UTRECHT UNIVERSITY

MASTER THESIS

ADVANCED PLANNING AND DECISION MAKING

The pickup and delivery problem with time-dependent travel times

Author:

V.R. BONS

Academic Supervisors:

dr. J.A. Hoogeveen

dr. ir. J.M. van den Akker

External Supervisor:

Drs. F. Paanakker

April 15, 2014

Abstract

In this thesis we consider the multi-depot vehicle routing problem with pickup and delivery requests with time windows, based on a real world case of a precision transport company. To be able to generate more realistic schedules we include time dependent travel times in our model. The algorithm can handle alternative pickup and delivery locations and times, and gives the possibility to generate solutions with a maximum shift time to make sure driver regulations are respected. This problem is solved using a simulated annealing approach combined with large neighborhood operators. The algorithm is guided towards better solutions by using a selection probability that prefers better changes, and a similarity value to select orders that are similar to each other. A comparison using a benchmark by Li & Lim shows that our algorithm is competitive on basic Pickup and Delivery instances with time windows. To allow for comparison of algorithms in further research we have created a benchmark for the problem with time dependency and alternative pickup and delivery options.

Keywords: Multi Depot, Multi Vehicle, Pickup & Delivery, Alternative Locations, Local Search, Simulated Annealing, Large Neighborhood Search, Time Dependent Travel Time, Insertion, Similarity, Benchmark

Preface

This thesis is written in conclusion of the master program Computing Science at Utrecht University, in the Netherlands. For a year I have researched a vehicle routing problem based on a practical case of a transport company, at Backbone Systems in Amersfoort. I would like to thank my supervisors from Utrecht University, Han Hoogeveen and Marjan van den Akker, for their support. Marjan van den Akker for introducing me to Ferns Paanakker from Backbone Systems, and helping me with finding a project. Han Hoogeveen for the good feedback and advice that he gave me during the project. I would also like to thank my supervisor at backbone systems, Ferns Paanakker. Firstly for giving me the opportunity to do this project at Backbone Systems. With this project he gave me an introduction to the field of work, and a good real world case to solve. Secondly for managing to find time to discuss my progress, ideas and struggles during the whole research, despite his busy schedule. In addition I would like to thank Nathalie van Zeijl, who works at Backbone Systems, for proofreading my thesis and giving useful feedback. Last but not least I would like to thank my roommates, friends, family and partner for keeping me motivated during the project.

TABLE OF CONTENTS

Abstract ii

Preface iii

1 Introduction 1

2 Problem Definition 3
2.1 Basic Problem . . . 3
2.2 Extensions . . . 5
2.2.1 Alternative Pickup and Delivery Locations . . . 5
2.2.2 Maximum Shift Time . . . 5
2.2.3 Time Dependent Travel Time . . . 6
2.3 Model . . . 6
2.3.1 Parameters . . . 6
2.3.2 Constraints . . . 7
2.3.3 Objective Function . . . 9

3 Literature 12
3.1 Approaches for solving the PDP . . . 12
3.1.1 Exact methods . . . 12
3.1.2 Metaheuristics . . . 13
3.2 Extensions of the PDP . . . 17
3.2.1 Time Dependant Travel Time . . . 17
3.2.2 Dynamic PDP . . . 18
3.3 Parallel Computing . . . 19
3.4 Summary . . . 19

4 Implementation 21
4.1 Algorithm . . . 21
4.1.1 Solution Representation . . . 23
4.1.2 Similarity . . . 23
4.1.3 Initialization . . . 25
4.1.4 Improvement phase . . . 27
4.2 Operators . . . 30
4.2.1 Small neighborhood operators . . . 30
4.2.2 Large neighborhood operators . . . 32
4.3 Time Dependency . . . 36
4.3.1 Travel time without time dependency . . . 36
4.3.2 Travel time with time dependency . . . 36
4.3.3 Calculation of travel time using periods . . . 37
4.4 Maximum shift time . . . 39
4.5 Inserting Orders . . . 41

4.5.1 Slack Values 41
4.5.2 Insertability 43
4.5.3 Insertion 48
4.5.4 Removal 48
4.6 Summary 49

5 Experiments & results **50**
5.1 Initialization methods 50
5.2 Operators 51
5.3 Temperature 54
5.4 Selection by probability 55
5.5 Large Neighborhood search 56
5.6 Benchmark : Li & Lim 58
5.7 Time dependency 61
5.8 Alternative pickups & deliveries 62
5.9 Maximum shift time 63
5.10 Benchmark : Time Dependency & Alternative Locations 63
5.11 Real world problems 67

6 Conclusion & Future Research **69**
6.1 Conclusion 69
6.2 Future Research Possibilities 71

Bibliography **73**

Chapter 1

Introduction

Transportation is an important issue. In the Netherlands alone there are over 10.000 companies involved in transporting goods. This brings forward many challenges, from packing and sorting the goods to planning the routes the vehicles take to deliver the orders. Because competition in this sector is tough, every advantage counts. Efficiency is a key factor in gaining this advantage. In this thesis we study methods for solving the routing of the vehicles as efficiently as possible for practical and theoretical cases.

The most common routing problem is the standard vehicle routing problem. An example of this problem is the delivery of mail. There is a fleet of vehicles that all depart from a given starting point, called the depot. At this depot they load the mail that has to be delivered to all the customers into the vehicles. The goal is to create the most efficient set of routes possible for this fleet of vehicles. With this type of vehicle routing many practical transportation problems can be modeled and solved. In this paper we will discuss a different kind of routing problem, namely the pickup and delivery problem (PDP). A good example of this is a moving company, which moves boxed office equipment from one location to another. Instead of a single starting location at which all the cargo is loaded, like in the previous example, in the PDP there are many locations where boxes have to be picked up. Once a box is picked up, it has to be delivered at a given destination. Instead of a group of locations that can all be treated the same, there are two different types of locations: pickup locations and delivery locations. Boxes are picked up at the pickup locations and then moved to their corresponding delivery location. Where in simpler cases pickups could be left out of the algorithm because they were all picked up at the same location, they now have to be included in the planning process.

In this thesis we will describe an algorithm that can solve such a PDP, with a number of extensions. The problem that will be studied is based on a real world transportation optimization case for a precision transport company. Precision transport means the transport of vulnerable, high value equipment like pianos, printers, medical and scientific equipment etc. Although the studied problem is based on this practical case, the algorithm is created to solve a more general problem that includes this particular problem. In essence the problem we are solving is a Pickup

and Delivery problem with time windows (PDPTW), where all the pickups and deliveries have a time window in which they can be serviced. Every day this company services a number of orders from their clients. Most of the orders are a combination of a pickup and a delivery at two different locations, but an order can also be an internal relocation where both the pickup and the delivery are at the same location. This company has a number of vehicles at its disposal to service these orders. These vehicles are all assigned to a depot, selected from a set of one or more depots that belong to the company. The vehicles start their day at the depot to which they are assigned, service their orders, and return to the same depot at the end of the day.

We create an algorithm to solve this problem using a combination of Simulated Annealing and Large Neighborhood Search, which will both be explained later. We also consider extensions to the basic problem that allow for more detailed problems to be solved: Time-dependent travel times, alternative pickups and deliveries, heterogenous vehicles and maximum shift times to accommodate the drivers.

Alternative pickups and deliveries can be used to give the algorithm more flexibility in planning the orders. From the alternative pickups and deliveries one pickup and one delivery should be chosen. With these alternatives a situation could be modeled where a pickup can be done on Monday morning at a work address or Thursday evening at a home address for example. From these alternatives the algorithm should choose the option that fits the schedule best.

In reality travel times are not always the same, they change over time. If this aspect could be included in the planning algorithms, more realistic schedules could be made. Also roads that are very likely to have traffic jams could be avoided during rush hour. Not only rush hours but any event that has a predictable influence on travel times could be included. An important feature of our algorithm is the possibility to include time-dependent travel times.

We will describe an algorithm to solve the basic PDPTW with the extensions of time-dependent travel time and alternative pickup and delivery locations. First, in Chapter 2, a description of the problem that is the subject of this study will be given. In Chapter 3 the literature on this subject that is already available is reviewed. Then in Chapter 4 we explain the algorithm we created to solve this problem, including the extensions to the basic problem. Chapter 5 will contain the results of the experiments we did with this algorithm as well as a comparison to similar algorithms. We end with a conclusion in Chapter 6, in which we reflect on the algorithm and review the advantages and disadvantages of the used approach. Also future research and improvements will be discussed in this last chapter.

Chapter 2

Problem Definition

2.1 Basic Problem

In the problem that we solve, a schedule has to be made that routes vehicles as efficiently as possible to be able to pick up and deliver a number of orders. In this chapter we will describe all the aspects of this in detail.

Orders & Depots

A problem instance contains $|O|$ orders, where O is the set that contains all orders. In the regular PDP every order consists of one pickup and one delivery node, where for every order the pickup is planned before the delivery, and both parts of the order are served by the same vehicle. This means that an order $o \in O$ consists of exactly one pickup and one delivery node. All pickup nodes together form the set P and all delivery nodes the set D. For every order $o \in O$ the pickup and delivery node have to be served, or none of them if the order can not be planned. It is impossible to have a pickup serviced without the delivery or vice versa.

In addition to the order nodes there is a set of depot nodes E. These represent the depots the vehicles depart from and return to. There has to be at least one depot node. All pickup and delivery nodes together with the depot nodes E form a graph where all nodes are connected to all other nodes. Every arc (i, j) going from node i to j gets assigned a driving distance d_{ij} and a driving time dt_{ij}. All pickups and deliveries have a time window $[a_i, b_i]$ in which they must be served. Service at location i takes $serviceTime_i$ units of time. The service must be finished before b_i.

Vehicles and Capacity

For servicing these orders a fleet of $|V|$ heterogenous vehicles is available. All of these vehicles start from a given depot $e \in E$ at which they end as well. Because vehicles can only carry a certain amount of cargo all vehicles $v \in V$ get a capacity Q_v. The capacity can be different for every vehicle. As there are different vehicles, these vehicles can be assigned different costs. That is why we introduce the fixed vehicle cost, $fcost_v$ and variable cost per time unit ($drivingcost_v$,$workingcost_v$ and

$waitingcost_v$) and per distance unit $vcost_v$. The fixed cost for a vehicle is the penalty contribution that is given for every day that this vehicle is in use. The variable cost can represent gasoline usage and drivers' wage for example.

A vehicle can never carry more cargo at the same time than its capacity allows for. Every pickup gets a positive load q_i equal to the weight of the object to be transported, and every delivery gets a load equal to the negative weight of the object that is delivered. In this way you can keep track of the total weight carried at every node by taking the sum of all loads from previous nodes visited by the same vehicle. Once a vehicle picks up an order i it keeps this order on board until it is delivered, which means that the available capacity of the vehicle is reduced by q_i until the order is delivered. A vehicle can not transfer picked up cargo to another vehicle.

In practice linear models for describing capacity are often not adequate. The number of orders that fit in a vehicle depends on the dimensions of the vehicle and the orders to be transported. For example, a package may not fit in a vehicle due to its dimensions, while the weight of the package does not exceed the capacity of the vehicle. For this research we have chosen to use a simple capacity model, because we want the model to be generic. For specific cases other ways of describing capacity could be used.

Routes

A planning can be made for a planning window of one or multiple days. Every vehicle $v \in V$ has a set of time periods W_v in which it is available. This set can be used to represent the days at which vehicle v is available in the planning window. To every time period $w \in W$ one route can be assigned that fits inside this time period.

A route r is a sequence $(r_1, r_2, .., r_k)$ of actions to be serviced. A route always starts and ends with an action at a depot node $e \in E$, and represents the schedule of a vehicle $v \in V$ for a single time period $w \in W_v$. The set of all routes that are driven forms the schedule.

Actions

An action a describes visiting a node $node_a \in N$ (where N is the set containing all pickup, delivery and depot nodes) and performing the service corresponding to that node. This can be a picking up an order, delivering and installing an order or the preparation of a vehicle at the depot. An action has a start time $start_a$, and an end time of $start_a + serviceTime_{node_a}$. The set of all actions is defined as A. The set of all planned actions for a subset of nodes $N' \subseteq N$ is A'_N.

2.2 Extensions

In addition to the basic Pickup and Delivery problem we introduce a number of extensions. These extensions allow for a broader range of problems to be solved. Another reason for these extensions is that because more information can be used, the algorithm should be able to generate solutions that relate to reality better.

2.2.1 Alternative Pickup and Delivery Locations

We add the possibility to define orders that have multiple alternative pickups and deliveries, of which one pickup and one delivery have to be chosen. These alternative actions have their own time window, and may also have different locations. Alternative pickups and deliveries can be used for the purpose of allowing multiple time window and location combinations per order to improve planning flexibility. For instance one can say that an order can be picked up on a Monday morning between 9:00 and 12:00 or at a Thursday afternoon between 15:00 and 17:00. If there are many orders scheduled for Monday morning, then we could plan the order at the Thursday.

This means that an order $o \in O$ consists of a set of pickup and delivery nodes P_o and D_o, instead of a single pickup and delivery. For every order o exactly one of the pickup nodes in P_o has to be serviced, and exactly one of the nodes in D_o (or none at all if the order is not scheduled). This addition may be useful in a number of situations that occur in practice, such as different delivery locations over time for traveling customers.

2.2.2 Maximum Shift Time

There are limits to the number of hours an employee can and may work contiguously. To make the solutions more realistic we introduce shifts. A shift starts and ends at a depot, and represents a period of contiguous work for one driver. For example if a route r starts at 5:00 am and ends at 10:00 pm. This adds up to a total of 17 hours. Say we use a realistic maximum work time for a driver of 8 contiguous hours per day. As these 17 hours exceed the maximum time a driver can work on one day this route could be split into three shifts of 7, 7 and 3 hours, which are all less than the maximum of 8 hours. After every shift the vehicle has to return to the depot to exchange drivers. Of course this takes up time because the vehicle has to return to the depot, and there has to be enough time for the drivers to exchange. The maximum time that is allowed for a shift is $maxshifttime$. This is the same for all drivers. The maximum time for a shift is not a hard constraint, however exceeding the shift time will add a penalty value. This choice is made because sometimes it may be desirable to allow for a little overtime to be able to plan more orders. However only small overtime values should be allowed, because drivers can not be expected to work more than they are allowed to. Although maximum shift time is considered when scheduling the actual driver schedules are not created. This

extension makes sure there are no shifts that are impossible to assign to any of the drivers. This approach only works with identical drivers. If the orders or vehicles have requirements that not all drivers meet, driver schedules have to be taken into account as well.

2.2.3 Time Dependent Travel Time

One of our main research goals is to implement an algorithm that can handle time-dependent travel times and distances. For achieving this, another parameter has to be added to all of the arcs: time. Every arc (i, j) going from node i to j now has a driving distance d_{ijt} and a driving time dt_{ijt} which depend on the time of departure t from node i. The distance can differ, because a different route from point a to b can become preferable. In this way you can include the effects of rush hours, weather conditions and events on the driving time to increase the accuracy of the planning. These time-dependent times work on the assumption that the FIFO (First In First Out) property holds. FIFO means that if you leave earlier you always arrive earlier or at the same time. A result of this is that if a vehicle drives an arc (i, j) at time t it always arrives at node j earlier or at the same time than when the departure time is later than t. Time dependent travel distances do not impose new restrictions, only a change in penalty contributions. A change in travel time however causes new restrictions on the possibility to add certain orders on a particular time, as the total travel time may increase and render the solution impossible.

2.3 Model

Given this problem description we create a model that describes the problem more formally. First we will describe the most important parameters that need to be taken into account when solving the problem. Secondly we will describe the constraints, and finally the objective function.

2.3.1 Parameters

These are the most important parameters. They are known beforehand, and are used to describe the problem instance.

P : set of Pickup Nodes.
D : set of Delivery Nodes.
E : set of Depot Nodes.
N : set of all Nodes $N = P \cup D \cup E$.
V : set of Vehicles.
O : set of Orders.
P_o : set of Pickup Nodes that belong to order $o \in O$.
D_o : set of Delivery Nodes that belong to order $o \in O$.
W_v : set of time windows $[ws_v, we_v]$ in which vehicle $v \in V$ is available.
Q_v : capacity of vehicle $v \in V$.

$depot_v$: indicates the depot $e \in E$ that vehicle $v \in V$ starts and ends its routes on.

$vcost_v, fcost_v, drivingcost_v, waitingcost_v, workingcost_v$: All the penalty values for vehicle $v \in V$. These values define weights in the objective function.

d_{ijt} : distance in centimeters between nodes i and j when departing from node i at time t, $d_{ijt} \geq 0 \forall (i,j) \in N \forall t$.

dt_{ijt} : time in seconds it takes to get from node i to node j when departing from node i at time t, $dt_{ijt} \geq 0 \forall (i,j) \in N \forall t$.

$[tws_i, twe_i]$: time window for node i. This node can only get service between tws_i and twe_i.

q_i : the value that is subtracted from the remaining capacity of the vehicle $v \in V$ that visits location i. The value of q_i is negative for deliveries and positive for pickups. For all other nodes q_i is 0.

$serviceTime_i$: the duration of service at node i.

$maxshifttime$: indicates the limit of time per shift.

$maxtime$: indicates the limit on the planning horizon.

2.3.2 Constraints

Because this problem is based on a practical case, it comes with many constraints. The constraints we will describe next are all hard constraints. This means that these constraints can not be violated.

Order constraints

All of the orders have the possibility to be either fully planned or not planned. This means that for every order there is the possibility to have exactly one pickup action planned and exactly one delivery action planned, or no actions planned at all. It is not possible to plan a delivery action without a pickup action and vice versa, or to plan more than one pickup or delivery action for an order.

Route and Shift constraints

There are a number of constraints on routes and shifts. Every vehicle $v \in V$ has a set of time windows $w \in W_v$ in which it is available. These time windows never overlap other time windows of the same vehicle, because two overlapping time windows could be described as a single time window. At most one route $r \in R$ can be assigned to every time window, if it satisfies the condition that all of the actions in r can be planned inside the time window of w. The starting time of the first action of this route should be greater than or equal to ws_w and the end time of the latest action should be less than or equal to we_w. Furthermore, the routes have to be chosen so that none of the order constraints are violated.

A shift is defined as the part of a route that starts and ends at a depot. Since all routes start and end at a depot, all the actions of a route have to be part of a

shift as well. A route can consist of one or multiple shifts, depending on the number of depot visits inside the route.

Time Constraints

Time imposes many constraints upon our problem. The first constraint is that all routes have to be inside the given planning window, i.e. the start time of all routes has to be greater than or equal to 0 and less than or equal to $maxtime$, the limit of the planning window. When an action a is planned it has to be planned inside its time window. It must start after tws_a and the service must be completed before twe_a; the start time of a must be less than or equal to $twe_a - serviceTime_a$.

A solution is invalid if the planning is too tight to allow for the vehicle to arrive at its next location in time. To guarantee that this will never happen we define a constraint on all actions: An action a can never be planned earlier than the earliest possible arrival time, given the properties of the previously planned action $prev_a$. The start time of action a ($start_a$) should be greater than or equal to the start time of $prev_a$, plus the service time at the previous location, and the travel time from the previous location to the location of a when departing at time $start_{prev_a} + serviceTime_{prev_a}$. This constraint ensures that an action is never planned earlier than it can be serviced. Since it is done for all actions it also ensures that none of the orders can be planned too late, because the constraint would be broken for the next action. It also ensures the trivial constraint that an order can never be planned later than its successor, or earlier than its predecessor.

Precedence Constraints

In the PDP pickup and delivery actions are not only constrained by time but they also put constraints on each other. For all orders the delivery action must always be planned later than the pickup action, because you can not deliver a package before it is even picked up. Also, it should not be possible to let the pickup and delivery actions of an order be delivered by a different vehicle, because transferring orders between vehicles is not allowed.

Capacity Constraints

Besides the time constraints there are other constraints on the vehicles to take into account. All the vehicles have a capacity Q_v, which indicates the amount of goods it can carry at any given time. The load a vehicle carries can never exceed Q_v at any moment in the planning. The orders' weight is subtracted from the remaining capacity of the vehicle at the moment the order is picked up and it is added again after the order is delivered.

2.3.3 Objective Function

Given this description of the problem our goal is to *minimize* the following function:

$$\alpha \cdot Unplanned(S) + \beta \cdot Vehicles(S) + \gamma \cdot Distance(S) + \delta \cdot Time(S) \qquad (2.1)$$

The objective function consists of four parts. $Unplanned(S)$ stands for the number of unplanned orders in solution S. $Vehicles(S)$ is the number of vehicles used in the solution. $Distance(S)$ is the total distance driven by all of the vehicles. $Time(S)$ is the sum of all time related penalties. The combination of these four parts are minimized, with the given weights α, β, γ and δ.

The objectives mentioned above are conflicting. Improving one of the objectives may deteriorate the other objectives. As a simple example, when optimizing cost and customer satisfaction, if no expenses are spared to make the customer as comfortable as possible, it will hurt the profits. On the other side, doing everything with minimal expenses might leave the customer with inferior service. In other words, there is no solution possible where both objectives are optimal. We deal with this by giving weights to the different objectives. More important objectives get a higher weight. In this way, if two objectives are conflicting, the solution that has the lowest cost of all objectives combined given their weights gets preferred.

The values for α, β, γ and δ can be de adjusted to fit the problem to be solved. The value of α is always relatively high, because planning all orders is often the most important objective (unless the goal is to find orders to outsource). The values for β, γ and δ depend on the problem to be solved. If drivers are relatively expensive the number of vehicles used is an important objective to minimize, because every vehicle needs a driver. If the variable cost of driving vehicles is relatively high we would prefer minimizing the total distance driven and the driving time.

Number of vehicles

Minimizing the number of vehicles used is necessary, because costs increase when more vehicles are used. Every vehicle needs a driver, and in addition the fleet size can be smaller if the number of vehicles needed to service the orders is lower. The objective is defined as follows:

$$\sum_{v \in V} \sum_{r \in R(S)} fcost_v \cdot drives_{vr} \qquad (2.2)$$

Here, $R(S)$ is the set of routes used in solution S. $drives_{vr}$ indicates if a vehicle $v \in V$ drives route r; the value of $drives_{vr}$ is 1 if vehicle v drives route r, and 0 otherwise.

Distance

We want solutions to be as efficient as possible. That is why we want to minimize the distance driven. Different vehicles can have a different cost per distance unit. Take for example a fleet of vehicles with a number of old inefficient vehicles and a number of new vehicles that are very eco-friendly and efficient. The company with this fleet would prefer to use the newer part of the fleet because these vehicles use less fuel per kilometer driven, which minimizes the driving cost and also contributes to a better reputation for the company. Therefore it should be possible to define the variable cost $vcost_v$ for each vehicle separately. This objective is defined as the total weighted distance driven by all vehicles, where the weight of a vehicle v is equal to its variable cost $vcost_v$.

Time

Another important aspect of the objective function is time. The time objective consists of three parts: Driving Time, Waiting Time, and Working Time. Driving time is the total time the vehicles spend traveling between locations. The waiting time between two actions is the idle time. When a vehicle departs at a location a at time t when work is finished, it would arrive at the next location b at time $t + t_{abt}$. If this arrival time at b is earlier than the start of work at location b there would be a waiting time which is the difference between the start of work at location b and the arrival time. The final part is the time spent on orders, which can differ if alternatives for pickups and deliveries are defined.
The weights of these three parts of the time objective can be adjusted for different situations. In a realistic problem, an important goal is to minimize the total cost. The total time of a shift indicates the number of hours a driver has to work. Shift time is variable, and depends on the amount of time it takes to service the orders in the shift. In this case waiting time is unwanted, because there is no productivity but there are still costs. However, equal cost for waiting time and driving time could lead to inefficient routes, because detours would have no effect (except for the distance penalty) on the value of the objective function. Weights are determined per vehicle. This is because different vehicles could have a different impact on the quality.

The second aspect in determining the time penalty is overtime. When we have determined a maximum shift time $maxshifttime$, we add an overtime penalty for every shift which time exceeds $maxshifttime$. The overtime penalty is equal to $\sum_{s \in S} max(0, shiftTime_s - maxshifttime)^x$. Here x determines the speed at which the overtime penalty increases. Because of the exponential formulation, the shift time can be exceeded by a small amount, but as the overtime increases the penalty quickly rises to a high value.

The time objective can be defined as follows:

$$\begin{aligned}&\sum_{v\in V}\sum_{w\in W_v}\sum_{r\in R}(travelcost_v \cdot travelTime(r) + waitingcost_v \cdot waitingTime(r)\\&+workingcost_v \cdot workingTime(r) + \sum_{s\in S_r}((duration(s) - maxshifttime)^x))\end{aligned} \tag{2.3}$$

Where $travelTime(r)$ is the time travelled in a route r, $waitingTime(r)$ is the time a vehicle is waiting in route r and $workingTime(r)$ is the total time spent working at location (the sum of service times) in route r. The set S_r contains all shifts driven in route r, and $duration(s)$ is the total duration of shift s.

Chapter 3

Literature

Many versions of the Pickup and Delivery problem (PDP) have been studied over the last decades. There are a number of different instantiations of the PDP. The generalized problem is a problem with characteristics that return in all of these different instantiations of the PDP. Savelsbergh and Sol [1995] describe this generalized version of the pickup and delivery problem. They define the generalized PDP to consist of a set of transportation requests which are serviced in a number of routes. A route starts at a given node and ends at another given node, which may be the same, and is driven by one vehicle k. Between the start and end node a number of pickup and a number of delivery nodes are included in the route. For every delivery node there is a corresponding pickup node. All of the nodes are served only once by a vehicle k. During the whole route the vehicle load never exceeds it capacity. The whole solution is a set of routes, for which it holds that every pickup and delivery node is visited exactly once. They also consider a number of more specific problems individually, including the pickup and delivery problem with time windows. On all of the problems that are covered additional literature is given. They divide the problems that are covered in three categories: Pickup and Delivery for goods, for passengers (Dial-a-Ride Problem), and the standard vehicle routing problem, where all the pickups are located at the depot. These problems are all PDPs, but they have different constraints and properties. The problem we are going to solve is a PDP for goods, with time windows.

3.1 Approaches for solving the PDP

In literature there are many ways in which the PDP for goods is solved. We will give an overview of the literature on this subject, categorized by the approach used to solve the problem.

3.1.1 Exact methods

Since the PDP is proven to be NP-hard (see Lenstra and Rinnooy Kan [1981]) solving these problems to optimality is very hard, and larger instances can not be solved in a reasonable amount of time. Often exact approaches in literature use a column

generation approach. For solving large mixed integer problems like the PDPTW branch-and-price (see Barnhart et al. [1998]) can be used. This approach works as follows. A mixed integer program can be expressed as an objective function that has to be minimized or maximized, a set of parameters, a set of decision variables and a set of constraints. Because in these problems there are often too many variables to consider, and most of these variables are not used at all, the problem is solved using a subset of the variables. The LP-relaxation of this subproblem is solved. Then, to find out whether improvement upon the current solution is possible a *pricing problem* is solved. With the pricing problem we find the variable that improves the solution most if included, and add this variable to the subset of variables to consider. If no such variable exists, and the solution is integral, the optimal solution is found. If it is not integral, branching is applied; for each node (branch) a lower bound is computed by solving the LP-relaxation again using column generation.

Sol and Savelsbergh [1994] used a branch-and-price approach to solve small sized instances (50 orders) of the PDPTW. They formulate the problem as a set partitioning problem. They consider two different implementations of the pricing problem, and two implementations of branching. Their results, on a set of randomly generated problem instances are good: most of the time the optimal solution is found. Furthermore they state that their algorithm can easily be adapted to an approximation algorithm to solve larger problem instances.

Although the problem can be solved very well using column generation techniques, even for a reasonable number of customers, we are uncertain if it would be possible to solve highly constrained large instances using the column generation approach. Especially with the addition of time-dependent travel times solving the problem would probably become very hard using this method. More literature on solving vehicle routing problems in an exact fashion can be found in the paper by Baldacci et al. [2010].

3.1.2 Metaheuristics

In problems where exhaustive search is not possible due to the size of the search space metaheuristics can be used. Metaheuristics guide the search process to try and find a good solution. These methods are not guaranteed to find the optimal solution, because not all possible solutions are tried, and the search may end in a local optimum. Because the PDP usually has a large search space, metaheuristics can be used to find good solutions in a reasonable amount of time. Also most of these methods are very flexible, making it easier to adapt to new problems. In literature many methods can be found that solve this problem using metaheuristics. We will summarize a few of these methods to give an understanding of the possibilities, and the previous accomplishments in this area.

First we will introduce some concepts often used in non-exact algorithms. All of these algorithms have a *current solution* (or in case of the genetic algorithms, a

population of solutions) to which small modifications are made to improve it. A simple greedy algorithm can be used to generate an initial solution to start with. The *neighborhood space* consists of all solutions that are reachable from the current solution by making a single modification of a specified type to it. The type of modification is determined by the *operator* used. An operator is a predefined method for making changes to a solution, for example removal and reinsertion of one random order. In this way the *search space*, which contains all possible solutions, is explored. All solutions in the search space have a fitness value, which describes their quality. The solution with the best fitness value is the *global optimum*. A solution that has no solutions in its neighborhood that are better is called a *local optimum*.

Hill climbing

One of the most basic methods for improving solutions of large problems is hill climbing (see Tovey [1985]). The hill climbing algorithm improves a solution by repeatedly selecting new solutions from the neighborhood space of the current solution. Only new solutions that are an improvement to the current solution are accepted. There is a large possibility that the algorithm gets stuck in a local optimum, because it has no way of escaping from local optima. The final solution depends on the starting solution, and the operators used. The problem of getting stuck in local optima can be partially overcome by restarting a number of times. In this way, different parts of the search space can be explored to find a better solution.

For the reasons mentioned above hill climbing is mainly used as part of a more sophisticated algorithm. The main advantage of hill climbing is that it is able to quickly improve a solution.

Tabu Search

Tabu search (see Glover [1989]) is a metaheuristic that keeps a list of previously visited solutions: the tabu list. In every step of the algorithm a neighborhood is searched and the best solution in that neighborhood that is not on the tabu list becomes the new current solution. This process is repeated for a given amount of time, or until no more improvement was found for a number of iterations.

Cordeau and Laporte [2003] have created a method using Tabu Search for solving a largely similar problem as the PDP, namely the Dial-a-Ride Problem (DARP). Here people are transported instead of goods. An example could be public transport in rural areas, where people have to make a call to be picked up. The difference with the problem we are treating is that people do not want to take large detours, because they want to be at their destination as fast as possible, while transported goods do not care about detours. To give more flexibility to the searching process the authors allow infeasible solutions. All the time constraints, and capacity constraints may be exceeded. To guide this possibly infeasible solution to a feasible one the penalties given for violations are dynamic. First they start with low

penalties for exceeding constraints. Over time these penalties get higher, so that at the end of the process infeasible solutions are too costly to be allowed. Allowing infeasible solutions makes it easier to find a solution to start the search process from.

Creating an initial solution is easy, as the authors allow time and capacity constraints to be broken. The initial solution is created by randomly assigning every request, consisting of a pickup and a delivery, to a randomly selected vehicle. The delivery is always placed after the pickup. In this way you are certain that a pickup always comes first, and the delivery is placed in the same route.

When a request is transferred to another route, computing the impact of this change can be costly, because all of the other requests in the route may be influenced by this change. The impact on the quality of the solution of removing a pair of vertices and inserting them into another route is computed with the help of the slack value F_i (see Savelsbergh [1992]). F_i indicates the forward slack of a node. Forward slack is the maximum time a node i can be moved forward in time without changing the end time of the route. First the time window constraint violations are minimized, then the route duration is minimized without increasing the penalty for exceeding time window constraints. The method they describe gives optimal departure and arrival times for a given route in a quick fashion. The impact is the difference between the result of this calculation and the routes before the transfer of nodes.

In their conclusion the authors state that their method is easily adapted to other routing problems with time windows, such as our version of the PDP.

Simulated Annealing

Simulated Annealing (see Kirkpatrick et al. [1983]) is a local search method like Tabu Search. It works with a temperature that gradually gets lower as the algorithm proceeds. At the start, when the temperature is high, worse solutions are accepted with high likelihood to be able to search a large part of the search space without getting stuck in local optima. As the temperature decreases the probability of allowing solutions worse than the current one decreases with it. Solutions that are worse than the current one get accepted with the probability value as described in Equation 3.1. Here *temperature* is the current temperature and $difference$ is the new fitness value minus the old fitness value, which is always a positive value because acceptance probability is only calculate when considering worse solutions, and the higher the fitness value the worse the solution is.

$$e^{\frac{-difference}{temperature}} \tag{3.1}$$

For a predetermined number of iterations or amount of time new solutions are chosen randomly from the neighborhood space, which is the space accessible by making a single modification to the current solution.

Şahin et al. [2012] wrote about their research on a special case of the PDPTW where they allow loads to be split among different vehicles. They use an interesting algorithm that combines Tabu Search with Simulated Annealing: Tabu Embedded Simulated Annealing (TESA). When selecting a new neighbour from the neighborhood space they do not take the best non-tabu neighbour. Instead they choose a new solution from a set of good non-tabu neighbours. If the selected neighbour is worse than the current solution, it is selected with a probability like in simulated annealing. For creating an initial solution they use an adaption of the savings algorithm by Clarke and Wright [1964]. After creating an initial solution the algorithm performs four stages for a number of times. Every stage uses a different neighborhood to search, until no more improvement is found. The authors of this paper state that the algorithm performed considerably better with simulated annealing and tabu search combined than with only tabu search. When they experimented with only Tabu Search the algorithm got stuck on local optima more often. Furthermore one of the disadvantages of Simulated Annealing, the lack of memory based decision making, is overcome by using tabu lists.

More literature on solving the PDPTW with simulated annealing can be found in Zidi et al. [2012].

Genetic Algorithms

Genetic algorithms are another good option for solving optimization problems. A genetic algorithm is based on the way natural evolution works. The algorithm starts with a population of solutions, called individuals, which are described as dna strings. This string usually consists of a number of binary parameters, but these variables can also get other values. The idea is that the population evolves over time by competing with each other.

First an initial population is created, this can just be a group of random solutions. In every iteration, which is called a generation, a set of individuals is selected. This selection is usually based on fitness of the individuals. There are different ways to make this selection. An example is to select the best half of the population, or let all individuals compete in a tournament to select the ones that perform best. The selected individuals are copied, and all of the copies are changed using mutation or recombination operators. Mutations modify an individual a little by changing one or multiple values. Recombination operators, as the name suggests, recombine two individuals by selecting a part of the first and a part of the second individual. Local search can be used to improve the modified copies before letting them compete with the old population again. After the mutation/recombination phase the algorithm starts over at the selection phase again. This is repeated until a certain criterion is met.

Chevrier et al. [2012] created a hybrid evolutionary algorithm for solving a DARP. Solutions are simply encoded as strings of pickups and deliveries in a given order for

every vehicle. The ordering in this string is the order in which the vehicle visits the nodes. In every string all the characters have to appear twice. The first occurrence is the pickup and the second occurrence is the delivery. It is assumed that all vehicles start and end at the depot, so this does not have to be included in the strings. The algorithm repeats a process of selection, crossover and mutation on a population. After the mutation step a hill climbing algorithm is applied using 2-opt and shifting of customers between routes, making the algorithm hybrid.

A relatively new evolutionary computing method which we will not discuss in this paper is particle swarm optimization. For those interested, Sombuntham and Kachitvichayanukul [2010] wrote two papers about this topic.

Large Neighborhood Search

Large Neighborhood Search (LNS) is another method that is often mentioned in literature about the PDP. Where the neighborhoods in the metaheuristic methods mentioned before only allow small modifications to the current solution, LNS neighborhoods are much larger and could allow for a large part of the solution to be changed in one iteration. These larger neighborhoods make it easier for the algorithm to escape from local optima, but naturally take more time to compute.

Ropke and Pisinger [2006] used LNS to solve the PDPTW. They use an adaptive approach, which means that the improvement history of all the operators defines the chance of using them again. Operators with a better chance of improving the solution get picked more often. The operator they use removes a number of orders from the solution and inserts them again. Different ways of removing and inserting orders can be combined to create operators. They combine this LNS method with simulated annealing. They keep a temperature just like in simulated annealing, which determines the chance that the changes made by the operator get accepted. While the LSN operators worked very well for improving a solution, they were not able to reduce the number of vehicles sufficiently. This is why they created a two-phase approach. In the first phase the number of vehicles is reduced by solving the problem until all orders are planned; then they remove a vehicle and try to solve the problem again. This continues until the number of vehicles is no longer sufficient to service all orders in the given time without violating the constraints. Then one more vehicle is added and the algorithm goes to phase two, which tries to improve this solution with the minimum number of vehicles as much as possible.

3.2 Extensions of the PDP

3.2.1 Time Dependant Travel Time

One of the main research questions of this thesis is how to include time dependent travel times in a proper way. The literature on vehicle routing problems using time

dependent travel times is relatively sparse.

Ichoua et al. [2003] created a model for solving this problem. There are many ways to describe time dependent travel times. Discrete travel time or cost functions could be used. This means that per time interval travel times can change to a new value. This could result in large sudden changes in travel time. In reality travel time increases and decreases continuously. The authors of this paper chose for a continuous travel time function, where they work with travel speeds instead of travel time. The travel speeds can change per unit of time and affect the travel time from location to location. They set the travel speeds for a number of points in time. The values in between these points are linearly interpolated. They use a parallel tabu search algorithm to solve the problem with time dependent travel times. A lateness penalty is used to indicate the badness of a solution. Total lateness is the sum of lateness for all actions. Calculating the influence of an insertion or removal of an action becomes more expensive with time dependent travel times, because the travel times for all actions can change as an effect of a single change.

A number of algorithms that are adapted for calculating travel times that depend on several factors such as weather conditions, traffic information and events are described by Van Zeijl [2013]. These time dependent algorithms are based on existing algorithms for finding shortest paths. For real-world problem instances this research can improve the solution quality considerably, because important topical information is considered when creating the routes.

More information on time dependency in Pickup and Delivery problems can be found in a paper by Donati et al. [2008]. They use an ant colony optimization approach for solving the problem with time dependent travel times.

3.2.2 Dynamic PDP

We define a PDP as static if all of the orders and driving times are known before the planning is made, and no changes can be made during the execution of the schedule. In many practical cases however the problem is dynamic, meaning that changes can be made to the problem during execution. New orders can be added and orders can be removed or modified. The need for these changes can be caused by vehicles getting stuck in traffic, and arriving late at their destination, or due to missing information at the time the schedule is made. These changes can be resolved in different ways. An option is to compute a static variant of the problem any time a change occurs, while trying to keep most of the old schedule intact. Also all actions that are performed before the moment the change occurs can no longer be modified. Because these new schedules have to be created on the fly, a fast algorithm is required. When adding new orders during execution a fast way to modify the existing solution is by using insertion heuristics to insert these new orders. An example of an insertion heuristic is cheapest insertion, where the orders are inserted at the position in the schedule which increases the fitness value the least.

More literature on the Dynamic PDP and insertion heuristics can be found in Lu and Dessouky [2006],Luo and Schonfeld [2007], Schilde et al. [2011] and Coslovich et al. [2006].

3.3 Parallel Computing

In the last few years the increase of computing power has shifted from higher clock frequencies for each core to more cores in parallel. This means that computer programs written in the traditional way no longer benefit from this increase in speed. It is desirable that an increase in computer cores results in an increase in speed, but this requires a new way of programming. Parallel programming is usually done by dividing the program in separable parts, which can be computed independently of each other. This way of programming is harder because you don't know which thread will finish first, and this uncertainty can introduce new bugs.

Subramanian et al. [2010] have developed a heuristic method for solving the PDP using parallel computing. For computing all the possibilities in the different neighborhoods worker threads can do these computations simultaneously while the master thread keeps track of the best improvement.

Parallel computing is very interesting for speeding up optimization problems on multi core systems. It is bound to get even more important when the number of cores in computers increases in the future. Also parallel computing on GPUs (Graphics Processing Units) could be a major improvement if done well, because GPUs have many cores that could all do their own computations.

The downside of parallel computation is that it is often hard to divide a task into many small tasks that can be run in parallel efficiently. If the algorithm has to wait for other threads to finish their computations often, the speed increase will be nullified.

3.4 Summary

In this section we discussed previous research on a number of methods for solving the pickup and delivery problem. From this research we can conclude that exact methods can be used to solve this problem, however these methods are not suited for solving larger problems, because the computation time increases exponentially with the problem size. Also, more complex problems in terms of constraints would be very hard to solve. Metaheuristic methods which do not guarantee finding an optimal solution can solve much larger problem instances.

We have also discussed literature that covers the pickup and delivery problem with time dependent travel times. This research describes algorithms adapted to solve

these problems, and methods for describing these problem instances.

Based on our findings in literature, we choose to use a simulated annealing approach. This approach is chosen because previous results with simulated annealing are quite good, and the algorithm is easily extended because of its simplicity. This is useful because it allows us to focus on the extensions of the PDPTW, such as time dependent travel time.

Chapter 4

Implementation

In this chapter the algorithm will be explained. First a high level overview is given, followed by more detailed descriptions of the different parts of the algorithm. We start by explaining the initialization process, then the improvement phase and the operators, the extensions to the algorithm, and finally the process of inserting and removing orders.

4.1 Algorithm

The algorithm we created (see Algorithm 1) uses a simulated annealing approach, because of the good results described in previous literature, and the flexibility. It starts by greedily creating a number of initial solutions, using one or multiple of the initialization methods we will describe later, and the best found solution is stored. The best initial solution that was found is improved using several operators that are randomly chosen at every iteration. These operators can use either a small or a large neighborhood. The algorithm starts with an initial *temperature*. This temperature influences the chance of worse solutions to be accepted, a higher temperature means a higher chance of accepting worse solutions. If the new solution has a better fitness value than the old one, the new solution always replaces the old one. The fitness value of a solution represents its quality, based on the objective function described in Chapter 2. The lower the fitness value the better the solution is. If the new solution is worse than the current solution, a chance $P(temperature, distance)$ is calculated based on the current temperature value and the difference between the fitness values of the old and the new solutions. The new solution gets accepted with probability $e^{\frac{-difference}{temperature}}$, where $difference$ is the difference in fitness value between the old and the new solution. Once the number of iterations has been reached we want to reduce the number of vehicles used. At this point there are two scenarios: all orders are planned, or there are still unplanned orders. In the first case we try to decrease the number of vehicles used. To do this the algorithm selects a randomly chosen vehicle and day, with a higher probability to select vehicles which serve fewer orders on that day. This vehicle is set as disabled for that day. The algorithm then returns to the improvement phase and tries to solve the problem with the disabled vehicle as in Ropke and Pisinger [2006]. In the second case, if some of the orders remain

Algorithm 1 An outline of our algorithm

```
 1: procedure ALGORITHM
 2:     overallBest ← empty_solution
 3:     for r=0 ; r<numberOfRuns ; r++ do
 4:         best ← createInitialSolution()
 5:         for i=1;i<initialSolutions;i++ do
 6:             newInitial ← createInitialSolution()
 7:             if newInitial.fitness < best.fitness then
 8:                 best ← newInitial
 9:             end if
10:         end for
11:         current ← best
12:         for m=0 ; m<numberOfMinimizations & t ≤ allowedTime ; m++ do
13:             for i=0 ; i<numberOfIterations & t ≤ allowedTime ; i++ do
14:                 newSolution ← applyRandomOperator(current)
15:                 if newSolution.fitness < current.fitness then
16:                     current ← candidate
17:                     if current.fitness < best.fitness then
18:                         best ← current
19:                     end if
20:                 else
21:                     difference ← candidate.fitness − current.fitness
22:                     if random() < P(temperature, difference) then
23:                         current ← candidate
24:                     end if
25:                 end if
26:                 temperature = temperature · α
27:             end for
28:             current ← best
29:             if |current.unplannedOrders| = 0 then
30:                 current ← removeVehicle(current)
31:             else
32:                 current ← restoreVehicles(current)
33:             end if
34:             current ← removeVehicle(current)
35:             temperature = startingTemperature
36:         end for
37:         best ← hillClimb(best)
38:         if best.fitness < overallBest.fitness then
39:             overallBest ← best
40:         end if
41:     end for
42:     return overallBest
43: end procedure
```

unplanned, all previously disabled vehicles are enabled again. The whole process can be repeated for a number of runs, or until the time limit for the algorithm is exceeded. Every time a solution is found that is better than the overall best solution, the overall best solution gets replaced. Finally this overall best solution is returned.

4.1.1 Solution Representation

The solutions that are generated consist of a number of actions that are planned within their time windows. On the highest level there are schedules for the vehicles, which consists of one or more routes. These routes are sequences of actions. A route can be divided into a number of shifts if it visits the depot within a route. Every subsequence of a route that starts and ends at a depot is a shift, so a route always consists of one or more shifts. Figure 4.1 illustrates this structure.

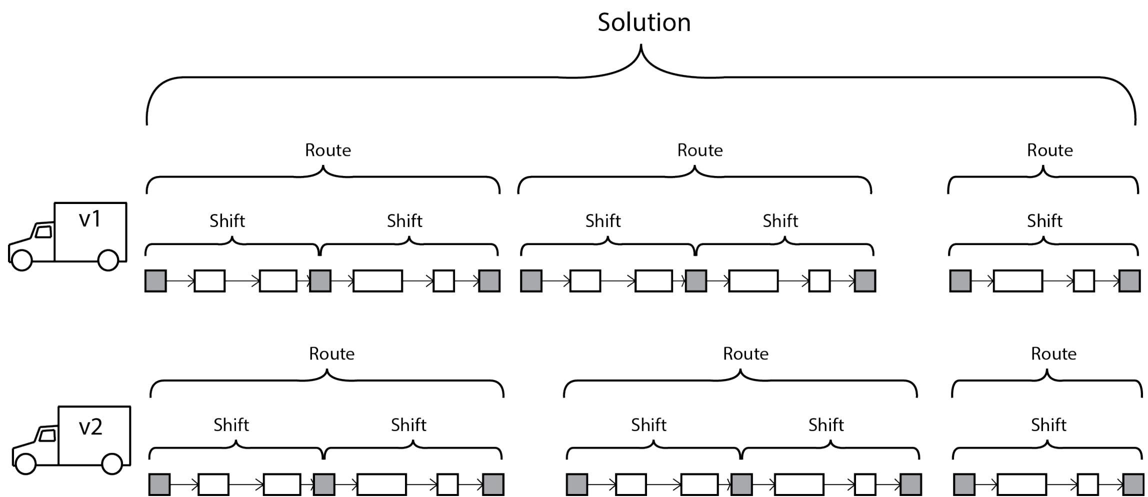

Figure 4.1: Representation of a solution with two vehicles. In this illustration the boxes are actions, and the gray boxes represent actions in which the depot is visited. The width of the boxes indicates the service time of the actions, and the length of the arrows indicate the travel time. For every time window $w \in W_v$ the vehicle can have a route assigned. A route can, for example, represent the schedule for a vehicle for a single day.

4.1.2 Similarity

Before the different parts of the algorithm are described in more detail, the concept of similarity has to be introduced first, because it is used throughout the algorithm. Similarity is a value, based on time and distance, used to describe how much orders, actions or groups of orders are alike. This similarity value is used to guide the algorithm; Orders that have a high similarity are more likely to produce a valid solution if they are swapped. Because of the time window restrictions many of the orders

that have low similarity have little influence on each other. Using this similarity factor in the operators increases the success rate. The concept of similarity is based on the *relatedness function* introduced by Shaw in Shaw [1997]. Too much guidance could lead to parts of the search space being skipped, and possibly good solutions not being found. This is why we do not always use similarity in the operators. Some operators, like combining two trips into one, always use similarity. Other operators use similarity with a probability, so that it does not limit the possibility to reach all parts of the search space.

Similarity between actions

We define the similarity factor $SF(a, b)$ between actions a and b as a combination of $timeSimilarity(a, b)$ and $locationSimilarity(a, b)$. To define the time similarity there are two possible scenarios. Either both orders are planned or only one of the orders is planned. When both of the orders are unplanned there is no need to calculate similarity because they have no influence on the insertion possibilities. The formula for calculating the time similarity value for both scenarios is given in Equation 4.1.

$$timeSimilarity(a, b) = \begin{cases} 1 - \frac{|start_a - start_b|}{maxTimeDifference} & \text{if both actions are planned} \\ 1 - \frac{|start_a - closestStart(b, start_a)|}{maxTimeDifference} & \text{if only action a is planned} \end{cases} \tag{4.1}$$

Here $maxTimeDifference$ is the largest time difference possible given the set of orders: $closestStart(b, start_a)$ is the time value closest to $start_a$ that the start time of b can attain in its time window.

The location similarity depends on the distance of the two locations from each other. The location similarity value can be calculated as in Equation 4.2.

$$locationSimilarity(a, b) = 1 - \frac{distance(a, b)}{maxDistance} \tag{4.2}$$

Here $maxDistance$ is the largest distance between any two actions in the problem instance.

The total similarity between actions a and b is the product of the time similarity and the distance similarity, as given in Equation 4.3.

$$SF(a, b) = timeSimilarity(a, b) \cdot locationSimilarity(a, b) \tag{4.3}$$

Similarity between orders

Using the similarity values for actions we can define a similarity value between two orders. We do this by taking the average of the similarity for the pickup action and the delivery action of both orders. When an order has more than one pickup or delivery alternative we use the values for the actions that are currently planned, because these are the ones that influence the solution. If one of the orders is not planned, we use the pickup action and delivery action from that unplanned order that have the highest similarity to the planned actions of the other order. The actions with highest similarity value are picked because it is most likely that these are the actions that will influence each other when making a change to the solution.

Similarity between routes

Similarity between routes can be defined as the average of similarity between the combinations of all orders in both routes. This can be used when trying to find two similar routes to merge into one.

4.1.3 Initialization

The first step of our algorithm is to generate an initial solution. The initial solution serves as a basis that can be improved in the second phase. This approach should be able to quickly generate a feasible solution. For generating initial solutions there are a few different approaches: Random insertion, Farthest insertion, Nearest insertion, Insertion by similarity, Sweep insertion and Cheapest insertion. The aim of these different initialization methods, which we will describe in this section, is to be able to create a wide variety of starting solutions.

The easiest way to create a feasible initial solution is starting with an **empty solution**, a solution where none of the orders are planned. With an empty start the simulated annealing algorithm does all the work. The other methods try to insert as many orders as possible, even if the penalty of inserting this order is extremely high. When creating a planning with maximum shift times this method, together with the hillclimbing method, can be preferable over the other methods. With maximum shift times it is often desirable to create an initial solution that does not exceed the maximum shift times by too much.

Random insertion is a fast initialization method. This method selects all the orders in random order and tries to plan them at their cheapest possible position at that moment. The cost of an insertion is based on the contribution to the fitness value of the solution. The worst-case running time of this method is $\mathrm{O}\big(o \cdot (max_n \cdot (n+s))\big)$, where n is the number of order actions, o is the number of orders and s is the number of shifts. An order usually has two order actions, but there can be more possibilities if alterative pickups or deliveries are given. For every order action the maximum number of possible insertion locations is $\mathrm{O}\big(max_n \cdot (s+n)\big)$, where max_n is the maximum number of order actions an order has in this problem

instance. This is because an order action can be inserted in an empty shift, or after every already inserted order action, of which there can be only two per order, and . The advantage of this method is that a wide variety of solutions can be created with the same method in a short amount of time by permuting the initial order.

The **Hillclimbing** method is like random insertion, but only if an insertion improves the solution it gets accepted. This method is useful for generating an initial solution with our implementation of maximum shift time; insertions are not accepted if they cause the duration of a shift to become too high.

Farthest insertion selects the order that has the highest minimum driving time from any of the depots, and inserts it in the best possible place. The driving time is chosen because this is a constraining factor in planning, and distance is not. The advantage of farthest insertion is that orders which are probably hard to plan because they take a long time to reach, are planned first, giving them a higher chance to be able to be planned. Before planning the orders have to be sorted by distance from the depots, and are inserted from farthest order to closest order. The running time is the same as with random insertion, because all the distances to the depots can be calculated once before the insertions start. The **Nearest Insertion** method is the same as Farthest Insertion, but instead of the farthest orders, the nearest orders are selected first.

Insertion by similarity is an initialization method we have developed that uses the similarity value between orders. To the best of our knowledge, this method has not been researched before. The first action is to select a random order. We insert this order at its cheapest insertion position. For the selection of the next order we sort all orders by similarity and select the most similar order to the one we have inserted last. This most similar order is also inserted at its cheapest insertion position. We continue this process until all orders are planned, or no more orders can be inserted without violating the hard constraints. The running time of this method is $\mathrm{O}\big(o \cdot ((o \cdot max_n^2) + max_n \cdot (n+s))\big)$, because for every order that we planned we look for an unplanned order which has highest similarity to the last planned order. We plan o orders, for which we have to do a similarity check on $\mathrm{O}\big(o\big)$ other orders. The similarity check is done in $\mathrm{O}\big(max_n^2\big)$ time, because the similarity between all actions of the two orders are checked.

The **Sweep Method**, based on the method proposed by Gillett and Miller [1974], first clusters all the orders around their nearest depots. Once the orders are clustered we start solving the initialization for all of the depots. For every depot D an extra coordinate A, at a different coordinate than D, is created. This extra coordinate is used to sort the orders that were assigned to this depot by angle. The location of the order is O. Then the angle $\angle ADO$ is calculated for every order, and the orders are sorted by increasing angle. The solutions are then created by selecting the orders that have the smallest angle first and inserting these orders at the best possible location. This creates solutions where all the routes handle a different side

of the area. The method is called the sweep method, because the area surrounding the depots is 'swept' in clockwise or counterclockwise direction, based on angle. The sweep method has the same worst-case running time as random insertion. The only difference for the running time is that all angles have to be calculated before the actual insertions start, but this does not influence the worst-case running time.

The **Cheapest Insertion** method calculates the cheapest insertion over a number of orders for every iteration, and inserts this order. The running time of this method is high because instead of just inserting one order at every iteration this algorithm inspects a number of orders before inserting one. If we calculate the cheapest insertions for all orders at every iteration, the worst case running time would be $O\big(o^2 \cdot (max_n \cdot (n + s))\big)$. We can reduce the running time by selecting a random set of orders with size *selectionSize* for which to calculate the cheapest insertions at every iteration, instead of doing this for all orders. This would reduce the worst-case running time to $O\big(o \cdot selectionSize \cdot (max_n \cdot (n + s))\big)$, but it would not guarantee that the cheapest insertion is chosen.

4.1.4 Improvement phase

The improvement phase starts by selecting the best result generated by the initialization phase. It then continues to improve this solution using a slightly modified version of the simulated annealing (see Kirkpatrick et al. [1983]) meta-heuristic.

Simulated annealing is based on the annealing process of metals, where metals are heated and then cooled in a controlled fashion to reduce defects in the structure. The simulated annealing algorithm works in a similar fashion, but instead of decreasing the freedom of movement of atoms over time, the freedom to move through the search space is reduced, until the algorithm reaches a stable state. The difference between the normal simulated annealing algorithm and our implementation is that we added some guidance towards better modifications. Also we introduced a number of large neighborhood operators to decrease the chance of getting stuck in a local optimum.

The pseudo code for the simulated annealing algorithm is given in Algorithm 2. Here *best* is the best solution found so far; *temperature* is the current temperature of the solution, which is decreased by multiplying it with a factor α after every iteration; *candidate* is the new solution found by applying a random operator; $solution.fitness$ is the fitness of *solution*, and *random*() generates a random number between 0 and 1.

For a number of iterations neighbors are selected from the neighborhood spaces defined by the operators we will describe later. The algorithm starts with a high temperature which provides a lot of freedom to move around in the search space. The temperature determines the probability of accepting new solutions that are worse than the current one. A higher temperature means a higher probability to accept

Algorithm 2 Simulated Annealing

```
1: procedure SIMULATED ANNEALING
2:     current ← createInitialSolution()
3:     best ← current
4:     temperature ←starting temperature
5:     while end condition not met do
6:         candidate ←random neighbouring solution to current
7:         if candidate.fitness < best.fitness then
8:             best ← candidate
9:             current ← candidate
10:        else
11:            if random() < P(temperature, candidate.fitness - best.fitness) then
12:                current ← candidate
13:            end if
14:        end if
15:        temperature ← temperature · α
16:    end while
17:    return best
18: end procedure
```

a solution that is worse than the current one. As time goes on the temperature is decreased gradually. This means that the solution becomes increasingly stable, until it reaches an equilibrium where it no longer changes. The reason worse solutions are allowed is to escape from local optima.

To move through the search space a number of operators are defined. All of these operators modify the solution in some way. The *neighborhood space* for an operator is defined as the set containing all solutions that can be reached by applying this operator. In our algorithm we have defined a number of fast operators which make small scale changes, these could for example move one order to another location in the planning. In addition to these small scale operators there are a number of operators that make changes on a larger scale, the large neighborhood operators. These can remove and reinsert a large number of orders in one iteration. All of the operators will be explained in detail in Section 4.2. In every iteration one of the operators is chosen at random, with a predetermined probability for selecting each operator. First the algorithm makes a choice between applying a small scale or large scale operator, which both have a predefined probability of being selected. Then it selects one of the operators in the chosen group at random. This operator is applied to the solution. The new, modified solution is then compared to the old solution to see if it gets accepted or rejected. The large neighborhood operators are introduced because we reason that due to the larger changes that are made, the algorithm can escape local optima that it would not be able to get out of by only using small scale operators. In the experiments and results chapter we experiment with the different

operators and determine good probabilities for selecting operators. We will also analyze the influence of the large neighborhood operators.

Temperature Values

If the new solution is better than the old one it is always accepted. If it is worse there is a probability that the solution is accepted. This acceptance probability is based on the difference between the fitness of the old and the new solution and the current temperature, and is defined in Equation 4.4. Here $difference$ is the difference between the fitness value of the old and the new solution. The probability function always provides a probability value between 0 and 1.

$$P(difference, termperature) = e^{\frac{-difference}{temperature}} \tag{4.4}$$

A starting and final temperature are chosen before running the algorithm. The value for α to cool the starting temperature down to the final temperature is calculated with Equation 4.5, where T_e is the desired final temperature and T_0 is the initial temperature.

$$\alpha = \frac{T_e}{T_0}^{\frac{1}{iterations}} \tag{4.5}$$

Temperature settings have a large influence on the quality of the produced solutions. The ideal values for the start and end temperature depend on the instance size, the penalty settings and the distances between the locations. In previous literature there are a number of methods defined for calculating good starting temperatures for different instances. We based our implementation on the method defined by Johnson et al. [1989]. For calculating a good initial temperature they define the acceptance ratio X of a given temperature for a certain solution. This acceptance ratio is the percentage of solutions which are worse than the current one, that are accepted. The starting temperature T_0 for a given initial acceptance ratio X_0 is defined in Equation 4.6.

$$T_0 = -\frac{\overline{\Delta E}}{ln(X_0)} \tag{4.6}$$

$\overline{\Delta E}$ is the expected change in energy, the equivalent of fitness in our approach. This expected change is based on a number of randomly selected modifications to the solution, where only modifications that make the solution worse are considered, because the acceptance rate of better solutions is 100%. For all these modifications the change in fitness is added to a list L, with values L_1 to L_k. The average acceptance probability over all the fitness differences in this list has to be equal to the initial acceptance rate X_0. We do this by solving Equation 4.7. Since $e^{\frac{-difference}{T}}$ is the probability a new solution that has a fitness value of $difference$ higher than

the current solution's fitness value gets accepted when the current temperature is T, the equation we want to solve is that the average of accepting a value from ΔE is equal to X_0. The larger the set L is, the more accurate the prediction for acceptance probability gets. In the experiments and results section an experiment will be discussed for finding good acceptance rates.

$$\frac{\sum_{i=0}^{i<k} e^{\frac{-L_i}{T_0}}}{k} = X_0 \tag{4.7}$$

At the initialization phase of the algorithm the previously described method can be used to determine good temperature values for different problems. To do this the set L will be generated by applying a number of operators to an initial solution.

Minimizing vehicles

Because we found that the algorithm was not good enough at minimizing the number of vehicles used we have added a mechanism, based on the approach of Ropke and Pisinger [2006], to the algorithm that handles vehicle minimization. Once the desired number of iterations for the simulated annealing has been reached the algorithm removes a random vehicle from the solution for a random day. The chance for the vehicles to be removed depends on how busy the schedule for this vehicle is. The less there is planned for a vehicle on a given day the larger the chance that it can be successfully removed. The thought behind this is that the orders of vehicles that don't have much to do can possibly be redistributed among other vehicles more easily. Once a vehicle is disabled the simulated annealing algorithm runs again and tries to plan all of the orders without the disabled vehicle. If not all orders are planned when the simulated annealing phase ends the removal of a vehicle is postponed. Also if the algorithm was unable to find a solution in the given time, after disabling a vehicle, the disabled vehicle is enabled again and the simulated annealing is ran again.

4.2 Operators

To be able to improve the solution, operators that define the neighborhood spaces have to be defined. Every operator has its own neighborhood. The goal of these operators is that when combined they make it possible to reach all of the solutions in the search space. In our algorithm there are two kinds of operators: The small neighborhood operators and the large neighborhood operators.

4.2.1 Small neighborhood operators

The small neighborhood operators are the foundation of our algorithm. These operators are fast because they only examine a small part of the search space. Because

they are fast they can make a lot of changes to the solution in a short period of time. This makes it possible to quickly improve the initial solution. We will describe all of the small neighborhood operators we use below.

Simple Pair Shift Operator

The most basic operator we use is *simple pair shift*. This operator selects an order at random. If the order is contained in the current solution, both its pickup action and delivery action are removed. From the set of all possible insertions for this order, an insertion is chosen based on its insertion cost; cheaper insertions have a higher probability of being chosen. Unplanned orders can also be selected, the algorithm will then try to insert these orders anywhere if possible, using the insertion probabilities.

Time Shift Operator

Inserting, removing or shifting orders in the current solution can cause gaps of waiting time inside shifts to be created. The solution may be improved by optimizing the starting times of the planned actions. The goal of time shift is to optimize the time for one shift. First a random shift is selected. When selecting a shift we prefer shifts that have a lot of waiting time, because these are the shifts that are most likely to improve most by optimizing the time. We try to make the selected shift as short as possible while we also include the effects of the new starting time of the shift on other shifts. To optimize the time penalty we select the first action of the shift and try to plan it with a number of different starting times that are possible for this action. The actions following the first action are moved backwards in time as far as possible without shifting the first action of the shift. This is done because the duration of a shift is minimized if the end time of a shift is as close as possible to the starting time of that shift. The best of all selected starting times is selected, and the start time of the shift is set to the selected starting time.

Swap Operator

The swap operator tries to find the best swap for a random order. First a random, planned or unplanned, order o_1 is selected. Then a list of all the orders this order can be swapped with is made. The swap is possible if both the pickups and the deliveries of o_1 and o_2 can be swapped. The best possible swap for o_1 is executed, if any swaps are possible.

Unplanned Insertion Operator

Unplanned orders have a very high penalty contribution. When an unplanned order can be planned it will probably lead to a large decrease in total penalty. For this reason, when there are unplanned orders, planning them has high priority. First a random order is selected from the list of all unplanned orders. Then we look for insertion possibilities for this order in the current planning. If an insertion is possible

we plan the order. If no insertion is possible, we will try to swap the unplanned order with any of the planned orders. After a swap is made the order that is swapped with the unplanned order is inserted again on the best possible position in the planning. If a swap is not possible either, we remove a number of orders. Then we try to insert the unplanned order first. After the unplanned order is inserted we try to reinsert the removed orders again.

Exchange Operator

When there is already a tight schedule it may be useful to create space to be able to move orders to other locations. The exchange operator selects a random non-empty route and removes a random small number of orders (in our experiments 2 to 5) from those routes. Then it selects a second route with a 75 percent chance to choose a route that is similar to this route, and it removes the orders from that route as well. It then tries to reinsert the orders that are removed from route 1 in route 2 and vice versa. For all orders that could not be inserted in the other route, the operator tries to insert those orders somewhere else in the planning. The insertion is done in the same way as with the simple pair shift operator. Due to the larger number of orders considered, the exchange operator could also be categorized as a Large Neighborhood Operator. We chose not to do this, because the neighborhood is not as large as with the ruin/recreate or route combination operators.

Consecutive Exchange Operator

The consecutive exchange operator is almost exactly the same as the normal exchange operator, but instead of removing random orders from route 1 and route 2, a small number of consecutive actions are selected. The orders to which those actions belong are removed from the planning, and reinserted in the same fashion as with the normal exchange operator.

4.2.2 Large neighborhood operators

For complex problems like the PDPTW we are solving, operators that make small changes to the solution can be unable to reach the best solutions in the search space Schrimpf et al. [2000]. For example, moving a single order to another route may increase the cost of the solution, while moving all orders in the route simultaneously could lead to a much better solution. These operators change a larger part of the solution. This may create the possibility to move to a better solution that was unreachable by the small neighborhood operators. The larger neighborhoods are not searched completely: heuristics are used to be able to execute these operators in a reasonable amount of time.

Ruin/Recreate

This operator is based on the operators described by Ropke and Pisinger [2006]. It removes up to n orders (where n is 20 in our implementation) from the solution

and reinserts them at new locations. This operator can use different removal and reinsertion mechanisms.

Removal methods : The removal methods select up to n orders to remove. The possible methods are listed here.

Random : Random removal is the simplest removal algorithm. It just removes n randomly selected orders.

Worst : Worst removal removes the orders that add most to the penalty value of the current solution. We calculate the penalty contribution of these orders by removing them from the solution and subtracting the old penalty value from the new one. The reason we select the most expensive orders is because these orders are more likely to be planned in a bad place.

Similar : When removing orders it would be useful if the orders can be swapped. We select one random order. Then we select $n - 1$ orders by similarity to the previously selected order.

Insertion methods : The insertion methods insert the removed orders into the solution again. These are the possibilities we consider.

Random insertion : The fastest way to insert the removed orders. We reinsert the orders one by one. Every order gets inserted at its best position at that moment.

Insertion by similarity : Similar insertion inserts one random order from the removed orders first on its cheapest insertion position. Then it inserts the order that is most similar to the order that was inserted first, and inserts this as well. This process continues until all removed orders are planned.

Greedy insertion : Greedy insertion inserts all the removed orders in a greedy fashion. There are two possibilities for the greedy insertion method, cheapest greedy and most expensive greedy. It calculates the best insertion positions for all the removed orders and selects the cheapest overall insertion, or the most expensive overall insertion from these insertions, based on which method is chosen. The process is repeated for all the removed orders that are not yet reinserted.

Regret insertion : The regret method is a little more sophisticated. For all the removed orders it checks the best insertion possibility in every route. It remembers the best and second best insertion. The orders are then sorted by difference between the best insertion value and the second best insertion value, the so called 'regret' value. The one with the biggest 'regret value' is inserted. The idea behind this method is that if an order with a high regret value can not be inserted at its best position anymore the loss in quality is high, because the difference between the best and second best insertion position is high.

Route Combination Operators

There are three operators that try to combine or remove routes. This can be useful when trying to reduce the number of vehicles used. The *combine route* operator tries to combine two routes into one by removing two randomly chosen routes. The orders that were in these routes are then reinserted. When the two routes fit into one the operator will combine the two routes, otherwise it will plan as many orders as possible in one route and try to plan the remaining unplanned orders at their best possible positions.

The *combine route remove N* operator tries to merge a route with similar orders. A random route is selected, and for that route a number of similar orders are removed. Removing these similar orders creates space for the orders in the selected route to be replanned. All orders in the selected route are planned at their new best position, and afterwards the removed orders are reinserted again if possible.

For problems with heterogenous vehicles there is need for a method that can combine two routes that are scheduled for low-capacity vehicles into one route for a high-capacity vehicle. Without such a method, small vehicles with low cost would always be preferred over larger more expensive vehicles, because the direct impact on the fitness value is lower with the small vehicles. The *combine routes into new* operator is the same as the combine route operator, with the difference that it tries to assign the orders in both routes to a new vehicle instead of one of the vehicles of the removed routes.

2-Opt* Operator

The 2-opt* operator is a 2-opt operator that is modified to work with the constraints of our problem. We based this operator on the modified 2-opt by Potvin and Rousseau [1995]. They modified the normal 2-opt operator to be able to cope with time windows. The standard 2-opt operator selects a number of subsequent locations in the route and reinserts them in reversed order.

Algorithm 3 The standard 2-opt procedure

1: **procedure** 2-OPT($route, i, k$)
2: $newRoute \leftarrow$ empty route
3: add $route[0]$ to $route[i-1]$ to $newRoute$
4: add $route[i]$ to $route[k]$ to $newRoute$ in reversed order
5: add $route[k+1]$ to the end of the route and add them to $newRoute$
6: return $newRoute$
7: **end procedure**

Reversing a part of a route in a routing problem with time windows using the standard operator however is likely to lead to an infeasible solution. Potvin and Rousseau created a modified 2-opt where the orientation of the original route is

preserved. Their operator takes two routes as input, we call these routes r_1 and r_2. One edge is chosen from r_1 and one edge is chosen from r_2. The routes are cut at these edges, creating routes existing of two parts. The two new routes new_1 and new_2 are created: new_1 exists of the first part of r_2 and the second part of r_1, new_2 exists of the first part of r_1 and the second part of r_2. With this new operator the chance of the new solution being feasible is much larger, especially when two routes are selected which are similar to one another.

In our problem however we also have to deal with precedence constraints imposed by the pickups and deliveries. Pickups are always served before their corresponding delivery. We extended the 2-opt* to also take this constraint into account. First, two routes are selected, with a 75% chance to select the second route by similarity to the first. All the delivery actions are removed from these routes. Then two new routes are created by applying Potvin and Rousseau's 2-opt* to these routes. Finally the delivery actions that were removed from the original routes are reinserted again. We start by inserting the delivery action belonging to the last delivery action, and then work backwards until all deliveries have been inserted. If a delivery can not be inserted in the new route, we also remove the pickup, because a valid solution can not have a planned pickup without its corresponding delivery being planned.

In our operator instead of the usual 2-opt many possibilities are examined instead of only one. Two routes are selected, from which one has to be non-empty. For both of these routes we select a number of edges, the split points. We apply the adapted 2-opt* to these routes with all combinations of the selected split points, and execute the best one. The goal of this operator is to find the best splitting points for the two routes to combine them to new routes. This is done by trying a large number of possibilities and finally selecting, and executing the best.

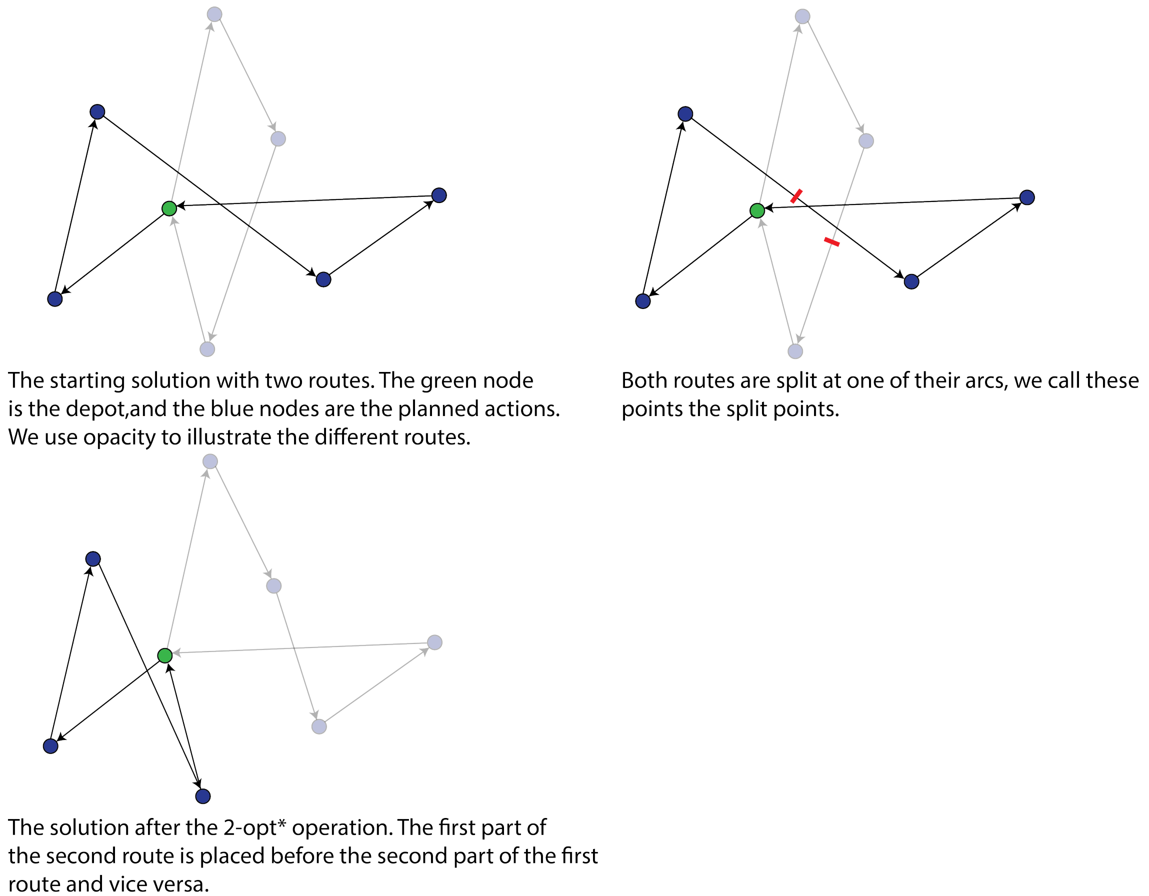

Figure 4.2: The 2-opt* operator.

4.3 Time Dependency

4.3.1 Travel time without time dependency

Calculation of travel time without time dependency is done using a *location matrix*. The location matrix contains all the distances and travel times between the locations. The travel time between two locations can be found by looking at the corresponding cell in the matrix. When there are n locations in the problem the location matrix has a size of n^2. To save on this we group order nodes by location: if there are multiple nodes that correspond to the same location, the location only has to be in the matrix once.

4.3.2 Travel time with time dependency

When working with time dependent travel times the algorithm works with an input consisting of discrete travel time values per time unit, for all pairs of locations. The only restriction is that these travel times have to follow the FIFO (First In First Out) principle, meaning that when a vehicle leaves a location earlier it will always arrive earlier or at the same time. For example, when a vehicle arrives at time *at*

when departing at time t it will arrive at a time $at' \leq at$ when departing at time $t' \leq t$. The input can consist of discrete travel time values per time unit for every pair of locations, or it can be calculated when it is needed using a fast algorithm.

4.3.3 Calculation of travel time using periods

To calculate the time dependent travel times different methods can be used. To save all travel times in a 3d matrix would require large amounts of memory, for n locations and a planning window of w time units we would have a location matrix of size $n^2 \cdot w$.

Instead of defining all travel time values exactly per time unit, we work with periods of time, which can be used to calculate all travel time values. A default travel time is given for every pair of locations (a, b). The time periods indicate periods of time in which the travel time differs from the default value. The value assigned to a time period for locations (a, b) indicates the time it would take to get from location a to location b, if the whole trip can be finished within the same time period. If the destination is not reached within the same time period, interpolation is applied (as illustrated in Figure 4.3.3). Travel times always follow the FIFO property due to the way this interpolation is done, no matter which values are given for the time periods.

We calculate the travel time for traveling from location a to location b when departing at time t using the procedure given in Algorithm 4. We start by finding the starting time and the end time of the period t is in: *periodStart* and *periodEnd*. This time period has a travel time, $currentTravelTime$ assigned to it, which is the time it would take to reach the destination if the whole trip could be finished in this period. The total travel time up to the current moment is $travelTime$, and $distanceCovered$ is the total distance covered so far. Until 100% of the drive is finished, and the destination is reached, we repeat the process of adding the progress towards the destination trough all the periods we go through. Say that for example we leave at location a at time 110 with the time periods given in Figure 4.3.3. Since period $T1$ starts at time 0 and ends at time 150, there are 40 time units left in period $T1$. Because the travel time in this period is 50 time units, 80% of the trip can be finished within the time that was left. The remaining 20% will be driven in time period $T2$, where the driving time is 75 time units. The remaining part will then be driven in 15 time units, giving a total travel time of 65 time units.

This method always respects the FIFO property. Going into a new time period in which it takes longer to reach the destination does not matter for the FIFO property, because there is no restriction on the delay imposed by going forward in time. When leaving earlier though it should not be possible to arrive at a later time. Say a time period $p1$ with very short travel times starts at time t. The period before it $p0$ has very high travel time. Leaving at time $t' \leq t$ should never cause the vehicle to arrive later than leaving at time t. We can prove that the FIFO property always holds.

Algorithm 4 Calculation of travel time between locations a and b with time dependency

```
1: procedure CALCULATETRAVELTIME(a, b, t)
2:     travelTime ← 0
3:     distance ← distance(a, b)
4:     distanceCovered ← 0
5:     periodStart ← previousTimePoint(t)
6:     periodEnd ← nextTimePoint(t)
7:     currentTravelTime ← currentTravelTime(periodStart)
8:     if periodEnd - t > currentTravelTime then
9:         return currentTravelTime
10:    else
11:        travelTime ← periodEnd − t
12:        while distanceCovered < distance do
13:            periodStart ← periodEnd
14:            periodEnd ← nextTimePoint(periodEnd)
15:            currentTravelTime ← currentTravelTime(periodStart)
16:            travelTimeLeft ← (1 − distanceCovered/distance) · periodStart
17:            periodDuration ← periodEnd − periodStart
18:            if travelTimeLeft ≤ periodDuration then
19:                return ⌈travelTime + secondsNeeded⌉
20:            else
21:                travelTime ← travelTime + periodDuration
22:                distanceCovered ← periodDuration/travelTimeLeft · (distance − distanceCovered)
23:            end if
24:        end while
25:    end if
26:    return ⌈travelTime⌉
27: end procedure
```

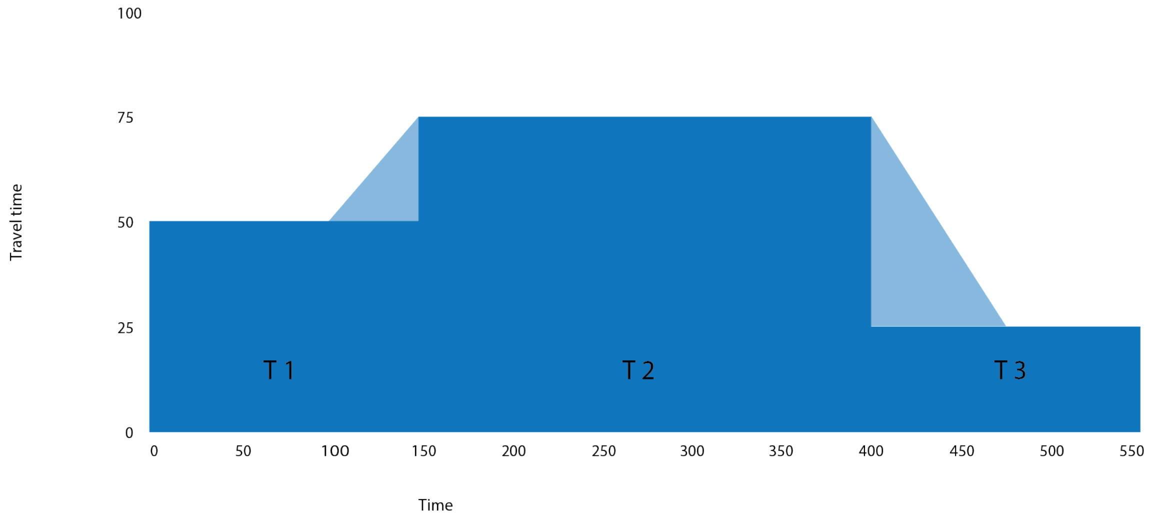

Figure 4.3: Representation of calculating travel time between a pair of locations (a, b), using interpolation between periods. There are three periods: T1, T2 and T3. The dark blue rectangles represent the defined travel time, and the light blue parts represent the interpolated travel time in between periods. On the y-axis the travel time is given, and on the x-axis the current time is given.

Theorem 4.3.1. *The arrival time at location b when leaving location a at time $t-1$ is never larger than the arrival time when leaving at time t.*

Proof.
Say we have two vehicles, v_1 and v_2, travelling from location a to location b. Vehicle v_1 departs from location a at time t, and vehicle v_2 departs from the same location at time $t' \geq t$. Say that vehicle v_2 takes the fastest route r to reach its destination b. If vehicle v_1 would take that same route it would arrive at b earlier than or at the same time as v_2, because it departs earlier. If vehicle v_2 catches up with vehicle v_1 somewhere in the route they would be at the same place at the same time, which by definition (see Algorithm 4) leads to both vehicles arriving at location b at the same time. Vehicle v_1 however drives its fastest route r' to reach b, and this route may be different from r. But r' can never take longer than r, because if this would be the case r' would not be the fastest route. This means that vehicle v_1 always arrives earlier than or at the same time as vehicle v_2. This proves that if a vehicle drives from a location a to a location b, and it departs earlier than another vehicle, it will always arrive at location b earlier than or at the same time as the vehicle that departs later. This means that the FIFO property is respected. □

4.4 Maximum shift time

We described the concept of a shift before as a part of a route that starts and ends at the depot. Shifts are introduced to be able to create routes that respect the maximum working times of drivers. A shift is a subsequence of actions in a route, which starts and ends at a depot node. In this problem, all drivers have the same

maximum working time. If a maximum time for shifts is defined we have to split routes that last too long into parts. The routes are divided into multiple shifts by adding a number of *shift start actions* to the routes. These actions represent the start of a new shift, and a visit to the depot to exchange drivers. The number of shift start actions that are added to a route r is $\lceil \frac{re_r - rs_r}{maxshifttime} \rceil - 1 + e$, where re_r is the end time of route r and rs_r is the start time of route r. This is the number of actions that is necessary to divide the route into shifts that are shorter than or equal to $maxshifttime$; e is the number of extra shifts that are added. A higher value for e means that there can be more shifts, which gives the algorithm more possibilities for distributing the actions among shifts. More possibilities however also means that the computing time increases, and a value of e that is too high implies that there are probably many empty shifts. Whether these shift start actions are used depends on the actions planned before and after it. If there are no orders planned in a shift, the shift is not actually performed because there is no need to have a driver for a shift with no actions.

To make sure there is no unnecessary waiting time the duration of the shift start actions is flexible. There is never waiting time before or after any visit to the depot. We let the vehicles depart from the depot so that they arrive exactly on time at their next location. Also when arriving at the depot earlier than the action is planned, this excess time is not counted as waiting time, because the driver's shift is over at the time he arrives at the depot, and the next shift starts when the next driver enters the vehicle.

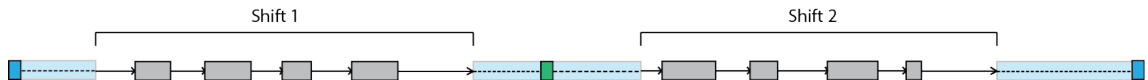

Figure 4.4: An example of a route with two shifts. The blue rectangles represent the start and end of the route, and therefore also the start of the first shift, and the end of the second shift. The green rectangle is a shift start action, this is where the first shift ends and the second shift begins. The solid arrows indicate travel time, whereas the dotted lines are waiting time. The transparent blue area indicates the use of the flexible duration of the depot actions. The duration of a depot action is stretched to remove all unnecessary waiting time. For example say we have a shift that is finished at 11:00 am, and a depot action that is scheduled to start at 12:00 am. The waiting time gap of one hour is then filled by extending the depot action to start at 11:00. The actual duration of the shift is indicated by the brackets on top of the representation.

This approach with a fixed number of Shift Start Actions can only be used when we have a finite route time. But since it is unlikely to make a planning with an unlimited planning window this approach can be used for most problem instances. If there would be an infinite planning window, an operator for adding or removing shift start actions would have to be introduced.

4.5 Inserting Orders

Because of the many constraints on our problem, the usual approach of simulated annealing where operators make random modifications to the solution, is not very effective. Many of the modifications will fail. That is why we have added some guidance to the operators. Because of this guidance we need to have more information on the insertions that are possible: the cost of insertion and which insertions are possible, and which are not.

4.5.1 Slack Values

Before we can explain how order insertions are found we need to introduce the concept of slack. An essential part of finding out where actions can be inserted is finding the earliest and latest insertion times of that particular action, for a given route. These earliest and latest insertion times depend on the time window of the action, the route it is planned in, the extra travel time that is needed when this action is inserted, and the actions that are planned before and after it. We use slack values to quickly determine if an action can be inserted on a particular location in a route.

Normal slack

When solving a problem instance without time dependent travel times, we use slack values based on the method used by Savelsbergh [1992]. All of the actions get a forward slack and backward slack value. These values indicate by how much the action can be shifted forwards or backwards respectively, without rendering the solution infeasible due to time constraints being violated. The forward slack FS_i for an action i can be defined through a recursive function:

$$FS_i = \begin{cases} min\{start_{i+1} + FS_{i+1} - (start_i + serviceTime_i + travelTime(i, i+1)), & \text{if i is not the} \\ twe_i - (start_i + serviceTime_i)\} = & \text{last action in} \\ min\{\texttt{maximum amount of time i can be shifted forward to} & \text{the route} \\ \texttt{still be able to service i+1 in time}, twe_i - (start_i + serviceTime_i)\} & \\ & \\ twe_i - (start_i + serviceTime_i) & \text{otherwise} \end{cases}$$

In this function i is the index of an action in the route. Furthermore $start_i$ is the start time of the action at i, $travelTime(i, j)$ is the travel time needed to get from the location of action i to j, $serviceTime_i$ is the time needed for servicing the action at i and tws_i and twe_i indicate the start and end time of the time window for the action at position i.

For all of the actions the amount they can be shifted forwards (their forward slack) depends on their time window, and the actions that are scheduled after an action in the same route. If an action is the last action in the route its forward slack only depends on its time window: The action can be moved to the end of its time window. If there are other actions planned after the action at i, the forward slack

depends on the starting time and forward slack of the next action, and its own time window. The action at i can be shifted towards the end of its own time window, until the action at $i+1$ can not shift any further. We only need to look at the next action because of the recursiveness of FS_i; the forward slack for an action i takes into account the forward slack of the following actions.

In the approach of Savelsbergh every action is planned as early as possible. In our approach we can plan actions anywhere in their time window. This makes it necessary to also define the backward slack of the actions.

$$BS_i = \begin{cases} min\{start_i - (start_{i-1} + serviceTime_{i-1} + travelTime(i-1,i)) - BS_{i-1}, & \text{if i is not the} \\ start_i - tws_i\} = & \text{first action in} \\ min\{\texttt{maximum amount of time i can be shifted backwards to} & \text{the route} \\ \texttt{without forcing i-1 to start too early}, twe_i - (start_i + serviceTime_i)\} & \\ & \\ start_i - tws_i & \text{otherwise} \end{cases}$$

All of the variables that are used in this function are the same as in the previous function. The backward slack function works in the same fashion as the forward slack function, but in a reversed order.

Every time an action is inserted, removed or had its starting time changed we update all the slack values of the other actions in the route.

Time dependent slack

For instances that use time dependent travel time, we use a method that works a little different than the method without time dependency, but the goal is still to be able to determine the possibility of an insertion by only looking at the previous and the next action. This method does not use forward slack and backward slack, but instead remembers the earliest and latest starting time for this action with the current ordering of the route, EST_i and LST_i for every action. Donati et al. [2008] use a method that is largely similar to this method. When we plan every action as early as possible, starting from the first action in the route, the starting times of the actions are the earliest starting times. We depart from every location as soon as possible. For every action i that is planned the earliest starting time is:

$$EST_i = \begin{cases} max\{EST_{i-1} + serviceTime_{i-1} + & \text{if i is not the first} \\ travelTime(i-1, i, EST_{i-1} + serviceTime_{i-1}), tws_i\} = & \text{action in the route} \\ max\{\texttt{earliest time it is possible to reach the location} & \\ \texttt{of i after service is finished at i-1}, tws_i\} & \\ & \\ tws_i & \text{otherwise} \end{cases}$$

This is true because travel times work according to the FIFO principle; planning an action later than its earliest possibility never reduces the earliest starting times of the actions following it, and neither does waiting after an action is finished.

For the latest starting times it works a little different. The FIFO principle does

not work the other way around. If the departure at a location is at $t + 1$ instead of t, the increase in travel time could be anything. This means we could get the wrong values when leaving everywhere as soon as possible. We still do plan every action as late as possible, starting from the last one. But instead of using the $travelTime(i, j, t)$ function, we now use the $travelTimeTo(i, j, t)$ function, which gives us the latest time we can leave location i to arrive at location j at or before time t. The latest insertion time for action i is defined as follows:

$$LST_i = \begin{cases} min(LST_{i+1} - travelTimeTo(i, i+1, LST_{i+1}) - serviceTime_i, & \text{if i is not the first} \\ twe_i - serviceTime_i) & \text{action in the route } = \\ max\{\texttt{latest start time at i without forcing i+1} & \\ \texttt{to start later than possible}, tws_i\} & \\ & \\ twe_i - serviceTime_i & \text{otherwise} \end{cases}$$

Every time an action is added or removed from a route we update the values. If an action is shifted in time, the earliest and latest insertion times do not have to be updated.

To create a more consistent representation, we will only use EST and LST from now on. The slack values we described for instances without time dependent travel times can be represented as EST and LST with the following conversion:

$$EST_i = start_i - BS_i \tag{4.8}$$

$$LST_i = start_i + FS_i \tag{4.9}$$

4.5.2 Insertability

To be able to select a good option for inserting an action we need a method to find out if an action can be placed on a given location in the planning. An action can be either a pickup or a delivery. We also need to know the cost of this insertion. An action can be inserted at a given location if it satisfies all of the constraints. Firstly, it must comply with all of the time constraints. Secondly, the capacity of the vehicle should not be exceeded by inserting this action.

Before any other calculation is done, we check if the action can possibly be inserted in the route. A route has a start and an end time. If the inserted action can not be planned between the bounds of this route, given the action's time windows and the travel time from and to the starting location of the route, we can immediately return that the insertion is impossible.

Earliest and latest insertion time

If the action can be inserted in the route, we need to know its earliest and latest insertion times for the position in the route we want to insert it on. Say we want to know if we can insert action i after action p and before action n. Then we can calculate the earliest and latest insertion times of the inserted action as follows:

$$earliest(i,p,n) = max(EST_p + serviceTime_p + travelTime(p,i,EST_p + serviceTime_p), tws_i) \quad (4.10)$$

$$latest(i,p,n) = min(LST_n - travelTimeTo(i,n,LST_n) - serviceTime_i, twe_i - serviceTime_i) \quad (4.11)$$

Because we know the earliest and latest starting times for all the actions in the route we only have to look at the action before and after the insertion position. If the earliest insertion time for inserting an action at a specific position in the route is smaller than or equal to the latest insertion time, the action can be inserted there. If this is not the case it is not possible to insert the action at that position without violating time constraints. The time window starting from the earliest insertion time $earliest(i,p,n)$, and ending at the latest insertion time $latest(i,p,n)$ is the time window in which action i can start if inserted between actions p and n.

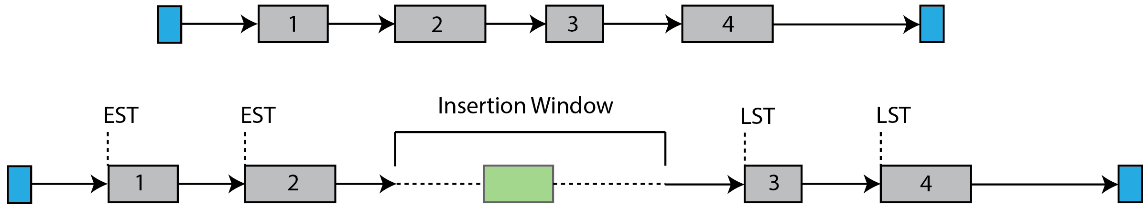

Figure 4.5: Illustration of the use of earliest and latest starting times, to determine if an action can be inserted. The rectangles represent actions. The grey rectangles are order actions, and the blue rectangles are depot actions. The arrows between the rectangles represent the travel times. The insertion window for an inserted action depends on the EST of the previous action and the LST of the next action.

Vehicle capacity

Another constraint that has to be met for an action to be insertable is the vehicle capacity constraint. For every inserted action, we store what the current load is at that action. When inserting a new pickup action we add the increase in load of the inserted action to the load the vehicle had after visiting the previous node. If this is lower than the vehicle's capacity we can insert the action. When inserting a pickup action we do not yet look at the influence this insertion has on the load at locations it visits later on in the route, while it could very well be that the capacity of the vehicle is exceeded later on in the route by this insertion. When checking for the insertion of the delivery action that belongs to the newly inserted pickup action however we look at all the actions between the newly inserted pickup and its delivery. If the vehicle's capacity is exceeded then we declare the insertion pair not insertable.

Best insertion time

Once the earliest and latest insertion times for an insertion of an action are known we can use these to find the best starting time. This is the starting time that, if the new action is inserted at that time, adds least to the total penalty of the solution. We want little waiting time since waiting time adds to the total duration of the shift. This is why we introduced a number of scenarios for which we can quickly find good options for the starting times. This method does not guarantee that the best possible starting times are found, but it can quickly find good options.

Insertion at the start of a shift : When inserting an order at the start of the shift it needs to be shifted forward until it arrives at the next location right in time. If the inserted action can not be shifted forward in time far enough because of its time window we plan it as late as possible. If the inserted action can not be planned early enough without shifting subsequent actions, it is planned as early as possible, and the following actions are shifted forward in time.

Insertion at the end of a shift : When inserting an order at the end of the shift, we insert it so that when leaving from the previous action we arrive there exactly on time. If this is not possible because of the inserted actions' time windows we make sure that the starting time is as close as possible to the time we wanted.

Insertion inside a shift : When inserting an action inside a shift we have a range of starting times to insert the action without creating more waiting time. This range is determined by the time at which we can arrive at the inserted location earliest without shifting the previous action backwards in time, and the time we can plan this order without shifting the next action forward in time.

Insertion in an empty shift : When inserting an action in an empty shift, we can insert the order anywhere in the range between its earliest insertion time and its latest insertion time. We select a number of values in this range, and choose the best one, based on the cost of the insertion at that time.

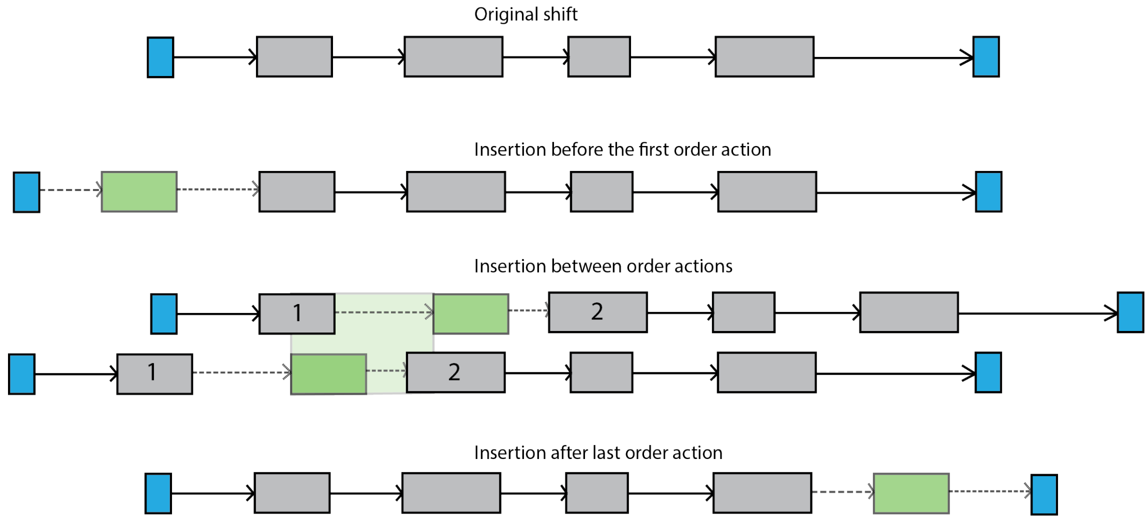

Figure 4.6: Finding the best insertion times. The green block is the inserted order, the grey blocks are already planned order actions, and the blue blocks are the start and end of the shift at the depot. For insertions at the start and end of a shift we only allow one starting time, the one that induces the least amount of waiting time possible in the shift. When inserting between other order actions there is a time window in which no waiting time is created. A predefined number of starting times for the inserted action are checked within this time period.

Calculating insertion cost

When we want to insert an order we calculate the cost of this insertion first. We do not calculate the insertion cost for all possible insertion times but only for the ones we check when finding the best insertion time.

Calculating the change in distance penalty is straightforward. When inserting an action i between actions p and n, where p is the previous action and n is the next action, we calculate the previous distance between p and n first. The new distance is the addition of the distance from p to i and from i to n. The change in distance penalty is then the difference between the old distance penalty and the new distance penalty. If distance is time dependent, the change in distance between all actions that are affected by the insertion must be recalculated.

The time penalty is the difference between the old time penalty for this shift and the penalty after inserting the action. We know the old penalty value for the route. The new time penalty is the difference in penalty caused by the inserted action, and all actions that it affects.

Finally, if we insert an action in an empty route we add the fixed cost for using the vehicle that drives this route to the insertion cost.

Finding all possible insertions

In our approach we find all possible insertion positions, for a limited number of starting times, when inserting an order $o \in O$. From this set of possible insertions an insertion can be selected, based on the cost increase that this insertion causes. First we find all possible insertions for all of the possible pickup actions $pa \in P_o$ that belong to this order. We do this by going through all the vehicles. For all the vehicles we go through all of its routes. First we check if the action fits in the time window of this route. Then for all the route actions ta, except for the last unload action we check if the pickup action can be inserted between ta and ta_{next}. For all the possible insertions we found for the pickup actions of o we then try to find possibilities for inserting the delivery actions. We do this by first inserting the pickup action at the previously found possible insertion. The check for possible delivery insertions is the same as that of the pickup insertions, but this time we only check if it can be inserted after route actions that are planned after pa. If both the pickup action and the delivery action can be planned we have a possible insertion pair. This procedure gives us a list of all possible insertion pairs.

For all the possible insertions we make an estimate of the cost of inserting the action at that position. The cost of the insertion pair is the addition of the cost of the insertion of the pickup and the cost of the insertion of the delivery.

Insertion by probability

From the list of possible insertion pairs, we can select the cheapest insertion. But because we do not use tabu search this could result in parts of the search space being excluded. To overcome this we select one of the possible insertion pairs by probability. The probabilities we use to select an insertion pair are calculated with the method shown in Algorithm 5.

Algorithm 5 Selecting insertion pairs by probability

1: **procedure** INSERTBYPOSSIBILITY($possibleInsertionPairs, bias$)
2: $sum \leftarrow 0$
3: **for** $pip \in possibleInsertionPairs$ **do**
4: $sum \leftarrow sum + \frac{1}{pip.insertionCost^{bias}}$
5: **end for**
6: **for** $pip \in possibleInsertionPairs$ **do**
7: $pip.selectionProbability \leftarrow \frac{1}{pip.insertionCost^{bias} \cdot sum}$
8: **end for**
9: **end procedure**

For selecting by probability higher insertion costs are worse, and need to get lower selection probabilities. For this reason we assign new values of goodness to the insertion pairs, where higher is better. These values are defined as $\frac{1}{pip.insertionCost^{bias}}$ where $bias$ is the bias towards better insertions, and $pip.insertionCost$ is the cost

of inserting the possible insertion pair. The *bias* factor can have any value higher than or equal to 0. The higher *bias* is, the higher the probability of selecting an insertion with a low insertion cost. Because the sum of all probabilities has to be 1 we have to divide these goodness values by the sum of all values, *sum*. This gives us the probabilities for selection. Using these probabilities one of the insertions is selected.

4.5.3 Insertion

Once a possible insertion pair is actually inserted we have to modify the current solution. Before inserting an action, first all other actions from the same order and the same type (pickup or delivery) are removed to avoid planning more than two actions per order. The fitness value is adjusted after this removal. This means that when we insert a pickup action any pickup actions belonging to the same order are removed from the solution. The same holds for delivery actions.

After inserting the action at the desired time, it may be that the actions before it and after it have to be shifted to create a valid solution again. Starting from the action p before the inserted action i we check if the addition of the start time of p, the duration of p and the travel time from p to i when arriving at $start_i$ is greater than $start_i$. If this is so the start time of p is set earlier so the vehicle can arrive at location i on time. This is done for all the actions preceding i in the same route. For the actions subsequent to the inserted action i we do the same. For every action, starting at n which is the first successor of i we calculate if we arrive at these locations later than their current starting time. If so we shift them forward enough to arrive exactly on time. This is always possible, because an insertion is chosen from the list of possible insertions.

After all the starting times have been updated the slack values will also change, so we need to recalculate the slack values for this route as well. At the same time we adjust the load values at all of the actions.

4.5.4 Removal

When removing an action we create more room to shift the orders, so we will need to update all the slack values for this route again. Also, after we have removed an action from a route that is not the first or last action in a shift, we create a waiting time gap in the shift. To fill this gap we first shift all actions subsequent to the removed action backwards in time as far as possible without having to shift any of the actions before the removed action. Then we shift all actions preceding the removed action forward as far as possible without moving the actions after the removed action. These new starting times are not necessarily optimal, because travel times may increase when shifted, but this can be adjusted by the *time shift* operator. In addition to moving the actions in the route, we also update all the load values.

4.6 Summary

In this chapter we have reviewed the algorithm we use for solving the Pickup and Delivery problem with time windows and time dependent travel time. A high level description of the algorithm was given in Section 4.1, followed by more detailed reviews of the different parts of the algorithm. A number of initialization methods have been introduced. Whereas most of these methods have been used and described in other literature, we introduced a new method for initializing solutions: initialization by order similarity. The similarity factor is also explained in this section. A number of small- and large neighborhood operators that are used to improve the solution have been introduced in Section 4.2. In Section 4.3 we discussed the implementation of time dependent travel times in our algorithm. We have also elaborated on how we can work with these time dependent travel times in an efficient fashion, using time dependent slack values. In Section 4.4 the implementation of maximum shift time for routes, using stop actions was discussed. Finally, in section 4.5, we described the process of the insertion and removal of orders, and what is needed for these to work with time dependent travel time and alternative pickups and deliveries.

Chapter 5

Experiments & results

In this chapter we will describe the experiments we performed to test the algorithm. Many parameter values have to be determined before running the algorithm, and most of these values influence each other. Also, the best values of the parameters can vary between different problems. We will start by finding the initialization method to use, determined by the quality of the produced solutions. Next the selection probabilities for all of the operators are determined by their average improvement. Then good values for temperature and the bias value for selecting better insertions are examined. Finally the influence of the large neighborhood operators is evaluated. With these values we solve two benchmarks: The benchmark for the basic PDPTW by Li and Lim [2003], and our own benchmark which extends this problem with time dependent travel times and alternative pickups and deliveries. We have also tested the influence on the quality and computing time of the additions we made to the algorithm: Time dependent travel time, alternative pickups and deliveries, and maximum shift times. The chapter is concluded with an evaluation of the requirements for solving real world problems.

The algorithm is written in java. All of the following experiments are run on a laptop with a 2.2 ghz quadcore cpu (i7-2670QM) with a maximum of 8 simultaneous threads, and 4 GB of memory.

5.1 Initialization methods

First we tested the initialization methods. Every initialization method was run five times on three different problem instances: one where the locations are clustered (*lc*_1_6_1), one where they are randomly distributed (*lr*_1_6_1) and a combination of the two (*lrc*_1_6_1). All these instances come from the Li & Lim benchmark (see Li and Lim [2003]). The results are given in Table 5.1. As can be read from the table, the insertion by similarity method outperforms the other methods on both the clustered and the combined instances, whereas the farthest insertion method is better at the randomly distributed instance. Because the insertion by similarity method produced the best solutions in two out of three instances, and was second at the randomly distributed instance, we will use initialization by similarity in the

Instance	Method	best routes	avg. routes	best km	avg. km	best tt	avg tt	best fitness	avg. fitness	avg. time (s)
lc_1_6_1	similarity	**61**	**61.0**	**15421**	**15527.4**	**4283**	**4312.6**	**15421.48**	**15527.72**	3.01
	random	95	98.4	31953	34206.4	8876	9501.6	31953.84	34207.03	**2.41**
	farthest	64	66.0	15486	16147.2	4301	4485.0	15486.58	16148.02	3.09
	nearest	68	70.2	17410	18283.0	4836	5078.4	17410.23	18283.61	2.53
	sweep	67	71.4	16477	18036.0	4577	5009.8	16477.99	18036.80	2.52
	cheapest	98	101.8	33464	36336.6	9295	10093.0	33464.92	36337.52	38.48
lr_1_6_1	similarity	67	**68.2**	31243	31873.8	8678	8853.4	31243.91	31874.74	**1.92**
	random	71	75.2	38270	39435.4	10630	10953.8	38270.91	39436.23	2.24
	farthest	72	75.2	**30012**	**30738.8**	**8336**	**8538.0**	**30012.47**	**30739.57**	2.23
	nearest	87	88.6	35187	36486.2	9774	10134.8	35187.52	36486.87	3.14
	sweep	80	82.4	32707	33847.2	9085	9401.6	32707.93	33848.13	2.84
	cheapest	**66**	**68.2**	33437	35946.0	9288	9984.6	33437.88	35946.87	36.73
lrc_1_6_1	similarity	**66**	**66.6**	**23069**	**23332.6**	**6408**	**6481.2**	**23069.94**	**23333.34**	2.31
	random	71	76.0	26567	27567.6	7379	7657.0	26567.78	27568.14	**1.77**
	farthest	70	71.2	23476	24064.8	6521	6684.4	23476.69	24065.66	1.95
	nearest	73	78.8	26818	28218.4	7449	7838.0	26818.83	28219.37	1.85
	sweep	69	71.4	24759	25486.8	6877	7079.2	24759.61	25487.62	2.19
	cheapest	73	75.0	26393	27681.2	7331	7689.0	26394.17	27681.95	33.59

Table 5.1: Comparison of initialization methods on three instances from different categories. We compare the best and average solutions generated by the initialization methods. Also the average time to complete the method is given in this comparison. The best values per problem instance are printed in bold font.

following experiments. Producing the best result with an initialization method does not mean that it always leads to the best final results after running the algorithm, because the generated starting solution may be more difficult to improve. However on the Li & Lim benchmark the algorithm seemed to find the best known solutions faster when using this initialization method.

5.2 Operators

For determining the probabilities for selecting the operators we do five runs for three different types of problem instance with 600 locations: One clustered, one randomly distributed, and one combined instance. The effectiveness of all the operators is tested by tracking the average success rate, the number of times the operator was called, and the average improvement on success over these five runs. All the experiments were given 10 minutes computing time.

Not all the operators are included in this experiment. We did a number of exploratory runs to see which operators were most competitive, and then tested these operators more extensively. All of these selected operators are used in every run, except for the time shift operator which is only used in the time dependent problem instances. For this reason this operator is not included in these experiments. The time shift operator however is the only operator in our algorithm which improves the time penalty, so it is always used in problems instances with time dependency. The results for the small scale operators are in Table 5.2, and the results for the large neighborhood operators are in Table 5.4. The probabilities to select these operators in the algorithm are given in Tables 5.3 and 5.5.

The probabilities for selecting a certain operator in an iteration are determined by the results of this experiment. The average improvement of the solution by an

Instance	Operator	Times called	Success rate	avg. impr.	Ranking
lc_1_6_1	simple pair insertion	20856	0.04%	48.51	3
	exchange	20785	0.34%	819.11	2
	exchange consecutive	20954	**0.19%**	**1611.95**	**1**
	insert non-planned	10418	0.04%	30.72	4
lr_1_6_1	simple pair insertion	19393	2.15%	748.45	3
	exchange	19630	**4.24 %**	**5335.10**	**1**
	exchange consecutive	19808	3.32%	4553.54	2
	insert non-planned	9944	2.70%	493.86	4
lrc_1_6_1	simple pair insertion	21235	6.73%	329.06	4
	exchange	21204	10.62%	3736.13	2
	exchange consecutive	21088	**10.23%**	**5609.65**	**1**
	insert non-planned	10513	7.44%	321.47	3

Table 5.2: A comparison of the small scale operators. The ranking of the operators is based on their success rate multiplied by the average improvement. The best operator is marked for all the instances.

Operator	pt. lc	pt. lr	pt. lrc	points	selection probability
simple pair insertion	2	2	1	5	1/4
exchange	3	4	3	10	1/2
exchange consecutive	4	3	4	11	11/20
insert non-planned	1	1	2	4	4/20

Table 5.3: The probabilities of selecting the small scale operators. These are based on the points (the inverted ranking) for their performance in the three problem instances. The three columns after the operator name indicate the number of points gained on the different instances (clustered, randomly distributed and combined).

operator is the success rate multiplied by the average improvement on success. For all operators we calculate weights for the three problem instances. The operators are ranked according to their weights, and are given points according to the inverse of their ranking, i.e. if the ranks go from 1 to 4, the operator that is ranked first gets 4 points. The sum of rankings over all instances determines the selection probability.

Instance	Operator	Times called	Success rate	Avg. impr.	Ranking
lc_1_6_1	Ruin/Recreate (random/fast)	1739	1.35%	621.33	4
	Ruin/Recreate (random/similar)	1659	0.86%	250.00	6
	Ruin/Recreate (similar/fast)	1654	1.79%	378.87	5
	Ruin/Recreate (similar/similar)	1652	1.11%	765.36	3
	2-opt*	1649	2.96%	17.43	7
	Combine routes	1618	1.17%	1650.53	2
	Combine remove	1680	**1.76%**	**3746.91**	1
lr_1_6_1	Ruin/Recreate (random/fast)	1789	12.94%	1689.50	4
	Ruin/Recreate (random/similar)	1773	7.90%	1766.78	6
	Ruin/Recreate (similar/fast)	1719	13.92%	1883.93	2
	Ruin/Recreate (similar/similar)	1770	8.29%	1819.63	5
	2-opt*	1693	8.88%	837.52	7
	Combine routes	1839	**6.57%**	**16837.95**	1
	Combine remove	1776	5.20%	4223.57	3
lrc_1_6_1	Ruin/Recreate (random/fast)	1874	18.97%	477.91	5
	Ruin/Recreate (random/similar)	1768	8.39%	3539.61	4
	Ruin/Recreate (similar/fast)	1955	17.92%	403.53	6
	Ruin/Recreate (similar/similar)	1824	9.71%	4070.19	3
	2-opt*	1768	18.01%	339.99	7
	Combine routes	1796	**6.49** %	**13600.48**	1
	Combine remove	1787	5.41%	13619.98	2

Table 5.4: A comparison of the large scale operators. The ranking of the operators is based on their success rate multiplied by the average improvement. The best operator is marked for all the instances.

Operator	pt. lc	pt. lr	pt. lrc	points	selection probability
Ruin/Recreate (random/fast)	4	4	3	11	11/84
Ruin/Recreate (random/similar)	2	2	4	8	8/84
Ruin/Recreate (similar/fast)	3	6	2	11	11/84
Ruin/Recreate (similar/similar)	5	3	5	13	13/84
2-opt*	1	1	1	3	3/84
Combine routes	6	7	7	20	20/84
Combine remove	7	5	6	18	18/84

Table 5.5: The probabilities of selecting the large neighborhood operators. These are based on the points (the inverted ranking) for their performance in the three problem instances. The three columns after the operator name indicate the number of points gained on the different instances (clustered, randomly distributed and combined).

5.3 Temperature

In Chapter 4, we explained how to find the starting temperature T_0 that roughly corresponds to a certain acceptance ratio X_0. The acceptance ratio is the fraction of worse neighbours to be accepted. An acceptance ratio of 0.5 means that half of the worse neighbours are accepted. To find a value for T_0 for a given acceptance ratio X_0, we use Equation 5.1.

$$\frac{\sum_{i=0}^{i<|L|} e^{\frac{-L_i}{T_0}}}{|L|} = X_0 \tag{5.1}$$

Here the set L is a set of randomly selected neighbours from a starting solution of the problem instance. The larger the set L is, the more accurate the starting temperature value for an acceptance ratio of X_0 can be determined. In our experiment, the set L has a size of 1000. The left side of the equation equals the average acceptance probability of all the elements in the set L. The probability of accepting a change to the current solution that is worse than the current solution is determined as $e^{\frac{-difference}{temperature}}$, where $difference$ is the new fitness value minus the old value. The numerator of the fraction is the sum of all acceptance probabilities in the set L.

Even if we know how to determine a starting temperature that corresponds to a given acceptance ratio, the best value for X_0 is still unknown. In the next experiment we will try to find the best value for X_0, by doing test runs with a number of different values. First, the fitness values for the desired acceptance ratios are calculated for the given problem instance, with the method described before. Then the algorithm is executed 5 times with 15000 iterations for starting acceptance ratios 0.3, 0.4, 0.5, 0.6 and 0.7 on a half clustered, half randomly distributed problem instance. We chose to test the acceptance ratio on this instance because it has the structure of both the clustered and the randomly distributed instances. Figure 5.1 illustrates the acceptance ratio to temperature function. The results of this experiment can be found in Table 5.6. As can be read from this table, the higher the starting temperature becomes, the higher the probability that the number of routes will be minimized, but the fitness value is best at a value of 0.5 or 0.6. Also the algorithm will improve faster with low temperatures. For the following experiments an initial acceptance ratio between of 0.55 will be used, because 0.5 produced the best average values, and 0.6 produced the best overall value.

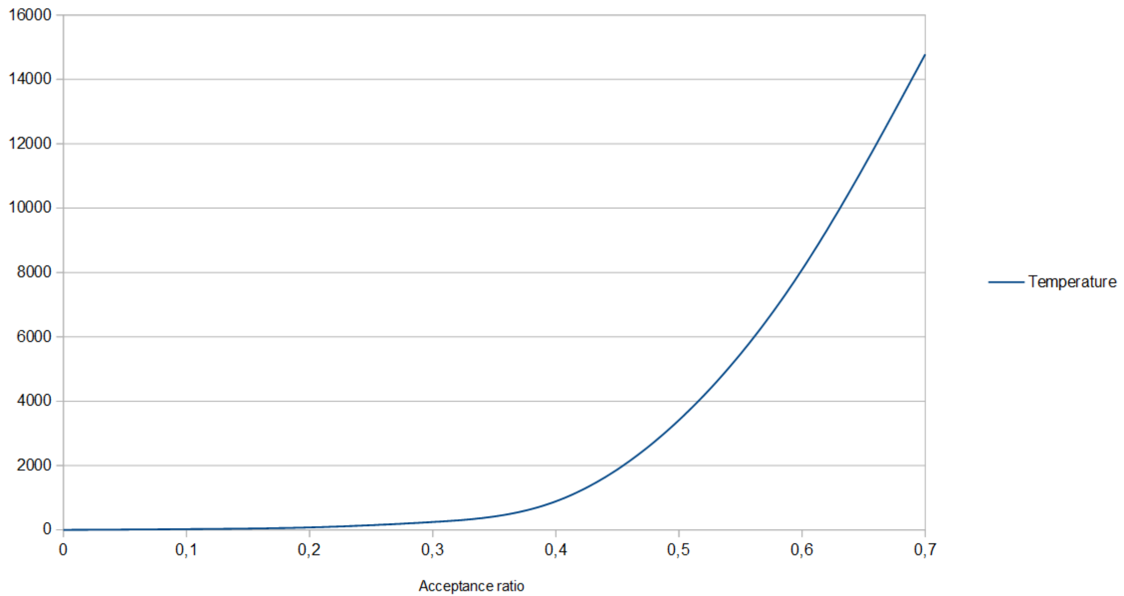

Figure 5.1: Acceptance ratio to temperature for problem instance lrc_1_6_1. The horizontal axis is the acceptance ratio and the vertical axis the temperature that gives such an acceptance ratio.

X_0	T_0	best routes	avg. routes	best km	avg. km	best tt	avg tt	best fitness	avg. fitness
0.3	246	55	55.5	17924	18367.2	4979	5101.7	866674.73	879501.33
0.4	888	54	54.8	**17855**	**18054.0**	**4959**	**5014.5**	860070.28	866960.60
0.5	3413	**53**	54.6	17930	18087.6	4980	5024.0	850256.23	**865556.52**
0.6	8092	**53**	54.6	17961	18088.1	4989	5024.1	**849979.89**	865565.02
0.7	14781	**53**	**54.4**	17981	18212.6	4995	5058.8	850145.27	865766.25

Table 5.6: Comparison of different initial acceptance ratios over 15000 iterations for instance $LRC_1_6_1$ with final temperature 1.

5.4 Selection by probability

To be able to reach a good solution faster, but still allowing the algorithm to explore all of the search space, we introduced selection by probability. This means that when selecting a new solution from a neighborhood, all options are considered, but there is a bias towards better options. When this *bias factor* is 0 no distinction is made between neighbors, because the selection weight $\frac{1}{insertionCost^{bias}}$ is the same for all insertion costs, and $x^0 = 1$ for all values. When the bias factor has a high value, better neighbors have a higher chance of being chosen. To see the influence of this bias factor we make a comparison between solutions with different bias values, where the other values are kept the same. These experiments are based on 5 runs, with 30.000 iterations each. As with all previous experiments the problem instance is $LRC_1_6_1$. The results of this experiment are listed in Table 5.7. As we expected

bias	best routes	avg. routes	best km	avg. km	best tt	avg tt	best fitness	avg. fitness
0.0	59	61.6	19323	20015.2	5367	5559.4	931382.04	969609.00
1.0	54	55.0	18006	18278.2	5001	5077.0	865205.84	872929.12
1.5	55	55.0	**17838**	**17960.0**	**4955**	**4988.6**	865153.98	867305.21
2.0	**53**	**54.0**	17870	18005.8	4964	5001.2	**849522.84**	**858110.35**
100.0	54	54.4	17972	18086.8	4992	5023.8	857509.07	863542.89

Table 5.7: Comparison of different bias values. a bias value of 1.0 means there is no bias at all, a value of 100 practically removes the probability factor, only the best neighbours are chosen.

only selecting the best neighbors restricts the search space too much, experiments with a bias value of 2.0 produced the best results with the given number of iterations. It may however be possible that the results with lower bias values may be better if given more time.

5.5 Large Neighborhood search

Because we reasoned that small changes may not be always enough to escape from local optima we introduced a number of large neighborhood operators. To test if these operators have a positive influence on the quality of the produced solutions, we compare the algorithm with and without LNS operators on a hard instance from the Li & Lim benchmark set. We compare how fast both algorithms converge and what the final results are on this problem instance. The algorithm was run for 30 minutes with 20.000 iterations per vehicle minimization stage. Every problem instance is ran 5 times. For the runs that use large neighborhood operators, there is a 1% chance that one of the LNS operators is chosen. As can be read from the results in Chart 5.2 and Table 5.8, the results are not unanimously in favor of using or not using LNS. Both produce equal results in the clustered instances. In the randomly distributed instances the algorithm seems to perform a little better without LNS, and in the combined instances the algorithm with LNS produces slightly better solutions. Although the results do not differ much, we choose to use the LNS operators for solving the benchmarks because they seem to find solutions with fewer vehicles better in some of the cases.

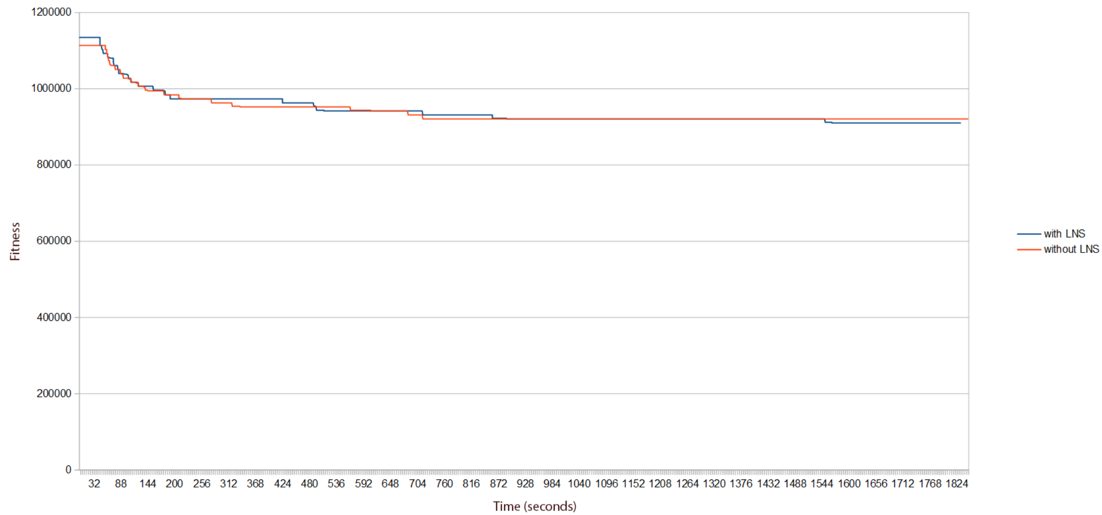

Figure 5.2: Convergence with, and without the use of LNS operators on the problem instance lrc1101.

Instance	LNS	best routes	avg. routes	best fitness	avg. fitness
lc181	yes	80	80.00	25184.38	25184.38
	no	80	80.00	25184.38	25184.38
lc281	yes	24	24.00	11687.06	11687.06
	no	24	24.00	11687.06	11687.06
lr181	yes	80	80.40	41841.85	**42159.43**
	no	80	**80.20**	**41236.00**	42170.89
lr281	yes	16	17.00	29961.97	30597.46
	no	16	**16.80**	**29961.22**	**30140.01**
lrc181	yes	**68**	70.40	**32640.93**	**33079.96**
	no	69	70.40	32675.99	33171.90
lrc281	yes	**21**	**21.80**	21799.20	**21954.61**
	no	22	23.20	**21670.45**	22281.18

Table 5.8: Comparison of running the algorithm with and without LNS (1% of the time) on a number of instances of the Li & Lim benchmark set. The best results are printed in bold font.

5.6 Benchmark : Li & Lim

To be able to determine how our algorithm performs we used benchmark instances by Li and Lim [2003]. The problems in this set are pickup and delivery with time windows problems. Many researchers published the results of their algorithms on this benchmark set, and the best known results are published online. For this reason we can use this benchmark to compare our results to other algorithms.

The set of benchmark problems consists of 354 pickup and delivery problem instances with time windows. The size of these problem instances ranges from around 100 to around 1000 locations (100, 200, 400, 600, 800 and 1000 locations), with half as many orders. The problems are divided in six problem classes: LC1, LC2, LR1, LR2, LRC1 and LRC2. In the sets LC1 and LC2 the orders are clustered, in LR1 and LR2 the orders are randomly distributed, and in LRC1 and LRC2 the orders are partially clustered and partially randomly distributed. The difference between the sets ending on 1 and the sets ending on 2 is that the sets ending on 1 have short planning horizons and the sets ending on two have a long planning horizon. In this experiment 36 instances from this benchmark will be tested; one instance from all of the problem classes, and all of the problem sizes.

The locations used in this benchmark are located on a plane, but they are not real geographical locations. Every location has an x and a y coordinate, and the distance and driving time between two locations is calculated as the distance between the two points on the plane, calculated with double precision. All orders consist of a pickup and a delivery action with a time window and a location. An order also has a demand, which indicates the capacity needed to transport this order. For every problem instance there is a given number of homogenous vehicles available with a given capacity. The benchmark has a hierarchical objective: The first objective is to minimize the number of vehicles used, and the second objective is to minimize the number of kilometers driven. The total number of kilometers driven is rounded to two decimals.

For this benchmark we use the results from the previous experiments to determine the settings. The temperature is set to have a starting acceptance ratio of 55% and a final acceptance ratio of 1%. We use the *similarity* initialization method, and the selection probabilities for the operators as determined before. The running time of the algorithm depends on the size of the instance to be solved. For every 100 locations the algorithm gets 10 minutes of computing time. Because the first priority is to minimize the number of vehicles we do a number of vehicle minimization rounds. Every round 20.000 iterations are run. If within these 20.000 iterations all orders are planned a vehicle is removed from the solution (based on the number of orders assigned to the vehicles, a vehicle with fewer orders assigned gets a higher probability of being removed), and the algorithm starts solving the problem again, with one vehicle less. If not all orders could be planned within these 20.000 iterations, all removed vehicles are restored again. The results are stated in Table 5.9. As can

be read from this table the algorithm performs very well on the clustered problem instances, for many of the instances the best known result is equaled. For some of the problems the driven distance is improved, but the number of vehicles used is higher than in the best known result. For the randomly distributed set it can be noticed that for the smaller problems the best known solutions are found, but for the larger problems the algorithm performs a little worse, although most of the time the solution with the best known number of vehicles is still matched. The last problem set, which is partially clustered and partially randomly distributed, has proven hardest to solve for our algorithm. Even though shorter routes are found for some of the instances, the algorithm often fails to equal the minimum number of vehicles needed.

Instance	routes (bk)	distance (bk)	routes (br)	distance (br)	routes (avg)	distance (avg)	time (min)
lc101	10	828.94	**10**	**828.94**	**10.0**	**828.94**	10
lc121	20	2704.57	**20**	**2704.57**	**20.0**	**2704.57**	20
lc141	40	7152.06	**40**	**7152.06**	**40.0**	**7152.06**	40
lc161	60	14095.60	**60**	14095.64	**60.0**	14095.64	60
lc181	80	25184.38	**80**	**25184.38**	**80.0**	**25184.38**	80
lc1101	100	42488.66	**100**	**42488.66**	**100.0**	**42488.66**	100
lc201	3	591.56	**3**	**591.56**	**3.0**	**591.56**	10
lc221	6	1931.44	**6**	**1931.44**	**6.0**	**1931.44**	20
lc241	12	4116.33	**12**	**4116.33**	**12.0**	**4116.33**	40
lc261	19	7977.98	**19**	**7977.98**	**19.0**	**7977.98**	60
lc281	24	11687.06	**24**	**11687.06**	**24.0**	**11687.06**	80
lc2101	30	16879.24	**30**	**16879.24**	**30.0**	**16879.24**	100
lr101	19	1650.80	**19**	**1650.8**	**19.0**	**1650.8**	10
lr121	20	4819.12	**20**	**4819.12**	**20.0**	**4819.12**	20
lr141	40	10639.75	**40**	10689.56	**40.0**	10764.18	40
lr161	59	22838.30	**59**	23326.27	59.6	23751.51	60
lr181	80	39315.92	**80**	41911.65	80.2	42372.05	80
lr1101	100	56903.88	**100**	60437.15	**100.0**	60990.64	100
lr201	4	1253.23	**4**	**1253.23**	**4.0**	**1253.23**	10
lr221	5	4073.10	**5**	**4073.1**	**5.0**	**4073.1**	20
lr241	8	9726.88	**8**	**9726.88**	**8.0**	9889.09	40
lr261	11	21945.30	12	**18960.75**	12.0	**19116.29**	60
lr281	15	33816.90	16	**30070.72**	16.4	**30836.63**	80
lr2101	19	45422.58	**19**	45924.79	19.4	46688.62	100
lrc101	14	1708.80	**14**	**1708.8**	**14.0**	**1708.8**	10
lrc121	19	3606.06	**19**	**3606.06**	**19.0**	3606.86	20
lrc141	36	8966.97	37	**8944.58**	37.0	8977.74	40
lrc161	53	17924.88	**53**	17994.47	53.2	18127.96	60
lrc181	67	32268.95	68	32501.92	68.2	33016.25	80
lrc1101	84	49315.30	86	50085.55	86.4	50385.15	100
lrc201	4	1406.94	**4**	**1406.94**	**4.0**	**1406.94**	10
lrc221	6	3605.40	7	**2997.06**	7.0	**2998.59**	20
lrc241	12	7471.01	13	**6646.02**	13.0	**6651.19**	40
lrc261	16	14817.72	17	**13122.63**	17.0	**13185.15**	60
lrc281	20	23289.40	21	**21461.99**	21.0	**21933.31**	80
lrc2101	22	35073.70	24	**31006.56**	24.0	**31276.39**	100

Table 5.9: The results of our algorithm on the Li & Lim Benchmark. (bk) is best known result, (br) is the best result our algorithm found over 5 runs and (avg) is the average result of our algorithm. Results that are equal to or better than the best known result are written in bold font.

5.7 Time dependency

Time dependent travel times can have a positive impact on the produced solutions, because they correspond to reality better. The disadvantage is that there are more computations to be done, which has a negative influence on the convergence speed of the solutions. To test the impact of time dependency on computing times we did 5 test runs with 5.000 iterations on randomly distributed problem instances from our benchmark set (dtr501, . . . ,dtr5001). We performed tests with time dependency where multiple possible starting times are considered when inserting actions. Also a simpler way of handling time dependency is tested, in which only a single starting time is considered at insertion to improve the computing time. Finally to make a comparison, the algorithm is run without time dependency on the same instances. The results of this experiment can be found in Table 5.10. As can be read from the table there is still a large difference between calculations with and without time dependency. With time dependency more computations have to be done, because one insertion or removal can influence all other orders in the same route. The cost with time dependent travel times in Figure 5.3 is higher because the fitness is calculated in a different way: in the run without time dependency there was no cost for waiting and working time.

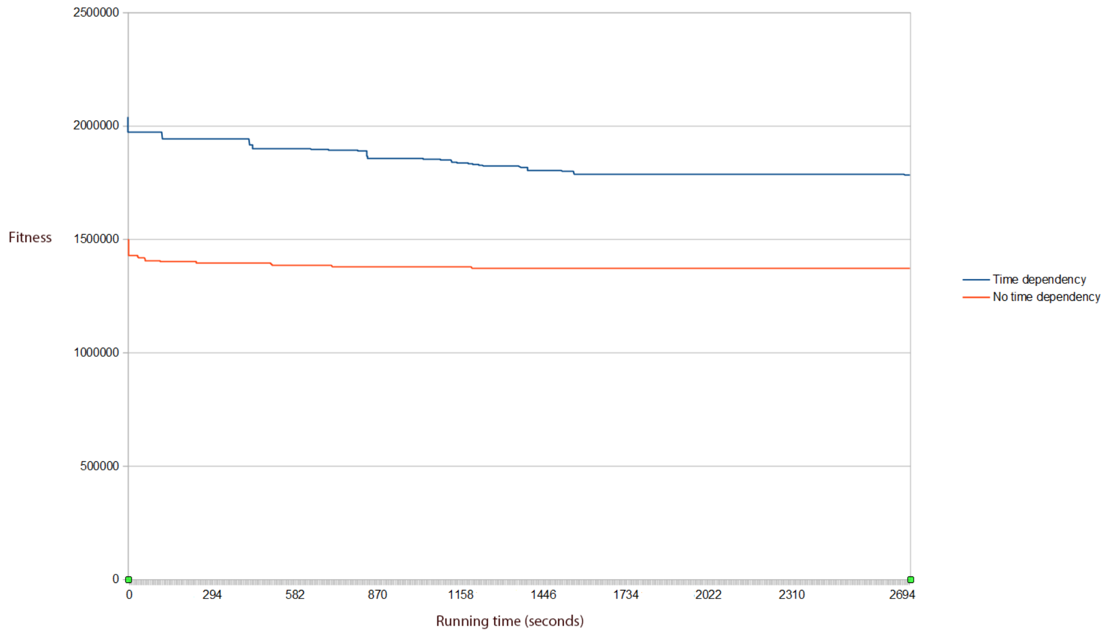

Figure 5.3: The difference in convergence speed between an instance with time dependence, and the same instance without time dependence.

Instance Size (orders)	Time dependency & Multiple Options (s)	Time dependency (s)	No time dependency (s)
50	279.46 (41.16 x)	39.91 (5.88 x)	6.79
100	417.18 (48.79 x)	85.85 (10.04 x)	8.55
200	1014.38 (72.46 x)	171.85 (12.28 x)	14.00
300	1189.69 (48.62 x)	271.77 (11.11 x)	24.47
400	1908.07 (54.10 x)	383.65 (10.88 x)	35.27
500	2925.21 (42.24 x)	400.68 (5.79 x)	69.26

Table 5.10: Comparison of computing time (in seconds) for 5000 iterations on problem instances of different sizes with and without time dependency. The factor the computation speed differs from the problem without time dependency is given between the parentheses for the time dependent columns.

5.8 Alternative pickups & deliveries

The second addition to the basic problem we test are the alternative pickups and deliveries. If there are more options available to consider it can be assumed that the computing time needed to solve an instance will increase. To test the influence of alternative locations on the computing time we have generated 5 instances with a differing number of alternatives for the pickups and deliveries. All instances consist of 201 locations, and 100 orders to be serviced. First we test an instance without alternative pickups and deliveries, as a baseline measurement. Then we test instances with an increasing maximum number of alternatives for both the pickups and deliveries, ranging from 1 alternative to 4 alternatives. We compare the average computing time and quality of the solutions in 10.000 iterations, over 5 runs. As can be read from Table 5.11, the computing time increases with the number of alternatives. Also the driven distance and the number of routes decreases, because better alternatives can be chosen.

max. alternatives	avg. time (s)	avg. routes	avg. distance driven
0	15.75	24.8	422485.80
1	23.88	21.6	351515.00
2	37.95	17.2	279654.20
3	54.98	17.0	270034.20
4	71.25	15.6	251232.40

Table 5.11: Comparison of solving 5 randomly generated problem instances, with 100 orders and 201 locations. The maximum number of alternatives applies to both the pickups and deliveries. For example with a maximum of 2 alternatives every pickup and delivery can have 1 to 3 locations / time windows to be considered.

max. shift time	avg. time (s)	avg. routes	avg. shifts	avg. distance	avg. orders planned
-	420.29	15.80	15.80	382382.60	100.0
$\frac{1}{2}$ day	766.41	18.80	22.60	436762.80	100.0
$\frac{1}{3}$ day	1689.09	19.60	32.60	507968.40	100.0
$\frac{1}{4}$ day	1413.90	20.00	43.00	578622.40	99.0

Table 5.12: Comparison of solving a randomly generated instance with 100 orders, using differing shift times. The fraction in the max. shift time column indicates in how many shifts the day is split.

5.9 Maximum shift time

The last addition that is tested is the introduction of the shift time constraint. For this experiment, a problem instance with 100 orders is generated. The orders in this instance have long time windows and small service times. Also, the number of vehicles is low (20), so the shift constraint can not be avoided by using more vehicles. We compare the results of the instance solved with 1 up to 4 shifts per day. The results as shown in Table 5.12 indicate that shorter maximum time for a shift leads to higher computing times. The reason for this is that more shifts need to be considered, which means more options for the insertion of actions. The driven distance also increases with the number of shifts, because between every two shifts the vehicle needs to return to its depot.

5.10 Benchmark : Time Dependency & Alternative Locations

Because to the best of our knowledge there is no benchmark available for pickup and delivery problems that include time dependency or alternative locations, we introduce a new benchmark that includes both these attributes.

The benchmark consists of 144 problems, and is divided in four sets of problems: Clustered orders with time dependency (dtc), Randomly distributed orders with time dependency (dtr), Clustered orders with time dependency and alternative locations (dtac) and Randomly distributed orders with time dependency and alternative locations (dtar). All problem instances have one central depot from which a fleet of heterogenous vehicles depart. The vehicles have different capacities, on which their cost is based. All vehicles have a fixed cost and a variable cost. The fixed cost is added to the fitness value if the vehicle if used, and the variable cost is the penalty per kilometer driven. For all four of the problem classes there are six instances for each of the problem sizes. There are problems with 50, 100, 200, 300, 400 and 500 orders. An order consists of a pickup and a delivery, with time windows. In the problem instances with alternative locations, both the pickups and deliveries can

have alternative locations with different time windows, from which one alternative has to be planned.

The problem instances are generated randomly given a number of parameters. A square area is created in which all the locations are placed. In randomly distributed instances all locations are completely random. To create clustered instances a cluster size and a number of locations per cluster is given. The clusters are then added randomly, with the restriction that they can not overlap previously added clusters. This is a trial-and-error process: when the location of a newly added cluster overlaps an existing cluster it is moved to a new random location, until it no longer overlaps any other clusters. If no non-overlapping location can be found after a number of tries the cluster is added to the last random location. The number of locations depends on the number of orders, the number of depots and the chance of alternative pickups and deliveries. The number of locations is twice the number of orders plus the number of depots. If there is a chance on alternative pickups and deliveries, the number of locations is the average number of locations needed to give every alternative a unique location. After all locations are created the pickups and deliveries for the orders are generated. For an order the pickup locations are chosen at random. The delivery locations are chosen by probability; locations that are close to a pickup location get a higher probability to be chosen as delivery location. The size of the time windows for the pickups and deliveries depends on the values for the parameters that are given. There is a maximum and a minimum time window size for both the pickups and the deliveries. The service time is determined in the same fashion. To make sure it is always possible to schedule an order, earliest and latest starting times for the time windows are determined. These values depend on the travel times and the service times of the pickups and deliveries for this order. For a delivery the earliest starting time is adapted to one of the corresponding pickup alternatives, chosen at random. The travel time is based on the size of the map. With a depot in the center of the map it should be possible to reach all of the orders within the given planning window. We have included time dependent travel times, and alternative pickups in the instance generator. For time dependent travel times, a number of periods with different travel times are set. The travel time modifier for these periods is between a minimum and a maximum value given as input by the user. The time per period and number of periods can also be set. Alternative pickups and deliveries can be generated with a given probability. Also the maximum number of alternatives is given. Pickup alternatives are generated independent of one another, and the location and time window of the deliveries each depend on a randomly selected pickup action of the same order. Finally the user can set a number of vehicles and depots to be used. The vehicles have a randomly generated capacity between a minimum and maximum value. Their cost can be random or dependent of their capacity. And all vehicles are randomly assigned to one of the available depots.

To be able to compare the quality of solutions the penalties that are given in Table

Attibute	Penalty per unit
Driving time	$vcost_v$
Waiting time	$0.25 \cdot vcost_v$
Working time	$0.25 \cdot vcost_v$
Fixed vehicle penalty	$fcost_v$

Table 5.13: Penalty values for different attributes. The sum of all penalties is the fitness value.

5.13 are used. The sum of all penalties is the fitness value of a solution. The distance between two locations is determined by the direct distance between the two points, as if they are in a 2d plane. The distance value is rounded to the nearest integer value. Without time dependent travel time, the driving time between two locations is the same as the distance. With time dependent travel time the travel time between locations is calculated as in Section 4.3.3. The travel time in a given time period is the multiplier for that period multiplied with the distance between the locations. All the penalty values are calculated with double precision, and the total value is rounded to two decimals.

Because the speed is lower when running the algorithm with time dependent travel times, the number of iterations for all the following experiments is set to 7500. The other settings are the same as in the Li & Lim benchmark experiment, with the exception that instead of the *2-opt** operator the *combine routes into new* operator is used. This operator is used here, because contrary to the Li & Lim benchmark, heterogenous vehicles are used here. If we would not use this operator the algorithm would always prefer smaller vehicles, because the cost of these vehicles is lower, so their direct impact on the fitness value is smaller. The *2-opt** was replaced, because it is a slow operator, and the computing time already increases by a considerable factor due to the time dependency.

Instance	rts (b)	fitness (b)	rts (avg)	fitness (avg)	Instance	rts (b)	fitness (b)	rts (avg)	fitness (avg)	time (min)
dtr501	13	531873.57	13.0	539442.87	dtar501	13	484807.02	13.4	506128.32	10
dtr502	18	550506.34	18.0	560749.22	dtar502	14	489506.95	14.8	505953.70	10
dtr503	17	524206.08	16.2	527105.53	dtar503	20	634500.29	20.2	637779.11	10
dtr504	13	525721.21	13.2	547794.36	dtar504	12	549547.60	12.0	559607.83	10
dtr505	13	458868.14	13.2	469051.41	dtar505	12	408704.23	12.8	426007.12	10
dtr506	17	570012.35	17.2	576555.00	dtar506	17	513331.35	16.2	515357.10	10
dtr1001	21	860470.83	22.4	895679.90	dtar1001	22	949136.27	22.0	961146.51	15
dtr1002	28	878028.81	28.6	899437.90	dtar1002	29	935581.85	28.8	954191.66	15
dtr1003	39	1092364.05	40.0	1116255.93	dtar1003	30	974109.51	30.4	988064.04	15
dtr1004	24	913861.38	24.0	938833.02	dtar1004	20	826954.94	19.8	837008.56	15
dtr1005	33	944017.30	32.2	956646.75	dtar1005	26	865159.88	26.2	882683.39	15
dtr1006	35	1026350.54	35.0	1038151.48	dtar1006	32	962312.46	33.2	983585.21	15
dtr2001	45	1819562.84	45.4	1851028.80	dtar2001	40	1753386.35	40.8	1785311.82	25
dtr2002	62	1877375.64	63.0	1895008.75	dtar2002	54	1626828.86	52.8	1651110.24	25
dtr2003	65	1922233.23	67.6	1947556.41	dtar2003	71	2002240.91	69.0	2022907.30	25
dtr2004	38	1747035.07	39.2	1783900.37	dtar2004	37	1601018.82	38.4	1649645.35	25
dtr2005	67	2021301.04	67.0	2043047.35	dtar2005	46	1552164.13	46.6	1598671.26	25
dtr2006	71	2114904.75	72.2	2151667.59	dtar2006	65	1989698.56	67.0	2008347.08	25
dtr3001	62	2524567.12	64.2	2555678.24	dtar3001	65	2556689.29	63.2	2579982.33	35
dtr3002	81	2492128.33	80.8	2527008.47	dtar3002	80	2485442.71	80.0	2513209.99	35
dtr3003	91	2817665.40	95.0	2857052.51	dtar3003	94	2993818.13	94.8	3012681.81	35
dtr3004	67	2529218.45	65.6	2569782.36	dtar3004	57	2224836.77	57.8	2270408.69	35
dtr3005	85	2648265.35	84.8	2690420.49	dtar3005	71	2277549.35	71.6	2293219.76	35
dtr3006	105	3029289.73	104.2	3052612.68	dtar3006	98	2879259.55	99.8	2938124.57	35
dtr4001	74	3000969.14	74.8	3028213.74	dtar4001	81	3249911.35	81.4	3283468.98	45
dtr4002	111	3448427.35	112.8	3487633.77	dtar4002	106	3362478.27	104.0	3420574.21	45
dtr4003	131	3881025.44	132.8	3956732.79	dtar4003	129	3734382.50	128.4	3790199.74	45
dtr4004	78	3059838.00	78.2	3093359.00	dtar4004	65	2919096.48	66.2	2937776.25	45
dtr4005	109	3289235.43	111.8	3334045.95	dtar4005	95	3034043.29	96.8	3094564.15	45
dtr4006	144	4107305.50	146.0	4183317.44	dtar4006	119	3589406.20	121.0	3628740.15	45
dtr5001	91	3755474.88	93.2	3803984.11	dtar5001	86	3781806.58	87.6	3814599.85	55
dtr5002	134	4016773.97	136.8	4075876.85	dtar5002	129	3963692.11	129.6	4008962.48	55
dtr5003	165	4905008.97	169.2	4956360.88	dtar5003	158	4861006.57	159.8	4949729.01	55
dtr5004	100	3984552.73	99.4	4018923.04	dtar5004	79	3513188.10	81.4	3578016.91	55
dtr5005	137	4263243.03	140.4	4311725.91	dtar5005	123	3816532.58	123.8	3906279.22	55
dtr5006	175	5107283.28	176.4	5159024.95	dtar5006	145	4249483.61	146.4	4309442.07	55
dtc501	10	274403.99	10.0	276570.48	dtac501	10	286279.85	10.0	293629.79	10
dtc502	13	286563.58	12.8	290769.46	dtac502	13	290376.42	13.0	294334.55	10
dtc503	12	295362.73	12.0	297249.13	dtac503	12	286616.68	11.6	290267.51	10
dtc504	8	233429.66	8.0	241500.14	dtac504	9	283082.56	9.0	301312.75	10
dtc505	10	270727.04	10.2	277954.95	dtac505	12	307289.68	12.6	316234.30	10
dtc506	10	251244.73	10.6	257902.22	dtac506	13	338512.78	13.0	342777.44	10
dtc1001	19	539642.21	18.6	544606.51	dtac1001	17	506584.20	17.0	518499.04	15
dtc1002	21	563385.06	21.2	568694.18	dtac1002	24	583340.17	23.6	585892.22	15
dtc1003	31	703326.38	31.6	716866.97	dtac1003	29	725742.15	29.0	739505.09	15
dtc1004	17	496647.93	17.0	508441.75	dtac1004	15	514138.24	15.0	533802.38	15
dtc1005	27	603668.78	27.6	610060.00	dtac1005	20	540241.31	20.0	549840.96	15
dtc1006	30	684398.74	30.6	694817.87	dtac1006	29	673162.25	30.0	688014.27	15
dtc2001	31	977544.06	31.6	1015527.72	dtac2001	30	1082151.57	30.4	1104254.03	25
dtc2002	43	1095830.30	44.2	1146289.74	dtac2002	43	1103674.03	43.6	1110158.11	25
dtc2003	60	1409877.23	60.2	1422090.93	dtac2003	57	1376448.38	55.2	1398989.12	25
dtc2004	29	975328.12	29.8	989078.94	dtac2004	32	1011759.59	33.2	1059172.43	25
dtc2005	41	1069204.58	42.4	1073176.92	dtac2005	40	1094857.89	40.4	1125066.43	25
dtc2006	57	1363133.79	57.0	1374376.84	dtac2006	52	1265118.69	52.4	1294353.53	25
dtc3001	52	1544340.67	52.4	1571762.66	dtac3001	45	1483029.18	45.0	1508085.61	35
dtc3002	66	1597360.13	67.2	1625941.06	dtac3002	67	1589905.10	65.8	1617411.17	35
dtc3003	88	2059577.07	88.2	2093673.41	dtac3003	84	2065789.47	85.0	2090483.86	35
dtc3004	43	1440533.40	45.4	1467857.23	dtac3004	41	1332246.51	40.0	1387295.51	35
dtc3005	69	1603475.54	69.4	1624825.51	dtac3005	65	1615453.22	64.4	1663337.87	35
dtc3006	83	2076867.30	83.0	2103065.15	dtac3006	80	1956851.31	79.8	1985634.03	35
dtc4001	61	1829955.07	61.0	1892228.36	dtac4001	56	1920984.49	55.2	1945029.28	45
dtc4002	90	2139448.80	91.8	2162985.71	dtac4002	87	2060177.08	87.2	2131542.77	45
dtc4003	125	2901733.02	126.6	2941880.75	dtac4003	118	2755109.06	116.6	2813792.62	45
dtc4004	54	1778278.25	55.4	1802755.25	dtac4004	55	1807669.90	56.4	1864604.56	45
dtc4005	91	2118274.91	91.2	2141262.62	dtac4005	83	2052917.34	83.4	2101187.76	45
dtc4006	118	2645011.77	118.8	2671897.52	dtac4006	109	2592449.26	110.2	2616629.32	45
dtc5001	63	2188278.73	65.6	2215141.65	dtac5001	63	2118154.83	66.2	2162558.90	55
dtc5002	97	2459795.50	99.0	2524590.93	dtac5002	102	2558151.65	103.8	2614840.04	55
dtc5003	152	3516777.63	151.4	3531512.03	dtac5003	139	3246479.97	139.0	3338553.20	55
dtc5004	72	2395093.39	71.8	2427058.58	dtac5004	68	2198083.48	69.6	2241442.99	55
dtc5005	110	2664649.61	113.6	2706279.21	dtac5005	106	2614577.18	107.2	2665146.81	55
dtc5006	155	3573142.96	157.4	3629362.74	dtac5006	130	3187680.57	131.6	3246432.25	55

Table 5.14: The results of our algorithm on our Benchmark. The best result over 5 runs is listed.

5.11 Real world problems

An important aspect of our research is real world applicability. In addition to be able to solve theoretical problems we want our algorithm to be useful in real world problems. That is why we based our problem on a real world case of a precision transport company. Such a problem is too complex to include all the details in an algorithm or model. There is always a tradeoff between detail and efficiency: incorporating too much detail in the model makes the search space larger, and thus makes it harder to find a good solution in a reasonable amount of time; too little detail could lead to creating unusable schedules, because important constraints are ignored.

The problem we solve is based on the data of a busy week of this precision transport company. This company has three depots at which 50 vehicles are stationed. In this planning window 1202 orders have to be serviced by these vehicles. This comes down to around 240 orders per day. We have shown that our algorithm is capable of solving problems of this size. The orders have a pickup and a delivery location, and can have one or more possible time windows in which they can be serviced. We extended the model to include alternative locations as well. Vehicles are driven by drivers who take a co-driver with them if they need to deliver heavy orders. These drivers and co-drivers have a maximum amount of time that they can work on a day. Also they have the right to take a break somewhere during the day. We chose to exclude the planning of the drivers from our model. Because in this situation it does not matter which driver drives a trip, the scheduling of the drivers can be done afterwards. This does not mean that we can ignore the constraints arising from the needs of the drivers. We chose to include the maximum time for shifts, and exclude the drivers breaks. Driver breaks are not included in our model, for the reason that they do not have a major impact on the schedules that are made. If there is waiting time between two orders, this can be used by the drivers to take a break. The drivers breaks can be split up in 15 minute periods, which would have a relatively small impact on the total schedule. Limits on shift time however have a larger impact on the produced schedules. This is because the vehicle has to be returned to the depot when changing shifts, to be able to exchange drivers. This means that the route for the vehicle has to be adjusted, to include a stop at the depot. A route created without this constraint may be very inefficient, or impossible to drive, in practice.

Travel time is an important aspect of the problem. A good estimation of travel time can lead to better results, and less delay. Travel times can change significantly during the day. Because we make a schedule in advance, we can not use real time data to adjust the schedule. Nonetheless travel times can be estimated based on historical data. For example traffic jams, or slower traffic due to bad weather conditions can be estimated to a certain extent. Our algorithm is able to handle time-dependent travel time, of which the input would be an estimation, to create more realistic schedules.

For this problem service times may be high, which means they can have a large

impact on the solutions. We allow for service time to be given for every pickup or delivery separately. A good estimation for these service times is important for creating good schedules.

Our algorithm can be used as a basis for solving large real world pickup and delivery problems in a reasonable amount of time. The extensions we made, based on this real world case, improve the usability of our algorithm for practical problems.

Chapter 6

Conclusion & Future Research

6.1 Conclusion

A lot of research has been done on the Pickup and Delivery problem with time windows. In our research we have extended the approaches for solving the basic PDPTW to be able to handle practical situations better. To reach this goal we proposed a method for handling time dependent travel times and a method for limiting the duration of a shift. In addition our algorithm allows for alternative pickup and delivery locations and times from which a choice can be made. These extensions make it possible to handle a wider range of problem instances, and to generate more realistic solutions.

The algorithm we created for solving these problems is a simulated annealing algorithm. A number of extensions to this algorithm were made to be able to reach a good solution faster, and to have a better chance of escaping local optima. We do not make completely random changes to the solution as in a regular simulated annealing approach. Instead we give the algorithm a little guidance towards better neighbours. The danger of this approach is that the algorithm could get stuck in local optima faster, because we allow less freedom for exploring the search space. To overcome this problem, we do not always select the best neighbours. Neighbours are selected by probability, based on their fitness value. Also operators that search a larger neighborhood are introduced. We chose to add guidance to the algorithm because of the many constraints the time windows pose. Randomly shifting orders would often lead to much worse solutions, and increasing the number of iterations needed to find a good solution.

To make the algorithm more efficient we have added a similarity factor, which determines the similarity between orders, actions or trips. This factor is used to select orders that are more alike in neighborhood moves, so that less invalid moves are made. We introduced a new initialization method based on this similarity value, which proved to be competitive. Also, integrating similarity in some of the operators proved to be an improvement: Better results were found in a shorter amount of time.

The main research subject was time dependent travel time. Our implementation of time dependent travel time is very flexible and can handle many types of input, as long as the travel times follow the FIFO principle. The input can be either discrete values, or a function or algorithm that calculates travel times on the fly. The implementation is based on the approach by Savelsbergh [1992], in which slack values are used. We have adapted this approach to be able to handle travel times that can change during a day, by introducing *time dependent slack*. The only disadvantage is that the computing times increase when working with time dependency. Results of our experiments (see Section 5.7) show that the computing time for the same number of iterations increases by factors from around 5 to 12, when considering one possible starting time for newly inserted actions.

In addition to the time dependency we added the possibility of having alternatives for pickups and deliveries. This addition can be used in practical situations like the possibility of delivering an order at two possible dates, or making a distinction between home and work address for a package to be delivered. Because all combinations of pickups and deliveries have to be considered when inserting an order, the computing times increase if there are more alternatives available. We have tested problem instances with up to 4 alternatives per action in Section 5.8. The computing time for solving problems with alternatives increases in a quadratic fashion. If all orders have two alternative pickup locations and two alternative delivery locations, there are four possible combinations per order. If we have 3 alternatives per pickup and per delivery this increases to nine possibilities, and so on.

In real world problems there are restrictions on the maximum time a driver may work on a day. We introduced an approach for handling this constraint. We introduced shifts that start and end at a depot, and have a maximum duration. A day is divided into a number of shifts, that are driven by different drivers. The depot is visited in between shifts to exchange drivers. Orders can be divided between shifts to in such a way that the maximum working time constraint is respected. The experiment in Section 5.9 shows that the more shifts there are the more computing time it costs to find a good solution. Also the total distance driven increases, because the vehicles have to return to the depot in between shifts.

For the PDPTW without time dependent travel time there was a benchmark by Li and Lim [2003] to which a comparison could be made. The results our algorithm found were often close to the best found solutions, although for some of the problem instances a solution with the minimum amount of vehicles used was not found. Because to the best of our knowledge there was no benchmark available for the PDPTW with time dependent travel times, or alternatives for the pickups and deliveries, we created a new benchmark so the quality of algorithms with these extensions can be compared. For all the problem instances in our benchmark set we have noted the best result found so far (see Table 5.14, for use in future research.

Although the algorithm we created is quite competitive and can be used as a foun-

dation for solving real world PDPTW problems, there are still many improvements possible. For example, running the algorithm with time dependency requires a lot of processing time. There are many possibilities for increasing efficiency in this algorithm. Smarter ways of recalculating fitness values based on changes to a solution, or faster methods for estimating the change in fitness when inserting or moving orders could improve the speed. Checking for order insertion possibilities is currently the largest bottleneck in speed, so it would be beneficial to make this more efficient.

6.2 Future Research Possibilities

There are some aspects in our research that can be improved by doing more extensive research. We also name a number of possible extensions that can help with solving realistic problems.

Firstly, our way of dealing with heterogenous vehicles is very simple, and can certainly be improved if researched further. We choose vehicles based on their cost, which is usually lower for smaller vehicles. Because of this, the algorithm prefers smaller cheaper vehicles over larger one. For servicing one order the small vehicle may be preferable, but if fewer larger vehicles could service a group of orders the larger vehicles might be more efficient, even though they are more expensive to use. To overcome this, we have an operator that tries to combine two similar routes into one. If two routes are driven by small vehicles, they can possibly be combined in one route for a large vehicle. A method that finds out the most efficient use of vehicles in a more sophisticated way could improve the solutions.

Furthermore more research can be done for maximum shift time. Our method creates a fixed number of shifts for a day with known duration, so it can never be used in problem instances with an unlimited planning window. Also this method can become inefficient for problems with long planning windows, because there would be many shifts to consider. A more flexible approach to this problem could be better and more efficient. This could be done by allowing shifts to be created and removed on the fly.

In addition to the limits on shift times, drivers need breaks during their day. Integrating breaks into the scheduling process is useful for creating schedules that comply to the conditions for drivers which need to be satisfied.

In practice often not everything is known at the time the schedule is made: Travel times can change due to unexpected events and/or orders can be added or cancelled. To cope with these events we have to be able to adjust the schedule to these dynamic changes. Drivers can not be expected to change their whole schedule every time something changes. This can be solved by applying a simulated annealing algorithm again, with penalties for deviations from the original solution. Everything that happened before the current time can not be modified anymore. The further

in the future modifications are made, the less they matter, and the lower their deviation penalty should be. For better applicability to real world cases, this extension would be a good research subject.

Every real world case has its own requirements. There are many more extensions that can be researched for use in these problems such as more realistic ways of describing capacity, schedules adapted to the skills and requirements of the personnel, real time modifications after a solution is created, methods for estimating pickup and delivery duration etc.

Bibliography

BIBLIOGRAPHY

Roberto Baldacci, Enrico Bartolini, Aristide Mingozzi, and Roberto Roberti. An exact solution framework for a broad class of vehicle routing problems. *Computational Management Science*, 7(3):229–268, 2010.

Cynthia Barnhart, Ellis L Johnson, George L Nemhauser, Martin WP Savelsbergh, and Pamela H Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations research*, 46(3):316–329, 1998.

Rémy Chevrier, Arnaud Liefooghe, Laetitia Jourdan, and Clarisse Dhaenens. Solving a dial-a-ride problem with a hybrid evolutionary multi-objective approach: Application to demand responsive transport. *Applied Soft Computing*, 12(4):1247–1258, 2012.

G Clarke and JW Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964.

Jean-François Cordeau and Gilbert Laporte. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B: Methodological*, 37(6):579–594, 2003.

Luca Coslovich, Raffaele Pesenti, and Walter Ukovich. A two-phase insertion technique of unexpected customers for a dynamic dial-a-ride problem. *European Journal of Operational Research*, 175(3):1605–1615, 2006.

Alberto V Donati, Roberto Montemanni, Norman Casagrande, Andrea E Rizzoli, and Luca M Gambardella. Time dependent vehicle routing problem with a multi ant colony system. *European journal of operational research*, 185(3):1174–1191, 2008.

Billy E Gillett and Leland R Miller. A heuristic algorithm for the vehicle-dispatch problem. *Operations research*, 22(2):340–349, 1974.

Fred Glover. Tabu search-part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

Soumia Ichoua, Michel Gendreau, and Jean-Yves Potvin. Vehicle dispatching with time-dependent travel times. *European journal of operational research*, 144(2): 379–396, 2003.

David S Johnson, Cecilia R Aragon, Lyle A McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning. *Operations research*, 37(6):865–892, 1989.

Scott Kirkpatrick, D. Gelatt Jr., and Mario P Vecchi. Optimization by simmulated annealing. *science*, 220(4598):671–680, 1983.

Jan Karel Lenstra and AHG Rinnooy Kan. Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227, 1981.

Haibing Li and Andrew Lim. Local search with annealing-like restarts to solve the vrptw. *European journal of operational research*, 150(1):115–127, 2003.

Quan Lu and Maged M Dessouky. A new insertion-based construction heuristic for solving the pickup and delivery problem with time windows. *European Journal of Operational Research*, 175(2):672–687, 2006.

Ying Luo and Paul Schonfeld. A rejected-reinsertion heuristic for the static dial-a-ride problem. *Transportation Research Part B: Methodological*, 41(7):736–755, 2007.

Jean-Yves Potvin and Jean-Marc Rousseau. An exchange heuristic for routeing problems with time windows. *Journal of the Operational Research Society*, 46 (12):1433–1446, 1995.

Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation science*, 40(4):455–472, 2006.

Mustafa Şahin, Gizem Çavuşlar, Temel Öncan, Güvenç Şahin, and Dilek Tüzün Aksu. An efficient heuristic for the multi-vehicle one-to-one pickup and delivery problem with split loads. *Transportation Research Part C: Emerging Technologies*, 2012.

Martin WP Savelsbergh. The vehicle routing problem with time windows: Minimizing route duration. *ORSA Journal on Computing*, 4(2):146–154, 1992.

Martin WP Savelsbergh and Marc Sol. The general pickup and delivery problem. *Transportation science*, 29(1):17–29, 1995.

Michael Schilde, Karl F Doerner, and Richard F Hartl. Metaheuristics for the dynamic stochastic dial-a-ride problem with expected return transports. *Computers & Operations Research*, 38(12):1719–1730, 2011.

Gerhard Schrimpf, Johannes Schneider, Hermann Stamm-Wilbrandt, and Gunter Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159(2):139–171, 2000.

Paul Shaw. A new local search algorithm providing high quality solutions to vehicle routing problems. *APES Group, Dept of Computer Science, University of Strathclyde, Glasgow, Scotland, UK*, 1997.

M Sol and Martin WP Savelsbergh. *A branch-and-price algorithm for the pickup and delivery problem with time windows*. Eindhoven University of Technology, Department of Mathematics and Computing Science, 1994.

Pandhapon Sombuntham and Voratas Kachitvichayanukul. A particle swarm optimization algorithm for multi-depot vehicle routing problem with pickup and delivery requests. In *Proceedings of the International MultiConference of Engineers and Computer Scientists 2010*, volume 3, 2010.

A Subramanian, LMA Drummond, C Bentes, LS Ochi, and R Farias. A parallel heuristic for the vehicle routing problem with simultaneous pickup and delivery. *Computers & Operations Research*, 37(11):1899–1911, 2010.

Craig A Tovey. Hill climbing with multiple local optima. *SIAM Journal on Algebraic Discrete Methods*, 6(3):384–393, 1985.

Nathalie Van Zeijl. Including practical information in route planning applications. Master's thesis, Erasmus Universiteit Amsterdam, 2013.

Issam Zidi, Khaled Mesghouni, Kamel Zidi, and Khaled Ghedira. A multi-objective simulated annealing for the multi-criteria dial a ride problem. *Engineering Applications of Artificial Intelligence*, 2012.