pair of

[3] In a real situation, we experienced that some stores stopped handling the low-priority transfers at all, since there was an outage of workers. However, the system expected the transfers to be processed so it did not plan any new transfers (not even the high-priority ones) for the affected goods.

stores together with a delivery type, we have an estimated duration (usually in working days) of the delivery together with the delivery time for the destination store. Ergo, we can compute the shortest path (in time units) for the goods from all stores to the store where the goods is missing (i.e. $Availability^* < 0$).

2.2 Irregularities

The previously described process has some irregularities in the real world, which make the problem harder to solve. The two most problematic scenarios are – queue jumping and storehouse outage. These irregularities can compromise the whole process and therefore we will pay more attention to them.

2.2.1 Queue jumping

When an item is ordered by a customer or when it is planned for a relocation, it is blocked by a shipment order and is no longer available. However, sometimes a salesman does not respect the blocking and sells the goods despite its being blocked. The reason for such an action can vary and in fact, is not important for our purposes, but the outcome is the same. Either there is a customer whose order was not processed in time or there is a planned transfer of goods which cannot be processed because the goods is no longer available in that store.

This is the result of a combined system where the inputs are generated either by computers or by humans. Since it cannot be avoided, the system should be able to handle such situations. If there is a planned low-priority transfer as described in section 2.1.2, the transfer can be cancelled and the item that was to be transferred can replace the missing goods. If there is no such goods available, the system should plan a transfer from another store, where the item is available.

2.2.2 Storehouse outage

Stores cannot be always open and operational, since they are tended to by human operators. A state, when the store is not open nor operational or even overloaded, is called a store outage. There are several recurring outages – weekends and state holidays. Sometimes there is an outage caused by an absence of the operator (leave or illness) and sometimes there is another reason for the outage such as a stock take. Generally, there are two types of outages – 1) closed store, 2) overloaded store.

In the first case, the store cannot receive nor send any goods to another branch. Therefore, it does not make sense to send there any items or plan transfers from that store. We can take into an account a planned outage, but with an unplanned outage we have to perform a certain action, for example cancel the planned transfers and plan the transfer from another store.

In the latter case, the store can receive and send goods marked for high-priority transfers (i.e. the transfers dealing with the negative *Availability* on a store), but will most probably not process the low-priority transfers. This issue is addressed by a variant of the main problem, as described in section 2.3.1.

2.3 Automated logistics problem formulation

We now formulate an *Automated logistics problem* (ALP), which is to be solved in this thesis.

The problem is to generate an optimal plan of transfers of goods which is demanded. The input data for the problem are generated from the ERP system and the output is written back to the ERP system, to be processed by the store operators. In the real system, the shipment orders can contain demands for more types of goods. However, for the simplicity of the model, we consider only demands with just one type of goods. The shipment orders containing more types of goods are represented by more demands, one for each type of goods.

We represent the time as an integer indicating the amount of elapsed time since the arbitrary time moment t_0 (e.g. the amount of seconds from the beginning of a Unix epoch, 1st January 1970 00:00 AM).

The problem uses a precomputed set of delivery paths. Each path contains a time of delivery, in case the goods is scheduled to be transferred along the path.

The sum of amounts of demands can exceed the available amount of goods in the stores. Therefore, not all of the demands can be processed. However, the system must plan transfers for as many goods as possible. Simultaneously, the order of precedence of demands in the store must be respected, i.e. the demand (which contains just one type of goods, as described earlier) can be resolved only if all older demands containing the same type of goods in the store were fully resolved.

The ALP is an on-line problem. It receives all unresolved demands for goods as an input and generates a plan, which is assumed to be immediately processed. If there remains any demand unresolved, it will be contained in the set of demands in the next run of the algorithm. When the demand is processed only partially, its *amount* is lowered by the scheduled amount. Finally, after the demand is resolved completely, it will not be included in the set of demands in the next runs of the algorithm.

The ALP should minimize the overall duration of the generated plan, i.e. sum of the durations of all planned transfers. The duration is defined as $duration = t_{end} - t_{start}$, where t_{start} is a time of shipment from the original store and t_{end} is a time of delivery to the destination store. However, since t_{start} is the same for all transfers, we can simply ignore it and use a sum of t_{end} of each generated transfer instead of the sum of durations.

The ALP is defined as follows. Given a 5-tuple ($\mathbf{G}, \mathbf{S}, \mathbf{P}, \mathbf{D}, \text{InStock}$), where:

- $\mathbf{S}$ is a set of stores.

- $\mathbf{G}$ is a set of goods.
- $\mathbf{P}$ is a set of possible paths between the stores. A path is a triple (s_o, s_d, t_d), where s_o is the original store, s_d is the destination store and t_d is the time of delivery to s_d, if the goods is sent along the path. We use the following notation: origin(p) means a parameter s_o of a path p, destination(p) means a parameter s_d and time(p) means a parameter t_d
- $\mathbf{D}$ is a set of demands. A demand is a 4-tuple $(s, g, amount, t)$, where $s \in \mathbf{S}$ is a store where is the demand placed, $g \in \mathbf{G}$ is goods demanded by the demand, an *amount* is the demanded amount of the goods and t is a time when the demand was placed. We use the following notation: store(d) means a parameter s of a demand d, goods(d) means a parameter g, amount(d) means a parameter *amount* and time(t) means a parameter t. The notation $d \prec d'$ means that time(d) < time(d').
- InStock: $\mathbf{G} \times \mathbf{S} \to \mathbb{N}$, a function returning the amount of items of goods, which is present in the store and is not scheduled for transfer.

Find an optimal schedule $\mathbf{Pl}$, which is a set of plans. A plan is a triple (p, d, *amount*), where $p \in \mathbf{P}$ is a path, along which the goods is sent, *amount* is the amount of goods to be transferred and $d \in \mathbf{D}$ is the demand which is solved by the plan. We use the following notation: path(π) means a parameter p of a plan π, demand(π) means a parameter d, amount(π) means a parameter *amount*, *goods*(π) means *goods*(*demand*(π)), origin(π) means origin(path(π)), destination(π) means destination(path(π)), and time(π) means time(path(π)).

The optimality of the schedule $\mathbf{Pl}$ is measured according to the objective function:

$$\min_{\mathbf{Pl}} \sum_{\pi \in \mathbf{Pl}} \mathrm{time}(\pi) \tag{2.3.1}$$

The schedule $\mathbf{Pl}$ must be compatible with the following constraints:

1) All possible transfers must be scheduled.

$$\forall g \in \mathbf{G}\colon \sum_{\substack{\pi \in \mathbf{Pl} \\ \mathrm{goods}(\pi)=g}} \mathrm{amount}(\pi) = \min \left\{ \sum_{s \in \mathbf{S}} \mathrm{InStock}(g, s);\ \sum_{\substack{d \in \mathbf{D} \\ \mathrm{goods}(d)=g}} \mathrm{amount}(d) \right\} \tag{2.3.2}$$

2) The sum of planned transfers cannot exceed the amount of goods in the store.

$$\forall g \in \mathbf{G}, \forall s \in \mathbf{S}\colon \sum_{\substack{\pi \in \mathbf{Pl} \\ \mathrm{goods}(\pi)=g \\ \mathrm{origin}(\pi)=s}} \mathrm{amount}(\pi) \le \mathrm{InStock}(g, s) \tag{2.3.3}$$

3) The sum of planned transfers cannot exceed the demanded amount for the particular demand.

$$\forall d \in \mathbf{D}\colon \sum_{\substack{\pi \in \mathbf{PI} \\ \text{demand}(\pi)=d}} \text{amount}(\pi) \leq \text{amount}(d) \tag{2.3.4}$$

4) All demands in a particular store must be processed in their order.

$$\forall g \in \mathbf{G}, \forall \pi \in \mathbf{PI}\colon \text{demand}(\pi) = d \Rightarrow \forall d' \prec d \wedge \text{store}(d) = \text{store}(d') \wedge$$
$$\wedge \text{goods}(d) = \text{goods}(d')\colon \text{amount}(d') = \sum_{\substack{\pi' \in \mathbf{PI} \\ \text{demand}(\pi')=d'}} \text{amount}(\pi') \tag{2.3.5}$$

2.3.1 ALP with low-priority demands

As discussed in section 2.1.2, we can place low-priority demands. These demands can be ignored, if the store becomes overloaded. To model this situation, we introduce an *Automated logistics problem with low-priority demands* (ALP-lp) as a variation to ALP. We describe only differences to the original problem.

ALP-lp is a 10-tuple ($\mathbf{G}, \mathbf{S}, \mathbf{P}, \mathbf{D}, \mathbf{T}$, InStock, Capacity, Priority, Day, Load), where

- $\mathbf{G}$, $\mathbf{S}$, $\mathbf{P}$, $\mathbf{D}$ and InStock have the same meaning as in ALP.
- $\mathbf{T}$ is a set of identifiers of days, when the goods is processed in some store.
- Capacity: $\mathbf{S} \rightarrow \mathbb{N}$ is a function which returns a capacity of a given store, i.e. the maximum daily amount of the items of goods processed by the store.
- Priority: $\mathbf{D} \rightarrow \{\text{high}, \text{low}\}$ is a function which returns a priority of the given demand.
- Day: $\mathbf{P} \times \mathbf{S} \rightarrow \mathcal{P}(\mathbf{T})$ is a function which for a given path and store returns identifiera of the days, when the goods is processed in the store, if sent along the path.
- Load: $\mathbf{S} \times \mathbf{T} \rightarrow \mathbb{N}_0$ is a function which for a given store and identifier of a day returns an amount of goods, which is already planned to be processed in the store during the day.

The objective function differs as we need to fulfil two goals, which are in contradiction. The task is to find the shortest schedule, just like in the case of ALP. However, the simple way of achieving that state is to ignore all low-priority demands. Therefore ALP-lp is a multiple-criteria optimization problem.

$$\min_{\mathbf{PI}} \sum_{\pi \in \mathbf{PI}} \text{time}(\pi) \tag{2.3.6}$$

$$\max_{\mathbf{Pl}} \quad \sum_{\pi \in \mathbf{Pl}} \mathrm{amount}(\pi) \tag{2.3.7}$$

The generated schedule **Pl** must be compatible with the following constraints:

1) The constraints (2.3.3) and (2.3.4) must be satisfied.
2) The order of high-priority orders in the store must be maintained.

$$\forall \pi \in \mathbf{Pl}\colon \mathrm{demand}(\pi) = d \Rightarrow \forall d' \prec d \wedge \mathrm{store}(d) = \mathrm{store}(d') \wedge$$
$$\wedge goods(d) = goods(d') \wedge Priority(d') = \mathrm{high}\colon \mathrm{amount}(d') = \sum_{\substack{\pi \in \mathbf{Pl} \\ \mathrm{demand}(\pi) = d'}} \mathrm{amount}(\pi) \tag{2.3.8}$$

3) The capacity of a store cannot be exceeded if possible. Therefore, each store should not process more items of goods per day than is in its capacity. Otherwise, the store cannot process low-priority demands.

$$\forall s \in \mathbf{S}, \forall t \in \mathbf{T} : Load(s,t) + \sum_{\substack{\pi \in \mathbf{Pl} \\ t \in \mathrm{Day}(path(\pi), s)}} \mathrm{amount}(d) \geq Capacity(s) \Rightarrow$$
$$\Rightarrow \nexists \pi \in \mathbf{Pl} : t \in Day(\mathrm{path}(\pi), s) \wedge \mathrm{Priority}(\mathrm{demand}(\pi)) = \mathrm{low} \tag{2.3.9}$$

Chapter 3

Related research

In this chapter, we review useful models and techniques which help us formulate a model for the problem described in chapter 2.

3.1 Transportation problems

In this section, we present a group of problems called *transportation problems*. What these problems have in common is that they solve a situation, when we need to transport items from one place to another by one or more vehicles. Using the transportation problems, we can model a classical blocks-world problem [35], as well as real applications like planning the schedules of aircraft flights, a plan for a robot etc.

In the problem formulation, there are three different kinds of entities. At first, there are locations, which are immobile. The second kind of entities are portables, which represent things which can be moved from one place to another. Finally, the third kind of entities are mobiles, which can move from one location to another. The mobiles can load and unload portables and transport them from one place to another. In [23], there is described a *transport task*, which combines locations, portables and movables and contains constrains such as limited fuel or the capacity of the mobiles. The goal of the transportation problems is to solve the transport task.

A transport task is a 9-tuple $(\mathbf{V}, \mathbf{M}, \mathbf{P}, l_0, \mathbf{P}_G, l_G, \mathit{fuel}_0, \mathit{cap}, \mathit{road})$, where

- $\mathbf{V}$ is a finite set of *locations*,
- $\mathbf{M}$ is a finite set of *mobiles*,
- $\mathbf{P}$ is a finite set of *portables*,
- $l_0 : (\mathbf{M} \cup \mathbf{P}) \to \mathbf{V}$ is the *initial location* function,
- $\mathbf{P}_G \subseteq \mathbf{P}$ is the set of *goal portables*,
- $l_G : \mathbf{P}_G \to \mathbf{V}$ is the *goal location* function,
- $\mathit{fuel}_0 : \mathbf{V} \to \mathbb{N} \cup \{\infty\}$ is the *initial fuel* function,
- $\mathit{cap} : \mathbf{M} \to \mathbb{N}$ is the *capacity* function, and finally
- $\mathit{road} : \mathbf{M} \to \mathcal{P}(V \times V)$ is the *roadmap* function.

We require that $V, M,$ and P are disjoint, and that $(V, \mathit{road}(m))$ is an undirected graph for all $m \in M$ (i.e., for all $m \in M$, the relation $\mathit{road}(m)$ is symmetric and irreflexive).

3.1.1 Vehicle routing problem

In logistics, there are widely used algorithms which can be classified as transportation problems. A large group of them is a variation to the Vehicle routing problem (VRP). The problem was first defined as a Capacitated vehicle routing problem (cVRP) in [12]. The problem deals with a situation where there is a need to plan routes of vehicles with a limited capacity from a central depot to destination depots. This problem is well researched and there exists many modifications to it.

In [8], there is defined a Vehicle routing problem with time windows (VRPTW). The key difference to cVRP is that the depots can be serviced within a specified time interval or time window. This variation of VRP is widely used and well researched and there are available many heuristics to solve it.

Another variation of VRP is a Stochastic vehicle routing problem (SVRP). As presented in [18], stochastic vehicle routing problems arise whenever some elements of the problem are random, for example stochastic demands or stochastic travel times. The paper describes an approach using stochastic programming. Such a stochastic program is modelled in two stages – the first is "a priori" solution and the second is a "corrective action". A stochastic program is usually modelled either as a *chance constrained program* (CCP), where the solution is searched with respect to a particular threshold for a probability of a failure, or as a *stochastic program with recourse* (SPR) minimising the expected cost of the second stage solution plus the expected net cost of a recourse. The SVRPs are usually modelled as mixed or pure integer stochastic programs or as Markov decision processes.

The most studied problem in this class is the *Vehicle Routing Problem with Stochastic Demands* (VRPSD). In this problem, customer demands are independent random variables. The next is the *Vehicle Routing Problem with Stochastic Customers* (VRPSC) with customers who are present with some probability but have deterministic demands. A combination of both previously mentioned problems is the Vehicle Routing Problem with Stochastic Customers and Demands (VRPSCD). This problem is extremely hard to solve.

3.1.2 Scheduling

In [7] the VRP is reformulated as a scheduling problem. However, these reformulations do not have the same performance. As described in the paper, in an optimisation task, the VRP is better in minimising the total travelled distance whereas the scheduling technology is better in finding the quickest solution. In the existence task was the VRP itself significantly worse than the scheduling technology. Though, the VRP technology can be used to improve results of the scheduling technology.

A different approach represents the DARSP described in [14]. The problem is to generate a schedule for a heterogeneous aircraft fleet covering a set of operational flight legs with known departure time windows, durations and profits according to the aircraft type. In the cited paper the problem is defined as follows: *"Given a heterogeneous aircraft fleet, a set of operational flight legs over a one-day horizon, departure time windows, durations and costs / revenues according to the aircraft type for each flight leg, find a fleet schedule that maximises profits and satisfies certain additional constraints."*

In the paper, there are two formulations of the DARSP. The first is based as a Set Partitioning with additional constraints and the second as the time constrained multi-commodity network flow formulation. For each of the formulations, the paper proposes a solution strategy. For the Set Partitioning approach, the branch-and-bound strategy is feasible and for the multi-commodity network flow the Dantzing-Wolfe or Lagrangean relaxation embedded in a branch-and-bound search tree.

3.1.3 Solution strategies

There exists several solution strategies to solve transportation problems. The first group of them is based on local search of state space. The strategy searches through the space of candidate solutions and moves from one solution to another until it finds an optimal solution. As an example, we mention Tabu search, described in [19, 20, 21]. This method maintains a list of a forbidden candidate solution (e.g. previously visited solutions or solutions violating a certain rule) to enhance the search performance.

In [4], the VRP is transformed to weighted matching problem, which is afterwards solved using the Parallel savings based heuristics algorithm (PSA). They present three variants of PSA, but only one of them is a polynomial algorithm with a time complexity $O(n^3)$.

Another approach is presented in [38], where the transportation tasks are solved using a stochastic technique. The paper presents a Monte Carlo Tree Search (MCTS), a stochastic optimization algorithm combining classical tree search with random sampling of the search space. The MCTS algorithm builds an asymmetric tree to represent the search space by repeatedly performing four steps, which are illustrated on figure 3.1.1

1) *Selection* – the tree is traversed from the root to a leaf using some criterion called *tree policy* to select the most urgent leaf.
2) *Expansion* – all applicable actions for the selected leaf node are applied and the resulting states are added to the tree as successors of the selected node.
3) *Simulation* – a pseudo-random simulation is run from the selected node until some final state is reached. During the simulation, the actions are selected by a *simulation policy*.
4) *Update/Back-propagation* – the result of the simulation is propagated back in the tree from the selected node to the root. Statistics of the nodes on this path are updated according to the result.

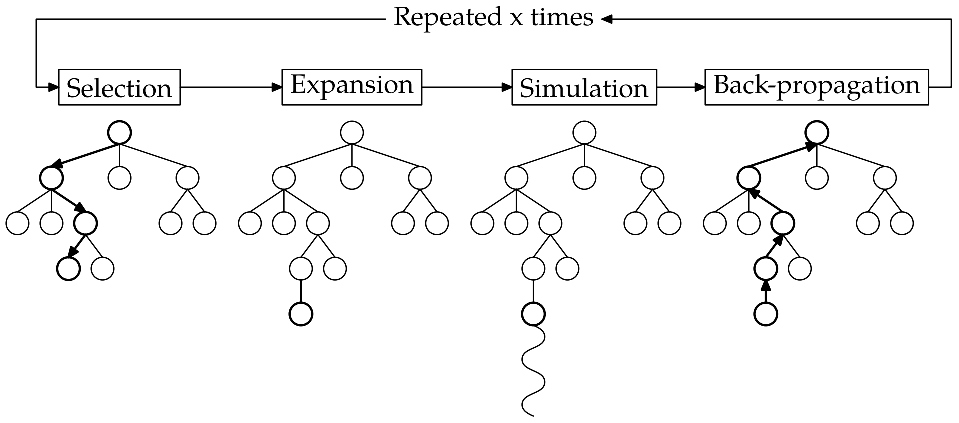

Figure 3.1.1. Basic schema of MCTS from [10]

For a non-stochastic tree search, we can use a "branch and bound" algorithm. It is a general algorithm for finding optimal solutions of optimization problems first described in [27] in 1960. The algorithm uses two tools, a splitting procedure and a bounds procedure. The algorithm systematically enumerates possible solutions and uses the upper and lower bounds to discard solutions which are not feasible. Therefore, there is no need to explore the whole state space to find a feasible solution. As presented in [37], there exists several bounds procedures.

In a survey paper [37], there are presented several bounds procedures for cVRP problems. The paper presents a difference between symmetric cVRP (scVRP) and asymmetric cVRP (acVRP) and proposes different bounds procedures for each of them. The cVRP is asymmetric if it has an asymmetric cost matrix. Otherwise, the cVRP is symmetric. The benchmarks showed that the best bounds procedure for an acVRP is based on the additive approach which considers, in sequence, different infeasible arc subsets so as to produce possibly a better overall lower bound. Likewise, for the scVRP the benchmarks shows that the best bounds procedure is based on b-matching, which is a counterpart to the symmetric version of the assignment relaxation for acVRP – a bounds procedure based on a transformation of the cVRP to the assignment problem.

By adding a cutting planes, we get a "branch and cut algorithm", described in detail in [31]. The branch-and-cut approach was applied to the VRP in [5]. Specifically, they used this approach to solve the Vehicle routing problem with satellite facilities (VRPSF).

3.1.4 Transportation problem and ALP

The transportation problems do not fit well with ALP. The key feature of a transportation problem is a fleet of vehicles with a limited capacity. However, in ALP there is no such condition. In ALP-lp, stores have a limited capacity. However, the stores correspond to locations in vehicle routing problems. On the other hand, in ALP and ALP-lp, there is no capacity constraint to the delivery routes.

They are provided by outsourced delivery companies and are in a practical case unlimited.

3.2 Network flows

Graph algorithms represent a different approach. The task of ALP is to optimally distribute goods from one store to another. This resembles network flows problem, which aims to find a maximum flow in a given network. Finding the maximum flow is not difficult. However, if we generalize the problem and try to search a maximum flow for more than one flow, the problem becomes NP-hard. As described in [28], a multicommodity flows problem involves a simultaneous shipping of multiple commodities through a single network so that the total amount of flows on each edge is no bigger than the capacity of the edge.

We can formally describe the Multicommodity flows as follows: Given an undirected graph $G = (V, E)$ with a positive capacity $c(uv)$ for each edge $uv \in E$ and a set of commodities numbered 1 through k, where each commodity i is specified by a source-sink pair $s_i, t_i \in V$ and a positive demand d_i. For each commodity i, an amount proportional to its demand d_i is shipped from its source s_i to its sink t_i. This gives a *single commodity flow* f_i specified by a set of edge flows $f_i(vw)$ on edges $vw \in E$ where each edge has an arbitrary direction to keep track of which way the flows travel across it. A positive edge flow $f_i(vw) > 0$ denotes a forward flow of commodity i with respect to the direction of edge vW, while a negative flow $f_i(vw) < 0$ denotes a backwards flow. A *multicommodity flow* f consists of k single commodity flows, one for each commodity. In a multicommodity flow f, the total flow $f(vw)$ on each edge $vw \in E$ equals the sum $\sum_{i=1}^{k} |f_i(vw)|$ of the single commodity flows on that edge.

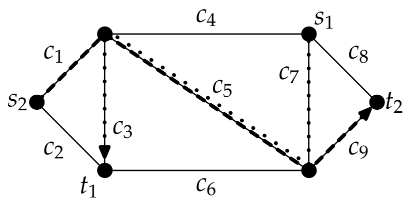

Figure 3.2.1. An example of multicommodity flows graph – edges have capacities c_1 through c_9. There are two flows highlighted, one from the original node (source) s_1 to its destination node (sink) t_1, and a second flow from source s_2 to sink t_2. The dotted line represents a flow for commodity 1 and a dashed line represents a flow for commodity 2.

However, as in the case of a transportation problem, this approach is not suitable for ALP because it solves a problem of limited capacity of connections between nodes with an unlimited capacity. On the other hand, the ALP has unlimited connections. Moreover, in ALP-lp nodes have a limited capacity.

```
function FORD–FULKERSON($G$) returns the maximum flow

  variables: $G$ .... $G = (V, E)$ a network with flow capacity $c$,
                      a source node $s$ and a sink node $t$
             $f$ ..... a network flow
             $c$ ..... a capacity of an edge or path
             $p$ ..... a path in a graph
             $(u, v)$ . an edge in $G$

  $f(u, v) \leftarrow 0$ for all edges $(u, v) \in G$
  while $\exists p$ from $s$ to $t$ in $G$, such that $\forall (u, v) \in p$: $c(u, v) > 0$ do:
    find $c(p) = \min\{c(u, v) : (u, v) \in p\}$
    for each $(u, v) \in p$ do:
      $f(u, v) \leftarrow f(u, v) + c(p)$
    end for
  end for
  return $f$
```

Algorithm A. A pseudo-code of a Ford-Fulkerson algorithm.

Still, the network flows can be used to model the ALP (but not the ALP-lp). We use a Ford-Fulkerson algorithm from [16] as a basis for an algorithm described in section 4.3. The Ford-Fulkerson algorithm is shown as algorithm A.

3.3 Constraint programming

Constraint programming is a programming paradigm which uses constraints to describe a solution rather than to program a way of achieving such a solution. Out of many applications, the constraint programming can be used to model network problems. We discuss it in section 3.3.2. A further description of constraint programming is contained in [13].

3.3.1 Constraint satisfaction problem

A *constraint satisfaction problem* (CSP) consists of:

- a set of variables $\mathbf{X} = \{x_1, \ldots, x_n\}$
- for each variable x_i a finite set D_i of possible values (its *domain*),
- a set of *constraints* restricting the values that the variables can simultaneously take.

A *solution* to a CSP is an assignment of value from its domain to every variable, in such a way that every constraint is satisfied. We may want to find:

- just one solution, with no preference as to which one,

- all solutions,
- an optimal, or at least a good solution, given some objective function defined in terms of some or all of the variables.

We can model all of the constraint problems using only the binary constraints (the constraints with two variables). The CSP can be represented as a constraint multigraph where the variables in the model are the nodes of the graph and the constraints over the variables in the model are the edges connecting the appropriate nodes. The edge $e(x, y)$ is *consistent* if there exists a value in D_y for each value in D_x so that the constraint is satisfied.

The CSP is *arc consistent* if all of the edges in the constraint graph are consistent in both directions. The problem which cannot be transformed to arc consistent state does not have a solution. However, the arc consistency does not ensure that the problem has a solution. For example let us have variables X, Y and Z, all with a domain $\{0, 1\}$ with constraints $X \neq Y$, $Y \neq Z$, $Z \neq X$. The problem is arc consistent, but is not feasible. However, the arc consistency is an important technique, since it helps to reduce the search space.

There are lots of algorithms to solve CSP. The most common of them is *maintaining arc consistency* (MAC) presented in algorithm B. This algorithm is currently the most used technique [6], since it combines the advantages of both search and maintenance of the solution consistency.

```
1.  function MAC–LABELLING($V$, $D$, $C$) returns labelling or false
2.
3.    variables: $V$ .... a set of variables,
4.               $D$ .... a set of domains for variables from $V$
5.               $C$ .....a set of constraints
6.               $x$ ..... a constrained variable
7.               $v$ ..... a possible value of a constrained variable
8.               $R$ ..... a return value, either a set of assigned values or false
9.
10.   if all variables from $V$ are assigned then return $V$ end if
11.   select not-yet assigned variable $x$ from $V$
12.   for each value $v$ from $D_x$ do:
13.     $(TestOK, D') \leftarrow consistent(V, D, C \cup \{x = v\})$
14.     if $TestOK = true$ then
15.       $R \leftarrow$ MAC–LABELLING$(V, D', C)$
16.       if $R \neq fail$ then return $R$ end if
17.     end if
18.   end for
19.   return fail
```

Algorithm B. MAC algorithm from [6].

The algorithm consists of two phases repeated until the solution is found or until it is proved to not exist. The first phase is filtering of the variable domain by eliminating as many as possible values of the domain. The filtering can be achieved by transforming the problem into an arc consistent problem. The filtration phase can end in three ways:

1) The domain of some variable is empty. Therefore, the problem has no solution.
2) The domains of all variables have just one element. The algorithm found the solution.
3) The domains of some variables have more than one element while the problem is arc consistent.

For problem P, which is arc consistent but the solution has not been found yet, we perform the distribution phase. In this phase we introduce a new constraint c and create two new problems, $P \cup \{c\}$ and $P \cup \{\neg c\}$. If the problem P has a solution, then at least one of the new problems has a solution. Afterwards, the filtering phase is run again. The typically used constraint c is of a form $x = v$ or $x < v$. However, any constraint can be used. The *first-fail* strategy is used to select a proper variable for the new constraint c. It picks a variable with the smallest domain. Thus, it will be faster to discover, whether the problem is feasible or not.

The ALP is an optimization problem. We now discuss techniques used to find an optimal solution according to a given objective function. For only a few problems, we can search through the whole search space and return a solution, which has the highest value of an objective function. For most of the problems, the time needed to find the best solution would be inadequately long. We can expect this behavior, since the CSP is an NP-complete technique [17]. However, there are many techniques to search the search space effectively to mostly find the best, or at least a good, solution. Commonly used technique is a method called *branch and bounds* presented in algorithm C. The key idea of a branch and bound method is restricting the upper bound of a domain of the variable representing the value of an objective function. The branch and bound method can be sped up by using both upper and lower bounds.

3.3.2 Constraint programming problems in networks

As described in [36], the problems in networks can be easily modelled using constraint programming. There are three usual types of the model – a link-based model, a path-based model and a node-based model. The models were originally formulated for placing demands over the computer networks. Therefore, bandwidth constraints which use a bandwidth value $\mathrm{bw}(d)$ are added. As we discussed earlier, these constraints are not suitable for an ALP and we will not add them into the ALP model.

In the *link-based model*, for each demand we have one decision variable per link, which states if the link is used for this demand or not.

```
1. function BB-MIN(Variables, V, Constraints) returns optimal solution
2.
3.   variables: Variables .... a set of variables,
4.              Constraints .. a set of constraints
5.              Solution ..... a candidate optimal solution
6.              NewSolution  a temporary variable for a current solution
7.              Bound ....... a current bound
8.              V ........... a variable with a value of an objective function
9.
10.  Bound ← sup
11.  NewSolution ← fail
12.  repeat:
13.     Solution ← NewSolution
14.     NewSolution ← Solve(Variables, Constraints ∪ {V < Bound})
15.     Bound ← value of V in NewSolution (if any)
16.  until NewSolution = fail
17.  return Solution
```

Algorithm C. Branch and bounds method from [6].

1) For each demand d and edge e, a variable X_{de} with domain $\{0, 1\}$ denotes whether the demand is routed over the edge.
2) For every demand d, we also have one $\{0, 1\}$ decision variable Z_d which indicates if the demand is accepted or not.

With the objective function

$$\max_{\{Z_d, X_{de}\}} \sum_{d \in D} \mathrm{val}(d) Z_d \tag{3.3.1}$$

and constraints:

$$\forall d \in \mathbf{D}, \forall n \in \mathbf{N} : \sum_{e \in \mathbf{OUT}(n)} X_{de} - \sum_{e \in \mathbf{IN}(n)} X_{de} = \begin{cases} -Z_d & n = dest(d) \\ Z_d & n = orig(d) \\ 0 & \text{otherwise} \end{cases} \tag{3.3.2}$$

$$\forall e \in \mathbf{E} : \sum_{d \in \mathbf{D}} \mathrm{bw}(d) X_{de} \leq \mathrm{cap}(e) \tag{3.3.3}$$

$$Z_d \in \{0, 1\} \quad X_{de} \in \{0, 1\}$$

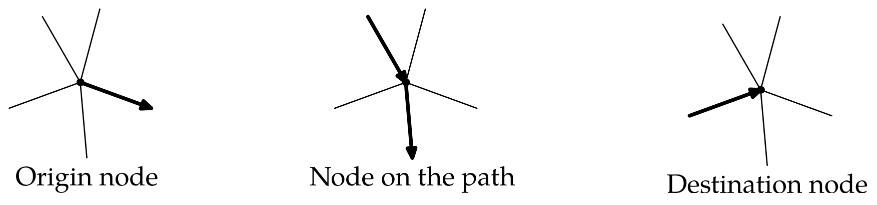

Figure 3.3.1. Effects enforced by the constraint imposed by the equation (3.3.2).

As we can see on figure 3.3.1, the constraint imposed by the equation (3.3.2) is equivalent to the first Kirchhoff's law known from electrical engineering. There must be either no links from and to the node or there must be exactly the same amount of links ending in the node as the links beginning in the node. There are two exceptions – the origin node which is a beginning of exactly one link and the destination node which is an end of exactly one link.

In the *path-based model* [36], the decision variables represent paths used to route demands.

1) For each demand d we assume there are $path(d)$ possible paths for the demand.
2) For each demand d and possible path i, we introduce a $\{0, 1\}$ variable Y_{id} which denotes whether the demand is routed over the path.
3) For each edge e and path i for demand d, a constant h^e_{id} indicates whether the path is routed over the edge.
4) For every demand d, we also have one $\{0, 1\}$ decision variable Z_d which indicates if the demand is accepted or not.

With the objective function

$$\max_{\{Z_d, T_{id}\}} \sum_{d \in \mathbf{D}} \mathrm{val}(d) Z_d \tag{3.3.4}$$

and constraints:

$$\forall d \in \mathbf{D} : \sum_{1 \le i \le path(d)} Y_{id} = Z_d \tag{3.3.5}$$

$$\forall e \in \mathbf{E} : \sum_{d \in \mathbf{D}} \mathrm{bw}(d) \sum_{1 \le i \le path(d)} h^e_{id} Y_{id} \le \mathrm{cap}(e) \tag{3.3.6}$$

$$Z_d \in \{0, 1\} \quad Y_{id} \in \{0, 1\}$$

In the *node-based model* [36, 34], the decision variables represent successor relations between network nodes.

For each demand $d \in \mathbf{D}$ and each node $k \in \mathbf{N}$ in the network, we introduce an integer decision variable S_{kd} with the following domain:

$$S_{kd} :: \begin{cases} \left\{sink(e) | e \in \mathbf{OUT}(k)\right\} & k = orig(d) \\ orig(d) & k = dest(d) \\ \{0\} \cup \left\{sink(e) | e \in \mathbf{OUT}(k)\right\} & \text{otherwise} \end{cases} \tag{3.3.7}$$

For each demand, the domain for a node contains all possible successors and the value 0, which indicates that the node is not used to route the demand. The destination node contains a back-link to the source. The constraint is set so that for every demand d the set

$$\left\{\langle k, S_{kd}\rangle | S_{kd} \neq 0, k \in \mathbf{N}\right\} \tag{3.3.8}$$

forms a cycle in the graph, as shown in figure 3.3.2.

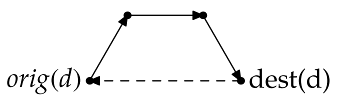

Figure 3.3.2. Node-based constraint model. The constraint imposed in the equation (3.3.7) enforces that the nodes on the path form a cycle – each node is pointing to another node on the path and the destination node points to the original point.

The capacity constraint for each node can be expressed with a cumulative constraint which uses two arguments, a set of task given by *start*, *duration* and *resource use* and a resource profile, given as a set of tuples *time point* and *resource limit*.

$$\forall i \in \mathbf{N}: \text{cumulative}\Big(\{\langle S_{id}, 1, bw(d)\rangle | d \in \mathbf{D}\}, \{\langle l, m\rangle | 0 \leq l \leq |\mathbf{N}|\}\Big) \tag{3.3.9}$$

where

$$m = \begin{cases} \infty & l = 0 \\ cap(e) & \exists e \in \mathbf{E}, \text{ st. } source(e) = i, sink(e) = l \\ 0 & \text{otherwise} \end{cases}$$

This model needs $|\mathbf{D}|$ cycle constraints and $|\mathbf{N}|$ cumulative constraints to express the conditions of the routing problem.

3.3.3 Constraint programming and ALP

ALP can be easily modelled as a constraint programming problem. The path-based model, as described in section 3.3.2, is suitable for ALP, since the expected number of paths is relatively low. Both link-based and node-based models solve not only the problem of optimal transfers, but they must deal with path-finding, as well. The duration of a path depends on a shipment time. Therefore, the constrained model would need to compute it. This capability would make the model unnecessarily complex. Therefore, we use a path-based model as a basis for the model solving an ALP.

3.4 Mixed integer programming

The CSP based model for ALP consists mostly of sum constraints. Constraint programming can handle such models. Though, there is a better technique for such a model. We will now present *Mixed integer programming* (MIP), which effectively deals with models based on sums and inequalities similar to the above mentioned model for planning of transfers in networks [9, 24].

3.4.1 Linear and integer programming

Mixed integer programming is a method derived from a linear programming method. Linear programming problem is stated as follows [11]:

Find the values of $\lambda_1, \lambda_2, \ldots, \lambda_n$ which maximize the linear form

$$\lambda_1 c_1 + \lambda_2 c_2 + \cdots + \lambda_n c_n \tag{3.4.1}$$

subject to the conditions that

$$\lambda_j \geq 0 \quad (j = 1, 2, \ldots, n) \tag{3.4.2}$$

and

$$\begin{array}{ccccccccc}
\lambda_1 a_{11} & + & \lambda_2 a_{12} & \cdots & + & \lambda_n a_{1n} & = & b_1 \\
\lambda_1 a_{21} & + & \lambda_2 a_{22} & \cdots & + & \lambda_n a_{2n} & = & b_2 \\
\vdots & & \vdots & \ddots & & & & \vdots \\
\vdots & & \vdots & & \ddots & & & \vdots \\
\vdots & & \vdots & & & \ddots & & \vdots \\
\lambda_1 a_{m1} & + & \lambda_2 a_{m2} & \cdots & + & \lambda_n a_{mn} & = & b_m
\end{array} \tag{3.4.3}$$

There exists a straightforward process to convert any linear program into the form specified in the equations (3.4.1), (3.4.2) and (3.4.3). The system of linear inequalities defines a polytope as a feasible region. The *simplex* algorithm, proposed in [11], finds an optimum by traversing along the edges of polytope

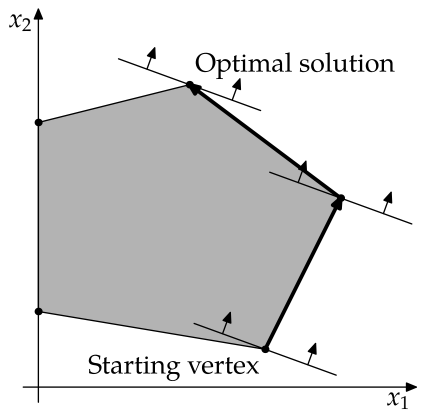

Figure 3.4.1. A twodimensional polytope defined by a feasible region. The steps of the simplex algorithm and the linear cost function (objective) in the respective vertices are demonstrated.

until it reaches the vertex of the optimum solution. Worst-case time complexity of the simplex algorithm is exponential. However, in the average case the algorithm is polynomial.

When we restrict the variables to contain only the integer values, we get the *Integer programming* problem (IP). Unfortunately, contrary to LP, the IP is an NP-complete problem [32]. The binary variant of logical programming, i.e. the linear programming where variables have a $\{0, 1\}$ domain is NP-complete and is a part of the Karp's 12 NP-complete problems [26]. Finally, when we combine linear and integer programming, we get the Mixed integer programming problem. This problem is NP-hard, as well. However, there exists several efficient solvers to this problem such as the SCIP system [3].

3.4.2 Mixed integer program

The *mixed integer program* is defined as follows [2]:

Given a matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$, and a subset $I \subseteq N = \{1, \ldots, m\}$, the mixed integer program $MIP = (A, b, c, I)$ is to solve

$$c^* = \min \left\{ c^T x | Ax \leq b, x \in \mathbb{R}^n, x_j \in \mathbb{Z} \text{ for all } j \in I \right\}$$

The vectors in the set $X_{MIP} = \left\{ x \in \mathbb{R}^n | Ax \leq b, x_j \in \mathbb{Z} \text{ for all } j \in I \right\}$ are called *feasible solutions* of MIP. A feasible solution $x^* \in X_{MIP}$ is called *optimal* if its objective value satisfies $c^T x^* = c^*$.

In contrast to constraint programming, in mixed integer programming, we are restricted to linear constraints, a linear objective function and integer or real-valued domains. We define l_j and u_j as lower and upper bounds of a variable x_j. It holds that $l_j \leq x_j \leq u_j$. Next, we define a set of binary variables as $B = \{j \in I | l_j = 0 \text{ and } u_j = 1\}$. There exist the following specializations of MIP:

- $I = \emptyset$: linear programs,
- $I = N$: integer programs,
- $B = I$: mixed binary programs, and
- $B = I = N$: Binary programs,

3.4.3 Solution strategies

As discussed in [2], the MIP solvers use previously discussed branch and bounds method to find an optimal solution. First a solver uses a branching rule to branch a search tree. Then, it selects the next subproblem from the search tree to be processed. Afterwards, the selected node is a subject to a domain propagation. Finally, the LP relaxation of a node is strenghtened by the use of a cutting plane separation algorithm.

A commonly used branching technique is *branching on variables*. For a problem Q, if x_j ($j \in I$), is some integer variable with a fractional value x'_j, we obtain two subproblems: one by adding the inequality $x_j \leq \lfloor x'_j \rfloor$ (the *left subproblem* denoted as Q_j^-) and one by adding the inequality $x_j \geq \lceil x'_j \rceil$ (the *right subproblem* or Q_j^+). More complex branching is rarely incorporated into general MIP solvers.

Once the problem is divided into two subproblems, the solving process can continue with any subproblem that is a leaf of the current search tree. However, the search must handle with two, usually opposing, goals:

- It must find good feasible MIP solutions to improve primal (upper) bound
- It must improve the global dual (lower) bound.

The integral LP solutions tend to be deep in the search tree [29]. Therefore, the *depth first search* seems to be the most natural choice. However, it does not fit well with the second goal. This goal can be achieved with *best first search* method, which always selects a subproblem with the smallest dual bound of all remaining leaves in the tree. As a combination of both methods we can use *best first search with plunging*. The method at first, as long as the current node has unprocessed children, selects one of the children as the next node. Otherwise, the plunging continues with one of the current node's siblings. If no more unprocessed children or siblings are available, the current plunge is completed and a leaf from the tree with best dual bound is selected.

Afterwards, the solver tightens domains of variables by inspecting the constraints and the current domains of other variables at a local subproblem in the search tree. This process is called a *domain propagation* or a *node processing*. LP relaxation cannot handle holes inside a domain. Therefore MIP solvers only alter bounds of a domain.

Finally, a cut separation phase occurs. The method iteratively refines a feasible set or objective function by adding new linear inequalities called *cuts*. There are two basic variants a *cut and branch* and *branch and cut* depending on whether the cutting planes are only generated at the root node or at local subproblems in the search tree, as well. Despite not being initially accepted, nowadays, the most common cut separation technique is *Gomory mixed integer cuts* described in [22]. The method first solves the problem while ignoring the integer requirement. If the found solution is not an integer, then the method finds a hyperplane with a solution on one side and all feasible integer points on the other. Afterwards, it transforms the hyperplane to a new inequality and adds it to the problem.

3.4.4 Mixed integer programming and ALP

Both the ALP and ALP-lp can be easily reformulated as a mixed integer programming problem. We can expect, that this model would have a better performance than CSP based model, since the path-based model as described in section 3.3.2 uses constraints which are not easily evaluated. On the other hand, the MIP solvers are specialized and tuned up for such a type of constraints.

Chapter 4

Algorithms for automated logistics

In this chapter, we propose algorithms for ALP, as defined in chapter 2.

4.1 Preliminary tasks

In this section we summarize the tasks, which must be performed prior to running an actual algorithm solving ALP

4.1.1 Schema of the system

The whole process is directed by documents in the ERP system. The system will first get information from the ERP system, plan new transfers and produce new documents to the ERP system. The basic schema of the algorithm is shown on figure 4.1.1.

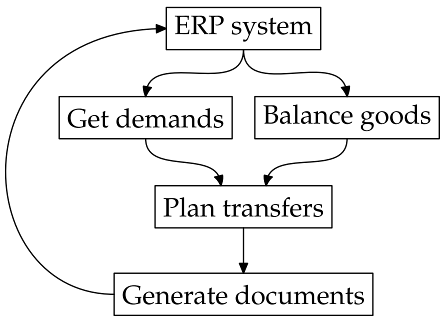

Figure 4.1.1. Basic schema of the algorithm

In general, the system performs the following steps:

1) The system loads data from ERP. It gets the information about pending transfers, availability of goods and demands for goods.
2) Afterwards, it receives demands from an external software system responsible for balancing the goods on stores.
3) Then, the system uses one of the algorithms to plan the transfers of goods to meet the requirements.
4) Finally, the system generates or alters the documents in ERP to perform plans generated in the Step 3.

4.1.2 Delivery time

For the purposes of proper planning, we need to estimate a duration of planned transfers. After the system creates a shipment order, an employee must process it. They must search for the demanded goods in the store, pack it and hand it over to a delivery company. As described in section 2.1.4, the delivery company stops only once in a while. If the goods is not prepared in time for the pick-up, it will be handed over in the next pick-up. To compute a delivery time, we use the algorithm D.

```
 1. function DELIVERY–TIME($path$, $time$)
 2.           returns a delivery time after transferring goods along the path
 3.   variables: $path$ .... a sequence of stores $s$ from the original store $s_0$ to the
 4.                          destination store $s_n$.
 5.              $time$ .... a shipment time
 6.              $T$ ....... a computed total time, initially zero
 7.              $s$ ........ currently examined store
 8.              $s'$ ....... temporary variable containing
 9.                          a next store on the path
10.
11.   $T \leftarrow time$
12.   $s \leftarrow$ first store on $path$
13.   while $s \neq$ last store on $path$ do:
14.     $s' \leftarrow$ next store on $path$ after $s$
15.     $T \leftarrow T$ + time needed to prepare shipment in $s$ at $T$
16.     $T \leftarrow$ next pick-up time at $s$ in direction to $s'$ after $T$
17.     $T \leftarrow T$ + transport duration from $s$ to $s'$ at $T$
18.     $T \leftarrow$ next delivery time at $s'$ in direction from $s$ after $T$
19.     $T \leftarrow T$ + time needed to recieve shipment in $s'$ at $T$
20.     $s \leftarrow s'$
21.   end while
22.
23.   return $T$
```

Algorithm D. Algorithm for computing the delivery time

The delivery companies as well as stores do not operate during weekends and public holidays. The algorithm D takes into account this condition on the lines 15, 17 and 19, since the prolongation is included in the delivery time and time needed to prepare and receive a shipment.

4.1.3 Recovering from the queue jumping

When a salesman jumps the queue, they remove an item reserved for a certain customer or planned for a transfer. Therefore, we must recover from a situation when the transfer is planned but there is no goods available. First, we must detect whether there was any goods planned for a transfer but is currently unavailable and perform a specific action. Thus, we check all goods on unresolved shipment orders using algorithm E.

```
function SO–AVAILABLE(shipment_order, commodity)
            returns available amount of commodity on shipment_order

   variables: shipment_order .... a shipment order to be checked
              commodity ........ a type of goods
              s .................. a store related to shipment_order
              onStock ............amount of commodity present at s
              ordered ............ amount of commodity on shipment_order
              blocked ............ amount of commodity blocked by
                                   previous shipment orders
              available ...........amount of commodity available at s,

   s ← get store from shipment_order
   onStock ← amount of goods on stock at s
   ordered ← amount of goods on shipment_order
   blocked ←amount of goods on shipment orders
             older than shipment_order at s
   available ← onStock − blocked

   if available ≤ 0 then
      return 0
   else if available ≥ ordered then
      return ordered
   else
      return available
   end if
```

Algorithm E. Algorithm computing the available amount of goods for a given shipment order.

If the amount of available items is smaller than the ordered amount, the shipment order cannot be processed. We could send the missing goods from another store, if available, but it is not effective since it would be better to transfer it from the store directly to a destination store of the shipment order. Therefore, the shipment order should be at least partially cancelled. This situation can be handled in two different ways. The simple solution is to do nothing. The transfer order was generated due to a demand from some other store. Therefore, if the demand is still valid it will be executed in the next run of the planning algorithm, if possible. Even though the original demand is delayed, at least it will be processed sometime in the future and will not wait until the goods will be replenished.

The second option to solve the missing goods is to rearrange other planned transfers which were set after the affected shipment order. Unfortunately, we cannot simply change the destination of a transfer. Each document is identified by series of documents and a sequence number which are unchangeable. The serie number carries an information about the type of the document. Moreover, in case of transport shipment orders, each serie is assigned to a pair of stores; the original and the destinated store. Therefore, we need to create a new document instead of changing the original one. However, this would not solve the problem since the shipment orders are processed in the order of their creation. As a result, we refuse this solution.

4.1.4 Recovering from the store outage

When a store outage occurs, the respective store cannot send or receive goods. In case of planned outages, they can be taken into account in the algorithm D for new transfers, since it is the same situation as weekends or public holidays. However, we still need to handle unprocessed transfers planned before the outage. If we are able to estimate a duration of the outage, we can check if it is possible to transfer the goods from another store faster. In such a case, we will cancel the shipment order in the outaged store and place a new shipment order in the other store. Nothing, however, can be done concerning other situations. If we do not know the estimated duration of the outage, we can use a predefined universal duration (e.g. two weeks) or we can simply deal with the situation in the same way as in the case of queue jumping described in section 4.1.3, because the goods on the shipment order is effectively missing.

4.2 Naïve algorithm

In this section, we will describe a naïve algorithm for ALP. This algorithm only covers the current insufficiency in stores and does not respect an order of precedence of shipment orders. However, this approach is still reasonable if there is enough goods in other stores and if we run the algorithm regularly. Even though the algorithm itself does not maintain an order of demands (i.e. the constraint (2.3.5) on page 16), if it is run regularly in small time steps, the constraint will be satisfied implicitly. The order will not be maintained only for the demands which arise between the time steps. The shorter the time step is, the lesser amount of demands is affected. The algorithm does not deal with capacity constraints. Therefore, it is not suitable for ALP-lp.

For each type of goods, the algorithm divides stores into two groups according to the available amount of goods. The first group of stores S (source stores) contains stores where the available amount of goods is positive. The second group of stores D (destination stores) contains stores where the available amount of goods is negative.

The algorithm is run separately for each type of goods. The algorithm picks up the first store in D and searches for the fastest transfer from a store in S. The algorithm is listed in detail as an algorithm F.

```
function PLAN–TRANSFERS–NAIVE($commodity$, $stores$)

  variables: $commodity$ .... a type of goods
             $stores$ .......... a set of stores
             $strengths$ ...... a set of sales strengths for each store
             $Availability_s$ ... availability of $commodity$ in store $s$
             $Availability^*_s$ ... availability of $commodity$ in store $s$
                                including goods on the way
             $S$ .............. a set of source stores
             $D$ ............. a set of destination stores
             $s_s, s_d$ .......... source and destination stores
             $missing$ ....... a current amount of missing items
                                of $commodity$
             $toTransfer$, an amount of $commodity$ to be transferred

  $S \leftarrow \{s_s | s \in stores, Availability_{s_s} > 0\}$
  $D \leftarrow \{s_d | s \in stores, Availability^*_{s_d} < 0\}$
  for each store $s_d \in D$ do:
    while $Availability^*_{s_d} < 0$ do:
      $s_s \leftarrow$ origin of $\left( \underset{path \text{ from } S \text{ to } s_d}{\arg\min} \{\text{DELIVERY–TIME}(path, now)\} \right)$
      if $s_s = null$ then
        break while
      end if
      $missing \leftarrow -Availability^*_{s_d}$
      $toTransfer \leftarrow \min\{missing, Availability_{s_s}\}$
      PLAN-TRANSFER($commodity, s_s, s_d, toTransfer$)
      $Availability^*_{s_d} \leftarrow Availability^*_{s_d} + toTransfer$
      $Availability_{s_s} \leftarrow Availability_{s_s} - toTransfer$
      if $Availability_{s_s} = 0$ then $S = S \setminus \{s_s\}$ end if
    end while
  end for
```

Algorithm F. Naïve algorithm for automated logistics.

The naïve algorithm is used to compare the Ford-Fulkerson based, CSP based, and MIP based algorithms with a trivial solution of ALP.

4.3 Ford-Fulkerson based algorithm

The more advanced algorithm is based on the Ford-Fulkerson algorithm described in section 3.2. This algorithm solves an ALP problem while maintaining the order of demands. However, it is not able to handle constraints preventing overloading.

For each of goods, we construct a graph as showed on figure 4.3.1. The capacities of edges from the source correspond to the amount of available goods in the particular store and the capacities of edges to the destination correspond to the demanded amount of goods in the stores. Then, we use the Ford-Fulkerson algorithm to find the maximum flow through the network. Augmenting paths generated by the algorithm are then transformed to documents in ERP. The algorithm differs from the original algorithm in a way, in which it selects an augmenting path to be added. First, we process demands in their orders. For each demand, we search for a suitable path with the shortest duration. Since the original Ford-Fulkerson algorithm allows us to place augmenting paths in any order and finds the maximum flow through network, it is guaranteed that our algorithm finds the maximum flow in the network, as well.

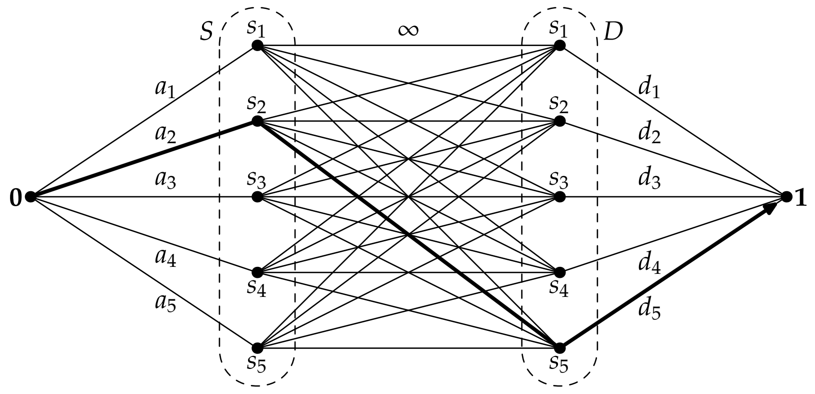

Figure 4.3.1. A bipartite graph construed for the Ford–Fulkerson based algorithm. There are five stores s_n with the amount of available goods a_n and the demanded amount d_n. The augmenting path $(\mathbf{0}, s_2, s_4, \mathbf{1})$ is highlighted. All edges have left-to-right direction. The edges are not drawn as arrows since due to their amount the picture would become unreadable.

Now we describe the algorithm in a detail. The Ford-Fulkerson finds the maximum flow in a given network. It searches for augmenting paths from the source to the sink of the network. An augmenting path is a path where all of its edges are not fully saturated. Thus, we can increase the flow through the network along the path. The new flow cannot exceed the capacity of any edges on the path.

For each type of goods, we construct a bipartite graph $G = (\mathbf{V}, \mathbf{E})$ consisting of a set of source stores $\mathbf{S}$ and a set of destination stores $\mathbf{D}$. Moreover, we add dummy nodes $\mathbf{0}$ and $\mathbf{1}$ representing the source and the sink of the network. The following conditions hold:

$$\mathbf{V} = \mathbf{S} \cup \mathbf{D} \cup \{\mathbf{0}, \mathbf{1}\} \tag{4.3.1}$$

$$\mathbf{S} \cap \mathbf{D} = \emptyset \tag{4.3.2}$$

$$\{\mathbf{0}, \mathbf{1}\} \cap (\mathbf{S} \cup \mathbf{D}) = \emptyset \tag{4.3.3}$$

$$\mathbf{E} = \{(\mathbf{0}, s) | s \in \mathbf{S}\} \cup \{(s_s, s_d) | s_d \in \mathbf{S}, s_d \in \mathbf{D}\} \cup \{(s, \mathbf{1}) | s \in \mathbf{D}\} \tag{4.3.4}$$

For each possible delivery route between the stores $s_s \in S$ and $s_d \in D$, there is an edge $(s_s, s_d) \in E$ with an unlimited capacity. Finally, for each node $s_s \in S$ there is an edge $(0, s_s) \in E$ with capacity representing the amount of available goods in store s_s. Similarly, for each node $s_d \in D$ there is an edge $(s_d, 1) \in E$ with capacity representing the demanded amount of goods in store s_d. We allow only integer flows.

As we mentioned earlier, the Ford–Fulkerson algorithm finds the maximum flow by placing augmenting paths until no other path can be placed without violating the capacity constraints. We generate shipment orders from the generated augmenting paths. Each path from s_s to s_d with a flow f is transformed to a shipment order for a transfer of f items of goods from store s_s to store s_d, if $s_s \neq s_d$.

The key part of the algorithm is an order in which the augmenting paths are tested and placed. As we defined in section 2.3, a demand is a 4-tuple $(s, g, amount, date)$. We sort all demands $d \in \mathbf{D}$ according to their *date* in ascending order. Next, we process the demands in their order and try to place the augmenting paths. For each demand, we generate feasible paths in a graph and select the path with the earliest delivery time (algorithm D on page 34). The pseudo-code of the algorithm is presented as algorithm G on page 40.

```
1.  function PLAN–TRANSFERS-FF($g$, $D$, $S$)
2.
3.     variables: $g$ ........... a goods
4.                $s$ ........... a set of stores
5.                $D$ .......... a set of demands
6.                $V$ .......... a set of nodes of a construed graph
7.                $E$ ........... a set of edges of a construed graph
8.                $cap$ ......... a capacity function of a construed graph
9.                $d$ ........... one particular demand
10.               $s, s'$ ......... stores
11.               $amount$ ..... an amount of $commodity$ on
12.                              a shipment order
13.               $i$ ........... an auxiliary iteration variable
14.               $paths$ ....... a set of paths
15.               $p$ ........... a path consisting of nodes $p[1]$, ..., $p[4]$
16.               $toTransfer$ ... an amount of goods planned to be transferred
17.
18.    sort $D$ by $date$ in ascending order
```

```
19. $V \leftarrow (S \times \{s, d\}) \cup \{\mathbf{0}, \mathbf{1}\}$
20. $E \leftarrow \emptyset$
21. **for each** $s \in S$ **do:**
22.    $E \leftarrow E \cup \{(\mathbf{0}, s_s), (s_d, \mathbf{1})\}$
23.    $cap(\mathbf{0}, s_s) \equiv \max\{0, Availability_{g,s}\}$
24.    $cap(s_d, \mathbf{1}) \equiv \max\{0, -Availability^*_{g,s}\}$
25.    **for each** $s' \in S$ do:
26.       $E \leftarrow E \cup \{(s_s, s'_d)\}$
27.       $cap(s_s, s'_d) \equiv \infty$
28.    **end for**
29. **end for**
30. **for each** $d \in D$ **do:**
31.    **if** $goods(d) \neq g$ **then** skip the iteration and proceed with another $d$ **end if**
32.    $s \leftarrow store(d)$
33.    $amount \leftarrow amount(d)$
34.    **while** $amount > 0$ **do:**
35.       $paths \leftarrow \Big\{(\mathbf{0}, s_s, s, \mathbf{1}) | cap(\mathbf{0}, s_s) > 0 \wedge cap(s_s, s) > 0 \wedge cap(s, \mathbf{1}) > 0\Big\}$
36.       **if** $paths = \emptyset$ **then**
37.          **break**
38.       **end if**
39.       $p \leftarrow \arg\min_{p \in paths} \{\text{DELIVERY–TIME}(p, now)\}$
40.       $toTransfer = \min\Big\{cap(p[1], p[2]), cap(p[2], p[3]), cap(p[3], p[4]), amount\Big\}$
41.       PLAN-TRANSFER($commodity, p[2], s, toTransfer$)
42.       $cap(\mathbf{0}, p[2]) \equiv cap(\mathbf{0}, p[2]) - toTransfer$
43.       $cap(p[2], p[3]) \equiv cap(p[2], p[3]) - toTransfer$
44.       $cap(p[3], \mathbf{1}) \equiv cap(p[3], \mathbf{1}) - toTransfer$
45.    **end while**
46. **end for**
```

Algorithm G. Ford-Fulkerson based algorithm.

4.4 CSP model

We propose a model based on the path-based model described in section 3.3.2. As we have mentioned, we are not limited by the capacity of edges and we plan to transfer lots of types of goods at once. However, we need to prevent the overload of stores. Therefore, we choose a path-based model and we propose a CSP model based on it. The model solves both ALP and ALP-lp. Since the ALP-lp is a multiple-criteria optimization problem, the model depends on two parameters α and β, one for each goal of ALP-lp.

- **S**, a set of stores.
- **G**, a set of goods.
- **D**, a set of demands. When a demand i has been placed before a demand j, it holds that $i < j$.

- $\mathbf{P}$, a set of all possible paths between stores.
- $\mathbf{T}$, a set of identifiers of days, when the stores process the goods if sent along any path from $\mathbf{P}$.
- Orig: $\mathbf{P} \to \mathbf{S}$, a function returning an original store of a given path.
- Dest: $\mathbf{P} \to \mathbf{S}$, a function returning a destination store of a given path.
- Amount: $\mathbf{D} \times \mathbf{G} \to \mathbb{N}_0$, a function returning an amount of goods demanded by a given demand.
- Availability: $\mathbf{G} \times \mathbf{S} \to \mathbb{Z}$, a function returning an amount of goods available in a given store. It is a difference between *OnStock* and a sum of all demands on a given store for the goods.
- Capacity: $\mathbf{S} \to \mathbb{N}$, a function returning a daily capacity of a given store.
- Duration: $\mathbf{P} \to \mathbf{T}$, a function returning a time needed to transfer goods along the path, when sent right after the execution of the algorithm.
- OnTheWay: $\mathbf{G} \times \mathbf{S} \to \mathbb{N}_0$, a function returning an amount of goods which is delivered to a given store.
- Pri: $\mathbf{D} \to \{\text{high}, \text{low}\}$, a function returning a priority for a given demand.
- Time: $\mathbf{P} \times \mathbf{S} \to \mathcal{P}(\mathbf{T})$, a function that for a given store and a given path returns a set of identifiers of the days when the goods sent along the path is processed in the store.
- Load: $\mathbf{S} \times \mathbf{T} \to \mathbb{N}_0$, a function that for a given store and a given identifier of a day returns the amount of goods which is already planned to be processed in the store during the day.

Next, we introduce constrained variables:

- $X^g_{dp} \in \{0, 1\}$, a decision variable representing whether the demand d for a goods g is at least partially transferred along the path p, used in the objective function.
- $Y^g_{dp} \in \mathbb{N}_0$, a variable containing an amount of goods g for a demand d transferred along a path p.
- $Z^g_d \in \mathbb{N}_0$, a variable containing a total amount of goods transferred for a demand d.

The CSP model consists of an objective function

$$\min \left\{ \alpha \sum_{g \in \mathbf{G}} \sum_{d \in \mathbf{D}} \sum_{p \in \mathbf{P}} X^g_{dp} \text{Duration}(p) - \beta \sum_{g \in \mathbf{G}} \sum_{d \in \mathbf{D}} \sum_{p \in \mathbf{P}} Y^g_{dp} \right\} \tag{4.4.1}$$

and the following constraints:

$$\forall g \in \mathbf{G}, \forall s \in \mathbf{S}\colon \sum_{d \in \mathbf{D}} \sum_{\substack{p \in \mathbf{P} \\ s = \mathrm{Orig}(p)}} Y^g_{dp} \leq \mathrm{Availability}(g, s) \tag{4.4.2}$$

$$\forall g \in \mathbf{G}, \forall d \in \mathbf{D}, \forall p \in \mathbf{P}, \mathrm{Dest}(p) \neq \mathrm{Dest}(d)\colon Y^g_{dp} = 0 \tag{4.4.3}$$

$$\forall g \in \mathbf{G}, \forall d \in \mathbf{D}\colon Z^g_d = \sum_{p \in P} Y^g_{dp} \tag{4.4.4}$$

$$\forall g \in \mathbf{G}, \forall d \in \mathbf{D}\colon Z^g_d \leq \mathrm{Amount}(d, g) \tag{4.4.5}$$

$$\forall g \in \mathbf{G}, \forall d \in \mathbf{D}, \forall p \in \mathbf{P}\colon Y^g_{dp} > 0 \Leftrightarrow X^g_{dp} = 1 \tag{4.4.6}$$

$$\forall g \in \mathbf{G}, \forall s \in \mathbf{S}, \forall d \in \mathbf{D}, d > \min\left\{i | i \in \mathbf{D}, \mathrm{Amount}(i, g) > 0, s = \mathrm{dest}(i)\right\} :$$
$$Z^g_d > 0 \Rightarrow \bigwedge_{\substack{i \in \mathbf{D}, i<d \\ s = \mathrm{Dest}(i) \\ \mathrm{Pri}(i) = \mathrm{high} \\ \mathrm{Amount}(i,g) > 0}} \left(Z^g_i = \mathrm{Amount}(i, g)\right) \tag{4.4.7}$$

$$\forall g \in \mathbf{G} : \sum_{\substack{d \in \mathbf{D} \\ \mathrm{Pri}(d) = \mathrm{high}}} Z^g_d = \min\left\{ \sum_{\substack{s \in \mathbf{S} \\ \mathrm{Availability}(g,s) > 0}} \mathrm{Availability}(g, s), \sum_{\substack{d \in \mathbf{D} \\ \mathrm{Pri}(d) = \mathrm{high}}} \mathrm{Amount}(d, g) \right\} \tag{4.4.8}$$

$$\forall s \in \mathbf{S}, \forall t \in \mathbf{T} : \mathrm{Load}(s, t) + \sum_{g \in \mathbf{G}} \sum_{d \in \mathbf{D}} \sum_{\substack{p \in \mathbf{P} \\ t = \mathrm{Time}(p,s)}} Y^g_{dp} \geq \mathrm{Capacity}(s) \Rightarrow$$
$$\Rightarrow \sum_{g \in \mathbf{G}} \sum_{\substack{p \in \mathbf{P} \\ t = \mathrm{Time}(p,s)}} \sum_{\substack{d \in \mathbf{D} \\ \mathrm{Pri}(d) = \mathrm{low}}} Y^g_{dp} = 0 \tag{4.4.9}$$

First, we add a constraint (4.4.2) restricting the amount of goods which can be transferred from a particular store. The sum of the transferred amounts cannot exceed the available amount of the goods in the store.

Next, we will add a constraint (4.4.3) which allows to serve a demand only with paths which end in its destination node.

For the demands itself, we define constraints (4.4.4) and (4.4.5) which limit the total flow of goods based on the value of the demand.

For the objective function, as described below, it is useful to introduce a constraint (4.4.6), which allows us to detect used paths.

To enforce an order of precedence, we introduce a constraint (4.4.7). We can plan a transfer for a demand only if all previous demands were resolved, i.e. the sum of goods which is currently on the way and of all already planned transfers for the goods covers at least the sum of demanded amount of goods on the demands.

Importantly, we must ensure that we will plan as many transfers as possible. Otherwise, the objective function (4.4.1) would prefer the empty solution (which is the shortest possible). Therefore, we introduce a constraint (4.4.8).

Finally, we introduce a constraint (4.4.9) which prevents the overload of stores. Each store has a given daily capacity which should not be exceeded if possible. We can always plan a high-priority transfer but we cannot plan a low-priority transfer through a store, if it would cause its overload.

For this model, we introduce an objective function (4.4.1). There are two requirements for the solution. First, the final plan should be as fast as possible. Secondly, as many demands as possible should be resolved. These two requirements are contradictory. If we decide to avoid all of the low-priority transfers, we will get faster plan than if we have resolved some of them. On the other hand, if we decide to resolve as many as possible low-priority transfers, the resulting plan does not have to be the fastest one. Therefore, we introduce the parameters α and β. We can use them to tune the model to our needs.

The model as proposed has one disadvantage. The number of constrained variables is huge even for small problems. The number of constrained variables is $O(|\mathbf{G}| \cdot |\mathbf{S}|^2 \cdot |\mathbf{D}|)$, since there are $|\mathbf{G}|$ goods, $|\mathbf{S}|^2$ paths and $|\mathbf{D}|$ demands. However, we can radically decrease the number of constrained variables. We do not have to post variables which would clearly not be satisfied. For example, the variable Y^g_{dp} should be posted only if the demand d demands a goods g and if its store is the same as the destination store of p. Moreover, we should put into a set $\mathbf{D}$ only demands, which could be potentially resolved (i.e. there is at least one item of goods available in stores) and into a set $\mathbf{G}$ put only goods which is requested by some of the demands already placed in $\mathbf{D}$.

4.4.1 CSP solver

To implement the previously stated model, we use a state of the art CSP solver. Nowadays, there are many solvers available but we seek for a cost optimal solution, which is preferably Java based, since it would easily merged with other Sunnysoft program environment. Therefore, based on the experience in [39] we used a Java based CSP solver Choco [1].

The proposed constraints can be directly rewritten in Choco. Thus, we do not need to reformulate the model to meet the available constraints. Still, there are some issues connected to the implementation. Some of them are discussed in the Appendix C.

The solver offers us to choose a search strategies out of many predefined strategies or to implement our own search strategy. By default, the solver uses a *minimum domain lower bound strategy* on all integer variables and *lexicographic upper bound strategy* on all binary variables. The minimum domain lower bound strategy selects a variable x using the first-fail strategy, described in section 3.3 and tries to set its value to the lower bound of its domain, i.e. $x = lower\_bound(D_x)$. In case of binary variables, all of them have a domain containing one or two elements. Therefore, Choco solver does not use the first-fail strategy. Instead, it simply selects the variables in order, in which they were posted to the solver and tries to assign them their current upper boundary.

The default strategy is usually not optimal, since it tries to assign values to all variables, disregarding their importance to the problem. For example, in our CSP based model, we have to introduce auxiliary variables, which hold results of the sum constraints. However, all of them depend exclusively on values of Y^g_{dp}. Therefore, we should alter the default strategy to only search through Y^g_{dp} variables, since all other variables in the problem are derived from them.

4.5 MIP model

The model described in section 4.4, is not the optimal model for CSP. In a typical CSP model, using several sum constraints poses no problem. However, the path-based model consists almost entirely out of sums. For such a model, the method is not suitable. However, with a slight modification, we can reformulate it as a MIP problem. MIP solvers are optimized for sum constraints. Compared to CSP solvers, MIP solvers are more effective in this type of problem [9].

The model uses the same sets and functions, as the CSP model described in section 4.4. We reuse constrained variables X^g_{dp}, Y^g_{dp} and Z^g_d and add the following new variables:

- $W^g_d \in \{0, 1\}$, a decision variable representing whether the demand d for a goods g is at least partially resolved.
- $U_{st} \in \{0, 1\}$, a decision variable representing whether the store s is overloaded at time t.

The MIP model consists of an objective function

$$\min \left\{ \alpha \sum_{g \in \mathbf{G}} \sum_{d \in \mathbf{D}} \sum_{p \in \mathbf{P}} X^g_{dp} Duration(p) - \beta \sum_{g \in \mathbf{G}} \sum_{d \in \mathbf{D}} \sum_{p \in \mathbf{P}} Y^g_{dp} \right\} \tag{4.5.1}$$

and following constraints:

$$\forall g \in \mathbf{G}, \forall s \in \mathbf{S}\colon \sum_{d \in \mathbf{D}} \sum_{p \in \mathbf{P}} Y^g_{dp} \leq Availability(g, s) \tag{4.5.2}$$

$$\forall g \in \mathbf{G}, \forall d \in \mathbf{D}, \forall p \in \mathbf{P}, \mathrm{Dest}(d) \neq \mathrm{Dest}(p)\colon Y^g_{dp} = 0 \tag{4.5.3}$$

$$\forall g \in \mathbf{G}, \forall d \in \mathbf{D}\colon Z_d^g = \sum_{p \in \mathbf{P}} Y_{dp}^g \tag{4.5.4}$$

$$\forall g \in \mathbf{G}\colon \sum_{\substack{d \in \mathbf{D} \\ \mathrm{Pri}(d)=\mathrm{high}}} Z_d^g = \min \left\{ \sum_{\substack{s \in \mathbf{S} \\ \mathrm{Availability}(g,s)>0}} \mathrm{Availability}(g,s); \sum_{\substack{d \in \mathbf{D} \\ \mathrm{Pri}(d)=\mathrm{high}}} \mathrm{Amount}(d,g) \right\} \tag{4.5.5}$$

$$\forall g \in \mathbf{G}, \forall d \in \mathbf{D}\colon Z_d^g \leq \mathrm{Amount}(d,g) \tag{4.5.6}$$

$$\forall g \in \mathbf{G}, \forall d \in \mathbf{D}, \forall p \in \mathbf{P}\colon Y_{dp}^g \leq 0 + \mu X_{dp}^g \tag{4.5.7}$$

$$\forall g \in \mathbf{G}, \forall d \in \mathbf{D}, \forall p \in \mathbf{P}\colon -Y_{dp}^g \leq -1 + \mu(1 - X_{dp}^g) \tag{4.5.8}$$

$$\forall g \in \mathbf{G}, \forall d \in \mathbf{D}\colon Z_d^g \leq 0 + \mu W_d^g \tag{4.5.9}$$

$$\forall g \in \mathbf{G}, \forall d \in \mathbf{D}\colon -Z_d^g \leq -1 + \mu(1 - W_d^g) \tag{4.5.10}$$

$$\forall g \in \mathbf{G}, \forall d \in \mathbf{D}, \exists d' \prec d, \mathrm{Pri}(d') = \mathrm{high}\colon$$

$$- \sum_{\substack{d' \in \mathbf{D} \\ d' \prec d \\ \mathrm{Pri(d')=high}}} Z_{d'}^g \leq - \sum_{\substack{d' \in \mathbf{D} \\ d' \prec d \\ \mathrm{Pri(d')=high}}} \mathrm{Amount}(d') + \mu(1 - W_d^g) \tag{4.5.11}$$

$$\forall s \in \mathbf{S}, \forall t \in \mathbf{T}\colon \mathrm{Load}(s,t) + \sum_{g \in \mathbf{G}} \sum_{d \in \mathbf{D}} \sum_{\substack{p \in \mathbf{P} \\ t \in \mathrm{Time}(p,s)}} Y_{dp}^g \leq \mathrm{Capacity}(s) - 1 + \mu U_{st} \tag{4.5.12}$$

$$\forall s \in \mathbf{S}, \forall t \in \mathbf{T}\colon \sum_{g \in \mathbf{G}} \sum_{\substack{d \in \mathbf{D} \\ \mathrm{Pri}(d)=\mathrm{low}}} \sum_{\substack{p \in \mathbf{P} \\ t \in \mathrm{Time}(p,s)}} Y_{dp}^g \leq 0 + \mu(1 - U_{st}) \tag{4.5.13}$$

Most of the constraints are the same as in CSP based model. We will not repeat their description and only add comments to the new constraints.

The equations (4.5.7) and (4.5.8) enforce the statement

$$Y_{dp}^g > 0 \Leftrightarrow X_{dp}^g = 1.$$

Here, we use a trick, which enables us to set a constraint in a way that only one of the two conditions is valid. It is either $Y_{dp}^g = 0 \wedge X_{dp}^g = 0$ or $Y_{dp}^g > 0 \wedge X_{dp}^g = 1$. First, we modify the conditions to contain non strict inequalities in the same direction: $Y_{dp}^g \leq 0 \wedge X_{dp}^g = 0$ or $-Y_{dp}^g \leq -1 \wedge X_{dp}^g = 1$. Then, we introduce an auxiliary constant μ with a large value, which is a coefficient for X_{dp}^{g} variable. When $X_{dp}^{g} = 0$, then the equation (4.5.7) becomes $Y_{dp}^{g} \leq 0$ and the equation (4.5.8) becomes $-Y_{dp}^{g} \leq -1 + \mu$. Since μ is a large number, the second constraint is always satisfied and only the first one is enforced. Similarly, when $X_{dp}^{g} = 1$ then the first constraint is always satisfied and the second one is enforced.

In the same way, the constraints (4.5.9) and (4.5.10) enforce the statement

$$Z_{d}^{g} > 0 \Leftrightarrow W_{d}^{g} = 1.$$

The variable W_{d}^{g} is used to "switch on" the constraint (4.5.11), which enforces the order of precedence of high priority order, i.e. the latter demand can be processed only if all previous high-priority demands are satisfied.

4.5.1 MIP solver

To solve MIP based problem, we use a state of the art solver, just like in the case of CSP based problem. We use solver SCIP [3], which is wrapped for Java language in OR-tools developed by Google [33]. The SCIP system is one of the fastest non-commercial solvers for mixed integer programming [30].

The SCIP system is freely available only for non-commercial use. As the OR-tools system provides a general interface for linear solvers, we can simply switch the underlying solver system. However, since the interface is general, it does not provide a way of sending information about the problem to the solver, as we did in case of CSP solver.

Chapter 5

Benchmarking

In this chapter, we experimentally evaluate the models defined in chapter 4. We focus on the quality of the output of the models and on the performance of the models.

5.1 Configuration of stores and deliveries

The configuration of stores and delivery routes used in benchmarks is presented on figure 5.1.1. There are six stores – central store (C), e-shop (E), Kovanecká street (K), passage Grossmann (G), Brno (B) and Plzeň (P). Three of them (central store, e-shop and Kovanecká street) are in one building. Therefore, the delivery is instant and can occur during the whole business day. The Grossmann store is located in Prague and its delivery is maintained by a courier company. Therefore, it is not instant but the goods is still delivered in the same day as it is shipped. It is shipped and delivered at a regular fixed time. Finally, other routes are inter-city routes which occur only once a day at a fixed time. The shipment and delivery times are specific for a particular store.

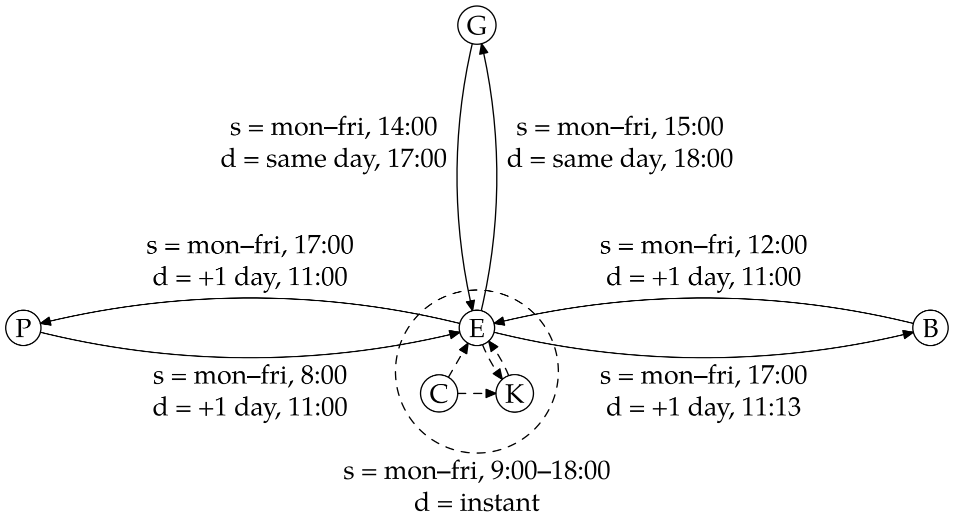

Figure 5.1.1. The stores and delivery routes used in benchmarking. The row *s* represents the conditions for a shipment and the row *d* contains the duration of the transfer and a corresponding delivery time along the path.

To compute a set of paths **P** between the stores, we use a Dijkstra's algorithm [15] on a directed graph displayed on figure 5.1.1. The paths are computed for each run once again, since the shortest path depends on a current simulation's date and time. The lengths of paths are computed using the algorithm D on page 34.

5.2 Performance analysis

In this section, we compare the performance of CSP-based model and MIP-based model under different circumstances. We used randomly generated data to compare MIP-based and CSP-based models.

5.2.1 Scalability

We measured the influence of the size of the input to the running time of the solver. With the same seed for the random numbers generator, we generated settings for benchmarks with the following parameters:

- Initial amount of goods: 100.
- Availability of each goods is a random number from range $[0, 50]$.
- Amount of demands: a random amount from range $[1, 10]$ for each goods. Each demand requests a random amount of goods from range $[1, 5]$
- Stores and paths between them: we used the layout described in section 5.1.

We run the benchmark in a loop and with each iteration, we increased the amount of goods by 100.

The results of the benchmarks are shown on figure 5.2.1. Both solvers run until they run out of memory. We set the Java heap space to 1500 MB. The MIP solver SCIP had another 1500 MB of system memory available.

5.2.2 CSP solution strategy

We tried various search strategies to figure out their impact on the solver performance. We run the benchmarks with the following parameters:

- Amount of goods: 1100.
- Availability of each goods is a random number from range $[0, 50]$.
- Amount of demands: a random amount from range $[1, 20]$ for each goods. Each demand requests a random amount of goods from range $[1, 5]$
- Stores and paths between them: we used the same layout as in the qualitative benchmark.

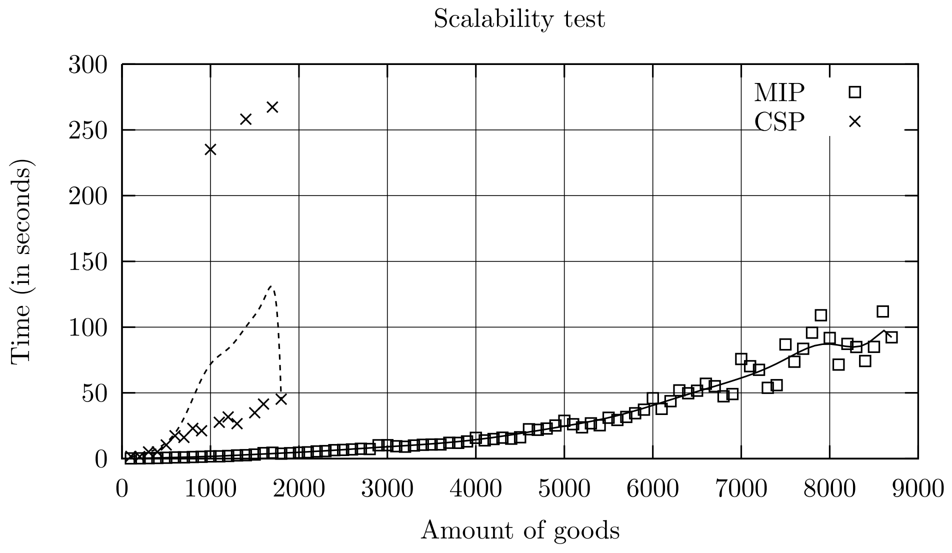

Figure 5.2.1. A running time of the model corresponding to the size of the input.

Solution strategy	**Running time [s]**
Default	23.799198
Minimum domain, lower bound	20.996632
Minimum domain, middle valued	**20.196771**
Minimum domain, upper bound	21.944280
Lexicographic order, lower bound	21.089646
Lexicographic order, upper bound	24.447447
Random variable, random value	20.900295

Table 5.2.1. A running time using various search strategies.

With these settings, at first, we did not specify any solution strategy. Thus, the solver used a default one as described in section 4.4.1. In other cases, we set the strategy for the Y_{dp}^{g} decision variables.

5.3 Qualitative analysis

In this benchmark, we let the Naïve algorithm, Ford-Fulkerson based algorithm, CSP-based model and MIP-based model deal with real data. In particular, we use data from 1st to 24th December 2013. Due to the increased pre-Christmas demand rate, this period is the most challenging part of the year. Moreover, this period is critical for the company Sunnysoft, and any failure would significantly lower the profit.

We perform a simulation of transfers using a simulator prepared for this purpose. The simulator performs a discrete event simulation. Based on the real data, it places demands, calls a planning procedure of the respective model (Naïve, Ford-Fulkerson, CSP-based and MIP-based). Based on the generated output, the simulator simulates the transfers as well as the replenishments of goods. A further description of a simulator is in the appendix D. The data used in the benchmark (i.e. demands, availability of goods and replenishments) are exported from the ERP system. The data files are described in appendix E.

For each planning algorithm, we run a simulation with a 1-minute, 5-minute and 30-minute-long time step.

We measure the following quantities:

- Average waiting time per goods in hours.
- Average waiting time per complete demand.
- Total amount of scheduled items.
- Total amount of resolved demands.

5.3.1 Experimental results

Based on the results of the performance analysis described above, we set the search strategy of CSP-based model to a *minimum domain middle value* and the time limit for search to be 1 minute. Other models were not modified.

	Naïve	**FF**	**CSP**	**MIP**
Average waiting time per goods [hour]	39.32	39.88	**39.25**	39.52
Average waiting time per demand [hour]	39.38	40.62	**38.51**	40.20
Total items scheduled	3752	3876	**4479**	3925
Total demands resolved:	1840	1882	**2162**	1898

Table 5.3.1. Qualitative analysis results. Parameters: no low-priority demands, time step = 1 hour

	Naïve	**FF**	**CSP**	**MIP**
Average waiting time per goods [hour]	**38.15**	39.62	38.25	38.79
Average waiting time per demand [hour]	38.30	38.80	**37.27**	39.24
Total items scheduled	3751	3873	**4268**	3945
Total demands resolved:	1836	1885	**2075**	1915

Table 5.3.2. Qualitative analysis results. Parameters: no low-priority demands, time step = 30 minutes

	Naïve	**FF**	**CSP**	**MIP**
Average waiting time per goods [hour]	**38.09**	39.65	38.40	39.05
Average waiting time per demand [hour]	38.25	38.80	**37.38**	38.50
Total items scheduled	3750	3873	**4187**	4010
Total demands resolved:	1826	1892	**2043**	1941

Table 5.3.3. Qualitative analysis results. Parameters: no low-priority demands, time step = 5 minutes

5.4 Stores overload

The main advantage of ALP-lp is the ability to minimize the overload of stores by dropping some of the low-priority orders. By generating the low-priority orders, we compare the models ability to prevent the overloads.

The input of benchmark is a combination of real data from the period 1st to 7th December 2013 and artificially generated low-priority demands, as described in section 5.4.1. The CSP-based model and MIP-based model were started with a time limit to find a solution set to 1 minute.

Both models use a function Day returning the identifiers of the days, when goods sent along a particular path is processed in a particular store. In the benchmark, if the original store of a path differs from the destination store, we assume, that the goods is processed in all stores on the path. However, if the original store of a path equals the destination store, the goods is not processed at all, since it is not handled by the operators, but the demand is resolved only with a virtual action in the ERP system. Therefore, the function Day returns an empty set.

The ALP-lp multiple-criteria optimization problem. Therefore, the models introduce parameters α and β to set the weights of the goals. In the benchmark, we vary the values of the parameters.

We measure the following quantities:

- Total low-priority demands scheduled.
- Total high-priority demands scheduled.
- Total days overloaded. It is a sum of days, when the store was overloaded, for each store (i.e. when the store s_1 was overloaded for two days and store s_2 was overloaded for three days, the total days overloaded would be five days).

We restricted the search time of solvers to one minute. Moreover, in case of the MIP-solver, we performed another benchmark with the search time of solver restricted to 5 minutes.

5.4.1 Low-priority demands

The low priority demands are generated once a day between 6:00 and 6:59. For each goods g the simulator computes the expected available amount of goods after resolving all of the demands. It uses the following formula:

$$ExpectedTotalAvailability_g = \sum_{s \in \mathbf{S}} InStock_{g,s} - \sum_{d \in \mathbf{D}} Demanded_{g,d}, \tag{5.4.1}$$

where $Demanded_{g,s}$ means the amount of goods g demanded by the demand d. Afterwards, we compute the desired amount on the stores. Since we want to balance the amounts of goods equally, we use the following formula

$$DesiredAmount_g = \left\lceil \frac{ExpectedTotalAvailability_g}{|\mathbf{S}|} \right\rceil, \tag{5.4.2}$$

and generate the low-priority demand d for each store s with

$$Demanded_{g,s} = DesiredAmount_g.$$

5.4.2 Experimental results

			CSP				
			$\alpha = 1$			$\alpha = 10$	$\alpha = 100$
	Naïve	**FF**	$\beta = 1$	$\beta = 10$	$\beta = 100$	$\beta = 1$	
LP	5444	23427	0	0	0	0	0
HP	201	282	525	529	530	520	518
DO	44	49	22	22	22	22	22

	MIP (1 min)				
	$\alpha = 1$			$\alpha = 10$	$\alpha = 100$
	$\beta = 1$	$\beta = 10$	$\beta = 100$	$\beta = 1$	
LP	5	4	5	1	0
HP	297	292	292	292	295
DO	16	14	14	14	12

Table 5.4.1. Stores overload benchmark results. **LP** means total amount of low-priority demands scheduled, **HP** means total amount of high-priority demands scheduled, and **DO** means the total amount of days, when a store was overloaded.

	MIP (5 min)				
	$\alpha = 1$			$\alpha = 10$	$\alpha = 100$
	$\beta = 1$	$\beta = 10$	$\beta = 100$	$\beta = 1$	
LP	3	2	5	0	0
HP	297	292	292	295	295
DO	16	14	14	12	12

Table 5.4.2. Results of the overloading benchmark, when the solver's time limit is set to 5 minutes. **LP** means total amount of low-priority demands scheduled, **HP** means total amount of high-priority demands scheduled, and **DO** means the total amount of days, when a store was overloaded.

5.5 Discussion

As we expected, the MIP-based model has far better performance than the CSP-based model. The problem contains several constraints, which are very poorly propagated by the CSP solvers. As shown in figure 5.2.1, the MIP-based model is able to handle larger problems while remaining quite efficient. We can recommend the CSP-based model only if the time needed to find the feasible solution does not matter.

In the qualitative analysis, both models proved that they are able to generate a better schedule than simple algorithms like Naïve algorithm and Ford-Fulkerson based algorithm. Contrary to the expectation, the CSP-based model generated better results than the MIP-based model. Still, the results of the MIP-based model are good enough to be used in a real application. As the results show, if we use the CSP-based model, it is better to use a time step of length of 1 hour. In the case of MIP solver, the best results were achieved with a 5-minute step.

The comparison of various search strategies for the CSP model showed us, that the most suitable search strategy is a selection of a variable with the minimum domain. The strategy assigns the variable a current middle value from its domain, i.e. $x = \frac{upper\_bound(D_x) - lower\_bound(D_x)}{2}$. Still, the differences between the strategies are not significant.

The CSP-based model does not generate good results if the low-priority demands are involved. The solver is not able to find a good solution during the given time limit. In case of the MIP-solver, the results are better. For a comparison, we tried to increase the search time limit for the MIP solver from 1 minute to 5 minutes. As we can see in table 5.4.2, the achieved results do not differ significantly.

Based on the benchmarks, we can recommend to use the MIP-based model as a planner of transfers in the company Sunnysoft.

However, we must notice the unexpected crushes of MIP solver. The solver fills the memory of a computer until it uses it completely and then it crashes, which affects the whole application, as well. Since the MIP solver is wrapped into Java programming language through a SWIG interface over a pre-compiled binary program, it is out of control of the program. It does not throw an exception or does not use any other mechanism to allow the application to recover from the state. Moreover, the Java garbage collector works in a different way. Normally, we would create a new `MPSolver` instance each time we perform a planning. After the return of the generated plan from the planning method, the solver instance would be disposed of by the garbage collector. However, in this case, it remains in the memory. Therefore, the process slowly fills the whole memory and then crashes. As a result, we need to hold the solver in memory all the time and clear its content. The second consequence is that the planning software needs an external watchdog, since it is not reliable and cannot control itself, as the crash of the solver leads to the crash of the whole application.

Still, if we take measures to avoid, or at least minimize, the above mentioned issues, it is very reasonable to use the MIP solver approach.

Chapter 6

Conclusion

In this thesis, we dealt with a real problem of a retail company Sunnysoft. First, we analysed the problem and discussed the currently used approaches to solve it. Afterwards, we reviewed existing models and solution strategies. From these techniques, we chose the CSP based approach inspired by [36]. We noticed, that the particular model would be more effective, if stated as Mixed integer programming problem. The benchmarks performed on the solvers confirmed, that the MIP-based model has a better performance on this particular problem.

As a result, we proved, that the CSP-based model is reasonable, especially on the provided real data. However, for this particular problem, MIP-based approach is more suitable due to its better performance.

As a next step, the MIP-based model should be tested in real use. Unfortunately, we did not perform a loading test of the model in the company. However, we performed the simulation on real data. Even though the real application always slightly differs from the simulation, we expect the model to be reasonably functional.

The planning of transfers is only one half of the whole problem. The second one is a prediction of the future needs. The goods should be allocated to stores not only as a result of current absence of goods, but it should be distributed across the stores based on the estimation of sales. Currently, the company has information about the history of sales stored in ERP. Moreover, the company can use information from other sources like price comparing websites. For example, the site Heureka.cz provides information about the price range of a particular goods in other companies. Based on these data, we can build a prediction software, which can be used both for optimal allocation of goods as well as for generating orders for goods from suppliers.

Bibliography

[1] Choco. [online, available at http://www.emn.fr/z-info/choco-solver/; accessed 2014-06-29], 2014.

[2] Tobias Achterberg. *Constraint Integer Programming*. Doctoral thesis, Technischen Universität Berlin, 2007.

[3] Tobias Achterberg, Timo Berthold, Thorsten Koch, and Kati Wolter. Constraint Integer Programming: A New Approach to Integrate CP and MIP. In *Integration of AI and OR techniques in constraint programming for combinatorial optimization problems*, volume 01, pages 6–20. Springer, Berlin, Heidelberg, 2008.

[4] Kemal Altinkemer and Bezalel Gavish. Parallel savings based heuristics for the delivery. *Operations Research*, 39(3):456–469, 1991.

[5] Jonathan F. Bard, Liu Huang, Moshe Dror, and Patrick Jaillet. A branch and cut algorithm for the VRP with satellite facilities. *IIE Transactions*, 30(9):821–834, September 1998.

[6] Roman Barták. Omezující podmínky: od sudoku po vesmírné aplikace. In *Umělá inteligence (5)*, pages 146–172. Academia, Praha, 2007.

[7] J. Christopher Beck, Patrick Prosser, and Evgeny Selensky. On the reformulation of vehicle routing problems and scheduling problems. In S Koenig and R Holte, editors, *LNAI 2371, Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation, SARA 2002*, pages 282–289. Springer, 2002.

[8] Olli Bräysy and Michel Gendreau. Vehicle Routing Problem with Time Windows, Part I: Route Construction and Local Search Algorithms. *Transportation Science*, 39(1):104–118, February 2005.

[9] Jordan Carsten and Andreas Drexl. A Comparison of Constraint and Mixed-Integer Programming Solvers for Batch Sequencing with Sequence-Dependent Setups. *ORSA Journal on computing*, 7(2):160–165, 1995.

[10] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search : A New Framework for Game AI. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 216–217, 2006.

[11] G. B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In T. C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. Wiley & Chapman-Hall, New York - London, 1951.

[12] G. B. Dantzig and J. H. Ramser. The Truck Dispatching Problem. *Management science*, 6(1):80–91, 1959.

[13] Rina Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[14] Guy Desaulniers, Jacques Desrosiers, Yvan Dumas, Marius M. Solomon, and François Soumis. Daily Aircraft Routing and Scheduling. *Management Science*, 43(6):841–855, June 1997.

[15] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[16] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[17] Michael R. Garey and David S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. Freeman & Comp., San Francisco, 1979.

[18] Michel Gendreau, Gilbert Laporte, and René Séguin. Stochastic vehicle routing. *European Journal of Operational Research*, 88(1):3–12, 1996.

[19] Fred Glover. Tabu Search - Part I. *ORSA Journal on computing*, 1(3):190–206, 1989.

[20] Fred Glover. Tabu Search - Part II. *ORSA Journal on computing*, 2(1):4–32, 1990.

[21] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[22] Ralph E. Gomory. Solving Linear Programming Problems in Integers. *Combinatorial Analysis*, 10:211–215, 1960.

[23] Malte Helmert. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2):219–262, February 2003.

[24] Petr Holub, Miloš Liška, Hana Rudová, and Pavel Troubil. Comparison of CP and IP Techniques for Data Transfer Planning. In *28th Workshop of the UK Special Interest Group on Planning and Scheduling*, pages 69–70, 2010.

[25] ISO. Data elements and interchange formats – Information interchange – Representation of dates and times. Standard 8601:2004, International Organization for Standardization, 2004.

[26] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, New York, 1972.

[27] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[28] Tishya Leong, Peter Shor, and Clifford Stein. Implementation of a combinatorial multicommodity flow algorithm. *Network Flows and Matching: First DIMACS Implementation Challenge*, 00:387–406, 1993.

[29] J. T. Linderoth and M.W.P. Savelsbergh. A Computational Study of Search Strategies for Mixed Integer Programming. *INFORMS Journal on Computing*, 11(2):173–187, 1999.

[30] H. Mittelmann. Mixed Integer Linear Programming Benchmark (MIPLIB2010). [online, available at http://plato.asu.edu/ftp/milpc.html; accessed 2014-07-27], 2014.

[31] Manfred Padberg and Giovanni Rinaldi. A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems. *SIAM Review*, 33(1):60–100, 1991.

[32] Christos H. Papadimitriou. On the complexity of integer programming. *Journal of the Association for Computing Machinery*, 28(4):765–768, October 1981.

[33] Laurent Perron. Operations research and constraint programming at Google. In *Principles and Practice of Constraint Programming - CP 2011*, page 2. Springer, Berlin Heidelberg, 2011.

[34] Lluís Ros, Tom Creemers, Evgueni Toutouta, and Jord Riera. A global constraint model for integrated roteind and scheduling on a transmission network. In *7th International Conference on Information Networks, Systems and Technologies*, pages 40–47, Minsk, 2001.

[35] Stuart Russell and Peter Norwig. Classical Planning. In *Artificial Intelligence. A Modern Approach*, pages 366–400. Prentice-Hall, Upper Saddle River, third edition, 2010.

[36] H. Simonis. Constraint applications in networks. In *Handbook of Constraint Programming*, volume 2, pages 875–903. Elsevier, 2006.

[37] Paolo Toth and Daniele Vigo. Models, relaxations and exact approaches for the capacitated vehicle routing problem. *Discrete Applied Mathematics*, 123:487–512, November 2002.

[38] Otakar Trunda and Roman Barták. Using Monte Carlo Tree Search to Solve Planning Problems in Transportation Domains. In Félix Castro, Alexander Gelbukh, and Miguel Mendoza Gonzáles, editors, *LNAI 8266, Proceedings of the 12th Mexican International Conference on Artificial Intelligence 2013*, pages 435–449. Springer, 2013.

[39] Michal Tuláček. *Constraint Solvers*. Bachelor thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2009.

Appendix A

List of abbreviations

For each abbreviation, we present its meaning and the page where it is first used.

acVRP	Asymmetric cVRP 21
ALP	Automated logistics problem 14
ALP-lp	Automated logistics problem with low-priority demands 16
CCP	Chance constrained program 19
CSP	Constraint satisfaction problem 23
cVRP	Capacitated vehicle routing problem 19
DARSP	Daily aircraft routing and scheduling problem 19
ERP	Enterprise resource planning software 9
IP	Integer programming 30
MAC	Maintaining arc consistency algorithm 24
MCTS	Monte Carlo Tree Search 20
MIP	Mixed integer programming 29
PSA	Parallel savings based heuristics algorithm 20
scVRP	Symmetric cVRP 21
SPR	Stochastic program with recourse 19
SVRP	Stochastic vehicle routing problem 19
VRP	Vehicle routing problem 19
VRPSC	Vehicle Routing Problem with Stochastic Customers 19
VRPSCD	Vehicle Routing Problem with Stochastic Customers and Demands 19
VRPSD	Vehicle Routing Problem with Stochastic Demands 19
VRPSF	Vehicle routing problem with satellite facilities 21
VRPTW	Vehicle routing problem with time windows 19

Appendix B

Contents of the CD

- *thesis.pdf* – electronic version of this thesis.
- *benchmarks.tar.gz* – benchmarks outputs.
- *sources* – sources of the application.
- *dist* – binaries of the simulator and models, compatible with Java version 8.
  - *dist/lib/mac* – a Google OR-Tools shared library for Apple computers
  - *dist/lib/linux* – a Google OR-Tools shared library for Linux 64 bit.

Appendix C

Implementation notes

The simulator and model implementations are programmed in Java programming language. We used a CSP solver Choco in version 3.2.0 available at `http://www.emn.fr/z-info/choco-solver/index.php?page=choco-3` and MIP solver SCIP in version 3.1.0 wrapped in the Google OR-tools, SVN revision 3723 available at `https://code.google.com/p/or-tools/source/checkout`.

C.1 Constraint solver

The constraint solver is a Java object of type `Solver`. The model is formulated with constrained variables and constraints of various types, posted in the solver. The Choco system does not distinguish between the model variables and solution variables. The problem solving consists of the following steps:

1) Creation of a `Solver` class instance.
2) Definition of constrained variables. The variables are created with a reference to the solver object.
3) Creation of constrains and their posting in the solver.
4) Solution search performed by the model. It can either search for any solution or it can search for the optimal solution with respect to the given objective function.
5) Reading of the values of constrained variables, defined in Step 2.

C.1.1 Variables and views

In our model, the variables have the following domains: $\mathbb{N}_0$, $\mathbb{Z}$, $\{0, 1\}$ and $\{y\}$. For the first two domains, we use a bounded `IntVar` variables. These are integer variables whose domain is an interval $[min, max]$. However, we cannot use an infinite bound. Instead, we bound the variable with the maximum possible value. However, we cannot use a Java constant `Integer.MAX_VALUE` (whose value is $2,147,483,647$) but we must use a Choco specific constant `Variable-Factory.MAX_INT_BOUND` with a value $21,474,836$. This value is high enough for our purposes. The variables with domain $\{0, 1\}$ are represented by objects of class `BoolVar`. Finally, the variables with single value domain, also called "fixed" variables, are constructed as `IntVar`. The variables are construed with the following methods of a `VariableFactory` class:

- `BoolVar bool(String name, Solver s)`
- `IntVar bounded(String name, int min, int max, Solver s)`
- `IntVar fixed(int value, Solver s)`

The solver offers simplified form for declaring variables, which are in the form yX or $(X + y)$, i.e. they are derived from an existing constrained variable and are multiplied or shifted with a constant. Indeed, we could write these variables as a combination of a new constrained variable and a constraint, for example declare a variable Y and post a constraint $Y = yX$. However, the Choco system offers an easier way. We can define a special variable type *view*, which is connected to a previously defined constrained variable or to another view. We use three types of view, an *offset* $(X = Y + z)$, a *scale* $(X = zY)$ and a *minus* $(X = -Y)$. The views can be used whenever we can use a normal constrained variable.

C.1.2 Constraints

To describe constraints, we use the following notation:

- $X, Y, \ldots$ for constrained variables
- $x, y, \ldots$ for constants.
- $\varphi, \psi, \ldots$ for constraints.
- $\cdot \square \cdot$ for a binary relation (i.e. $<, \leq, =, \geq, >, \neq$)

For all of the following constraints, the relation variable `op` has one of the following values: `<` for $<$, `<=` for $\leq$, `=` for $=$, `>=` for $\geq$, `>` for $>$ and finally `!=` for $\neq$.

The constraints are represented by Java objects of a generic type **Constraint**. Therefore, we can combine them as we did, for example, in the case of the constraint (3.1.6).

In the model, there are the following types of constraints:

$$X \square y \tag{3.1.1}$$

$$X \square Y \tag{3.1.2}$$

Both constraints can be formulated using the `IntConstraintFactory` as follows:

- `Constraint arithm(IntVar var1, String op, IntVar var2)`
- `Constraint arithm(IntVar var1, String op, int constant).`

$$\sum X \square Y \tag{3.1.3}$$

There are two constraints for the constraint (3.1.3). The first is a simplified form for the case of $\square \sim =$ and the second one is for general relation operator $\square$. The constraints are provided by `IntConstraintFactory`:

- `Constraint sum(IntVar[] vars, IntVar sum)`
- `Constraint sum(IntVar[] vars, String op, IntVar sum)`

$$\sum X \square y \tag{3.1.4}$$

The Choco system does not contain a constraint which allows us to directly formulate the constraint (3.1.4). Instead, we can define a dummy constrained variable D and post two constraints $\sum X \square D, D = y$. The second way, which is used in our implementation, is to introduce a dummy fixed constrained variable D with a domain $\{y\}$ and then post a constraint $\sum X \square D$.

$$\bigwedge \varphi \tag{3.1.5}$$

The constraint (3.1.5) is written using the `LogicalConstraintFactory` constraint `and` as follows:

- `Constraint and(Constraint... constraints)`

$$\varphi \Rightarrow \psi \tag{3.1.6}$$

The implication constraint (3.1.6) is based on the `LogicalConstraintFactory` constraint `ifThen` with the following signature:

- `Constraint ifThen(Constraint ifCon, Constraint thenCon)`

$$\varphi \Leftrightarrow \psi \tag{3.1.7}$$

In Choco, there is no special constraint for the equivalence relation. However, we can simply rewrite it as two implications: $(\varphi \Rightarrow \psi) \wedge (\varphi \Leftarrow \psi)$.

C.1.3 Solutions

The solver is able to run in two modes. In the first one, it searches for any solution, which is feasible for the given sets of variables and constraints. The application can request the solutions iteratively one after another, or all together. In the second case, the solver can find an optimal solution with respect to a given objective function. The objective function value is represented by a constrained variable. The solver receives the constrained variable and a direction of the optimization, i.e. whether the final value of the variable should be minimum or maximum.

However, there is one problem with the optimal solution finder. Compared to "find any solution" strategy, which clearly indicated, that there is at least one feasible solution (otherwise it returns `false`), the optimal solution by itself does not indicate, that it did not find anything. We have to specially ask the solver, whether the problem is feasible or not.

C.2 Mixed integer programming solver

The Google OR-tools provide a general interface for mixed integer programming problem formulation. Simultaneously, it wraps the following MIP solvers, both commercial and free:

- GPLK
- CLP
- CBC
- SCIP
- Sulum
- Gurobi

The solver is an object of type `MPSolver` which calls methods of particular solver binary. The binary is contained in an external dynamic library, generated by OR-tools. Since the solver itself is not under control of the Java program, and the garbage collector operates on this object differently, we advise to reuse the solver object between the simulations and clear it with the `MPSolver.clear()` method.

Just like in the case of CSP solver, the model is construed from variables and constraints. There are three types of variables available – real numbers, integers and binary numbers (i.e. 0 and 1). The variables are construed using the factory methods on the solver object:

- `solver.makeBoolVar(String name);`
- `solver.makeIntVar(double lower, double upper, String name);`
- `solver.makeVar(double lower, double upper, String name);`

which returns the instances of the `MPVariable` class.

There is only one type of a constraint:

$$\sum_i c_i x_i \in [low, up],$$

where x_i are the constrained variables and c_i are their respective coefficients. Naturally, we can use it as follows:

- When we set $low = -\infty$, we get the constraint $\sum_i c_i x_i \leq up$.
- When we set $up = \infty$, we get the constraint $\sum_i c_i x_i \geq low$.
- Finally, when we set $up = low$, we get the constraint $\sum_i c_i x_i = low$.

The value of ∞ is provided by the solver with a method `solver.infinity()`. First, we state the bounds of the constraints and call a factory method on `MPSolver` object, which returns an object of type *MPConstraint*:

```
    MPConstraint c = solver.makeConstraint(low,up);
```

Afterwards, we set the non-zero coefficients:

```
    c.setCoefficient(variable, constant);
```

The objective function is set in a similar manner. By calling the `solver.` `.objective()` method, we get the `MPObjective` object. The object contains a method `setCoefficient` which can be used in the same way as in the case of `MPConstraint` object, described above. The `MPObjective` object allows us to specify, whether we want to minimize or maximize the value of an objective function.

Finally, we call method `solver.solve()` which starts the search for the best solution. If the solution is found, the values of variables are available through a method `solutionValue()` on their respective objects.

Appendix D

Description of a simulator

As we described in chapter 2, the whole process is controlled by documents. Therefore, based on the data in ERP, we are able to establish the *InStock* values for each store in any given moment. Moreover, we have the demands represented by shipment orders and also warehouse receipts which indicate if and when there was an increase in the amount of goods in a given store.

In order to test our algorithm, we created a simulator which generates demands for the planner and applies the generated plan.

1) *Intialization*. For a given date d, the simulator sets $Availability_{g,s} = InStock_{g,s}$ based on the history of stock supplies. Then, it prepares a sequence of demands $\mathbf{D}$ and a sequence of store replenishments $\mathbf{R}$, both ordered by date, where the date of the first element of both sequences is after d. We introduce a sequence of shedules $\mathbf{Sch} = \emptyset$.
2) *Pre-planning phase*. In this phase, the simulator sets $d' = d$ and increases d by a predefined step. Then, it constructs a sequence $\mathbf{D}' \subset \mathbf{D}$ of demands, which occurred since d'. Next, it updates an $Availability_{g,s}$ with replenishments from $\mathbf{R}$ which occurred since d' and removes them from $\mathbf{R}$. Similarly, it processes schedules in $\mathbf{Sch}$. For each schedule $sch \in \mathbf{Sch}$ which is planned between d' and d, the simulator lowers the corresponding $OnTheWay_{g,s}$ and increases $Availability_{g,s}$. Then it computes a set of shortest paths $\mathbf{P}$ between the stores with shipment time d.
3) *Planning*. The simulator runs the selected planner algorithm with $\mathbf{D}'$ and $\mathbf{P}$ as an input, producing a set of plans $\mathbf{Pl}$.
4) *Update phase*. Each plan $pl \in \mathbf{Pl}$ contains an information about the original store s_o, the destination store s_d, goods to be transferred s_o, its *amount* and a reference to a demand d, which is resolved by the plan. According to the plan, the simulator decreases the requested amount of goods g in d. If the demand d is completely resolved, i.e. it does not demand any goods, it is removed from $\mathbf{D}$. Then, it decreases $Availability_{g,s_o}$ and increases $OnTheWay_{g,s_d}$ by *amount*. Finally, it adds a new schedule to $\mathbf{Sch}$ for transfer of *amount* goods g to store s_d with a scheduled time based on the shortest path in $\mathbf{T}$ between s_o and s_d. Afterwards, it continues with phase 2).

D.1 Output of the simulator

The simulator generates a CSV file containing the scheduled plans. The file contains the following fields:

1) Row header, always `@s`.
2) Simulation date of generating of plan.

3) Estimated date of arrival of the goods to the destination store.
4) Id of the goods.
5) Id of the original store.
6) Id of the destination store.
7) Date of the placing of the demand.
8) Amount of goods scheduled for the demand.
9) Priority of the demand. Possible values are `Hi` for high-priority demands and `Lo` for low-priority demands.
10) Indication, whether the demand was fully resolved. Possible values are `1` for yes and `0` for no.

Appendix E

Description of data files

We used real data for benchmarking. The data are generously provided by the company Sunnysoft and contain information about supplies and demands in fiscal year 2013/14. The data files are located on the CD provided with this thesis. All files are in an XML format described bellow. We use a `teletype` script to denote `<tags>` and their `parameters`.

E.1 Demands

- File name: *demands.xml*
- Root element: `<demands>`
- For each demand there is an element `<demand>` with a parameter `date` containing a date of placing the demand in ISO 8601 format [25] and a parameter `store` with the id of a store, where the demand is placed.
- For each item of the demand, there is an element `<item>` with a parameter `goods` which contains an id of requested goods and a parameter `amount` containing the demanded amount of goods.

  An example of *demands.xml*:

```
<?xml version="1.0"?>
<demands>
  <demand date="2013-07-01T00:02:15+02:00" store="e">
    <item goods="541741" amount="1" />
    <item goods="524796" amount="1" />
    <item goods="527972" amount="1" />
  </demand>
  <demand date="2013-07-01T00:27:48+02:00" store="e">
    <item goods="538318" amount="1" />
    <item goods="537576" amount="1" />
  </demand>
    ...
</demands>
```

E.2 Goods and the history of stock availability

- File name: *goods.xml*
- Root element: `<goods>`
- For each article, there is an element `<article>` with a parameter `id` containing the id of the article.

- For each event in the stock availability history, there is an element `<history>` with a parameter `date` containing the date of the event in ISO 8601 format [25].
- For each store, there is an element `<store>` with a parameter `store` containing the id of the store, a parameter `onStock` containing the amount of the article in the store and a parameter `onTheWay` containing the amount of an article which is currently being transferred from other stores.

An example of *goods.xml*:

```
<?xml version="1.0"?>
<goods>
  <article id="520003">
    <history date="2014-07-01T00:00:00+02:00">
      <store store="c" onStock="0" onTheWay="0" />
      <store store="k" onStock="0" onTheWay="0" />
      <store store="b" onStock="0" onTheWay="0" />
      <store store="g" onStock="0" onTheWay="0" />
      <store store="e" onStock="0" onTheWay="0" />
      <store store="p" onStock="0" onTheWay="0" />
    </history>
  </article>
  <article id="520848">
    <history date="2013-07-04T11:48:37+02:00">
      <store store="c" onStock="0" onTheWay="0" />
      <store store="k" onStock="1" onTheWay="0" />
      <store store="b" onStock="3" onTheWay="0" />
      <store store="g" onStock="3" onTheWay="0" />
      <store store="e" onStock="2" onTheWay="0" />
      <store store="p" onStock="2" onTheWay="0" />
    </history>
    <history date="2013-07-09T11:51:12+02:00">
      <store store="c" onStock="0" onTheWay="0" />
      <store store="k" onStock="1" onTheWay="0" />
      <store store="b" onStock="3" onTheWay="0" />
      <store store="g" onStock="3" onTheWay="0" />
      <store store="e" onStock="1" onTheWay="0" />
      <store store="p" onStock="2" onTheWay="0" />
    </history>
    ...
  </article>
  ...
</goods>
```

E.3 Stores

- File name: *stores.xml*
- Root element: `<stores>`
- For each store, there is an element `<store>` with a parameter `id` containing the identifier of the store used in other files and a parameter `capacity` denoting a maximum daily amount of goods processed in the store.

  An example of *stores.xml*:

```
<?xml version="1.0"?>
<stores>
  <store id="c" capacity="50" />
  <store id="k" capacity="50" />
  <store id="b" capacity="50" />
  ...
</stores>
```

E.4 Deliveries

- File name: *deliveries.xml*
- Root element: `<deliveries>`
- For each delivery, there is an element `<delivery>` with mandatory parameters `from` denoting the original store, `to` denoting the destination store, `time` denoting a time or a time range of possible shipment, `day` denoting a day of week or a range of days of week of possible shipment (0 = Sunday, 1 = Monday etc.), and a `type` which can be *instant* if the delivery does not take any notable length of time (for example if it is only an accountancy operation[4] or from one room to another) or *carrier*, if the goods is shipped by an external delivery company. In such case, there are two other parameters. The first is a parameter `duration` with the estimated duration of a transfer in days and the second is `delivery_time` with the time of expected delivery to the store.

  An example of *deliveries.xml*:

```
<?xml version="1.0"?>
<deliveries>
  <delivery from="c" to="k" time="9:00-17:00" day="1-5"
    type="instant" />
  <delivery from="b" to="e" time="12:00" day="1-5"
    type="carrier" duration="1" delivery_time="11:00" />
  ...
```

[4] The stores can be either some physical entity like a dedicated room, or it can be only a virtual entity in the ERP system. In such case, we can move goods from store to store just by generating respective documents without any physical contact with the goods.

```
</deliveries>
```

E.5 Storings

- File name: *storings.xml*
- Root element: `<storings>`
- Constains information about the replenishment of the stores. For each storing event there is an element `<storing>` with a parameter `store`, a parameter `goods` containing an id of the replenished goods, a parameter `amount` containing the amount of replenished goods, and a parameter `date` containing the date of the event in ISO 8601 format [25].

  An example of *storings.xml*:

```
<?xml version="1.0"?>
<storings>
  <storing store="c" goods="539895" amount="1"
                     date="2013-07-01T10:26:27+02:00" />
  <storing store="c" goods="540804" amount="1"
                     date="2013-07-01T10:26:27+02:00" />
  <storing store="c" goods="542243" amount="4"
                     date="2013-07-01T10:26:27+02:00" />
  <storing store="c" goods="536648" amount="3"
                     date="2013-07-01T10:44:47+02:00" />
  ...
</storings>
```