Optimization on Home Care

Kirstine Thomsen

Summary

The purpose of this project is to investigate how methods from operational research can be applied in the home care sector. A problem very similar to the VRPTW arises, when scheduling the routes for the caretakers. The problem is very complex, and hence it is simplified in this project. The conditions are limited to include the time windows of the visits, the working hours of the caretakers, visits locked to caretakers and if two caretakers share a visit. In a shared visit two caretakers have to start and finish the visit at the same time. The aim is to minimize the travelling time and maximize the number of visits, which are attended by a regular caretaker.

An intelligent insertion heuristic is applied on the problem. The insertion heuristic uses the regret measure to evaluate where the best insertion position is. The solutions found via the insertion heuristic are used as initial solutions for a tabu search, which allow infeasible solutions.

The results show, that when maximizing the number of visits with a regular caretaker, the total travelling time is likely to increase. The initial solutions found by the insertion heuristic are improved by the tabu search up to 27 %.

The solutions are compared with solutions found by a programme called ABP. The ABP incorporates more wishes and conditions. The results show, that the solutions found by the methods investigated in this project are better in all cases.

The conclusion drawn from this project is that it is possible to get high quality initial solutions by applying an intelligent insertion heuristic, if the only some of all the wishes and conditions are fullfilled. These high quality solutions can also be improved by applying the tabu search method used in this project.

Keywords: Home care, operational research, insertion heuristic, tabu search, VRPTW, time windows, working hours, shared visits, travelling time.

Resumé (in Danish)

Formålet med dette projekt er at undersøge hvordan metoder fra operationsanalyse kan anvendes i hjemmehjælpssektoren. Et problem, som ligner VRPTW, opstår, når hjemmehjælpernes ruter skal planlægges. Problemet er meget komplekst, og derfor er det simplificeret i dette projekt. Betingelserne er begrænsede til at omfatte tidsvinduer, hjemmehjælpernes arbejdstider, besøg låst på hjemmehjælpere og hvis to hjemmehjælpere deler et besøg. Ved et delt besøg skal to hjemmehjælpere starte og afslutte besøget samtidig. Målet er at minimere vejtiden og maksimere antallet af besøg foretaget af en fast hjemmehjælper.

En intelligent indsættelsesheuristik er anvendt på problemet. Indsættelsesheuristikken bruger et fortrydelsesmål til at vurdere hvor den bedste indsættelsesposition er. Løsningerne fundet vha. indsættelsesheuristikken er brugt som initielle løsninger i en tabusøgning, som tillader ugyldige løsninger.

Resultaterne viser, at når antallet antallet af besøg med en fast hjemmehjælper maksimeres, stiger den totale vejtid i de fleste tilfælde. De initielle løsninger fundet med indsættelsesheuristikken er forbedret op til 27 % med tabusøgningen.

Løsningerne er sammenlignet med løsninger fundet af et program med navnet ABP. Programmet ABP inkorporerer flere ønsker og betingelser. Resultaterne viser, at løsningerne fundet vha. de undersøgte metoderne i dette projekt er bedre i alle tilfælde.

Fra dette projekt kan man drage den konklusion, at det er muligt at få løsninger af god kvalitet ved at anvende en intelligent indsættelsesheuristik, hvis kun nogle af de mange ønsker og betingelser er opfyldte. Disse løsninger kan forbedres ved at anvende tabusøgningsmetoden i dette projekt.

Nøgleord: hjemmehjælp, operationsanalyse, indsættelsesheuristik, tabusøgning, VRPTW, tidsvinduer, arbejdstider, delte besøg, vejtid.

Preface

This report is the result of a M.Sc. project made by the under signing. The project is worth 35 European Credit Transfer System (ECTS) credits and is prepared in the period from September 2005 until March 2006 at the Department of Informatics and Mathematical Modelling (IMM) at the Technical University of Denmark (DTU). The supervisors are Jesper Larsen at IMM and René Munk Jørgensen at the Centre for Traffic and Transport (CTT), DTU.

The project is carried out with support from the company Zealand Care, who delivered data and knowledge, whenever I needed it. I would like to thank them for their great help.

I would like to thank my supervisors for many inspiring discussions and very useful advises. They both showed great interest in the project, which was very encouraging. They helped me so much.

I also thank my colleagues at the office for a wonderful time with good laughs and help. They were the best company one could imagine, when writing this thesis.

I thank the people, that I share a kitchen with at my residence hall, who prepared meals for me in the weeks before the dead line and listened to my eternal talking about the project.

I thank my family, who was in my thoughts through the whole project. They have been very supportive as always.

My uncle died in 1988 in a traffic accident, while writing his master thesis at DTU. I dedicate this project to him and his mother, my grandmother, who died in 2004. She has been my inspiration.

Lyngby, March 2006
Kirstine Thomsen

List of Symbols

This list contains all the symbols used in this report with their meaning and the page numbers for their first occurrence.

α is the latest starting time, page 45

β is the price for violating the same starting times for shared visits , page 45

γ is the price for violating the latest working time, page 45

δ is the modification factor for the prices α, β and γ, page 46

θ is the number of iterations, where it is tabu to reinsert a visit in a route during the tabu search, page 52

λ is the diversification factor, page 53

μ is the price for letting a non-regular caretaker visit a citizen, page 11

ω_{ij} is a binary variable, which indicates whether the two visits i and j form a shared visit, page 11

ϕ_r^i is a binary variable, which indicates whether the visit i is locked to caretaker r, page 9

Ψ is the number of unlocked visits without a regular caretakers. The shared visit only contribute by 1 in Ψ, if the two caretakers attending the shared visit are not regular, page 16

ρ is the number of times visit v has been inserted in the route r during the local search, page 53

σ_z^i is a binary variable, which indicates whether citizen z is the citizen at visit i, page 9

τ_z^o is a binary variable, which indicates whether caretaker o is regular at citizen z, page 7

$A(x)$ is the total violation of the latest starting times for visits in the solution x, page 45

a_i is the earliest starting time at visit i, page 9

$\hat{A}(x)$ is the total violation of the time windows for the visits in the solution x, page 55

$B(x)$ is the total violation of equal starting times for shared visits in the solution x, page 45

b_i is the latest starting time at visit i, page 9

$\hat{B}(x)$ is the total violation of the equal starting times for shared visits in the solution x, page 56

$C(x)$ is the cost of a feasible solution x, page 12

$c_1(v, r, p_r)$ is the additional cost, when inserting the not shared visit v in route r at position p_r, page 29

$c_1(v, r_1, r_2, p_{r_1}, p_{r_2})$ is the additional cost, when inserting the shared visit v in the distinct routes r_1 and r_2 at the positions p_{r_1} and p_{r_2}, page 31

$c_2(v)$ is the regret measure for inserting visit v, page 29

d_i is the duration of visit i, page 9

f_i is the finishing time at visit i, page 11

$G(x)$ is the total violation of the latest finishing times for routes in the solution x, page 45

$\hat{G}(x)$ is the total violation of the working hours in the routes in the solution x, page 55

g_o is the earliest starting time for caretaker o, page 6

h_o is the latest finishing time for caretaker o, page 6

i is a visit in the set $\mathcal{V}$, page 8

j is a visit in the set $\mathcal{V}$, page 8

l_i is the arrival time at visit i, page 11

m is the total number of routes in the set $\mathcal{R}$, page 9

n is the number of visits in the set $\mathcal{V}$, page 8

$N(x)$ is the neighbourhood to the solution x, page 43

n_r is the number of visits in route r, page 9

$\mathcal{O}$ is the set of caretakers, page 6

o is a caretaker in the set $\mathcal{O}$, page 6

p is a position in a route, page 24

$P(x)$ is the penalty function in the solution x, page 53

PF_i The push forward of the starting time for visit i, page 27

$\mathcal{R}$ is the set of routes, page 9

r is a route in the set $\mathcal{R}$, page 9

$\mathcal{S}$ is the solution space for the problem, page 12

s_i is the starting time at visit i, page 11

T is the total travelling time in a solution, page 16

$t(v, r, p_r)$ is the additional travelling time, when inserting visit v in route r at position p_r, page 28

$t_{z_1 z_2}$ is the transportation time between citizen z_1 and z_2 , page 8

$\bar{\mathcal{V}}$ is the set of the not scheduled visits, page 23

$\mathcal{V}$ is the set of visits, page 8

$\underline{\mathcal{V}}$ is the set of the scheduled visits, page 23

v is a visit in the set $\mathcal{V}$, page 8

w_i is the waiting time at visit i, page 11

x is a solution in the solution space $\mathcal{S}$, page 12

x_{ijr} indicates whether caretaker r goes from visit i to visit j, page 15

$\mathcal{Z}$ is the set of citizens, page 7

z is a citizen in the set $\mathcal{Z}$, page 7

Contents

Chapter 1

Introduction

"As long as possible in your own home"

(In Danish *"Længst muligt i eget hjem"*)

-A slogan in Danish social politics, 1987

1.1 Motivation and History

The elderly people make up an increasing proportion of the populations in the western world caused by an increasing average life age. This is a conclusion drawn from the development over the last century. The social systems in the western countries do therefore experience a larger demand for help to the elderlies. These demands are mainly payed via the taxes and as the proportion of tax payers is not increasing there is an imbalance.

In Denmark the situation is very similar according to the Danish Statistics Bank [Ban]. Hundred years ago in 1905 the proportion of persons with an age above 64 years was 6,62 %, and in 1955 it was 9,68 %. Last year in 2005 the proportion was 15,01 %. The forecast for the future years shows the same tendency, because the proportion is forecasted to be 18,54 % in 2015.

The Danish social politics is aware of the increasing demand for help to elderlies. The home care system was introduced in 1958 according to [Soc] when it became possible for retirement pensioners and handicapped to receive home care. Now it was possible for the elderlies to choose between receiving help in their own home or at a rest home. In 1987 the government introduced a senior residence reform to build more elderly residences. The new slogan was "as long time as possible in your own home".

The home care system in Denmark is organized and administrated by the municipalities. Typically every municipality is divided into small areas called districts and each area is serviced by a group of caretakers. There are different types of caretakers e.g. helpers, assistants, nurses and unskilled. They perform the *visits* in the homes of the *citizens* needing help. Administrating the social care includes more hours for planning the *routes* for the caretakers, because the demand for care is increasing, and also because planning the routes is a complex problem.

One of the ways to handle the increasing demand is by using operational research (OR). There are many problems in the administration of social care where OR can be applied. Until today there has though been a lack of research on that subject, see also section 2.5. The reasons for the lack of research could be many. OR is normally applied in technical areas e.g. production, whereas home care is considered a nontechnical area. For this reason there might have been ignorance of OR in the home care system and also the other way around there might have been ignorance of home care in the OR research environment.

As a student at the Technical University of Denmark, I got very interested in the world of OR, and therefore I expected to make my final master thesis in this area. In my holidays I had a very challenging and experience-giving job as a substitute caretaker in the home care during 6 years from 2000 until 2005. For this reason I saw the opportunity for combining my two interests in the thesis. By chance I found out that Jesper Larsen and René Munk Jørgensen were making a programme for scheduling the caretakers' routes. See more on their programme in chapter 5. They were luckily willing be supervisors in this project.

1.2 Purpose of the Project

The aim for this project is to investigate how methods from OR can be applied on home care.

One goal of this project is to refine the existing solution methods and develop new ones. To presentate and investigate the problem better, the problem is simplified and the horizon of planning is shortened to one day.

A new insertion method is developed on the basis of an already known insertion heuristic where the assignment of visits to caretakers and the generation of routes is done in the same process. The solution found by the insertion heuristic will form an initial solution for a variant of tabu search.

The tabu search has proved to be very efficient in other similar problems and is therefore applied on the problem in this project.

Jesper Larsen and René Munk Jørgensen have as mentioned in the previous section developed a programme for automatic scheduling the caretakers' routes and it is an aim to compare the problem and the methods in this project with their programme.

Another focus of the project is on the trade-off, that can arise, when one both wishes to minimize the travelling time and maximize the number of regular caretakers visiting the citizens. One should have the opportunity to adjust which weight the two wishes have proportional to each other.

The different developed solution methods are at the end compared with respect to solution quality and running times. The comparison is made between the developed methods in this project and the method in the programme developed by Jesper Larsen and René Munk Jørgensen.

1.3 Outline of the Report

Chapter 2 introduces the problem treated in this project by describing the problem and its mathematical formulation. The chapter also goes into previously written literature on similar problems.

The insertion heuristic is described in chapter 3, where a small example illustrates how the heuristic works.

A variant of the tabu search heuristic is introduced in chapter 4 with descriptions of different variations inside the heuristic.

Chapter 5 describes the problem and the methods in the programme developed by Jesper Larsen and René Munk Jørgensen and the deviations from the limited problem in this project.

Chapter 6 shows the results of the different comparisons between methods in this project and the methods in the programme described in chapter 5.

Chapter 7 discusses the use of OR in the home care system.

There are many issues left after this project to be investigated and these are introduced in chapter 8.

Chapter 9 gives the final conclusions on the project.

Chapter 2

The Vehicle Routing Problem with Time Windows and Shared Visits

The vehicle routing problem with time windows and shared visits (VRPTWSV) is described in section 2.1. The mathematical model for the problem is given in section 2.2 and the complexity of the problem is found in section 2.3. The problem is commented in section 2.4 and a literature review on similar problems is given in section 2.5.

2.1 Description of the Problem

The problem and its different elements are described in this section. Many of the symbols used in the remaining part of the report are also introduced in this section.

2.1.1 What is a Caretaker ?

There are different types of *caretakers*, which depend on the caretakers' education, e.g. nurses, assistants and unskilled.

The set of caretakers in the problem is symbolized by $\mathcal{O}$, where o is a caretaker in the set $\mathcal{O}$.

Each caretaker has an earliest starting time g_o, and a latest finishing time h_o. The time interval $[g_o, h_o]$ is also named the *working hours* for caretaker o.

2.1.2 What is a Citizen ?

The *citizens* in this problem are elderlies and handicapped, who receive help at home payed by the government. The municipalities are administrating the help. Before a citizen can receive help, the citizen is examined to determine how much and how often the citizen needs help.

All the citizens in the problem form the set $\mathcal{Z}$, where the citizen z is one of the citizens in the set.

Figure 2.1 shows how the citizens could be situated. Notice that the *administration office* is also considered a citizen in this problem.

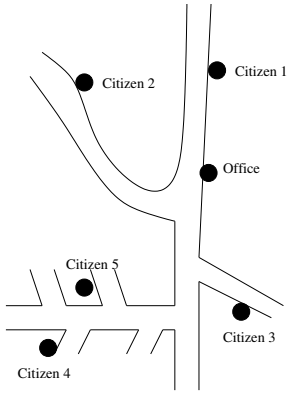


Figure 2.1: How the citizens could be situated at a map.

Each citizen has at least one caretaker as a contact person. A contact person is in this project also called a *regular caretaker*. In most cases it is preferable that regular caretakers attend the visits at the citizens, because when an elderly person meets a new person often in his home, he may feel insecure. It also eases the job for the caretakers, because they get to know the procedures after being at the citizen many times and can perform better. Another and very important reason for having a regular caretaker is to have somebody responsible for keeping an eye on how the situation evolves at the citizen, see more on this topic in chapter 7. The parameter τ_z^o indicates whether caretaker o is regular at citizen z.

$$\tau_z^o = \begin{cases} 1 & \text{if the caretaker } o \text{ on is regular at citizen } z \\ 0 & \text{elsewise} \end{cases}$$

No regular caretakers are assigned to the office.

The travelling time between two citizens z_1 and z_2 are measured in minutes and given as the parameter $t_{z_1 z_2}$.

2.1.3 What is a Visit?

A *visit* is performed when a caretaker visits a citizen to help him. For instance a visit can include helping an elderly person taking a shower in the morning and helping her making the breakfast. A visit can also include helping a young handicapped by cleaning her home. The help needed is also called a *demand*, and some demands require special qualifications from the caretakers. One could imagine a situation, where a citizen needs to get his medicine for the next week dosed into a box with 7 separate holes. Only the caretakers educated in medicine are allowed to do this job. For other types of demands some caretakers are more qualified than others. For instance a assistant is more qualified than the unskilled for helping citizens suffering from dementia.

The set of visits is $\mathcal{V}$, where $v \in \mathcal{V}$. Often another notation is used in this report, where $i \in \mathcal{V}$ or $j \in \mathcal{V}$. The number of visits in $\mathcal{V}$ is n.

Sometimes the caretakers have to visit the same citizen more than one time the same day. In figure 2.2 is a sketch of how the citizens and visits one day could be situated along a road network. The numbers of the visits are given to clearify that this example is a situation with more visits than citizens. Compare this figure to 2.1 to find the corresponding citizens. In the example in figure 2.2 citizen 4 and 5 both have two visits one day and citizen 1 and 2 have three visits one day, while citizen 3 only has one visit. The office has 4 visits, because two caretakers visit the office twice in this example.

All caretakers have at least one visit at the office during a day. There could be several reasons for visiting the office. One reason could be to check in and out from work. Another reason could be to get and hand in a special key for *key boxes* at the office. In most municipalities they have a very special key system. There is a key box by the citizens, that can not or do not wish to open their door themselves. In a key box is the key to the front door. In some municipalities it is possible to take the key home after work, but in others it isn't, because the staff will not risk loosing the key. The key can

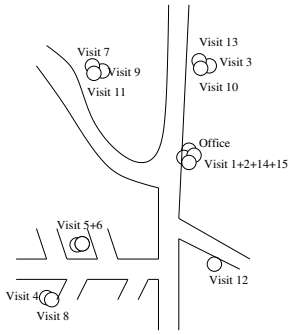


Figure 2.2: How the visits could be situated on a map.

open the front door in hundreds of homes, and therefore it is handled with care. A final reason for visiting the office is to have a break.

A visit at the office can be *locked* to a caretaker to ensure for instance that the caretaker has a break.

$$\phi_r^i = \begin{cases} 1 & \text{if visit } i \text{ is locked to caretaker } r \\ 0 & \text{elsewise} \end{cases}$$

To every visit i belongs a citizen z. The relationship between a visit and a citizen is given by the parameter σ_z^i.

$$\sigma_z^i = \begin{cases} 1 & \text{if citizen } z \text{ is the citizen at visit } i \\ 0 & \text{elsewise} \end{cases}$$

For each of the visits $i \in \mathcal{V}$ a *time window* $[a_i, b_i]$ is determined, where a_i is the earliest starting time and b_i is the latest starting time.

Each visit has a fixed *duration* d_i, which is given in minutes.

2.1.4 What is a Route?

A *route* consists of a sequenced group of visits. Figure 2.3 illustrates how visits can form two routes.

The arrows in figure 2.3 only indicate the order of the visits, and they do not show the physical path taken between the visits. There are two routes in the left sketch in figure 2.3, and they are separated in the boxes to get a better overview.

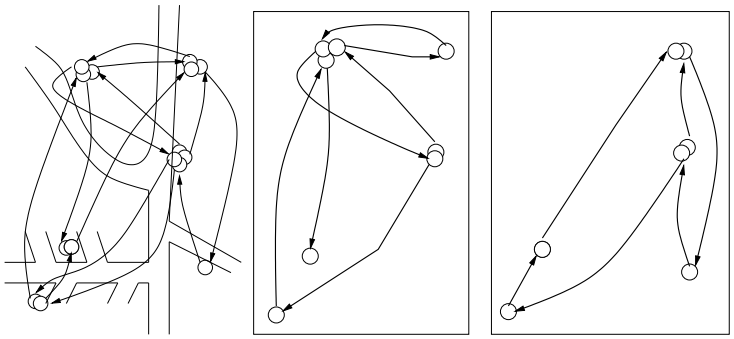


Figure 2.3: How visits can form a route.

The set of routes is $\mathcal{R}$, where route $r \in \mathcal{R}$. The total number of routes is m. The number of visits in the route r is n_r.

Each caretaker takes one route. The caretaker having route r is also referred to as r in the remaining part of this report, except in chapter 8.

2.1.5 What is a Shared Visit?

There may arise special situations when taking care of elderly or handicapped citizens, and in some cases it is necessary to have two caretakers attending a visit. These visits are called *shared visits*. This could concern a citizen that has to be transfered with a lift between the bed and the wheel chair. The situation is shown at figure 2.4.

Figure 2.4: Two caretakers using the lift to transfer a citizen. Lars-Ole Nejstgaard has painted this picture, see [Nej].

As shown on the picture in figure 2.4 the situation with a lift requires two caretakers. A shared visit is split into two separate visits i and j with the

same time window, citizen and duration. In the general situation the two persons could start or finish at different times as long as they both are present while the lift is used.

The parameter ω_{ij} indicates whether two visits i and j constitute a shared visit.

$$\omega_{ij} = \begin{cases} 1 & \text{if visit } i \text{ and visit } j \text{ form a shared visit} \\ 0 & \text{elsewise} \end{cases}$$

2.1.6 The Limitations

The planning horizon for the problem is limited to one day to make it easier to manage the problem.

Another assumption in the problem is that all caretakers have the same qualifications, and the visits do not have different types of demands.

In this project it is also assumed, that two caretakers attending a shared visit start and finish at the same time to make the problem less complex and therefore easier to explore.

2.1.7 The Objective of the Problem

The objective of the problem is to minimize the total travelling time and the number of unlocked visits without a regular caretaker, when finding out which caretaker should attend which visit and in which order.

The value of having a regular caretaker attending a visit at a citizen is measured by the parameter μ. This price μ is given in minutes, and is the price per visit for letting a non-regular caretaker attend the visit at a citizen. This means that if you let a regular caretaker attend the visit, you save μ minutes. A way to interpretate this price is: for how much extra time would one let the regular caretaker travel compared to the non-regular caretaker to reach the citizen? In the case with a shared visit, only one of two caretakers needs to be regular in this problem to avoid paying the price μ.

2.1.8 Representation of a Solution

A way to represent a solution is by using a Gantt-diagram as in figure 2.5. The time indicators for arrival time, waiting time, starting time and finishing

time for visit $i \in \mathcal{V}$ are formalized with the mathematical terms l_i, w_i, s_i and f_i. Figure 2.5 illustrates the terms. The solution in figure 2.5 only has one route with two visits 1 and 2, where the first visit is at citizen z_1 and the second visit is at citizen z_2.

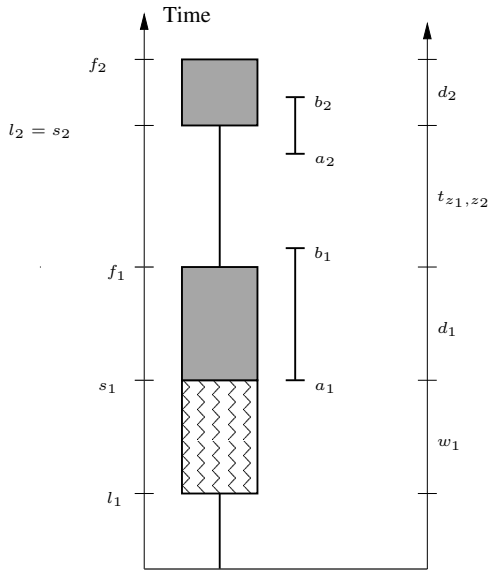


Figure 2.5: An example of the time schedule, where the grey boxes indicate working and the boxes with zigzag line are waiting time.

The caretaker on this route in figure 2.5 arrives at l_1 to visit 1 and waits for w_1 minutes until he starts at $s_1 = a_1$, because the time window opens at a_1. Visit 1 takes d_1 minutes and the caretaker finishes the visit at f_1. Afterwards the caretaker travels for t_{z_1,z_2} minutes until he reaches visit 2, where he can start immediately at $s_2 = l_2$, because the time window opened at a_2. The visit 2 lasts for d_2 minutes, and it is finished at f_2. Notice that all time windows are satisfied.

2.2 The Mathematical Model

The set of feasible solutions for the problem is named the *solution space* $\mathcal{S}$, where each solution in $\mathcal{S}$ is named x. Each feasible solution x has a cost $C(x)$.

The problem can be formulated as an *optimization problem*

$$\begin{aligned} &\min C(x) \\ &\text{Subject to} \quad x \in \mathcal{S} \end{aligned}$$

which can be written in the short form $\min\{C(x)|x \in \mathcal{S}\}$. The objective of the problem is to find the solution $x \in \mathcal{S}$, where the cost $C(x)$ is smallest.

The problem can the formulated as a mathematical flow model. The representation of the problem is a complete *graph* $G(\mathcal{V} \cup \mathcal{D}, \mathcal{E})$ with the vertices $\mathcal{V} \cup \mathcal{D}$ and the links $\mathcal{E}$. A solution is a set of arcs, that form routes, where all the nodes are contained in the routes.

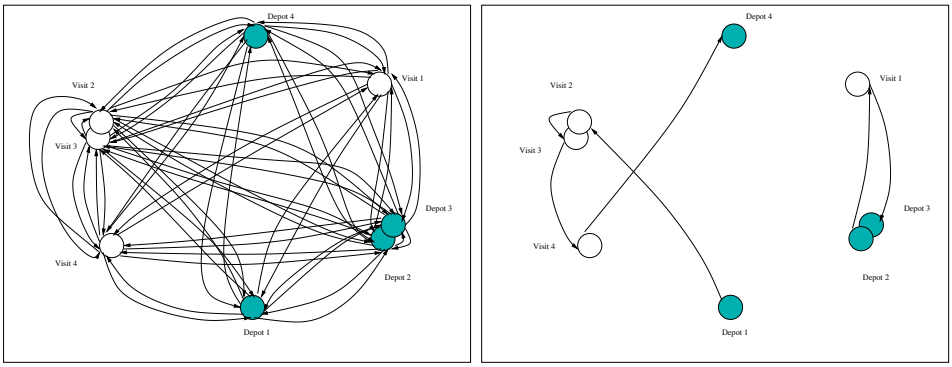


Figure 2.6: The problem and a solution represented in a graph. The colored nodes are depots, and the white nodes are visits.

To the left in figure 2.6 is an example of a problem. Between every pair of vertices are two links, one in each direction. To the right in figure 2.6 is a solution to the problem, where all the nodes are distributed in two routes. Notice that the nodes are divided into two groups $\mathcal{V}$ and $\mathcal{D}$; visits and depots. A depot is where a caretaker starts or ends his route, and in most cases it is at home. The depots are only considered in the mathematical formulation, but will not be taken into account in the remaining part of this report.

The mathematical model formulation is very alike the formulation for *the Vehicle Routing Problem with Time Windows* (VRPTW), which is an extension of *the Capacitated Vehicle Routing Problem* (CVRP). As the title of the problem indicates this problem concerns routing vehicles with a capacity, which is also the case in the VRPTW.

The mathematical model for VRPTW is introduced here to make a better comparison to the VRPTWSV.

The group of vehicles is $\mathcal{R}$, where $r \in \mathcal{R}$. Each vehicle r has the start depot D_r^+ and the end depot D_r^-. There is also a group of customers in the

problem situated at different locations. The group of customer is $\mathcal{V}$. Each of the customers should be serviced by one of the vehicles. Every customer $i \in \mathcal{V}$ has a demand q_i. The vehicle r has an upper limit Q_r for the capacity. Every customer i has a time window $[a_i, b_i]$ for the service. The objective of the VRPTW is to find the shortest total travelling time, where the travelling time between two customers i and j is t_{ij}.

The first decision variables in the VRPTW is

$$x_{ijr} = \begin{cases} 1 & \text{if the vehicle } r \text{ uses the link between customer } i \text{ and customer } j \\ 0 & \text{elsewise} \end{cases}$$

which indicates if the vehicle r travels from customer i to customer j. Another decision variable is y_{ir} which is the load on the vehicle r when arriving to customer i. The last decision variable is s_i, which is the starting time at customer i.

$$\min \sum_{i\in\mathcal{V}} \sum_{j\in\mathcal{V}} \sum_{r\in\mathcal{R}} c_{ijr} x_{ijr} \tag{2.1}$$

$$\begin{aligned}
\sum_{r\in\mathcal{R}} \sum_{j\in\mathcal{V}} x_{ijr} &= 1 \quad \forall\, i \in \mathcal{V} && (2.2)\\
\sum_{j\in\mathcal{V}} x_{D_r^+ jr} &= 1 \quad \forall\, r \in \mathcal{R} && (2.3)\\
\sum_{i\in\mathcal{V}} x_{iD_r^- r} &= 1 \quad \forall\, r \in \mathcal{R} && (2.4)\\
\sum_{i\in\mathcal{V}} x_{ikr} - \sum_{j\in\mathcal{V}} x_{kjr} &= 0 \quad \forall\, k \in \mathcal{V},\ r \in \mathcal{R} && (2.5)\\
y_{D_r^+ r} &= 0 \quad \forall\, r \in \mathcal{R} && (2.6)\\
y_{D_r^- r} &= 0 \quad \forall\, r \in \mathcal{R} && (2.7)\\
y_{ir} &\leq Q_r \quad \forall\, k \in \mathcal{V},\ r \in \mathcal{R} && (2.8)\\
y_{ir} + q_i - M(1 - x_{ijr}) &\leq y_{jr} \quad \forall\, i, j \in \mathcal{V},\ r \in \mathcal{R} && (2.9)\\
s_i + t_{ij} - M(1 - x_{ijr}) &\leq s_j \quad \forall\, i, j \in \mathcal{V},\ r \in \mathcal{R} && (2.10)\\
s_i &\geq a_i \quad \forall\, i \in \mathcal{V} && (2.11)\\
s_i &\leq b_i \quad \forall\, i \in \mathcal{V} && (2.12)
\end{aligned}$$

$$\begin{aligned}
y_{ir} &\geq 0 \quad \forall\, i \in \mathcal{V},\ r \in \mathcal{R} && (2.13)\\
s_i &\geq 0 \quad \forall\, i \in \mathcal{V} && (2.14)\\
x_{ijr} &\in \{0,1\} \quad \forall\, i, j \in \mathcal{V},\ r \in \mathcal{R} && (2.15)
\end{aligned}$$

Constraint (2.2) ensures that every customer is serviced, and (2.3) and (2.4) ensure that each vehicle r leaves its start depot D_r^+ and enters its end depot D_r^- exactly once. The flow balance is kept by (2.5), where the flow into a customer also should leave the customer. The load y_{ir} at the start depot D_r^+ and at the end depot D_r^- is set to zero for each vehicle r in (2.6) and (2.7). The load should not exceed the capacity Q_r for each vehicle r, which is guaranteed by constraint (2.8). The constraint (2.9) describes how the load on a vehicle r is evolving. The starting time s_i is found for every customers i using constraint (2.10), where $s_j \geq s_i + t_{ij}$ if the vehicle r travels from customer i to j. The next constraints (2.11) and (2.12) ensure the starting time s_i for every customer i to be within the time window $[a_i, b_i]$.

The VRPTWSV is an extended version of the VRPTW, where the customers correspond to the visits $\mathcal{V}$ and the vehicles correspond to the caretakers $\mathcal{R}$. The extension includes the shared visits, working hours and locked visits, but the VRPTWSV does not involve capacity.

The decision variables are

$$x_{ijr} = \begin{cases} 1 & \text{if the caretaker on route } r \text{ uses the link between visit } i \text{ and } j, \\ 0 & \text{elsewise,} \end{cases}$$

s_i is the starting time at visit i and f_i is the finishing time at visit i .

$$\begin{aligned} \min \sum_{r\in\mathcal{R}} \sum_{i,j\in\mathcal{V}} \sum_{z_1,z_2\in\mathcal{Z}} \sigma_{z_1}^i \sigma_{z_2}^j t_{z_1 z_2} x_{ijr} + \qquad (2.16) \\ \mu \sum_{r\in\mathcal{R}} \sum_{i,j\in\mathcal{V}} \sum_{z\in\mathcal{Z}} (1-\phi_r^i)(1-\tau_z^r)(1-\omega_{ij})\sigma_z^i x_{ijr} + \\ \frac{\mu}{2} \sum_{r_1,r_2\in\mathcal{R}} \sum_{i,j\in\mathcal{V}} \sum_{k_1,k_2\in\mathcal{V}} \sum_{z\in\mathcal{Z}} (1-\phi_{r_1}^i)(1-\phi_{r_2}^j)(1-\tau_z^{r_1})(1-\tau_z^{r_2})\omega_{ij}\sigma_z^i x_{ik_1r_1} x_{jk_2r_2} \\ = \min C(X) = \min T + \mu\Psi \end{aligned}$$

$$\sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{V}} x_{ijr} = 1 \quad \forall\, i \in \mathcal{V} \tag{2.17}$$

$$\sum_{j\in\mathcal{V}} x_{D_r^+ jr} = 1 \quad \forall\, r \in \mathcal{R} \tag{2.18}$$

$$\sum_{i\in\mathcal{V}} x_{iD_r^- r} = 1 \quad \forall\, r \in \mathcal{R} \tag{2.19}$$

$$\sum_{i\in\mathcal{V}} x_{ikr} - \sum_{j\in\mathcal{V}} x_{kjr} = 0 \quad \forall\, k \in \mathcal{V},\ r \in \mathcal{R} \tag{2.20}$$

$$s_i + d_i = f_j \quad \forall\, i \in \mathcal{V} \tag{2.21}$$

$$s_i \omega_{ij} \leq s_j \quad \forall\, i, j \in \mathcal{V} \tag{2.22}$$

$$x_{ijr} s_i \geq g_r \quad \forall\, i, j \in \mathcal{V},\ r \in \mathcal{R} \tag{2.23}$$

$$x_{ijr} f_j \leq h_r \quad \forall\, i, j \in \mathcal{V},\ r \in \mathcal{R} \tag{2.24}$$

$$f_i + \sum_{z_1 z_2} \sigma_{z_1}^i \sigma_{z_2}^j t_{z_1 z_2} - M(1 - x_{ijr}) \leq s_j \quad \forall\, i, j \in \mathcal{V},\ r \in \mathcal{R} \tag{2.25}$$

$$s_i \geq a_i \quad \forall\, i \in \mathcal{V} \tag{2.26}$$

$$s_i \leq b_i \quad \forall\, i \in \mathcal{V} \tag{2.27}$$

$$\sum_{j\in\mathcal{V}} \phi_r^i x_{ijr} = 1 \quad \forall\, i \in \mathcal{V}\ r \in \mathcal{R} \tag{2.28}$$

$$s_i \geq 0 \quad \forall\, i \in \mathcal{V} \tag{2.29}$$

$$x_{ijr} \in \{0, 1\} \quad \forall\, i, j \in \mathcal{V},\ r \in \mathcal{R} \tag{2.30}$$

The objective function in the mathematical model for VRPTWSV is different from the objective function in VRPTW. It is split into three parts.

The first part concerns the total travelling time between the citizens corresponding to the visits.

The second part concerns the number of unlocked and unshared visits, where a non-regular caretaker is attending the visit. Splitting up the term gives that $(1-\phi_r^i) = 1$, when visit i is not locked by caretaker r, and $(1-\tau_z^r) = 1$ when caretaker r is not regular at citizen z. The visits i are not shared when $(1-\omega_{ij}) = 1$ for all visit $j \in \mathcal{V}$. The citizen z is the citizen at visit i, when $\sigma_z^i = 1$ and the visit i is actually attended by caretaker r, when $x_{ijr} = 1$ for all visits $j \in \mathcal{V}$.

The third and last part of the objective function concerns the number of shared and unlocked visits, where neither of the caretakers r_1 and r_2 attending the shared visit are regular. This is the case, when $(1-\tau_z^{r_1})(1-\tau_z^{r_2}) = 1$. Notice that there is only one citizen z for a shared visit. The price for a shared visit without any regular caretaker is μ, but because a shared visit

is divided into two visits, each shared visit appears twice in the objective function. The last part of the objective function is therefore multiplied by $\mu/2$.

The objective function can also shortly be written as $T + \mu\Psi$, where T is the total travelling time, and Ψ is the number of unlocked visits without a regular caretaker, and a shared visit only contributes by 1 in Ψ if none of the two caretakers attending the visits are regular.

The capacity constraints (2.6), (2.7), (2.8) and (2.9) from the VRPTW model are removed in the mathematical model for VRPTWSV.

All the other constraints in the VRPTW model are similar or the same with a new interpretation in the mathematical model for VRPTWSV. Constraint (2.17) ensures that every visit is done once by the caretaker r. Each caretaker r starts in a origin point D_r^+ and ends in the destination point D_r^-, which is ensured by the constraints (2.18) and (2.19). The flow balance constraint (2.20) ensures that when a caretaker comes to a visit k, he also leaves the visit. The constraint (2.25) determines the starting times of the visits. If the visit j is after visit i on route r, the starting time for visit j should be at least the finishing time for visit i plus the travelling time $t_{z_i z_j}$ between the citizen z_i for visit i and the citizen z_j for visit j. The constraint contains a sufficient big number M. It is sufficient big if it equals the last finishing time of all end depots. Every visit i has a hard time window $[a_i, b_i]$ for the starting time, and the constraints (2.26) and (2.27) ensure that the visit can only start within its time window.

One of the new constraints in the model is (2.22). If the two visits i and j constitute a shared visit, the constraint (2.22) ensures, that $s_i = s_j$. The caretakers r_1 and r_2 for the two visits i and j must be different, but this is already ensured because the two visits start at the same time and the constraint (2.25) ensures that a caretaker is not performing more than one visit at the same time.

Two other new constraints are (2.23) and (2.24), which ensure no violation of the earliest starting time g_r or latest finishing time h_r for each caretaker r. The last new constraint is (2.28) on the locked visits. If a visit i is locked to be attended by caretaker r, the constraint (2.28) makes sure it is satisfied.

2.3 Complexity

In this section the complexity theory is introduced and the complexity of the VRPTWSV is found. The complexity theory is introduced in chapter

6 in the book [Wol98] and it is the background for the introduction in this section.

There are two types of problems: the *optimization problem*

$$\min\{C(x)|x \in S\}$$

and its corresponding *decision problem*

$$\text{Is there a solution } x \in S \text{ with the value } C(x) \leq k?$$

The decision problems can have only two answers; a "yes" or a "no". There are different classes of decision problems.

$\mathcal{NP}$ is the class of decision problems, where it can be proven in polynomial time, that the answer is "yes", if this is the case. The optimization problem with a corresponding decision problem in the $\mathcal{NP}$ class can be solved by answering the decision problem a polynomial number of times.

$\mathcal{P}$ is the class of decision problems in $\mathcal{NP}$ that can be solved in polynomial time.

$\mathcal{NPC}$ is the class of $\mathcal{NP}$-*complete* decision problems, which is a subset of $\mathcal{NP}$. An instance of a problem in $\mathcal{NP}$ can be converted in polynomial time to an instance of a problem in $\mathcal{NPC}$. (The problems in $\mathcal{NP}$ are *polynomial reducible* to the problems in $\mathcal{NPC}$.) This means that the problems in $\mathcal{NP}$ are not more difficult than the problems in $\mathcal{NPC}$. The optimization problem with a corresponding $\mathcal{NP}$-complete decision problem is $\mathcal{NP}$-*hard*.

The big question is whether $\mathcal{P} = \mathcal{NP}$. If this is not the case then there exist a set $\mathcal{NP} \setminus \mathcal{P}$ with decision problems, which are not solvable in polynomial time. Figure 2.7 illustrates this situation.

The collary 6.1 at page 85 in [Wol98] says that if any $\mathcal{NP}$-complete decision problem is solvable in polynomial time ($\mathcal{P} \cap \mathcal{NPC} \neq \emptyset$) then $\mathcal{P} = \mathcal{NP}$. The proof for this corollary is also given. Firstly suppose that the decision problem Q is in the class $\mathcal{P} \cap \mathcal{NPC}$ and $R \in \mathcal{NP}$. R is polynomially reducible to Q, because $Q \in \mathcal{NPC}$ and by the definition of $\mathcal{NPC}$. The decision problem Q is also in $\mathcal{P}$, and because R is polynomially reducible to Q, then R must also lie in the class $\mathcal{P}$ according to lemma 36.3, page 931 in [CLR90]. This

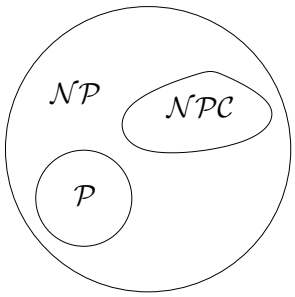


Figure 2.7: The decision problem classes, when $\mathcal{P} \neq \mathcal{NP}$

proves that $\mathcal{NP} \subset \mathcal{P}$ and because it is already known by the definition of $\mathcal{P}$ that $\mathcal{P} \subset \mathcal{NP}$ then $\mathcal{P} = \mathcal{NP}$

It immidiately follows that if $\mathcal{P} \neq \mathcal{NP}$ then all problems in $\mathcal{NPC}$ are not solvable in polynomial time ($\mathcal{P} \cap \mathcal{NPC} = \emptyset$) by using the fact that $A \Rightarrow B$ is logical equivalent to $\neg B \Rightarrow \neg A$.

It has not yet proven that $\mathcal{P} = \mathcal{NP}$ nor $\mathcal{P} \neq \mathcal{NP}$. For all the problems in $\mathcal{NPC}$ there have not yet been found polynomial solution methods, and hence it is so far assumed that $\mathcal{P} \neq \mathcal{NP}$.

The VRPTW is a *NP-hard* optimisation problem, and because the VRPTWSV is an extension of VRPTW, the VRPTWSV is also *NP-hard*.

2.4 Comments

The time between the arrival time and opening time of a visit is called waiting time, but it should not always be considered as waiting time, because the time often can be used efficiently at the previous citizen, or as individual time for the caretaker. In my point of view it is very important to avoid stress in the home care, because it affects some citizens in an unstable psychological condition. The stress also has a negative effect on the caretakers, who may have more days of illness, which is expensive to the employeers and tax payers. The waiting time shall for this reason not be considered totally as a waste of time.

There are different strategies for handling the shared visits. One option is to fix the starting times by setting the time windows sufficient tight, but this will decrease the solution space. Instead of splitting the visit up, one could also handle it as a single visit. The flow of caretakers through the node representing the visit will then be two in and two out.

The VRPTWSV satisfies, that each caretaker is not doing more than one visit at the time, but it does not satisfy that each citizen only has one visit at the time. This may in some situations not be appropriate, for instance if one caretaker comes to bandage a wound and another caretaker comes to give the citizen a bath. Often these two demands are put together in one visit, but not always, because the two demands may require very different caretaker skills as in the example mentioned. It is possible to model that some types of demands are not allowed to overlap at the same citizen, and only when it is a shared visit it is allowed to be more than more caretaker at the citizen.

2.5 Literature Review

The CVRP problem has been studied intensively over the last decades as a problem in the area of operational research, because it has many applications. The book [TV01] gives a good overview over CVRP with its applications, solution methods and variants such as VRPTW.

The special feature in VRPTWSV is the shared visit with two separate visits, which can be viewed as two operations starting at the same time and having the same duration. This special feature might be applicable in other areas than home care, for instance when synchronizing chemical processes. It is not possible to find any literature on this special feature. The applications are probably in areas not well known to me, because I do not think that this feature is only special in the home care sector.

To my knowledge there is little literature on the operational research methods developed for solving problems in the home care sector. The article [EFR03] is one of the few on this subject. The problem is formulated as a set partioning model. The constraints are very similar to the constraints in the VRPTWSV. The additional constraints are

- Each visit has demands which must be met by the caretakers qualified for the type of demands.
- Each caretakers has given working areas.
- Certain visits are grouped in such way that the same staff member must do all those visits.

The interesting fact is that there are also shared visits in their problem. In the article they also split up the shared visit in two parts, but they fix the

starting times of each part to satisfy the constraint on their starting times. This approach does not use the information on time windows for the shared visit.

It is allowable in their methods to violate some of the constraints. They operate with an individual penalty function for violating each type of constraint.

The method used to solve the problem is the *repeated matching method*.

Solving the first matching problem gives the next new matching problem and so forth until the method reach a stopping criterion. The matching problems are both solved by a heuristic and an exact method.

Initially when performing the repeated matching method there is one route for each visit and also one route without any visits for each caretaker. The routes with only one visit and no caretaker assigned are penalized very high to force them to match with routes with caretakers. When combining two routes all visits may go to one of the routes, or the visits can be mixed. The repeated matching method stops, when no improvements are achieved.

The heuristic for finding matchings performs better than the exact method in one case and the other way around in the other case.

The result is a saving about 20 % for the travelling time and the total working time is reduced by 7 % because of less planning time.

Another article on home care is [BF05]. The article defines the a mathematical problem, where both hard and soft time windows are declared for both visits and caretakers. The soft time window can be violated by paying a penalty.

The other properties in the model deviating from VRPTWSV are:

- The demands of the visits have to meet the qualifications of the caretakers
- A soft constraint on preferences on citizens or certain demands, which may be ignored by paying a penalty. The VRPTWSV only includes preferences on the regular caretakers.
- A fair distribution of the difficult demands over all the caretakers

The model in the article does not include shared visits.

The solution method is split in two parts: one for assigning the visits to the nurses and one for finding an optimal sequencing for the visits assigned to

each caretaker. The first part is performed by using a heuristic for finding an initial solution and two improvement heuristics. The second part is a LP-problem, which is solved by an exact method.

The preprocessing finds the possible caretakers for each visit, and if there is only one possible caretaker, the visit is assigned to her. The precedences of the visits are also determined according to the time window. This correspond to removing some of the directed arcs in a graph presentation of the problem.

The information on the precedences is used every time it is needed to find the ordering of the visits within a route. All feasible permutations are investigated to find the optimal solution to a LP problem. The optimal solution gives the best starting times of the visits in a route given their order.

Constraint programming is used for finding an initial solution, where the visits are assigned to caretakers. The improvement heuristics used are *Simulated Annealing* and *Tabu Search*, where the move consists of removing a visit from one position and inserting it in another position. The other position may be in another route.

The methods are tested in different ways, both combined and separately. The tabu search heuristic turns out to perform better than the simulated annealing. The combination of the constraint programming with the tabu search produces the best solutions for the test instances.

Chapter 3

The Insertion Heuristic

A heuristic is a method for finding a good solution to a problem. The set of feasible solutions for a problem is called a *solution space*. The heuristic does not go through all solutions in the solution space, and the solution found at the end is not guaranteed to be optimal.

This insertion heuristic is an extension of the method proposed in [PJM93], where the routes are filled in parallel instead of filling the routes one by one. When performing a insertion, it is possible to insert in one of all the routes instead of not being able to choose the insertion route. The parallel insertion method has to be extended because of the shared visits in the VRPTWSV problem and the limited working hours for the caretakers.

The number of routes is fixed to m and there are n visits to insert in these routes. The set of visits is $\mathcal{V}$ and it is divided into two subsets. The subset $\bar{\mathcal{V}}$ contains all the visits not scheduled and the subset $\underline{\mathcal{V}}$ contains all the scheduled visits. Each route r correspond to a caretaker r. The fixed number of routes m is found by looking at a previous found schedule and using the number of the caretakers in that situation. If this number is not sufficient, it is necessary to add extra caretakers until there are enough to cover all visits. The number of visits on each route r is n_r.

Initially all m routes contain at least one *seed-visit*. The seed-visits are the visits at the office, which are locked to a certain caretaker $r \in \mathcal{R}$. The set $\underline{\mathcal{V}}$ contains the seed-visits initially. Notice that the depots D_r^+ and D_r^- (which could be represented as the start and end visits in the homes of the caretakers) in the mathematical model (2.16) - (2.30) would have been seed-visits. These depots are not present in the data used for the results in chapter 6. The visits at the office can not count as depots in the mathematical model of the problem, because they are not necessarily in the beginning and the end of the routes e.g. a break is an intermediate visit on a route.

Each initial route r with seed-visits forms a *partial constructed route* r, the visits are named $i_0, \ldots, i_{n_r-1}$. Initially all routes are feasible, because none of the time windows nor working hours are violated and if there are any shared visits, they start at the same time. A insertion is feasible if it does not affect the feasibility of all routes.

The question in every iteration of the heuristic is which visit v should be inserted, in which route r^* and in which *position* p_{r^*} ? Which visit v to insert in which route r is evaluated by calculating the extra travelling time and adding extra time μ if the caretaker r^* is not regular at visit v's citizen z_v. If the insertion is not feasible, the extra time is set to a sufficiently high number. The objective is to minimize the extra time, and therefore it is chosen in each iteration to use the cheapest combination of v, r^* and p_{r^*} in comparison with other combinations for insertion.

The number of a candidate position corresponds to the position number the visit would have, if it was inserted. If visit v is a candidate for position p it will be placed between visit i_{p-1} and i_p. Figure 3.1 illustrates an example of this. Because the index of the positions starts with 0, the last visit has index $n_r - 1$. The number of candidate positions in route r can lie in the interval 0, 1, ..., n_r, because it is possible to insert a visit as the first or last visit in the route or as an intermediate visit.

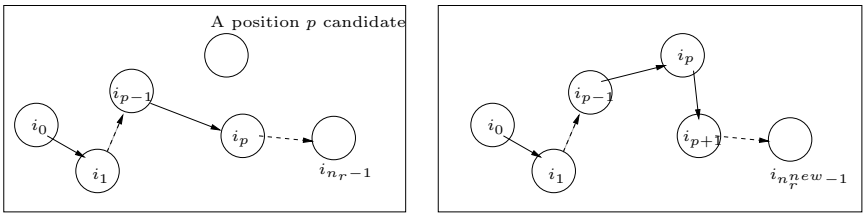


Figure 3.1: The numbering of the positions in a partially constructed route

If inserting v after i_{p-1} in route r^* the arrival time l_v for the visit v is set to be the finishing time $f_{i_{p-1}}$ of visit i_{p-1} plus the travelling time between the two citizens $z_{i_{p-1}}$ and z_v. If the visit is inserted at position 0, then the arrival time is set to the maximum of the visit's opening time a_v and the earliest starting time on the route g_{r^*}.

$$l_v = \begin{cases} \max\{a_v, g_{r^*}\} & \text{if } p = 0 \\ f_{i_{p-1}} + t_{z_v, z_{i_{p-1}}} & \text{if } p \in \{1, 2, \ldots, n_{r^*}\} \end{cases} \tag{3.1}$$

The starting time s_v is the largest of the arrival time l_v, the opening time of the time window a_v and the starting time of the route g_{r^*}. The starting

time is set to the opening time a_v, if the caretaker arrives before the visit v opens and the starting time is set to the arrival time l_v, if the caretaker arrives after the opening time.

$$s_v = \max\{l_v, a_v\} \tag{3.2}$$

After inserting the visit v in position p, the positions of the *succeeding visits* will increment by one, and the new number of visits in route r^* will be $n_{r^*}^{new} = n_{r^*}+1$. The arrival and starting times of the succeeding visits may be *pushed forward* to ensure feasibility, but if there already were sufficient waiting time between visit i_{p-1} and i_p, visit v can be inserted without pushing the succeeding visits.

Inserting a shared visit is more complex, because it has to be inserted in two distinct routes r_1 and r_2. The shared visit is split into two separate visits with the same duration, time window and citizen. This makes it easier to insert the shared visit into two routes.

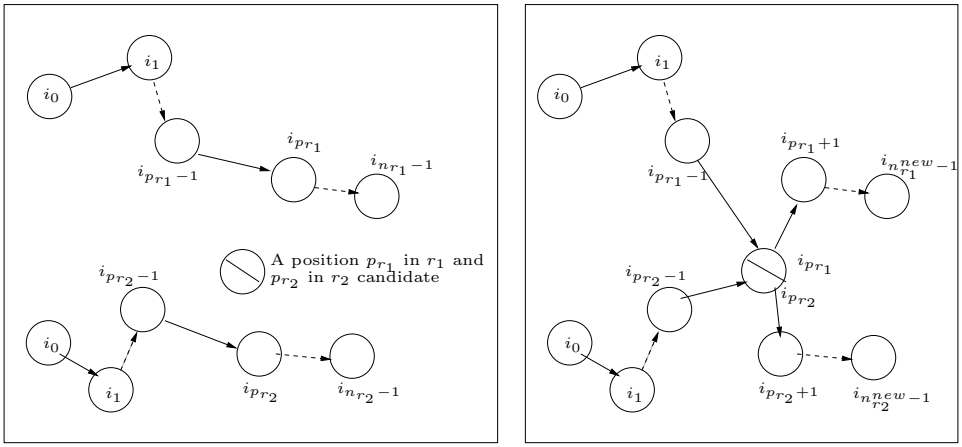


Figure 3.2: The numbering of the positions in two routes r_1 and r_2

In figure 3.2 the scenario is shown, when a shared visit is inserted in position p_{r_1} on route r_1 and position p_{r_2} on route r_2. The shared visit v is split in two visits v_1 and v_2, and each of the visits should start at the same time. The arrival time for each visit is calculated as in (3.1). This implies that the arrival times on each of the two routes are not necessarily the same, for instance if the shared visit is not inserted at the position 0 in neither of the routes, and the finishing times of the previous visits plus the travelling times are different.

When inserting a shared visit into two routes containing a shared visit already, some positions are not feasible. The two routes rightmost in figure 3.2 contain a shared visit, and if a new shared visit v is inserted, both parts of the shared visit v should be situated before or after $i_{p_{r_1}}$ and $i_{p_{r_2}}$.

The starting time for a shared visit v is the maximum of the two arrival times, the starting times for the two caretakers and the opening time of the time window.

$$s_v = \max\{l_{v_1}, l_{v_2}, a_v\} \tag{3.3}$$

After inserting the shared visit v, the succeeding visits on both routes may have to be pushed forward.

When pushing the visits forward it is necessary to pay attention to those visits, that are shared, because if one part of the shared visit is pushed forward, then the other part should equally be pushed forward, if it not already pushed forward. The other part is in another route, and therefore the succeeding visits in the other route should also be pushed forward.

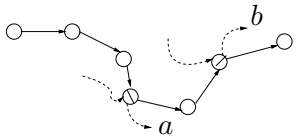


Figure 3.3: There are two shared visits

Figure 3.3 shows a part of a route. If the node leftmost should be pushed, the succeeding visits at the same route might be pushed and also the visits after the shared visits on the two dotted routes a and b.

In some situations the shared visits and the succeeding visits are already pushed. The figure illustrates how the route a can lead back the initial route. If firstly the visits on the initial route are pushed and afterwards the visits on route a, then shared visit v is already pushed one time, when pushing the visits on route a, and may not need to be pushed more.

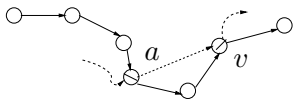


Figure 3.4: The shared visit v is already pushed

Sometimes is it not feasible to insert a visit v in a position p_{r^*}. The opening times of the visits and the earliest starting times for caretakers are never

violated because of (3.2) and (3.3). The equal starting times for shared visits are not violated either, because the starting times of the separated parts v_1 and v_2 are set to s_v.

One of the things that might cause infeasibility is the violation of the closing time in the time window. The topic is treated in lemma 1.1 in [Sol87]. The first part (3.4) of the lemma ensures that the starting time of visit v is before the closing time of the time window.

$$s_v \leq b_v \tag{3.4}$$

The second part of the lemma checks the feasibility for the succeeding visits. Inserting a visit v may cause the succeeding visits to be pushed forward. Therefore (3.5) should be checked before inserting the visit.

$$s_{i_k} + PF_{i_k} \leq b_{i_k}, \qquad \text{for } p_{r^*} \leq k \leq n_{r^*} - 1 \tag{3.5}$$

The other thing that might cause infeasibility is the violation of the latest finishing h_r time for each caretaker r. Let f_r denote the finishing time of the last visit in route r. Inequality (3.6) should be respected to ensure feasibility.

$$f_r \leq h_r \tag{3.6}$$

3.1 The Implementation of the Functions

All the functions are implemented in Java and the source code is found in appendix B.1 and B.2. An overview of the heuristic is shown in figure 3.5, where the connections between the subfunctions are depicted.

The heuristic contains two steps, one for calculating the insertion cost and one for inserting the best visit. Each of the steps are furthermore divided into two parts, one if the current visit is shared and one if it is not. Initially all the unlocked visits are placed in the set $\bar{\mathcal{V}}$. For each v of the visits in $\bar{\mathcal{V}}$ the additional cost of inserting it into a route is calculated. The costs are calculated in almost the same way as in the article [PJM93], where the additional travelling time $t(v, r, p_r)$ for inserting a visit v in route r at position p_r is calculated as

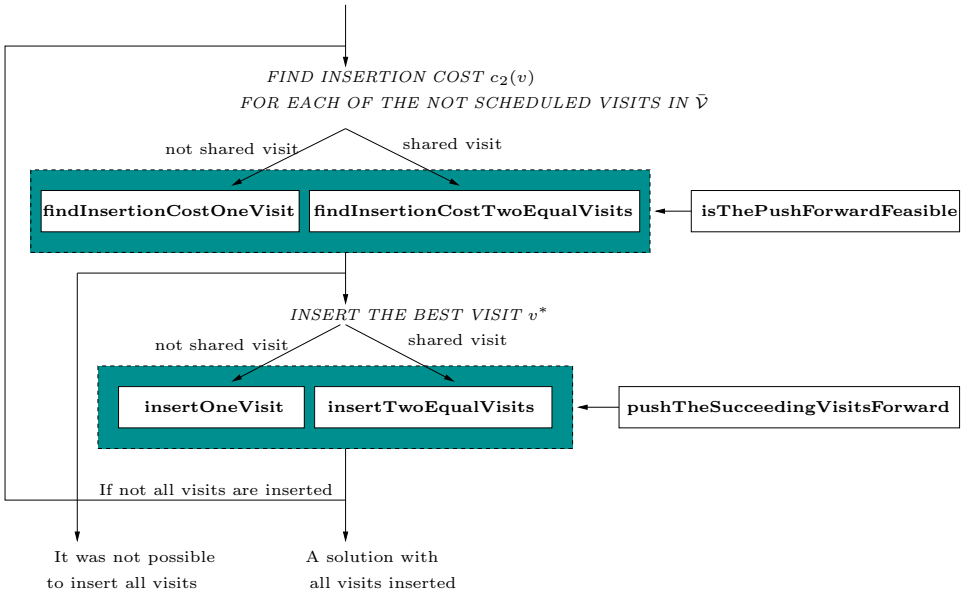


Figure 3.5: A flow chart with the functions

$$t(v,r,p_r) = \begin{cases} t_{z_v,z_{i_{p_r}}} & \text{if } p_r = 0 \\ t_{z_{i_{p_r-1}},z_v} + t_{z_v,z_{i_{p_r}}} - t_{z_{i_{p_r-1}},z_{i_{p_r}}} & \text{if } p_r \in \{1,2,\ldots,n_r-1\} \\ t_{z_{i_{p_r-1}},z_v} & \text{if } p_r = n_r \end{cases} \tag{3.7}$$

where $z_{i_{p_r-1}}$, z_v and $z_{i_{p_r}}$ denote the citizens at the visits i_{p_r-1}, v and i_{p_r}. If the visit is inserted as a intermediate visit the previous travelling cost was $t_{z_{i_{p_r-1}},z_{i_{p_r}}}$ and the new travelling cost is $t_{z_{i_{p_r-1}},z_v} + t_{z_v,z_{i_{p_r}}}$. The difference between the old and the new travelling time gives the extra travelling time $t(v,r,p_r)$. If the visit v is inserted at start or last in the route, the new travelling time is only $t_{z_v,z_{i_{p_r}}}$ or $t_{z_{i_{p_r-1}},z_v}$.

Some adjustments are made to calculate the additional cost, when the caretaker is not regular or if the inserted visit is shared. When the visit v^* with the highest cost $c_2(v^*)$ is found, it is inserted into the best position. The two steps in the heuristic are repeated until all the visits are inserted or until it is not possible to insert more visits. The functions for finding the insertion costs contain a recursive help function, which examines if it is feasible to push the succeeding visits forward. The functions for inserting a visit have a similar function for pushing the succeeding visits forward.

3.1.1 Finding the Cost of Inserting a Not Shared Visit

The not shared visit v for insertion is from the set visits $\bar{\mathcal{V}}$.

The additional cost for inserting visit v in position p_r in route r takes its starting point in the objective function (2.16). The additional travelling time $t(v, r, p_r)$ is given in (3.7). Furthermore there might be an additional cost of μ, if the caretaker r is not regular at citizen z_v, which is indicated by $\tau^r_{z_v} = 0$. The total addition of cost is in (3.8).

$$c_1(v, r, p_r) = t(v, r, p_r) + \mu(1 - \tau^r_{z_v}). \tag{3.8}$$

where $r \in \mathcal{R}$ and $p_r \in \{0, 1, \ldots n_r\}$.

If is not feasible to insert visit v at position p_r, the cost $c_1(v, r, p_r)$ is set to a sufficiently high number M.

For each route r the best position p^*_r is found, where

$$c_1(v, r, p^*_r) = \min\{c_1(v, r, p_r) | p_r \in \{0, 1, \ldots, n_r\}\} \qquad \text{where } r \in \mathcal{R}.$$

Similarly the best route r^* with its corresponding best position $p^*_{r^*}$ is found, where

$$c_1(v, r^*, p^*_{r^*}) = \min\{c_1(v, r, p^*_r) | r \in \mathcal{R}\}$$

The regret measure is measuring how good the cost $c_1(v, r^*, p^*_{r^*})$ is in comparison with the costs for the other routes.

$$c_2(v) = \frac{1}{m-1} \sum_{r \neq r^*} (c_1(v, r, p^*_r) - c_1(v, r^*, p^*_{r^*})) \tag{3.9}$$

If only the route r^* is feasible, the cost $c_2(v)$ is high and it is preferable to insert visit v. As seen in figure 3.5 the visit v^* with the highest cost $c_2(v)$ is inserted.

The pseudo code for calculating the cost $c_2(v)$ is in algorithm 1. All the costs $c_1(v, r, p_r)$, $c_1(v, r, p^*_r)$ and $c_1(v, r^*, p^*_{r^*})$ are initially set to a sufficiently big number M in line 15, 4 and 2. When the insertion in p_r is found feasible, the cost $c_1(v, r, p_r)$ is calculated in line 17, otherwise it stays equal

to M. For each route r the best position p_r^* is found by comparing each cost $c_1(v, r, p_r)$ with the current best cost $c_1(v, r, p_r^*)$ in line 19. Similarly the best cost $c_1(v, r, p_r^*)$ of each route r is compared with the currently best cost $c_1(v, r^*, p_{r^*}^*)$ in line 21 to find the best route r^*.

Algorithm 1 findInsertionCostOneVisit(v)

1: $feasibleInsertion = false$
2: $c_1(v, r^*, p_{r^*}^*) = M$
3: **for** $r \in \mathcal{R}$ **do**
4: $c_1(v, r, p_r^*) = M$
5: **for** $p_r \in \{0, 1, \ldots, n_r\}$ **do**
6: Calculate $\hat{l}_v$ and $\hat{s}_v$ using (3.1) and (3.2)
7: $feasiblePosition = true$
8: **if** $\hat{s}_v \leq b_v$ and $\hat{s}_v + d_v \leq h_r$ **then**
9: $\hat{l}_{i_{p_r}} = \hat{s}_v + d_v + t_{z_v, z_{i_{p_r}}}$
10: $PF_{i_{p_r}} = \max\{0, \hat{l}_{i_{p_r}} - l_{i_{p_r}} - w_{i_{p_r}}\}$
11: If $p_r < n_r$ then $feasiblePosition =$ **isThePushForwardFeasible**$(PF_{i_p}, p_r, r, 0, 0)$
12: **else**
13: $feasiblePosition = false$
14: **end if**
15: $c_1(v, r, p_r) = M$
16: **if** $feasiblePosition$ **then**
17: $feasibleInsertion = true$ and calculate $c_1(v, r, p)$ using (3.8)
18: **end if**
19: If $c_1(v, r, p_r) < c_1(v, r, p_r^*)$ then set $c_1(v, r, p_r^*) = c_1(v, r, p_r)$ and $p_r^* = p$
20: **end for**
21: If $c_1(v, r, p_r^*) < c_1(v, r^*, p_{r^*}^*)$ then set $c_1(v, r^*, p_{r^*}^*) = c_1(v, r, p_r^*)$ and $r^* = r$
22: **end for**
23: Calculate $c_2(v)$ using (3.9)
24: **return** $c_2(v)$, r^*, $p_{r^*}^*$ and $feasibleInsertion$

In algorithm 1 it is investigated if it is feasible to place visit v in position p_r in route r. This investigation includes calculations of new arrival and starting times, if the visit was inserted. All the new times are indicated by a hat ($\hat{}$), because it is only hypothetical. The new starting times have to be within the time windows and the working hours, and if they do not satisfy this, the insertion will not be feasible.

The arrival time $\hat{l}_v$ and the starting time $\hat{s}_v$ for the visit v is calculated in line 6. It is a necessary condition that $\hat{s}_v \leq b_v$ according to (3.4) for the insertion to be feasible. It is also necessary that $\hat{s}_v \leq h_r$ according to (3.6).

These are not the only conditions, because it should be investigated if the succeeding visits can be pushed forward without violating any of conditions (3.5) and (3.6). The arrival time $\hat{l}_{i_{p_r}}$ to the next visit i_{p_r} is calculated in line 9 and the push forward of visit i_{p_r} is calculated in line 10. The push forward is zero, if the caretaker was waiting more than $\hat{l}_{i_{p_r}} - l_{i_{p_r}}$ minutes. A push forward of the starting time will be performed, if $\hat{l}_{i_{p_r}} - l_{i_{p_r}} > w_{i_{p_r}}$. The push forward will be $\hat{l}_{i_{p_r}} - l_{i_{p_r}} - w_{i_{p_r}}$.

3.1.2 Finding the Cost of Inserting a Shared Visit

When inserting a shared visit v, the costs are calculated differently, because the shared visits have to be inserted in two distinct routes. Instead of finding the best route for the unscheduled visit, the best pair of routes is found.

The total additional travelling cost is found as $t(v, r_1, p_{r_1}) + t(v, r_2, p_{r_2})$ using (3.7), and it is divided by 2 to make it comparable to the cost for a not shared visit in (3.8), when choosing which visit to insert. If none of the routes r_1 and r_2 has the regular caretaker for citizen z_v, then the additional cost is μ, because $\tau_{z_v}^{r_1} = 0$ and $\tau_{z_v}^{r_2} = 0$. If at least one of the caretakers is regular at the citizen z_v, there is no additional cost.

$$\begin{aligned} c_1(v, r_1, r_2, p_{r_1}, p_{r_2}) = \quad & (t(v, r_1, p_{r_1}) + t(v, r_2, p_{r_2}))/2 + \\ & \mu(1 - \tau_{z_v}^{r_1})(1 - \tau_{z_v}^{r_2}), \end{aligned} \tag{3.10}$$

where $r_1 \in \{j_0, j_1, \ldots, j_{m-2}\}$, $r_2 \in \{r_1 + 1, \ldots, j_{m-1}\}$, $p_{r_1} \in \{0, 1, \ldots n_{r_1}\}$, and $p_{r_2} \in \{0, 1, \ldots n_{r_2}\}$.

For each pair of routes r_1 and r_2 the best pair of positions $p_{r_1}^*$ and $p_{r_2}^*$ is found.

$$\begin{aligned} c_1(v, r_1, r_2, p_{r_1}^*, p_{r_2}^*) = \quad & \min c_1(v, r_1, r_2, p_{r_1}, p_{r_2}) \\ & \text{st. } p_{r_1} \in \{0, 1, \ldots, n_{r_1}\}, \\ & \quad p_{r_2} \in \{0, 1, \ldots, n_{r_2}\}, \end{aligned}$$

where $r_1 \in \{j_0, j_1, \ldots, j_{m-2}\}$ and $r_2 \in \{r_1 + 1, \ldots, j_{m-1}\}$.

Afterwards the best combination of routes r_1^* and r_2^* is found with the corresponding best pair of positions $p_{r_1^*}^*$ and $p_{r_2^*}^*$.

$$\begin{aligned} c_1(v, r_1^*, r_2^*, p_{r_1^*}^*, p_{r_2^*}^*) = \quad & \min \ c_1(v, r_1, r_2, p_{r_1}^*, p_{r_2}^*) \\ & \text{st. } r_1 \in \{j_0, j_1, \ldots, j_{m-2}\}, \\ & \quad r_2 \in \{r_1, \ldots, j_{m-1}\}. \end{aligned}$$

The regret measure $c_2(v)$ is again calculated, and it is divided by the number of combinations of two routes $m(m-1)/2$ minus 1. Notice that there should

be more than two routes, otherwise it will not be possible to calculate the regret measure.

$$c_2(v) = \frac{1}{m(m-1)/2-1} \sum_{r_1 \neq r_1^*, r_2 \neq r_2^*} (c_1(v, r_1, r_2, p_{r_1}^*, p_{r_2}^*) - c_1(v, r_1^*, r_2^*, p_{r_1^*}^*, p_{r_2^*}^*)) \tag{3.11}$$

The costs $c_1(v, r_1, r_2, p_{r_1}, p_{r_2})$, $c_1(v, r_1, r_2, p_{r_1}^*, p_{r_2}^*)$ and $c_1(v, r_1^*, r_2^*, p_{r_1^*}^*, p_{r_2^*}^*)$ are initially set to a sufficiently large number M in line 22, 5 and 2. After each new combination of the positions p_{r_1} and p_{r_2}, it is investigated in line 24 if the new cost $c_1(v, r_1, r_2, p_{r_1}, p_{r_2})$ is smaller than the currently best cost $c_1(v, r_1, r_2, p_{r_1}^*, p_{r_2}^*)$. Similarly after each combination of routes it is investigated in line 29 if the cost of that combination $c_1(v, r_1, r_2, p_{r_1}^*, p_{r_2}^*)$ is better than the currently best cost $c_1(v, r_1^*, r_2^*, p_{r_1^*}^*, p_{r_2^*}^*)$.

In algorithm 2 it is investigated what would happen, if a visit would be inserted at position p_{r_1} in route r_1 and position p_{r_2} in route r_2. The caretaker r_1 arrives at $\hat{l}_{v_1}$ and caretaker r_2 arrives at $\hat{l}_{v_1}$, which is calculated in line 8 along with the hypothetical starting time $\hat{s}_v$.

Afterwards the conditions (3.4) and (3.6) are checked for both caretakers on the routes r_1 and r_2 in line 10. If the conditions are met, the investigation continues. Firstly it is investigated if it is feasible to insert the visit at position p_{r_1} in route r_1. The new arrival time $\hat{l}_{i_{p_{r_1}}}$ to the next visit in route r_1 is calculated in line 11 and the push forward $PF_{i_{p_{r_1}}}$ is calculated in line 12. It is in line 13 tried if the push forward is feasible. If it is feasible, it is investigated if the visit can be inserted at position p_{r_2} in route r_2. It is done in a similar manner by calculating the new arrival time $\hat{l}_{i_{p_{r_2}}}$ for the next visit in line 15 and the push forward $PF_{i_{p_{r_2}}}$ in line 16. If the insertion is infeasible then $c_1(v, r_1, r_2, p_{r_1}, p_{r_2})$ is set to M, otherwise it is calculated as in (3.10).

3.1.3 When is it Feasible to Push a Visit Forward?

The conditions (3.4) and (3.6) are checked in algorithm 1 or 2. In algorithm 3 the other condition (3.5) of the feasibility conditions is checked together with (3.6).

Algorithm 3 is a recursive function for checking the feasibility. In line 6 it is checked if it is feasible to push visit i_k forward. If it is a shared visit, the function continues checking if it is feasible to push the other part $i_{p_{r_2}}$ of

Algorithm 2 findInsertionCostTwoEqualVisits(v)

$feasibleInsertion = false$
$c_1(v, r_1^*, p_{r_1^*}^*, r_2^*, p_{r_2^*}^*) = M$
for $r_1 \in \{j_0, j_1, \ldots, j_{m-2}\}$ **do**
  for $r_2 \in \{r_1 + 1, \ldots, j_{m-1}\}$ **do**
    $c_1(v, r_1, r_2, p_{r_1}^*, p_{r_2}^*) = M$
    for $p_{r_1} \in \{0, 1, \ldots, n_r\}$ **do**
      for $p_{r_2} \in \{0, 1, \ldots, n_r\}$ **do**
        Calculate $\hat{l}_{v_1}$ and $\hat{l}_{v_2}$ using (3.1) and $\hat{s}_v$ using (3.3).
        $feasiblePosition = true$
        if $\hat{s}_v \leq b_v$ and $\hat{s}_v + d_v \leq \min\{h_{r_1}, h_{r_2}\}$ **then**
          $\hat{l}_{i_{p_{r_1}}} = \hat{s}_v + d_v + t_{z_v, z_{i_{p_{r_1}}}}$
          $PF_{i_{p_{r_1}}} = \max\{0, \hat{l}_{i_{p_{r_1}}} - l_{i_{p_{r_1}}} - w_{i_{p_{r_1}}}\}$
          If $p_{r_1} < n_{r_1}$ then $feasiblePosition =$ **isThePushForwardFeasible**($PF_{i_{p_{r_1}}}$,p_{r_1}, r_1, r_1,p_{r_2})
          if $feasiblePosition$ **then**
            $\hat{l}_{i_{p_{r_2}}} = \hat{s}_v + d_v + t_{z_v, z_{i_{p_{r_2}}}}$
            $PF_{i_{p_{r_2}}} = \max\{0, \hat{l}_{i_{p_{r_2}}} - l_{i_{p_{r_2}}} - w_{i_{p_{r_2}}}\}$
            If $p_{r_2} < n_{r_2}$ then $feasiblePosition =$ **isThePushForwardFeasible**($PF_{i_{p_{r_2}}}$,p_{r_2}, r_2,r_1,p_{r_1})
          end if
        else
          $feasiblePosition = false$
        end if
        $c_1(v, r_1, r_2, p_{r_1}, p_{r_2}) = M$
        If $feasiblePosition = true$ then set $feasibleInsertion = true$ and calculate $c_1(v, r_1, r_2, p_{r_1}, p_{r_2})$ using (3.10)
        if $c_1(v, r_1, r_2, p_{r_1}, p_{r_2}) < c_1(v, r_1, r_2, p_{r_1}^*, p_{r_2}^*)$ **then**
          set $c_1(v, r_1, r_2, p_{r_1}^*, p_{r_2}^*) = c_1(v, r_1, r_2, p_{r_1}, p_{r_2})$ and $p_{r_1}^* = p_{r_1}$ and $p_{r_2}^* = p_{r_2}$
        end if
      end for
    end for
    if $c_1(v, r_1, r_2, p_{r_1}^*, p_{r_2}^*) < c_1(v, r_1^*, p_{r_1^*}^*, r_2^*, p_{r_2^*}^*)$ **then**
      set $c_1(v, r_1^*, p_{r_1^*}^*, r_2^*, p_{r_2^*}^*) = c_1(v, r_1, r_2, p_{r_1}^*, p_{r_2}^*)$ and $r_1^* = r_1$ and $r_2^* = r_2$
    end if
  end for
end for
Calculate $c_2(v)$ using (3.11)
return $c_2(v)$, r_1^*, $p_{r_1^*}^*$, r_2^*, $p_{r_2^*}^*$ and $feasibleInsertion$

Algorithm 3 isThePushForwardFeasible$(PF_{i_k}, k, r, r_f, k_f)$

```
1: feasible = true
2: if r = r_f and k ≤ k_f then
3:    feasible = false
4: end if
5: if feasible and PF_{i_k} > 0 then
6:    if s_{i_k} + PF_{i_k} > b_{i_k} or s_{i_k} + PF_{i_k} + d_{i_k} > h_r then
7:       feasible = false
8:    end if
9:    if feasible and visit i_k is shared with visit i_{p_{r_2}} in route r_2 then
10:      if s_{i_{p_{r_2}}} + PF_{i_k} > b_{i_{p_{r_2}}} or s_{i_{p_{r_2}}} + PF_{i_k} + d_{i_{p_{r_2}}} > h_{r_2} then
11:         feasible = false
12:      end if
13:      if feasible and p_{r_2} < n_{r_2} − 1 then
14:         PF_{i_{p_{r_2}+1}} = max{0, PF_{i_k} − w_{i_{p_{r_2}+1}}}
15:         feasible = isThePushForwardFeasible(PF_{i_{p_{r_2}+1}}, p_{r_2} + 1, r_2, r_f, k_f)
16:      end if
17:   end if
18:   if feasible and p_r < n_r − 1 then
19:      PF_{i_{k+1}} = max{0, PF_{i_k} − w_{i_{k+1}}}
20:      feasible = isThePushForwardFeasible(PF_{i_{k+1}}, k + 1, r, r_f, k_f)
21:   end if
22: end if
23: return feasible
```

the visit equally forward. If this is feasible and $i_{p_{r_2}}$ is not the last visit on the route r_2, the function investigates if the succeeding visits can be pushed forward by calling itself with the next position and the push forward as arguments in line 15.

If this also is feasible, and we are not at the end of route r the next push forward is checked. The push forward of the next visit is calculated in line 19 as is it shown at page 256 in the article [Sol87].

$$PF_{i_{k+1}} = \max\{0, PF_{i_k} - w_{i_{k+1}}\}$$

If the waiting time at the next visit is greater than the length of the current push forward, then the push forward is set to zero.

The numbers r_f and k_f indicate a forbidden route and position combination. It is necessary to have this check in line 2, if the candidate routes r_1 and r_2 for a shared visit, already contain a shared visit.

The figure 3.6 illustrates a situation where a shared visit is tried inserted in two routes r_1 and r_2, where there already is a shared visit. The candidate positions are p_1 and p_2. If the other shared visit u is between p_1 and p_2 a cycle can arise in algorithm 3, but this is avoided by defining position p_2 in route r_2 as forbidden when pushing forward from position p_1 in route r_1.

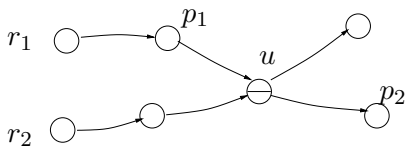


Figure 3.6: When pushing forward from position p_1 in route r_1 it is forbidden to meet a position less than or equal to position p_2 in route r_2

If the push forward is not feasible or the visit investigated is at the end of the route or the push forward is 0, then the recursion stops and returns whether the push forward is feasible or not.

3.1.4 Inserting a Not Shared Visit

When inserting a not shared visit, its properties are set according to the new position. The arrival, waiting, starting and finishing time are set in line 1, 2, 3 and 4 and v is inserted in line 5 in algorithm 4.

Algorithm 4 insertOneVisit(v, p_r, r)

1: set l_v and s_v according to (3.1) and (3.2)
2: set $s_v = \max\{a_v, l_v\}$
3: set $w_v = s_v - l_v$
4: set $f_v = s_v + d_v$
5: Insert v at position p_r in route r
6: $\bar{\mathcal{V}} = \bar{\mathcal{V}} \setminus v$
7: $\underline{\mathcal{V}} = \underline{\mathcal{V}} \cup v$
8: **if** $p_r < n_r - 1$ **then**
9: calculate $\hat{l}_{i_{p_r+1}} = f_v + t_{z_v, z_{i_{p_r+1}}}$
10: $PF_{i_{p_r}} = \max\{0, \hat{l}_{i_{p_r+1}} - l_{i_{p_r+1}} - w_{i_{p_r+1}}\}$
11: set $l_{i_{p_r+1}} = \hat{l}_{i_{p_r+1}}$
12: **pushTheSucceedingVisitsForward**$(PF_{i_{p_r+1}}, p_r + 2, r)$;
13: **end if**

In algorithm 4 the next visit is used to find the push forward in line 9. When finding the push forward, the arrival, waiting, starting and finishing time for the next visit is also set.

If the visit after the next visit is not the last visit in the route, the recursive function in algorithm 6 is called.

3.1.5 Inserting a Shared Visit

The algorithm 5 corresponds very much to algorithm 4. The only difference is that everything is done twice: once for each part v and w of the shared visit.

Algorithm 5 insertTwoEqualVisits$(v, w, p_{r_1}, p_{r_2}, r_1, r_2)$

1: set l_v and l_w according to (3.1)
2: set $s_v = s_w$ according to (3.3)
3: set $w_v = s_v - l_v$ and $w_w = s_w - l_w$
4: set $f_v = s_v + d_v$ and $f_w = s_w + d_w$
5: Insert v at position p_{r_1} in route r_1
6: Insert w at position p_{r_2} in route r_2
7: set $\bar{\mathcal{V}} = \bar{\mathcal{V}} \setminus \{v, w\}$
8: set $\underline{\mathcal{V}} = \underline{\mathcal{V}} \cup \{v, w\}$
9: **if** $p_{r_1} < n_{r_1} - 1$ **then**
10: calculate $\hat{l}_{i_{p_{r_1}+1}} = f_v + t_{z_v, z_{i_{p_{r_1}+1}}}$
11: $PF_{i_{p_{r_1}+1}} = \max\{0, \hat{l}_{i_{p_{r_1}+1}} - l_{i_{p_{r_1}+1}} - w_{i_{p_{r_1}+1}}\}$
12: set $l_{i_{p_{r_1}+1}} = \hat{l}_{i_{p_{r_1}+1}}$
13: **pushTheSucceedingVisitsForward**$(PF_{i_{p_{r_1}+1}}, p_{r_1} + 1, r_1)$;
14: **end if**
15: **if** $p_{r_2} < n_{r_2} - 1$ **then**
16: calculate $\hat{l}_{i_{p_{r_2}+1}} = f_w + t_{z_v, z_{i_{p_{r_2}+1}}}$
17: $PF_{i_{p_{r_2}+1}} = \max\{0, \hat{l}_{i_{p_{r_2}+1}} - l_{i_{p_{r_2}+1}} - w_{i_{p_{r_2}+1}}\}$
18: set $l_{i_{p_{r_2}+1}} = \hat{l}_{i_{p_{r_2}+1}}$
19: **pushTheSucceedingVisitsForward**$(PF_{i_{p_{r_2}+1}}, p_{r_2} + 1, r_2)$;
20: **end if**

3.1.6 Push the Succeeding Visits

The algorithm 6 calculates the current push forward $PF_{i_{p_r}}$ from the previous push forward $PF_{i_{p_r-1}}$ as seen in line 6. The succeeding visits are pushed by using the same function again. The algorithm is recursive and stops if the end of a route is reached.

Algorithm 6 pushTheSucceedingVisits$(PF_{i_{p_r}}, p_r, r)$

1: set $s_{i_{p_r}} = s_{i_{p_r}} + PF_{i_{p_r}}$
2: set $w_{i_{p_r}} = s_{i_{p_r}} - l_{i_{p_r}}$
3: set $f_{i_{p_r}} = f_{i_{p_r}} + PF_{i_{p_r}}$
4: **if** $p_r < n_r - 1$ **then**
5: set $l_{i_{p_r+1}} = l_{i_{p_r+1}} + PF_{i_{p_r}}$
6: $PF_{i_{p_r+1}} = \max\{0, PF_{i_{p_r}} - w_{i_{p_r+1}}\}$
7: **pushTheSucceedingVisitsForward**$(PF_{i_{p_r+1}}, p_r + 1, r)$;
8: **end if**
9: **if** visit i_{p_r} is shared with visit $i_{p_{r_2}}$ in route r_2 **then**
10: $PF_{i_{p_{r_2}}} = s_{i_{p_r}} - s_{i_{p_{r_2}}}$
11: set $s_{i_{p_{r_2}}} = s_{i_{p_{r_2}}} + PF_{i_{p_{r_2}}}$ and set $w_{i_{p_{r_2}}} = s_{i_{p_{r_2}}} - l_{i_{p_{r_2}}}$
12: set $f_{i_{p_{r_2}}} = f_{i_{p_{r_2}}} + PF_{i_{p_{r_2}}}$
13: **if** $p_{r_2} < n_{r_2} - 1$ **then**
14: set $l_{i_{p_{r_2}+1}} = l_{i_{p_{r_2}+1}} + PF_{i_{p_r}}$
15: $PF_{i_{p_{r_2}+1}} = max\{0, PF_{i_{p_{r_2}}} - w_{i_{p_{r_2}+1}}\}$
16: **pushTheSucceedingVisitsForward**$(PF_{i_{p_{r_2}+1}}, p_{r_2} + 1, r_2)$;
17: **end if**
18: **end if**

In algorithm 6 the current visit may be a shared visit.

If one half of a shared visit is pushed, the other half should also be pushed, if it is not already pushed forward. Therefore the push forward for the other half equals the difference in starting times in line 10. Figure 3.7 illustrates how firstly visit 1, 2, 3 and 4 are pushed forward. Afterwards visit 5, 6, 7 and 8 are pushed forward, and then the visits 3 and 4 once more.

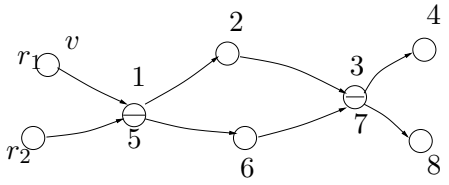


Figure 3.7: A shared visit may be pushed more than once

3.2 Example of the Heuristic

In this section the heuristic is illustrated by a small example. There are 5 citizens in the example. The locations of the citizens are displayed in figure 3.8.

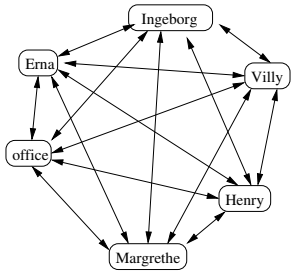


Figure 3.8: Map of the 5 citizens

and the corresponding distance matrix is in table 3.1

The properties of the visits are given in table 3.2. The time is relative and it is set to the number of minutes after 8 am. For instance the time 480 is a representation of the time 4 pm. Visit 12 and 13 constitute a shared visit, and there is one visit 0 and 14 for each caretaker.

There are only two caretakers in this example, a substitute caretaker is added, because it is not possible with the heuristic to put all the visits into only two routes. The caretaker 0 is regular at citizen 1 and 2, the other caretaker 1 is regular at citizen 3, 4 and 5, while the last caretaker 2 is not regular at any of the citizens, because he/she is a substitute.

	Office	Erna	Ingeborg	Villy	Margrethe	Henry
Office	0	4	8	7	6	7
Erna	4	0	5	7	15	10
Ingeborg	8	5	0	5	10	8
Villy	7	7	5	0	9	7
Margrethe	6	15	10	9	0	4
Henry	7	10	8	7	4	0

Table 3.1: The distances in the small example with 5 citizens

Visit i	Opening time a_i	Closing time b_i	Duration d_i	Citizen z_i
0	0	0	0	Office
1	0	60	90	Erna
2	180	270	60	Erna
3	420	480	30	Erna
4	60	120	60	Ingeborg
5	270	300	30	Ingeborg
6	300	330	90	Ingeborg
7	0	60	50	Villy
8	240	330	30	Villy
9	30	90	30	Margrethe
10	390	360	60	Margrethe
11	180	270	60	Henry
12	0	120	60	Henry
13	0	120	60	Henry
14	480	480	0	Office

Table 3.2: The properties of the visits in the example

The extra cost μ is set to 15. This is the price in minutes for not having the regular caretaker at a citizen.

Initially the three caretakers only have a start and an end visit at the office in their routes. The initial schedule without the visits at the office is shown in figure 3.9 and it is observed that the caretakers are only waiting. The waiting time is illustrated as vertical zigzag lines.

The visit to be inserted is found by calculating the cost $c_2(v)$ as in (3.9). Using (3.8) the costs $c_1(v, r, p_r)$ are calculated. The visit 1 has the best route $r^* = 0$ with position $p^*_{r^*} = 1$, where the cost is $c_1(v, r^*, p^*_{r^*}) = c_1(1, 0, 1) = t_{office,Erna} + t_{Erna,office} - t_{office,office} = 8$. If inserted in route $r = 1$ or $r = 2$ the best and only position is $p^*_1 = p^*_2 = 1$. The costs are $c_1(1, r, p^*_r) =$

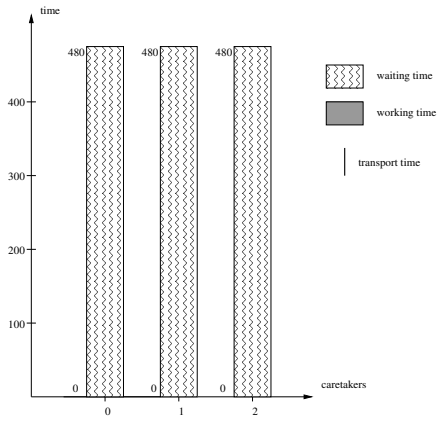


Figure 3.9: The initial schedule for the 3 caretakers

$c_1(1,1,1) = c_1(1,2,1) = 8 + 15 = 23.$

$$c_2(1) = \frac{1}{m-1} \sum_{r \neq r^*} (c_1(1, r, p_r^*) - c_1(1, r^*, p_{r^*}^*)) = \frac{1}{3-1}(23 - 8 + 23 - 8) = 15$$

There are other visits with the same price, but visit 1 is the first, and hence it is inserted as illustrated in figure 3.10, where the light grey box indicates the working time.

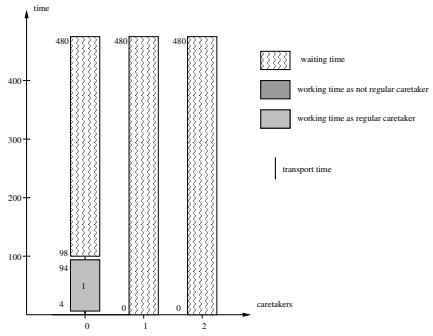


Figure 3.10: The schedule for the 3 caretakers with visit 1

After inserting more visits, the schedule is now as shown in figure 3.11

The shared visit v consists of visit 12 and 13. The citizen is Henry. The cost of inserting visit 12 and 13 is found. There are $m(m-1)/2 = 3(3-1)/2 =$ 3 combinations of routes with the caretakers (0,1), (0,2) and (1,2). For

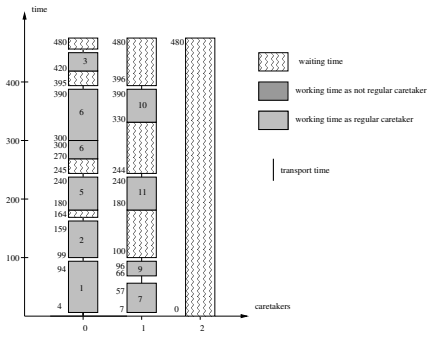


Figure 3.11: The schedule for the 3 caretakers with more visits

instance the best positions in route 1 and 2 are $p_1^* = 3$ and $p_2^* = 1$. The cost is calculated with (3.10)

$$
\begin{aligned}
c_1(v,1,2,p_1^*,p_2^*) = & \\
c_1(v,1,2,3,1) = \; & (t_{Margrethe,Henry} + t_{Henry,Henry} - t_{Margrethe,Henry} \\
& + t_{Office,Henry} + t_{Henry,Office} - t_{Office,Office})/2 \\
& + 15(1-1)(1-0) \\
= \; & (4 + 0 - 4 + 7 + 7 - 0)/2 + 0 = 7
\end{aligned}
$$

All the other costs are

$$c_1(v,0,1,p_0^*,p_1^*) = c_1(v,0,1,2,3) = 10 \text{ and}$$

$$c_1(v,0,2,p_0^*,p_2^*) = c_1(v,0,2,2,2) = 17$$

The best combination is $r_1^* = 1$ and $r_2^* = 2$ and $c_2(v) = \frac{1}{3-1}(10-7+17-7) = 6.5$. This is the largest value of $c_2(v)$ for the remaining not scheduled visits. Therefore the visits 12 and 13 are inserted as in figure 3.12.

The final schedule is seen in figure 3.13, where luckily all the citizens get a visit from their regular caretaker. Henry gets a shared visit from caretaker 1 and 2, where number 1 is regular, but number 2 is not regular. This is not punished in the cost. It is enough if only one of them is regular.

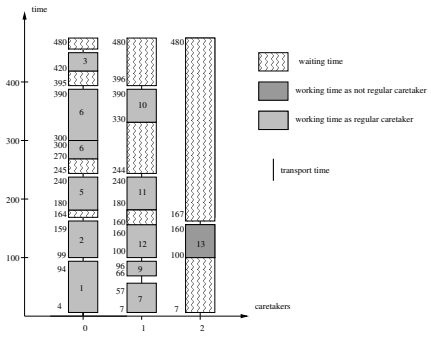


Figure 3.12: The schedule for the 3 caretakers with the shared visit consisting of visit 12 and 13

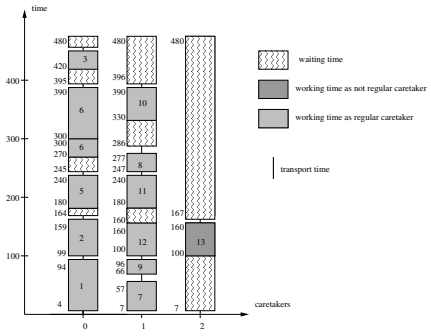


Figure 3.13: The schedule for the 3 caretakers with all visits

3.3 Complexity

The worst case computation time is $\mathcal{O}(n^4)$, because in each of the $\mathcal{O}(n)$ iterations, it is tested for each of $\mathcal{O}(n)$ visits if can be inserted in one of the $\mathcal{O}(n)$ positions. When testing if a visit can be inserted, the already inserted visits can be pushed. In the worst case scenario $\mathcal{O}(n)$ visits are pushed.

Chapter 4

Tabu Search

Tabu search is based on the concept of performing changes to a solution in the attempt to obtain an improved solution. The change of a solution is called a *move* from one solution to another. The solutions that can be reached from a solution x by making moves is called a *neighbourhood*. More formally a neighbourhood function N can be described as a mapping from the solution space $\mathcal{S}$ to the same solution space $\mathcal{S}$

$$N : \mathcal{S} \rightarrow \mathcal{S}$$

The argument for the neighbourhood is a solution $x \in \mathcal{S}$ and the output is a subset of $\mathcal{S}$. The neighbourhood of x is denoted $N(x)$, where each $\hat{x} \in N(x)$ is a neighbour to x.

The slides [Sti05] give a set of useful tools for evaluating neighbourhoods and how they are used. The special issues to consider when using neighbourhoods are.

Coverage: Whether the neighbourhood is able to cover the whole solution space by one type of move, or if several types of moves are required. It is not necessarily bad when the whole solution space can not be reached, if the good solutions can be reached

Guidance: Should the best solutions in the neighbourhood be chosen? It could have both advantages and disadvantages. An advantage would be obtaining good solutions faster, but the disadvantages are more computation time or maybe misleading the search.

Continuity: Neighbourhood solutions should be related, otherwise the search becomes random. A relationship could be that a good solution x implies a good neighbour $\hat{x}$.

Size: The size of the neighbourhood is of importance, because it can both be too small or too big. A neighbourhood too small can not perform a good search and the risk of ending up in a locale optimum arises. Neighbourhoods too large demand too much computational effort.

Feasibility: It should be considered whether the good solutions are on the border to infeasibility, and if it should be avoided to obtain infeasible solutions. In the case, where infeasible solutions are obtained, how should they be handled?

Local search is based on the idea to find better solutions by making moves and looking in the neighborhood for better solutions. There is a problem of *guidance* if one always go from worse to better solution, because it is possible to get stuck in a locale optimum. Figure 4.1 illustrates this situation, where the local search is stuck in a valley.

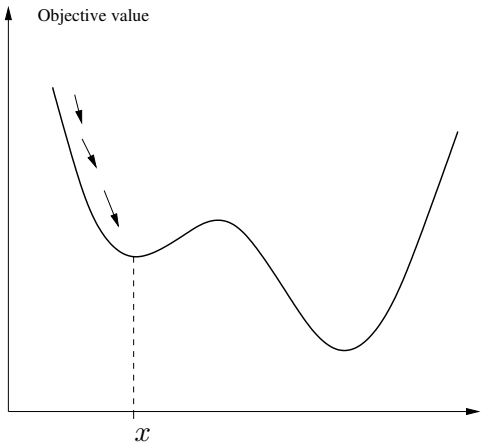


Figure 4.1: A locale minimum

To avoid ending up in a locale optimum, it is allowed to go to worse solutions. This may cause a new problem; *cycling*, which means that the search revisits solutions, that have been visited before. This can be avoided by having a *tabu strategy*. The tabu search has a tabu strategy, which for instance can forbid the solutions in the last iterations or the last moves to be used again. Which tabu strategy is best depends on the type of the problem.

Tabu search is a metaheuristic. A metaheuristic a heuristic, which can be adapted to different problems. As other heuristics a metaheuristic does not

necessarily find the optimal solution. The tabu search has showed to be very efficient for many kinds of problems, and is therefore one of the most used metaheuristics.

In the VRPTWSV, it is difficult to perform a move without violating any of the constraints, hence it would be a good idea to allow infeasible solutions. One might also expect the good solutions to lie on the border of *feasibility*. The question is how to treat the infeasible solutions? The article [CLA01] gives an answer to this question, where *unified tabu search* is introduced on the VRPTW. The main feature of this method is that the problem is relaxed in the same way as Lagrangean relaxation, where it is possible to violate certain constraints, and the violation is penalized. The penalties varies dependent on the number of times a certain constraint is violated.

In this case with VRPTWSV it is possible to violate the constraints (2.22), (2.24) and (2.27).

Relaxing the constraint (2.27) involve accepting starting a visit later than the latest starting time b_i for every visit $i \in \mathcal{V}$. The cost for every minute of violation is α. The total violation is given in (4.1).

$$A(x) = \sum_{i \in \mathcal{V}} \max\{s_i - b_i, 0\} \tag{4.1}$$

Relaxing the constraint (2.22) is the same as allowing the two separate visits in a shared visit to start at different times. The price for this violation is β per minute. The total violation is given in (4.2). Notice that the multiplication by 1/2, which is applied, because every pair of visits (i,j) is included twice in the summation.

$$B(x) = \frac{1}{2} \sum_{i,j \in \mathcal{V}} \omega_{ij} |s_i - s_j| \tag{4.2}$$

Relaxing the last constraint (2.24) is equivalent to allowing caretakers to work later than the latest finishing time h_r for every $r \in \mathcal{R}$. The price per minute for this violation is γ. The total violation is given in (4.3), where $f_r = \max\{f_i | i \in r\}$.

$$G(x) = \sum_{r \in \mathcal{R}} \max\{f_r - h_r, 0\} \tag{4.3}$$

The prices α, β and γ are positive and they are modified after each iteration in the search. The positive factor δ is the modification factor. If a constraint among (2.27), (2.22) or (2.24) is violated, it implies that the corresponding price α, β or γ is multiplied by $(1 + \delta)$. If the constraint is not violated, it implies that the price will be divided by $(1 + \delta)$.

The already known cost function $C(x)$ is given as $T + \mu\Psi$ in the objective function (2.16) in the mathematical model for the VRPTWSV. The cost $C(x)$ forms together with $A(x)$, $B(x)$ and $G(x)$ the new cost $F(x)$ as in (4.4). The use of $F(x)$ is explained in section 4.4.

$$F(x) = C(x) + \alpha A(x) + \beta B(x) + \gamma G(x) \tag{4.4}$$

Notice that the prices α, β and γ in the article [CLA01] correspond to violations, that are different from the violations in this project, namely the violation of the maximum load capacity at the vehicles, the violation of the routes' durations and the violation of the latest starting times at customers.

Algorithm 7 shows the pseudo code for the tabu search, where the method starts in an initial solution x. In this project the initial solution is a feasible solution found by the insertion heuristic described in chapter 3. The moves and neighbourhood functions used in this project are introduced in section 4.1 and 4.2.

Algorithm 7 Tabu Search

```
x is a initial solution
x* is the best feasible solution currently
if x is feasible then
    x* := x
    C(x*) := C(x)
else
    C(x*) := ∞
end if
while stop criterion not reached do
    choose the best solution x̂ ∈ N(x), which is not tabu or satisfies the aspiration criteria
    if x̂ is feasible and C(x̂) < C(x*) then
        x* := x̂
        C(x*) := C(x̂)
    end if
    Update α, β and γ
    x := x̂
end while
```

For every iteration is the best non-tabu solution in the neighbourhood chosen. The section 4.3 describes what characterizes a non-tabu solution. The evaluation used for finding the best solution includes a factor for diversifying the search, which will be introduced in the section 4.4. If the best solution turns out to be tabu, it can still be chosen, if it satisfies the aspiration cri-

terion described in section 4.5. The stopping criterion for the method is introduced in section 4.6.

In the article [CLA01], they suggest violating the latest starting times for visits, but not the earliest starting times, and this is discussed in section 4.7.

The section 4.8 contains a discussion on how the starting times of the visits in a route are updated, when removing a visit from a route or inserting a visit in a route.

4.1 Moves

A move is a way to get from one solution to another solution in the solution space. It is important to ensure a *continuity* when going from one solution to another, otherwise the search will be totally random. The moves represented in this section ensure continuity. The primary moves in the VRPTW can be split up into 3 different types

1. *Relocating vertices*
2. *Exchanging vertices*
3. *Exchanging edges*

These moves are described in [Sav92] page 151-153, where they are named *relocate*, *exchange* and *cross*.

The relocation of vertices between routes is illustrated with two routes in figure 4.2 where the vertex i is relocated from the upper route to the lower route.

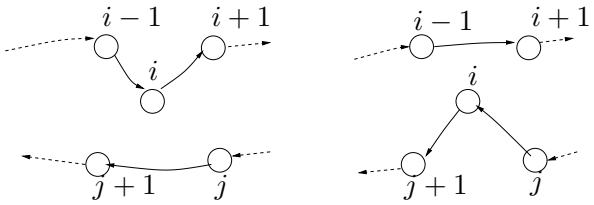


Figure 4.2: Relocating vertex i

The relocation of vertices can also be applied within a route. When dealing with VRPTW the relocation between routes is preferred, because of the time

windows and the opportunity of exploring more different new solutions. By performing the relocation between routes, one can also end up having made a relocation within a route. Or considered a new variant of relocating in [Or76], which introduced relocating a chain of consecutive vertices instead of only moving one vertex.

Exchanging vertices are illustrated at figure 4.3, where the two vertices i and j are exchanged. This correspond to performing a relocating twice, where the vertex i is relocated in one move and the other vertex j is relocated in the other move.

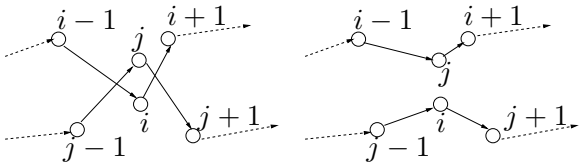


Figure 4.3: Exchanging vertices

An extension of exchanging vertices could start in the idea of relocating chains of vertices from one route to another, where a chain of the last visits in one route is exchanged with another chain of the last visits in another. This exchange corresponds to an exchange of edges also called a cross as seen in figure 4.4, where the edges $(i, i+1)$ and $(j, j+1)$ are exchanged with the edges $(j, i+1)$ and $(i, j+1)$.

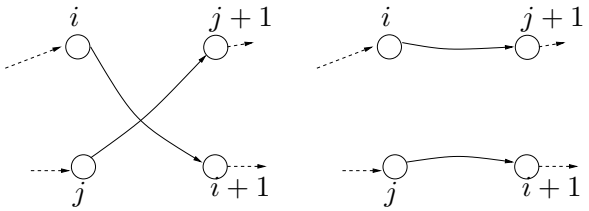


Figure 4.4: Exchanging edges

Exchanging two edges is also named a *2-opt exchange*. This procedure can also be expanded to for instance a *3-opt exchange* and so on. These exchanges were introduced by Lin in [Lin65] For each move there are two opportunities for how to perform the move: within a route or between routes. There first opportunity will only perform well in problems, where the time windows of the visits are very wide. This opportunity will not be used, because the VRPTWSV does not necessarily have wide time windows. The other opportunity fits well the VRPTWSV, when the cost ψ is large, because one has the opportunity of moving visits from routes with non-regular caretakers to routes with regular caretakers.

All the moves presented here are all simple and quick moves, but the relocating is the most simple, because the other two moves can be performed by doing more than one relocation. The thesis [Brä01] compares different between-routes improvement heuristics. The results of the comparison is listed in table 9, page 116 in the thesis [Brä01]. The results for relocation, exchanging of nodes or edges show that the relocation move seems to perform better than the other quick moves. Based on these good results, the relocation move is chosen to be applied in the tabu search in this project.

4.1.1 The Implementation of a Move

In the VRPTWSV only the unlocked visits can be moved. The locked visits e.g. start visits and breaks are locked to a caretaker, and can therefore not be moved.

Performing a move involves two functions. One for removing a visit v from a route $\bar{r}$ and one for inserting the visit in another route r^*. The insertion function in algorithm 4 in chapter 3 is used again without the sets $\underline{\mathcal{V}}$ and $\bar{\mathcal{V}}$. The push forward function in algorithm 6 is also used again with the change that it only performs the push forward within one route. The push forward does not pay attention to the starting time of the shared visits, because the starting times do not need to be synchronous, when constraint (2.22) is relaxed. The starting time of a inserted visit i is set to $\max\{l_i, a_i\}$, where the arrival time l_i is given by (3.1). The insertion and push forward function is changed to calculate the new total violations of latest starting times given in (4.1), the new total violation of same starting times for shared visits given in (4.2) and the new total violation of latest finishing times for the workers given in (4.3).

The new total violation of latest starting time $A(\hat{x})$ is calculated by adding the differences between the old and the new violations for all visits in route r^* succeeding the inserted visit, including the inserted visit v and the previously succeeding visits in $\bar{r}$. The calculation is

$$A(\hat{x}) = A(x) + \sum_k (\max\{\hat{s}_k - b_k, 0\} - \max\{s_k - b_k, 0\}),$$

where k belongs to the set of visits including v and the succeeding visits in route r^* or the previously succeeding visits in route $\bar{r}$.

The new total violation of the synchronous starting times for a shared visit $B(\hat{x})$ is only calculated if the inserted visit v or if one of the succeeding visits i in the route r^* or $\bar{r}$ is shared. In such case the difference between the old

and new violation of synchronous starting times for those eventually shared visits is added to the total violation of the synchronous starting time for the shared visits in this manner

$$B(\hat{x}) = B(x) + \sum_{k,j}(|\hat{s}_k - \hat{s}_j| - |s_k - s_j|)$$

where k is a visit in the set of the shared visits which may include v and the succeeding visits in route r^* or the previously succeeding visits in route $\bar{r}$ if they are shared and j is the other half of each eventually shared visit k.

The new total violation of the latest finishing time is calculated by finding the old and new latest finishing times in the route r^* and $\bar{r}$. The difference is added to the total violation of the latest finishing times

$$\begin{aligned} G(\hat{x}) &= G(x) + \max\{\hat{f}_{r^*} - h_{r^*}, 0\} - \max\{f_{r^*} - h_{r^*}, 0\} + \\ &\quad \max\{\hat{f}_{\bar{r}} - h_{\bar{r}}, 0\} - \max\{f_{\bar{r}} - h_{\bar{r}}, 0\} \end{aligned}$$

where f_{r^*} is the finishing time of the last visit in route r^* in the old solution x and the $\hat{f}_{r^*}$ is the finishing time of the last visit in the route r^* in the solution $\hat{x}$, which could be a new visit, if v is inserted at the last position. The notation is similar for route $\bar{r}$.

When removing the visit v, a push-back function is defined. It is very simple. The visit after v has defined a new starting time, which is the maximum of the new arrival time and the earliest starting time. The difference of the old and the new starting time is the push backward for all the succeeding arrival times in the route considered. The new starting times of the succeeding visits k is $\max\{l_k, a_k\}$. This push backward function does again not pay any special attention to the shared visits, because the starting times do not need to be synchronous, when constraint (2.22) is relaxed.

The push functions used in the implementation of a move do only pay attention to those times, that are delayed e.g. the starting times greater than the latest starting times and the finishing times greater than the latest finishing times. The push functions can not pay attention to the starting times, that can be too early e.g. a part of a shared visit starts before the other. In section 4.7 it is suggested how to avoid this problem by solving a LP-problem. A comparison of the strategies is in section 4.8, where the problem with the push functions is illustrated by an example in subsection 4.8.1.

4.2 Neighbourhoods

The *size* of the neighbourhood can be too small or too big. Different sizes of neighbourhoods are tried in this project for making a comparison.

The whole neighbourhood $N(x)$ would be to take all unlocked visits in x and try to relocate them to all other positions. This neighbourhood is very big, because each visit v can be inserted in $\mathcal{O}(n)$ positions, and because there are unlocked $\mathcal{O}(n)$ visits there would be $\mathcal{O}(n^2)$ solutions in $N(x)$. The three neighbourhoods N_1, N_2 and N_3, which are smaller parts of the whole neighbourhood, are firstly investigated.

1. The first neighbourhood N_1 takes a random visit among the unlocked visits and finds a random route r^* different from $\bar{r}$ for insertion. There are approximately n/m positions in a route, which is equivalent to approximately n/m solutions in $N_1(x)$, so the size of the neighbourhood is $\mathcal{O}(n)$.

2. The second neighbourhood N_2 also takes a random visit among the movable visits and tries all other routes different from $\bar{r}$ for insertion. There are $\mathcal{O}(n)$ solutions in $N_2(x)$

3. The last neighbourhood N_3 takes a random route $\bar{r}$ where it uses all the visits in $\bar{r}$ to be inserted in all other routes different from $\bar{r}$. There are approximately n^2/m solutions in $N_3(x)$, so the size of the neighbourhood is $\mathcal{O}(n^2)$.

Using random functions for finding v, $\bar{r}$ or r^* ensures better *coverage*, where solutions spread out in the solution space S can be reached.

Using the approach with smaller neighbourhoods N_1, N_2 and N_3 turns out not to be suitable for this type of problem, where it is possible to violate both the latest starting times for the visits, the latest working hours for the caretakers and the synchronous starting times for shared visits. The reason for this is found in the randomness. In neighbourhood N_1 the situation can arise where many time windows are violated, but because of the random function there is very little chance of finding all the visits, where the time windows are violated, and move them in a better position.

In neighbourhood N_2 the chance of finding the visits, where the time windows are violated, is larger and it is even larger for the neighbourhood N_3. Equally the chance of finding the route, where the working times is violated is also increasing from N_1 to N_2 and more to N_3. The figure 4.5 shows an instance, where the vertical axis show the total violations versus the number

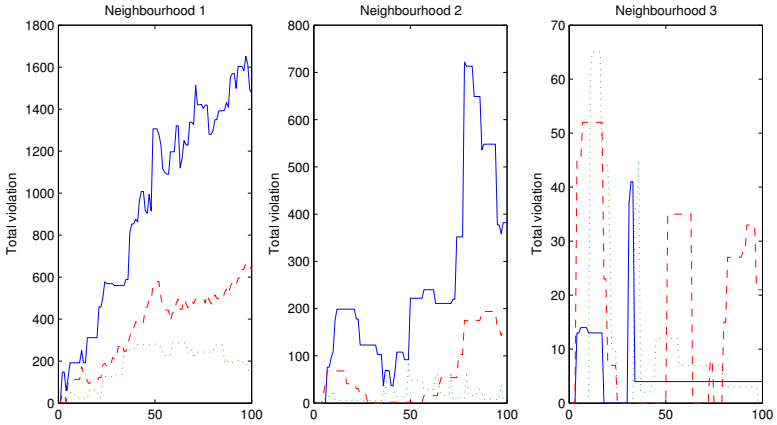


Figure 4.5: The solid line shows the violation on time windows, the dotted line shows the violation on synchronous starting times for shared visits and the dashed line shows the violation on working hours

of iterations. The instance is data from 17th of February, with $\mu = 15$ and the tabu search is performed with $\theta = 10$, $\lambda = 2$ and $\delta = 1$.

It is observed leftmost in figure 4.5 how the total violation for time windows, the starting times for shared visits and workinghours is increasing very much. In the middle in figure 4.5, the plot shows less increasing violations.

The rightmost plot shows how the total violations are variating much more, but only initially they all are zero, which means no feasible solution is found.

The example in figure 4.5 illustrates why none of the neighbourhoods N_1, N_2 or N_3 are used for further research. Instead the whole neighbourhood is used in the tabu search.

4.3 Tabu Strategy

The tabu strategy used in this project is the one represented in the article [CLA01]. When removing a visit v from a route $\bar{r}$, it is tabu to reinsert the visit for the next θ iterations. This strategy is very appropriate for the VRPTWSV, because it uses only a small part of the solutions without saving the whole solution, which is another tabu strategy. The visit v in $\bar{r}$ is called an *attribute* of the previous solution, and hence the tabu strategy is based on attributes.

The number of iterations θ is static in the article and in this project, but

one could think of situations, where a dynamic θ would be preferable e.g. depending on the search. Good solutions can force a small θ and bad solution can force a larger θ, see the article [CR97] on reactive tabu search.

4.4 Diversification

To ensure a *coverage* of the solution space, it is necessary to introduce a diversification in the tabu search. In the article [CLA01] they introduce a diversification based on the parameter ρ_r^v, which is the number of times a visit v has been added to a route r during the search. For every solution the values ρ_r^i are summed for the visits in all the routes.

Diversification also depends on the size of the problem indicated by $\sqrt{nm}$, because the larger the problem is, the larger is the need for diversification. The need for diversification is also larger if the solution value $C(\hat{x})$ is large, because the solution is of poor quality. The need for diversification is finally also determined by a diversification factor λ, which is adjustable, because the need for diversification can depend on the problem structure.

The total penalty function is

$$P(\hat{x}) = \begin{cases} \lambda C(\hat{x})\sqrt{nm}\sum_{i,j\in\mathcal{V}}\sum_{r\in\mathcal{R}}\rho_r^i x_{ijk} & \text{if } F(\hat{x}) \geq F(x) \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

The penalty function $P(\hat{x})$ is zero, if the solution $\hat{x}$ is as good or better than the previous solution.

When choosing a solution $\hat{x}$ in the neighbourhood $N(x)$ the function $F(\hat{x})$ in 4.4 is used as well as a new penalty function $P(\hat{x})$. The best solution in the neighbourhood $N(x)$ is the one with the smallest $F(\hat{x}) + P(\hat{x})$ value.

4.5 Aspiration Criterion

The aspiration criterion is the condition for when to override the tabu status of a best solution $\hat{x}$ in the neighbourhood $N(x)$. Stefan Røpke suggests a number of different conditions in the slides [Røp05].

- When the solution $\hat{x}$ is the global best.
- When the solution $\hat{x}$ is the best in the region, e.g. the region of certain properties.

- When the solution $\hat{x}$ is the recent best in the recent iterations

The aspiration criterion gives a *guidance* to the search. The second aspiration criterion is the one chosen in the article [CLA01] and is therefore also used in this project, because the VRPTWSV is very similar to the problem in the article. The first aspiration criterion has been tried in this project, but is does not perform well, because the tabu strategy is based on the attributes. If the costs $F(\hat{x}_1)$ or $F(\hat{x}_2)$ of the solutions $\hat{x}_1$ or $\hat{x}_2$ with v in route r^* or in $\bar{r}$ are lower that the best known cost $F(x^*)$, the visit v can be moved back and forth between the routes r^* and $\bar{r}$ according to aspiration criterion 1, for many iterations until α, β and γ are large enough for the costs $F(\hat{x}_1)$ or $F(\hat{x}_2)$ not to be smaller than $F(x^*)$ anymore. Therefore the aspiration criterion should be based on the attributes. For each attribute is the best known cost saved, and when an attribute is tabu, it is checked if the solution cost is better than the best known for that attribute.

4.6 Stopping Criterion

There are many possibilities for stopping criterions, and some of them are listed here.

- No. of iterations
- No. of iterations without improvement
- Time

The first stopping criterion is the one used in this project, because it is easy to apply and interpretate.

4.7 A New Relaxation: LP-model

The tabu search includes push functions, which do only pay attention to delayed starting times. Instead of using push functions, when inserting or removing a visit, a LP-problem can be solved. The LP-model can be modelled in order to pay attention to the starting times, which are too early.

The tabu search heuristic uses the condition, that the good solutions may be on the border to infeasilibity in the VRPTWSV. It only uses the part of the

border where the latest starting time at a visit is violated, the latest finishing time at a route is violated or the synchronous starting times for shared visits are violated. The figure 4.6 illustrates the situation. The feasible solution space is indicated by a thick line surrounding it. The thick line indicates the border to infeasibility. The dotted line is the border surrounding the solutions reached during the tabu search. Some of solutions were infeasible, because they were outside the feasible solution space, but notice that the border to infeasibility is not reached all over solution space.

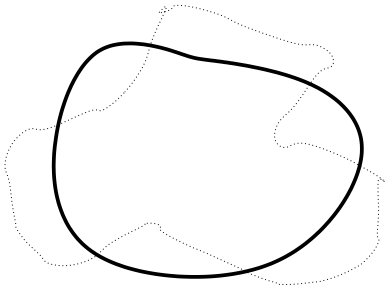


Figure 4.6: The solution space

The rest of the border to infeasibility is used to spread the search, where also the earliest starting time at visit can be violated and the earliest starting time at a route can be violated. It correspond to removing the constraints (2.26) and (2.23) in the mathematical model of the VRPTWSV model. The figures 4.7, 4.8 and 4.9 illustrate the scenarios which can arise.

Instead of only violating the latest starting time for the visits, it could also be possible to violate the earliest starting time as seen in figure 4.7, where the three possible scenarios are depicted. The new total violation of the time windows is $\hat{A}$.

$$\hat{A}(x) = \sum_{i \in \mathcal{V}} (\max\{a_i - s_i, 0\} + \max\{s_i - b_i, 0\}) \tag{4.6}$$

As seen in figure 4.8 it is be possible to violate both the latest finishing time and the earliest starting time. The earliest starting time is $s_r = \min\{s_i | i \in r\}$ and the latest finishing time is $f_r = \max\{f_i | i \in r\}$. The total violation of the working hours is given in (4.7).

$$\hat{G}(x) = \sum_{r \in \mathcal{R}} (\max\{g_r - s_r, 0\} + \max\{f_r - h_r, 0\}) \tag{4.7}$$

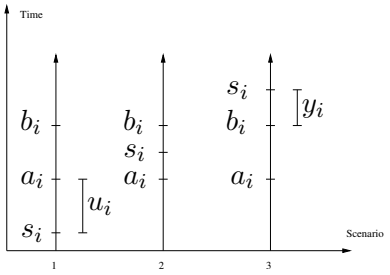


Figure 4.7: The three scenarios possible for visit i

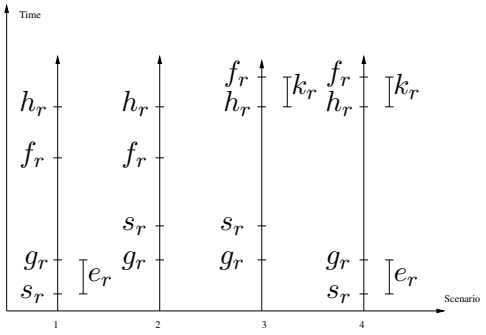


Figure 4.8: The four scenarios possible for route r

The violation of the synchronous starting time for shared visits is already seen in the beginning of this chapter, and is displayed in figure 4.9. The total violation of this constraint is still

$$\hat{B}(x) = B(x) = \frac{1}{2} \sum_{i,j \in \mathcal{V}} \omega_{ij} |s_i - s_j|$$

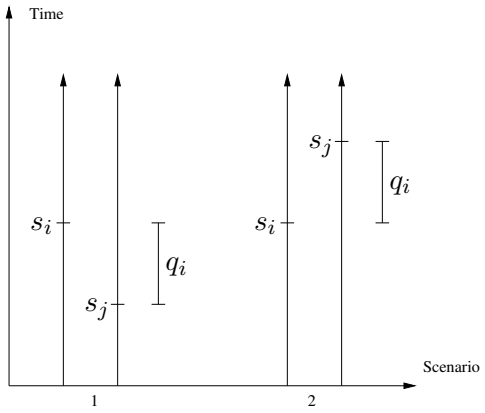


Figure 4.9: The two scenarios for the couple of shared visits i and j

When inserting a visit v in a given position in a route r^*, the starting time of the inserted visit can be set in various ways. The starting time of the visit interacts with the other visits in the route r^*. Instead of "just" setting the starting time for v and letting the other visits adapt (be pushed forward) to that starting time, the LP-model (4.8)-(4.13) finds the starting times for all the visits in the route where the objective is to minimize the violation of the time windows, working hours and starting times for shared visits for that route.

$$\min \alpha \sum_{i \in r^*} (u_i + y_i) + \beta \sum_{i \in r^*} q_i + \gamma (e_{r^*} + k_{r^*}) \tag{4.8}$$

$$
\begin{aligned}
s_i + d_i &= f_i \quad \forall\, i \in \text{route } r^* && (4.9)\\
f_i + \sum_{z_1 z_2} \sigma_{iz_1}\sigma_{jz_2} t_{z_1 z_2} - M(1 - x_{ijr^*}) &\leq s_j \quad \forall\, i,j \in \text{route } r^* && (4.10)\\
a_i - s_i &\leq u_i \quad \forall\, i \in \text{route } r^* && (4.11)\\
s_i - b_i &\leq y_i \quad \forall\, i \in \text{route } r^* && (4.12)\\
g_{r^*} - s_i &\leq e_{r^*} \quad \forall\, i \in \text{route } r^* && (4.13)\\
f_i - h_{r^*} &\leq k_{r^*} \quad \forall\, i \in \text{route } r^* && (4.14)\\
(s_i - s_j)\omega_{ij} &\leq q_i \quad \forall\, i \in \text{route } r^* && (4.15)\\
(s_j - s_i)\omega_{ij} &\leq q_i \quad \forall\, i \in \text{route } r^* && (4.16)\\
s_i,\ u_i,\ y_i,\ q_i,\ e_{r^*},\ k_{r^*} &\geq 0 \quad \forall\, i \in \text{route } r^* && (4.17)
\end{aligned}
$$

The model uses the information on the order of the visits in the route to decide when to let the visits start. The model is also used for the route $\bar{r}$, where the visit v is removed, instead of "just" pushing the visits back. Notice that if the visits i and j constitute a shared visit, the starting time s_j for the other visit is a parameter.

4.7.1 Implementation of a Move

The solution x is changed to the solution $\hat{x}$ by performing a move consisting of removing visit v from route $\bar{r}$ and inserting it in route r^*. The LP-model (4.8)-(4.13) can be solved to optimality for both the route $\bar{r}$ and r^* using a solver, e.g. *Cplex*.

After solving the LP-problems the total violations $\hat{A}(x)$, $\hat{B}(x)$ and $\hat{G}(x)$ are updated in another way than the one represented in subsection 4.1.1, because not only the succeeding visits in r^* after v change their starting times and the previously succeeding visits in $\bar{r}$, but all visits in both r^* and $\bar{r}$.

The total violation of the time windows is updated by calculating the difference between the old and the new violation of the time windows for all the visits in the routes r^* and $\bar{r}$. The difference is added to $\hat{A}$.

$$\hat{A}(\hat{x}) = \hat{A}(x) + \sum_{i \in r^*, \bar{r}} (\hat{u}_i - u_i + \hat{y}_i - y_i)$$

where $u_i = \max\{a_i - s_i, 0\}$ and $y_i = \max\{s_i - b_i, 0\}$, which are given in the optimal solutions for the LP-problem in (4.8)-(4.13).

The updating of the time differences between the shared visits is calculated using the old and new violation for the shared visits in the routes r^* and $\bar{r}$.

$$\hat{B}(\hat{x}) = \hat{B}(x) + \sum_{i \in r^*, \bar{r}} (\hat{q}_i - q_i)$$

The total violation of working hours is updated after inserting visit v in route r^* by using the old and the new violation in that route.

$$\hat{G}(\hat{x}) \quad = \quad G(x) + \sum_{r \in \{r^*, \bar{r}\}} (\hat{k}_r - k_r + \hat{e}_r - e_r)$$

The optimal solutions found by a solver will be used for setting the start values for the visits in the routes r^* and r, and there is no need for a push forward or backward function. This tabu search approach with solving a LP-model is implemented, but because of lack of time, it has not been tested.

4.8 The Strategies for Inserting and Removing a Visit

When inserting visit v in route r^* or removing v from route $\bar{r}$, new starting times for some of the visits are found. There are different strategies for finding the new starting times. In this section 3 kinds af strategies will be compared.

Strategy 1: The starting times for the visit v, the succeeding visits in route r^* and the previously succeeding visits in route $\bar{r}$ are found by using push functions.

Strategy 2: The starting times for all visits in route r^* or $\bar{r}$ are found by using a LP-model.

Strategy 3: The starting times for all visits in all routes are found by using a LP-model.

4.8.1 Strategy 1

When inserting a visit v, its new starting time is found, while the succeeding visits are just pushed forward. The same problem arises when removing a visit from route $\bar{r}$. The succeeding visits in route $\bar{r}$ are just pushed backward.

The new starting times found are the earliest possible, and in this subsection it is shown how it in some cases is the best and in other cases it is not the best.

If there are no shared visits in the data, the solution can only be infeasible when the latest starting time b_i or the latest finishing time h_r is violated. This is only true, when the arrival times are set according to (3.1) and the starting times are $\max\{l_i, a_i\}$. The starting times are set to be as soon as possible, which is also a good idea, if one wants to minimize the violations. The example in the figures 4.10 and 4.11 explains why.

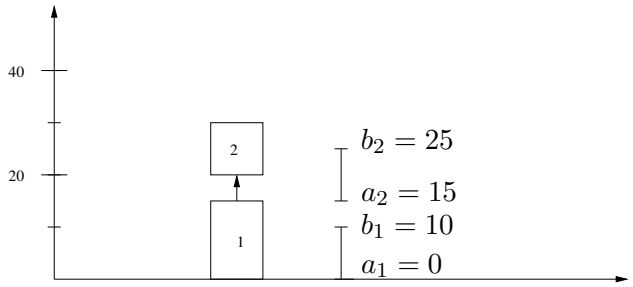


Figure 4.10: The cost

The example in figure 4.10 shows how the visit 1 is inserted in route r^* before visit 2, which is pushed forward. Their time windows are $[a_1, b_1]$ and $[a_2, b_2]$. The latest finishing time h_{r^*} for this route r^* is assumed to be sufficiently late to avoid being violated.

The price α for violating a latest starting time b_i is set to $\alpha = 2$. The figure 4.11 shows the violation cost for violating the latest starting times as a function of the starting time s_1. When inserting the visit 1 at its earliest feasible insertion time $s_1 = 0$, the violation costs are zero, as shown in figure 4.11. After 5 minutes b_2 is violated and the cost increases, and after 10 minutes also b_1 is violated and the slope of the cost function increases again.

This example in the figures 4.10 and 4.11 illustrates, why the best starting time is earliest possible when it is only possible to violate the latest starting time at a visit and the latest finishing time at a route.

In the first tabu search, it was possible to violate the latest starting time b_i, the latest finishing time f_r and that the starting times for the shared visits have to be synchronous. The earliest possible starting time $\max\{l_i, a_i\}$ is used when inserting visits, even though the following example in figure 4.12 and 4.13 shows, that it is not always the best.

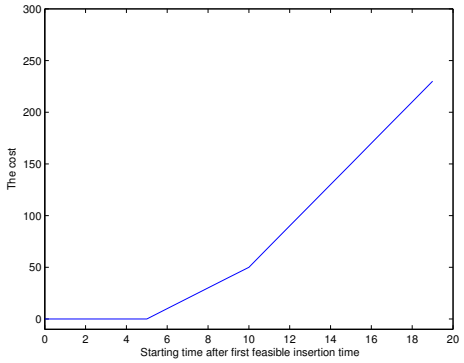


Figure 4.11: The cost

When the data includes shared visits, and it is possible to violate the synchronous starting times for shared visits the violation cost is not necessarily increasing monotonously. An example could be having a shared visit in a route, where a visit is inserted before the shared visit. The shared visit on this current route starts too early in comparison with the other part of the shared visit, which is placed in another route. Therefore it might be preferable to start the inserted visit later to push the shared visit closer to its other half.

The figure 4.12 shows an example of how the visits can be situated after inserting visit 1 at its earliest possible starting time.

The visits 1 and 3 are not shared visits with the time windows $[a_1, b_1]$ and $[a_3, b_3]$. The visits 2, 4, 5 and 6 in figure 4.12 are shared visits and their time windows are assumed to be wide enough for not being violated in this example. The other halfs of the shared visits are named $\hat{2}$, $\hat{4}$, $\hat{5}$ and $\hat{6}$ and their starting times are 35, 85, 110 and 135. The latest finishing time h_{r^*} for this route r^* is also assumed to be late enough to avoid being violated in this example.

For every starting time s_1 a violation cost is defined. The violation cost is only calculated for route r^* as the violation of the latest starting times for visit 1 and 3 and difference in starting times from their other part in the shared visits 2, 4, 5 and 6, when $\alpha = 2$ and $\beta = 10$. The figure 4.13 shows the violation cost as a function of s_1, where the origin is set to the first feasible insertion time.

The cost in figure 4.13 is monotonously decreasing from $s_1 = 0$ to 15. This is caused by the fact that visit 2 is moved closer to $s_{\hat{2}}$. Within this interval small changes in the slope occurs to make the cost less decreasing. The

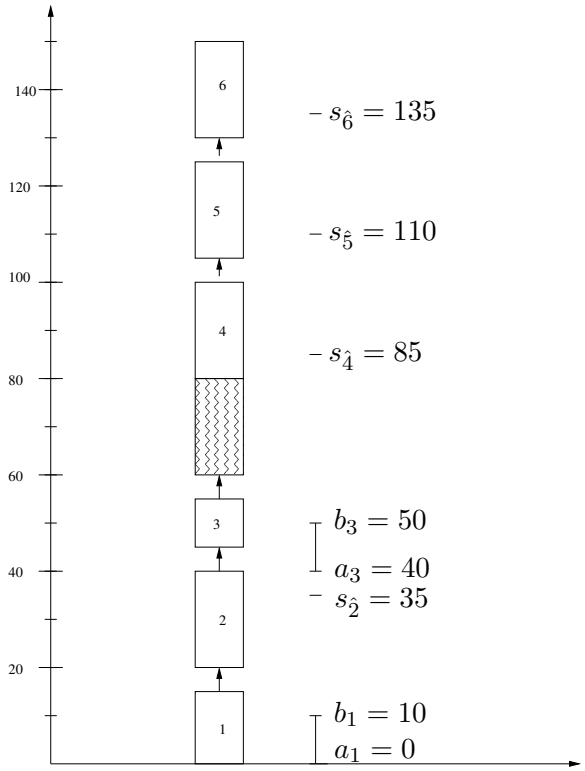


Figure 4.12: The cost

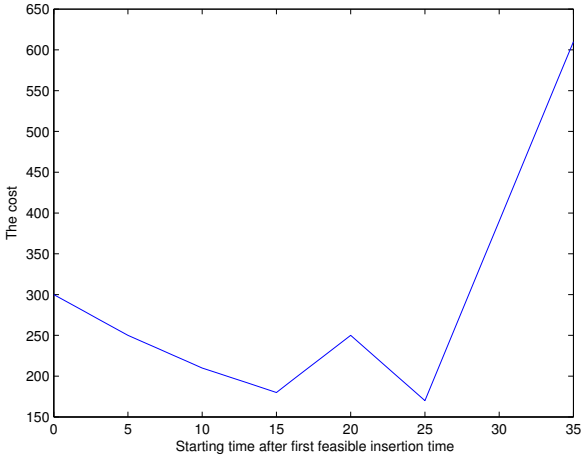


Figure 4.13: The cost

changes occur after $s_1 = 5$ because visit 3 starts too late and after $s_1 = 10$ when visit 1 starts too late. After 16 the cost increases, when visit 2 is pushed away from $s_{\hat{2}}$. After $s_1 = 20$ the cost decreases again, because the visits 4,5 and 6 are pushed closer to $\hat{4}$, $\hat{5}$ and $\hat{6}$. At $s_1 = 25$ a global minimum is reached when the visits 4, 5, and 6 reach $\hat{4}$, $\hat{5}$ and $\hat{6}$. Afterwards the costs only increases as the shared visits 4, 5 and 6 are pushed away from the other halfs.

The changing of the parameters α, β and γ complicate the situation even more, because in some cases it would be preferable to have big time differences between the starting times of shared visits and less violation of the latest starting times on visits, but in the next iterations, it might not be that preferable anymore. The positive property of this insertion strategy is that it can be done in approximately constant time, and if there are not many shared visits, it also performs well. For this reason it is chosen as the strategy used in the tabu search.

4.8.2 Strategy 2

The tabu search with the LP-model introduced in section 4.7 finds the best starting times for all the visits in route $\bar{r}$ or r^*, when taking the violation of the time windows, working hours and starting times for the shared visits into account. This gives a violation cost, which is better or the same as in strategy 1.

The LP-problem is solved in polynomial time, but the tabu search performs slow, because of the interaction between the solver and the main function.

4.8.3 Strategy 3

The optimal insertion strategy is to find new starting times for all visits in all routes, when inserting or removing a visit. This strategy should minimise the violation cost. The strategy gives the best violation cost. This problem demands a greater computational effort, but is still solved in polynomial time.

Chapter 5

The ABP Programme

The abbreviation ABP is short for Automatic Visit Scheduling (in Danish "Automatisk BesøgsPlanlægning"). René Munk Jørgensen and Jesper Larsen from DTU have developed the ABP programme for scheduling visits in the home care sector. This programme was initiated as a research project for the company Zealand Care in September 2004 and taken over 20th December 2005. The programme is tested in Søllerød municipality January - March 2006. There is a very short introduction on the company Zealand Care in section 5.1

The insertion heuristic and tabu search introduced in chapter 3 and 4 are compared with the ABP programme. The problem in ABP is explained in section 5.2 and the solution method used in ABP is described in section 5.3.

5.1 Who is Zealand Care?

The company is described at the homepage [Car]. It is a Danish joint-stock company with 132 employees at the end of 2004. The company was founded in 1995. The company is centered on the social- and health care industry. The main business activities concern disability equipment services, consulting and information technology.

5.2 The ABP Problem Compared with the VRPTWSV

The purpose of the programme is to make a long term schedule for the visits in a home care district subject to a number of constraints. The value of a schedule is determined by different objectives.

It is possible to adjust the parameters for the visits, citizens and caretakers and thereby changing the constraints and objectives in the ABP problem. Because the VRPTWSV is a problem with less constraints than the one solved by ABP, some of the parameters are set to zero or switched of to make a fair comparison of the methods.

5.2.1 The Constraints

There are more constraints in the ABP problem than in the VRPTWSV. A solution for the ABP problem is always a solution in VRPTWSV. Figure 5.1 is a plot of the solution space $\mathcal{S}_2$ for the ABP problem and the solution space $\mathcal{S}_1$ for the VRPTWSV.

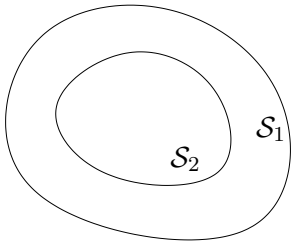


Figure 5.1: Solution spaces

In $\mathcal{S}_1$ it could be possible to get a better solution value than in $\mathcal{S}_2$, because there are the same and more solutions. To make a better comparison of the solution methods for the two problems, the ABP problem is relaxed, which means that the solution space $\mathcal{S}_2$ is enlarged to almost reach $\mathcal{S}_1$.

In the following the constraints for the ABP problem are described. It is explained for each constraint if it is relaxed in the comparison and how it is done.

Visits' Time Windows For each visit is a time window, which indicates the earliest starting time and latest finishing time. The time windows in the VRPTWSV indicate the earliest and latest starting time. This first type of time window is converted to the latter one in the comparison by subtracting the duration of the visit from the latest finishing time.

Locking Visits It is possible to lock a visit in different ways. It can be locked to a specific caretaker to ensure the same caretaker is attending the visit always. The visit can also be locked by time, and will therefore always be performed at the same time. Finally it can also be locked

to be performed in a certain time period of minimum one day. The start visits and the breaks will be locked by caretaker, but all the other visits will remain unlocked in the comparison case.

Series of Visits In ABP it is possible as mentioned before to make a schedule for more than one day. It is possible to design series of visits. A series could for instance be a cleaning visit Thursday afternoon every second week, or help with personal hygiene every morning all week. Because the VRPTWSV only focuses on one day, this will not affect the comparison.

Overlap of Visits Visits with certain kinds of demands may overlap. An example could one caretaker cleaning the house and another caretaker giving the citizen a bath. Because the VRPTWSV does not operate with demands, this is not possible when making the comparison.

Shared Visits It is possible to create a shared visit by making two visits and linking them together. They can be linked in different ways for instance if the caretakers have to start at the same time or if one of the caretakers has to start minimum half an hour before the other caretaker. In the comparison only the linking with caretakers starting at the same time was intended to be used, because the VRPTWSV does only take this situation into account. Unfortunately this function was not working in ABP, when the comparison was made.

Extra Time Between Visits A minimum extra time between the visits can be set. The function will not be used in the comparison.

Forbidden Caretakers at Citizens Some caretakers do not want to visit a citizen because of a social factor or other factors. The other factor could be if the citizen has pets and the caretaker is allergic to pets. This can also be the other way around if the citizen does not want visits from a specific caretaker. Such relationships are not present in the VRPTWSV, and the lists of forbidden relationships between caretakers and citizens are emptied in the comparison.

Succeeding Visits Two citizens could live together. In such situation it is possible to assure, that the two citizens get a visit right after each other by the same caretaker. This function is not used, when the comparison is done.

Caretakers' Working Hours Each caretaker has an earliest starting time and a latest finishing time. These working hours are equivalent to the ones in the VRPTWSV, and are therefore not changed in the comparison.

Caretakers' Transportation Means It is possible to set the type of transportation mean for each caretaker. In the comparison there is no discrimination between different kind of transportation means, and therefore all transportation means are set to cars in the comparison.

The solution space $\mathcal{S}_1$ is enlarged to almost reach $\mathcal{S}_2$ by performing the changes listed above.

5.2.2 The Objectives

There are two special issues to take into consideration when comparing the solutions found by the methods in the ABP programme and the solutions found by the methods in this project.

1. The objective function in the VRPTWSV is used for comparison, because the objective function in ABP is not known exactly. This may cause a poor comparison, and therefore to make the comparison as good as possible the objective function in ABP is approximated to the objective function in the VRPTWSV
2. There may arise a problem, if the good solutions with respect to the objective function in the VRPTWSV is situated in the solution space $\mathcal{S}_1 \setminus \mathcal{S}_2$. This will only be a problem if $\mathcal{S}_1 \subset \mathcal{S}_2$.

There are given examples of what can happen if the objective function in ABP is different from the objective function in the VRPTWSV, and examples of how the difference between $\mathcal{S}_1$ and $\mathcal{S}_2$ can give a poor comparison.

At the end of this subsection it is explained how the objective function in the ABP programme is approximated to the objective function in the VRPTWSV.

It it easier to compare the two objective functions, if we assume the two solution spaces $\mathcal{S}_1$ and $\mathcal{S}_2$ to be equal. In figure 5.2 it is illustrated how there could be two objective functions; one for the ABP problem and one for VRPTWSV.

The two optimal solutions are x_1^* and x_2^* with the values c_1^* and c_2^*. The two solutions have to be compared. Because only the formula for objective function in VRPTWSV is known it is used for comparison. The solution value for x_2^* is c_2, which is worse than c_1^* in the example in figure 5.2.

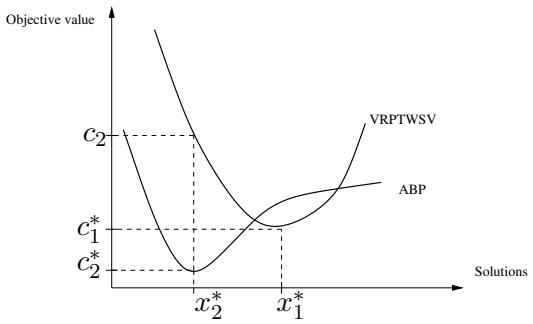


Figure 5.2: Two optimal solutions compared with the objective function in VRPTWSV, when the solution spaces are equal.

The optimal solution value c_2 will not necessarily always be worse than c_1^*, it could also be just as good, but never better.

If the solutions $\hat{x}_1$ and $\hat{x}_2$ with the values $\hat{c}_1$ and $\hat{c}_2$ are found but not necessarily are the optimal solutions, one can not be sure that $c_2 \geq \hat{c}_1$. Figure 5.3 depicts an example of this.

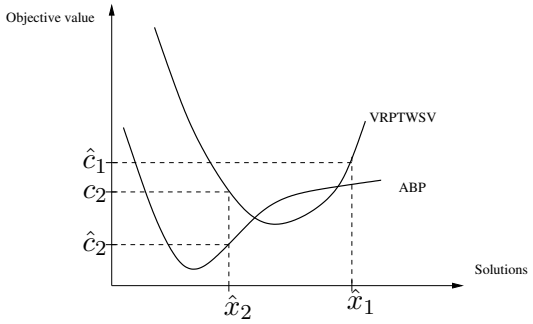


Figure 5.3: Two solutions compared with the objective function in VRPTWSV, when the solution spaces are equal.

It shall be kept in mind, that when performing the method in ABP for finding the solution $\hat{x}_2$ it is not known that the comparative objective value is c_2 and not $\hat{c}_2$ and the method might therefore go to the "wrong" places searching for a good solution $\hat{x}_2$.

Now follows an example of what can happen if $\mathcal{S}_1 \setminus \mathcal{S}_2 \neq \emptyset$ and the good solutions are situated in $\mathcal{S}_1 \setminus \mathcal{S}_2$. Then the method in ABP has worse changes of searching the right places for good c_2 values. An example of this is in figure 5.4, where the horizontal axis illustrates the solution space $\mathcal{S}_1$.

Figure 5.4 shows how it could be not possible to find a solution $\hat{x}_2$ with good values for the objective function in VRPTWSV.

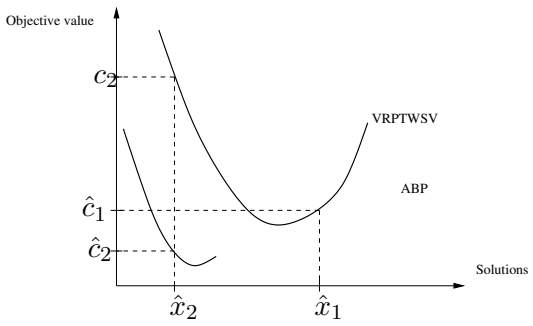


Figure 5.4: Two solutions compared with the objective function in VRPTWSV, when the solution spaces are not equal

In the following the objective function in the ABP problem is described. It is explained how it is approximated to the objective function in VRPTWSW if it is necessary for the comparison.

Demand and Priority for Visits Each visit has a type of demand, which for instance could be personal hygiene or cleaning. The priority of a visit is determined by the type of demand e.g. personal hygiene could be more important than cleaning. The objective is to schedule the most important visit, when not all visits can be scheduled. The VRPTWSV does not distinguish between different types of visits, and therefore all visits are set to have the same demand, which is practical help, when performing the comparison.

Regular Caretakers at Citizens Each citizen has three regular caretakers. The regular caretakers are weighted equal or higher than the non regular caretakers to be the ones performing the visits at each citizen. The weights can be set as parameters, and they are set at different levels in the comparison.

Minimize Caretakers' Travel Time Each caretaker should travel as short time a possible. This objective is the same in the VRPTWSV and will not influence the comparison.

Caretakers' Qualifications and Fitness of Visits' Demands The caretakers can be categorized according to different qualifications. The categories are for instance helpers, assistants and nurses. Each category has different qualifications in different areas such as cleaning, normal personal hygiene, complex personal hygiene etc. A helper is for instance preferred for practical help such as cleaning. An objective is that the qualifications fit the demands as well as possible. There are no demands or qualifications in the VRPTWSV. All demands are

set to practical help in the comparison. All caretakers are defined as helpers to avoid that some caretakers are preferable to others in the comparison.

Caretakers Cost A cost is defined for letting a caretaker start, which is minimized. This cost is set to zero in the comparison.

Caretakers' Spare Time after last Visit The ABP maximizes the spare time after the last visit for each caretaker. The VRPTWSV does not take that into a account, and therefore the solution methods for the VRPTWSV have an advantage when the comparison is done with the objective function in the VRPTWSV.

The weights in the objective functions are difficult to approximate to each other, and hence different parameter settings are tried. The other difference between the objective functions is the maximization of the spare time after the last visit.

5.3 The Solution Method

The method is described in the Programme for Development, which is not yet published and hence not available. It consists of several phases, where the main ideas will be explained in this section.

The first two phases form a two-phase method as described by Olli Bräysy on page 23 in [Brä01]. In the first phase the vertices are assigned to vehicles, and in this case it corresponds to assigning the visits to caretakers. Which visit to assign to which route is determined by a score. The score reflects the different factors such as if the caretaker is regular at the visit, the travelling time, the caretakers qualifications and other factors.

The second phase schedule the route for each caretaker. Scheduling the visits in a route is a travelling salesman problem. Finding a solution involves finding the order of the visits and their starting times, such that the time windows are not violated.

The last phases have three purposes: inserting the remaining visits, adapt the solution to the demands or inexpediences and reoptimize the solution.

When performing the two first phases not all visits are necessarily inserted in the solution. There are different approaches to insert the visits. One approach is trying to insert the remaining visits immediately after the first two phases or when there is more space after a reoptimization. Another approach is to remove some of the already inserted visits with worst scores

to give space for the remaining visits. Those removed visits will be tried inserted later. The insertion of a not scheduled visit is done in such manner that the visits with best scores are tried inserted first. This insertion strategy is the same and independent of whether some visits were removed first.

The two first phases do not take into consideration the connection between shared visits nor the connection between days. Some of the demands may be violated and there may also be inexpediences. Therefore the solution is adapted to respect the demands and take the inexpediences into consideration.

The method also contains phases for improvement of the solution. The inter-route exchange such as described on page 152 in [Sav92] is used. The concept is illustrated in figure 5.5.

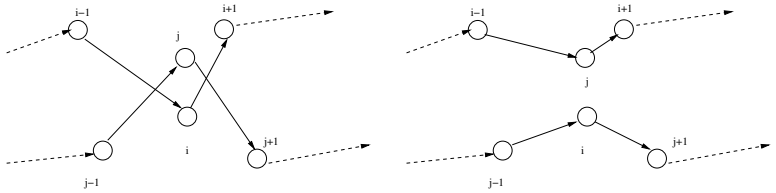


Figure 5.5: Swap the two visits i and j

Two visits in two different routes exchange position as illustrated in figure 5.5. The improvement is done for routes within days and between days.

The solution method is iterative and uses tools from different areas in operational research, because there are many different types of demands to take into consideration.

Chapter 6

Results

The insertion heuristic and the tabu search is tested. The data used for the tests is described in 6.1. The sections 6.2 and 6.3 contain information about how the data is preprocessed and the data instances.

The insertion heuristic is tested in section 6.4. Before testing the tabu search in section 6.6, the parameters are tuned in section 6.5.

The results are compared with the results from ABP in section 6.7.

6.1 Description of the Data

The data used for parameter tuning and performance testing is from Søllerød municipality in Denmark. The municipality had 31900 habitants by the end of 2005, where 6572 of these were above 64 years old. It corresponds 20,6 % of the population, which is higher than the average percentage in Denmark. There were around 1300 citizens, that received home care at that time. These informations on the Søllerød municipality are found at the home page [Ban].

The ABP system has a data bank with informations on visits, citizens and caretakers in one home care district in Søllerød municipality. Some of the data in the data bank was not available when the ABP was developed, because the time windows or the regular caretakers were not defined in that district. These informations are therefore estimated by Zealand Care by using the old schedules, which show when the visits were attended previously and by whom. The time window for a visit is therefore set according to when the visit was attended previously or by defining a time window for each type of demand e.g. a bath in the morning should start between 8 am. and 10

am. The regular caretakers are also assigned to citizens according to whom were visiting the citizen often previously and also by paying attention to the citizens' demands and the caretakers qualifications. There is not payed attention to the locations of the citizens, where a caretaker is regular, and there might therefore be a long distance between them.

The ABP is set to produce text files with the data relevant for this project: travelling times between citizen locations, time windows of visits, duration of visits, the relationship between visits and citizen locations, working hours for the caretakers and the regular caretakers for the visits. The data is read using Java functions, see section B.4.

Identifying a citizen is not done directly from the input file, but by finding unique combinations of location numbers and three regular caretakers. There may be more than one citizen living at a address, but it is not sure that they have the same three regular caretakers. If they have, then they are considered as one citizen when reading the data.

6.2 Preprocessing of the Data

The data is changed to be able to find a feasible schedule for the visits with the insertion heuristic, which needs seed visits (start visits and breaks). The data is also changed in order to investigate how the methods perform on data with shared visits.

Add shared visits: Two of the citizens in the data are given two shared visits each. It is chosen only to have two citizens, that need shared visits, because a shared visit does not appear often normally. A citizen who needs to be lifted, normally needs the help every day, and hence the shared visits are the same every day.

Changing time windows: The time windows are as before mentioned estimated. Sometimes they conflict with the travelling times. There are for instance visits that can only start 5 minutes after the start visit finishes, but they are situated more than 5 minutes away from the office. In such a situation the time window is changed in order for the caretaker to be able to there on time. Other visits can only be attended at the same time as the break, and in such case, the time window is changed as well.

Add a sufficient number of caretakers: Firstly the number of caretakers is decreased to a point, where some of the visits are not scheduled. Afterwards the number of caretakers is increased iteratively by one

caretaker at the time until the number of caretakers is sufficient for finding an initial feasible solution.

Adjust the number of start visits: The number of start visits does not correspond with the necessary number of caretakers. The start visit starts at 7:45 and continues for 15 minutes, and every caretaker being at work at 7:45 is given a start visit.

Adjust the number of breaks: The number of breaks does not correspond with the number of caretakers working until at least when the break finishes. The break starts at 12.00 and last for 30 minutes. All the caretakers working until at least until 12:30 are given a break. The group has a weekly meeting on Wednesdays, which will be considered as break in the sense that everybody working until at least the finishing time of the meeting have to be there.

Changing the working time: If a worker only works between 8 am and 12 am, she can not have a start visit or a break. Therefore her earliest starting time is set to 7:45. In this way she will also get all the news on the citizens, before starting visiting the citizens.

Set the transportation mean to car: The travelling time between two visits is estimated in ABP by using the distance between the visits. The estimation also uses informations on the average speed if the transportation mean is a bicycle or informations on the speed limits on the roads used to come from one visit to another if the transportation mean is a car. To always have the same travelling times between a pair of visits, the transportation mean is set to a car for all caretakers.

The data is also changed in the manner explained in chapter 5. The changes to the data set from week 11 are made in the graphical user interface in the ABP, which is made by Claus Pedersen. The changes made in the data from week 9 and 10 are made when reading the data, because it is a very time consuming process to change the properties of all the visits manually in the GUI. The addition of shared visits are done when reading the data in all instances.

The data set is realistic with the adjustments made, because the changes made are not deviating from what a planner should do in practice, when a schedule is made, and the distance between the citizens in the district are of an order, that makes it necessary for all caretakers to travel in car.

6.3 The Data Instances

The data used for results is from week 9, 10 and 11 in this year 2006. Only the week days are used, because they include more visits and the visits vary more.

The table 6.1 displays all the problems considered, where shared visits are in the data set. Every day there are 4 shared visits, except the 16th and 17th, March, where one of the two citizens with shared visits does not receive any help, maybe because she is in the hospital. The number of caretakers, citizens, visits, start visits and breaks are displayed in table 6.1. The number of visits also include the start visits, breaks and both parts in the shared visits. The number of citizens also includes the office as one citizen.

Day in year 2006	Number of caretakers	Total number of visits	Number of start visits	Number of break visits	Number of citizens	Lower bound on Ψ	Upper bound on T	Average travel-ling time
27.02a	20	166	19	16	106	23	2090	5.80
28.02a	19	169	18	14	108	27	1975	5.59
01.03a	19	168	16	19	92	18	2101	5.70
02.03a	19	185	17	17	111	31	2452	5.79
03.03a	20	185	18	18	118	25	2369	5.74
06.03a	18	162	17	15	97	21	1991	5.81
07.03a	18	173	17	15	106	23	2040	5.77
08.03a	18	165	16	19	93	19	2090	5.52
09.03a	19	180	17	18	104	31	2301	5.69
10.03a	19	179	17	18	100	22	2335	5.68
13.03a	19	165	17	16	92	22	2104	5.84
14.03a	18	166	16	14	100	27	2021	5.70
15.03a	18	151	16	18	83	16	1861	5.73
16.03a	14	139	13	13	94	23	1755	5.71
17.03a	15	145	14	14	96	22	1847	5.76
Average	17.0	166.5	16.5	16.3	100.0	21.9	2088.8	5.72

Table 6.1: The data from week 9, 10 and 11 with shared visits, 2006

The upper bound for total travelling time T is found by sorting all the distances between all visits, and adding up the $n - m$ highest distances. The lower bound for the number Ψ of unlocked visits without a regular caretaker is found by going through all visits and checking if their regular caretaker is working on that current day. If there is a shared visit it is only counted once if the regular caretaker is not at work that day. The start visits and breaks are not counted in Ψ, because these visits are locked.

The average travelling time is found by finding all the distances between all visits on the day considered and dividing it by the total number of distances. It is very important to notice that the average of the average distance is 5.7, which will be used as an estimate of the mean value.

The table 6.2 displays a modified version of the data set in table 6.1. This dataset gives the opportunity of evaluating how the shared visits affect the insertion heuristic and the tabu search. This data set also has another purpose, because ABP is not able to handle shared visits at the time of writing, and hence the data set without shared visits is used for the comparison with the method in ABP.

Day in year 2006	Number of caretakers	Total number of visits	Number of start visits	Number of break visits	Number of citizens	Lower bound on Ψ	Upper bound on T	Average travelling time
27.02b	20	166	19	16	106	23	2090	5.80
28.02b	19	161	18	14	108	27	1886	5.57
01.03b	19	160	16	19	92	18	1995	5.72
02.03b	19	177	17	17	111	31	2324	5.80
03.03b	20	177	18	18	118	25	2224	5.74
06.03b	18	154	17	15	97	21	1903	5.83
07.03b	18	165	17	15	106	23	1943	5.63
08.03b	18	157	16	19	93	19	1968	5.78
09.03b	19	172	17	18	104	31	2178	5.70
10.03b	19	179	17	18	100	22	2232	5.69
13.03b	19	157	17	16	92	22	1935	5.86
14.03b	18	158	16	14	100	27	1910	5.71
15.03b	18	143	16	18	83	16	1781	5.75
16.03b	14	135	13	13	94	23	1699	5.72
17.03b	15	141	14	14	96	22	11785	5.75
Average	17.0	160.1	16.5	16.3	100.0	21.9	1990.2	5.74

Table 6.2: The data from week 11 without shared visits, 2006

The difference between the datasets with and without the shared visits is found in the number of visits, the upper bound on T and the average travelling time. The lower bounds on Ψ in table 6.1 and 6.2 are the same, because at least one of regular caretakers for all the shared visits were at work in the data instances.

Notice that the days with shared visits are marked with an "a" and the days without shared visits are marked with a "b".

6.4 The Performance of the Insertion Heuristic

The insertion heuristic only has one parameter μ, which indicates how much longer time a planner would let a regular caretaker drive to reach a citizen, where the caretaker is regular. For every visit, where the caretaker is not regular the cost μ is added.

$$\min C(x) = T + \mu\Psi$$

Variating the parameter μ gives different results. It is decided to set μ at three different levels.

$\mu = 0$: It is very interesting to consider what will happen if $\mu = 0$, because it implies that only the travelling time will be minimized. It can be used as a bench mark when evaluating how the travelling time might increase, then the cost μ increases.

$\mu = 5.7$: This is the value of the estimate for the average travelling time between two visits. It can be interpreted as the minimization of the travelling time is just as important as minimizing the number of visits with a regular caretaker. This value is very interesting, because the result of a questionnaire investigation described in the report [Tho05] showed that the citizens found it equally important to have a regular caretaker and that was extra time. The extra time can be gained by minimizing the total travelling time.

$\mu = 11.4$: This value is used to find the results, if one consider the regular caretakers twice as important as the travelling time.

The insertion heuristic is run on a SUN Fire 3800, with a 1200 Mhz Sun SPARC processor, and the run times are in the interval from 0.5 to 2.6 seconds for test instances, where the average is approximately 1.1 seconds.

6.4.1 The Data with Shared Visits

The results when using the insertion heuristic for the problems with shared visit from week 9, 10 and 11 are displayed in table 6.3.

It is found in table 6.3 that the total travelling time is much lower than the upper bound for all values of μ, but it not possible to tell, how good T is on basis of the upper bound, because the upper bound is calculated without paying attention to the structure of the problems.

Day	$\mu = 0$			$\mu = 5.7$			$\mu = 11.4$		
	T	Ψ	$C(x)$	T	Ψ	$C(x)$	T	Ψ	$C(x)$
27.02a	583	115	583	613	72	1023.4	653	66	1405.4
28.02a	563	123	563	619	89	1126.3	635	90	1661.0
01.03a	539	108	539	603	75	1030.5	652	56	1290.4
02.03a	600	133	600	626	82	1093.4	647	70	1445.0
03.03a	608	134	608	652	91	1170.7	702	70	1500.0
06.03a	558	111	558	603	74	1024.8	635	57	1284.8
07.03a	592	121	592	622	71	1026.7	634	60	1318.0
08.03a	580	109	580	581	73	997.1	612	63	1330.2
09.03a	584	123	584	676	76	1109.2	665	62	1371.8
10.03a	656	132	656	653	65	1023.5	704	50	1274.0
13.03a	548	115	548	564	67	945.9	595	46	1119.4
14.03a	563	111	563	560	66	942.2	617	56	1255.4
15.03a	508	99	508	516	59	854.3	547	48	1094.2
16.03a	414	93	414	488	67	869.9	500	52	1092.8
17.03a	474	105	74	485	58	815.6	543	43	1033.2
Average	558.4	108.8	558.4	554.4	72.3	1003.6	622.7	59.2	1298.4

Table 6.3: The initial solutions found by the insertion heuristic for the data instances with shared visits.

When μ increases one observes in table 6.3, that Ψ decreases, while the total travelling time T is increasing in the most cases. This is not surprising since there is a conflict between minimizing the total travel time and minimizing the number of visits without regular caretaker. When the problem is relaxed by setting $\mu = 0$, the optimal solution value of T will be the same or decreased compared to when $\mu > 0$.

It would be interesting to see what would happen for more values of μ, and hence a small example is made. The μ value is increased from 0 to 50 when using the insertion heuristic on the data instance 27.02a. The plot to the left in figure 6.1 shows T as a function of μ and the plot to the right shows Ψ as a function of μ.

This example shows how there is a tendency of increasement for T, while there is a tendency of decreasement for Ψ, when μ is increased, but the functions are not monotonous increasing or decreasing. At $\mu = 1$ there is a reduction in the total travelling time T in the graph to the left in figure 6.1, which may be caused by finding a good total travelling time, while looking for a solution with less visits without a caretaker. The example in figure 6.1 shows, that the two objectives of minimizing T and Ψ are not always conflicting, when using the insertion heuristic, which do not necessarily find

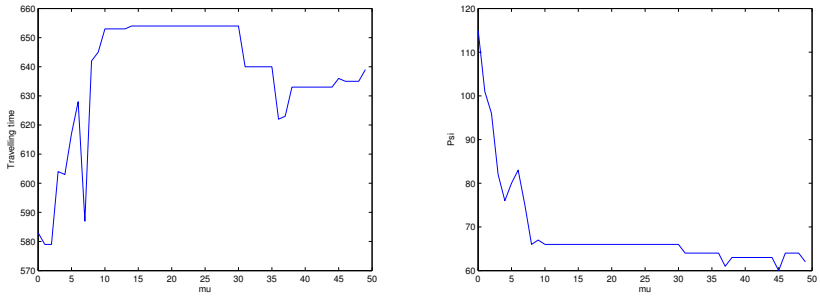


Figure 6.1: How the total travelling time and the number of visits without a regular caretaker evolves when varying the parameter μ, in the dataset 27.02a

the optimal solutions.

6.4.2 The Data without the Shared Visits

The insertion heuristic may perform differently on data without shared visits, and hence the data without shared is also used for testing the insertion heuristic.

In table 6.4 it is observed, that the average cost $C(x)$ is reduced. The reason for this is the reduced total number of visits, because the shared visits are removed. For this reason T and Ψ are also decreased.

The conclusion drawn on this test, is that the shared visits do not seem to have any impact on the results found by using the insertion heuristic.

6.5 The Parameter Tuning

The parameters in the tabu search are tuned to perform better, when testing it on the data from week 11. The data used for the parameter tuning is from week 9 and 10. It is important to notice, that the data set is split in two parts, and the part used for the parameter tuning is not used for the testing.

The tabu search is set to run for 100 iterations in the parameter tuning and in the test.

The parameters interesting for tuning are listed below with the test values. It is interesting to follow, how the changes on these parameters effect the tabu search.

Day	$\mu = 0$			$\mu = 5.7$			$\mu = 11.4$		
	T	Ψ	$C(x)$	T	Ψ	$C(x)$	T	Ψ	$C(x)$
27.02b	526	117	526	584	73	1000.1	597	60	1281.0
28.02b	554	123	554	571	87	1066.9	617	84	1574.6
01.03b	549	108	549	565	69	958.3	593	49	1151.6
02.03b	583	131	583	622	88	1123.6	672	67	1435.8
03.03b	594	124	594	633	79	1083.3	652	77	1529.8
06.03b	533	103	533	559	71	963.7	600	49	1158.6
07.03b	560	116	560	582	72	992.4	594	54	1209.6
08.03b	548	110	548	555	71	959.7	583	53	1187.2
09.03b	574	125	574	596	76	1029.2	583	53	1187.2
10.03b	581	119	581	597	62	950.4	658	42	1136.8
13.03b	543	109	543	554	59	890.3	602	40	1058.0
14.03b	518	115	518	489	66	865.0	539	48	1086.2
15.03b	484	99	484	516	49	795.3	527	33	903.2
16.03b	442	97	442	577	57	801.9	486	51	1067.4
17.03b	471	100	471	501	43	746.1	523	37	944.8
Average	537.3	113.1	537.3	560.1	68.1	948.4	554.1	53.1	1195.5

Table 6.4: The initial solutions found by the insertion heuristic for the data set without shared visits

- δ has the test values $\{0.00, 0.25, 0.50, \ldots, 2.50\}$. It is the factor for managing the values of α, β and γ.
- λ has the test values $\{0, 5, 10, 15, 20\}$. It is the diversification factor.
- θ has the test values $\{0, 5, 10, 15, 20\}$. It is the number of number iterations, where it is forbidden to reinsert a visit i into route $\bar{r}$.

The parameters α, β and δ are not tuned.

- α is the price for violating the time windows, and it is set to the fixed value 1.
- β is the price for violating the equal starting times for shared visits and it is also set to 1.
- γ is the price for violating the working hours, and it is as the other prices also set to 1.

The problem in [CLA01] is very similar to the VRPTWSV, and hence it is reasonable to use the same conditions as in the article. All the prices α,

β and γ are set to 1 in the article, and this procedure is followed in this project.

The tabu search method is parameter tuned for the values 0, 5.7 and 11.4 of μ.

6.5.1 The Data with Shared Visits and $\mu = 0$

The table 6.5 gives an overview over the range of the solutions found by the tabu search. The solutions with the smallest improvement of the initial solution are the worst solutions found in the tabu search. Similarily the solutions with the largest improvement of the initial initial are the best solutions found in the tabu search.

Day	Worst solution found x^o				Best solution foundx^*			
	T	Ψ	Cost	Improvement from the initial solution	T	Ψ	Cost	Improvement from the initial solution
27.02a	576	115	576	1.2 %	468	117	468	19.7 %
28.02a	558	123	558	0.9 %	498	124	498	11.5 %
01.03a	539	108	539	0.0 %	472	109	472	12.4 %
02.03a	598	133	598	0.3 %	533	133	533	11.2 %
03.03a	601	134	601	1.2 %	567	134	567	6.7 %
06.03a	555	110	555	0.5 %	513	109	513	8.1 %
07.03a	558	121	558	5.7 %	507	125	507	14.4%
08.03a	576	109	576	0.7 %	500	109	500	13.8%
09.03a	564	123	564	3.4 %	499	121	499	14.6%
10.03a	649	131	649	1.1 %	573	130	573	12.7%
Average				1.5 %				12.5 %

Table 6.5: The worst and best solution with $\mu = 0$

In one case the tabu search does not find an improvement of the original solution. The largest improvement on 19.7 % is reached for the first day 27th, February. The figures A.1 and A.2 show where the best and worst solution values are found for combinations of λ, θ and δ. The best solutions have the value $C(x^*)$ and the worst solutions have the solution value $C(x^o)$.

The figure A.1(e) is not used for finding the best combination of λ, θ and δ, because the range from the worst solution value to the best is not big enough to see any variation. The remaining figures show how the best set of parameters is very dependent on the problem considered. Only small tendencies can be observed. The figures show how the parameter δ is very

important for the solutions found within one day, because the good solutions tend to be in a horizontal layer in the figures. In most cases the good solutions are situated with $\delta = 1.25$ as a center. There also seem to be a small tendency to get good solution values for higher values of θ, and the good solutions are seldom found at $\theta = 0$, which indicates that the tabu criterion answers its purpose. The interesting issue is that the diversification parameter λ does not have any effect, which may be caused by the fact, that the tabu search only runs for 100 iterations.

6.5.2 The Data with Shared Visits and $\mu = 5.7$

When the parameter μ is increased, it is interesting to see how the quality of the solutions evolve. The best solutions are only improved by 10.1 %. The solutions found for the data set from the 9th March are not used in the evaluation of the best set of parameters, because the best solution found only is 4 % better than the worst and initial solution.

Day	Worst solution found (x^o)				Best solution found (x^*)			
	T	Ψ	Cost	Improvement from the initial solution	T	Ψ	Cost	Improvement from the initial solution
27.02a	610	72	1020.4	0.3 %	536	56	855.2	16.4 %
28.02a	611	89	1118.3	0.7 %	567	78	1011.6	10.2 %
01.03a	603	75	1030.5	0.0 %	573	62	926.4	10.1 %
02.03a	626	82	1093.4	0.0 %	584	68	971.6	11.1 %
03.03a	651	90	1164.0	0.6 %	649	75	1076.5	8.1 %
06.03a	595	94	1016.8	0.8 %	561	67	942.9	8.0 %
07.03a	662	71	1026.7	0.0 %	549	61	896.7	12.7 %
08.03a	577	73	993.1	0.4 %	543	63	902.1	9.5 %
09.03a	676	76	1109.2	0.0 %	615	65	985.5	11.2 %
10.03a	653	65	1023.5	0.0 %	646	59	982.3	4.0%
Average				0.3 %				10.1 %

Table 6.6: The worst and best solution with $\mu = 5.7$

The figures A.3 and A.4 illustrate how the best solutions are situated for each set of the parameters δ, λ and θ. They do not show one single area, where all good solutions are gathered. The good solutions are spread out over the whole domain. The good solutions are both placed above and under $\delta = 1.25$. The parameter θ does not seem to have much influence on the value of the solutions found. It is only possible to conclude that the θ

parameter, should not be set to 0, because the good solutions are seldom at $\theta = 0$.

6.5.3 The Data with Shared Visits and $\mu = 11.4$

The parameter μ is raised to 11.4, and the average improvement of the best solution found is raised to 11.8%. The positions of the good values in different parameter settings can be found in the figures A.5 and A.6. The data from the 6th March is not used in the parameter tuning, because the span between the values of the best solutions and the worst solution is much lower than the average span for the other data sets used. The figures A.5 and A.6 do not give a clear idea of where the best set of parameters is situated. This may indicate, that the best parameter setting is very problem dependent. The best parameter setting may depend on the positions of the shared visits. If they situated in routes, where many visits are inserted or removed, it is more difficult for the tabu search to find a feasible solution, because the push forward and push backward function do not pay attention to the shared visits.

Day	Worst solution found (x^o)				Best solution found (x^*)			
	T	Ψ	Cost	Improvement from the initial solution	T	Ψ	Cost	Improvement from the initial solution
27.02a	647	66	1399.4	0.4 %	635	45	1148.0	18.3 %
28.02a	635	90	1661.0	0.0 %	585	66	1337.4	19.5 %
01.03a	646	56	1284.4	0.4 %	664	43	1154.2	9.7 %
02.03a	647	70	1445.0	0.0 %	634	59	1306.6	9.6 %
03.03a	706	68	1481.2	1.3 %	671	64	1400.6	6.6 %
06.03a	630	57	1278.9	0.5 %	600	56	1238.4	3.6 %
07.03a	634	60	1318.0	0.0 %	612	42	1090.8	17.2 %
08.03a	612	63	1330.2	0.0 %	603	50	1173.0	11.8 %
09.03a	665	62	1371.8	0.0 %	658	44	1159.6	15.5 %
10.03a	704	50	1274.0	0.0 %	662	46	1186.4	6.9 %
Average				0.3 %				11.8 %

Table 6.7: The worst and best solution with $\mu = 11.4$

If the best parameter setting is problem dependent, it is also interesting to see if the same parameter setting for one data instance is the best for all the used values of μ. The parameter tuning for the day 27th February, shows that the best parameter setting is $\delta = 0.75$, θ in the interval 5 to 30 and the parameter λ free. The same tendency is observed for the other days, which

indicates, that the best parameter setting depends more on the data, than the value μ.

6.5.4 The Data without Shared Visits and $\mu = 0$

The structure of the data without shared visits is different, and hence the tabu search may perform differently. The table 6.8 shows how the best solutions found are improved much from the initial solutions.

Day	Worst solution found (x^o)				Best solution found (x^*)			
	T	Ψ	Cost	Improvement from the initial solution	T	Ψ	Cost	Improvement from the initial solution
27.02b	510	116	510	3.0 %	413	116	413	21.5 %
28.02b	544	123	544	1.8 %	436	118	436	21.3 %
01.03b	546	108	546	0.5 %	439	111	439	20.0 %
02.03b	579	131	579	0.7 %	446	133	446	23.5 %
03.03b	582	124	582	2.0 %	526	121	526	11.4 %
06.03b	519	103	519	2.6 %	425	105	425	20.3 %
07.03b	534	116	534	4.6 %	438	117	438	21.8 %
08.03b	530	110	530	3.3 %	430	113	430	21.5 %
09.03b	558	124	558	2.8 %	443	126	443	22.8 %
10.03b	567	119	567	2.4 %	524	116	524	9.8 %
Average				2.4 %				19.4 %

Table 6.8: The worst and best solution with $\mu = 0$

The figures A.7 and A.8 show how good solutions are obtained for almost all investigated parameter settings of δ, λ and θ. The worst solutions found tend to be at $\theta = 0$ and $\delta = 0$, which again shows that the tabu criterion satisfy its purpose. It also shows that variating the costs α and γ is a good idea. The parameter β is not variating, because there are no shared visits.

6.5.5 The Data without Shared Visits and $\mu = 5.7$

The table 6.9 shows how the tabu search performs, when the parameter μ is raised to 5.7. The best average improvement from the initial solution is still just under 20 %.

The figures A.9 and A.10 show that the good solutions are still situated outside $\theta = 0$ and $\delta = 0$. The figure A.10(b) has a group of good solutions for

Day	Worst solution found(x^o)				Best solution found (x^*)			
	T	Ψ	Cost	Improvement from the initial solution	T	Ψ	Cost	Improvement from the initial solution
27.02b	581	72	991.4	0.9 %	488	49	767.3	23.3 %
28.02b	560	87	1055.9	1.0 %	484	65	854.5	19.9 %
01.03b	558	67	939.9	1.9 %	508	48	781.6	18.4 %
02.03b	620	88	1121.6	0.2 %	581	63	940.1	16.3 %
03.03b	618	76	1051.2	3.0 %	549	52	845.4	22.0 %
06.03b	541	64	905.8	6.0 %	484	49	763.3	20.8 %
07.03b	571	72	981.4	1.1 %	485	51	775.7	21.8 %
08.03b	555	71	959.7	0.0 %	524	45	780.5	18.7 %
09.03b	586	74	1007.8	7.7 %	521	47	788.9	27.8 %
10.03b	567	63	926.1	2.6 %	542	46	804.2	15.4 %
Average				2.4 %				18.4 %

Table 6.9: The worst and best solution with $\mu = 5.7$

high values δ. It may be caused by the fact, that it is not possible to violate the constraint on synchronous starting times for shared visits. This implies less possibilities for violations, and each violation will have to penalized more relatively to the solution value $C(x)$ in (4.4), to be eliminated. The figures A.9(d) and A.9(e) do not not give the same result, because bad solutions are situated for high values of δ. For this reason it is not possible to set the δ at a good value, and it is only possible to determine from the figures that a good value of δ is different from 0.

6.5.6 The Data without Shared Visits and $\mu = 11.4$

The table 6.10 contains an overview over the solutions found by using tabu search with the parameter $\mu = 11.4$. The improvements from the initial solutions are better than those observed in table 6.8 and 6.9.

The data set from the 6th of March will not be used for the parameter tuning, because the range between the good and bad solutions is small relatively to the ranges obtained for the other data sets.

The figures A.11 and A.12 have more bad solutions for the settings of the parameters than for $\mu = 0$ and $\mu = 5.7$. Especially the data set from the 7th March has many bad solutions, and the best solutions are situated for $\delta = 2$. This δ value gives bad solutions for the data from the 27th and 28th February, which is conflicting. It can be concluded that the relationship

Day	Worst solution found (x^o)				Best solution found (x^*)			
	T	Ψ	Cost	Improvement from the initial solution	T	Ψ	Cost	Improvement from the initial solution
27.02a	585	59	1257.6	1.8 %	575	31	928.4	27.5 %
28.02a	609	67	1372.8	12.8 %	568	50	1138.0	27.7 %
01.03a	593	49	1151.6	0.0 %	606	28	925.2	19.7 %
02.03a	672	67	1435.8	0.0 %	602	42	1080.8	24.7 %
03.03a	652	77	1529.8	0.0 %	616	47	1151.8	24.7 %
06.03a	600	49	1158.6	0.0 %	586	29	916.0	20.9 %
07.03a	584	53	1188.2	1.8 %	551	35	950.0	21.5 %
08.03a	578	47	1113.8	6.2 %	549	27	856.8	27.8 %
09.03a	602	53	1206.2	0.0 %	584	40	1040.0	13.0 %
10.03a	655	42	1133.8	0.3 %	642	37	1063.8	6.4 %
Average				2.3 %				21.4 %

Table 6.10: The worst and best solution with $\mu = 11.4$

between the structures of the problems and the good parameter setting is a subject of future investigation, because the results are not convincing when only variating μ.

6.6 The Performance of the Tabu Search

The parameter setting is chosen to cover all instances of μ and both data sets with or without shared visits. Based on the observations on the data with shared visits it is chosen to set the parameter δ in the middle of the interval, because both good solutions are found above and below $\delta = 1.25$. The value of λ seems not to matter much for the quality of the solutions, and is for this reason also set in the middle $\lambda = 10$. The last parameter θ is set as high as possible ($\theta = 30$), because there is a tendency of finding good solutions for higher values of θ.

The data from week 11 is used for testing the performance of the tabu search, and the tabu search is run for 100 iterations. The test is also performed on a SUN Fire 3800 computer, with a 1200 Mhz Sun SPARC processor, and the run times are in the interval from 13 to 31 seconds for test instances, where the average is approximately 19 seconds.

Firstly the data set with shared visits and $\mu = 0$ is tested. The total travelling time T and number Ψ of unlocked visits without a regular caretaker in

Day	T	Ψ	Cost	Improvement from the initial solution
13.03a	530	114	530	3.2 %
14.03a	503	111	503	10.7 %
15.03a	483	97	483	4.9 %
16.03a	406	93	406	1.9 %
17.03a	458	105	458	3.4 %
Average	476.0	104.0	476.0	4.8%

Table 6.11: The solution found with $\mu = 0$

the solutions found with the chosen parameter setting are displayed in table 6.11. The average improvement is 4.8 %, which a relative low improvement.

When the value μ is increased to 5.7, the average improvement is also increased to 6.0 %, which is indicated in table 6.12.

Day	T	Ψ	Cost	Improvement from the initial solution
13.03a	558	65	928.5	1.8 %
14.03a	530	58	860.6	8.7 %
15.03a	516	55	829.5	2.9 %
16.03a	439	64	803.8	7.6 %
17.03a	478	46	740.2	9.2 %
Average	504.2	57.6	832.5	6.0 %

Table 6.12: The solution found with $\mu = 5.7$

The average improvement is increased to 10.2 %, when the value μ is increased to 11.4, see the table 6.13. There might be a relationship between the average improvement and the value μ, because it is easier to find better solutions, if the cost of having a visit in the "wrong" route with a not regular caretaker is very high. Moving this visit to a route with a regular caretaker implies a large decrease in the objective function value.

The tabu search is also tested on the data set without the shared visits. The table 6.14 contains the results, and it shows that the average improvement is high for $\mu = 0$. Comparing this with table 6.11, where the average improvement was 4.8 %, this result with an average improvement on 17.6 % indicates how the tabu search is better designed for problems without

Day	T	Ψ	Cost	Improvement from the initial solution
13.03a	577	29	907.6	18.9 %
14.03a	615	50	1185.0	5.6 %
15.03a	576	34	963.6	14.4 %
16.03a	503	43	993.2	9.1 %
17.03a	534	41	1001.4	3.1 %
Average	561.0	39.4	1010.6	10.2 %

Table 6.13: The solution found with $\mu = 11.4$

shared visits, because the push forward and backward functions are more appropriate, when it is only possible to violate the latest starting times for visits and the latest finishing times for caretakers.

Day	T	Ψ	Cost	Improvement from the initial solution
13.03b	439	111	439	19.2 %
14.03b	404	114	404	22.0%
15.03b	395	94	395	18.4 %
16.03b	388	99	388	12.2 %
17.03b	395	99	395	16.2%
Average	404.2	103.4	404.2	17.6 %

Table 6.14: The solution found with $\mu = 0$

The table 6.15 contains the results from the tests, when μ is increased to 5.7. The results show that the average improvement is increased from 17.6 % to 22.2 %.

The last test performed on the tabu search is with μ increased to 11.4, and the results are displayed in table 6.16. These results show that the average improvement is not increased from $\mu = 5.7$, which may be caused by good initial solutions.

The best improvement reached for the data with shared visits is 19 % for the instance 13.03a and $\mu = 11.4$, and the best improvement reached for the data without the shared visits is 27 % for the instance 13.03a and $\mu = 11.4$. These results show, how the push functions do not pay attention to the shared visits.

Day	T	Ψ	Cost	Improvement from the initial solution
13.03b	487	44	737.8	17.1 %
14.03b	468	49	747.3	24.1%
15.03b	480	33	668.1	16.0 %
16.03b	385	39	607.3	24.3 %
17.03b	444	35	643.5	14.8 %
Average	452.8	40	680.8	22.2 %

Table 6.15: The solution found with $\mu = 5.7$

Day	T	Ψ	Cost	Improvement from the initial solution
13.03b	538	26	834.4	26.8 %
14.03b	518	31	871.4	19.8 %
15.03b	515	29	845.6	6.4 %
16.03b	448	37	869.8	18.5 %
17.03b	483	31	836.4	11.5 %
Average	412.4	30.8	851.5	16.6 %

Table 6.16: The solution found with $\mu = 11.4$

The figure 6.6 shows how the total violations vary. The instance is from the 13th, March, with shared visits and the chosen parameter setting.

It is observed in figure 6.6 how the violations are changing through the iterations. It is important to notice, that in some of the iterations, all the violations are zero, and there the tabu search has found a feasible solution.

It has to be emphasized, that performing the tabu search is slower than finding the initial solution with the insertion heuristic, and hence the tabu search can only be applied, if the planner of the routes has extra time or patience to wait for better solutions.

6.7 Comparison with the Solutions From the ABP

The solutions in the tables 6.3-6.4 and 6.11 -6.16 found by the methods in this project are compared with the solutions found by the ABP programme

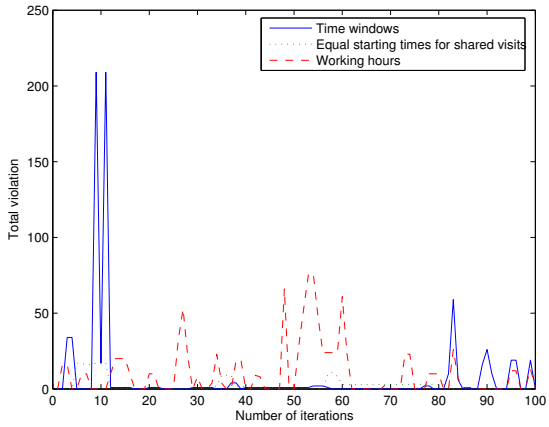


Figure 6.2: The solid line shows the violation on time windows, the dotted line shows the violation on the synchronous starting times for shared visits and the dashed line shows the violation on the working hours

for the test data in week 11. When comparing the results it has to be taken into consideration, that the ABP programme is very restricted on the runtime. The run time of the programme is approximately 100 milliseconds, and additional time is used for reading from the data base and plotting the solution graphically. Another issue to take into consideration is that the ABP programme tries to maximize the spare time after the last visit in each route, with the purpose to let the caretaker go home, when he has finished his last visit.

In ABP it is possible to set 3 parameters for weighting the importance of three different issues:

1. The total travelling time
2. The regular caretakers
3. The qualications of the caretaker compared with the demands of the visits.

The weights have to sum up to 100, and the last parameter is always set to 0 in these tests. The three different settings of the parameters used are: (100,0,0), (50,50,0) and (33,67,0). The parameter μ used in VRPTWSV and the parameters in ABP are only comparable, in the situation when $\mu = 0$ and (100,0,0). Hence the results from each setting of the parameters in ABP is compared with $\mu = 0$, 5.7 and 11.4 in the VRPTWSV. The comparison is

performed by calculating the deviations from the T and Ψ in the solutions found by ABP.

In table 6.17 a comparison between ABP and the insertion heuristic is performed. The most interesting comparison is made with the performance of the insertion heuristic and $\mu = 0$, because both methods try only to minimize the total travelling time. The insertion heuritic finds solutions, that are 25% better on average. The table also contains the results of the comparison, when $\mu = 7.4$ and $\mu = 11.4$, because it can be observed that the total travelling time is still better in the solutions found by the insertion heuristic, when it also tries to minimize Ψ.

	ABP		Insertion $\mu = 0$		Insertion $\mu = 5.7$		Insertion $\mu = 11.4$	
Day	T	Ψ	T	Ψ	T	Ψ	T	Ψ
13.03b	734	108	- 26.0 %	+ 0.9 %	- 24.5 %	- 45.4 %	- 18.0 %	- 63.0 %
14.03b	743	112	- 30.3 %	+ 2.7%	- 33.0 %	- 41.1 %	- 27.5 %	- 57.1 %
15.03b	620	100	- 21.9 %	- 1.0 %	- 16.8 %	- 51.0 %	- 15.0 %	- 67.0 %
16.03b	589	100	- 25.0 %	- 3.0 %	- 19.0 %	- 43.0 %	- 17.5 %	- 49.0 %
17.03b	601	141	- 21.6 %	- 29.1 %	- 16.6%	- 69.5 %	- 13.0 %	- 73.8 %
Average	647.4	112.2	- 25.0 %	- 5.9 %	- 22.0 %	- 50.0 %	- 18.2 %	- 62.0 %

Table 6.17: The ABP solutions with the weights (100,0,0)

	ABP		Tabu Search $\mu = 0$		Tabu Search $\mu = 5.7$		Tabu Search $\mu = 11.4$	
Day	T	Ψ	T	Ψ	T	Ψ	T	Ψ
13.03b	734	108	- 40.2 %	+ 2.8 %	- 33.7 %	- 59.3 %	- 26.7 %	- 75.9 %
14.03b	743	112	- 45.6 %	+ 1.8 %	- 37.0 %	- 56.3 %	- 30.3 %	- 72.3 %
15.03b	620	100	- 36.3 %	- 6.0 %	- 22.6 %	- 67.0 %	- 16.9 %	- 71.0 %
16.03b	589	100	- 34.1 %	- 1.0 %	- 34.6 %	- 61.0 %	- 23.9 %	- 63.0 %
17.03b	601	141	- 34.3 %	- 29.8 %	- 26.1 %	- 75.2 %	- 19.6 %	- 78.0 %
Average	657.4	122.2	- 38.1 %	- 6.4 %	- 28.8 %	- 63.8 %	- 23.5 %	- 72.0 %

Table 6.18: The ABP solution with the weights (100,0,0)

The parameters in ABP are set to 50 for the first parameter and also 50 for the second parameter. The solutions are described in table 6.19 and 6.20. It is observed, that the number Ψ of unlocked visits without regular caretakers is decreased, but the insertion heuristic with $\mu = 5.7$ performs better, because Ψ is decreased by 37.2 % on average and T is decreased by 34.1 % on average.

This comparison is not fair, because setting the parameters to (50,50,0) does not mean, that the regular caretakers are as important as the travelling time.

On purpose the travelling time is set to be weighted higher than the regular caretakers. This adjustment of the parameters is performed, because ABP has its focus on the ATA time. The abbreviation is short for "face to face" time (in Danish "Ansigt Til Ansigt tid"), and hence the programme mainly focuses on minimizing the total travelling time. The users of the programme are thereby motivated to assign the regular caretakers in a manner, which minimizes the total travelling time e.g. a caretaker is only set to be regular for citizens situated with small distances between them. It is possible to change the adjustment of the parameters, but it is not performed in these tests, because the tests should show how the programme performs with the adjustments chosen to use in practice.

	ABP		Insertion $\mu = 0$		Insertion $\mu = 5.7$		Insertion $\mu = 11.4$	
Day	T	Ψ	T	Ψ	T	Ψ	T	Ψ
13.03b	835	99	- 35.0 %	+ 10.1 %	- 33.7 %	- 40.4 %	- 27.9 %	- 59.6 %
14.03b	782	100	- 33.8 %	+ 15.0 %	- 37.5 %	- 41.0 %	- 31.1 %	- 52.0 %
15.03b	805	81	- 39.9 %	+ 22.2 %	- 35.9 %	- 35.9 %	- 34.5 %	- 59.3 %
16.03b	690	85	- 35.9 %	+ 14.1 %	- 30.9 %	- 32.9 %	- 29.6 %	- 40.0 %
17.03b	742	84	- 36.5 %	+ 19.0 %	- 32.5 %	- 32.1 %	- 29.5 %	- 56.0 %
Average	770.8	89.8	- 36.2 %	+ 16.1 %	- 34.1 %	- 37.2 %	- 30.5 %	- 53.4 %

Table 6.19: The ABP solutions with the weights (50,50,0)

	ABP		Tabu Search $\mu = 0$		Tabu Search $\mu = 5.7$		Tabu Search $\mu = 11.4$	
Day	T	Ψ	T	Ψ	T	Ψ	T	Ψ
13.03b	835	99	- 47.4 %	+ 12.1 %	- 41.7 %	- 55.6 %	- 35.6 %	- 73.7 %
14.03b	782	100	- 48.3 %	+ 14.0 %	- 40.2 %	- 51.0 %	- 33.8 %	- 69.0 %
15.03b	805	81	- 50.9 %	+ 16.0 %	- 40.4 %	- 59.3 %	- 36.0 %	- 64.2 %
16.03b	690	85	- 43.8 %	+ 16.5 %	- 44.2 %	- 54.1 %	.35.1 %	- 56.5 %
17.03b	742	84	- 46.8 %	+ 17.9 %	- 40.2 %	- 58.3 %	34.9 %	- 63.1 %
Average	770.8	89.8	- 47.4 %	+ 15.3 %	- 41.3 %	- 55.7 %	- 35.1 %	- 65.0 %

Table 6.20: The ABP solutions with the weights (50,50,0)

When the parameters are changed from (50,50,0) til (33,67,0) a small effect is seen on average of Ψ, which decreases from 89.8 to 88.2. The reason for this is the focus on the ATA time. The insertion heuristic with $\mu = 5.7$ performs better than the ABP with (33,67,0), because the T is decreased by 34% on average and Ψ is decreased by 38 % on average.

The comparison with the tabu search illustrates how much the solution can be improved by allowing a longer computation time. In table 6.18 a average

	ABP		Insertion $\mu = 0$		Insertion $\mu = 5.7$		Insertion $\mu = 11.4$	
Day	T	Ψ	T	Ψ	T	Ψ	T	Ψ
13.03b	845	100	- 35.7 %	+ 9.0 %	- 34.4 %	- 41.0 %	- 28.8 %	- 60.0 %
14.03b	807	101	- 35.8 %	+ 13.9 %	- 39.4 %	- 34.7 %	- 33.2 %	- 52.5 %
15.03b	789	76	- 38.7 %	+ 30.3 %	- 34.6 %	- 35.5 %	- 33.2 %	- 56.6 %
16.03b	667	87	- 33.7 %	+ 11.5 %	- 28.5 %	- 34.5 %	- 27.1 %	- 41.4 %
17.03b	747	77	- 36.9 %	+ 29.9 %	- 32.9 %	- 44.2 %	- 30.1 %	- 51.9 %
Average	771.0	88.2	- 36.2 %	+ 18.9 %	- 34.0 %	- 38.0 %	- 30.5 %	- 52.5 %

Table 6.21: The ABP solutions with the weights (33,67,0)

improvement on 38 % of T is obtained, when $\mu = 0$. In the tables 6.20 and 6.22 the improvement are large also for $\mu = 7.4$ and $\mu = 11.4$. The number Ψ is reduced by more than 50 % and T is reduced by more than 40 % in table 6.22 for $\mu = 5.7$.

	ABP		Tabu Search $\mu = 0$		Tabu Search $\mu = 5.7$		Tabu Search $\mu = 11.4$	
Day	T	Ψ	T	Ψ	T	Ψ	T	Ψ
13.03b	845	100	- 48.0 %	+ 11.0 %	- 42.4 %	- 56.0 %	- 36.3 %	- 73.0 %
14.03b	807	100	- 49.9 %	+ 14.0 %	- 42.0 %	- 51.0 %	- 35.8 %	- 69 %
15.03b	789	76	- 49.9 %	+ 23.7 %	- 39.2 %	- 56.6 %	- 34.7 %	- 61.8 %
16.03b	667	87	- 41.8 %	+ 13.8 %	- 42.3 %	- 55.2 %	- 32.8 %	- 57.5 %
17.03b	747	77	- 47.1 %	+ 28.6 %	- 40.6 %	- 54.5 %	- 35.3 %	- 59.7 %
Average	771.0	88.2	- 47.3 %	+ 18.2 %	- 41.3 %	- 54.7 %	- 35.0 %	- 64.2 %

Table 6.22: The ABP solutions with the weights (33,67,0)

The performance of the insertion heuristic is better than the ABP programme in all cases. There are many reasons to this. One reason is the restricted computation time provided for ABP, and another reason is that the ABP is designed to take more constraints into consideration. The insertion heuristic in this project is more advanced, because it uses a regret measure.

These tests show, that when simplifying problem, solutions of higher quality can be reached. The next question is whether the insertion heuristic described in this project can be applied in the ABP programme. It would demand an improvement of the running time, especially when the heuristic has to take more constraints into consideration.

The method in ABP is used in various ways, where planning one day is a less demanding task on the number of computations. The task of planning

all the visits of new citizen is a very demanding task on the number of computations, because all the new visits have to inserted in each week in period of 8 weeks, where some visits have several days as an option for insertion. If the running time of the insertion heuristic can not improved sufficiently for being applied for planning all the visits for a new citizen, it might be applicable for scheduling the visits one day with the necessary extensions on more constraints made. The issue on a low running time is very important for the ABP programme to be well received in the offices in the different home care districts.

The solutions used before the ABP was developed were in many cases infeasible, because many visits on a route were overlapping, and the caretaker was scheduled to do more than one thing at the time. The solutions found with the ABP or the insertion heuristic in this project can not be compared with infeasible solutions.

Chapter 7

Discussion

This is an discussion on how methods from operations research can be applied in the home care sector, because good solutions do not necessarily imply applications of the methods.

The administration staff in the home care sector normally have no knowledge on operational research, and this may turn out to be a barrier, if one wants to apply the methods from operational research in the home care sector. They will not have the ability to validate if a solution is good. Their validations are based on a comparison with the usual applied solutions, which might not be very good, and hence a solution of poor quality will be chosen.

Some of the terms in operational research are new to the administration staff. For instance a time window is a new conception to many people, and hence they will not take advantage of it, because they do not know it. Instead they lock the visits to be performed at a given time, which implies solutions of lower quality. In the ABP system it is possible to choose to lock the time for a visit. This function is superfluous, because the time can be locked, by setting the time window as tight as the duration of the visit.

The whole concept of computing may also be new to many in an administration staff, because they do not know or do not care about that providing a longer computation time often gives better solutions.

It needs to be considered how the staff can gain useful information on how to validate a solution or set time windows. One way is by arranging seminars on the topic.

The wishes of the caretakers, citizens and employers are often many and conflicting. For instance when citizens want to have same caretaker always, and the caretakers want to change the citizens sometimes.

The danger of making it possible to fulfill too many wishes is that the administration staff is not able to handle them, because considering how to set the parameters and setting them is a demanding task.

ABP weights the importance of the three regular caretakers, where the first one is the most important. I would suggest when applying ABP, that the first caretaker is also responsible for following the situation at the citizen. Normally the situation at each citizen is written down in a journal, but by making one caretaker responsible for checking the situation, the caretaker gets more responsibility and the situation is better observed.

The topic on how to apply the methods from operational research is very complex, because of different wishes from the tax payers, the citizens and the caretakers. This project has mainly focused on the citizens by including the regular caretakers, which showed to give good results.

Chapter 8

Further Investigation

Future work could be to develop alternative methods for solving the VRPTWSV introduced in chapter 2. The *column generation* is proposed as an alternative method in section 8.1. The VRPTWSV is just one kind of a problem in the home care sector, and section 8.2 presents other problems.

8.1 Other Methods: Column Generation

The column generation is a method, which splits up a problem in two parts: a *subproblem* and a *master problem*. The solutions found in the subproblem are used as parameters in the master problem and the dual variables found in the master problem are used as parameters in the subproblem. The figure 8.1 illustrates this interaction between the master- and subproblem. The next subsections will go into deeper detail on how the master- and subproblem can be formulated in the VRPTWSV. Three different approaches are introduced.

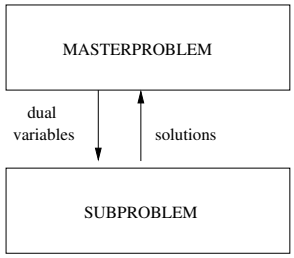


Figure 8.1: Column Generation

8.1.1 The Master Problem

The master problem can be formulated as a mathematical model with *legal routes*. A legal route r performed by caretaker o is a route that starts in D_o^- and ends in D_o^+, where D_o^- and D_o^+ could be the home of the caretaker. The visits in a legal route are sequenced and they start within their time windows. They are also scheduled to be attended by the caretaker o within the working hours of the caretaker o, and if any visits are locked to a specific caretaker, it is also satisfied.

The binary decision variable in the master problem is given by

$$y_r^o = \begin{cases} 1 & \text{if caretaker } o \text{ performs route } r \\ 0 & \text{otherwise.} \end{cases}$$

The parameter κ_r^i is given by

$$\kappa_r^i = \begin{cases} 1 & \text{if visit } i \text{ is in route } j \\ 0 & \text{otherwise} \end{cases}$$

The starting time for a visit i in route r is the parameter s_r^i.

The objective of the master problem is to choose the best set of legal routes without violating a number of constraints. The price for caretaker o doing route r is p_r^o, which is given in the objective function (2.16). The price p_r^o is the total travelling time on the route r plus a penalty μ if the caretaker o is not regular at some of the visits in the route. The penalty for a shared visit without any regular caretaker is μ in (2.16), but in this master problem the penalty is set to μ for each non regular caretaker to avoid having a price depending on the combination of routes and thereby objective function is simplified.

The mathematical model for the master problem is

$$\begin{aligned}
\min \quad & \textstyle\sum_{r,o} p_r^o y_r^o & & & & (8.1)\\
\text{st} \quad & \textstyle\sum_{r,o} y_r^o \kappa_r^i & \geq 1 \quad & \forall\, i \in \mathcal{V} \qquad (\pi_i) & (8.2)\\
& \textstyle\sum_{r} y_r^o & \leq 1 \quad & \forall\, o \in \mathcal{O} \qquad (\mu_o) & (8.3)\\
& \omega_{pq} \textstyle\sum_{o\in\mathcal{O}} (y_{r_1}^o \kappa_{r_1}^p s_r^p - y_{r_2}^o \kappa_{r_2}^q s_r^q) & \leq 0 \quad & \forall\, p,q \in \mathcal{V}, & (8.4)\\
& & & \forall\, r_1, r_2 \in \hat{\mathcal{R}} \quad (\eta_{ijr_1r_2}) &
\end{aligned}$$

The constraint (8.2) ensures that all visits are attended and constraint (8.3) ensures that each caretaker o does not perform more than one route. The constraint (8.4) ensures, that both parts p and q of a shared visit start at the same time. Notice that if visit p is in route r_1 and this route is actually performed by any caretaker then $\sum_{o\in\mathcal{O}} y^o_{r_1}\kappa^p_{r_1} = 1$.

The dual variables π_i, μ_o or $\eta_{ijr_1r_2}$ measure how much the objective function value of a solution would marginally change, if the right hand side of (8.2), (8.3) or (8.4) is marginally changed.

Each column in the master problem correspond to one legal route. The set $\mathcal{R}$ of legal routes is very huge, and it would be preferable only to have a subset of legal routes $\hat{\mathcal{R}}$, when solving the master problem.

8.1.2 An Initial Solution to the Master problem

It is necessary to have an initial feasible solution to the master problem to be able to calculate the dual variables. Some of the initial routes in $\hat{\mathcal{R}}$ routes are already given, because some of the visits are locked to caretakers. All the visits locked to the same caretaker are put into the same route.

All the remaining unlocked visits are inserted in *dummy routes*, $\tilde{\mathcal{R}}$ in the following way: let every route $\tilde{r}$ consist of one visit. In this way route $\tilde{r}_1$ consists of visit v_1 and route $\tilde{r}_2$ consists of visit v_2 etc. for all the remaining visits. The decision variables will be

$$\kappa^{v_1}_{\tilde{r}_1} = \kappa^{v_2}_{\tilde{r}_2} = \cdots = 1$$

The starting time for every visit in all the routes $\hat{\mathcal{R}}$ is set to the opening time of the time window.

$$s^i_r = a_i \quad \forall\, i \in \mathcal{V},\ \ \forall\, r \in \hat{\mathcal{R}} \quad \text{if } \kappa^i_r = 1$$

The constraint (8.4) in the master problem will not be violated, when setting the starting times as above, because the both parts p and q in a shared visit have equal time windows.

It will be necessary to include a set of *dummy caretakers*, $\tilde{\mathcal{O}}$ in the set $\mathcal{O}$ to ensure the initial solution to be feasible. The total number of caretakers m in $\mathcal{O}$ has to be equal to the number of routes in $\hat{\mathcal{R}}$ initially, because each caretaker can only perform one route and all visits have to be attended according to (8.3) and (8.2). The cost $p^{\tilde{o}}_r$ for assigning a dummy caretaker $\tilde{o}$

to any route is set very high, and equally $p^o_{\tilde{r}}$ for assigning any caretaker to a dummy route $\tilde{r}$ is also set very high. This implies, that the dual variables $\mu_{\tilde{o}}$ and π_i initially will be very high, when the dummy caretakers and the dummy routes are in the initial solution.

When performing the column generation with shared visits, there is a dependency between the columns. One has to consider what happens every time a new column is generated in the subproblem.

Problem with shared visits: If the route contains a part p of shared visits with the starting time s^p_r then there should be a route r_2 in the subset $\hat{\mathcal{R}}$, where the other part q of the shared visit starts at the same time. Otherwise no feasible solution can be found for the master problem. The paradox is that parameter η_{pqrr_2} in the subproblem rewards a starting time s^p_r different from $s^q_{r_2}$, see (8.5) .

The initial starting time of the shared visit will never change, because the subproblem only finds one new route and it can not violate the constraint (8.4) on synchronous starting times for each shared visit. This may prevent the column generation from finding the optimal solution, because it will not be able to search in the whole solution space.

To avoid this problem, one can initially add a new kind of dummy routes to $\hat{\mathcal{R}}$ in the master problem for every possible starting time for all parts of the shared visits. If there are $b_p - a_p$ possible starting times for the part p in a shared visit, then $b_p - a_p$ routes only containing visit p is added, where p starts at different times are added. The costs $p^o_{\tilde{r}}$ of these routes are set very high, which implies that the dual variable π_p also will be high, if these dummy routes are in the solution.

8.1.3 The Subproblem

The subproblem generates the columns for the master problem by finding legal routes. The best route is defined as the one with most negative *reduced cost* according to Dantzig's rule.

The reduced cost is calculated using the dual variables in the master problem, and the reduced cost for letting caretaker o do route r is

$$\hat{p}^r_o = p^r_o - \sum_{i \in \mathcal{V}} \kappa^i_r \pi_i - \mu_o - \omega_{ij} \sum_{r_2 \in \hat{\mathcal{R}}} \sum_{i,q \in \mathcal{V}} (\kappa^i_r s^i_r - \kappa^q_{r_1} s^q_{r_2}) \eta_{iqrr_2}, \tag{8.5}$$

where π_i is the cost of every visit i included in route r and η_{iqrr_2} is the cost of making the starting times between the parts p and q in a shared visit different, where q is situated in a route r_2 among the routes in $\hat{\mathcal{R}}$. The negative reduced cost for a variable y_o^r, which is not in the solution indicates how much p_r^o can be decreased before the optimal solution would change and y_o^r is > 0 in an optimal solution, see page 96 in [Mar04].

The mathematical formulation for the subproblem contains the constraints (2.18) - (2.30) except (2.22), which are

- Each caretaker only starts once in his start depot.
- Each caretaker also finishes once in his end depot.
- When a caretaker arrives to a visit, different from the end depot, the caretaker should also leave the visit.
- Each caretaker has an earliest starting time.
- Each caretaker also has a latest finishing time.
- A visit is only allowed to start when the previous visit is finished and the caretaker has travelled from the previous visit to the current visit.
- Each visit has an earliest starting time
- Each visit has a latest starting time
- The visits locked to a caretaker are performed by the caretaker, they are locked to.

Finding the best legal route is a $\mathcal{NP}$-hard problem, because it is a modified version of the *shortest path problem with time windows* - a problem which is already $\mathcal{NP}$-hard. The subproblem does also try to minimize the travelling time, because p_r^o describes the travelling time and is in the objective function, but it also has other objectives. The modification from the shortest path problem with time windows is also found in the constraints on locked visits, working times and shared visits.

The parameters κ_r^i and s_r^i are the decision variables in this subproblem, and when a solution is found, κ_r^i and s_r^i are given to the master problem.

The interaction between the master- and subproblem as illustrated in figure 8.1 continues until, no more routes with negative reduced costs can be found, and thereby the optimal solution is found.

8.1.4 A Second Way to Handle the Shared Visits in Column Generation

The constaint (8.4) in the master problem makes the problem harder to solve, and therefore a new idea would be to the relax master problem, where (8.4) is removed. The initial solution is again found in the way proposed in subsection 8.1.2.

A procedure for forcing the solution in the master problem to be integral is needed. The column generation is initially run without paying attention to the starting times of the shared visits. When no more routes with negative reduced costs can be found, it is investigated if the solution is feasible. It is feasible if all pairs of parts p and q in shared visits start at the same time without using any of the dummy routes, where p or q starts at all possible times. To obtain a feasible solution it may be necessary to make pushes forward and backward in the routes in the solution. The pushes can of cause only be done in a manner where none of the time windows or working hours are violated. If it is not possible to obtain a feasible solution, because the starting times for a pair p and q can not be set equal, the method forces the starting times to be equal, by adding the constraint

$$\omega_{pq} \sum_{o \in \mathcal{O}} (y^o_{r_1} \kappa^p_{r_1} s^p_r - y^o_{r_2} \kappa^q_{r_2} s^q_r) \leq 0 \qquad \forall\, r_1, r_2 \in \hat{\mathcal{R}} \qquad (\eta_{ijr_1r_2}) \qquad (8.6)$$

to the master problem. The columns generated previously are not removed because they may be a part of an optimal solution. Actually it could be that the optimal set of routes are among the previously generated routes, but they were not chosen in the first round, when it was possible to violate (8.6). If the optimal set of routes is not among the previously generated routes, the column generation starts again with the interaction between the master- and subproblem with the new constraint in the master problem. Because of the high costs of the dummy columns with the parts p of shared visits starting at all possible times, the dual variables π_p are very high, if the dummy columns are in the solution. This implied that the subproblem will find new columns replacing the dummy columns. The whole set of replacement columns for p and q will contain pairs of routes where the newly added constraint is satisfies. When no more routes with negative reduced cost are found it is again investigated whether the solution found is feasible as it was done in the previous round. The method continues until an optimal solution is found.

8.1.5 A Third Way to Handle Shared Visits in Column Generation

A third way to handle the shared visits is by defining a new master- and subproblem. Each column in the masterproblem should correspond to a pair of routes. The set of paired routes in the master problem is $\hat{\mathcal{R}}^2$. This change makes it possible to have a pair of routes r_1 and r_2 with a shared visit, where one part p is in r_1 and the other part q is in r_2. There should though still be payed attention to the other shared visits, because route r_1 could for instance contain another shared visit where the part l is in a third route r_3. The figure 8.2 illustrates such situation. Then one has to ensure that $\hat{\mathcal{R}}^2$ has a pair of routes where l starts at the same time as k. This is ensured by making the initial solution as proposed in subsection (8.1.2), where the shared visits in pairs of routes start at all possible times.

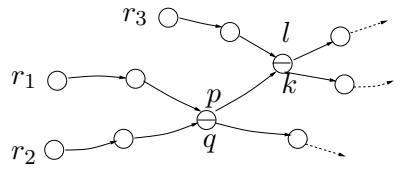


Figure 8.2: The three routes r_1, r_2 and r_3

These thoughts on how to use column generation for finding a solution to the VRPTWSV show, that it is very complex, because of the dependence between the columns.

8.2 Other Problems

In the home care sector, there are many other problems, which can be solved using methods from operational research. Some of those problems are presented in this section.

8.2.1 Extending the VRPTWSV

The VRPTWSV can be extended to include the types of demands for each visit and the qualifications of the caretakers. Another extension could be to include priorities on visits. The extensions can be performed in a similar way like the regular caretakers are included in the problem.

8.2.2 Disruption

The daily work in the home care sector is very dynamic. There may happen all kinds of events both before and after starting attending the visits.

The kinds of events, that may happen before attending the first visits, can be taken into account when scheduling the daily plan. Such event could be a caretaker calling to say he is ill and can not go to work, or a message saying that a citizen is coming home from the hospital.

All kinds of other events can also happen during the execution of daily plan.

Typical events are visits which last longer than expected. The durations of the visits used in the scheduling phase in the morning, are therefore not valid any longer. The duration of a visit increases for instance if somebody forgot to close a catheter well, and there is urine all over the floor, which have to be washed.

Another kind of event is if a citizen suddenly needs more visits, if the situation of the citizen gets worse and he needs more help.

An interruption model would take the events into consideration and make a new plan for the remaining part of the daily plan with the objective to make as few changes as possible.

8.2.3 Rosters

Another problem is to schedule the *rosters* for the caretakers. The scheduling of the *rosters* should both depend on the wishes of the caretakers and the demands of the citizens. For instance a caretaker should have his weekly day-off on Wednesdays, if none of his regular citizens need help on Wednesdays. Another example is a caretaker, who wishes to finish early on Mondays. A good roster would set her finishing time at 1 pm. if that is feasible. A cost of giving in to wishes could be included in the model. As it is today in Søllerød municipality, the rosters for the caretakers are given by KMD (former Kommune Data).

Chapter 9

Conclusion

In this thesis the problem of scheduling the routes in the home care is considered. The problem is limited to only take some of the conditions into consideration. The conditions included are the time windows of the visits, the working hours of the caretakers, the visits locked to careatakers, and the shared visits. The special feature in the problem is the shared visit, which is a visit where two caretakers start and finish the visit at the same time. These shared visits imply a dependence between the routes. The problem has two objectives. One objective is to minimize the total travelling time and the other objective is to minimize the number of visits without a regular caretaker.

The aim of this thesis is fulfilled by developing methods for solving the problem. An insertion heuristic is extended to handle the shared visits in an intelligent way. To determine the best insertion positions, a regret measure is used.

A tabu search heuristic is also changed to be applied to the problem. The tabu search allows infeasible solutions, where the violations are penalized by costs, which are not constant. The violations are allowed for time windows of the visits, the working hours of the caretakers and the starting times of the shared visits.

The move used in the tabu search is the relocation of a visit from one route to another. Neighbourhoods of smaller size have been tried, but without good results, because the violations were increasing too much without the search reached a feasible solution. The implemented tabu search uses the whole neighbourhood.

It is considered how one could use different strategies for finding the new starting times times in the route where a visit is removed or in the route,

where a visit is inserted. The strategy applied in this thesis includes push forward and backward functions. This strategy is chosen because of a short run time and good results, when there are few shared visits as there are in the data instances. The disadvantage of this strategy is that the push functions do not pay attention to the shared visits, and another strategy including a LP-model is suggested.

The developed solution methods are tested, and it is found, that the performance of the insertion heurisic does not depend on the shared visits in the test instances. The running time of the insertion heuristic in approximately 1.1 seconds. An example showed that the minimizing the total travel time and the number of visits without a regular caretaker is not always conflicting, when using the insertion heuristic.

When tuning the parameters for the tabu search, the best settings of the parameters turned out to depend on the problem structure, and hence more investigation can be performed in this area.

The tabu search performed best on the data instances without shared visits, where an improvement up to 27 % is reached. For the data with shared visits, the best improvement reached is 19 %. The reason for this is, that the push forward and backward functions do not pay attention to the shared visits.

The solution methods are compared with the programme ABP developed for more complex problems, and the tests showed that the solutions found by both the insertion heuristic and the tabu search are better than the solutions obtained in ABP. When only considering the total travelling time the solutions found the insertion heuristic gave an improvement average of 25% and tabu search gave an improvement average on 38 %. The number of visits without a regular caretaker in the solutions can not be compared directly, because the ABP focuses more on the travelling time. The results show, that the number of visits without a regular caretaker can be improved, while the total travelling time is also improved. All the results show how it is possible to get high quality solutions, if the problem is limited.

Bibliography

[Ban] Danish Statistics Banc. www.statistikbanken.dk.

[BF05] Stefan Bertels and Torsten Fahle. A hybrid setup for a hybrid scenario: combining heurisitcs for the home health care problem. *Computer & Operations Research*, 33:2866–2890, 2005.

[Brä01] Olli Bräysy. *Local Search and Variable Neighborhood. Search Algorihms for the Vehicle Routing Problem with Time Windows*. Universitas Wasaensis Vaasa, 2001.

[Car] Zealand Care. www.zealandcare.dk, 25th january 2006.

[CLA01] J-F Cordeau, G. Laporte, and Mercier A. A unified tabu search heuristic for vehicle rounting problems with time windows. *Journal of the Operational Research Society*, 52:928–936, 2001.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT press under a joint production-distribution agreement with the McGraw-Hill Book Compagny, 1990.

[CR97] Wen Chyuan Chlang and Robert A Russell. A reactive tabu search metaheuristic for the vehicle routing problem with time windows. *INFORMS Journal on Computing*, 9:417–430, 1997.

[EFR03] Patrik Eveborn, Patrik Flisberg, and Mikael Rönnqvist. Staff planning in the swedish home care system. *Optimal Solutions AB, Teknikringen 1A, SE-58330 Linköping, Sweden and Division of Optimization, Linköping Institue of Technologu, SE-58183 Linköping, Sweden*, 2003.

[Lin65] S. Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44:2245–2269, 1965.

[Mar04] Richard Kipp Martin. *Large Scale Linear and Integer Optimization: a Unified Approach*. KluwerAcademic Publishers, 2004.

[Nej] Lars-Ole Nejstgaard. The drawning is found at the homepage www.forflyt.dk. the secretariat of working environment (in danish: Arbejdsmiljøsekretariatet) has given the permission to use the drawning in this report.

[Or76] I. Or. *Traveling Salesman-Type Combinatorial Problems and their Relation to the Logistics of Regional Blood Banking*. Ph.D. Thesis, Northwestern University, Evanston, U.S.A., 1976.

[PJM93] Jean-Yves Potvin and Rousseau Jean-Marc. A parallel route building algorithm for the vehicle routing and scheduling problem with time windows. *European Journal of Operational Research*, 66:331–340, 1993.

[Røp05] Stefan Røpke. *Heuristics (Slides from the course 02715 Large-Scale Optimization)*. February, 2005.

[Sav92] W. P. Savelsbergh, Martin. The vehicle routing problem with time windows: Minimizing route duration. *ORSA Journal on Computing*, 4:146–154, 1992.

[Soc] Socialministeriet. www.social.dk.

[Sol87] Marius M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35:254–265, 1987.

[Sti05] Thomas Stidsen. *Project Introduction and Neighborhood Choice - Rules for neighborhoods? (Slides from the course 02715 Large-Scale Optimization)*. Informatics and Mathematical Modeling, Technical University of Denmark, 2005.

[Tho05] Kirstine Thomsen. Planlægning i hjemmeplejen, en spørgeskemaundersøgelse. 2005.

[TV01] Paolo Toth and Daniele Vigo. *The Vehicle Routing Problem*. SIAM Monographs on Discrete Mathematics and Applications, 2001.

[Wol98] Laurence A. Wolsey. *Integer Programming*. Wiley-Interscience publication, 1998.

Appendix A

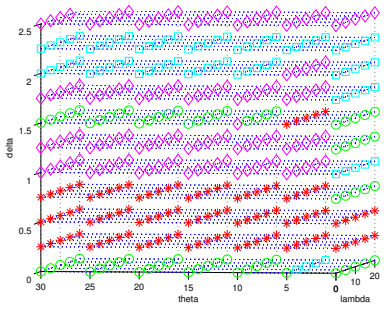
The Figures for Parameter Tuning

The figures in this chapter are used for the parameter tuning. All the figures are 3-d plots, because the parameter tuning involve three parameters δ, θ and λ. The solution value of each solution is indicated using a scala with colors and symbols, which is explained in table A.1.

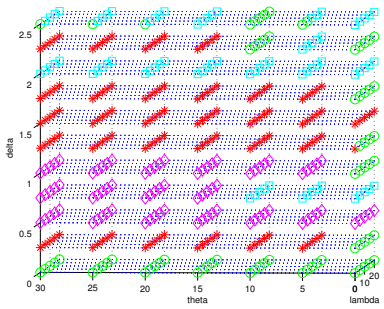
Symbol	Color	Interval for the solution value
Star (*)	Red	$[C(x^*), C(x^*) + (C(x^o) - C(x^*))/4[$
Diamond ($\diamond$)	Pink	$[C(x^*) + (C(x^o) - C(x^*))/4, C(x^*) + (C(x^o) - C(x^*))/2[$
Square ($\square$)	Blue	$[C(x^*) + (C(x^o) - C(x^*))/2, C(x^*) + 3(C(x^o) - C(x^*))/4[$
Circle ($\circ$)	Green	$[C(x^*) + 3(C(x^o) - C(x^*))/4, C(x^o)]$

Table A.1: The symbols used for the figures in the parameter tuning, where $C(x^*)$ is the cost of the best solution found and $C(x^o)$ is the cost of the worst solution found

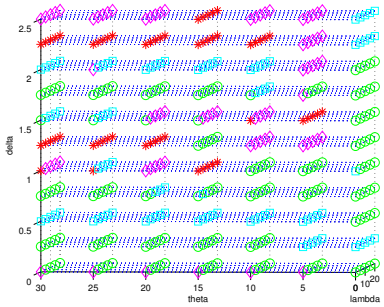
In the figures the parameter μ is changing between the values 0, 5.7 and 11.4. For each setting of the parameter μ, two data sets are used; one with shared visits and one without.



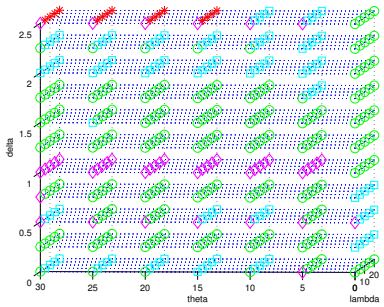
(a) Monday 27th, February 2006. The best solution has the cost 468. The worst solution has the cost 576



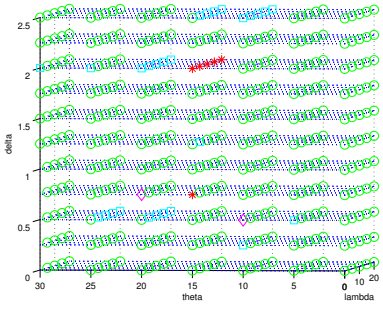
(b) Tuesday 28th, February 2006. The best solution the cost 498. The worst solution has the cost 558



(c) Wednesday 1st, March 2006. The best solution has the cost 472. The worst solution has the cost 539

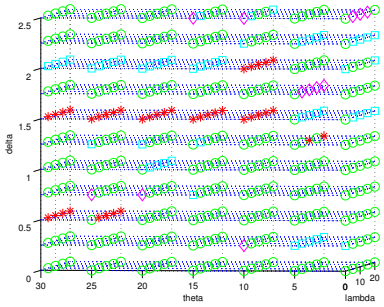


(d) Thursday 2nd, March 2006. The best solution has the cost 533. The worst solution has the cost 598

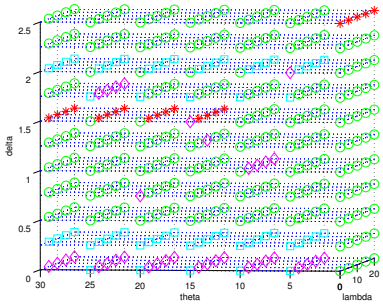


(e) Friday 3rd, March 2006. The best solution has the cost 567. The worst solution has the cost 601

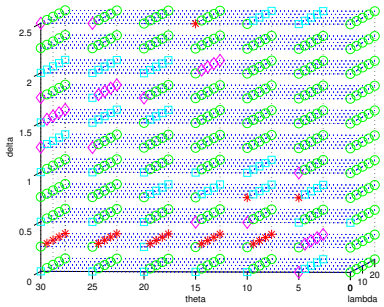
Figure A.1: The results from parameter tuning with $\mu = 0$, week 9, where data contain shared visits.



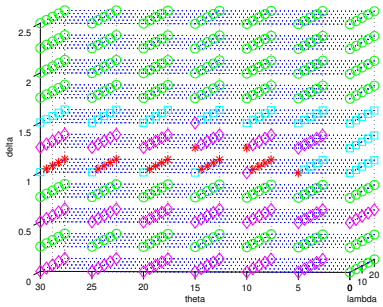
(a) Monday 6th, March 2006. The best solution has the cost 513. The worst solution has the cost 555



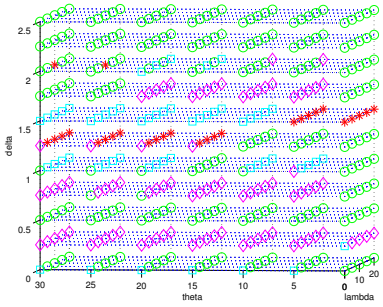
(b) Tuesday 7th, February 2006. The best solution has the cost 507. The worst solution has the cost 558



(c) Wednesday 8th, March 2006. The best solution has the cost 500. The worst solution has the cost 576

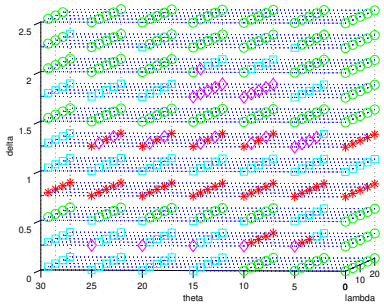


(d) Thursday 9th, March 2006. The best solution has the cost 499. The worst solution has the cost 564

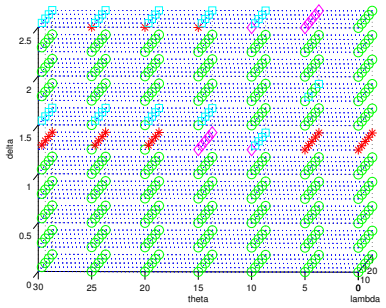


(e) Friday 10th, March 2006. The best solution has the cost 573. The worst solution has the cost 649

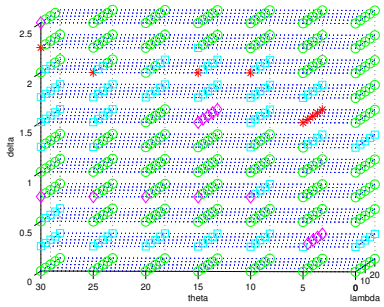
Figure A.2: The results from parameter tuning with $\mu = 0$, week 10, where data contain shared visits



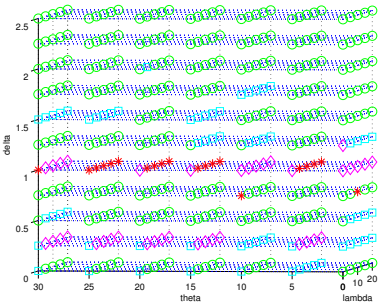
(a) Monday 27th, February 2006. The best solution has the cost 855.2. The worst solution the cost 1020.4



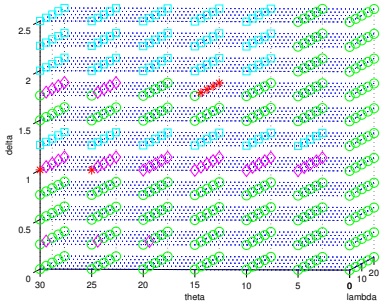
(b) Tuesday 28th, February 2006. The best solution has the cost 1011.6. The worst solution has the cost 1118.3



(c) Wednesday 1st, March 2006. The best solution has the cost 926.4. The worst solution has the cost 1030.5

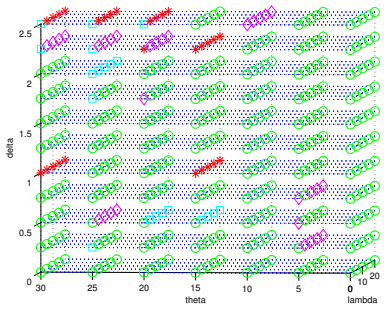


(d) Thursday 2nd, March 2006. The best solution has the cost 971.6. The worst solution has the cost 1093.4

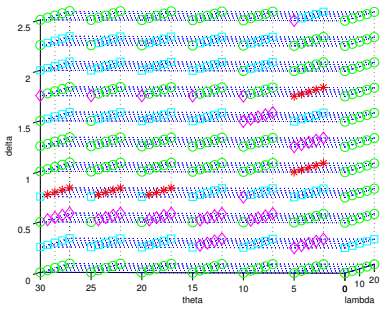


(e) Friday 3rd, March 2006. The best solution has the cost 1076.5. The worst solution has the cost 1164

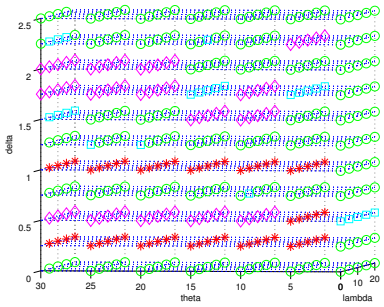
Figure A.3: The results from parameter tuning with $\mu = 5.7$, week 9, and where data contain shared visits.



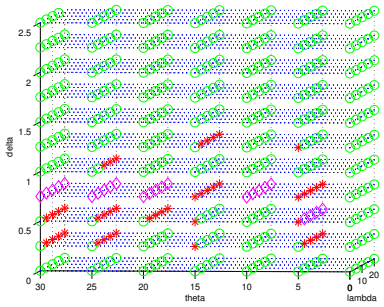
(a) Monday 6th, March 2006. The best solution has the cost 942.9. The worst solution has the cost 1016.8



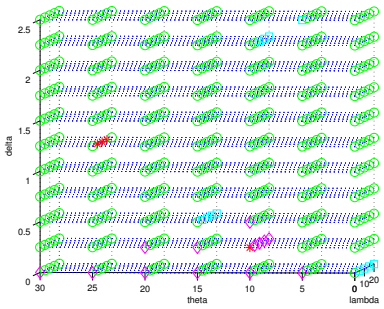
(b) Tuesday 7th, February 2006. The best solution has the cost 896.7. The worst solution has the cost 1026.7



(c) Wednesday 8th, March 2006. The best solution has the cost 902.1. The worst solution has the cost 993.1

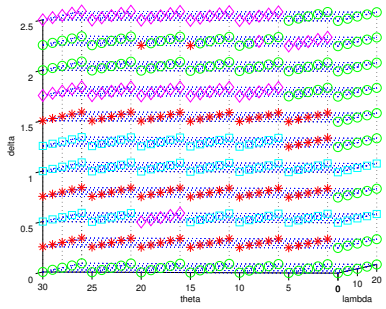


(d) Thursday 9th, March 2006. The best solution has the cost 985.5. The worst solution has the cost 1109.2

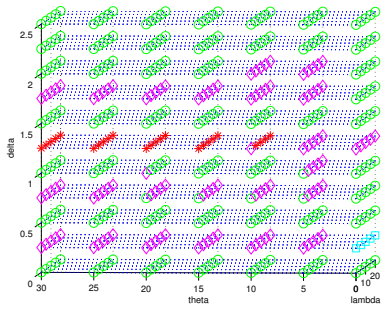


(e) Friday 10th, March 2006. The best solution has the cost 982.3. The worst solution the cost 1023.5

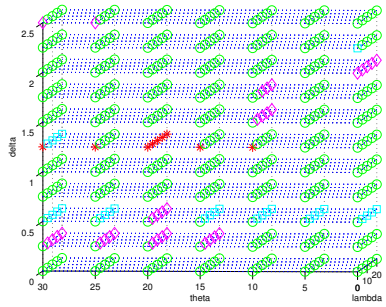
Figure A.4: The results from parameter tuning with $\mu = 5.7$, week 10, and where data contain shared visits.



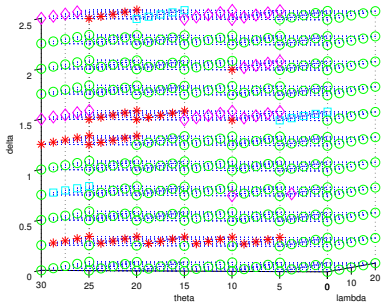
(a) Monday 27th, February 2006. The best solution has the cost 1148.0. The worst solution the cost 1399.4.



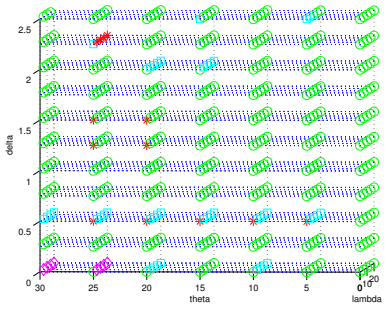
(b) Tuesday 28th, February 2006. The best solution has the cost 1337.4. The worst solution the cost 1661.0.



(c) Wednesday 1st, March 2006. The best solution has the cost 1154.2. The worst solution the cost 1284.4.

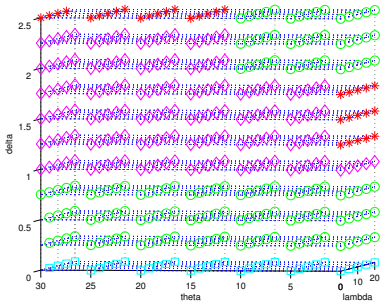


(d) Thursday 2nd, March 2006. The best solution has the cost 1306.6. The worst solution the cost 1445.0.

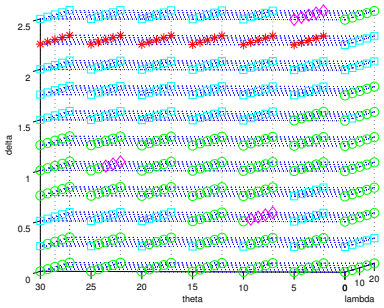


(e) Friday 3rd, March 2006. The best solution has the cost 1400.6. The worst solution the cost 1481.2.

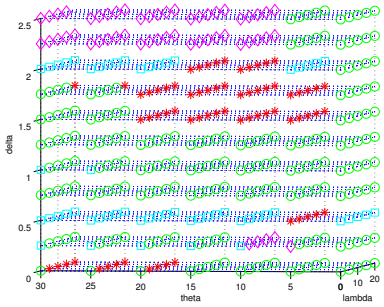
Figure A.5: The results from parameter tuning with $\psi = 11.4$, week 9 and where data contain shared visits



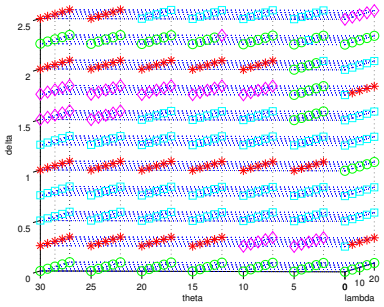
(a) Monday 6th, March 2006. The best solution has the cost 1238.4. The worst solution the cost 1278.9.



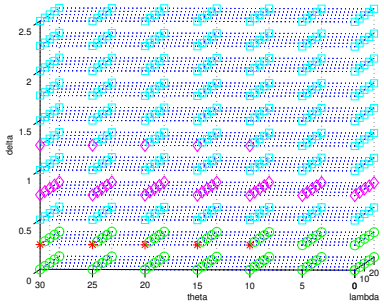
(b) Tuesday 7th, February 2006. The best solution has the cost 1090.8. The worst solution the cost 1318.0.



(c) Wednesday 8th, March 2006. The best solution has the cost 1330.2. The worst solution the cost 1173.0.

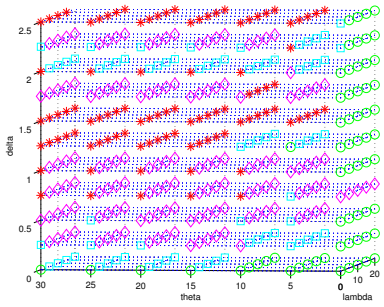


(d) Thursday 9th, March 2006. The best solution has the cost 1159.6. The worst solution the cost 1371.8.

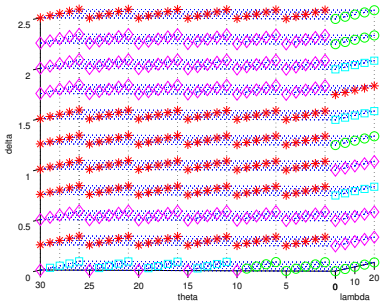


(e) Friday 10th, March 2006. The best solution has the cost 1186.4. The worst solution the cost 1274.0.

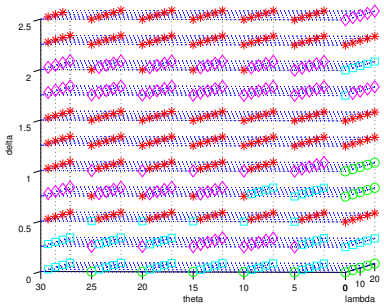
Figure A.6: The results from parameter tuning with $\psi = 11.4$, week 10, and where data contain shared visits



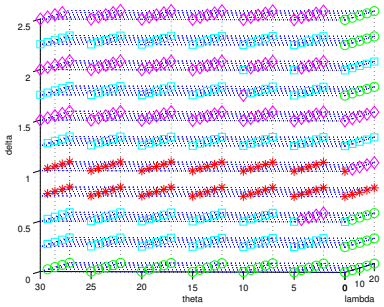
(a) Monday 27th, February 2006. The best solution has the cost 413. The worst solution has the cost 510



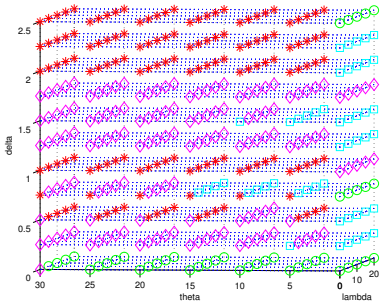
(b) Tuesday 28th, February 2006. The best solution the cost 436. The worst solution has the cost 544



(c) Wednesday 1st, March 2006. The best solution has the cost 439. The worst solution has the cost 546

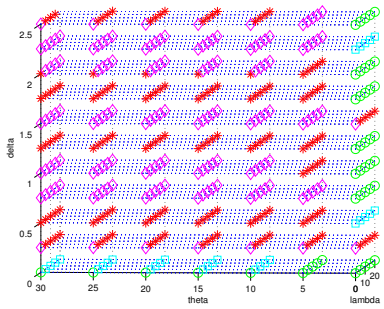


(d) Thursday 2nd, March 2006. The best solution has the cost 446. The worst solution has the cost 579

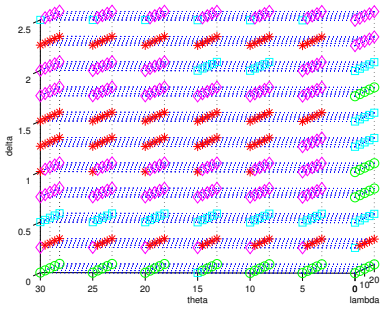


(e) Friday 3rd, March 2006. The best solution has the cost 582. The worst solution has the cost 526

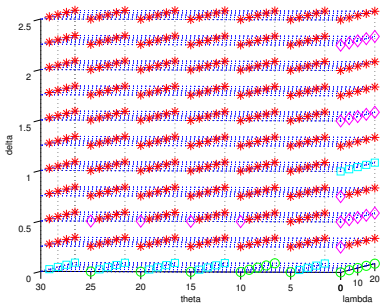
Figure A.7: The results from parameter tuning with $\mu = 0$, week 9. The data do not contain shared visits.



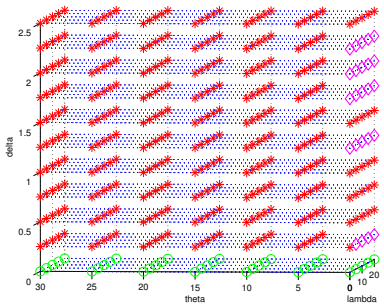
(a) Monday 6th, March 2006. The best solution has the cost 425. The worst solution has the cost 519



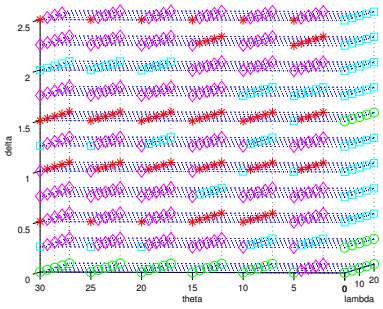
(b) Tuesday 7th, February 2006. The best solution has the cost 438. The worst solution has the cost 534



(c) Wednesday 8th, March 2006. The best solution has the cost 430. The worst solution has the cost 530

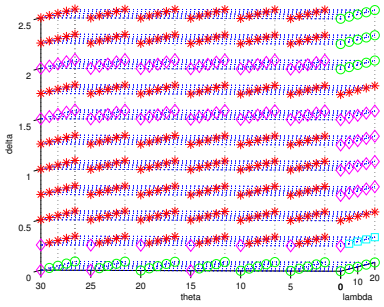


(d) Thursday 9th, March 2006. The best solution has the cost 443. The worst solution has the cost 558

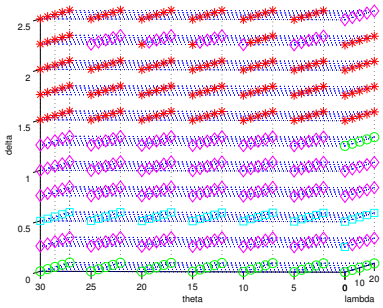


(e) Friday 10th, March 2006. The best solution has the cost 524. The worst solution has the cost 567

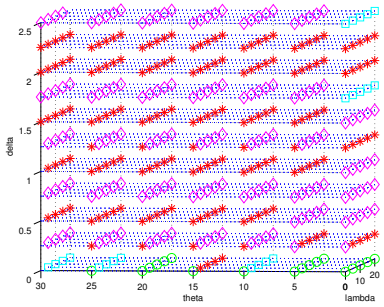
Figure A.8: The results from parameter tuning with $\psi = 0$, week 10. The data do not contain shared visits.



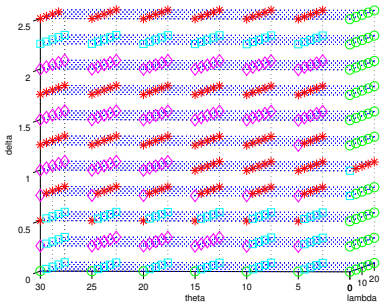
(a) Monday 27th, February 2006. The best solution has the cost 767.3. The worst solution has the cost 991.4



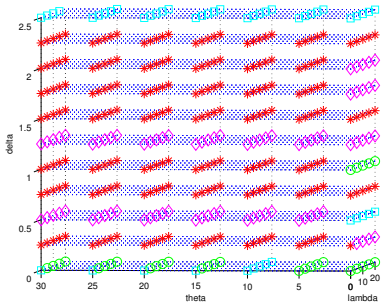
(b) Tuesday 28th, February 2006. The best solution the cost 854.5. The worst solution has the cost 1055.9



(c) Wednesday 1st, March 2006. The best solution has the cost 781.6. The worst solution has the cost 939.9

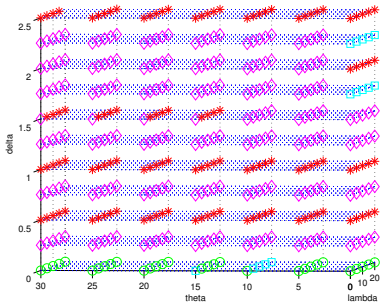


(d) Thursday 2nd, March 2006. The best solution has the cost 940.1. The worst solution has the cost 1121.6

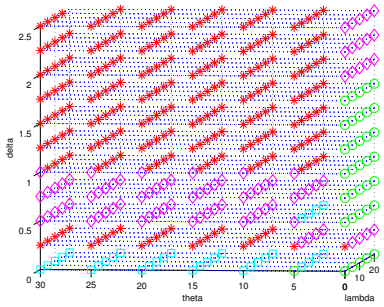


(e) Friday 3rd, March 2006. The best solution has the cost 845.4. The worst solution has the cost 1051.2

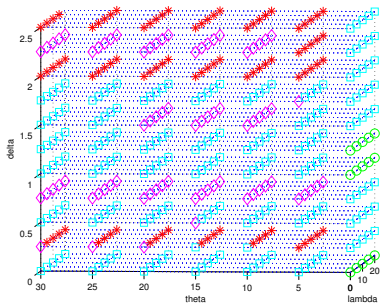
Figure A.9: The results from parameter tuning with $\mu = 5.7$, week 9. The data do not contain shared visits.



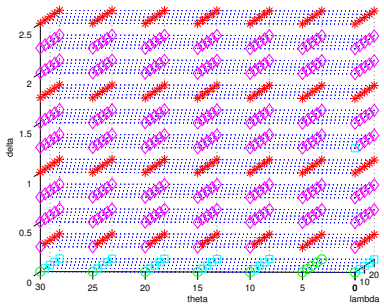
(a) Monday 6th, March 2006. The best solution has the cost 763.3. The worst solution has the cost 905.8



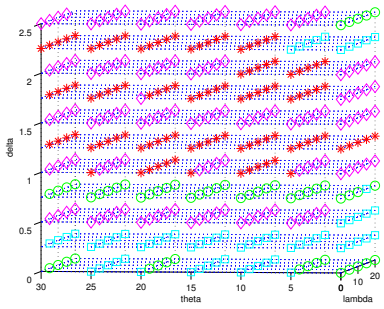
(b) Tuesday 7th, February 2006. The best solution has the cost 775.7. The worst solution has the cost 981.4



(c) Wednesday 8th, March 2006. The best solution has the cost 780.5. The worst solution has the cost 959.7

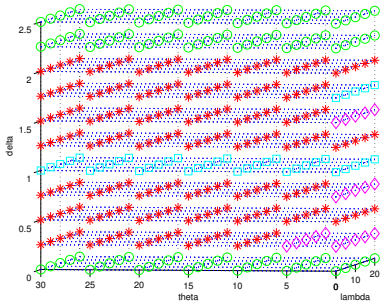


(d) Thursday 9th, March 2006. The best solution has the cost 788.9. The worst solution has the cost 1007.8

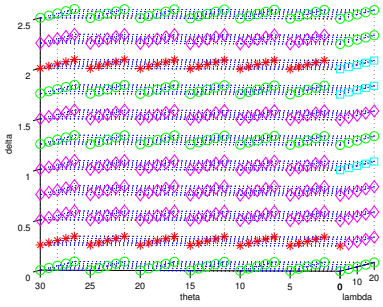


(e) Friday 10th, March 2006. The best solution has the cost 804.2. The worst solution has the cost 926.1

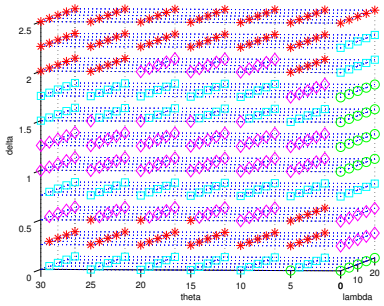
Figure A.10: The results from parameter tuning with $\psi = 5.7$, week 10. The data include no shared visits.



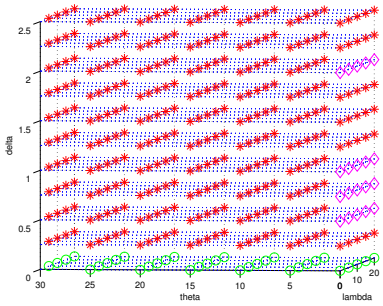
(a) Monday 27th, February 2006. The best solution has the cost 928.4. The worst solution the cost 1257.6.



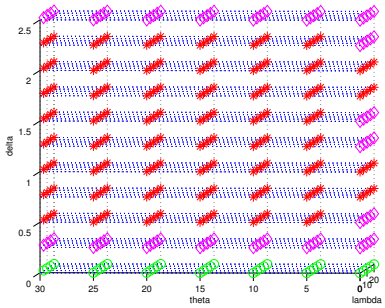
(b) Tuesday 28th, February 2006. The best solution has the cost 1138.0. The worst solution the cost 1372.8.



(c) Wednesday 1st, March 2006. The best solution has the cost 1151.6. The worst solution the cost 925.2.

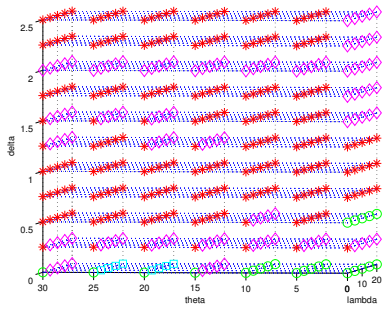


(d) Thursday 2nd, March 2006. The best solution has the cost 1435.8. The worst solution the cost 1080.8.

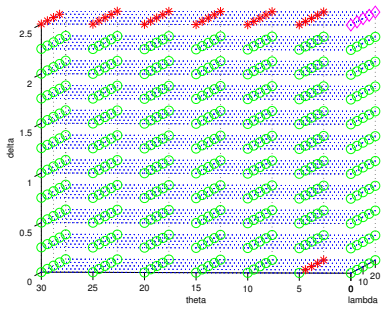


(e) Friday 3rd, March 2006. The best solution has the cost 1529.8. The worst solution the cost 1151.8.

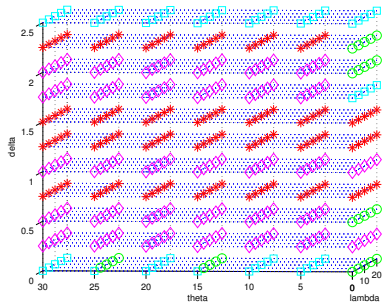
Figure A.11: The results from parameter tuning with $\psi = 11.4$, week 9. The data is without shared visits.



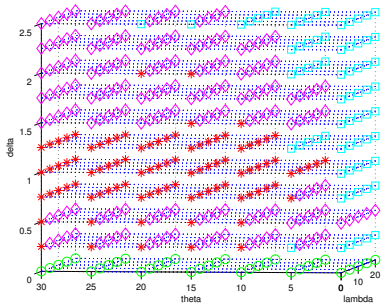
(a) Monday 6th, March 2006. The best solution has the cost 1158.6. The worst solution the cost 916.0.



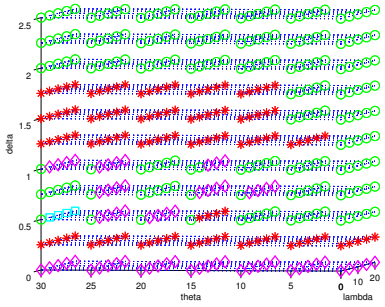
(b) Tuesday 7th, February 2006. The best solution has the cost 1188.2. The worst solution the cost 950.0.



(c) Wednesday 8th, March 2006. The best solution has the cost 856.8. The worst solution the cost 1113.8.



(d) Thursday 9th, March 2006. The best solution has the cost 1040.0. The worst solution the cost 1206.2.



(e) Friday 10th, March 2006. The best solution has the cost 1063.8. The worst solution the cost 1133.8.

Figure A.12: The results from parameter tuning with $\psi = 11.4$, week 10. The data is without shared visits.

Appendix B

The Source Code

B.1 The Source Code for the Objects

B.1.1 Citizen.java

```
import java.util.*;

public class Citizen{

    private int num; //the number of the citizen
    private int distanceNum; // The original numbe of the citizen

    private Worker worker1; // the number of the first regular worker
    private Worker worker2; // the number of the second regular worker
    private Worker worker3; // the number of the third regular worker

 public Citizen(int n1, Worker w1, Worker w2, Worker w3, int n2){
num = n1; worker1 = w1; worker2 = w2; worker3 = w3; distanceNum = n2;
    }


    public Citizen(int n, Worker w1, Worker w2, Worker w3){
num = n; worker1 = w1; worker2 = w2; worker3 = w3; distanceNum = -1;
    }

    public Citizen() {
num = -1; worker1 = new Worker(); worker2 = new Worker(); worker3 = new Worker(); distanceNum = -1;
    }

    public boolean isTheWorkerRegular(Worker w){
boolean answer = false;

int n = w.number();
int n1 = worker1.number();
int n2 = worker2.number();
int n3 = worker3.number();

if( (n == n1 | n == n2) | n == n3){
    answer = true;
}

return answer;
    }

    public int number(){return num;}
    public void setNumber(int n){num = n;}

    public int distanceNumber(){return distanceNum;}
    public void setDistanceNumber(int n){distanceNum = n;}
```

```
    public Worker worker1(){return worker1;}
    public void setWorker1(Worker w1){
worker1 = w1;
    }

    public Worker worker2(){return worker2;}
    public void setWorker2(Worker w2){
worker2 = w2;
    }

    public Worker worker3(){return worker3;}
    public void setWorker3(Worker w3){
worker3 = w3;
    }

    public Citizen copy(){
Citizen newCitizen = new Citizen();

newCitizen.setNumber(num);
newCitizen.setDistanceNumber(distanceNum);

Worker newWorker1 = worker1.copy();
newCitizen.setWorker1(newWorker1);

Worker newWorker2 = worker2.copy();
newCitizen.setWorker2(newWorker2);

Worker newWorker3 = worker3.copy();
newCitizen.setWorker3(newWorker3);

return newCitizen;

    }

    public String toString(){

String output = "citizen "+  num + "/" + distanceNum;
output += "("+ worker1.number() + "," + worker2.number() + "," + worker3.number() + ")";
return output;
    }
    //public String toString(){String output = "citizen "+  originalNum; return output;}

}
```

B.1.2 Route.java

```
import java.math.*;
import java.util.*;

public class Route {

    public Vector v =  new Vector();

    private int num; // The number of the visit.

    private Worker work; // The worker doing this route.

    public Route(int n, Worker w){num = n; work = w;}

    public Route(){num = -1; work = new Worker();}

     //Insert a visit in the route
    public void insert(int i, Visit vi){v.add(i, vi);}

    // Remove a visit in a route
    public void removeVisitAt(int i){v.removeElementAt(i);}

    public void remove(Visit vi){boolean bool = v.remove(vi);}

    //Number of visits in the route
    public int length(){return v.size();}

    //Get Visit at position i in the path
    public Visit get(int i){return (Visit) v.get(i);}

    public void empty(){v.clear();}

    public int number(){return num;}
    public void setNumber(int n){num = n;}
```

```
    public Worker worker(){return work;}
    public void setWorker(Worker w){work = w;}

    public double violation(){

int l = v.size();

Visit lastVisit = (Visit) v.get(l-1);
double finish = lastVisit.finish();

double violation = Math.max(finish-work.finish(),0);
return violation;
    }

    public Route copy(){
Route newRoute = new Route();

// Copy the visits in the vector
for(int i = 0; i < v.size(); i++){
    Visit currentVisit = (Visit) v.get(i);
    Visit newVisit = currentVisit.copy();
    newRoute.insert(i,newVisit);
}

// Copy the numbers
newRoute.setNumber(num);

//Copy the worker
Worker newWorker = work.copy();
newRoute.setWorker(newWorker);

return newRoute;
    }

    public String toString(){

int l = v.size();

Visit lastVisit = (Visit) v.get(l-1);
double finish = lastVisit.finish();

double violation = Math.max(finish-work.finish(),0);

String output = "ROUTE " + num + "/" +  work.originalNumber()  + ": [" + work.start()+ ", " + work.finish()+"]";
output += ", violation = " +  violation + "\n";

//String output = "ROUTE " + work.originalNumber()  + ": [" + work.start()+ ", " + work.finish()+"]\n";

for(int i=0; i<l; i++)
    {
Visit current = (Visit) v.get(i);
output += current.toString() + "\n";
    }

return output;

    }

 public String toString(double[][] distance){

int l = v.size();

Visit lastVisit = (Visit) v.get(l-1);
double finish = lastVisit.finish();

double violation = Math.max(finish-work.finish(),0);

String output = "ROUTE " + num + "/" +  work.originalNumber()  + ": [" + work.start()+ ", " + work.finish()+"]";
output += ", violation = " +  violation + "\n";

//String output = "ROUTE " + work.originalNumber()  + ": [" + work.start()+ ", " + work.finish()+"]\n";

for(int i=0; i<l; i++)
    {
Visit current = (Visit) v.get(i);
double travel = 0;
if(i < l-1){

    Visit next = (Visit) v.get(i+1);
    travel = distance[current.citizen().number()][next.citizen().number()];
}
output += current.toString() + ", travel = " + travel + "\n";
    }
```

```
return output;

    }

}
```

B.1.3 Visit.java

```
import java.math.*;

public class Visit {

    private double open; //the relative time in minutes when the time window starts
    private double closed; //the relative time in minutes when the time window ends
    private double duration; //the duration of the visit in minutes
    private Citizen citizen; //the number of the citizen

    private int number; //the number of the visit

    private boolean byTwo; //alpha = 1 if the visit is done by two persons, 0 otherwise

    private boolean removable; // Indicates whether it is possible to move the visit
    // For instance, break is not removable

    private boolean isPlanned; // Indicate whether the visit is scheduled

    // VARIABLES TO BE SET
    private double arrival; // Arrival Time
    private double wait; // Waiting Time
    private double start;   //s_i The starting time for the visits.
    private double finish; // Finish time

    // The position of the visit when it is inserted
    private int routeNumber;
    private int position;

    public Visit(){
open = -1; closed = -1; duration = -1; citizen = new Citizen(); number = -1;
arrival = -1; wait = -1; start = -1; finish = -1; byTwo = false; removable = true;
routeNumber = -1; position = -1; isPlanned = false;
    }

    public Visit( int n, double o, double cl, double d, Citizen c){
open = o; closed = cl; duration = d; citizen = c; number = n;
arrival = -1; wait = -1; start = -1; finish = -1; byTwo = false; removable = true;
routeNumber = -1; position = -1; isPlanned = false;
    }

    public Visit( int n, double o, double cl, double d, Citizen c, boolean two, boolean remove){
open = o; closed = cl; duration = d; citizen = c;
number = n; arrival = -1; wait = -1; start = -1; finish = -1;byTwo = two; removable = remove;
routeNumber = -1; position = -1; isPlanned = false;
    }

    public Visit( int n, double o, double cl, double d, Citizen c, boolean two, boolean remove,  double a,
  double w, double s, double f){
open = o; closed = cl; duration = d; citizen = c; number = n;
arrival = a; wait = w; start = s; finish = f; byTwo = two; removable = remove;
routeNumber = -1; position = -1; isPlanned = false;
    }

    public Visit( int n, double o, double cl, double d, Citizen c, boolean two, boolean remove,  double a,
  double w, double s, double f, int r, int p){
open = o; closed = cl; duration = d; citizen = c; number = n;
arrival = a; wait = w; start = s; finish = f; byTwo = two; removable = remove;
routeNumber = r; position = p; isPlanned = false;
    }

 public Visit( int n, double o, double cl, double d, Citizen c, boolean two, boolean remove,  double a,
       double w, double s, double f, int r, int p, boolean i){
open = o; closed = cl; duration = d; citizen = c; number = n;
arrival = a; wait = w; start = s; finish = f; byTwo = two; removable = remove;
routeNumber = r; position = p; isPlanned = i;
    }

    public void setOpen(double o){open = o;}
    public double open(){return open ;}
```

```
    public void setClosed(double cl){closed =cl;}
    public double closed(){return closed;}

    public void setDuration(double d){duration = d;}
    public double duration(){return duration;}

    public void setCitizen(Citizen c){citizen = c;}
    public Citizen citizen(){return citizen;}

    public void setNumber(int n){number = n;}
    public int number(){return number;}

    public void setArrival(double a){arrival = a;}
    public double arrival(){return arrival;}

    public void setWaitingTime(double w){wait = w;}
    public double waitingTime(){return wait;}

    public void setStart(double s){start = s;}
    public double start(){return start;}

    public void setFinish(double f){finish = f;}
    public double finish(){return finish;}

    public void setIsShared(boolean two){byTwo = two;}
    public boolean isShared(){return byTwo;}

    public boolean removable(){return removable;}
    public void setRemovable(boolean remove){
removable = remove;}

    public int routeNumber(){return routeNumber;}
    public void setRouteNumber(int r){routeNumber = r;}

    public int position(){return position;}
    public void setPosition(int p){position = p;}

    public boolean isPlanned(){return isPlanned;}
    public void setIsPlanned(boolean i){isPlanned = i;}


    public double violation(){

double violation;
violation = Math.max(start-closed,0);
return violation;
    }

    public Visit copy(){

Visit newVisit = new Visit();

newVisit.setOpen(open);
newVisit.setClosed(closed);
newVisit.setDuration(duration);

Citizen newCitizen = citizen.copy();
newVisit.setCitizen(newCitizen);

newVisit.setNumber(number);
newVisit.setArrival(arrival);
newVisit.setWaitingTime(wait);
newVisit.setStart(start);
newVisit.setFinish(finish);
newVisit.setIsShared(byTwo);
newVisit.setRemovable(removable);
newVisit.setRouteNumber(routeNumber);
newVisit.setPosition(position);
newVisit.setIsPlanned(isPlanned);

return newVisit;
    }


    public String toString(){

double violation = Math.max(start-closed,0);

String output = " Visit " + number + " with " + citizen.toString();
output += ", arrive " + arrival + ", wait " + wait + ", start " + start;
output += ", [" + open + "," + closed + "]";
output += ", last " + duration + ", finish " + finish;
output += ", route " + routeNumber + ", pos "+ position ;
output += ", violation = " + violation;
```

```
if(byTwo){
    output += " *";
}

//output += "\n";

return output;}

}
```

B.1.4 Worker.java

```
import java.util.*;

public class Worker{

    private int num; //the number of the worker
    private int originalNum; // The number in the data set

    private double start; // the starting time
    private double finish; // the finishing time

 public Worker(int n1,  double s, double f, int n2){
num = n1; start = s; finish = f;  originalNum = n2;
    }

    public Worker(int n, double s, double f){
num = n; start = s; finish = f;  originalNum = -1;
    }

    public Worker(){
num = -1; start = -1; finish = -1; originalNum = -1;
    }

    public int number(){return num;}
    public void setNumber(int n){num = n;}

    public int originalNumber(){return originalNum;}
    public void setOriginalNumber(int n){originalNum = n;}

    public double start(){return start;}
    public void setStart(double s){start = s;}

    public double finish(){return finish;}
    public void setFinish(double f){finish = f;}

    public Worker copy(){

Worker newWorker = new Worker();
newWorker.setNumber(num);
newWorker.setOriginalNumber(originalNum);
newWorker.setStart(start);
newWorker.setFinish(finish);

return newWorker;

    }

    public String toString(){String output = "worker "+  num; return output;}

}
```

B.1.5 Solution.java

```
import java.util.*;

public class Solution{

    // All the routes
    private Route[] allRoutes;

    // All the visits
    private Visit[] allVisits;
```

```
    // The costs
    private Cost cost;

    // Is the solution feasible
    private boolean isTheSolutionFeasible;


    // The constructor
    public Solution(Route[] a1, Visit[] a2, boolean i, Cost c){
allRoutes = a1;
allVisits = a2;
isTheSolutionFeasible = i;
cost = c;
    }

    // The constructor
    public Solution(Route[] a1, Visit[] a2, boolean i){
allRoutes = a1;
allVisits = a2;
isTheSolutionFeasible = i;
cost = new Cost();
    }

    // Overloading the constructor again
    public Solution(){
allRoutes = new Route[0];
allVisits = new Visit[0];
isTheSolutionFeasible = true;
cost = new Cost();
    }

    public void setAllRoutes(Route[] a){
allRoutes = a;}
    public Route[] allRoutes(){
return allRoutes;}

    public void setAllVisits(Visit[] a){allVisits = a;}
    public Visit[] allVisits(){return allVisits;}

    public void setCost(Cost c){
cost = c;}
    public Cost cost(){
return cost;}

    public void setIsFeasible(boolean i){
isTheSolutionFeasible = i;}
    public boolean isFeasible(){
return isTheSolutionFeasible;}

    public void setTotalTravelTime(double t){
cost.setTotalTravelTime(t); }
    public double totalTravelTime(){return cost.totalTravelTime();}

    public void setNoOfVisitsWithoutRegularWorker(int n){
    cost.setNoOfVisitsWithoutRegularWorker(n);}
    public int noOfVisitsWithoutRegularWorker(){return cost.noOfVisitsWithoutRegularWorker();}

    public void setDifferenceStartingTimesSharedVisits(double d){
cost.setDifferenceStartingTimesSharedVisits(d);}
    public double differenceStartingTimesSharedVisits(){
return cost.differenceStartingTimesSharedVisits();}

    public void setViolationOfTimeWindows(double v){
cost.setViolationOfTimeWindows(v);}
    public double violationOfTimeWindows(){
return cost.violationOfTimeWindows();}

    public void setViolationOfWorkingHours(double v){
cost.setViolationOfWorkingHours(v);}
    public double violationOfWorkingHours(){
return cost.violationOfWorkingHours();}

    public double c(double psi){
double c;
c = cost.totalTravelTime() + psi*cost.noOfVisitsWithoutRegularWorker();
return c;
    }

    public double f(double psi, double alpha, double beta, double gamma){

double f;
f = cost.totalTravelTime() + psi*cost.noOfVisitsWithoutRegularWorker();
f += alpha*cost.violationOfTimeWindows() ;
```

```
f += beta*cost.differenceStartingTimesSharedVisits();
f += gamma*cost.violationOfWorkingHours();
return f;
    }

    public Solution copy(){

Solution newSolution = new Solution();

// Copy all routes
Route[] newAllRoutes = new Route[allRoutes.length];
for(int i = 1; i < allRoutes.length; i++){
    Route currentRoute = allRoutes[i];
    Route newRoute =  currentRoute.copy() ;
    newAllRoutes[i] = newRoute;
}
newSolution.setAllRoutes(newAllRoutes);

// Copy all the visits
Visit[] newAllVisits = new Visit[allVisits.length];
for(int j = 1; j < allVisits.length; j++){
    //System.out.println("j = " + j);
    Visit currentVisit = allVisits[j];
    //System.out.println("currentVisit = " + currentVisit);
    Visit newVisit = currentVisit.copy();
    newAllVisits[j] = newVisit;
}
newSolution.setAllVisits(newAllVisits);

// Copy the costs
Cost newCost = cost.copy();
newSolution.setCost(newCost);

// Copy the feasibility
newSolution.setIsFeasible(isTheSolutionFeasible);

return newSolution;

    }

}
```

B.1.6 Cost.java

```
import java.util.*;

public class Cost{

    // The costs
    private double totalTravelTime;
    private int noOfVisitsWithoutRegularWorker;
    private double differenceStartingTimesSharedVisits;
    private double violationOfTimeWindows;
    private double violationOfWorkingHours;

    // The constructor
    public Cost(double t, int n, int d, double v1, double v2){
totalTravelTime = t;
noOfVisitsWithoutRegularWorker = n;
differenceStartingTimesSharedVisits = d;
violationOfTimeWindows = v1;
violationOfWorkingHours = v2;
    }

    public Cost(double t, int n){
totalTravelTime = t;
noOfVisitsWithoutRegularWorker = n;
differenceStartingTimesSharedVisits = 0;
violationOfTimeWindows = 0;
violationOfWorkingHours = 0;
    }

    // Overloading the constructor again
    public Cost(){
totalTravelTime = 0;
noOfVisitsWithoutRegularWorker = 0;
differenceStartingTimesSharedVisits = 0;
violationOfTimeWindows = 0;
violationOfWorkingHours = 0;
```

```
    }

    public void setTotalTravelTime(double t){
totalTravelTime = t; }
    public double totalTravelTime(){return totalTravelTime;}

    public void setNoOfVisitsWithoutRegularWorker(int n){
    noOfVisitsWithoutRegularWorker = n;}
    public int noOfVisitsWithoutRegularWorker(){return noOfVisitsWithoutRegularWorker;}

    public void setDifferenceStartingTimesSharedVisits(double d){
differenceStartingTimesSharedVisits = d;}
    public double differenceStartingTimesSharedVisits(){
return differenceStartingTimesSharedVisits;}

    public void setViolationOfTimeWindows(double v){
violationOfTimeWindows = v;}
    public double violationOfTimeWindows(){
return violationOfTimeWindows;}

    public void setViolationOfWorkingHours(double v){
violationOfWorkingHours = v;}
    public double violationOfWorkingHours(){
return violationOfWorkingHours;}

    public double c(double psi){
double cost;
cost = totalTravelTime + psi*noOfVisitsWithoutRegularWorker;
return cost;
    }

    public double f(double psi, double alpha, double beta, double gamma){

double cost;
cost = totalTravelTime + psi*noOfVisitsWithoutRegularWorker;
cost += alpha*violationOfTimeWindows ;
cost += beta*differenceStartingTimesSharedVisits;
cost += gamma*violationOfWorkingHours;
return cost;
    }

    public Cost copy(){

Cost newCost = new Cost();

// Copy the costs
newCost.setTotalTravelTime(totalTravelTime);
newCost.setNoOfVisitsWithoutRegularWorker(noOfVisitsWithoutRegularWorker);
newCost.setDifferenceStartingTimesSharedVisits(differenceStartingTimesSharedVisits);
newCost.setViolationOfTimeWindows(violationOfTimeWindows);
newCost.setViolationOfWorkingHours(violationOfWorkingHours);

return newCost;

    }

}
```

B.2 The Source Code for the Insertion Heuristic

B.2.1 Insertion.java

```
import java.util.*;
import java.math.*;

class Insertion{

    public int nCitizens; //The number of citizens
    public int nVisits; //The number of visits
    public int maxVisitNumber; //The largest visit number
    public int nRoutes; //The number of routes
    public int nWorkers; // The nuber of workers
      public double[][] distance; //The distances in minutes between citizens
    public Visit[] allVisits; // Array with all the visits
    public Vector allVisitsVector; // Vector with all the initial visits
     public int nStartVisits; // Number of start visits
```

```
    public int nBreakVisits; // Number of breaks
    //public int firstStartVisitNumber; // The number of the first start visit. The rest is increasing by 1
    //public int firstBreakVisitNumber; // The number of the first break;
    public Worker[] allWorkers; // Array with all workers
    public int[] theOtherVisit; // What is the number of the corresponding visit, if the visit is shared.
    public Vector allNotPlannedVisits = new Vector(); //Vector with all not planned visits
    public Vector allPlannedVisits = new Vector(); //Vector with all planned visits
    public Route[] allRoutes; //Array with all routes
    public double totalTravelTime;
    public int noOfVisitsWithoutRegularWorker;
    public double priceNotRegularWorker;

    // The number of iterations
    public int nIterations;

    public Insertion(Data data){

nCitizens = data.nCitizens();
maxVisitNumber = data.maxVisitNumber();
nWorkers = data.nWorkers();
nRoutes = data.nRoutes();
distance = data.distance();
allVisits = data.allVisits();
allWorkers =  data.allWorkers();
allRoutes = data.allRoutes();
theOtherVisit = data.theOtherVisit();
nBreakVisits = data.nBreakVisits();
nStartVisits = data.nStartVisits();
nVisits = data.nVisits();

    }

    public Solution start(double price){

//System.out.println("maxVisitNumber  = " +maxVisitNumber);
// Make a vector with visits not scheduled
for(int i = 1; i <= maxVisitNumber; i++){


    Visit currentVisit = allVisits[i];
    //System.out.println("currentVisit.number()  = " + currentVisit.number());
    // If the visit is not removed or simply not there
    if(!currentVisit.isPlanned()){
allNotPlannedVisits.add(currentVisit);
    }

}

/*// PRINT OUT THE ROUTES
System.out.println("\n INITIAL ROUTES \n");
for(int j = 1; j <= nRoutes; j++){
    Route currentRoute = allRoutes[j];
    // PRINT OUT THE ROUTES
    System.out.println(currentRoute.toString() + "\n");
    }*/


//System.out.println("allNotPlannedVisits.size() = " + allNotPlannedVisits.size());

// A function to check the lower bound of how many visits there are visits their regular worker at work
// which do have the regular worker at job.
int noOfVisitsWithoutRegularWorkerAtWork_LB = 0;
int noOfVisitsWithRegularWorkerAtWork = 0;
Vector allVisitsNotInvestigated = new Vector();
// Make a vector with the not investigated visits
for(int i = 1; i <= maxVisitNumber; i++){

    Visit currentVisit = allVisits[i];

    // If the visit is not removed or simply not there
    if(!currentVisit.isPlanned()){
allVisitsNotInvestigated.add(currentVisit);
    }
}

while(allVisitsNotInvestigated.size() != 0){

    Visit currentVisit = (Visit) allVisitsNotInvestigated.get(0);
    Citizen currentCitizen = currentVisit.citizen();

    // While we have not found a regular worker
    boolean isThereARegularWorker = false;
    int j = 1;
    while(!isThereARegularWorker & j <= nWorkers){
```

```
Worker currentWorker = allWorkers[j];
    // Indication whether the current worker is equal to one of the regular workers
isThereARegularWorker = currentCitizen.isTheWorkerRegular(currentWorker);
j++;
    }

    if(!isThereARegularWorker){
    noOfVisitsWithoutRegularWorkerAtWork_LB++ ;
    }else{
noOfVisitsWithRegularWorkerAtWork++;
    }

    if(currentVisit.isShared()){

int theOtherVisitNumber = theOtherVisit[currentVisit.number()];
Visit theOtherVisit = allVisits[theOtherVisitNumber];

allVisitsNotInvestigated.remove(theOtherVisit);
    }

    allVisitsNotInvestigated.remove(currentVisit);
}

//System.out.println("noOfVisitsWithoutRegularWorkerAtWork_LB = " + noOfVisitsWithoutRegularWorkerAtWork_LB);
//System.out.println("noOfVisitsWithRegularWorkerAtWork = " +noOfVisitsWithRegularWorkerAtWork);


// The costs
totalTravelTime = 0;
noOfVisitsWithoutRegularWorker = 0;
priceNotRegularWorker = price;

nIterations = 0;

// When to stop the while loop
boolean stop = false;

//Initial time
Date startTime = new Date();

// HERE THE INSERTION STARTS!!
while(allNotPlannedVisits.size() != 0 & !stop){
    //while(nIterations < 12){

    // Is set to true, when a feasible insertion is found
    boolean thereIsAFeasiblePosition = false;

    //// System.out.println("\n Iterations = " + nIterations + "\n");

    // PRINT OUT THE ROUTES
    //for(int j = 1; j <= nRoutes; j++){
    // Route currentRoute = allRoutes[j];
// PRINT OUT THE ROUTES
//// System.out.println(currentRoute.toString() + "\n");
    //}

    Visit bestVisit = (Visit) allNotPlannedVisits.get(0);
    double bestCost = -1000; // For comparison
    int bestPosition1 = 0;
    int bestPosition2 = 0;
    int bestRouteNumber1 =  1;
    int bestRouteNumber2 =  1;


    // FOR ALL THE NOT SCHEDULED VISITS:
    for(int u = 0; u < allNotPlannedVisits.size(); u++ ){

//// System.out.println("u = " + u);
///System.out.println("allNotPlannedVisits.size() = " + allNotPlannedVisits.size());

// Which visit is currently observed?
Visit currentVisit = (Visit) allNotPlannedVisits.get(u);
//System.out.println("currentVisit = " + currentVisit);

double currentCost;
boolean feasibleVisit;
int position1;
int position2;
int routeNumber1;
int routeNumber2 ;

if(currentVisit.isShared()){

    InsertionCostTwoVisits currentInsertionCost = findInsertionCostTwoEqualVisits(currentVisit);
```

```
    currentCost = currentInsertionCost.cost();
    feasibleVisit = currentInsertionCost.isThereAFeasiblePosition();
    position1 = currentInsertionCost.bestPosition1();
    position2 = currentInsertionCost.bestPosition2();
    routeNumber1 = currentInsertionCost.bestRouteNumber1();
    routeNumber2 = currentInsertionCost.bestRouteNumber2();

    if(feasibleVisit){
thereIsAFeasiblePosition = true;
// Find the best visit number
// System.out.println(">currentCost =" + currentCost);
if(currentCost > bestCost){
    bestCost = currentCost;
    //  System.out.println(">bestCost =" + bestCost);
    bestVisit = currentVisit;
    // System.out.println(">bestVisit.number() ="+  bestVisit.number());
    bestPosition1 = position1;
    // System.out.println(">bestPosition1 ="+ bestPosition1 );
    bestPosition2 = position2;
    // System.out.println(">bestPosition2 ="+ bestPosition2 );
    bestRouteNumber1 = routeNumber1;
    // System.out.println(">bestRouteNumber1 ="+ bestRouteNumber1 );
    bestRouteNumber2 = routeNumber2;
    //System.out.println(">bestRouteNumber2 ="+ bestRouteNumber2 );
}
    }
}
else{
    InsertionCostOneVisit currentInsertionCost = findInsertionCostOneVisit(currentVisit);
    currentCost = currentInsertionCost.cost();
    feasibleVisit = currentInsertionCost.isThereAFeasiblePosition();
    position1 = currentInsertionCost.bestPosition();
    routeNumber1 = currentInsertionCost.bestRouteNumber();

    if(feasibleVisit){
thereIsAFeasiblePosition = true;
//// System.out.println("Feasible solution, >currentCost =" + currentCost);
// Find the best visit number
if(currentCost > bestCost){
    bestCost = currentCost;
    //// System.out.println(">bestCost ="+  bestCost);
    bestVisit = currentVisit;
    //// System.out.println(">bestVisit.number() ="+  bestVisit.number());
    bestPosition1 = position1;
    //// System.out.println(">bestPosition1 ="+ bestPosition1 );
    bestRouteNumber1 = routeNumber1;
    //// System.out.println(">bestRouteNumber1 ="+ bestRouteNumber1 );
}
    }


}
    }


    if(thereIsAFeasiblePosition){


if(bestVisit.isShared()){

    // How to find the other visit!!
    int theOtherVisitNumber = theOtherVisit[bestVisit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];
    //// System.out.println("the other visit number = " + theOtherVisit.number());
    insertTwoEqualVisits(bestVisit, theOtherVisit, bestPosition1, bestPosition2,
 bestRouteNumber1, bestRouteNumber2);
}
else{
    //// System.out.println(">bestVisit ="+  bestVisit);
    //// System.out.println(">bestPosition1 ="+  bestPosition1);
    //// System.out.println(">bestRouteNumber1 ="+  bestRouteNumber1);
    insertOneVisit(bestVisit, bestPosition1, bestRouteNumber1);
}
    }
    else{
System.out.println("IT IS NOT POSSIBLE TO SCHEDULE ALL VISITS");
System.out.println("There are " + allNotPlannedVisits.size() + " missing");
for(int i = 0; i < allNotPlannedVisits.size(); i++){
    Visit currentNotPlannedVisit = (Visit) allNotPlannedVisits.get(i);
    System.out.println(currentNotPlannedVisit);
}

stop = true;
    }
```

```
    nIterations++;
}

Date stopTime = new Date();

long time = stopTime.getTime()-startTime.getTime();


// PRINT OUT THE ROUTES
System.out.println("\n FINAL ROUTES \n");
for(int j = 1; j <= nRoutes; j++){
    Route currentRoute = allRoutes[j];
    // PRINT OUT THE ROUTES
    System.out.println(currentRoute.toString(distance) + "\n");
}

// CHECK THE TRAVELLING TIME
double totalTravelTimeCheck = 0;;
for(int j = 1; j <= nRoutes; j++){
    Route route = allRoutes[j];
    for(int p = 1; p < route.length(); p++){
Visit visit = (Visit) route.get(p);
    // The previous visit
    Visit previousVisit = (Visit) route.get(p-1);
    Citizen previousCitizen = previousVisit.citizen();
    // The current visit
    visit = (Visit) route.get(p);
    Citizen citizen = visit.citizen();
    // Calculate travel time
    totalTravelTimeCheck += distance[previousCitizen.number()][citizen.number()];
    }
}
System.out.println("Total travelling time check= " + totalTravelTimeCheck);
System.out.println("The solution is found in the time " + time);


// When stop is false -> there s a feasible solution
// When stop is true -> the solution is not feasible
Solution initialSolution;
initialSolution = new Solution(allRoutes, allVisits, !stop);
initialSolution.setTotalTravelTime(totalTravelTime);
initialSolution.setNoOfVisitsWithoutRegularWorker(noOfVisitsWithoutRegularWorker);

Vector allVisitsDuringTheDay = new Vector();
for(int j = 1; j <= nRoutes; j++){
    Route currentRoute = allRoutes[j];
    for(int i = 0; i < currentRoute.length(); i++){
Visit currentVisit = (Visit) currentRoute.get(i);
allVisitsDuringTheDay.add(currentVisit);
    }
}


/*
System.out.println("nVisits = " + nVisits);
System.out.println("nWorkers = " + nWorkers);
System.out.println("nStartVisits = " + nStartVisits);
System.out.println("nBreakVisits = " + nBreakVisits);
System.out.println("nCitizens = " + nCitizens);


// Find out how many of the citizens are visited on that day!!
int[] isTheCitizenVisitedThatDay = new int[nCitizens+1];
int nCitizensOnThatDay = 0;
for(int i = 0; i < allVisitsDuringTheDay.size()-1; i++){
    Visit visit = (Visit) allVisitsDuringTheDay.get(i);
    Citizen citizen = visit.citizen();
            int citizenNumber = citizen.number();
    if(isTheCitizenVisitedThatDay[citizenNumber] == 0){
isTheCitizenVisitedThatDay[citizenNumber] = 1;
nCitizensOnThatDay++;
    }

    }

System.out.println("Number of citizens visited on that day = " + nCitizensOnThatDay);


// The total number of arcs necessary is nVisits - nRoutes
int nArcsNecessary = nVisits-nRoutes;
System.out.println("nArcsNecessary = " + nArcsNecessary);
double nArcsDouble = (allVisitsDuringTheDay.size()-1)*allVisitsDuringTheDay.size();
```

```
int nArcs = (int) nArcsDouble;
System.out.println("nArcs = " + nArcs);
double[] sortedDistances = new double[nArcs];
// Find the nArcs shortest arcs (e.g. travelling times between citizens of the visits)
int nDistances = 0;
double sumDistances = 0;
for(int i = 0; i < allVisitsDuringTheDay.size()-1; i++){
    Visit visit1 = (Visit) allVisitsDuringTheDay.get(i);
    Citizen citizen1 = visit1.citizen();
            int citizen1number = citizen1.number();
    for(int j = i+1; j < allVisitsDuringTheDay.size(); j++){
Visit visit2 = (Visit) allVisitsDuringTheDay.get(j);
Citizen citizen2 = visit2.citizen();
int citizen2number = citizen2.number();
double distanceForth = distance[citizen1number][citizen2number];
double distanceBack = distance[citizen2number][citizen1number];
sumDistances += distanceForth+distanceBack  ;

int d = 0;
// the distances are sorted in decreasing order!
while(distanceForth <  sortedDistances[d]){d++;}
//push forward
for(int k = nDistances; k >= d; k-- ){
   if(d != 0){
   sortedDistances[k] = sortedDistances[k-1];
   }
}
sortedDistances[d] = distanceForth;

nDistances++;

d = 0;
// the distances are sorted in decreasing order!
while(distanceBack <  sortedDistances[d]){d++;}
//push forward
for(int k = nDistances; k >= d; k-- ){
   if(d != 0){
   sortedDistances[k] = sortedDistances[k-1];
   }
}
sortedDistances[d] = distanceBack;

nDistances++;

    }

}

// for(int k = 0; k < nDistances; k++){
//System.out.print( sortedDistances[k] + ",");
//}
//System.out.println( "\n");
//System.out.println("sortedDistances.length = "+sortedDistances.length);

double totalDistanceLB = 0;
double totalDistanceUB = 0;
for(int i = 0; i < nArcsNecessary; i++){
    totalDistanceUB += sortedDistances[i];
    totalDistanceLB += sortedDistances[nArcs-i-1];
}
System.out.println("noOfVisitsWithoutRegularWorkerAtWork_LB = " + noOfVisitsWithoutRegularWorkerAtWork_LB);
System.out.println("noOfVisitsWithRegularWorkerAtWork = " +noOfVisitsWithRegularWorkerAtWork);

System.out.println("totalDistanceLB = " + totalDistanceLB);
System.out.println("totalDistanceUB = " + totalDistanceUB);

System.out.println("nDistances = " + nDistances);
System.out.println("sumDistances = " + sumDistances);
*/

// A CHECK FOR VIOLATION
double violationRoute = 0;
double violationVisits = 0;
double violationSharedVisits = 0;
for(int j = 1; j <= nRoutes; j++){
    Route currentRoute =  allRoutes[j];
    violationRoute += currentRoute.violation();
    for(int i = 0; i < currentRoute.length(); i++){
Visit currentVisit = (Visit) currentRoute.get(i);
violationVisits += currentVisit.violation();
if(currentVisit.isShared()){
    int theOtherVisitNumber = theOtherVisit[currentVisit.number()];
    Visit theOther = allVisits[theOtherVisitNumber];
    violationSharedVisits += currentVisit.start() - theOther.start();
```

```
}
    }
}

boolean feasible = false;

if(!stop & violationRoute == 0 & violationVisits == 0 & violationSharedVisits == 0){
    feasible = true;
}

initialSolution.setIsFeasible(feasible);

System.out.println("The solution is feasible = " + initialSolution.isFeasible());

return initialSolution;

    }


    // *************** COST OF SINGLE VISIT ***************************
    //
    // Is there a feasible position for the visit ?
    // If yes! where? In which route and in which position ?
    // What is the cost of this best position ?
    //
    // ****************************************************************
    private InsertionCostOneVisit findInsertionCostOneVisit(Visit visit){

Citizen citizen = (Citizen) visit.citizen();
int citizenNumber = citizen.number();

// If there is somewhere in a route in a position a feasible insertion.
// When a feasible insertion position is met, this indicator is set to true.
boolean feasibleInsertion = false;

// For comparision within all the routes
double bestRouteCost = 1000; // c_1(v,r^*,p_{r^*}^*)

// The cost of each route
double[] routeCost = new double[nRoutes+1]; // c_1(v,r,p_r^*)
int[] bestRoutePosition =  new int[nRoutes+1]; //p_r^*
int bestRouteNumber = 1 ; //r^*

// FOR EACH ROUTE
for(int r = 1; r <= nRoutes; r++){

    // Which route is currently observed?
    Route route = allRoutes[r];


    // Who drives on this route?
    Worker worker = route.worker();

    //Is she/he the regular worker for currentCitizen?
    int theta  = 0;
    if(citizen.isTheWorkerRegular(worker)){
theta = 1;
    }

    // The cost if not feasible to insert the visit
    routeCost[r] = 1000;

    // FOR EACH PLACEMENT/POSITION IN THE CURRENT ROUTE
    for(int p = 0; p <= route.length(); p++){

//>>> THE START TIME <<<<<<
double startVisit;

if(p == 0){
    startVisit = Math.max(visit.open(), worker.start());
}else{

    // The previous visit p-1
    Visit previousVisit = (Visit) route.get(p-1);
    Citizen previousCitizen = (Citizen) previousVisit.citizen();
    int previousCitizenNumber = previousCitizen.number();

    // New starting time
    double arrivalVisit = previousVisit.finish() + distance[previousCitizenNumber][citizenNumber];
    startVisit = Math.max(visit.open(), arrivalVisit );
}


//>>> INDICATION WHETHER THE POSITION IS FEASIBLE <<<<<<
```

```
boolean feasiblePosition = true;

if(p < route.length()){

    // Check lemma 1.1, page 256 in Solomon1987 to see if the insertion position is feasible
    if(startVisit <= visit.closed() & startVisit + visit.duration() <= worker.finish()){

// The immidiately next visit p
Visit nextVisit = (Visit) route.get(p);
Citizen nextCitizen = (Citizen) nextVisit.citizen();
int nextCitizenNumber = nextCitizen.number();

// Push forward time
double arrivalNextVisit = startVisit + visit.duration() + distance[citizenNumber][nextCitizenNumber];
double pushForward = Math.max(0, arrivalNextVisit - nextVisit.arrival() - nextVisit.waitingTime());

/* if(nIterations == 23 & visit.number() == 37 & r ==11 & p == 2){
System.out.println("\nstart\n " );
}*/

// Check the next positions with push forward starting with the next visit currently in position k
// The visit is not inserted yet, and therefore the next visit is the one situated at position p
feasiblePosition = isThePushForwardFeasible(pushForward, p, r,0,0);

    }
    else{ feasiblePosition = false;}


    }else{

    // Check lemma 1.1, page 256 in Solomon1987 to see if the insertion position is feasible
     if(startVisit <= visit.closed() &  startVisit + visit.duration() <= worker.finish()){
 feasiblePosition = true;
     }
     else{
 feasiblePosition = false;
     }
}

// >>>> THE TOTAL COST <<<<<<
// If the position was feasible
double positionCost = 1000; //c_1(v,r,p_r) the cost if not feasible

if(feasiblePosition) {

    feasibleInsertion = true;

    //>>>> THE DISTANCE COST <<<<<<<<<<
    double distanceCost = 0;

    if(p == 0){

// The immidiately next visit p
Visit nextVisit = (Visit) route.get(p);
Citizen nextCitizen = (Citizen) nextVisit.citizen();
int nextCitizenNumber = nextCitizen.number();

// The distance-cost (xtra travel time)
distanceCost = distance[citizenNumber][nextCitizenNumber];
    }
    if(p < route.length() && p > 0){

// The previous visit p-1
Visit previousVisit = (Visit) route.get(p-1);
Citizen previousCitizen = (Citizen) previousVisit.citizen();
int previousCitizenNumber = previousCitizen.number();

// The immidiately next visit p
Visit nextVisit = (Visit) route.get(p);
Citizen nextCitizen = (Citizen) nextVisit.citizen();
int nextCitizenNumber = nextCitizen.number();

// The distance-cost (xtra travel time)
double newDistance = distance[previousCitizenNumber][citizenNumber] +
    distance[citizenNumber][nextCitizenNumber];
double oldDistance = distance[previousCitizenNumber][nextCitizenNumber] ;
distanceCost = newDistance - oldDistance;
    }
    if(p == route.length()){

// The previous visit p-1
Visit previousVisit = (Visit) route.get(p-1);
Citizen previousCitizen = (Citizen) previousVisit.citizen();
int previousCitizenNumber = previousCitizen.number();
```

```
// The distance-cost (xtra travel time)
distanceCost = distance[previousCitizenNumber][citizenNumber] ;
    }

    positionCost = distanceCost + priceNotRegularWorker*(1-theta);
}

//System.out.println("positionCost = " +  positionCost + ", position = " + p);

if(positionCost < routeCost[r]){
    routeCost[r] = positionCost;
    bestRoutePosition[r] = p;
}
    }

    // The cost of this current route = lowest positionCost
    if(routeCost[r] < bestRouteCost){
bestRouteCost = routeCost[r];
bestRouteNumber = r;
    }
}

// The cost for this visit is the sum of the routes ' deviations from the best route.
//// System.out.println("\n THE COSTS");
double visitCost = 0;
for(int r = 1; r <= nRoutes; r++){
    if(r != bestRouteNumber){
visitCost = visitCost + routeCost[r] - bestRouteCost;
    }
}
// The average deviation
visitCost = visitCost/(nRoutes-1);

InsertionCostOneVisit cost = new InsertionCostOneVisit(bestRouteNumber, bestRoutePosition[bestRouteNumber],
       visitCost, feasibleInsertion);

return cost;
    }


    // ******************* COST OF A VISIT PAIR  ********************************************
    //
    // If there are two workers at the same visit, the visit is split up into a pair of visits
    // The time window, citizen and duration are the same for the two visits.
    // Is there feasible positions for the a pair of visits.
    // If yes! where? In which routes and in which positions ?
    // What is the cost of these best positions ?
    //
    // ***************************************************************
    private InsertionCostTwoVisits findInsertionCostTwoEqualVisits(Visit visit){

// If there is somewhere in two route two position a feasible insertion.
// When a feasible insertion position is met, this indicator is set to true.
boolean feasibleInsertion = false;

// The two visits have the same citizen
Citizen citizen = (Citizen) visit.citizen();
int citizenNumber = citizen.number();

int visitNumber = visit.number();
int theOtherVisitNumber = theOtherVisit[visitNumber];
Visit theOtherVisit = allVisits[theOtherVisitNumber];

// The costs
double[][] twoRoutesCost = new double[nRoutes+1][nRoutes+1]; //c_1^*(v,r_1,r_2,p_{r_1}^*,p_{r_2}^*)
Positions[][] bestPositions = new Positions[nRoutes+1][nRoutes+1];
for(int route1number = 0; route1number < nRoutes+1; route1number++){
    for(int route2number = 0; route2number < nRoutes+1; route2number++){
Positions positions = new Positions();
bestPositions[route1number][route2number] = positions;
//positions.setPosition1(0);
//positions.setPosition2(0);
    }
}

int bestRoute1Number = 0; // r_1^*
int bestRoute2Number = 0; // r_2^*

// For comparison to find the lowest twoRoutesCost
double bestTwoRoutesCost = 1000; //c_1(v,r_1^*,r_2^*,p_{r_1^*}^*,p_{r_2^*}^*)

// FOR EACH ROUTE
for(int r1 = 1; r1 <= nRoutes-1; r1++){
```

```
    Route route1 = allRoutes[r1];

    // Who has this route 1
    Worker worker1 = route1.worker();

     //Is she/he the regular worker for currentCitizen?
    int theta1  = 0;
    if(citizen.isTheWorkerRegular(worker1)){
theta1 = 1;
    }

    // FOR THE OTHER ROUTES
    for(int r2 = r1+1; r2 <= nRoutes; r2++){

Route route2 = allRoutes[r2];

// Who has this route 2
Worker worker2 = route2.worker();

//Is she/he the regular worker for currentCitizen?
int theta2  = 0;
if(citizen.isTheWorkerRegular(worker2)){
    theta2 = 1;
}

// For comparison within all the positions
// The cost if not feasible to insert the visit
twoRoutesCost[r1][r2] = 1000; // c_1^*(v,r_1,r_2,p_{r_1}^*,p_{r_2}^*) = 1000

// The position in the first route
for(int p1 = 1; p1 <= route1.length(); p1++){

    // The position in the second route
    for(int p2 = 1; p2 <= route2.length() ; p2++){


// >>> THE ARRIVAL TIMES <<<<
double arrivalVisit1;
double arrivalVisit2;

if(p1 == 0){
    arrivalVisit1 = Math.max(visit.open(),worker1.start());
}else{

    // The previous visit
    Visit previousVisit1 = (Visit) route1.get(p1-1);
    Citizen previousCitizen1 = (Citizen) previousVisit1.citizen();
    int previousCitizenNumber1 = previousCitizen1.number();

    // The arrival time to currentVisit1
    arrivalVisit1 = previousVisit1.finish() + distance[previousCitizenNumber1][citizenNumber];
}

if(p2 == 0){
    arrivalVisit2 = Math.max(visit.open(),worker2.start());
}else{

    // The previous visit on route 2
    Visit previousVisit2 = (Visit) route2.get(p2-1);
    Citizen previousCitizen2 = (Citizen) previousVisit2.citizen();
    int previousCitizenNumber2 = previousCitizen2.number();

    // The arrival time to currentVisit2
     arrivalVisit2 = previousVisit2.finish() +
 distance[previousCitizenNumber2][citizenNumber];
}

// >> THE STARTING TIME <<<<<<<<<<<<
double latestArrivalVisit = Math.max(arrivalVisit1, arrivalVisit2);
double startVisit = Math.max(visit.open(), latestArrivalVisit);

// >>> INDICATION WHETHER THE POSITION IS FEASIBLE <<<<<<<<<<<<<<<
boolean feasiblePosition = true;

double finishVisit = startVisit + visit.duration();
double minWorkerFinish = Math.min(worker1.finish(), worker2.finish());

if(p1 < route1.length()){

    // Check lemma 1.1, page 256 in Solomon1987
    if(startVisit <= visit.closed() & finishVisit <= minWorkerFinish){

// The immidiately next visit on route 1
```

```
Visit nextVisit1 = (Visit) route1.get(p1);
Citizen nextCitizen1 = (Citizen) nextVisit1.citizen();
int nextCitizenNumber1 = nextCitizen1.number();

// Push forward time on route 1
double arrivalNextVisit1 = startVisit + visit.duration()+
    distance[citizenNumber][nextCitizenNumber1];
double pushForward1 = Math.max(0, arrivalNextVisit1 -
       nextVisit1.arrival() - nextVisit1.waitingTime());

// Is the push forward feasible on route 1
feasiblePosition = isThePushForwardFeasible(pushForward1, p1, r1, r2, p2);
    }
    else{
feasiblePosition = false;
    }

}else{

    // Check lemma 1.1, page 256 in Solomon1987
    if(startVisit > visit.closed() | finishVisit > minWorkerFinish){
feasiblePosition = false;
    }
}


// Do we want to check the other route as well??
// We do not need to check lemma 1.1, because
// if it feasible => startingTime < closingTime, because the two visits have same starting and closing times
if(feasiblePosition){
    // THE POSITION p2 IS NOT THE LAST ON THE ROUTE
    if(p2 < route2.length()){

// The immidiately next visit on route 2
Visit nextVisit2 = (Visit) route2.get(p2);
Citizen nextCitizen2 = (Citizen) nextVisit2.citizen();
int nextCitizenNumber2 = nextCitizen2.number();

// Push forward time on route 2
double arrivalNextVisit2 = startVisit + visit.duration()+
    distance[citizenNumber][nextCitizenNumber2];
double pushForward2 = Math.max(0, arrivalNextVisit2 -
       nextVisit2.arrival() - nextVisit2.waitingTime());


// Is the push forward feasible on route 2
feasiblePosition = isThePushForwardFeasible(pushForward2, p2, r2,r1,p1);

    }
}

// >>>> THE TOTAL COST <<<<<<
// If the position was feasible
double positionCost = 1000; // c_1(v,r_1,r_2,p_{r_1},p_{r_2} ) (if the insertion is not feasible)


if(feasiblePosition){

    feasibleInsertion = true;

    // >>> DISTANCE COST 1 <<<<<
    double distanceCost1 = 0;

    if(p1 == 0){

// The immidiately next visit on route 1
Visit nextVisit1 = (Visit) route1.get(p1);
Citizen nextCitizen1 = (Citizen) nextVisit1.citizen();
int nextCitizenNumber1 = nextCitizen1.number();

// The distance-cost (xtra travel time)
distanceCost1 = distance[citizenNumber][nextCitizenNumber1];;
    }
    if(p1 < route1.length() && p1 > 0){

// The previous visit
Visit previousVisit1 = (Visit) route1.get(p1-1);
Citizen previousCitizen1 = (Citizen) previousVisit1.citizen();
int previousCitizenNumber1 = previousCitizen1.number();

// The immidiately next visit on route 1
Visit nextVisit1 = (Visit) route1.get(p1);
Citizen nextCitizen1 = (Citizen) nextVisit1.citizen();
int nextCitizenNumber1 = nextCitizen1.number();
```

```
// The distance-cost (xtra travel time)
double newDistance1 = distance[previousCitizenNumber1][citizenNumber] +
    distance[citizenNumber][nextCitizenNumber1];
double oldDistance1 = distance[previousCitizenNumber1][nextCitizenNumber1] ;
distanceCost1 = newDistance1 - oldDistance1;
    }
    if(p1 == route1.length()){

// The previous visit
Visit previousVisit1 = (Visit) route1.get(p1-1);
Citizen previousCitizen1 = (Citizen) previousVisit1.citizen();
int previousCitizenNumber1 = previousCitizen1.number();

// The distance-cost (xtra travel time)
distanceCost1 = distance[previousCitizenNumber1][citizenNumber] ;
    }

    // >>> DISTANCE COST 2 <<<<<<<
    double distanceCost2 = 0;
    if(p2 == 0){

// The immidiately next visit on route 2
Visit nextVisit2 = (Visit) route2.get(p2);
Citizen nextCitizen2 = (Citizen) nextVisit2.citizen();
int nextCitizenNumber2 = nextCitizen2.number();

//The distance cost
distanceCost2 = distance[citizenNumber][nextCitizenNumber2];

    }
    if(p2 < route2.length() && p2 > 0){

// The previous visit on route 2
Visit previousVisit2 = (Visit) route2.get(p2-1);
Citizen previousCitizen2 = (Citizen) previousVisit2.citizen();
int previousCitizenNumber2 = previousCitizen2.number();

// The immidiately next visit on route 2
Visit nextVisit2 = (Visit) route2.get(p2);
Citizen nextCitizen2 = (Citizen) nextVisit2.citizen();
int nextCitizenNumber2 = nextCitizen2.number();

// The distance-cost (xtra travel time)
double newDistance2 = distance[previousCitizenNumber2][citizenNumber] +
    distance[citizenNumber][nextCitizenNumber2];
double oldDistance2 = distance[previousCitizenNumber2][nextCitizenNumber2] ;
distanceCost2 = newDistance2 - oldDistance2;

    }
    if(p2 == route2.length()){

// The previous visit on route 2
Visit previousVisit2 = (Visit) route2.get(p2-1);
Citizen previousCitizen2 = (Citizen) previousVisit2.citizen();
int previousCitizenNumber2 = previousCitizen2.number();

// The distance-cost (xtra travel time)
distanceCost2 = distance[previousCitizenNumber2][citizenNumber] ;
    }

    positionCost = (distanceCost1 + distanceCost2)/2 + priceNotRegularWorker*(1-theta1)*(1-theta2);
}

if(positionCost < twoRoutesCost[r1][r2]){


    twoRoutesCost[r1][r2] = positionCost;
    Positions positions = bestPositions[r1][r2];
    positions.setPosition1(p1);
    positions.setPosition2(p2);


}

    }// End position 2 loop


}// End position 1 loop

if(twoRoutesCost[r1][r2] < bestTwoRoutesCost){
    bestTwoRoutesCost = twoRoutesCost[r1][r2];
    bestRoute1Number = r1;
    bestRoute2Number = r2;
```

```
}


    } // End route 2 loop

} // End route 1 loop


double visitCost = 0;
for(int r1 = 0; r1 < nRoutes-1; r1++){
    for(int r2 = r1+1; r2 < nRoutes; r2++){

if((r1 != bestRoute1Number) || (r2 != bestRoute2Number)){
    visitCost = visitCost + twoRoutesCost[r1][r2] - bestTwoRoutesCost;
}
    }
}

// The number of combinations of route 1 and 2
// (equal to number of element in upper triangle without the diagonal)
int numberOfCombinations = nRoutes*(nRoutes-1)/2;
//// System.out.println("numberOfCombinations= " +  numberOfCombinations);
// The average deviation
visitCost = visitCost/(numberOfCombinations-1);

int bestPosition1 = bestPositions[bestRoute1Number][bestRoute2Number].position1();
int bestPosition2 =  bestPositions[bestRoute1Number][bestRoute2Number].position2();

InsertionCostTwoVisits cost;
cost  = new InsertionCostTwoVisits(bestRoute1Number, bestRoute2Number, bestPosition1,
   bestPosition2, visitCost, feasibleInsertion);

return cost;

    }


    // ************************************************************
    // Is it possible to push the visit in a certain position
    // and in a certain route a certain number of minutes forward ?
    //
    // ************************************************************
    private boolean isThePushForwardFeasible(double currentPushForward, int currentPosition, int currentRouteNumber,
     int forbiddenRouteNumber, int forbiddenPosition){

Route currentRoute = allRoutes[currentRouteNumber];

Worker worker = currentRoute.worker();

boolean feasible = true;
Visit currentVisit = (Visit) currentRoute.get(currentPosition);


if(currentRouteNumber == forbiddenRouteNumber & currentPosition <= forbiddenPosition){
    feasible = false;
}

if(feasible & currentPushForward > 0){

    double newStart = currentVisit.start() + currentPushForward;

    // Check current visit
    if(newStart > currentVisit.closed() | newStart + currentVisit.duration() > worker.finish() ){
feasible = false;
    }


    // Check the other part of the visit, if it is shared
    if(currentVisit.isShared() & feasible){

int currentVisitNumber = currentVisit.number();
int theOtherVisitNumber = theOtherVisit[currentVisitNumber];
Visit theOtherVisit = allVisits[theOtherVisitNumber];
int theOtherRouteNumber = theOtherVisit.routeNumber();
Route theOtherRoute = allRoutes[theOtherRouteNumber];
Worker theOtherWorker = theOtherRoute.worker();
int theOtherPosition = theOtherVisit.position();
double theOtherNewStart = theOtherVisit.start() + currentPushForward;

if( theOtherNewStart > theOtherVisit.closed() |
    theOtherNewStart + currentVisit.duration() > theOtherWorker.finish()){
    feasible = false;
}
```

```
if(theOtherPosition < theOtherRoute.length()-1 & feasible){
    int nextPositionTheOtherRoute = theOtherPosition +1;
    Visit nextVisitTheOtherRoute = (Visit) theOtherRoute.get(theOtherPosition +1);
    double nextPushForwardTheOtherRoute = Math.max(0, currentPushForward - nextVisitTheOtherRoute.waitingTime());
    feasible = isThePushForwardFeasible(nextPushForwardTheOtherRoute, nextPositionTheOtherRoute,
theOtherRouteNumber,forbiddenRouteNumber,forbiddenPosition);
}

    }

    // Check the succeeding visits in the current route
    if(feasible & (currentPosition < currentRoute.length()-1)){
int nextPosition = currentPosition + 1;
Visit nextVisit = (Visit) currentRoute.get(nextPosition);
double nextPushForward = Math.max(0, currentPushForward - nextVisit.waitingTime());
feasible = isThePushForwardFeasible(nextPushForward, nextPosition, currentRouteNumber,
    forbiddenRouteNumber,forbiddenPosition);

    }
}

return feasible;
    }


    private void insertOneVisit(Visit visit, int position, int routeNumber){

Citizen citizen = (Citizen) visit.citizen();
int citizenNumber = citizen.number();
Route route = allRoutes[routeNumber];

// >>> THE NUMBER OF VISITS WITHOUT REGULAR CARETAKER
Worker worker = route.worker();
if(!citizen.isTheWorkerRegular(worker)){
    noOfVisitsWithoutRegularWorker++;
}

// >>> THE ARRIVAL AND STARTING TIMES <<<<<
double arrivalVisit;
double startVisit;

if(position == 0){

    // Find the properties
    arrivalVisit = Math.max(visit.open(),worker.start());
    startVisit = arrivalVisit;
}
else{

    // The previous visit
    Visit previousVisit = (Visit) route.get(position-1);
    Citizen previousCitizen = (Citizen) previousVisit.citizen();
    int previousCitizenNumber = previousCitizen.number();

    // Find the properties
    arrivalVisit = previousVisit.finish() + distance[previousCitizenNumber][citizenNumber];
    startVisit = Math.max(visit.open(), arrivalVisit);
}

// >>>> SET THE PROPERTIES <<<<<
visit.setArrival(arrivalVisit);
visit.setStart(startVisit);
visit.setWaitingTime(visit.start() - visit.arrival());
visit.setFinish(startVisit + visit.duration());

// >>>> INSERT THE VISIT <<<<
route.insert(position,visit);
allNotPlannedVisits.remove(visit); // Not good, goes trough all elements to find visit
allPlannedVisits.add(visit);

// >>> SET THE ROUTE NUMBER AND POSITION
visit.setRouteNumber(routeNumber);
visit.setPosition(position);
visit.setIsPlanned(true);

// >>>> PUSH FORWARD <<<<<<<
if(position < route.length()-1){

    // The next visit
    Visit nextVisit = (Visit) route.get(position+1);
    Citizen nextCitizen = (Citizen) nextVisit.citizen();
    int nextCitizenNumber = nextCitizen.number();

    // How much should the next visit be pushed forward
```

```
    double arrivalNextVisit = visit.finish() + distance[citizenNumber][nextCitizenNumber];
    double pushForward = Math.max(0, arrivalNextVisit - nextVisit.arrival() - nextVisit.waitingTime());

    // Change the arrival properties for the next visit
    nextVisit.setArrival(arrivalNextVisit);

    // Go further. We are sure not to be at the end
    pushTheSucceedingVisitsForward(pushForward, position + 1, route);

}

// >>>> ADDITION OF TRAVEL TIME <<<<<<<<<<<<

if(position == 0){

    // The next visit
    Visit nextVisit = (Visit) route.get(position+1);
    Citizen nextCitizen = (Citizen) nextVisit.citizen();
    int nextCitizenNumber = nextCitizen.number();

    // The addition of travel time
    totalTravelTime += distance[citizenNumber][nextCitizenNumber];
}
if(position < route.length()-1 && position > 0){

    // The previous visit
    Visit previousVisit = (Visit) route.get(position-1);
    Citizen previousCitizen = (Citizen) previousVisit.citizen();
    int previousCitizenNumber = previousCitizen.number();

    // The next visit
    Visit nextVisit = (Visit) route.get(position+1);
    Citizen nextCitizen = (Citizen) nextVisit.citizen();
    int nextCitizenNumber = nextCitizen.number();

    // The addition of travel time
    double oldDistance = distance[previousCitizenNumber][nextCitizenNumber];
    double newDistance = distance[previousCitizenNumber][citizenNumber] +
distance[citizenNumber][nextCitizenNumber];
    totalTravelTime += newDistance - oldDistance;
}
if(position == route.length()-1){

    // The previous visit
    Visit previousVisit = (Visit) route.get(position-1);
    Citizen previousCitizen = (Citizen) previousVisit.citizen();
    int previousCitizenNumber = previousCitizen.number();

    // The addition of travel time
    totalTravelTime += distance[previousCitizenNumber][citizenNumber];
}
    }


    private void insertTwoEqualVisits(Visit visit1, Visit visit2, int position1, int position2,
      int routeNumber1, int routeNumber2){

// The citizen is the same for both visits
Citizen citizen = (Citizen) visit1.citizen();
int citizenNumber = citizen.number();

// The routes 1 and 2
Route route1 = allRoutes[routeNumber1];
Route route2 = allRoutes[routeNumber2];

Worker worker1 = route1.worker();
Worker worker2 = route2.worker();

// >>> THE NUMBER OF VISITS WITHOUT REGULAR CARETAKER <<<<<<
if(!citizen.isTheWorkerRegular(worker1) & !citizen.isTheWorkerRegular(worker2)){
    noOfVisitsWithoutRegularWorker += 1;
}

// >>> THE ARRIVAL AND STARTING TIMES <<<<<
double arrivalVisit1;
double arrivalVisit2;

if(position1 == 0){

    arrivalVisit1 = Math.max(visit1.open(), worker1.start());

}else{

    // The previous visit on route 1
```

```
    Visit previousVisit1 = (Visit) route1.get(position1-1);
    Citizen previousCitizen1 = (Citizen) previousVisit1.citizen();
    int previousCitizenNumber1 = previousCitizen1.number();

    arrivalVisit1 = previousVisit1.finish() + distance[previousCitizenNumber1][citizenNumber];
}
if(position2 == 0){

    arrivalVisit2 = Math.max(visit2.open(),worker2.start());

}else{

    // The previous visit on route 2
    Visit previousVisit2 = (Visit) route2.get(position2-1);
    Citizen previousCitizen2 = (Citizen) previousVisit2.citizen();
    int previousCitizenNumber2 = previousCitizen2.number();

    arrivalVisit2 = previousVisit2.finish() + distance[previousCitizenNumber2][citizenNumber];
}

double latestArrivalVisit = Math.max(arrivalVisit1, arrivalVisit2);
double startVisit = Math.max(visit1.open(), latestArrivalVisit);

// >>> SET THE PROPERTIES <<<<<
visit1.setArrival(arrivalVisit1);
visit2.setArrival(arrivalVisit2);
visit1.setStart(startVisit);
visit2.setStart(startVisit);
visit1.setWaitingTime(startVisit - arrivalVisit1);
visit2.setWaitingTime(startVisit - arrivalVisit2);
visit1.setFinish(startVisit + visit1.duration());
visit2.setFinish(startVisit + visit2.duration());

// >>> INSERT IN THE ROUTES <<<<
route1.insert(position1,visit1);
route2.insert(position2,visit2);
allNotPlannedVisits.remove(visit1); // Not good, goes trough all elements
allNotPlannedVisits.remove(visit2); // Not good, goes trough all elements
allPlannedVisits.add(visit1);
allPlannedVisits.add(visit2);

// >>> SET THE ROUTE NUMBER AND POSITION <<<
visit1.setRouteNumber(routeNumber1);
visit2.setRouteNumber(routeNumber2);
visit1.setPosition(position1);
visit2.setPosition(position2);
visit1.setIsPlanned(true);
visit2.setIsPlanned(true);

// >>> PUSH FORWARD <<<<<<<

if(nIterations == 36){
    System.out.println("hej");
}


if(position1 < route1.length()-1){

    // The next visit
    Visit nextVisit1 = (Visit) route1.get(position1+1);
    Citizen nextCitizen1 = (Citizen) nextVisit1.citizen();
    int nextCitizenNumber1 = nextCitizen1.number();

    // How much should the next visits be pushed forward
    double arrivalTimeNextVisit1 = visit1.finish() + distance[citizenNumber][nextCitizenNumber1];
    //double newStartNextVisit1 = Math.max(nextVisit1.open(), arrivalTimeNextVisit1);
    //double pushForward1 = newStartNextVisit1 - nextVisit1.start();

    //double arrivalNextVisit = visit.finish() + distance[citizenNumber][nextCitizenNumber];
double pushForward1 = Math.max(0, arrivalTimeNextVisit1 - nextVisit1.arrival() -
       nextVisit1.waitingTime());

    // Change the arrival properties for nextVisit1 and nextVisit2
    nextVisit1.setArrival(arrivalTimeNextVisit1);

    // Go further. We are sure not be at the end at route 1 nor route 2
    pushTheSucceedingVisitsForward(pushForward1, position1+1, route1);

}


if(position2 < route2.length()-1){

    // The next visit
```

```
    Visit nextVisit2 = (Visit) route2.get(position2+1);
    Citizen nextCitizen2 = (Citizen) nextVisit2.citizen();
    int nextCitizenNumber2 = nextCitizen2.number();

    // How much should the next visits be pushed forward
    double arrivalTimeNextVisit2 = visit2.finish() + distance[citizenNumber][nextCitizenNumber2];
    // double newStartNextVisit2 = Math.max(nextVisit2.open(), arrivalTimeNextVisit2);
    //double pushForward2 = newStartNextVisit2 - nextVisit2.start();

    //double arrivalNextVisit = visit.finish() + distance[citizenNumber][nextCitizenNumber];
double pushForward2 = Math.max(0, arrivalTimeNextVisit2 - nextVisit2.arrival() - nextVisit2.waitingTime());

    // Change the arrival properties for nextVisit1 and nextVisit2
    nextVisit2.setArrival(arrivalTimeNextVisit2);

    // Go further. We are sure not be at the end at route 1 nor route 2
    pushTheSucceedingVisitsForward(pushForward2, position2+1, route2);

}


// >>>> THE ADDITION OF TRAVELLING TIME ON ROUTE 1 <<<<<<

if(position1 == 0){

    // The next visit
    Visit nextVisit1 = (Visit) route1.get(position1+1);
    Citizen nextCitizen1 = (Citizen) nextVisit1.citizen();
    int nextCitizenNumber1 = nextCitizen1.number();

    totalTravelTime += distance[citizenNumber][nextCitizenNumber1];
}
if(position1 < route1.length()-1 && position1 > 0){

    // The previous visit
    Visit previousVisit1 = (Visit) route1.get(position1-1);
    Citizen previousCitizen1 = (Citizen) previousVisit1.citizen();
    int previousCitizenNumber1 = previousCitizen1.number();

    // The next visit
    Visit nextVisit1 = (Visit) route1.get(position1+1);
    Citizen nextCitizen1 = (Citizen) nextVisit1.citizen();
    int nextCitizenNumber1 = nextCitizen1.number();

    // The addition of travel time
    double oldDistance1 = distance[previousCitizenNumber1][nextCitizenNumber1];
    double newDistance1 = distance[previousCitizenNumber1][citizenNumber] +
distance[citizenNumber][nextCitizenNumber1];
    totalTravelTime += newDistance1  - oldDistance1 ;
}
if(position1 == route1.length()-1){

    // The previous visit
    Visit previousVisit1 = (Visit) route1.get(position1-1);
    Citizen previousCitizen1 = (Citizen) previousVisit1.citizen();
    int previousCitizenNumber1 = previousCitizen1.number();

    // The addition of travel time
    totalTravelTime += distance[previousCitizenNumber1][citizenNumber];
}

// >>> THE ADDITION OF TRAVELLING TIME ON ROUTE 2

if(position2 == 0){

    // The next visit
    Visit nextVisit2 = (Visit) route2.get(position2+1);
    Citizen nextCitizen2 = (Citizen) nextVisit2.citizen();
    int nextCitizenNumber2 = nextCitizen2.number();

    totalTravelTime += distance[citizenNumber][nextCitizenNumber2];
}
if(position2 < route2.length()-1 && position2 > 0){

    // The previous visit
    Visit previousVisit2 = (Visit) route2.get(position2-1);
    Citizen previousCitizen2 = (Citizen) previousVisit2.citizen();
    int previousCitizenNumber2 = previousCitizen2.number();

    // The next visit
    Visit nextVisit2 = (Visit) route2.get(position2+1);
    Citizen nextCitizen2 = (Citizen) nextVisit2.citizen();
    int nextCitizenNumber2 = nextCitizen2.number();
```

```
    // The addition of travel time
    double oldDistance2 = distance[previousCitizenNumber2][nextCitizenNumber2];
    double newDistance2 = distance[previousCitizenNumber2][citizenNumber] +
distance[citizenNumber][nextCitizenNumber2];
    totalTravelTime += newDistance2 - oldDistance2 ;
}
if(position2 == route2.length()-1){

    // The previous visit
    Visit previousVisit2 = (Visit) route2.get(position2-1);
    Citizen previousCitizen2 = (Citizen) previousVisit2.citizen();
    int previousCitizenNumber2 = previousCitizen2.number();

    // The addition of travel time
    totalTravelTime +=  distance[previousCitizenNumber2][citizenNumber];
}



    }


    // ************************************************************
    // Push the visit in a certain position
    // and in a certain route a certain number of minutes forward.
    //
    // ************************************************************
    private void pushTheSucceedingVisitsForward(double currentPushForward,
int currentPosition, Route currentRoute){


Visit currentVisit = (Visit) currentRoute.get(currentPosition);

// Change the properties for the currentVisit
currentVisit.setStart(currentVisit.start() + currentPushForward);
currentVisit.setWaitingTime(currentVisit.start() - currentVisit.arrival());
currentVisit.setFinish(currentVisit.finish() + currentPushForward);

// Change the position for the currentVisit
currentVisit.setPosition(currentPosition);

// Go further
if(currentPosition< currentRoute.length()-1){
    Visit nextVisit = (Visit) currentRoute.get(currentPosition+1);
double nextPushForward = Math.max(0, currentPushForward - nextVisit.waitingTime());
double arrivalTimeNextVisit = nextVisit.arrival() + currentPushForward;
nextVisit.setArrival(arrivalTimeNextVisit);
pushTheSucceedingVisitsForward(nextPushForward, currentPosition+1, currentRoute);
}

// If the visit has another half (another person is doing the same visit)
if(currentVisit.isShared()){

    int theOtherVisitNumber = theOtherVisit[currentVisit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];
    int theOtherRouteNumber = theOtherVisit.routeNumber();
    Route theOtherRoute =  allRoutes[theOtherRouteNumber];
    int theOtherPosition = theOtherVisit.position();

    // Change the properties for theOtherVisit
    // The arrival time and the position do not change
    //theOtherVisit.setStart(theOtherVisit.start() + currentPushForward);
    double pushForwardTheOther = currentVisit.start()-theOtherVisit.start();
    theOtherVisit.setStart(theOtherVisit.start() + pushForwardTheOther);
    theOtherVisit.setWaitingTime(theOtherVisit.start() - theOtherVisit.arrival());
    theOtherVisit.setFinish(theOtherVisit.finish() + currentPushForward);

    // Go further. The last visit on a route has position route.length()-1
    if(theOtherPosition < theOtherRoute.length()-1){
Visit nextVisit2 = (Visit) theOtherRoute.get(theOtherPosition+1);
double nextPushForward2 =  Math.max(0, currentPushForward - nextVisit2.waitingTime());
double arrivalTimeNextVisit2 = nextVisit2.arrival() + currentPushForward;
nextVisit2.setArrival(arrivalTimeNextVisit2);
pushTheSucceedingVisitsForward(nextPushForward2, theOtherPosition+1, theOtherRoute);
    }
}


    }
}
```

B.2.2 InsertionCostOneVisit.java

```
public class InsertionCostOneVisit {


    // VARIABLES TO SET
    private int r; //the best route number
    private int p; //the best position (equal to the position of the visit before)
    private double c; //the cost of the visit
    private boolean feasible; //If there is a feasible position;

    public InsertionCostOneVisit(int ro, int po, double co, boolean f ){
r = ro; p = po; c= co; feasible = f;
    }

    public int bestRouteNumber(){return r ;}

    public int bestPosition(){return p ;}

    public double cost(){return c ;}

    public boolean isThereAFeasiblePosition(){return feasible;}
}
```

B.2.3 InsertionCostTwoVisits.java

```
public class InsertionCostTwoVisits {


    // VARIABLES TO SET
    private int r1; //one of the best route number
    private int r2; //the other one of the best route number
    private int p1; //the best position in route r1
    private int p2; //the best position in route r1
    private double c; //the cost of inserting the pair of visits
    private boolean feasible; //If there are two feasible positions;

    public InsertionCostTwoVisits(int ro1, int ro2, int po1, int po2, double co, boolean f ){
r1 = ro1; r2 = ro2; p1 = po1; p2 = po2; c= co; feasible = f;
    }

    public int bestRouteNumber1(){return r1 ;}
    public int bestRouteNumber2(){return r2 ;}

    public int bestPosition1(){return p1 ;}
    public int bestPosition2(){return p2 ;}

    public double cost(){return c ;}

    public boolean isThereAFeasiblePosition(){return feasible;}
}
```

B.2.4 Positions.java

```
import java.math.*;
import java.util.*;

public class Positions{

     private int position1;
    private int position2;

    public Positions(){
position1 = -1;
position2 = -1;
    }

    public Positions(int p1, int p2){
position1 = p1;
position2 = p2;
    }

    public int position1(){return position1;}
```

```
    public void setPosition1(int p1){ position1 = p1;}

    public int position2(){return position2;}
    public void setPosition2(int p2){ position2 = p2;}


    public Positions copy(){
Positions newPositions = new Positions();

newPositions.setPosition1(position1);
newPositions.setPosition2(position2);

return newPositions;
    }

    public String toString(){

String output = "";
output += "position1 =" + position1;
output += " position2 =" + position2;

return output;

    }
}
```

B.3 The Source Code for the Tabu Search

B.3.1 TabuSearch.java

```
import java.io.*;
import java.util.*;
import java.math.*;
import java.util.Random;;

/*

This local search takes all routes route and finds the best one
of the visits in route to put into a new random route.
When a visit i is removed from a route r, it forbidden for the next theta iterations
to put the visit back.
The cost of the solution depends on the visits in the solution.
How many times have each of the visits  been added to the route, in which they are situated.
In this way the function will tru to find unexplores solutions

*/

/*

There are two routes

-route1 where the visit v is removed
-route2 where the visit is inserted

There are 3 phases

- PHASE1 initial
- PHASE2 v is removed from route1
- PHASE3 v is inserted in route2

*/

class TabuSearch{

    // THE VARIABLES FROM THE DATA
    public int nCitizens; //The number of citizens
    public int nVisits; //The number of visits
    public int maxVisitNumber; // The largest number a visit can have
    public int nRoutes; //The number of routes
    public double[][] distance; //The distances in minutes between citizens
    public int[] theOtherVisit; // What is the number of the corresponding visit, if the visit is shared.

    // THE PRICES
    public double alpha; // price of violation of closing times
    public double beta; // price of difference in starting times
    public double gamma; // prive in violation of working hours
```

```
    public double psi; //price for a not regular caretaker

    // A TABU LIST
    public int theta; // The length of the tabu
    public int[][] previousIteration;

    // The aspiration criterion
    public Cost[][] bestCostForEachAttribute;

    // The number of times a attribute has been used
    public int[][] rho;

    // The sum of the rho values is initially set to zero
    int rhoSum = 0;

    // The parameter to control the intensity of the diversification
    double lambda;

    // The modification factor
    double delta;

    // Initialize the random number generator
    long seed = 1;
    public Random rand = new Random();
    //public Random rand = new Random(seed);

    // Fandt bedre løsning med seed = 1, alpha, beta,gamma = 1, lambda = 0, theta = 1, og delta = 1.5

    // A number of iterations
    int nIterations ;
    int maxIterations = 100;

    public Improvement4Fast(Data data){

nCitizens = data.nCitizens();
nVisits = data.nVisits();
maxVisitNumber = data.maxVisitNumber();
nRoutes = data.nRoutes();
distance = data.distance();
theOtherVisit = data.theOtherVisit();

    }

    public Solution start(Solution initialFeasibleSolution, double alpha, double beta, double gamma,
  double psi, int theta, double lambda, double delta){

//System.out.println("FAST");

System.out.println("ite\tA\tB\tG\ttrav\tnoReg\tC\tv\troute1\troute2\tpos\talpha\tbeta\tgamma");

// The last iteration is set to -theta
// The best solution for each attribute is set to be really bad!!
Cost initialBestCostForEachAttribute = new Cost();
initialBestCostForEachAttribute.setTotalTravelTime(Double.MAX_VALUE);
initialBestCostForEachAttribute.setNoOfVisitsWithoutRegularWorker(nVisits);
initialBestCostForEachAttribute.setDifferenceStartingTimesSharedVisits(Double.MAX_VALUE);
initialBestCostForEachAttribute.setViolationOfTimeWindows(Double.MAX_VALUE);
initialBestCostForEachAttribute.setViolationOfWorkingHours(Double.MAX_VALUE);
previousIteration = new int[maxVisitNumber+1][nRoutes+1];
bestCostForEachAttribute = new Cost[maxVisitNumber+1][nRoutes+1];
for(int i = 1; i <= maxVisitNumber; i++){
    for(int r = 1; r <= nRoutes; r++){
previousIteration[i][r] = -theta;
bestCostForEachAttribute[i][r] =  initialBestCostForEachAttribute;
    }
}

// Now overwrite the bestSolutionForEachAttribute for these attributes in initialFeasibleSolution
Route[] allRoutesInitialFeasibleSolution = initialFeasibleSolution.allRoutes();
for(int r = 1; r <= nRoutes; r++){
    Route route = allRoutesInitialFeasibleSolution[r];
    for(int i = 0; i < route.length(); i++){
Visit visit = (Visit) route.get(i);
bestCostForEachAttribute[visit.number()][r] = initialFeasibleSolution.cost();
    }
}

  // A counter of iterations
nIterations = 0;

// The currently best solution (s*)
Solution bestFeasibleSolution = new Solution();
bestFeasibleSolution = initialFeasibleSolution.copy();
double best_c = bestFeasibleSolution.c(psi);
```

```
//// System.out.println("best_c = " + best_c);

// The first solution (s)
Solution solutionPHASE1 = initialFeasibleSolution.copy();
Cost costPHASE1 = solutionPHASE1.cost();
double current_f = solutionPHASE1.f(psi,alpha,beta,gamma);
//// System.out.println("current_f = " + current_f);

// The number of times the attribute has been added to the solution
rho =  new int[maxVisitNumber+1][nRoutes+1];

// Choose a stop criterion
while(nIterations < maxIterations){
    nIterations++;

    //// System.out.println("\nnIterations = " + nIterations + "\n");

    // The current routes and visits
    Route[] allRoutesPHASE1 = solutionPHASE1.allRoutes();
    Visit[] allVisitsPHASE1 = solutionPHASE1.allVisits();

    costPHASE1 = solutionPHASE1.cost();

    // Indicates whether a new solution is found
    boolean foundNewSolution = false;

// The route number for the random route
int bestRoute1number = 0;

// The position in route 1
int bestPositionRoute1 = 0;

// The route number for the other part of the route, if the visit v is shared
int theOtherRouteNumber = 0;

// The best route number different from the random route number
int bestRoute2number = 0;

// The best position for the visit v in route 2
int bestPositionRoute2 = 0;

// The best visit number from the random route
int bestVisitNumber = 0;

// The temporary sum of rhos is used until a new solution is found
int tempRhoSum;

// The currently best cost in the neighbourhood
double bestNeighbourhoodCost;

// The best cost/solution found
Cost bestSolutionInNeighbourhoodCost = new Cost();

// The forbidden visit numbers (if the best is tabu)
int[] forbiddenVisitNumbers = new int[nVisits];
int nForbiddenVisits = 0;
    // and their forbidden routes
int[] forbiddenRoute2numbers = new int[nRoutes+1];
int nForbiddenRoutes2 = 0;

// The best position solution in the neighbourhood ( \bar{s})
// Solution bestSolutionInNeighbourhood = solutionPHASE1.copy();

    // Remember that the properties are still the ones for currentSolution

while(!foundNewSolution){

    // INITILIZE EVERY THING
    bestRoute1number = 0;
    bestPositionRoute1 = 0;
    theOtherRouteNumber = 0;
    bestRoute2number = 0;
    bestPositionRoute2 = 0;
    bestVisitNumber = 0;
    tempRhoSum = 0;
    bestNeighbourhoodCost = Double.MAX_VALUE;
    bestSolutionInNeighbourhoodCost = new Cost();


    for(int route1number = 1; route1number <= nRoutes; route1number++){

    //// System.out.println("randomRouteNumber = " + randomRouteNumber);
    Route route1 = allRoutesPHASE1[route1number];
```

```
    // For all the visits in the route
    for(int positionRoute1 = 0; positionRoute1 < route1.length(); positionRoute1++){

Visit currentVisit = (Visit) route1.get(positionRoute1);
int currentVisitNumber = currentVisit.number();
//// System.out.println("currentVisitNumber = " + currentVisitNumber);

if(currentVisit.isShared()){
    int theOtherVisitNumber =  theOtherVisit[currentVisitNumber];
    Visit theOtherVisit = allVisitsPHASE1[theOtherVisitNumber];
    theOtherRouteNumber = theOtherVisit.routeNumber();
}

// If the visit is removable
if(currentVisit.removable()){

    Cost costPHASE2 = costRemoveVisit(allRoutesPHASE1, allVisitsPHASE1, route1number, positionRoute1,costPHASE1);


    for(int route2number = 1; route2number <= nRoutes ; route2number++){


if((route2number != route1number) && (route2number != theOtherRouteNumber)){

    boolean theMoveIsForbidden = false;
    for(int v = 0; v < nForbiddenVisits; v++){
for(int r = 0; r < nForbiddenRoutes2; r++ ){

    if(forbiddenVisitNumbers[v] ==  currentVisitNumber & forbiddenRoute2numbers[r] ==route2number ){
theMoveIsForbidden = true;
    }

}

    }


    if(!theMoveIsForbidden){

// The route that we will try for insertion
Route route2 = allRoutesPHASE1[route2number];

// Try all the positions in that route
for(int positionRoute2 = 0; positionRoute2 <= route2.length(); positionRoute2++){

    // Insert the random visit in the random route
    Cost costPHASE3 = costInsertVisit(allRoutesPHASE1, allVisitsPHASE1, currentVisitNumber,
      route2number, positionRoute2,costPHASE2);

    // c(\bar{s})
    double c = costPHASE3.c(psi);


    // f(\bar{s}) = c(\bar{s}) + \alpha q(\bar{s}) + \beta d(\bar{s}) + \gamma w(\bar{s})
    double f = costPHASE3.f(psi,alpha,beta,gamma);
    ///// System.out.println("f = " + f);

    // The penalty to diversify the search p(\bar{s})
    // The attribute value rho[currentVisitNumber][r] is raised by one
    // The attribute value rho[currentVisitNumber][randomRouteNumber] is removed
    double p;
    if(f >= current_f){
tempRhoSum = rhoSum + rho[currentVisitNumber][route2number]+1 - rho[currentVisitNumber][route1number];
p = lambda*Math.sqrt(nVisits*nRoutes)*tempRhoSum;
    }else{
p = 0;}
    //// System.out.println("penalty = " + penalty);

    // f(\bar{s}) + p(\bar{s})
    double cost_f_plus_p =  f + p;

    //// System.out.println("cost_f_plus_p  = " + cost_f_plus_p  );

    // Is the cost of this position better??
    if( cost_f_plus_p < bestNeighbourhoodCost){

bestRoute1number = route1number;
bestRoute2number = route2number;
bestPositionRoute1 = positionRoute1;
bestPositionRoute2 = positionRoute2;
bestVisitNumber = currentVisitNumber;
bestNeighbourhoodCost = cost_f_plus_p;
bestSolutionInNeighbourhoodCost = costPHASE3;
//// System.out.println("bestNeighbourhoodCost = " + bestNeighbourhoodCost);
```

```
//// bestSolutionInNeighbourhood = solutionPHASE3.copy();

    }
}
}

    }
    }

}
    }
       }

    // If the new attribute IS NOT tabu
    if(nIterations - previousIteration[bestVisitNumber][bestRoute2number] > theta){
//System.out.println("bestVisitNumber = " + bestVisitNumber);
//System.out.println("bestRoute2number = " + bestRoute2number);
// try if the cost for this attribute is better the one registred in bestCostForEachAttribute
if(bestSolutionInNeighbourhoodCost.f(psi,alpha,beta,gamma) <
   bestCostForEachAttribute[bestVisitNumber][bestRoute2number].f(psi,alpha,beta,gamma)){
    bestCostForEachAttribute[bestVisitNumber][bestRoute2number] = bestSolutionInNeighbourhoodCost;
}
    foundNewSolution = true;

    }
    // The new attribute IS tabu!
    else{

if(bestSolutionInNeighbourhoodCost.f(psi,alpha,beta,gamma) <
   bestCostForEachAttribute[bestVisitNumber][bestRoute2number].f(psi,alpha,beta,gamma)){
    // System.out.println("Aspiration");
    bestCostForEachAttribute[bestVisitNumber][bestRoute2number] = bestSolutionInNeighbourhoodCost;
    foundNewSolution = true;
}
// If it is not better, we have to find the next best solution
else{
    foundNewSolution = false;
    forbiddenVisitNumbers[nForbiddenVisits] = bestVisitNumber;
    nForbiddenVisits++;
    forbiddenRoute2numbers[nForbiddenRoutes2] = bestRoute2number;
    nForbiddenRoutes2++;
    }
}

    }

    // Remove bestVisitNumber from bestRoute1 and insert it in bestRoute2
    Solution solutionPHASE2 = removeVisit(solutionPHASE1, bestRoute1number, bestPositionRoute1);
    Solution solutionPHASE3 = insertVisit(solutionPHASE2, bestVisitNumber, bestRoute2number, bestPositionRoute2);

    // The number of times this attribute has been added to the solution
    rho[bestVisitNumber][bestRoute2number]++;

    // The sum of the rhos is set to the temperary sum of rhos
    rhoSum += rho[bestVisitNumber][bestRoute2number] - rho[bestVisitNumber][bestRoute1number];

     //Change the previous iteration for the removed attribute
    // So that that it is forbidden to add this attribute the next theta iterations.
    previousIteration[bestVisitNumber][bestRoute1number] = nIterations;

    // Is the solution feasible
    if(solutionPHASE3.isFeasible() && solutionPHASE3.c(psi) < best_c){
bestFeasibleSolution = solutionPHASE3.copy(); // s* = \bar{s}
best_c = solutionPHASE3.c(psi) ; // c(s*) = c(\bar{s})
    }

    /*Route[]  allRoutesPHASE3 = solutionPHASE3.allRoutes();
    double violationRoute = 0;
    double violationVisits = 0;
    for(int j = 1; j <= nRoutes; j++){
Route currentRoute =  allRoutesPHASE3[j];
violationRoute += currentRoute.violation();
for(int i = 0; i < currentRoute.length(); i++){
    Visit currentVisit = (Visit) currentRoute.get(i);
    violationVisits += currentVisit.violation();
}
} */

    String str = nIterations+"";
    str += "\t"+solutionPHASE3.violationOfTimeWindows();
    str += "\t"+solutionPHASE3.differenceStartingTimesSharedVisits();
    str += "\t"+solutionPHASE3.violationOfWorkingHours();
    /* str += "\t"+solutionPHASE3.totalTravelTime();
    str += "\t"+solutionPHASE3.noOfVisitsWithoutRegularWorker();
```

```
    str += "\t"+solutionPHASE3.c(psi);
    str += "\t"+bestVisitNumber;
    str += "\t"+bestRoute1number;
    str += "\t"+bestRoute2number;
    str += "\t"+bestPositionRoute2;
    str += "\t"+alpha;
    str += "\t"+beta;
    str += "\t"+gamma;
    str += "\t"+bestCostForEachAttribute[bestVisitNumber][bestRoute2number].f(psi,alpha,beta,gamma);*/
    //str += ", violationRoute = " + violationRoute;
    //str += ", violationVisits = " + violationVisits;

    //str += "\t"+solutionPHASE3.f(psi,alpha,beta,gamma);
    //str += "\t\t"+(short)alpha+"\t\t"+(short)beta+"\t\t"+(short)gamma;

    System.out.println(str);

    // Update the prices, update alpha
    if(solutionPHASE3.violationOfTimeWindows() == 0 ){
alpha = alpha/(1+delta);
    }else{
alpha = alpha*(1+delta);
    }

    //// System.out.println("alpha = " + alpha);

    // Update beta
    if(solutionPHASE3.differenceStartingTimesSharedVisits() == 0){
beta = beta/(1+delta);
    }else{
beta = beta*(1+delta);
    }

    ////  System.out.println("beta = " + beta);

    // Update gamma
    if(solutionPHASE3.violationOfWorkingHours() == 0){
gamma = gamma/(1+delta);
    }else{
gamma = gamma*(1+delta);
    }

    // The current solution is now the one found is the neighbourhood
    solutionPHASE1 = solutionPHASE3.copy(); // s = \bar{s}
    current_f = solutionPHASE1.f(psi,alpha,beta,gamma);
    //// System.out.println("current_f = " + current_f);

    }

return bestFeasibleSolution;
}



  // A function to CALCULATE the addition of cost when inserting one visit in position p in route r
    private Cost costInsertVisit(Route[] allRoutes, Visit[] allVisits, int visitNumber, int routeNumber,
 int p,  Cost c){

Cost cost = c.copy();

// The cost
double totalTravelTime = cost.totalTravelTime();
int noOfVisitsWithoutRegularWorker = cost.noOfVisitsWithoutRegularWorker();
double differenceStartingTimesSharedVisits = cost.differenceStartingTimesSharedVisits();
double violationOfClosingTimes = cost.violationOfTimeWindows();
double violationOfWorkingHours = cost.violationOfWorkingHours();

// The route
Route route = allRoutes[routeNumber];

// The visit
Visit visit = allVisits[visitNumber];

// The citizen at the visit
Citizen citizen = visit.citizen();

// The worker on the route
Worker worker = route.worker();

// CALCULATE THE NUMBER OF VISITS WITHOUT THE REGULAR WORKER

// If it is a shared visit
if (visit.isShared()){
```

```
    int theOtherVisitNumber = theOtherVisit[visit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];
    int theOtherRouteNumber = theOtherVisit.routeNumber();
    Route theOtherRoute = allRoutes[theOtherRouteNumber];
    Worker theOtherWorker = theOtherRoute.worker();

    // The citizen is the same (of cause!! )
    if(!citizen.isTheWorkerRegular(worker) & !citizen.isTheWorkerRegular(theOtherWorker)){
noOfVisitsWithoutRegularWorker++;
    }

}
// If it is not a shared visit
else{

    if(!citizen.isTheWorkerRegular(worker)){
noOfVisitsWithoutRegularWorker++;
    }
}

cost.setNoOfVisitsWithoutRegularWorker(noOfVisitsWithoutRegularWorker);

// >>>> THE ARRIVAL AND STARTING TIMES <<<<<<<<<
double start;
double arrival;

if(p == 0){
    arrival = Math.max(visit.open(),worker.start());
    start = arrival;
}
else{
    // The previous visit
    Visit previousVisit = (Visit) route.get(p-1);
    Citizen previousCitizen = previousVisit.citizen();
    arrival = previousVisit.finish() + distance[previousCitizen.number()][citizen.number()];
    start = Math.max(visit.open(), arrival);
}

// >>>> THE ADDITION OF TRAVEL TIME <<<<<<<<<
// The route will never be empty because of the seed-visits
// Remember that the visit is not inserted yet, therefore the next visit is at position p
if(p == 0){

    // The next visit
    Visit nextVisit  = (Visit) route.get(p);
    Citizen nextCitizen = nextVisit.citizen();

    totalTravelTime  +=  distance[citizen.number()][nextCitizen.number()];
    }
if(p > 0 & p < route.length()){

    // The previous visit
    Visit previousVisit = (Visit) route.get(p-1);
    Citizen previousCitizen = previousVisit.citizen();

    // The next visit
    Visit nextVisit  = (Visit) route.get(p);
    Citizen nextCitizen = nextVisit.citizen();

    // The distance-cost (xtra travel time)
double newDistance = distance[previousCitizen.number()][citizen.number()] +
    distance[citizen.number()][nextCitizen.number()];
double oldDistance = distance[previousCitizen.number()][nextCitizen.number()] ;
totalTravelTime += newDistance - oldDistance;
}
if(p == route.length()){

    // The previous visit
    Visit previousVisit = (Visit) route.get(p-1);
    Citizen previousCitizen = previousVisit.citizen();

    totalTravelTime  +=  distance[previousCitizen.number()][citizen.number()];
}

cost.setTotalTravelTime(totalTravelTime);


// HOW MUCH IS THE CURRENT VISIT OUTSIDE ITS TIME WINDOWS ?
double newViolationOfClosingTimes = Math.max(start - visit.closed(),0);
violationOfClosingTimes += newViolationOfClosingTimes;
cost.setViolationOfTimeWindows(violationOfClosingTimes);


// IS THERE A DIFFERENCE IN STARTING TIME IF IT IS A SHARED VISIT?
```

```
if(visit.isShared()){

    int theOtherVisitNumber = theOtherVisit[visit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];

    double newDifferenceStartingTimesSharedVisits = Math.abs(theOtherVisit.start() - start);
    differenceStartingTimesSharedVisits +=  newDifferenceStartingTimesSharedVisits;
    cost.setDifferenceStartingTimesSharedVisits(differenceStartingTimesSharedVisits);
}


// >>>>>> PUSH FORWARD THE NEXT VISIT <<<<<<<<<<<<<<<<
// The next visit is at position p
if(p < route.length()){

    // The next visit
    Visit nextVisit  = (Visit) route.get(p);
    Citizen nextCitizen = nextVisit.citizen();

    double arrivalNextVisit = start + visit.duration() +  distance[citizen.number()][nextCitizen.number()];
    double pushForward = Math.max(0, arrivalNextVisit - nextVisit.arrival() - nextVisit.waitingTime());

    cost = costPushForward(allRoutes,allVisits, pushForward, routeNumber, p, cost);
}else{

    Visit oldLastVisit = route.get(p-1);
    Visit newLastVisit = visit;

    // CALCULATE THE DEVIATIONS FROM WORKING TIME
    double oldViolationAfter = Math.max(oldLastVisit.finish() - route.worker().finish() ,0);
    double newViolationAfter = Math.max(start + visit.duration() - route.worker().finish() ,0);
    violationOfWorkingHours +=  newViolationAfter - oldViolationAfter;
    cost.setViolationOfWorkingHours(violationOfWorkingHours);
}

return cost;
    }

    // A function to CALCULATE the addition of cost when pushing the visits forward
    private Cost costPushForward(Route[] allRoutes,Visit[] allVisits, double pushForward,
 int routeNumber, int p, Cost c){

Cost cost = c.copy();

// The cost
double differenceStartingTimesSharedVisits = cost.differenceStartingTimesSharedVisits();
double violationOfClosingTimes = cost.violationOfTimeWindows();
double violationOfWorkingHours = cost.violationOfWorkingHours();

// The route
Route route = allRoutes[routeNumber];

// The current visit
Visit visit = (Visit) route.get(p);

// The new starting time
double start = visit.start() + pushForward;

// IS THERE A DIFFERENCE IN STARTING TIME IF IT IS A SHARED VISIT?
if(visit.isShared()){

    int theOtherVisitNumber = theOtherVisit[visit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];

    double oldDifferenceStartingTimesSharedVisits = Math.abs(theOtherVisit.start() - visit.start());
    double newDifferenceStartingTimesSharedVisits = Math.abs(theOtherVisit.start() - start);
    differenceStartingTimesSharedVisits += newDifferenceStartingTimesSharedVisits -
oldDifferenceStartingTimesSharedVisits;
    cost.setDifferenceStartingTimesSharedVisits(differenceStartingTimesSharedVisits);
}

// HOW MUCH IS THE CURRENT VISIT OUTSIDE ITS TIME WINDOWS ?

double oldViolationOfClosingTimes = Math.max(visit.start() - visit.closed(),0);
double newViolationOfClosingTimes = Math.max(start - visit.closed(),0);
violationOfClosingTimes += newViolationOfClosingTimes - oldViolationOfClosingTimes ;
cost.setViolationOfTimeWindows(violationOfClosingTimes);


// Go further
if(p < route.length()-1){

    Visit nextVisit = (Visit) route.get(p+1);
```

```
    double nextPushForward = Math.max(0, pushForward - nextVisit.waitingTime());
    cost = costPushForward(allRoutes,allVisits,pushForward, routeNumber, p +1,cost);
}
else{

    // CALCULATE THE DEVIATIONS FROM WORKING TIME
    double oldViolationAfter = Math.max(visit.finish() - route.worker().finish() ,0);
    double newViolationAfter = Math.max(start + visit.duration() - route.worker().finish() ,0);
    violationOfWorkingHours +=  newViolationAfter - oldViolationAfter;
    cost.setViolationOfWorkingHours(violationOfWorkingHours);
}

return cost;
    }

    // A function to CALCULATE the addition of cost when removing one visit in position p in route r
     private Cost costRemoveVisit(Route[] allRoutes, Visit[] allVisits, int routeNumber, int p, Cost c){

Cost cost = c.copy();

// The cost
double totalTravelTime = cost.totalTravelTime();
int noOfVisitsWithoutRegularWorker = cost.noOfVisitsWithoutRegularWorker();
double differenceStartingTimesSharedVisits = cost.differenceStartingTimesSharedVisits();
double violationOfWorkingHours = cost.violationOfWorkingHours();
double violationOfClosingTimes = cost.violationOfTimeWindows();

// The route
Route route = allRoutes[routeNumber];

// The visit
Visit visit = (Visit) route.get(p);

// The citizen at the visit
Citizen citizen = visit.citizen();

// The worker on the route
Worker worker = route.worker();

// CALCULATE THE NUMBER OF VISITS WITHOUT THE REGULAR WORKER
// If it is a shared visit
if (visit.isShared()){

    int theOtherVisitNumber = theOtherVisit[visit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];
    int theOtherRouteNumber = theOtherVisit.routeNumber();
    Route theOtherRoute = allRoutes[theOtherRouteNumber];
    Worker theOtherWorker = theOtherRoute.worker();

    // The citizen is the same (of cause!! )
    if(!citizen.isTheWorkerRegular(worker) & !citizen.isTheWorkerRegular(theOtherWorker)){
noOfVisitsWithoutRegularWorker--;
    }

}
// If it is not a shared visit
else{

    if(!citizen.isTheWorkerRegular(worker)){
noOfVisitsWithoutRegularWorker--;
    }
}

cost.setNoOfVisitsWithoutRegularWorker(noOfVisitsWithoutRegularWorker);

// >>>> SUBTRACTION OF TRAVEL TIME <<<<<<<<<
// If you want to remove a visit from an empty, then en travel time does not change
if(route.length() > 1){
    if(p == 0){

// The next visit
Visit nextVisit  = (Visit) route.get(p+1);
Citizen nextCitizen = nextVisit.citizen();

totalTravelTime  -=  distance[citizen.number()][nextCitizen.number()];
    }
    if(p > 0 & p < route.length()-1){

// The previous visit
Visit previousVisit = (Visit) route.get(p-1);
Citizen previousCitizen = previousVisit.citizen();

// The next visit
Visit nextVisit  = (Visit) route.get(p+1);
```

```
Citizen nextCitizen = nextVisit.citizen();

// The distance-cost (xtra travel time)
double newDistance1 = distance[previousCitizen.number()][citizen.number()] +
    distance[citizen.number()][nextCitizen.number()];
double oldDistance1 = distance[previousCitizen.number()][nextCitizen.number()] ;
totalTravelTime -= newDistance1 - oldDistance1;
    }
    if(p == route.length()-1){

// The previous visit
Visit previousVisit = (Visit) route.get(p-1);
Citizen previousCitizen = previousVisit.citizen();

totalTravelTime  -=  distance[previousCitizen.number()][citizen.number()];
    }
}

cost.setTotalTravelTime(totalTravelTime);

// THE CHANGE IN VIOLATION OF THE CLOSING TIMES
double oldViolationOfClosingTimes =  Math.max(visit.start() - visit.closed(),0);
violationOfClosingTimes -= oldViolationOfClosingTimes;
cost.setViolationOfTimeWindows(violationOfClosingTimes);


// THE DIFFERENCE IN STARTING TIMES
if(visit.isShared()){

    int theOtherVisitNumber = theOtherVisit[visit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];

    double oldDifferenceStartingTimesSharedVisits = Math.abs(theOtherVisit.start() - visit.start());
    differenceStartingTimesSharedVisits -=  oldDifferenceStartingTimesSharedVisits;
    cost.setDifferenceStartingTimesSharedVisits(differenceStartingTimesSharedVisits);
}

// THE REMOVAL OF THE VISIT
//route.removeVisitAt(p);

// THE SUCCEEDING VISITS WILL BE PUSHED BACKWARD.
// The next visit is at position p+1
if(p < route.length()-1){

    Visit nextVisit = (Visit) route.get(p+1);
    Citizen nextCitizen = nextVisit.citizen();
    double arrivalNextVisit;

    if(p == 0){
arrivalNextVisit= Math.max(nextVisit.open(),worker.start());
    }
    else{

// The previous vist
Visit previousVisit = (Visit) route.get(p-1);
Citizen previousCitizen = previousVisit.citizen();
arrivalNextVisit = previousVisit.finish() + distance[previousCitizen.number()][nextCitizen.number()];

    }

    double newStartNextVisit = Math.max(arrivalNextVisit,nextVisit.open());
    double pushBackward = nextVisit.start() - newStartNextVisit;


    cost = costPushBackward(allRoutes,allVisits, pushBackward, routeNumber, p+1,cost);
}else{

    Visit oldLastVisit = route.get(route.length()-1);
    Visit newLastVisit = route.get(route.length()-2);

    // CALCULATE THE DEVIATIONS FROM WORKING TIME
    double oldViolationAfter = Math.max(oldLastVisit.finish() - route.worker().finish() ,0);
    double newViolationAfter = Math.max(newLastVisit.finish() - route.worker().finish() ,0);

    violationOfWorkingHours +=  newViolationAfter - oldViolationAfter;
    cost.setViolationOfWorkingHours(violationOfWorkingHours);

}

return cost;

    }

    // A function to CALCULATE the addition of cost when pushing the visits back ward
```

```
     private Cost costPushBackward(Route[] allRoutes, Visit[] allVisits, double pushBackward,
   int routeNumber, int p, Cost c){

Cost cost = c.copy();

// The cost
double differenceStartingTimesSharedVisits = cost.differenceStartingTimesSharedVisits();
double violationOfClosingTimes = cost.violationOfTimeWindows();
double violationOfWorkingHours = cost.violationOfWorkingHours();

// The route
Route route = allRoutes[routeNumber];

// The current visit
Visit visit = (Visit) route.get(p);

// The new starting time
double start = Math.max(visit.arrival()-pushBackward, visit.open());

// IS THERE A DIFFERENCE IN STARTING TIME IF IT IS A SHARED VISIT?
if(visit.isShared()){

    int theOtherVisitNumber = theOtherVisit[visit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];

    double oldDifferenceStartingTimesSharedVisits = Math.abs(theOtherVisit.start() - visit.start());
    double newDifferenceStartingTimesSharedVisits = Math.abs(theOtherVisit.start() - start);
    differenceStartingTimesSharedVisits += newDifferenceStartingTimesSharedVisits -
oldDifferenceStartingTimesSharedVisits;
    cost.setDifferenceStartingTimesSharedVisits(differenceStartingTimesSharedVisits);
}

// HOW MUCH IS THE CURRENT VISIT OUTSIDE ITS TIME WINDOWS ?
double oldViolationOfClosingTimes = Math.max(visit.start() - visit.closed(),0);
double newViolationOfClosingTimes = Math.max(start - visit.closed(),0);
violationOfClosingTimes += newViolationOfClosingTimes - oldViolationOfClosingTimes ;
cost.setViolationOfTimeWindows(violationOfClosingTimes);

// Go further
if(p < route.length()-1){

    Visit nextVisit = (Visit) route.get(p+1);
    double arrivalTimeNextVisit = nextVisit.arrival() - pushBackward;
    cost = costPushBackward(allRoutes,allVisits,pushBackward, routeNumber, p +1,cost);
}
else{

    // CALCULATE THE DEVIATIONS FROM WORKING TIME
    double oldViolationAfter = Math.max(visit.finish() - route.worker().finish() ,0);
    double newViolationAfter = Math.max(start + visit.duration() - route.worker().finish() ,0);

    violationOfWorkingHours +=  newViolationAfter - oldViolationAfter;
    cost.setViolationOfWorkingHours(violationOfWorkingHours);
}

return cost;

    }



    //////// ACTUAL INSERTION!!!!!!!!!!!!

    // A function to CALCULATE the addition of cost when inserting one visit in position p in route r
    // This function actually inserts a visit in the route and returns a solution
    private Solution insertVisit(Solution solution, int visitNumber, int routeNumber, int p){

// The solution is copied into a new solution, because we do not want the solution changed
// Only the new solution will be operated on
Solution newSolution = solution.copy();

// All the routes and visits in the solution
Route[] allRoutes = newSolution.allRoutes();
Visit[] allVisits = newSolution.allVisits();

// The cost
double totalTravelTime = newSolution.totalTravelTime();
int noOfVisitsWithoutRegularWorker = newSolution.noOfVisitsWithoutRegularWorker();
double differenceStartingTimesSharedVisits = newSolution.differenceStartingTimesSharedVisits();
double violationOfClosingTimes = newSolution.violationOfTimeWindows();
double violationOfWorkingHours = newSolution.violationOfWorkingHours();

// The route
Route route = allRoutes[routeNumber];
```

```
// The visit
Visit visit = allVisits[visitNumber];

// The citizen at the visit
Citizen citizen = visit.citizen();

// The worker on the route
Worker worker = route.worker();

// CALCULATE THE NUMBER OF VISITS WITHOUT THE REGULAR WORKER

// If it is a shared visit
if (visit.isShared()){

    int theOtherVisitNumber = theOtherVisit[visit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];
    int theOtherRouteNumber = theOtherVisit.routeNumber();
    Route theOtherRoute = allRoutes[theOtherRouteNumber];
    Worker theOtherWorker = theOtherRoute.worker();

    // The citizen is the same (of cause!! )
    if(!citizen.isTheWorkerRegular(worker) & !citizen.isTheWorkerRegular(theOtherWorker)){
noOfVisitsWithoutRegularWorker++;
    }

}
// If it is not a shared visit
else{

    if(!citizen.isTheWorkerRegular(worker)){
noOfVisitsWithoutRegularWorker++;
    }
}

newSolution.setNoOfVisitsWithoutRegularWorker(noOfVisitsWithoutRegularWorker);

// >>>> THE ARRIVAL AND STARTING TIMES <<<<<<<<<
double start;
double arrival;

if(p == 0){
    arrival = Math.max(visit.open(),worker.start());
    start = arrival;
}
else{
    // The previous visit
    Visit previousVisit = (Visit) route.get(p-1);
    Citizen previousCitizen = previousVisit.citizen();
    arrival = previousVisit.finish() + distance[previousCitizen.number()][citizen.number()];
    start = Math.max(visit.open(), arrival);
}

// >>>> THE ADDITION OF TRAVEL TIME <<<<<<<<<
// The route will never be empty because of the seed-visits
// Remember that the visit is not inserted yet, therefore the next visit is at position p
if(p == 0){

    // The next visit
    Visit nextVisit  = (Visit) route.get(p);
    Citizen nextCitizen = nextVisit.citizen();

    totalTravelTime  +=  distance[citizen.number()][nextCitizen.number()];
    }
if(p > 0 & p < route.length()){

    // The previous visit
    Visit previousVisit = (Visit) route.get(p-1);
    Citizen previousCitizen = previousVisit.citizen();

    // The next visit
    Visit nextVisit  = (Visit) route.get(p);
    Citizen nextCitizen = nextVisit.citizen();

    // The distance-cost (xtra travel time)
double newDistance = distance[previousCitizen.number()][citizen.number()] +
    distance[citizen.number()][nextCitizen.number()];
double oldDistance = distance[previousCitizen.number()][nextCitizen.number()] ;
totalTravelTime += newDistance - oldDistance;
}
if(p == route.length()){

    // The previous visit
    Visit previousVisit = (Visit) route.get(p-1);
```

```
    Citizen previousCitizen = previousVisit.citizen();

    totalTravelTime  +=  distance[previousCitizen.number()][citizen.number()];
}

newSolution.setTotalTravelTime(totalTravelTime);


// HOW MUCH IS THE CURRENT VISIT OUTSIDE ITS TIME WINDOWS ?
double newViolationOfClosingTimes = Math.max(start - visit.closed(),0);
violationOfClosingTimes += newViolationOfClosingTimes;
newSolution.setViolationOfTimeWindows(violationOfClosingTimes);


// IS THERE A DIFFERENCE IN STARTING TIME IF IT IS A SHARED VISIT?
if(visit.isShared()){

    int theOtherVisitNumber = theOtherVisit[visit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];

    double newDifferenceStartingTimesSharedVisits = Math.abs(theOtherVisit.start() - start);
    differenceStartingTimesSharedVisits +=  newDifferenceStartingTimesSharedVisits;
    newSolution.setDifferenceStartingTimesSharedVisits(differenceStartingTimesSharedVisits);
}

// CALCULATE THE DEVIATIONS FROM WORKING TIME
// Is it possible to find the correct route???
Visit oldLastVisit = route.get(route.length()-1);
double oldViolationAfter = Math.max(oldLastVisit.finish() - worker.finish() ,0);

// >>>> SET THE PROPERTIES <<<<<
visit.setArrival(arrival);
visit.setStart(start);
visit.setWaitingTime(visit.start() - visit.arrival());
visit.setFinish(start + visit.duration());

// >>>> INSERT THE VISIT <<<<
route.insert(p,visit);


// Change the position and route number for the currentVisit
visit.setRouteNumber(routeNumber);
visit.setPosition(p);

// >>>>>> PUSH FORWARD THE NEXT VISIT <<<<<<<<<<<<<<<<

if(p < route.length()-1){

    // The next visit
    Visit nextVisit  = (Visit) route.get(p+1);
    Citizen nextCitizen = nextVisit.citizen();

    double arrivalNextVisit = start + visit.duration() +  distance[citizen.number()][nextCitizen.number()];
    double pushForward = Math.max(0, arrivalNextVisit - nextVisit.arrival() - nextVisit.waitingTime());
    nextVisit.setArrival(arrivalNextVisit);

    newSolution = pushForward(newSolution, pushForward, routeNumber, p+1);
}

// CALCULATE THE DEVIATIONS FROM WORKING TIME
// Is it possible to find the correct route???
//Visit newFirstVisit =  route.get(0);
Visit newLastVisit = route.get(route.length()-1);
double newViolationAfter = Math.max(newLastVisit.finish() - worker.finish() ,0);

violationOfWorkingHours +=  newViolationAfter - oldViolationAfter;
newSolution.setViolationOfWorkingHours(violationOfWorkingHours);

// Set the cost
//newSolution.setCost(newSolution.totalTravelTime() + psi*newSolution.noOfVisitsWithoutRegularWorker());

// Find out if the solution is feasible
if(newSolution.violationOfTimeWindows() == 0 && newSolution.violationOfWorkingHours() == 0 &&
   newSolution.differenceStartingTimesSharedVisits() == 0){
    newSolution.setIsFeasible(true);
}else{
    newSolution.setIsFeasible(false);
}

return newSolution;
    }

    // A function to CALCULATE the addition of cost when pushing the visits forward
    private Solution pushForward(Solution solution, double pushForward, int routeNumber, int p){
```

```
// All the routes and visits in the solution
Route[] allRoutes = solution.allRoutes();
Visit[] allVisits = solution.allVisits();

// The cost
double differenceStartingTimesSharedVisits = solution.differenceStartingTimesSharedVisits();
double violationOfClosingTimes = solution.violationOfTimeWindows();

// The route
Route route = allRoutes[routeNumber];

// The current visit
Visit visit = (Visit) route.get(p);

// The new starting time
double start = visit.start() + pushForward;

// IS THERE A DIFFERENCE IN STARTING TIME IF IT IS A SHARED VISIT?
if(visit.isShared()){

    int theOtherVisitNumber = theOtherVisit[visit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];

    double oldDifferenceStartingTimesSharedVisits = Math.abs(theOtherVisit.start() - visit.start());
    double newDifferenceStartingTimesSharedVisits = Math.abs(theOtherVisit.start() - start);
    differenceStartingTimesSharedVisits += newDifferenceStartingTimesSharedVisits -
oldDifferenceStartingTimesSharedVisits;
    solution.setDifferenceStartingTimesSharedVisits(differenceStartingTimesSharedVisits);
}

// HOW MUCH IS THE CURRENT VISIT OUTSIDE ITS TIME WINDOWS ?

double oldViolationOfClosingTimes = Math.max(visit.start() - visit.closed(),0);
double newViolationOfClosingTimes = Math.max(start - visit.closed(),0);
violationOfClosingTimes += newViolationOfClosingTimes - oldViolationOfClosingTimes ;
solution.setViolationOfTimeWindows(violationOfClosingTimes);


// Change the properties for the currentVisit
visit.setStart(start);
visit.setWaitingTime(start - visit.arrival());
visit.setFinish(visit.finish() + pushForward);

// Change the position for the currentVisit
visit.setPosition(p);
allVisits[visit.number()] = visit;

// Go further
if(p < route.length()-1){

    Visit nextVisit = (Visit) route.get(p+1);

    double nextPushForward = Math.max(0, pushForward - nextVisit.waitingTime());
    double arrivalTimeNextVisit = nextVisit.arrival() + pushForward;
    nextVisit.setArrival(arrivalTimeNextVisit);
    solution = pushForward(solution,pushForward, routeNumber, p +1);
}

return solution;
    }

    // A function to CALCULATE the addition of cost when removing one visit in position p in route r
    // and actually REMOVING IT
    private Solution removeVisit(Solution solution, int routeNumber, int p){

// There should not be done anything to solution
// Therefore the solution is copied into a new route, which is the one operated on
Solution newSolution = new Solution();
newSolution = solution.copy();

// All the routes and visits in the solution
Route[] allRoutes = newSolution.allRoutes();
Visit[] allVisits = newSolution.allVisits();

// The cost
double totalTravelTime = newSolution.totalTravelTime();
int noOfVisitsWithoutRegularWorker = newSolution.noOfVisitsWithoutRegularWorker();
double differenceStartingTimesSharedVisits = newSolution.differenceStartingTimesSharedVisits();
double violationOfWorkingHours = newSolution.violationOfWorkingHours();
double violationOfClosingTimes = newSolution.violationOfTimeWindows();

// The route
Route route = allRoutes[routeNumber];
```

```
// The visit
Visit visit = (Visit) route.get(p);

// The citizen at the visit
Citizen citizen = visit.citizen();

// The worker on the route
Worker worker = route.worker();

// CALCULATE THE NUMBER OF VISITS WITHOUT THE REGULAR WORKER
// If it is a shared visit
if (visit.isShared()){

    int theOtherVisitNumber = theOtherVisit[visit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];
    int theOtherRouteNumber = theOtherVisit.routeNumber();
    Route theOtherRoute = allRoutes[theOtherRouteNumber];
    Worker theOtherWorker = theOtherRoute.worker();

    // The citizen is the same (of cause!! )
    if(!citizen.isTheWorkerRegular(worker) & !citizen.isTheWorkerRegular(theOtherWorker)){
noOfVisitsWithoutRegularWorker--;
    }

}
// If it is not a shared visit
else{

    if(!citizen.isTheWorkerRegular(worker)){
noOfVisitsWithoutRegularWorker--;
    }
}

newSolution.setNoOfVisitsWithoutRegularWorker(noOfVisitsWithoutRegularWorker);

// >>>> SUBTRACTION OF TRAVEL TIME <<<<<<<<<
// If you want to remove a visit from an empty, then en travel time does not change
if(route.length() > 1){
    if(p == 0){

// The next visit
Visit nextVisit  = (Visit) route.get(p+1);
Citizen nextCitizen = nextVisit.citizen();

totalTravelTime  -=  distance[citizen.number()][nextCitizen.number()];
    }
    if(p > 0 & p < route.length()-1){

// The previous visit
Visit previousVisit = (Visit) route.get(p-1);
Citizen previousCitizen = previousVisit.citizen();

// The next visit
Visit nextVisit  = (Visit) route.get(p+1);
Citizen nextCitizen = nextVisit.citizen();

// The distance-cost (xtra travel time)
double newDistance1 = distance[previousCitizen.number()][citizen.number()] +
    distance[citizen.number()][nextCitizen.number()];
double oldDistance1 = distance[previousCitizen.number()][nextCitizen.number()] ;
totalTravelTime -= newDistance1 - oldDistance1;
    }
    if(p == route.length()-1){

// The previous visit
Visit previousVisit = (Visit) route.get(p-1);
Citizen previousCitizen = previousVisit.citizen();

totalTravelTime  -=  distance[previousCitizen.number()][citizen.number()];
    }
}

newSolution.setTotalTravelTime(totalTravelTime);

// THE CHANGE IN VIOLATION OF THE CLOSING TIMES
double oldViolationOfClosingTimes =  Math.max(visit.start() - visit.closed(),0);
violationOfClosingTimes -= oldViolationOfClosingTimes;
newSolution.setViolationOfTimeWindows(violationOfClosingTimes);


// THE DIFFERENCE IN STARTING TIMES
if(visit.isShared()){
```

```
    int theOtherVisitNumber = theOtherVisit[visit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];

    double oldDifferenceStartingTimesSharedVisits = Math.abs(theOtherVisit.start() - visit.start());
    differenceStartingTimesSharedVisits -=  oldDifferenceStartingTimesSharedVisits;
    newSolution.setDifferenceStartingTimesSharedVisits(differenceStartingTimesSharedVisits);
}


// THE VIOLATION OF WORKING HOURS
//Visit oldFirstVisit =  route.get(0);
Visit oldLastVisit = route.get(route.length()-1);
//double oldViolationBefore = Math.max(worker.start()-oldFirstVisit.start(),0);
double oldViolationAfter = Math.max(oldLastVisit.finish() - worker.finish() ,0);

// THE REMOVAL OF THE VISIT
route.removeVisitAt(p);

// THE SUCCEEDING VISITS WILL BE PUSHED BACKWARD. (p is now the next position !!!)
if(p < route.length()){

    Visit nextVisit = (Visit) route.get(p);
    Citizen nextCitizen = nextVisit.citizen();
    double arrivalNextVisit;

    if(p == 0){
arrivalNextVisit= Math.max(nextVisit.open(),worker.start());
    }
    else{

// The previous vist
Visit previousVisit = (Visit) route.get(p-1);
Citizen previousCitizen = previousVisit.citizen();
arrivalNextVisit = previousVisit.finish() + distance[previousCitizen.number()][nextCitizen.number()];

    }

    nextVisit.setArrival(arrivalNextVisit);

    double newStartNextVisit = Math.max(arrivalNextVisit,nextVisit.open());
    double pushBackward = nextVisit.start() - newStartNextVisit;

    newSolution = pushBackward(newSolution, pushBackward, routeNumber, p);
}


// CALCULATE THE DEVIATIONS FROM WORKING TIME
// Is it possible to find the correct route???
if(route.length() > 0){
    //Visit newFirstVisit =  route.get(0);
    Visit newLastVisit = route.get(route.length()-1);
    //double newViolationBefore = Math.max(worker.start()-newFirstVisit.start(),0);
    double newViolationAfter = Math.max(newLastVisit.finish() - worker.finish() ,0);
    violationOfWorkingHours +=   newViolationAfter -  oldViolationAfter;
    newSolution.setViolationOfWorkingHours(violationOfWorkingHours);
}
else{
    violationOfWorkingHours -=   oldViolationAfter;
    newSolution.setViolationOfWorkingHours(violationOfWorkingHours);
}

// Set the cost
//newSolution.setCost(newSolution.totalTravelTime() + psi*newSolution.noOfVisitsWithoutRegularWorker());

// Find out if the solution is feasible
if(newSolution.violationOfTimeWindows() == 0 && newSolution.violationOfWorkingHours() == 0 &&
   newSolution.differenceStartingTimesSharedVisits() == 0){
    newSolution.setIsFeasible(true);
}else{
    newSolution.setIsFeasible(false);
}

return newSolution;

    }

    // A function to CALCULATE the addition of cost when pushing the visits back ward
    // and actually PUSHING BACK WARD
    private Solution pushBackward(Solution solution, double pushBackward, int routeNumber, int p){

// All the routes and visits in the solution
Route[] allRoutes = solution.allRoutes();
Visit[] allVisits = solution.allVisits();
```

```
// The cost
double differenceStartingTimesSharedVisits = solution.differenceStartingTimesSharedVisits();
double violationOfClosingTimes = solution.violationOfTimeWindows();

// The route
Route route = allRoutes[routeNumber];

// The current visit
Visit visit = (Visit) route.get(p);

// The new starting time
double start = Math.max(visit.arrival(), visit.open());

// IS THERE A DIFFERENCE IN STARTING TIME IF IT IS A SHARED VISIT?
if(visit.isShared()){

    int theOtherVisitNumber = theOtherVisit[visit.number()];
    Visit theOtherVisit = allVisits[theOtherVisitNumber];

    double oldDifferenceStartingTimesSharedVisits = Math.abs(theOtherVisit.start() - visit.start());
    double newDifferenceStartingTimesSharedVisits = Math.abs(theOtherVisit.start() - start);
    differenceStartingTimesSharedVisits += newDifferenceStartingTimesSharedVisits -
oldDifferenceStartingTimesSharedVisits;
    solution.setDifferenceStartingTimesSharedVisits(differenceStartingTimesSharedVisits);
}

// HOW MUCH IS THE CURRENT VISIT OUTSIDE ITS TIME WINDOWS ?
double oldViolationOfClosingTimes = Math.max(visit.start() - visit.closed(),0);
double newViolationOfClosingTimes = Math.max(start - visit.closed(),0);
violationOfClosingTimes += newViolationOfClosingTimes - oldViolationOfClosingTimes ;
solution.setViolationOfTimeWindows(violationOfClosingTimes);


// Set the properties
visit.setStart(start);
visit.setWaitingTime(start - visit.arrival());
visit.setFinish(start + visit.duration());
visit.setPosition(p);
allVisits[visit.number()] = visit;

// Go further
if(p+1 < route.length()){

    Visit nextVisit = (Visit) route.get(p+1);
    double arrivalTimeNextVisit = nextVisit.arrival() - pushBackward;
    nextVisit.setArrival(arrivalTimeNextVisit);
    solution = pushBackward(solution,pushBackward, routeNumber, p +1);
}

return solution;

    }


}
```

B.4 The Source Code for Reading Data

```
import java.io.*;
import java.util.*;

class Data{

    public int nCitizens = 0;
    public int nVisits = 0;
    public int nWorkers = 0;
    public int nRoutes = 0;

    // The distance matrix between the ditsIDs
    public double[][] distanceDIST = new double[2000][2000];

    public double[][] distance;

    // All the visits
    public Visit[] allVisits = new Visit[400];

    // All the routes
```

```
    public Route[] allRoutes;

    // The largest visit number
    public int maxVisitNumber = 0;

    // All the workers
    public Worker[] allWorkers = new Worker[200];

    // All the citizens
    public Citizen[] allCitizens = new Citizen[200];

    // The number of the other part of a shared visit
    public int[] theOtherVisit = new int[400];

    // From an ID to a number
    public int[] id2workerNumber = new int[3000];

    //public int[] workerNumber2id = new int[200];

    // The number of initial breaks and start visits
    public int nInitialBreakVisits = 0;
    public int nInitialStartVisits = 0;

    // Number of removed visits for other causes
    public int nRemovedVisits = 0;

    // Number of non-existent visits
    public int nNonExistentVisits = 0;

    // Number of added shared visits
    public int nAddedSharedVisits = 0;

    // The number of breaks
    public int nBreakVisits =0;

    public int nStartVisits =0;

     // Constructor (input could be a string)
    public Data(String distFileName, String resourcesFileName, String visitFileName){

// Find allWorkers[] and nWorkers
// The workers are initialized
readResources(resourcesFileName);

// Find allCitizens[], nCitizens, allVisits[] and nVisits
// The citizens and the visits are initialized.
readVisits(visitFileName);

    // Find distance[][]
readDistances(distFileName);
    }

  public void addExtraWorkersAndAttachAllWorkersToRoutes(int nExtraWorkers){

      // More workers
      if(nExtraWorkers > 0){
  // A substitute worker with number nWorkers+1 which available from 450 to 930
  for(int i = 0; i <nExtraWorkers; i++){
      nWorkers++;
      Worker substitute = new Worker(nWorkers, 465, 930);
      allWorkers[nWorkers] = substitute;
  }
      }

      // Less workers
      if(nExtraWorkers < 0){
  nWorkers += nExtraWorkers;
      }

      //System.out.println("nWorkers = " + nWorkers);

// Attach the workers to the routes
nRoutes = nWorkers;
allRoutes = new Route[nRoutes+1];
for(int r = 1; r <= nRoutes; r++){
    Route currentRoute =new Route(r,allWorkers[r]);
    allRoutes[r] = currentRoute;
}

    }

    public void removeStartVisits(int nStartVisits, int firstStartVisitNumber){

nInitialStartVisits += nStartVisits;
```

```
// Remove the breaks
for(int b = 0;  b < nStartVisits; b++){
     Visit currentVisit = allVisits[firstStartVisitNumber + b];
     currentVisit.setRemovable(false);
     currentVisit.setIsPlanned(true);
     allVisits[firstStartVisitNumber + b] = currentVisit;
}

    }

    public void removeBreakVisits(int nBreakVisits, int firstBreakVisitNumber){

nInitialBreakVisits += nBreakVisits;

// Remove the breaks
for(int b = 0;  b < nBreakVisits; b++){
    Visit currentVisit = allVisits[firstBreakVisitNumber + b];
    currentVisit.setRemovable(false);
    currentVisit.setIsPlanned(true);
    allVisits[firstBreakVisitNumber + b] = currentVisit;
}

    }

    public void removeVisits(int[] visitNumbers){

nRemovedVisits += visitNumbers.length;

for(int i = 0; i < visitNumbers.length; i++){

    int currentVisitNumber = visitNumbers[i];
    Visit currentVisit = allVisits[currentVisitNumber];
    currentVisit.setRemovable(false);
    currentVisit.setIsPlanned(true);
    allVisits[currentVisitNumber] = currentVisit;

}
    }



    public void nonExistentVisits(int[] visitNumbers){

nNonExistentVisits += visitNumbers.length;

Visit dummyVisit = new Visit();
dummyVisit.setRemovable(false);
dummyVisit.setIsPlanned(true);

for(int i = 0; i < visitNumbers.length; i++){

    int currentVisitNumber = visitNumbers[i];
    allVisits[currentVisitNumber] = dummyVisit;
}
    }

    public void nonExistentVisit(int visitNumber){

nNonExistentVisits++;

Visit dummyVisit = new Visit();
dummyVisit.setRemovable(false);
dummyVisit.setIsPlanned(true);
allVisits[visitNumber] = dummyVisit;

    }


    public void addNewStartVisitsAndBreaksToTheRoutes(double durationBreakVisit){

// The start time for the start visit
double openStartVisit = 465;
double closedStartVisit = 465;
double durationStartVisit = 15;

// The start time for the break
double openBreakVisit = 720;
double closedBreakVisit = 720;

// A dummy worker
//Worker dummyWorker = new Worker(0, 0, 2000);

Citizen office = new Citizen();
```

```
// Find the office with the distance number 528
boolean foundOffice = false;
int j = 1;
while(!foundOffice & j <=nCitizens){

    Citizen currentCitizen = allCitizens[j];

    if(528 == currentCitizen.distanceNumber()){
office = currentCitizen;
foundOffice = true;
    }
    j++;
}

// The number of breaks
nBreakVisits = 0;

// The number of start Visits
nStartVisits = 0;

// New start visits and breaks are made, because the data set might not contain sufficient of these
for(int r = 1; r <= nRoutes; r++){

    // The current route
    Route currentRoute = allRoutes[r];

    // The current worker
    Worker worker = currentRoute.worker();

    // The current position in the route
    int currentPosition = 0;

    if(openStartVisit >= worker.start()){

nStartVisits++;

// Set the properties for the start visit
maxVisitNumber++;
Visit startVisit = new Visit(maxVisitNumber,openStartVisit,
     closedStartVisit,durationStartVisit,office);
startVisit.setArrival(openStartVisit);
startVisit.setStart(openStartVisit);
startVisit.setWaitingTime(0);
startVisit.setFinish(openStartVisit + durationStartVisit);
startVisit.setRemovable(false);
startVisit.setIsPlanned(true);

// Insert the start visit
currentRoute.insert(currentPosition,startVisit);
startVisit.setRouteNumber(r);
startVisit.setPosition(currentPosition);

// Insert the visit in the array of all visits
allVisits[maxVisitNumber] = startVisit;

currentPosition++;
    }

    if(openBreakVisit + durationBreakVisit <= worker.finish()){

// Raise the number of breaks by 1
nBreakVisits++;

// Set the properties for the break
maxVisitNumber++;
Visit breakVisit = new Visit(maxVisitNumber,openBreakVisit,
     closedBreakVisit,durationBreakVisit,office);
breakVisit.setArrival(openStartVisit + durationStartVisit);
breakVisit.setStart(openBreakVisit);
breakVisit.setWaitingTime(openBreakVisit - breakVisit.arrival());
breakVisit.setFinish(openBreakVisit + durationBreakVisit);
breakVisit.setRemovable(false);
breakVisit.setIsPlanned(true);

// Insert the break
currentRoute.insert(currentPosition,breakVisit);
breakVisit.setRouteNumber(r);
breakVisit.setPosition(currentPosition);

// Insert the visit in the array of all visits
allVisits[maxVisitNumber] = breakVisit;
    }
```

```
}

    }


    public void calculateTheNumberOfVisitsAndRemoveSuperFlousElements(){

// The last visits in the array "allVisits" are removed
// As well as the visits removed previously (e.g. breaks and start visits)
Visit[] allVisitsTemp = new Visit[maxVisitNumber+1];
nVisits = maxVisitNumber - nInitialStartVisits-nInitialBreakVisits - nRemovedVisits - nNonExistentVisits;

// Calculate the total number of visits
for(int i = 1; i <= maxVisitNumber; i++){

    Visit currentVisit = allVisits[i];
    //// System.out.println("currentVisit = " + currentVisit);
    allVisitsTemp[i] = currentVisit;
}

//System.out.println("nVisits = " + nVisits);

// Overwrite the old allVisits with the new
allVisits = new Visit[maxVisitNumber+1];
allVisits = allVisitsTemp;
    }


    public void makeSharedVisit(int visitNumber1, int visitNumber2){

Visit visit1 = allVisits[visitNumber1];
visit1.setIsShared(true);
Visit visit2 = allVisits[visitNumber2];
visit2.setIsShared(true);
theOtherVisit[visitNumber1] = visitNumber2;
theOtherVisit[visitNumber2] = visitNumber1;
    }

    public void addSharedVisit(int distIDCitizen, int workerID1, int workerID2, int workerID3,
       double open, double closed, double duration){

int worker1number = id2workerNumber[workerID1];
int worker2number = id2workerNumber[workerID2];
int worker3number = id2workerNumber[workerID3];

Citizen citizen = new Citizen();

// Find corresponding citizen
boolean foundCitizen = false;
int j = 1;
while(!foundCitizen & j <=nCitizens){

    Citizen currentCitizen = allCitizens[j];

    if(distIDCitizen == currentCitizen.distanceNumber()){
int currentWorker1number = currentCitizen.worker1().number();
int currentWorker2number = currentCitizen.worker2().number();
int currentWorker3number = currentCitizen.worker3().number();

if( worker1number == currentWorker1number & worker2number == currentWorker2number &
    worker3number == currentWorker3number){
    foundCitizen = true;
    citizen = currentCitizen;
}

    }
    j++;
}


nAddedSharedVisits++;
maxVisitNumber++;

Visit visit1 = new Visit(maxVisitNumber,open,closed,duration,citizen);
visit1.setIsShared(true);
allVisits[maxVisitNumber] = visit1;

nAddedSharedVisits++;
maxVisitNumber++;

Visit visit2 = new Visit(maxVisitNumber,open,closed,duration,citizen);
visit2.setIsShared(true);
allVisits[maxVisitNumber] = visit2;
```

```
theOtherVisit[visit1.number()] = visit2.number();
theOtherVisit[visit2.number()] = visit1.number();


}


    public void changeDistance(int citizenNumber1, int citizenNumber2, double newDistance){

distance[citizenNumber1][citizenNumber2] = newDistance;
    }


    public void changeTimeWindow(int visitNumber, double open, double closed){

Visit visit = allVisits[visitNumber];

visit.setOpen(open);
visit.setClosed(closed);

    }



    public int nCitizens(){
     return nCitizens;}

    public int nVisits(){
     return nVisits;}

    public int maxVisitNumber(){
return maxVisitNumber;}

    public int nWorkers(){
     return nWorkers;}

    public int nRoutes(){
     return nRoutes;}

    public int nBreakVisits(){
return nBreakVisits;
    }

 public int nStartVisits(){
return nStartVisits;
    }

    public double[][] distance(){
     return distance;}

    public Visit[] allVisits(){
     return allVisits;}

 public Route[] allRoutes(){
     return allRoutes;}

    public Worker[] allWorkers(){
     return allWorkers;}

    public int[] theOtherVisit(){
return theOtherVisit;}


    private void readDistances(String distFileName){

     // Open the file
InputFile input = new InputFile(distFileName);
input.skipFirstLine();

while(input.getTokenType() != StreamTokenizer.TT_EOF){

    int distID1 = (int) input.getTokenNumber();
    boolean flag1 = input.next();
    boolean flag2 = input.next();
    int distID2 = (int) input.getTokenNumber();
    boolean flag3 = input.next();
    boolean flag4 = input.next();
    double dist = input.getTokenNumber();

    //System.out.print("distID1 =  " + distID1 + " | ") ;
    //System.out.print("distID1 =  " + distID2 + " | ") ;
    //System.out.println("dist =  " + dist) ;

    //System.out.print("citizenNumber1 =  " + citizenNumber1 + " | ") ;
```

```
    //System.out.print("citizenNumber2 =  " + citizenNumber2 + " | ") ;
    //System.out.println("dist =  " + dist + "\n") ;

    // The distance is rounded down
    distanceDIST[distID1][distID2] = Math.floor(dist);

    // The flag for the next line
    boolean flagNextLine = input.next();
}


// Insert the distance in the distance matrix for the citizens
distance = new double [nCitizens+1][nCitizens+1];

for(int i = 1; i <= nCitizens; i++ ){

    Citizen citizen1 = allCitizens[i];
    int citizen1number = citizen1.number();
    int distID1 = citizen1.distanceNumber();

    for(int j = 1; j<= nCitizens; j++  ){

Citizen citizen2 = allCitizens[j];
int citizen2number = citizen2.number();
int distID2 = citizen2.distanceNumber();
//System.out.println("citizen1number = " + citizen1number);
//System.out.println("citizen2number = " + citizen2number);
//System.out.println("distID1 = " + distID1);
//System.out.println("distID2 = " + distID2);
//System.out.println(" = " + distanceDIST[distID1][distID2]);
distance[citizen1number][citizen2number] = distanceDIST[distID1][distID2];

    }

}
    }

    private void readResources(String resourcesFileName){

InputFile input = new InputFile(resourcesFileName);
input.skipFirstLine();

while(input.getTokenType() != StreamTokenizer.TT_EOF){

    int id = (int) input.getTokenNumber();
    boolean flag1 = input.next();
    boolean flag2 = input.next();
    double start =   input.getTokenNumber();
    boolean flag3 = input.next();
    boolean flag4 = input.next();
    double finish =  input.getTokenNumber();

    //System.out.print("distID =  " + distID + " | ") ;
    //System.out.print("start =  " + start + " | ") ;
    //System.out.print("finish =  " + finish + " \n");

    nWorkers++;

    id2workerNumber[id] = nWorkers;
    //workerNumber2id[nWorkers] = id;

    // Find all the workers
    Worker w = new Worker(nWorkers, start, finish,id);
    allWorkers[nWorkers] = w; // The number of workers start by 1

    // The flag for the next line
    boolean flagNextLine = input.next();

}

    }

    private void readVisits(String visitsFileName){

InputFile input = new InputFile(visitsFileName);
StreamTokenizer st = input.getStream();
input.skipFirstLine();

//allCitizens = new Citizen[nCitizens+1];

// A dummy worker (to be regular at the office)
Worker dummyWorker = new Worker(0, 0, 2000,-1);

while(input.getTokenType() != StreamTokenizer.TT_EOF){
```

```
    maxVisitNumber++;

    int visitNumber = (int) input.getTokenNumber();
    boolean flag1 = input.next();
    boolean flag2 = input.next();
    int idCitizen =  (int) input.getTokenNumber();
    boolean flag3 = input.next();
    boolean flag4 = input.next();
    int idWorker1 = (int) input.getTokenNumber();
    boolean flag5 = input.next();
    boolean flag6 = input.next();
    int idWorker2 = (int) input.getTokenNumber();
    boolean flag7 = input.next();
    boolean flag8 = input.next();
    int idWorker3 = (int) input.getTokenNumber();
    boolean flag9 = input.next();
    boolean flag10 = input.next();
    double start =  input.getTokenNumber();
    boolean flag11 = input.next();
    boolean flag12 = input.next();
    double finish = input.getTokenNumber();
    boolean flag13 = input.next();
    boolean flag14 = input.next();
    double duration = input.getTokenNumber();

    // The time window in the data set require the whole visit to be inside
    // Here we demand the starting time to be within the time window.
    // The latest starting time is the old time windows finish time minus the duration
    double newFinish = finish - duration;

    //System.out.print("visitNumber =  " + visitNumber + " | ") ;
    // System.out.print("distIDCtizen =  " + distIDCitizen + " | ") ;
    // System.out.print("distIDWorker1 =  " + distIDWorker1 + " | ") ;
    // System.out.print("distIDWorker2 =  " + distIDWorker2 + " | ") ;
    // System.out.print("distIDWorker3 =  " + distIDWorker3 + " | ") ;
    // System.out.print("start =  " + start + " | ") ;
    // System.out.print("finish =  " + finish + " | ");
    // System.out.print("duration =  " + duration + " \n ") ;


    // The citizen
    Citizen citizen = new Citizen();

    // Find corresponding citizen
    boolean foundCitizen = false;
    int j = 1;
    while(!foundCitizen & j <=nCitizens){

Citizen currentCitizen = allCitizens[j];

if(idCitizen == currentCitizen.distanceNumber()){
    int currentWorker1number = currentCitizen.worker1().originalNumber();
    int currentWorker2number = currentCitizen.worker2().originalNumber();
    int currentWorker3number = currentCitizen.worker3().originalNumber();

    if( idWorker1 == currentWorker1number & idWorker2 == currentWorker2number
& idWorker3 == currentWorker3number){
foundCitizen = true;
citizen = currentCitizen;
    }

}
j++;
    }

    // System.out.print("citizenNumber=  " + citizenNumber + " \n ") ;

    // if the citizen is not inserted yet in allCitizens
    if(!foundCitizen){

nCitizens++;

Worker worker1, worker2, worker3;

// The citizens are initialized

// There might not be a favorible worker: distIDWorker1 = -1
if(idWorker1 == -1 ){
    worker1 = dummyWorker;
}
else{
    int workerNumber1 = id2workerNumber[idWorker1];
```

```
    // The worker 1 might not be at work that day :  allWorkers[workerID1] = null
    if(allWorkers[workerNumber1] == null){worker1 = dummyWorker; }
    else{worker1 = allWorkers[workerNumber1]; }
}

if(idWorker2 == -1){
worker2 = dummyWorker;
}
else{
    int workerNumber2 = id2workerNumber[idWorker2];

    // The worker 2 might not be at work that day :  allWorkers[workerID1] = null
    if(allWorkers[workerNumber2] == null){worker2 = dummyWorker; }
    else{worker2 = allWorkers[workerNumber2]; }
}

if(idWorker3 == -1){
    worker3 = dummyWorker;
}
else{
    int workerNumber3 = id2workerNumber[idWorker3];

    // The worker 3 might not be at work that day :  allWorkers[workerID1] = null
    if(allWorkers[workerNumber3] == null){worker3 = dummyWorker; }
    else{worker3 = allWorkers[workerNumber3]; }
}

citizen = new Citizen(nCitizens, worker1, worker2, worker3, idCitizen);

// System.out.print("worker1.number() =  " + worker1.number() + " | ") ;
// System.out.print("worker2.number() =  " + worker2.number() + " | ") ;
// System.out.println("worker3.number() =  " + worker3.number()) ;

// Put in the citizen
allCitizens[citizen.number()] = citizen;

    }

    // The visitID to set the number of the line
    Visit v = new Visit(visitNumber, start, newFinish, duration, citizen);
    allVisits[visitNumber] = v; // The number of visits start by 1

    // The flag for the next line
    boolean flagNextLine = input.next();

}

//System.out.println("maxVisitNumber in file=  " + maxVisitNumber + "\n") ;


//for(int i = 1; i <= nCitizens; i++){
//   System.out.println(allCitizens[i].toString());

//}

    }

    public Solution readPlan(String planFileName){

InputFile input = new InputFile(planFileName);
input.skipFirstLine();

// All the routes
Route[] allRoutes = new Route[nWorkers +1];


// The cost
double totalTravelTime = 0;
int noOfVisitsWithoutRegularWorker = 0;
double cost;

// THE INITIAL SOLUTION
for(int r = 1; r <= nWorkers; r++){
    Route currentRoute = new Route(r,allWorkers[r]);
    allRoutes[r] = currentRoute;
}

int counter = 0;

int nScheduledVisits = 0;

while(input.getTokenType() != StreamTokenizer.TT_EOF){
//for(int i = 1; i < 2; i++){
    int idWorker = (int) input.getTokenNumber();
```

```
    boolean flag1 = input.next();
    boolean flag2 = input.next();
    int visitNumber = (int) input.getTokenNumber();
    boolean flag3 = input.next();
    boolean flag4 = input.next();
    int distIDcitizen = (int) input.getTokenNumber();
    boolean flag5 = input.next();
    boolean flag6 = input.next();
    double start = input.getTokenNumber();

    //// System.out.print("distIDworker =  " + distIDWorker + " | ") ;
    //// System.out.print("visitNumber =  " + visitNumber + " | ") ;
    //// System.out.print("distIDcitizen =  " + distIDcitizen + " | ") ;
    //// System.out.println("start =  " + start) ;

    counter++;

    // If the worker is on job
    if(idWorker != -1){

nScheduledVisits++;

int workerNumber = id2workerNumber[idWorker]; // Equals the route number
Worker worker = allWorkers[workerNumber];
Route route = allRoutes[worker.number()];
Visit visit = allVisits[visitNumber];
Citizen citizen = visit.citizen();

// SET THE ROUTE NUMBER
visit.setRouteNumber(route.number());

// THE NUMBER OF VISITS WITHOUT THE REGULAR CARETAKER
// Do not count the start visits and the breaks
//System.out.println("Visit.number() = " + visit.number());
//System.out.println("Visit.isPlanned() = " + visit.isPlanned());
//System.out.println("Worker.number() = " + worker.number());
//System.out.println("Citizen.number() = " + citizen.number());
//System.out.println("citizen.isTheWorkerRegular(worker) = " + citizen.isTheWorkerRegular(worker));
if(!citizen.isTheWorkerRegular(worker) & !visit.isPlanned()){
    noOfVisitsWithoutRegularWorker++;
    //System.out.println("noOfVisitsWithoutRegularWorker = " +noOfVisitsWithoutRegularWorker );
}

// THE POSITION FOR THE VISIT
int position = 0;
boolean stop = false;

if(route.length() >0){

    Visit nextVisit = (Visit) route.get(position);

    while(start > nextVisit.start() & !stop){

position++;

if(position < route.length()){
    nextVisit = (Visit) route.get(position);
}
else{
    stop = true;
}
    }

}

// >>> SET THE START AND FINISH TIME
visit.setStart(start);
visit.setFinish(start + visit.duration());

// >>>> INSERT THE VISIT <<<<
route.insert(position,visit);
    }
    // The flag for the next line
    boolean flagNextLine = input.next();
}

// SET THE POSITIONS, TRAVEL TIME, ARRIVAL AND WAITING  TIME
for(int j = 1; j <= nWorkers; j++){

    Route route = allRoutes[j];

    if(route.length() > 0){
int p = 0;
Visit visit = (Visit) route.get(p);
```

```
// Set the position
visit.setPosition(p);
// Set the arrival
visit.setArrival(visit.start());
// Set the waiting time
visit.setWaitingTime(0);

for(p = 1; p < route.length(); p++){
    // The previous visit
    Visit previousVisit = (Visit) route.get(p-1);
    Citizen previousCitizen = previousVisit.citizen();
    // The current visit
    visit = (Visit) route.get(p);
    Citizen citizen = visit.citizen();
    // Set the position
    visit.setPosition(p);
    // Calculate travel time
    totalTravelTime += distance[previousCitizen.number()][citizen.number()];
    // Set the arrival
    double arrival = previousVisit.finish() +
distance[previousCitizen.number()][citizen.number()];
    visit.setArrival(arrival);
    // Set the waiting time
    visit.setWaitingTime(visit.start()-visit.arrival());
}
    }
}


// PRINT OUT THE ROUTES
for(int j = 1; j <= nWorkers; j++){
    Route currentRoute = allRoutes[j];
    // PRINT OUT THE ROUTES
    System.out.println(currentRoute.toString(distance) + "\n");
}

System.out.println("nScheduledVisits = " + nScheduledVisits + "\n");

Solution solution;
solution = new Solution(allRoutes, allVisits, true);
solution.setTotalTravelTime(totalTravelTime);
solution.setNoOfVisitsWithoutRegularWorker(noOfVisitsWithoutRegularWorker);

return solution;
    }

}
```