MASTER

Modeling and solving a real-life load building and routing problem in the retail industry

van Bemmelen, M.M.K.J.

Award date:
2012

Link to publication
Master’s Thesis
Modeling and Solving a Real-life Load Building and Routing Problem in the Retail Industry

M.M.K.J. van Bemmelen

August 23, 2012
Acknowledgements

First and foremost I would like to thank my supervisor Wim Nuijten for his ideas, guidance and contributions throughout this project. Especially his critical comments on the many draft versions of the report where much appreciated.

Also, I would like to thank Tom van Woensel for his input on which constraints to select, and Duygu Tas, for helping me build the MIP model.

Finally I would like to thank my friends and family for their continued support throughout this project.
Abstract

The research on routing problems reported on in literature is often involved with relatively simple models, where a lot of the complexity as found in practice is left aside. In this thesis we model and solve a real-life load building and routing problem, where amongst others we handle having a heterogeneous fleet, site resources, hard time windows, shipment to shipment incompatibility, reverse logistics, time-dependent travel times, precedence constraints between shipments, multiple depots, and variable unloading times. We solve the problem of both deciding on the load building, i.e., deciding on how shipments are combined, and of deciding on the routing, i.e., deciding which shipments are picked up and delivered by which truck and when. For this we have developed a Scatter Search algorithm, as well as an exact solution method that uses Mixed Integer Programming (MIP) to solve the problem. Because of the complexity of the real-life instances, the exact algorithm is tested only on small instances. We present a computational study where we show that the MIP model is able to solve instances of up to 15 shipments to optimality in 2.5 hours and that the Scatter Search algorithm we developed generates feasible schedules of excellent quality in limited computation time both for well-known academic instances as for large, realistic instances.
Contents

1 Introduction 4

2 Problem description 6
  2.1 Mathematical representation 6
  2.2 MIP model 10
  2.3 Complexity 12

3 Solution method 13
  3.1 Insertion heuristic 14
  3.2 Local Search 17
  3.3 Scatter Search 20
  3.4 Restricted Dynamic Programming 20
    3.4.1 Dynamic Programming for the TSP 21
    3.4.2 Will it work for the TD-VRPTW? 21
    3.4.3 Implementation 22

4 A Real-life TD-VRPTW Problem 24
  4.1 Base instance 24
  4.2 Instances 26

5 Computational results 28
  5.1 Parameters 28
    5.1.1 n-insertion 28
    5.1.2 Scatter Search 31
  5.2 Solomon benchmarks 34
  5.3 TD-VRPTW 37
  5.4 MIP Model 38
  5.5 Restricted Dynamic Programming 39

6 Conclusion 40

A Notations 44

B MIP model 45
Chapter 1

Introduction

The Vehicle Routing Problem (VRP) was introduced by Dantzig and Ramses in 1959, and is an important problem in the field of logistics. Since its introduction many heuristics and exact algorithms have been proposed to solve the VRP. This thesis studies a generalization of the Capacitated Vehicle Routing Problem (CVRP), which can be described as follows (Laporte [2009]). Given:

- A single warehouse
- A set of customers
- A set of demands requiring a certain capacity
- A set of arcs between the warehouse and customers and between customers
- A cost per arc (can be travel time, distance, or travel cost)
- A vehicle with maximum capacity

The problem to be solved is then to determine a set of cost-minimizing routes such that:

- All routes start and end at the warehouse
- All customers with a non-zero demand are visited exactly once
- For each of the routes the total required capacity of the demands does not exceed the vehicle capacity

The goal of this project is to model and solve a realistic version of the VRP with a broad set of constraints, which has not been studied in literature. Both the complexity of the problem, in terms of the constraints, and the size of the instances solved, are new. To solve the instances we use a Scatter Search algorithm based on literature, and within its framework we developed a First Solution Heuristic (FSH) based on the insertion heuristic as described by [Solomon, 1987], and a Local Search algorithm based on [Braysy and Gendreau, 2005a]. Apart from the Scatter Search algorithm, we developed a MIP model
that can produce exact solutions to the problem, and a Restricted Dynamic Programming algorithm based on [Gromicho et al., 2008].

There are many papers describing the classic VRP and its variants. There is however limited research done on real-life load building and routing problems, and as such, high quality papers on realistic versions of the VRP are rather scarce. A well studied extension to the VRP is the Vehicle Routing Problem with Time Windows (VRPTW). [Braysy and Gendreau, 2005a,b] and Desaulniers et al. [2008] present overviews of the most commonly used algorithms to solve the VRPTW. The VRPTW is however still a rather academic problem. A more real-life problem is studied by [Ceselli et al., 2009] who describe a column generation algorithm that solves the VRP with many additional requirements and constraints, including multiple vehicle capacities, time windows associated with depots and customers, incompatibility constraints between goods, depots, vehicles, and customers, maximum route length and duration, upper limits on the number of consecutive driving hours and compulsory drivers rest periods, the possibility of skipping some customers and using express courier services instead of the given fleet to fulfill some orders, the option of splitting up the orders, and the possibility of open routes that do not terminate at depots. This problem was solved for instances with up to 100 customers. Another real-life problem is studied by [Belfiore and Yoshizaki, 2009], which solves a problem as it was stated by a major Brazilian retail group, solving the problem with constraints including having a heterogeneous fleet, vehicle to site incompatibility, time windows and split deliveries. This problem was solved using the Scatter Search algorithm, for problem sizes up to 519 customers. A comparison between the routes actually covered by the company, and the solution created by the algorithm showed significant improvement in terms of total distribution cost.

An important real-life property found in transportation problems in the retail industry are time-dependent travel times, also known as dynamic travel times, where travel time depends on the time of departure, modeling traffic conditions such as rush hours. The VRP with dynamic travel times is described by van Woensel et al. [2008], using queueing theory to obtain travel times, Kok et al. [2011] proposing a post-processor to determine optimal departure times for the vehicle routes, and [Kuo et al., 2009], which introduces a separate calculation model to calculate the total operation time of all vehicles, and uses a modified Tabu Search to optimize the sequence of customers visited in the routes. Bettinelli et al. [2011] describes a version where multiple warehouses are considered, using a branch-and-cut-and-price algorithm. As the problem we study is NP-hard (Section 2.3), the computation time to solve an instance to optimality increases exponentially with the size of the instance. Because of this we will use heuristics rather than exact algorithms. In this project we have used Scatter Search to solve the Time Dependent Vehicle Routing Problem with Time Windows (TD-VRPTW, described in chapter 2), which is chosen because of its relative ease of implementation, and its effectiveness on complex routing problems as shown in [Belfiore and Yoshizaki, 2009], and the quality of its solutions as shown in [Russell and Chiang, 2006].
Chapter 2

Problem description

In this thesis we study a problem as found in practice in the retail industry, where we have multiple customers that each can have multiple orders in a single day, where multiple warehouses serve these customers using a heterogeneous fleet of vehicles, and incompatibility constraints exist between goods, vehicles, and customers. Furthermore a strict precedence constraint exists between frozen and ambient goods, as some vehicles can carry both by means of a movable wall separating a freezer compartment from a compartment with an ambient temperature. In this way, ambient goods always need to be delivered before frozen goods in the same tour. Backhauls are also included, with an identical precedence constraint, since vehicles can only start picking up backhaul items when all other goods have been delivered. Warehouses are assumed to contain all goods, and a tour starts and ends at the same warehouse. In the solution we decide for every shipment from which warehouse and in which route it is delivered. Decisions are made based on the cost, which should be minimized in the solution. The next section describes a mathematical representation of the problem stated above.

2.1 Mathematical representation

An instance of the Time Dependent Vehicle Routing Problem with Time Windows and Reverse Logistics (TD-VRPTW) consists of the following data:

- A set of vehicle types $P$, with for every $p \in P$ a capacity $q_p$, a fixed cost $c_{f}^{p}$ giving the cost to use one vehicle of type $p$, a cost per unit of distance $c_{d}^{p}$, a cost per time unit $c_{t}^{p}$, and a type $y_{p}$ indicating what type of shipment the vehicle can transport (only frozen, only ambient, or both).

- A set of warehouses $W$ from which the vehicles can depart. It is assumed that a vehicle visits exactly one warehouse per tour such that it starts and ends its route at
the same warehouse.

- A set of destinations $D$, each $d \in D$ with a maximum allowed load $MQ_d$ indicating the capacity of the largest vehicle that is permitted to deliver shipments. Furthermore, each destination $d \in D$ has a resource $sr_d$ indicating the number of vehicles that can load or unload simultaneously, and a binary variable $w_d$ which is 1 when the destination is a warehouse, and 0 otherwise. Note thus that $W \subseteq D$. Furthermore, each destination $d \in D$ has a Fixed Location Delay ($FLD_d$).

- A set of arcs $A$, with for every pair $(i, j) \in D \times D$ a distance $s_{ij}$ and a travel time $tt_{ijt}$, travel times are defined in 10 minute intervals, and as such $tt_{ijt}$ is given by the interval in which departure time $t$ falls.

- A set of shipments $I$, where each $i \in I$ has an amount of items $N_i$, and a type $y_i$ indicating whether the shipment is ambient, frozen, or a reverse shipment to the warehouse. Each shipment has a destination $d_i$ and a hard time window given by $rd_i$ (release date), and $dd_i$ (due date). Furthermore, the time spent unloading the items of each shipment $i \in I$ is given by $ST_i$ (service time) and is considered to be a constant time per item, and as such is directly linked to $N_i$. $SIS$ is the Shipment Incompatibility Set which is a set of shipment pairs where a pair $(i, i') \in SIS$ denotes that shipments $i$ and $i'$ cannot be transported in the same vehicle.

A solution to an instance of the TD-VRPTW consists of a number of routes $R$ which for each route $r \in R$ consist of a set of destinations $D_r$ that are to be visited, a set of arcs $A_r$ to be traversed and a set of shipments $I_r$ that are to be delivered. Each route is divided in three subroutes, an ambient route $r_a$, a frozen route $r_f$ and a backhaul route $r_b$, such that $A^a_r, A^f_r$, and $A^b_r$ respectively denote the arcs of the ambient, frozen, and backhaul subroutes of $r$. Destinations are assumed to be visited at most once per subroute. Furthermore, each route has a vehicle type $v_r \in P$.

To ensure that all constraints are met, a definition for the start of service and departure time at the destinations is needed. Since destinations can have multiple shipments delivered to them, $\alpha_i$ denotes the time the vehicle starts to deliver shipment $i$ at the destination. Similarly $\delta_i$ denotes the time at which the vehicle can leave the destination where it has delivered shipment $i$. Start of service time thus becomes:

$$\alpha_j = \max((\delta_i + tt_{di,d_j,i}, rt_i) + SRD$$

Where $i$ is the shipment that has been delivered directly prior to shipment $j$ in the same route, and $SRD$ (Site Resources Delay) is the total time spent waiting on other vehicles that are delivering shipments at this destination ($d_j$). Departure time can be calculated as:

$$\delta_i = \alpha_i + FLD_{di} + ST_i$$
Note that vehicles do not necessarily leave the depot at time 0, since shipments can only be delivered after the start of their time window \((rt_i)\). The quality of the solution is determined by the total cost of the resulting schedule. The total time spent in a route is the time between the departure time at the warehouse and the arrival time at the warehouse at the end of the route, and is denoted as \(t_r\) and the total distance (the sum of all arcs traversed in the route) as \(s_r\). The total cost of a schedule thus becomes:

\[
\sum_{r \in R} c^f_{vr} + t_r \cdot c^t_{vr} + s_r \cdot c^s_{vr}
\]

Now that all necessary variables have been defined, a more formal definition of the constraints can be formulated:

- All shipments are delivered, i.e. we must ensure that each shipment in the input data is delivered in one of the routes.
  \[
  \bigcup_{r \in R} I_r = I
  \]

- We also need to make sure that no shipment is delivered more than once, since the previous constraint only ensures that all shipments are delivered at least once,
  \[
  \sum_{r \in R} |I_r| = |I|
  \]

- No vehicle carries more items than its capacity, i.e. for each route \(r \in R\)
  \[
  \sum_{i \in (I^a_r \cup I^f_r)} N_i \leq q_{vr}
  \]
  and
  \[
  \sum_{i \in I^b_r} N_i \leq q_{vr}
  \]

- Precedence constraints between ambient, frozen, and backhaul shipments, i.e. for every route \(r \in R\), every ambient, frozen and backhaul shipment \((i \in I^a_r, j \in I^f_r \text{ and } k \in I^b_r)\) in that route, it must hold that
  \[
  \alpha_i \leq \alpha_j \leq \alpha_k
  \]

- Vehicle to site incompatibility, i.e. for each route \(r \in R\), and every destination \(d \in D_r\) visited in that route
  \[
  q_{vr} \leq MQ_d
  \]
• Hard time windows, for each shipment $i \in I$

\[ rd_i \leq \alpha_i \leq dd_i \]

• Shipment to shipment incompatibility, i.e. for each route $r \in R$, and every two shipments $i, i' \in I_r$ delivered in that route

\[ (i, i') \notin SIS \]

• Site resources: Let $R_{dt}$ be the set of routes in which the vehicle serving route $r$ is present at destination $d \in D$ at time $t \in \mathbb{N}$, i.e. there is a shipment $i \in I_r$ such that

\[ d_i = d \land \alpha_i \leq t \leq \delta_i \]

The constraint then says that for each $d \in D$ and $t \in \mathbb{N}$

\[ |R_{dt}| \leq sr_d \]
2.2 MIP model

The following model is a Mixed Integer Programming (MIP) model. Using MIP technology requires that a problem is expressed in terms of linear constraints over numerical decision variables where a linear objective function is to be optimized. A solution to the model consists of an assignment of values to the decision variables. In the case of the TD-VRPTW, the linear constraints obviously express the constraints that the resulting vehicle schedule must adhere to, while the objective function describes that the combined cost of all routes should be minimized. In the model we present here, the shipments and warehouses are modeled as nodes in a graph. The most important decision variable is $X_{ijv}$, which is a binary variable that has the value 1 if vehicle $v$ visits node $j$ immediately after node $i$, and 0 otherwise. Continuous variable $S_{iv}$ models the start of service time of vehicle $V$ at node $i$. The other variables are composed of the input data, where $V$ is the set of vehicles and $q_v$ is the maximum capacity of vehicle $v$. The set $I$ is the set of all shipments, and is composed of the sets $A$, $F$, and $B$, respectively the Ambient, Frozen, and Backhaul shipments. The set of warehouses is denoted by $W$, which makes the total set of all nodes $N$ the union between sets $W$ and $I$. The set $C$ denotes the set of customers, where each $c \in C$ has a set of nodes from the set $I$, to model that customers can have multiple shipments. $MQ_c$ denotes the size of the largest vehicle that may deliver any shipment of customer $c$, and $SR_c$ denotes the maximum amount of vehicles that may be present at any given time at the nodes of customer $c$. $T$ denotes the scheduling horizon, and ranges between 0 and 1020 minutes. This scheduling horizon is divided a set of intervals $TI$, where $b_{ti}$ and $e_{ti}$ denote the beginning and end of interval $ti \in TI$, and $tt_{ij}$ denotes the travel time between node $i$ and $j$ when leaving in interval $t$. $FLD$ denotes the fixed delay at each node, and $ST$ denotes the extra delay per delivered item.

- The objective of the MIP model is to minimize
  \[ \sum_{v \in V} \sum_{i \in N} \sum_{j \in N} c_{ij} \cdot X_{ijv} \]

Such that:

- All items are delivered:
  \[ \sum_{v \in V} \sum_{j \in N} X_{ijv} = 1, \text{ for all } i \in I \]

- Each vehicle must leave exactly one warehouse:
  \[ \sum_{w \in W} \sum_{j \in N} X_{wjv} = 1, \text{ for all } v \in V \]

- Each vehicle must arrive at exactly one warehouse:
  \[ \sum_{w \in W} \sum_{i \in N} X_{iwv} = 1, \text{ for all } v \in V \]

- Each vehicle arrives at the same warehouse it left from (cyclic routes):
  \[ \sum_{j \in I} X_{wjv} - \sum_{i \in I} X_{iwv} = 0, \text{ for all } v \in V \text{ and } w \in W \]
• No loops, i.e. a node can not be visited directly after itself: 
  \( X_{iiv} = 0, \) for all \( i \in I \) and \( v \in V \)

• Capacity constraints: 
  \( \sum_{i \in I} q_i \sum_{j \in N} X_{ijv} \leq q_v, \) for all \( v \in V \)

• Time window constraints: 
  – Start of service, i.e. the earliest possible time vehicle \( v \) can start delivering the shipment at node \( j \) is the sum of the start of service time at the previous node \( i \), the travel time between the two nodes, and the delay at node \( i \). This should of course only hold if node \( j \) is visited directly after node \( i \), which is handled by the term \( M_{ij}(1 - X_{ijv}) \), in which \( M_{ij} \) is a constant that is large enough to cancel out the rest of the term.

  \[
  S_{iv} + tt_{ij,v} + FLD + q_i \cdot ST - M_{ij}(1 - X_{ijv}) \leq S_{jv}, \text{ for all } i, j \in N \text{ and } v \in V
  \]

  Note that \( tt_{ij,v} \) was inserted here for readability, in the model, travel time depends on the time the vehicle has finished delivering the shipment, i.e. the time it leaves the destination. Furthermore, the traveltime values are stored in an array of size \( N \times N \times \text{intervals} \), and a decision variable can not be used to index arrays. The actual travel time is therefore given by:

  \[
  \sum_{t_i \in T_i} (tt_{ij,ti} \cdot (b_{ti} \leq S_{iv} + FLD + q_i \cdot ST \leq e_{ti}))
  \]

  – Hard time windows: 
    \( rt_i \leq S_{iv} \leq dd_i \), for all \( i \in N \) and \( v \in V \)

• Precedence constraints between ambient, frozen and backhaul shipments: 
  \( S_{i_v} \leq S_{j_v} \leq S_{k_v}, \) for all \( i \in A, j \in F, k \in B, \) and \( v \in V \)

• Shipment to shipment incompatibility: 
  \( (\sum_{i \in N} X_{i_kv} + \sum_{j \in N} X_{j_lv}) \leq 1, \) for all \( v \in V, \) and \((k,l) \in SIS\)

• Vehicle to customer incompatibility: 
  \( \sum_{i \in N} X_{ijv} \cdot q_v \leq MQ_c, \) for all \( v \in V, \) and \( c \in C \) where \( j \in c \)

• Site resources, i.e. for every time \( t \) it should hold that the amount of vehicles with a start of service time less than \( t \), and a departure time higher than \( t \), is less than, or equal to, \( SR_c \):

  \[
  \sum_{v \in V} \sum_{i \in c} ((S_{iv} \leq t \leq (S_{iv} + FLD + q_i \cdot LD)) \cdot X_{ijv}) \leq SR_c \text{ for all } t \in T, c \in C, \text{ and } j \in N
  \]

  The above constraint is however not linear, since we multiply a decision variable
\( X_{ijv} \) with an expression that depends on an other decision variable \( S_{iv} \). Linearizing results in the following constraint:

\[
\sum_{v \in V} \sum_{i \in C} (((S_{iv} \leq t) + ((t \leq (S_{iv} + FLD + q_i \cdot LD)) == 2) - (1 - X_{ijv})) \leq SR_c \text{ for all } t \in T, c \in C, \text{ and } j \in N.
\]

### 2.3 Complexity

To prove that the standard VRP is NP-hard, one needs to prove that a problem that is known to be NP-hard can be reduced to it. It is trivial to see that the Traveling Salesman Problem (TSP) is a special case of the TD-VRPTW, and since the TSP is proven to be NP-hard [Garey and Johnson, 1979], the TD-VRPTW is also NP-hard. To prove NP-completeness, one needs to also show that the problem is in NP, which means that a solution can be verified in polynomial time. The decision version of the TD-VRPTW (given a cost \( c \) is there a solution with a cost less than \( c \)) is easily shown to be in NP, since verifying the solution can be trivially done in polynomial time.
Chapter 3

Solution method

Due to TD-VRPTW being NP-hard and the size of the real-life instances, solving these instances to optimality is generally considered infeasible. It is therefore common practice to use heuristics to find a good solution within acceptable time. Amongst the most popular heuristics is the Local Search (LS) algorithm and its many variants, which alter a given initial solution by applying changes that affect the solution locally. Possible next solutions are selected from the solution neighborhood, where the solution neighborhood consists of the solutions that can be reached in one move, where a move is defined to be a change in the solution, for instance changing the order in which some shipments are delivered in a route. Hill climbing is a basic version of LS which stops when no improvements can be made anymore. We then say a local optimum is reached.

When the solution quality of local optima is insufficient, methods can be used to escape local optima, and improve the solution further. An example of such a method is Tabu Search. Tabu Search keeps a list of moves that are marked ‘tabu’, which usually includes recent solutions that could bring you back to solutions you recently visited. This aims to ensure that the algorithm does not find the same solution repeatedly, including thus getting stuck in a local optimum.

Alternate methods to escape local optima are meta-heuristics, which are heuristics that guide other heuristics in the best direction. An example of such a meta heuristic is Scatter Search, and is used in [Belfiore and Yoshizaki, 2009] to solve a large, real-life VRP that includes time windows and split deliveries. Russell and Chiang [2006] also uses Scatter Search to compute solutions to the well known Solomon benchmarks [Solomon, 1987], and proves that it is comparable to the (then) newest solution methods.

The Scatter Search principle is to combine two solutions to form one new solution, much like Genetic Algorithms, see ([Beasley et al., 1993a] and [Beasley et al., 1993b]). The intuition is that if a part of a route is present in two good solutions, it is assumed to be a cost effective partial route, and is therefore added to the new route. The Scatter Search heuristic is discussed in detail in Section 3.3.

For the TD-VRPTW, we propose an implementation of the Scatter Search algorithm, using LS to improve individual solutions, and Scatter Search to combine several solutions to create a new best solution. Below, an overview of the proposed algorithm is shown.
1. Generate a set of initial solutions, using the insertion heuristic as described in Section 3.1
2. Improve these solutions using LS, which is described in Section 3.2
3. Combine each pair of improved solutions using the Scatter Search principle, saving only $n$ best solutions
4. Iterate step 2-3 until a maximum number of iterations is reached
5. Return the best solution in the set

A computational study has been done to determine the optimal values for the amount of initial solutions, the amount of solutions combined in step 3, and the total amount of iterations of the algorithm, and can be found in Section 4.3.

3.1 Insertion heuristic

In order to obtain a first solution, the insertion heuristic as proposed by Solomon is adapted to incorporate the additional constraints. The heuristic works by inserting shipments one by one in the ‘best’ position of the current partial solution. The best position is determined by a weighted sum:

$$w_1 \ast ad + w_2 \ast at + w_3 \ast \text{ind}$$

where $ad$ stands for the added distance, $at$ stands for added time, and $\text{ind}$ stands for the impact on succeeding shipments, since the insertion of a shipment can cause shipments further in the route to be delivered later. $\text{ind}$ is defined to be the sum of all these differences in delivery time for succeeding shipments. By varying the weights, several different initial solutions can be obtained. The order in which the shipments are inserted can also have an impact on the quality of the first solution, so before shipments are inserted they are sorted based on again a weighted sum:

$$w_4 \ast dw + w_5 \ast tww + w_6 \ast rd + w_7 \ast N_i$$

where $dw$ stands for the distance to the warehouse, $tww$ stands for the width of the time window, $rd$ stands for the release date, and $N_i$ is the number of items in the shipment. The heuristic works as follows. First, the cost of adding a new route in which the shipment is delivered is calculated, after which the cost is calculated for every possible position where the shipment can be inserted. Every arc in the partial solution is a possible position to insert the new shipment, and inserting a shipment is done by replacing such an arc by two new arcs, connecting the starting point of the original arc to the destination of the inserted shipment, and the destination of the inserted shipment to the end point of the original arc. The partial solution is then updated with the cheapest option, and the next shipment is inserted, until no more shipments are left to insert. Since we have a site resources constraint, which creates a dependency between routes, it is not guaranteed that a
shipment can be inserted in a new route or in any position in the partial solution, even if a feasible solution does exist. Intuitively, if a feasible solution exists, there exists a feasible set of routes for each destination in which all shipments to that destination are delivered without violating constraints. If the situation as described above occurs, i.e. a shipment can not be inserted in the current partial solution, all shipments that are to be delivered to its destination are removed from the partial solution, and the set of feasible routes for that destination is created and added to the partial solution. This guarantees a feasible initial solution.

Handling the constraints in the TD-VRPTW:

- Heterogeneous fleet:
  When adding a new route, the cheapest vehicle that can transport the shipment is selected. When updating a route, the capacity of the current vehicle is checked, and a larger vehicle is selected as needed, extra cost is the difference between the fixed cost of both vehicles.
  Complexity: linear in the number of vehicle types $O(|P|)$

- Hard time windows:
  For a new route, the optimal departure time is calculated (delivering the shipment as early as possible within the time window of the shipment). Optimality in this case means that the most slack for succeeding shipments is left. When updating a route, the entire route is recalculated to make sure no time windows are violated. If the shipment is the first in the route, departure time is recalculated.
  Complexity: in the worst case, all shipments are in the same route, and the arrival time needs to be checked for each shipment, so $O(|I|)$

- Site resources:
  Whenever a shipment is added to a route, the arrival and departure times are recalculated. To ensure the number of trucks at a destination at any given time does not exceed the limit of the site, all arrival and departure times in all routes in the solution need to be checked. If a conflict is found, the recently added shipment can only be delivered when the conflicting truck has departed (the truck waits at the destination until the conflicting truck leaves). The new arrival time thus becomes the departure time of the conflicting truck. This new arrival time may again conflict with other trucks, so the process is repeated until no conflicts are found. Complexity: Since the arrival times that are stored in the routes are ordered, finding a conflicting shipment in a route can be done in logarithmic time using binary search. Since inserting a shipment can cause the succeeding shipments to have a different arrival time, a search needs to be performed for every succeeding shipment. Let $s$ be the maximum number of shipments any destination has, $r$ the number of routes, then the complexity becomes $O(s \times r \log(|I|))$
• Shipment to shipment incompatibility:
The input contains a list of shipments that can not be transported in the same vehicle. Inserting a shipment in a route can only be done when no incompatible shipments are in the current route. Complexity: For every route \( r \) in the solution, we need to check if there is an incompatible shipment, so \( O(|I|) \)

• Reverse logistics and precedence constraints between shipments:
Each route consists of three subroutes, delivering ambient shipments, delivering frozen shipments, and picking up reverse shipments. When a new route is created, the process is identical for all three subroutes. When a route is updated, the new shipment is inserted in the corresponding subroute. Complexity: \( O(1) \)

• Shipment to vehicle incompatibility:
Not all vehicles can transport both frozen and ambient shipments, which limits the number of vehicles that can be selected for a new route. When a route is considered for inserting a shipment that can not be transported by the current vehicle, a different vehicle is selected if possible. If no vehicle is available with the required capabilities, the shipment can not be inserted in that route. Complexity: \( O(|P|) \)

• Vehicle to site incompatibility:
Some destinations have a limit on the size of the vehicle that is able to deliver. Inserting a shipment in a route can only be done when the total capacity of the route does not exceed this limit. Complexity: \( O(1) \)

• Time-dependent travel time:
In general we have \( t \) travel time values, which depends on the scheduling horizon, and the amount of intervals the scheduling horizon is divided in. In this thesis every entry in the distance matrix has \( t \) travel time values. When calculating viability of a route and its length, the travel time between two destinations is selected from this distance matrix based on the departure time. Complexity: \( O(1) \)

• Multiple warehouses:
For a new route, the warehouse closest to the destination is selected. When updating a route, the warehouse is not changed. Complexity: linear in the number of warehouses, so \( O(|W|) \)

• Variable unloading times:
Location delay depends on the fixed location delay (the minimum time waiting at all locations) and the size of the shipment that has to be unloaded at the destination. Location delay is calculated when calculating the route viability and length. Complexity: \( O(1) \)
Optimizations
The most time consuming step in the insertion heuristic is calculating the route length (in terms of time), because this is also where the Site Resources constraint is checked. To minimize computation time in the insertion step, the amount of times the length of a route is calculated must therefore be minimized. This is done by only checking positions in a route for which it is possible that the result is cheaper than the current cheapest option. The extra travel time is estimated as the difference between the total travel time in the two new arcs, and the travel in the original arc. If the extra travel time results in a delivery time of the next shipment that is later then the end of its time window (the due date) that means the shipment can not be inserted in this position. Likewise, an estimate can be made on the delivery time of the new shipment. If this delivery time is later than the end of its time window (its due date) than this also means that the shipment can not be inserted in this position, and we can even conclude that the shipment can not be inserted in any position later in the route. The extra distance can be estimated in the same manner as the travel time, and with both we can estimate the extra cost for inserting the shipment in this position. If this cost is higher than the current best known extra cost, we can conclude that this position will not yield a cheaper solution.

3.2 Local Search
To improve the solutions that are found by the insertion heuristic, a Local Search algorithm is developed, based on the algorithms described in [Braysy and Gendreau, 2005a]. This Local Search algorithm uses five different moves, that are described below.

- n-insertion
  n-insertion removes $n$ random shipments from the solution, and then reinserts them using the insertion algorithm described above. When $n$ is small, it is possible to find the optimal order to reinsert the shipments, by trying all possible orders and selecting the best one. When $n$ is too large, the shipments are inserted in the same order as they were removed. An example of n-insertion where $n$ is 1 is shown below.

![Example of n-insertion](image)

The stopping criteria for this move is a maximum number of iterations, i.e. a maximum number of times $n$ shipments are removed and reinserted. The choice for the
value of \( n \), as well as the number of iterations, is discussed in Chapter 5, Computational results.

- **2-opt**
The basic idea of the two-opt move is to remove two existing edges, and insert two new edges, as shown in the picture below.

![2-opt diagram](image)

As can be seen, the arcs \((i, i+1)\) and \((j, j+1)\) have been removed, and the arcs \((i, j)\) and \((i+1, j+1)\) have been added. Also note that the direction of the arcs between \( j \) and \( i+1 \) have been changed. This move is performed systematically for every 2 arcs in every route, restarting whenever an improvement is found, and stopping when no more improvements can be found.

- **Route-elimination**
In route elimination, the goal is to lower the amount of routes in the solution. This is done by removing one of the routes, and reinserting each shipment using the insertion heuristic. This move is performed for every route in the solution, and the solution is updated whenever an improvement is found.

- **Exchange**
The exchange move exchanges the position of two shipments in different routes, as illustrated below.

![Exchange diagram](image)

This move is performed for every pair of shipments in the solution, provided they have the same type. the solution is updated whenever an improvement is found.
- In-exchange
  In-exchange is identical to the exchange move, with the difference that in-exchange only exchanges shipments in the same route.
3.3 Scatter Search

As stated above the Scatter Search heuristic combines two solutions to form one new solution aiming to contain high quality properties from both original solutions. The heuristic works as follows: let \( A \) be a solution with \( n \) routes and \( B \) a solution with \( m \) routes, and \( A_i \) the \( i^{th} \) route in \( A \) and \( B_k \) the \( k^{th} \) route in \( B \). The first step is to create a table of size \( n \times m \), and store in element \((i, k)\) the amount of common shipments in routes \( A_i \) and \( B_k \). Creating a new solution starts with the routes with the most common shipments, and adding a partial route to the new solution comprised of these common shipments. The order in which the common shipments are visited is taken from the route with the smallest distance. The warehouse closest to the first shipment in the new route is added to the start and end of the new route, thereby completing the route. After this, the row and column of the combined routes is deleted from the table, and this is repeated until there are no routes left that can be combined, either because the remaining routes have no common destinations, or all routes have been combined. Note that if \( n \neq m \) there will always be at least 1 route that can not be combined. The solution that has now been created only contains the shipments that were in overlapping parts of a route, which means that there are shipments remaining that need to be inserted to complete the new solution. The remaining shipments are added using the insertion heuristic that is also used to create the initial solutions.

The full algorithm combines the above three ideas in the following way: first, a set of initial solutions is generated by the insertion heuristic, which are then optimized using LS. For the Scatter Search algorithm to be effective, this set of solutions needs to be diverse. This diversity is achieved by varying the weights that determine the best position in the route to insert a new shipment, and the order in which the shipments are inserted in the FSH, \( (w_1 - w_7, \) see Section 3.1). The effect of varying these weights is that the LS algorithm focuses on a different aspect of the cost function. Increasing \( w_1 \) for example will result in a solution with a smaller distance. This set of optimized solutions is then combined in all possible ways using the Scatter Search principle, and each newly created solution is optimized using LS. The combination and optimization step is then iterated for a maximum amount of iterations. The best value for this maximum amount of iterations, as well as the number of optimized solutions generated before the first combination step is discussed in Chapter 5, Computational results.

Note that for large problem instances, it becomes increasingly time consuming to keep a large population of solutions. The question may therefore arise whether it is useful to use Scatter Search at all, rather than only using LS. This question is also answered in Chapter 5.

3.4 Restricted Dynamic Programming

In the paper "Restricted dynamic programming: a flexible framework for solving realistic VRPs", [Gromicho et al., 2008] a framework is proposed that can handle most constraints
that exist in a real-life environment. This makes it an excellent solution method to compare to the Scatter Search algorithm. The algorithm combines the Restricted Dynamic programming heuristic for the TSP proposed by [Malandraki and Dial, 1996], which is applied through the Giant Tour Representation (GTR) introduced by [Funke and Grunert, 2005]. The next section describes an overview of the algorithm, its weaknesses and expectations on the performance on the TD-VRPTW, followed by the results of a basic implementation.

3.4.1 Dynamic Programming for the TSP

The TSP considers the problem of visiting a set of \( n \) nodes \( N = \{0, \ldots, n-1\} \) exactly once, starting and ending in node 0 and minimizing the total distance travelled. The algorithm proposed by [Malandraki and Dial, 1996] builds a solution by expanding a set of states \((S, j), j \in S, S \subset N \setminus \{0\}\), which represent a partial tour starting at node 0 and visiting all nodes in \( S \), ending in \( j \). The cost of a state \( C(S, j) \) is considered to be the minimal cost to complete the partial tour. Expansion of the states is done in a series of stages, where the first stage consists of \( N-1 \) states, which are \((\{i\}, i), \forall i \in N \setminus \{0\}\). Expanding a state consists of adding an unrouted node to its partial tour. A next stage consists of the states that are created by expanding all states from the previous stage to all unrouted nodes. A dominance check is performed every stage, to ensure that no two states are expanded to the next stage that have visited the same nodes, and end in the same node. To restrict the solution space, a variable \( H \) is introduced, which limits the number of states that are expanded into the next stage. Furthermore, a variable \( E \) determines the amount of closest (in terms of distance) unvisited nodes that each state will be expanded to, creating a kind of beam search [R. Bisiani, 1987].

To use this algorithm for a VRP, The GTR of Funke and Grunert [2005] is used, which adds a start and end node for every available vehicle to the set of nodes \( N \). Consider \( m \) as number of vehicles available, and \( r^v, v = \{1, \ldots, m\} \) as the route of vehicle \( v \). The routes are ordered, which means that the end node of route \( r^v \) will always be connected to the start node of route \( r^{v+1} \).

With every state expansion, a feasibility check is performed to determine if any of the constraints has been violated, if this is so, the state expansion is discarded. This feasibility enables the flexibility of the solution, because a change in the constraints will only lead to a change in this feasibility check.

3.4.2 Will it work for the TD-VRPTW?

An interesting question is of course whether this approach can work for the TD-VRPTW. When looking at that one can identify a number of potential issues. The algorithm is for example unable to deal with an unknown number of available vehicles, since the appropriate number of start and end nodes needs to be created. To incorporate this, one would have to either determine an upper bound on the needed number of vehicles (the number of nodes,
for instance), or let the number of vehicles (and thus the number of start and end nodes) grow as the states are expanded.

Another issue arises if multiple depots are considered, the starting location (depot) for each vehicle needs to be known beforehand, since this determines the distances between the start and end nodes. Assuming the number of vehicles is known, this would increase the state space with a factor of \((\text{number of vehicles}) \times (\text{number of depots})\) if all possibilities are considered.

The feasibility check performed at each state expansion is comparable to the feasibility check performed when a neighboring solution is checked in an LS algorithm. This feasibility check can encompass exactly as many constraints as in the DP algorithm. The premise that any Local Search algorithm needs to be rewritten when a new constraint is added is, for some constraints, also true for the DP algorithm.

When the problem size increases, the number of states increases exponentially. The variable \(\text{H}\) does limit the state space, although every added customer adds an additional stage to be computed. When customers have multiple orders (which is common in real-life situations) this would mean having as many stages as orders, which will have a significant impact on the computation time. As the total number of possible states increases, the quality of the solution will decrease, as only \(\text{H}\) states are expanded to the next stage.

In [Gromicho et al., 2008], performance is tested on the Solomon benchmarks. These benchmarks have up to 100 customers, with 1 order per customer. The computation times for these examples already reach 15 minutes in some cases. Considering a real-life problem with hundreds of customers and multiple orders per customer might be infeasible with this algorithm.

### 3.4.3 Implementation

To test the effectiveness of the Dynamic Programming algorithm, a basic implementation was done that could handle the Solomon instances as well as the large TD-VRPTW instances.

The crucial component (in terms of processing time) of the algorithm is the dominance check. The dominance check ensures that no states are expanded to a next stage that are dominated by another state. For two states \(a\) and \(b\) that deliver the same shipments, and are expanded from the previous stage with the same shipment, state \(a\) is considered to dominate state \(b\) if its cost is not higher than that of state \(b\), its remaining capacity is not less than that of \(b\), and its departure time is not later than that of state \(b\). This dominance check ensures that no two identical states are expanded to the next stage, and makes sure the optimal solution is never removed in this step. For large values of \(\text{H} (>1000)\) it becomes infeasible to perform this dominance check for every state in a stage. A hash function is therefore needed to ensure that the dominance check is only performed on states that have a high chance of being dominated. The hash function used in this implementation was very simple, and proved to be effective for the Solomon instances. Each state has a bitset that denotes the shipments that already have been delivered, and the hash value was the sum of the position of the positive bits in the bitset. For a problem with 100 customers,
this results in a value in the range (0..5050). When this implementation was tested on the Solomon benchmarks, the results were comparable to the results reported in [Gromicho et al., 2008], the results on problem R207 for instance, was a distance of 1098 and 3 routes. This result was found in 13 minutes, which is slightly below the average computation time reported in [Gromicho et al., 2008].

The strong point of this algorithm is the flexibility and the wide range of constraints it can handle, which makes it an excellent candidate for an alternative solution method for the TD-VRPTW. The computation time needed to solve the large-scale TD-VRPTW is expected to be very long, since the states need to be expanded through 800 stages before the solutions are constructed. The solution space is also very large, which means that the amount of states that are expanded to the next stage is only a small fraction of the total solution space, which is expected to have a negative impact on the quality of the solution. To give an illustration of the impact of the solution size: if all states are expanded to the next stage ($H$ is set to infinity), the third stage will have 510,081,600 states, both feasible and infeasible. This means that if the state space is restricted (let’s assume for the illustration that $H$ is set to 100,000, and that 90% of the states is infeasible due to the constraints), more than 99.8% of the possible solutions are ignored, based on the quality of the partial solution where only 3 shipments have been delivered. In reality, setting $H$ to 100,000 is infeasible for the base instance of the TD-VRPTW, because the computation time for an $H$ value of 10,000 already passes 5 hours.
Chapter 4

A Real-life TD-VRPTW Problem

As explained, the goal of this project is to model and solve a real-life vehicle routing problem as found in the retail industry. To that end, several instances have been created that aim to simulate a problem as it might exist in real-life. We based these instance on the experience we had through the extensive contacts we had with a company in the retail industry. The largest instance that has been created contains 271 destinations, of which 2 destinations act as a warehouse, and as such, have no shipments delivered to them. The remaining 269 destinations have up to 3 shipments, resulting in a total of 800 shipments.

4.1 Base instance

This section describes the base instance of the real-life problem. It describes the input data, and justifies some of the choices made regarding the input data and constraints. Note that for some of the constraints and input variables no data was known, and were estimated to values we think are realistic.

Destinations

In real-life, the destinations of any logistical problem are typically not randomly distributed, nor very clustered. To ensure a realistic distribution of destinations, we decided to use the locations of actual supermarkets in the Netherlands. In this case, the locations of the Jumbo supermarkets have been used. The distance between the supermarkets have been estimated based on their latitude and longitude, where travel by road is estimated to be 20% further than the distance "as the crow flies". We assume all supermarkets are served by 2 warehouses, located in Vechel and Beilen, the locations of two of Jumbo’s distribution centers. For the vehicle to site incompatibility constraint, we estimated that 10% of the destinations are located in places the largest vehicles can not reach. Finally, a destination is assumed to be able to handle only one vehicle at a time.
Travel time

Vehicles are estimated to have an average speed of 60 km/h, which combined with the distance gives the base travel time. To incorporate time dependency, vehicles are assumed to have a delay at certain points in the day, based on information by the Dutch ’verkeersinformatiedienst’. Fig. 4.1 shows the factor by which the base travel times are multiplied.

![Figure 4.1: The peaks (moments in time where the traffic is most dense) are at respectively 08:40 AM and 06:40 PM.](image)

Vehicles

The largest truck that is used on the Dutch roads is a truck with a capacity of 52 pallets. In the base inempty bottles beer crates transport stance where all constraints are considered, 4 different vehicles are considered, with capacities of respectively 26, 38, 42, and 52 pallets. It is assumed that all vehicles can carry backhaul shipments, since these are picked up after all other goods are delivered. The two largest trucks can carry both frozen and ambient shipments, using the already mentioned movable wall. The smallest truck can only carry frozen shipments, and the second smallest only ambient shipments.

Shipments

The average amount of ambient items (pallets) is estimated to be 15 for a typical supermarket. Since there can be a large difference in demand between supermarkets, each shipment consists of a random amount between 5 and 25 items. The amount of frozen items is estimated to be between 2 and 15 items. Backhaul shipments consist of empty pallets and packaging that needs to be returned to the warehouse, and is therefore related to the amount of frozen and ambient shipments. Empty pallets are typically stacked in stacks of 8, and the amount of packaging is estimated to be between 0 and 6 pallets. The total amount of backhaul items is therefore estimated to be

\[ \text{round}(\text{#ambient} + \text{#frozen})/8) + \text{package} \]
Time Windows

Each shipment has a time window, which describes the earliest and latest time at which a vehicle may arrive to deliver the shipment. 4 different time windows are considered:

- early delivery: 6 am - 9 am
- afternoon delivery: 11 am - 2 pm
- evening delivery: 5 pm - 8 pm
- no time window: 6 am - 9 pm

In the base instance, 50% of shipments have a time window that is randomly selected between the early, afternoon, and evening delivery time windows.

Shipment to shipment incompatibility

The Shipment Incompatibility Set is a set of shipments that cannot be transported in the same vehicle. In the base instance such a constraint exists between 10% of all shipments, which means that for an instance with 800 shipments, 40 pairs are added to the list, between 80 shipments.

4.2 Instances

To determine the impact of the different constraints, several instances have been created, that only have a subset of the constraints. In most instances only one constraint is left out, so the impact on the resulting schedule can be seen.

- No time windows:
  Time windows of all shipments are set to be 6 am - 9 pm.

- Homogeneous fleet:
  In this instance only 1 type of vehicle is used, which is the largest truck used in the base instance. As a result the vehicle to site incompatibility and shipment to vehicle incompatibility are omitted.

- Time independent travel times:
  Travel times in this instance are independent of departure time.

- No shipment to shipment incompatibilities:
  The Shipment Incompatibility list is empty

- No vehicle to site incompatibility:
  The $MQ_d$ variable for each destination $d$ is set to the largest vehicle available.
• No shipment to vehicle incompatibility:
  All available vehicles can carry both ambient and frozen shipments.

• Unconstrained:
  Site resources and vehicle to site incompatibility are ignored, and all time windows are set to 5 am - 10 pm.
Chapter 5

Computational results

5.1 Parameters

This section describes the computational study that has been done to determine the best values for the different parameters in the Scatter Search algorithm. All tests shown in this section were performed on the base instance of the TD-VRPTW, where the site resources constraint was ignored. The reason this instance is used instead of the base instance with site resources, is that a large amount of tests needed to be done, and including the site resources constraint results in a much higher computation time, while the difference in the cost of the resulting solution is small (see Section 5.2).

5.1.1 n-insertion

The n-insertion move has several parameters that can be changed that have an effect on both the computation time and the effectiveness of the move. The most obvious parameter is of course the value for $n$, which controls the amount of shipments removed from, and then reinserted into, the solution. It is interesting to know whether it is more effective to perform 1-insertion 1000 times, or 10-insertion 100 times. The graph below is an overview of the improvements made after 1000 iterations of n-insertion, where $n$ is varied between 1 and 10. This test was performed on the initial solution as provided by the insertion heuristic. The results have been normalized based on computation times.
As can be seen, the lower values for $n$ prove to be more effective. This becomes more apparent when the test is done on a solution that has been improved. In this case, the other 4 moves have been performed on the initial solution used in the previous test. This is an important test, because the Local Search algorithm is run mostly on solutions that have been improved, since after each iteration of the Scatter Search algorithm the newly created solution is improved using Local Search.

The reason that the amount of improvement decreases with larger values for $n$ is that the insertion heuristic that provides the initial solution, and also reinserts the shipments, is very greedy. As can be seen, a value of 1 for $n$ is the best choice. The downside of 1-insertion is that it gets stuck in a local optimum more easily than n-insertion with higher
values for $n$, as can be seen in the next graph, where 1-insertion is iterated 4000 times, and compared to a mixed version where respectively 1, 2, 3, and 4-insertion is iterated 400 times each, resulting in an equal amount of reinsertions.

As can be seen, the improvement per reinsertion of both 1-insertion and the mixed-insertion is comparable, and while 1-insertion shows almost no improvements after 3000 iterations, mixed-insertion continues to improve its solution. Based on these results, mixed-insertion as described above is used in the Local Search algorithm.

**Randomness**

The n-insertion move selects the shipments that are to be removed from the solution in a random way. To analyse the effect of this randomness 5 runs of the LS algorithm were compared, each with a different random seed, and are shown in the graph below.
Each line in this graph represents the cost of a solution, and how it is improved over time by the LS algorithm. From this graph we can conclude that the random seed indeed has an influence on the quality of the solution, and that the difference in the resulting cost (less than 1%) is small enough to conclude that we can safely ignore the effects of the random seed on the final solution of the Scatter Search algorithm.

5.1.2 Scatter Search

As discussed in Section 3.3, we need to decide how many initial solutions are generated and optimized, as well as determine the best number of iterations the combination step is performed. Each iteration of the Scatter Search algorithm combines a fixed number of solutions in every possible way. To keep the computation time within acceptable limits, this number of solutions must be low, the best value we found for this was 4, which means that in every iteration of Scatter Search 6 new solutions are created.

The diversity in the initial solutions is created by varying the weights that are discussed in Section 3.1. It is interesting to know if using these weights has a beneficial effect, i.e. what is the quality of the solution when we do not sort the input, and determine the best insertion position based purely on cost? Tests have shown that for some instances sorting the input has a negative effect on the initial solutions, i.e. in those instances no value for weights $w_4 - w_7$ was found that resulted in a cheaper initial solution. A possible explanation for this is that in the input data the shipments are ordered based on destinations, and as such the insertion heuristic is more likely to insert shipments to the same destination in the same route. The best values for $w_4 - w_7$ we found were:

\begin{align*}
  w_4 &= 2 \\
  w_5 &= 0 \\
  w_6 &= 0 \\
  w_7 &= 1
\end{align*}

This means that if we do sort the input, the best sort order we found is based on the distance to the closest warehouse, and the number of items in the shipment. Since this is identical for all instances, these weights were used for all of them.

To generate 4 diverse initial solutions, two variations of weights $w_1 - w_3$ were used, and for each of them two initial solutions were made, one in which the input was sorted, and one in which it was not.

Since the instances of the TD-VRPTW are large and complex, the number of Scatter Search iterations that can be done in reasonable computation time will be low, as every iteration 6 new solutions need to be created and optimized. The following graph shows the results of two iterations of Scatter Search.
What we can see here are the results of each of the optimization steps that are performed by LS. The blue lines represent the costs of the individual solutions over time, where the first 4 lines represent the 4 initial solutions and the last blue line represents the best solution found in the two combination steps, which is again optimized using LS. The red lines separate the Scatter Search iterations. As we can see, the first iteration of Scatter Search results in the best solution that is obtained, which means that in this case, the second iteration of Scatter Search was not beneficial. Further testing proved that after 5 iterations still no better solution was obtained. These results are of course different for each instance of the TD-VRPTW, however the extra computation time needed to calculate more iterations is not justified by the benefits gained by doing so. We therefore have decided to use 2 Scatter Search iteration for all instances of the TD-VRPTW.

To answer the question that was posed in Section 3.3, whether it was useful to use Scatter Search as opposed to only using LS, we did a test in which we checked if LS indeed reaches a local optimum. The results of this test are shown in the graph below.
As can be seen, the Local Search algorithm shows little to no improvements after the second iteration, and this remains true even if the number of iterations is increased beyond the 8 shown here. The best result found by this Local Search algorithm is 284129, whereas the Scatter Search algorithm results in a cost of 275848, a large difference showing the merit of using Scatter Search over using a pure Local Search implementation.
5.2 Solomon benchmarks

Another method of determining the quality of the algorithm is to solve well known problems, and compare its solutions to the best known solutions for that problem. The most famous problems are the Solomon benchmarks, which have been solved by many different algorithms. All instances we consider of the Solomon benchmarks consist of 100 customers that each have time windows and a certain demand. The problems are separated in 6 distinct groups, c1, c2, r1, r2, rc1, and rc2. When a ‘c’ occurs in the problem name, that means that the customers are clustered, whereas an ‘r’ in the problem name means a random distribution of customers. Logically, ‘rc’ in the problem name means a mix between clustered and randomly distributed customers. Problem classes c1, r1, and rc1 have small trucks with a capacity of 200, and problem classes c2, r2, and rc2 have larger trucks with a capacity of 1000. The current best known solutions are either proven to be, or estimated to be close to, the optimal solutions. The table below shows the results of the Scatter Search algorithm, the best known solutions, and a comparison between the two. Note that to limit the computation time, the calculations were spread over several machines. Normalizing the computation times is therefore difficult. All results shown below were found within 20 minutes, and the fastest machine used in the calculations had an Intel Core2 Duo CPU @2.50GHz, with 3Gb RAM.
<table>
<thead>
<tr>
<th>Problem name</th>
<th>Distance</th>
<th>Vehicles</th>
<th>Distance</th>
<th>Vehicles</th>
<th>Distance (%)</th>
<th>Vehicles</th>
</tr>
</thead>
<tbody>
<tr>
<td>c101:</td>
<td>828.94</td>
<td>10</td>
<td>828.94</td>
<td>10</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>c102:</td>
<td>828.94</td>
<td>10</td>
<td>828.94</td>
<td>10</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>c103:</td>
<td>853.44</td>
<td>10</td>
<td>828.06</td>
<td>10</td>
<td>2.97</td>
<td>0</td>
</tr>
<tr>
<td>c104:</td>
<td>824.78</td>
<td>10</td>
<td>824.78</td>
<td>10</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>c105:</td>
<td>828.94</td>
<td>10</td>
<td>828.94</td>
<td>10</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>c106:</td>
<td>828.94</td>
<td>10</td>
<td>828.94</td>
<td>10</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>c107:</td>
<td>898.94</td>
<td>10</td>
<td>827.3</td>
<td>10</td>
<td>7.97</td>
<td>0</td>
</tr>
<tr>
<td>c108:</td>
<td>828.94</td>
<td>10</td>
<td>828.94</td>
<td>10</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>c109:</td>
<td>828.94</td>
<td>10</td>
<td>828.94</td>
<td>10</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>c201:</td>
<td>591.56</td>
<td>3</td>
<td>591.56</td>
<td>3</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>c202:</td>
<td>591.56</td>
<td>3</td>
<td>591.56</td>
<td>3</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>c203:</td>
<td>591.17</td>
<td>3</td>
<td>591.17</td>
<td>3</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>c204:</td>
<td>653.45</td>
<td>3</td>
<td>590.6</td>
<td>3</td>
<td>9.62</td>
<td>0</td>
</tr>
<tr>
<td>c205:</td>
<td>588.88</td>
<td>3</td>
<td>588.88</td>
<td>3</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>c206:</td>
<td>588.49</td>
<td>3</td>
<td>588.49</td>
<td>3</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>c207:</td>
<td>588.32</td>
<td>3</td>
<td>585.5</td>
<td>3</td>
<td>0.48</td>
<td>0</td>
</tr>
<tr>
<td>c208:</td>
<td>588.32</td>
<td>3</td>
<td>588.32</td>
<td>3</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>r101:</td>
<td>1705.58</td>
<td>19</td>
<td>1645.79</td>
<td>19</td>
<td>3.51</td>
<td>0</td>
</tr>
<tr>
<td>r102:</td>
<td>1500.1</td>
<td>17</td>
<td>1486.12</td>
<td>17</td>
<td>0.93</td>
<td>0</td>
</tr>
<tr>
<td>r103:</td>
<td>1250.89</td>
<td>14</td>
<td>1292.68</td>
<td>13</td>
<td>-3.34</td>
<td>1</td>
</tr>
<tr>
<td>r104:</td>
<td>1069.68</td>
<td>10</td>
<td>1007.24</td>
<td>9</td>
<td>5.84</td>
<td>1</td>
</tr>
<tr>
<td>r105:</td>
<td>1413.95</td>
<td>15</td>
<td>1377.11</td>
<td>14</td>
<td>2.61</td>
<td>1</td>
</tr>
<tr>
<td>r106:</td>
<td>1314.29</td>
<td>13</td>
<td>1251.98</td>
<td>12</td>
<td>4.74</td>
<td>1</td>
</tr>
<tr>
<td>r107:</td>
<td>1166.68</td>
<td>11</td>
<td>1104.66</td>
<td>10</td>
<td>5.32</td>
<td>1</td>
</tr>
<tr>
<td>r108:</td>
<td>1010.24</td>
<td>10</td>
<td>960.88</td>
<td>9</td>
<td>4.89</td>
<td>1</td>
</tr>
<tr>
<td>r109:</td>
<td>1250.27</td>
<td>12</td>
<td>1194.73</td>
<td>11</td>
<td>4.44</td>
<td>1</td>
</tr>
<tr>
<td>r110:</td>
<td>1132.44</td>
<td>12</td>
<td>1118.59</td>
<td>10</td>
<td>1.22</td>
<td>2</td>
</tr>
<tr>
<td>r111:</td>
<td>1111.73</td>
<td>11</td>
<td>1096.72</td>
<td>10</td>
<td>1.35</td>
<td>1</td>
</tr>
<tr>
<td>r112:</td>
<td>984.36</td>
<td>10</td>
<td>982.14</td>
<td>9</td>
<td>0.23</td>
<td>1</td>
</tr>
<tr>
<td>Problem name</td>
<td>Distance</td>
<td>Vehicles</td>
<td>Best known</td>
<td>Distance</td>
<td>Vehicles</td>
<td>Difference:</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>----------</td>
<td>------------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>r201:</td>
<td>1263.27</td>
<td>4</td>
<td>1252.37</td>
<td>4</td>
<td>0.86</td>
<td>0</td>
</tr>
<tr>
<td>r202:</td>
<td>1088.91</td>
<td>4</td>
<td>1191.7</td>
<td>3</td>
<td>-9.44</td>
<td>1</td>
</tr>
<tr>
<td>r203:</td>
<td>990.29</td>
<td>3</td>
<td>939.5</td>
<td>3</td>
<td>5.13</td>
<td>0</td>
</tr>
<tr>
<td>r204:</td>
<td>761.5</td>
<td>3</td>
<td>825.52</td>
<td>2</td>
<td>-8.41</td>
<td>1</td>
</tr>
<tr>
<td>r205:</td>
<td>1054.62</td>
<td>3</td>
<td>994.42</td>
<td>3</td>
<td>5.71</td>
<td>0</td>
</tr>
<tr>
<td>r206:</td>
<td>951.7</td>
<td>3</td>
<td>906.14</td>
<td>3</td>
<td>4.79</td>
<td>0</td>
</tr>
<tr>
<td>r207:</td>
<td>830.32</td>
<td>3</td>
<td>890.61</td>
<td>2</td>
<td>-7.26</td>
<td>1</td>
</tr>
<tr>
<td>r208:</td>
<td>744.37</td>
<td>2</td>
<td>726.75</td>
<td>2</td>
<td>2.37</td>
<td>0</td>
</tr>
<tr>
<td>r209:</td>
<td>966.84</td>
<td>3</td>
<td>909.16</td>
<td>3</td>
<td>5.97</td>
<td>0</td>
</tr>
<tr>
<td>r210:</td>
<td>989.03</td>
<td>3</td>
<td>939.34</td>
<td>3</td>
<td>5.02</td>
<td>0</td>
</tr>
<tr>
<td>r211:</td>
<td>827.95</td>
<td>3</td>
<td>885.71</td>
<td>2</td>
<td>-6.98</td>
<td>1</td>
</tr>
<tr>
<td>rc101:</td>
<td>1693.35</td>
<td>15</td>
<td>1696.94</td>
<td>14</td>
<td>-0.21</td>
<td>1</td>
</tr>
<tr>
<td>rc102:</td>
<td>1521.89</td>
<td>14</td>
<td>1554.75</td>
<td>12</td>
<td>-2.16</td>
<td>2</td>
</tr>
<tr>
<td>rc103:</td>
<td>1330.85</td>
<td>12</td>
<td>1261.67</td>
<td>11</td>
<td>5.20</td>
<td>1</td>
</tr>
<tr>
<td>rc104:</td>
<td>1154.9</td>
<td>10</td>
<td>1135.48</td>
<td>10</td>
<td>1.68</td>
<td>0</td>
</tr>
<tr>
<td>rc105:</td>
<td>1585.48</td>
<td>15</td>
<td>1629.44</td>
<td>13</td>
<td>-2.77</td>
<td>2</td>
</tr>
<tr>
<td>rc106:</td>
<td>1431.13</td>
<td>13</td>
<td>1424.73</td>
<td>11</td>
<td>0.45</td>
<td>2</td>
</tr>
<tr>
<td>rc107:</td>
<td>1270.93</td>
<td>11</td>
<td>1230.48</td>
<td>11</td>
<td>3.18</td>
<td>0</td>
</tr>
<tr>
<td>rc108:</td>
<td>1156.62</td>
<td>11</td>
<td>1139.82</td>
<td>10</td>
<td>1.45</td>
<td>1</td>
</tr>
<tr>
<td>rc201:</td>
<td>1441.32</td>
<td>4</td>
<td>1406.91</td>
<td>4</td>
<td>2.39</td>
<td>0</td>
</tr>
<tr>
<td>rc202:</td>
<td>1271.13</td>
<td>4</td>
<td>1365.65</td>
<td>3</td>
<td>-7.44</td>
<td>1</td>
</tr>
<tr>
<td>rc203:</td>
<td>1387.34</td>
<td>3</td>
<td>1049.62</td>
<td>3</td>
<td>24.34</td>
<td>0</td>
</tr>
<tr>
<td>rc204:</td>
<td>830.63</td>
<td>3</td>
<td>798.41</td>
<td>3</td>
<td>3.88</td>
<td>0</td>
</tr>
<tr>
<td>rc205:</td>
<td>1352.74</td>
<td>4</td>
<td>1297.19</td>
<td>4</td>
<td>4.11</td>
<td>0</td>
</tr>
<tr>
<td>rc206:</td>
<td>1204.86</td>
<td>3</td>
<td>1146.32</td>
<td>3</td>
<td>4.86</td>
<td>0</td>
</tr>
<tr>
<td>rc207:</td>
<td>1031.28</td>
<td>4</td>
<td>1061.14</td>
<td>3</td>
<td>-2.90</td>
<td>1</td>
</tr>
<tr>
<td>rc208:</td>
<td>877.34</td>
<td>3</td>
<td>828.14</td>
<td>3</td>
<td>5.61</td>
<td>0</td>
</tr>
<tr>
<td>totals:</td>
<td>58253.35</td>
<td>431.00</td>
<td>57176.41</td>
<td>405.00</td>
<td>1.85</td>
<td>26.00</td>
</tr>
</tbody>
</table>

As can be seen, the Scatter Search algorithm has difficulty finding the optimal number of vehicles, as evidenced by the results on instances r110, rc102, rc105, and rc106. Apart from this, the algorithm performs very well, scoring on average within 2% of the best known distance, and within 7% of the best known amount of vehicles. Although this is no conclusive proof that the results of the Scatter Search algorithm on the TD-VRPTW are also close to the optimal solution, it does prove that the algorithm is capable of producing good quality solutions.
5.3 TD-VRPTW

The two tables below display the computational results of the Scatter Search algorithm on the real-life instances that are described in Section 4.2. Note that the optimal solution to these instances is not known, because of the size and complexity of the instances.

Determining the quality of the results of the Scatter Search algorithm on the TD-VRPTW is difficult, since there is no optimal solution to which we can compare our results. Computing a lower bound on the number of vehicles is interesting because it provides an insight to the quality of the solutions in terms of the number of vehicles. A simple lower bound on the number of vehicles would be the total amount of items delivered in the ambient and frozen shipments, divided by the capacity of the largest vehicle. The backhaul shipments are omitted in that reasoning because they are picked up after all ambient and frozen shipments have been delivered. The total amount of items in the ambient and frozen shipments is 4949 in all full-sized instances, and the largest vehicle can carry 52 of those items. The theoretical minimal amount of vehicles (and therefore the minimal amount of routes) is thus 97. Because of the amount of constraints in the instances, it is probable that the optimal solution will have more vehicles than this lower bound. The unconstrained instance is created to see whether the algorithm can indeed reach this bound.

<table>
<thead>
<tr>
<th>instance</th>
<th>cost</th>
<th>travel time</th>
<th>distance</th>
<th>vehicles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard:</td>
<td>277735</td>
<td>50869.2</td>
<td>24682.3</td>
<td>125</td>
</tr>
<tr>
<td>No site resources:</td>
<td>275692</td>
<td>51786.8</td>
<td>23772.8</td>
<td>122</td>
</tr>
<tr>
<td>No time windows:</td>
<td>244940</td>
<td>42256</td>
<td>20809.5</td>
<td>113</td>
</tr>
<tr>
<td>No vehicle to site inc:</td>
<td>262401</td>
<td>48081.2</td>
<td>22246.3</td>
<td>107</td>
</tr>
<tr>
<td>No shipment to shipment inc:</td>
<td>282156</td>
<td>52106.9</td>
<td>24480.8</td>
<td>129</td>
</tr>
<tr>
<td>No shipment to vehicle inc:</td>
<td>262576</td>
<td>47518.5</td>
<td>22796.5</td>
<td>122</td>
</tr>
<tr>
<td>Homogeneous fleet:</td>
<td>270791</td>
<td>48042.4</td>
<td>21902.2</td>
<td>109</td>
</tr>
<tr>
<td>Unconstrained:</td>
<td>233566</td>
<td>39522.8</td>
<td>19190.1</td>
<td>99</td>
</tr>
</tbody>
</table>

What we can see from these results is that the solution to the unconstrained instance is indeed close to the calculated lower bound on the number of vehicles. The most influential constraint is clearly the time window constraint, which is also expected because introducing time windows increases the average amount of time at the destinations (a vehicle now may also have to wait before it can deliver a shipment) as well as restrict the number of feasible routes. The instance where a homogeneous fleet is used, also results in a relatively low amount of vehicles, which is explained by the size of the vehicles in that instance, since now all used vehicles can carry a load of 52 items. An unexpected result is the high cost for the instance without shipment to shipment incompatibilities, which is the highest cost found for any of the instances. An explanation might be that the Shipment Incompatibility Set prevents that some solutions are found that lead to a local optimum. Further research should be done to determine the exact cause.
5.4 MIP Model

The results of the MIP model is shown below, the shown result was optimized based on distance and number of vehicles. In the large instances, total travel time was also included in the cost function, including time in the cost function for the MIP model however resulted in a sharp increase in computation time, while the aim of the MIP was only to show that the Scatter Search algorithm can indeed find optimal solutions.

<table>
<thead>
<tr>
<th>solution method</th>
<th>cost</th>
<th>travel time</th>
<th>distance</th>
<th>vehicles</th>
<th>computation time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIP</td>
<td>7128.725</td>
<td>2295.749</td>
<td>295.79</td>
<td>3</td>
<td>8927</td>
</tr>
<tr>
<td>Scatter Search</td>
<td>7128.725</td>
<td>2295.749</td>
<td>295.79</td>
<td>3</td>
<td>17</td>
</tr>
</tbody>
</table>

As can be seen, both solution methods find the same, optimal, solution. It is interesting to see the large difference between the computation times in the solution methods, showing the difficulty in proving optimality in a problem with this complexity.
5.5 Restricted Dynamic Programming

The Restricted Dynamic Programming algorithm was implemented because it can handle real-life constraints, which makes it an excellent framework to compare to the Scatter Search algorithm.

The results on the base instance of the TD-VRPTW is shown below:

<table>
<thead>
<tr>
<th>instance</th>
<th>cost</th>
<th>travel time</th>
<th>distance</th>
<th>vehicles</th>
<th>computation time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>326648</td>
<td>52326.9</td>
<td>36331.2</td>
<td>226</td>
<td>19827</td>
</tr>
</tbody>
</table>

Results of the Scatter Search algorithm:

<table>
<thead>
<tr>
<th>instance</th>
<th>cost</th>
<th>travel time</th>
<th>distance</th>
<th>vehicles</th>
<th>computation time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>275848</td>
<td>50137.1</td>
<td>23082.9</td>
<td>119</td>
<td>11423</td>
</tr>
</tbody>
</table>

As can be seen, the quality of this solution is very poor compared to that of the Scatter Search algorithm. The conclusion is that although the Dynamic Programming algorithm is very flexible, and can handle many different constraints, the quality of the solution diminishes with large problem sizes.
Chapter 6

Conclusion

In this project we have modeled and solved the TD-VRPTW which is a real-life load building and routing problem as found in the retail industry. We discussed the complexity of such a problem, and developed several solution methods for it. The most notable of these is the Scatter Search algorithm, for which we developed an insertion heuristic to generate initial solutions, and a Local Search algorithm to improve these solutions. The Scatter Search algorithm can combine several solutions, to escape local optima. Another solution method we developed is Restricted Dynamic Programming, which is a construction algorithm that constructs a large number of solutions simultaneously. Lastly we developed an exact solution method that solves the problem using MIP technology. The computational results show that the Scatter Search algorithm can solve all instances within reasonable computation time. The quality of these solutions is difficult to estimate, since this is the first time such a problem has been solved. We did show that an unconstrained instance of the problem can be solved close to the theoretical lower bound on the number of vehicles. We also show that a very small instance with all the constraints can be solved to optimality by the Scatter Search algorithm. This optimal solution was first solved by IBM ILOG CPLEX, using the MIP model. The Scatter Search algorithm was also used to solve the famous Solomon benchmarks, and it is shown that it can compete with the state-of-the-art solution methods that have solved the Solomon benchmarks to their current best known solutions. In another attempt to estimate the quality of the solutions to the real-life instances, we implemented the Dynamic Programming algorithm described in [Gromicho et al., 2008]. As we expected, the algorithm loses its efficiency when the problem size increases. The main contributions of this thesis are the extensions we made to the insertion heuristic as introduced by [Solomon, 1987], the Local Search algorithm as described in [Braysy and Gendreau, 2005a], and the Scatter Search algorithm as described in [Belfiore and Yoshizaki, 2009], so that they can handle the constraints of the TD-VRPTW.
Future work

Possible interesting future work is to include even more real-life constraints. Two examples of these are split deliveries, where a shipment can be split over several vehicles, and drivers regulations, where the working hours of the truck drivers are constrained so that they comply with European legislation. Another interesting future direction will be to introduce handling uncertainty like stochastic travel times or stochastic demands. Some work in this area has already been done, Ak and Erera [2007] for example describes a variant of the VRP in which the demands of the customers are stochastic, instead of fixed. Their proposed solution combines a tabu search algorithm with a vehicle pairing strategy, which allows for destinations to be added to the route of the paired vehicle if the current vehicle has no capacity remaining.

Also if we stick to the TD-VRPTW as defined in this thesis, there is a lot of interesting future work that can be done:

One thing can be to develop different First Solution Heuristics. These can be used to create better, more diverse, initial solutions, and can also be used in the Local Search and Scatter Search algorithms. Another direction to improve this work is to do computational studies to further optimize the parameters that are used in the Scatter Search. The choice of the weights described in Section 3.1, for example, might still have room for improvement. In this computational study one could also investigate why a higher number of iterations of the Scatter Search does not result in better solutions. Finally, Section 5.2 shows that the algorithm has difficulty finding the optimal number of vehicles, improving the insertion algorithm in this respect might be an interesting research topic.

The MIP model also still has room for improvement, since the computation time as reported is very high compared to the number of shipments in the solved instance. More specifically, one could improve the formulation of the site resources constraint. The current formulation:

\[
\sum_{v \in V} \sum_{i \in c} (((S_{iv} \leq t \leq (S_{iw} + FLD + q_i \cdot LD)) \cdot X_{ijv}) \leq SR_c) \text{ for all } t \in T, c \in C, \text{ and } j \in N
\]

sums up all vehicles \(v\) at customer \(c\) at time \(t\), while we only need the sum of vehicles \(v\) at customer \(c\) at an arrival time of a vehicle:

\[
\sum_{v \in V} \sum_{i \in c} (((S_{iw} \leq S_{iw} \leq (S_{iw} + FLD + q_i \cdot LD)) \cdot X_{ijv}) \leq SR_c) \text{ for all } w \in V \setminus \{v\}, c \in C, \text{ and } j \in N
\]

Furthermore, it might be interesting to see what the effect of the individual constraints is on both the computation time and resulting solution.
Bibliography


Appendix A

Notations

CVRP = Capacitated Vehicle Routing Problem
GTR = Giant Tour Representation
LS = Local Search
MIP = Mixed Integer Programming
SS = Scatter Search
TD-VRPTW = Time Dependent Vehicle Routing Problem with Time Windows
TSP = Traveling Salesman Problem
VRP = Vehicle Routing Problem
VRPTW = Vehicle Routing Problem with Time Windows
Appendix B

MIP model

/*----------------------------------------------------------
 * OPL 12.3 Model
 * Author: matthijs
 * Creation Date: Jun 13, 2012 at 2:55:07 PM
 ----------------------------------------------------------*/

{string} Vehicles = ...;
{string} Customers = ...;
{string} Shipments = ...;
tuple CustType {
    int destId;
    int MQ;
    int WH;
    int SR;
}

tuple Pair{
    string ship1;
    string ship2;
};
{Pair} SIL = ...;
int FLD = 10;
tuple ShipType {
    int destId;
    int items;
    string type;
    int eat;
    int lat;
}
tuple VehicleType {
  int Capacity;
  intFixedPrice;
  intPricePerTime;
  intPricePerDistance;
  {string} Types;
}
tuple Arc {
  int start;
  int end;
  float distance;
  float TT[0..101];
}
CustType CustInfo[Customers] = ...;
VehicleType VehicleInfo[Vehicles] = ...;
ShipType ShipInfo[Shipments] = ...;
{Arc} Arcs = ...;
{string} A = {s | s in Shipments: ShipInfo[s].type == "a"};
{string} F = {s | s in Shipments: ShipInfo[s].type == "f"};
{string} B = {s | s in Shipments: ShipInfo[s].type == "b"};
{string} W = {s | s in Shipments: ShipInfo[s].type == "w"};
{string} C = A union F union B;
{string} N = A union F union B union W;
range T = 0..10;
float ArcDist[Customers][Customers];
float ArcTime[Customers][Customers][0..101];
float Cost[Shipments][Shipments][Vehicles];
execute INITIALIZE {
  for (var a in Arcs) {
    for (var i in Customers) {
      for (var j in Customers) {
        if (a.start == CustInfo[i].destId) {
          if (a.end == CustInfo[j].destId) {
            ArcDist[i][j] = a.distance;
            for (var t = 0; t < 102; t++)
              ArcTime[i][j][t] = a.TT[t];
          }
        }
      }
    }
  }
  for (var ii in Shipments) {
for (var jj in Shipments)
for (var k in Customers)
for (var l in Customers)
  if (ShipInfo[ii].destId == CustInfo[k].destId)
    if (ShipInfo[jj].destId == CustInfo[l].destId)
      for (var v in Vehicles)
        Cost[ii][jj][v] =
        ArcDist[k][l]*VehicleInfo[v].PricePerDistance;
    }
  }
}
}
}

dvar boolean X[Shipments][Shipments][Vehicles];
dvar float S[N][Vehicles];
dexpr float objective =
  sum(i, j in Shipments, v in Vehicles)
    Cost[i][j][v]*X[i][j][v] +
  sum(v in Vehicles) VehicleInfo[v].FixedPrice *
  sum(i in W, j in C)X[i][j][v];
minimize objective;

constraints {
  forall (i in C)
    deliver_all_shipments : (sum(v in Vehicles, j in N )
      X[i][j][v] == 1);
  forall (i in N, v in Vehicles)
    leave_all_nodes : sum(j in N)
      X[i][j][v] == sum(h in N) X[h][i][v];
  forall (v in Vehicles)
    start_at_wh : sum(i in W, j in N)
      X[i][j][v] == 1;
forall (v in Vehicles, w in W)
cyclic_routes: sum(i in C)
X[i][w][v] = sum(j in W)

forall (v in Vehicles)
end_at_wh: sum(i in N, j in W)
X[i][j][v] = 1;

c1 in Customers : CustInfo[c1].destId == ShipInfo[i].destId,
c2 in Customers : CustInfo[c2].destId == ShipInfo[j].destId
Start_of_service: (S[i][v] + (sum(t in 0..101)
(ArcTime[c1][c2][t] * (t*10 <= S[i][v] <= ((t+1)*10)))) +
FLD + ShipInfo[i].items) - 10000*(1-X[i][j][v]) <= S[j][v];

forall (i in N: ShipInfo[i].items !=0, v in Vehicles)
no_loop: X[i][i][v] = 0;

forall (i in A, j in F, v in Vehicles)
precedence_ambient_frozen: S[i][v] <= S[j][v];

forall (i in F, j in B, v in Vehicles)
precedence_frozen_backhaul: S[i][v] <= S[j][v];

forall (i in N, v in Vehicles)
time_windows: ShipInfo[i].eat <= S[i][v] <= ShipInfo[i].lat;

capacity: (sum(i in A union F) ShipInfo[i].items * sum(j in N)X[i][j][v]) <= VehicleInfo[v].Capacity;

forall (v in Vehicles)
backcapacity: (sum(i in B) ShipInfo[i].items * sum(j in N)X[i][j][v]) <= VehicleInfo[v].Capacity;

forall (v in Vehicles, c in Customers)
forall (i, j in N: ShipInfo[j].destId == CustInfo[c].destId)
vehicle_site_inc: VehicleInfo[v].Capacity * X[i][j][v] <= CustInfo[c].MQ;

forall (c in Customers, t in T, j in N)
site_resources: (sum(i in C: ShipInfo[i].destId == CustInfo[c].destId,
\text{v in Vehicles)
\quad (((S[i][v] \leq t) + (t \leq (S[i][v]+\text{FLD+ShipInfo}[i].\text{items}))) \leq 2) - (1-X[i][j][v])) \leq \text{CustInfo}[c].\text{SR};

\text{forall (v in Vehicles, k in SIL)
\quad shipment_{shipment_{inc}}: \sum(i \in N)X[i][k.\text{ship1}][v] + \sum(i \in N)X[i][k.\text{ship2}][v] \leq 1;}

\text{forall (v in Vehicles)
\quad ship_{veh_{inc}}: \sum(i, j \in N: \text{ShipInfo}[j].\text{type in VehicleInfo}[v].\text{Types}) X[i][j][v] = \sum(i, j \in N) X[i][j][v];
}