ERASMUS UNIVERSITEIT ROTTERDAM

A reactive GRASP with path relinking for the pick-up and delivery problem with cross-docks

Master's thesis by

Gerhard van Ginkel

Student number: 355735
Faculty: Erasmus School of Economics
Department: Econometrics
August 28, 2017

Supervisors:
Dr. W. Van den Heuvel
Prof. Dr. J.A. dos Santos Gromicho
Dr. Ir. A.L. Kok

Abstract

The pick-up and delivery problem with cross-docks is the problem of serving a number of transportation requests using a limited heterogeneous vehicle fleet. Each request has to be picked-up from and delivered to a specific location. Cross-docks can be used to temporarily store a product, such that another vehicle can deliver the product to its final destination. Our task is to construct routes against the lowest cost such that all requests are handled and time, capacity and capability constraints are met. Very liberal routing policies are allowed, which makes that this thesis is as far as we know the first in literature that studies this problem. This thesis mathematically defines this problem and presents a heuristic for it that combines a Greedy Randomized Adaptive Search Procedure with a ruin and recreate method for local search and Path Relinking, Furthermore, this heuristic is embedded in an Ant Colony Optimization framework. The heuristic is tested on 21 instances with up to 25 requests. The results indicate that sizable cost decrease can be obtained by applying more liberal routing policies and a dynamic assignment of requests to cross-docks.

Contents

1 Introduction

Cross-docking is a logistic strategy in which orders from several suppliers are consolidated in cross-dock facilities and redistributed to the customers. Cross-docks differ from traditional warehouses in the sense that received products are typically loaded within 12 hours after arrival, without storing them in between in dedicated storage areas.

Therefore, cross-docking can offer many benefits in a distribution network. Besides reducing transportation cost due to the consolidation of orders, cross-docking also results in less inventory holding cost and an increased cycle time [Agustina et al., 2010]. Furthermore, cross-docking can be compulsory when physical or legal constraints require different modes of transport for the pick-up and for the delivery.

In order to maximize the advantages of cross-docking, scholars have developed a wide variety of mathematical models to improve efficiency in different application areas. These areas range from location decision problems to dock door assignment problems and multimodal freight transportation problems. For further details on these topics, we refer the interested reader to the surveys of Kara and Taner [2011], Agustina et al. [2010] and SteadieSeifi et al. [2014].

1.1 Research problem

This thesis focuses on a routing problem. The goal is to determine optimal routes where orders are picked-up and delivered at a specific location within given time windows. These orders can be transported via one of the cross-docks, but direct delivery is also a possibility. Vehicles can perform so-called milk runs, in which they pick-up or deliver multiple orders in one trip. Furthermore, not all vehicles from the heterogeneous vehicle fleet are allowed to visit all locations; a restriction that often occurs in practice in urban areas. The research takes into account various realistic cost types and the time consumption associated with cross-docking activities.

A typical application area of such a milk run system with cross-docking is the automobile manufacturing industry, in which a lot of low-volume orders of parts from different suppliers should be delivered to car manufacturers [Sadjadi et al., 2009]. Moreover, the model in our thesis can also be simplified and applied to context with more high-volume orders, such as the retailing industry where all orders are sourced from one or a few depots.

1.2 Research motivation

This thesis conducts a research in collaboration with ORTEC. ORTEC is one of the world leaders in optimization software for business processes. They develop custom made solutions for a wide variety of business problems, such as loading, routing, workforce planning and warehousing. Offices of ORTEC can be found in all over the world, which is indicative for their diverse international client base. Besides the consumer role of ORTEC, they also work closely together

with the academic world. Employees publish scientific articles regularly and are often asked to present these at conferences. Also, since ORTEC operates in the practical world, they can fuel the academic world with benchmark problems and new research areas.

1.2.1 Product challenges at ORTEC

ORTEC's core strategy is to develop one product that can serve the whole market. Due to the large variety in requirements among different customers, this product is highly customizable. However, ORTEC feels that in some areas the product can be improved in order to be able to adapt better to the needs of the client. Prior to the research conducted in this thesis, an orientation project was set up to get an overview of the most pressing issues for which the product of ORTEC could be made more flexible to the client's needs. To get an idea which problems were most valuable to tackle, they were mainly evaluated on three points. First of all, the number of clients that face this problem should be substantial. Secondly, it should be expected that the gain of an improved solution method is substantial compared to existing solution methods. Finally, the problem should be challenging from a scientific perspective.

Several problems met this criteria. Some examples are departure smoothing (include restrictions on truck depot dock capacity), platooning (save fuel by driving trucks close behind each other), intermodal transport (consider multiple modes of transport between an origin-destination pair) and multi-manning (include the possibility of two drivers on one truck).

1.2.2 Dynamic cross-docking

The product challenge under consideration in this thesis concerns dynamic cross-docking. The counterpart of dynamic cross-docking is static cross-docking. In static cross-docking, orders are already assigned to a specific cross-dock before optimization. The problem then boils down to solving a vehicle routing problem with dependencies between orders; a problem for which ORTEC already has developed state-of-the-art algorithms. Dynamic cross-docking differs from static cross-docking in the sense that orders are not assigned beforehand to a cross-dock. In the optimization process, it is both decided whether an order should be cross-docked or not, and to which cross-dock it should be sent.

Currently, dynamic cross-docking in ORTEC is mostly done manually or on the basis of some straightforward heuristics. This means that a mathematical approach could lead to substantial efficiency benefits for all clients that require cross-docking.

1.3 Bridging theory and practice

A case of a customer of ORTEC is used as a benchmark to compare the proposed theoretical solution methods with the current solution methods of ORTEC. Since in general, the cases to which these methods can be applied are

widely different from each other regarding the objective function, restrictions and supply chain structure, it is impossible to create a one-size-fits-all solution. At the same time, the proposed models should be applicable to multiple cases.

Therefore, on the one hand, the models will be general, such that the cases are a simplified version of the problem. On the other hand, the models are limited and incorporate only the most important restrictions and supply chain features. Small adoptions and additions can be used to customize the model according to the client's needs.

The features that are included in our model are those that constitute the main reasons for ORTEC's clients to perform cross-docking. A heterogeneous fleet should be included since consolidation is a main driver behind cross-docking. Furthermore, capabilities are accounted for since cross-docks function as facilities where goods are loaded to vehicles that are allowed to perform the delivery. Finally, multiple pick-up points, cross-docks and delivery points are included is the model in order to mimic the supply chain structure of the cases. Time windows are also included as they heavily influence the structure of the optimal solution.

In this way, although making a vast amount of simplifications, a reasonably reliable comparison between existing client-specific solution methods and the proposed general solution method can be made. The results however should be translated carefully to practice, and are merely indicative for the performance achieved when applying the methods to the complete cases.

Besides providing useful applications of this research to ORTEC, we mainly aim to make a relevant contribution to the existing scientific literature.

1.4 Thesis overview

In section 2, we give an overview of the scientific literature that relates to our problem. Section 3 defines our problem mathematically by means of a mixed integer programming model. We describe our research on exact solution methods for this problem in section 4. Besides the exact approach, section 5 describes an extensive heuristic solution methodology. The results of this heuristic can be found in section 6, together with the data description. Finally, in section 7 we draw conclusions and propose avenues for further research.

2 Literature review

The review of Guastaroba et al. [2015] classifies the problem under consideration in this thesis as a special case of the Pick-up and Delivery Problem with Cross-Docks (PDPCD). At a cross-dock, products arrive on inbound vehicles. The products are unloaded from the vehicles, consolidated with other orders, and sent away on outbound vehicles to their destinations. The main distinguishing feature of cross-docks relative to other storage facilities, is that products often stay less than 12 hours at a cross-dock. Typical application areas are therefore fashion and perishable or refrigerated products.

The goal of the PDPCD is to find routes that handle all pick-up and delivery requests at the lowest cost. The PDPCD extends the classical PDP in providing the opportunity to ship loads via a cross-dock. The PDPCD is closely related to the two-Echelon Capacitated Vehicle Routing Problem (2E-CVRP). The main difference is that the latter assumes that all orders originate from a single depot.

The first studies on PDPCD's were mainly focused on many-to-many problems, where all orders consist of the same homogeneous product and thus can be sourced from multiple suppliers. Due to their limited applicability, the focus has shifted to one-to-one problems, where each order has a specific pick-up and delivery location.

2.1 Pure PDPCD

In the most basic form of the one-to-one PDPCD, each request has a load and a specific pick-up and delivery location. These requests must be handled by a limited and homogeneous fleet of capacitated vehicles. Vehicles start at the cross-dock, then perform pick-ups, followed by unloading and loading operations at the cross-dock and finally handle the deliveries and return to the cross-dock. Note that the PDPCD in this basic form falls apart in two independent CVRP's, one before the cross-dock operations and one after the cross-dock operations. In the literature, authors study several additional problem specifications that link the two stages together.

Wen et al. [2009] study a basic PDPCD with additional time-window constraints on both pick-up and delivery. Also, they consider a fixed and variable loading time at the cross-dock, which intertwine the pick-up and delivery stages of the problem even more. They give a MILP-formulation for the problem and propose a Tabu Search (TS) heuristic to solve it. To avoid getting stuck in local minima, they allow infeasible solutions, but only against a penalty which increases every iteration. In order to investigate completely different regions of the solution space, they embed the TS within an Adaptive Memory Procedure, which repeats the TS from different starting points. Since insertion moves in TS are computationally expensive in the PDPCD due to the interdependency between routes, Wen et al. [2009] alleviate the computational burden by estimating the cost of such a move. It turned out that computation times decreased up to a factor 30, with only a small deterioration of the best found solution value. Wen et al. [2009] test their method on a dataset that is based on a Danish logistics company. Lower bounds for the optimal solution of these instances were found by simplifying the problem to two independent VRPTW's and solving these to optimality. Their methods produce solutions for 200 supplier-customer pairs with a less than 5% optimality gap.

The same problem as in Wen et al. [2009] is studied by Tarantilis [2013]. They also use TS and combine it with a multi-restart algorithm that start TS from newly generated initial solutions, based on the information extracted from a reference set of solutions. Applying their adaptive memory components and solution recombination schemes leads to solutions which are a substantial improvement on the solutions reported by Wen et al. [2009]. Furthermore, they

extend their work by incorporating the possibility that some vehicles do not have to return to the depot, as often happens in practice when vehicles are rented from third party logistics providers.

More recently, Morais et al. [2014] build further on the work of Wen et al. [2009] and Tarantilis [2013] by applying greedy heuristics combined with three different sophisticated iterated local search heuristics. By circumventing TS and narrowing the search to the space of feasible solutions, they make their algorithm more efficient. They also have a restart procedure which uses a combination of existing elite routes in order to explore different parts of the solution space. Their results show a substantial increase in performance compared to existing solution methods.

2.2 Hybrid PDPCD

An important assumption in these papers is that no vehicle performs a direct route: each vehicle has to visit a cross-dock between pick-ups and deliveries. Liu et al. [2003] were among the first to study the hybrid PDPCD, which drops this assumption. They study the PDPCD in its most basic form with only an unlimited vehicle fleet and the possibility of direct shipment as an additional specification. Therefore, they consider two types of routes: direct routes that perform pick-ups and deliveries without visiting a cross-dock in between, and routes that stop at the cross-dock in order to load or unload. They solve this problem with a local search method. The key characteristic of this method is that they divide all requests in two subsets. The first subset is handled by direct routes, while the requests in the second subset are cross-docked. Given these two subsets, they solve the resulting routing problem with the Clark and Wright [Clarke and Wright, 1964] heuristic for the CVRP. The local search part of their heuristic consists of moving requests from one subset to another, based on an estimate of the savings that move will obtain. Based on a set of randomly generated instances, they conclude that allowing these two types of routes instead of only one type saves on average 10% in traveled distance.

Santos et al. [2011] study a PDPCD with handling cost. They develop an exact binary linear programming formulation that is able to solve instances of 50 requests that are extracted from the dataset of Wen et al. [2009]. In Santos et al. [2013], they extend their work by also allowing direct routes. They show that on average a cost saving of 3.3% can be obtained as a result of the introduction of direct deliveries.

Qu and Bard [2012] drop one of the most important assumptions in the basic PDPCD. They do not longer require that pick-ups are handled before deliveries, and cross-docks can be visited multiple times by the same vehicle. Furthermore, also direct routes are allowed. In contrast with all previous research on the PDPCD, the heuristic that they develop therefore cannot longer handle the PDPCD as two related CVRP's. This makes it harder to find good solutions, as they can no longer exploit the structural properties of such a problem. To find solutions, they develop a Greedy Randomized Adaptive Search Procedure (GRASP). In the construction phase, multiple solutions are generated in par-

allel by greedily inserting requests in existing routes. A hashing procedure is used to decrease the number of insertion positions that need to be checked. The resulting solutions are improved by using an adaptive large neighborhood search with several removal and insertion heuristics. Since their problem was unique in the literature, they applied their methods on self-generated instances with 25 requests. The heuristic was not always able to find feasible solutions, but the found feasible solutions were within 1% optimality for 88% of the instances. A more detailed description of these instances and the procedure to reach optimality can be found in section 6.1.1.

2.3 Path relinking

In Resendel and Ribeiro [2005], the authors give an overview of the application of path relinking in GRASP algorithms. For the fundamental ideas behind GRASP and path relinking, we refer the reader to the papers of respectively Feo and Resende [1995] and Glover et al. [2000]. The combination of path relinking and GRASP has been successful in a wide range of applications. Pure GRASP algorithms improved both in solution value as in the time to target value when it was combined with a path relinking procedure. Most closely related to our problem is the research of Nguyen et al. [2012] that studies the two Echelon Location Routing Problem (2E-LRP). They combine a GRASP algorithm with a learning process and path relinking. Path relinking is not performed on two final solutions, but on a big tour representation of these solutions, which covers the rudimentary structure of the solution. In the path relinking procedure, a path is created between two different big tours by gradually transforming one big tour in the other. The intermediate found big tours are the basis of the newly created intermediate solutions. In this way, they circumvent the problem of infeasible solutions along the path between two solutions. They test their methods on instances with 200 requests and 10 satellites, and report that the path relinking component significantly improves results.

2.4 Contribution to literature

The problem studied in this thesis is most closely related to that of Qu and Bard [2012]. We extend their problem definition with some additional realistic features, such as a heterogeneous fleet with capabilities, fixed and variable loading time and cost and multiple cross-docks. The solution methodology used in this thesis is a combination of GRASP and path relinking, which is embedded in an ant colony optimization framework to make the algorithm reactive. The GRASP part of the algorithm makes extensive use of a ruin and recreate procedure (see for example the paper of Schrimpf et al. [2000]). A more detailed overview of the solution methodology can be found in section 5.1. To the best of our knowledge, the combination of GRASP and path relinking is not applied before on the PDPCD. Furthermore, the ant colony optimization framework also has not been applied to the PDPCD, or more widely, to any pick-up and

delivery problem with transshipment opportunities. Therefore, our research provides insight in both a new problem as well as a new solution methodology.

3 Problem definition

In this section, the appropriate notation is introduced and the research problem is formally defined as a Rich Pick-up and Delivery problem with Transfers (RPDPT) using a network flow formulation. The goal of the RPDPT is to minimize the total cost, while delivering all requests and satisfying all time, capacity and capability restrictions.

3.1 Notation

The RPDPT is described on the directed graph $G = (N, A)$. The node set $N = P \cup T \cup B$, where P is the node set that contains all customer requests and T is the node set associated with all the transshipment locations $\tau \in \mathcal{T}$. Each transshipment location $\tau \in \mathcal{T}$ is associated with multiple nodes in T, which will be explained in the next paragraph. The set B contains the start and end nodes of all vehicles. The arc set A consists of all feasible arcs between nodes in N. Each node i is associated with a geographical position $\mathcal{G}(i)$.

Each customer request $c \in C$ is associated with a load $L_c \in \mathrm{R}_{>0}$, which should be picked up at node $i_c \in P$ and delivered at node $j_c \in P$. This can be either done by one vehicle or by two different vehicles. When one vehicle provides the service, it picks the request up at i_c and subsequently delivers is at j_c. When two different vehicles deliver the service, the first vehicle brings the request from i_c to a transshipment delivery node $j_{c\tau} \in T$ and the second vehicle brings the request from transshipment pickup node $i_{c\tau}$ to j_c. Each node $i \in P$ is associated with a time window $[a_i, b_i]$ in which service should start. The transshipments points are not associated with a time window.

Each vehicle $k \in K$ is stationed at node $b_k \in B$ and should return to node $b'_k \in B$. The sequence of nodes visited by a vehicle is called a route $r \in R$. Vehicle k has a capacity of Q_k which may never be exceeded along its route. Each vehicle k has its own capabilities, meaning that it can only visit nodes $N_k \subseteq N$. Every vehicle can visit all transshipment nodes, so $N_k \cap T = T, \forall k \in K$. The set $K(i)$ contains all vehicles that can visit node i. Furthermore, vehicle k has a fixed (un)loading time σ_k^f, variable (un)loading time σ_k^v per load unit, fixed usage cost of δ_k^f, variable usage cost of δ_k^v per kilometer, fixed (un)loading cost of cl_k^f and a variable (un)loading cost of cl_k^v per load unit.

The travel time between two nodes $i, j \in N$ is T_{ij} and the corresponding distance equals D_{ij}. Note that travel times and distances are equal for all vehicles $k \in K$.

3.2 Additional specifications

In addition to the problem definition given in section 3.1, we have a few additional problem specifications. First of all, the size of a request does not exceed the capacity of the largest vehicle which can reach the corresponding pick-up and delivery nodes. This is a weak assumption, as the model allows for splitting larger orders manually beforehand in separate smaller orders, which can be picked up by different vehicles.

Secondly, the fixed (un)loading time and cost is only incurred once when multiple requests are sequentially (un)loaded at the same location. This specification is motivated by practice. The fixed cost and time are mainly constituted by actions like parking a vehicle and making it ready for loading, which only need to be executed once per location.

The fixed vehicle usage cost δ_k^f is only incurred when vehicle k visits other nodes than b_k and b'_k.

When a vehicle arrives at a location in order to (un)load, it first incurs the fixed loading time and then it incurs the variable loading time. When picking up requests at transshipment locations, the request must be available right after the vehicle finishes its fixed loading tasks.

Time windows on the pick-up and delivery nodes of requests are identical for requests with the same geographical location. Moreover, a time window of $[a_i, b_i]$ on node i only requires that the vehicle that handles the corresponding request should arrive within this time window on the geographical location associated with the request. For example, if a vehicle starts loading a request before time b_i and subsequently loads another request after time b_i at the same geographical location, this would still be a feasible solution.

3.3 Network flow formulation

3.3.1 Auxiliary graph

Now the RPDPT is formulated as a network flow problem. Our formulation is loosely based on the MILP formulation of Wen et al. [2009]. As a consequence of the specific constraints in our rich problem, the construction of the graph on which this formulation is based, requires special attention. Since multiple requests can be required to be picked up from or delivered to the same geographical location, the nodes of the graph could represent those geographical locations. However, since vehicles are allowed to return to the same location, this will lead to difficulties regarding the modeling of time. Another possibility is that each node would represent a single request. The shortcoming of this modeling approach lies in the specific way the time windows on requests are defined. The modeling of such a restriction with this node structure is cumbersome, as the time window on a node now depends on the route of a vehicle.

Since conventional modeling approaches cannot fully capture the requirements of the RPDPT, we introduce the concept of *geo-nodes*. A vehicle must visit a geo-node each time it moves to a node in P that has a different geographical location than the previously visited node. We set time windows on the geo-

nodes instead of the nodes in P, which allows us to handle the time constraints of the RPDPT properly. More precisely, the directed graph $G' = (N', A')$ on which the network flow formulation is defined, is constructed as follows.

Step 1. *Create nodes.* The following node sets are introduced:

- P: contains for all $c \in C$ the pick-up node i_c and the delivery node j_c.
- T: contains for all $c \in C$ and $\tau \in \mathcal{T}$ a delivery node $j_{c\tau}$ and a pick-up node $i_{c\tau}$.
- B: contains for all $k \in K$ a start node b_k and an end node b'_k.

Step 2. *Create arcs.* We include all arcs in A that would make the graph $(P \cup T \cup B, A)$ a complete directed graph. In order to decrease the solution time of the MILP model, arcs that can never be part of a feasible solution could be removed beforehand. We removed all arcs from delivery nodes to pick-up nodes, all arcs from end nodes to start nodes, all arcs between start and between end nodes and all arcs from one node to nodes which can never be reached in time by either vehicle.

Step 3. *Create geo-nodes.* In this step the set of geo-nodes Γ is constructed. A geo-node is inserted between all nodes, where the head of the corresponding arc is incident with a node $p \in P$, and the tail of the arc in incident with a node that has a different geographical location than p. For example, the arc (b_k, j_c) would be replaced by the two arcs (b_k, γ_{j_c}) and (γ_{j_c}, j_c), where γ_{j_c} is the newly created geo-node. These new arcs are included in A to create the arc set A'. We now have constructed the graph $G' = (N', A')$, where $N' = P \cup T \cup B \cup G$.

Step 4. *Assign arc cost c^k_{ij}, arc time t_{ij}, arc capacity q_{ij} and time windows.* Initially, we set all cost, time and capacity to zero and set all time windows infinitely wide. Then we iteratively increment the values associated with the arcs as described below. The symbol Δ indicates that the associated value should be incremented.

- $\forall (i,j) \in A', k \in K$: $\Delta c^k_{ij} = D_{ij} \cdot \delta^v_k$ and $\Delta t^k_{ij} = T_{ij}$
- $\forall (i,j) \in A' | i = b_k \wedge j \neq b'_k, k \in K$: $\Delta c^k_{ij} = \delta^f_k$
- $\forall (i,j) \in A' | i \in \Gamma \vee (j \in T \wedge \mathcal{G}(i) \neq \mathcal{G}(j)), k \in K$: $\Delta c^k_{ij} = cl^f_k$ and $\Delta t^k_{ij} = \sigma^f_k$
- $\forall c \in C, \tau \in \mathcal{T}, i \in \{i_c, i_{c\tau}\}, j \in N' | (i,j) \in A', k \in K$: $\Delta c^k_{ij} = cl^v_k \cdot L_c$, $\Delta t^k_{ij} = \sigma^v_k \cdot L_c$ and $q_{ij} = L_c$
- $\forall c \in C, \tau \in \mathcal{T}, i \in \{j_c, j_{c\tau}\}, j \in N' | (i,j) \in A', k \in K$: $\Delta c^k_{ij} = cl^v_k \cdot L_c$, $\Delta t^k_{ij} = \sigma^v_k \cdot L_c$ and $q_{ij} = -L_c$
- $\forall i \in \Gamma, j \in N' | (i,j) \in A'$: $[a_i, b_i] = [a_j, b_j]$

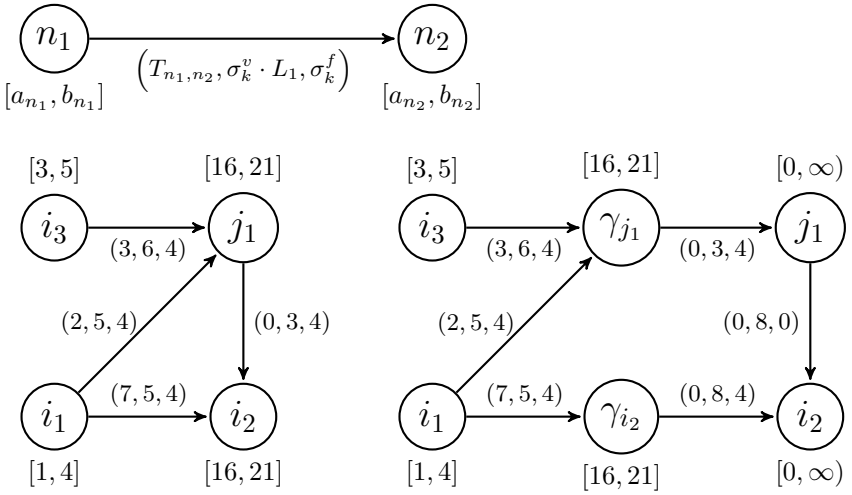

Figure 1: Example: modification of a graph after introduction of geo-nodes

In figure 1, we provide a small example of this process. At the left, we see a few nodes and arcs of a graph before the introduction of geo-nodes. In this graph, the route $i_1 \to j_1 \to i_2$ is not feasible, as we cannot arrive in time at node i_2. In a RPDPT instance however, this route could be feasible. Node j_1 and i_2 have the same geographical location, so loading and unloading operations at these nodes may exceed the end of the time window, as long as we arrive at node j_1 in time. Therefore, geo-nodes are introduced in the graph at the right of figure 1. We can see that the time cost for loading at node i_1 and the time cost for traveling from node i_1 to geo-node γ_{j_1} are incurred before arriving at γ_{j_1}. This node γ_{j_1} has the same time window as node j_1 in the original graph, such that the vehicle is enforced to arrive there in time. However, as can been seen from the values associated with arc $\gamma_{j_1} \to j_1$, the unloading time cost for node j_1 is only incurred when arriving at j_1. Since this node has no time window, (un)loading operations at j_1 may exceed the end of the time window of node j_1 in the original graph, as it should be able to. Furthermore, looking at arc $j_1 \to i_2$, we see that no travel time cost or fixed loading cost are incurred since these nodes represent the same geographical location. Therefore, node i_2 also has no time window since these loading operations are allowed to exceed the time window of node i_2 in the original graph.

3.3.2 MIP formulation

The MIP formulation of the RPDPT is based on the graph G'. We define the following variables:

$$x_{ij}^k = \begin{cases} 1 \text{ if vehicle } k \text{ travels from node } i \text{ to node } j \\ 0 \text{ else;} \end{cases}$$

s_i = the time at which the fixed loading tasks at node i are finished

q_i = the capacity of the vehicle at node i before it performs (un)loading tasks

In addition, M is a sufficiently large constant. We can now formulate the RPDPT as follows:

$$\text{minimize} \quad \sum_{(i,j)\in A'} \sum_{k\in K} c_{ij}^k x_{ij}^k \tag{1}$$

$$\text{subject to} \quad \sum_{j\in N'} \sum_{k\in K(i)} x_{ij}^k = 1, \forall i \in P \tag{2}$$

$$\sum_{j\in N'} x_{ij}^k - \sum_{j\in N'} x_{ji}^k = 0, \forall i \in N' \setminus B, k \in K \tag{3}$$

$$\sum_{j\in N'} x_{b_k j}^k = 1, \forall k \in K \tag{4}$$

$$\sum_{i\in N'} x_{i b_k'}^k = 1, \forall k \in K \tag{5}$$

$$\sum_{m\in K\setminus\{k\}} \sum_{j\in N'} x_{b_k j}^m = 0, \forall k \in K \tag{6}$$

$$\sum_{j\in N'} x_{j_c j}^k - \sum_{j\in N'} x_{i_c j}^k = \sum_{\tau\in\mathcal{T}} \sum_{j\in N'} x_{i_{c\tau} j}^k - \sum_{\tau\in\mathcal{T}} \sum_{j\in N'} x_{j_{c\tau} j}^k, \forall c \in C, k \in K \tag{7}$$

$$\sum_{j\in N'} x_{j_c j}^k + \sum_{j\in N'} x_{i_c j}^k \geq \sum_{\tau\in\mathcal{T}} \sum_{j\in N'} x_{i_{c\tau} j}^k + \sum_{\tau\in\mathcal{T}} \sum_{j\in N'} x_{j_{c\tau} j}^k, \forall c \in C, k \in K \tag{8}$$

$$\sum_{j\in N'} \sum_{k\in K} x_{j_{c\tau} j}^k = \sum_{j\in N'} \sum_{k\in K} x_{i_{c\tau} j}^k, \forall c \in C, \tau \in \mathcal{T} \tag{9}$$

$$s_j \geq s_i + t_{ij}^k - M\left(1 - x_{ij}^k\right), \forall i, j \in N', k \in K \tag{10}$$

$$q_j \geq q_i + q_{ij} - M\left(1 - \sum_{k\in K} x_{ij}^k\right), \forall i, j \in N' \tag{11}$$

$$q_i \leq Q_k + M\left(1 - \sum_{j\in N'} x_{ij}^k\right), \forall i \in N', k \in K \tag{12}$$

$$s_{i_c} \leq s_{j_{c\tau}} \leq s_{i_{c\tau}} \leq s_{j_c}, \forall c \in C, \tau \in \mathcal{T} \tag{13}$$

$$s_{i_{c\tau}} \geq s_{j_{c\tau}} + \sigma_k^v \cdot L_c, \forall c \in C, \tau \in \mathcal{T} \tag{14}$$

$$a_i \leq s_i \leq b_i, \forall i \in \Gamma \tag{15}$$

$$x_{ij}^k \in \{0,1\}, \forall (i,j) \in A', k \in K \tag{16}$$

$$s_i, q_i \geq 0, \forall i \in N' \tag{17}$$

The objective function in equation (1) minimizes the total cost. Note that all relevant costs are included in the parameters c_{ij}^k. Constraints (2) state that each

request should be picked up and delivered by a capable vehicle. All capability constraints are implicitly taken care of by these constraints. Constraints (3) constitute the flow balance constraints: when a vehicle enters a node, it should also leave that node. The exceptions on this rule are taken care of in constraints (4) and (5), that state that a vehicle should start and end at their designated nodes. Constraints (6) state that vehicles may not start from other nodes than their own. Note that the subtour elimination constraints are not required in this formulation, as they are already implicitely defined by the time constraints. At least one node of a cycle cannot satisfy constraints (10).

Together, constraints (7) and (8) define all feasible request-handling combinations that a vehicle can perform. Table 1 lists the five possible variable configurations that satisfy these constraints. Note that a vehicle is not allowed to load a request at a transshipment point in order to unload it at another transshipment point. A vehicle however is allowed to temporarily store a request at a transshipment point. Constraints (9) require that requests are unloaded and loaded at the same transshipment location.

Table 1: Possible configurations of request-handling variables

k picks up c $\sum_{j \in N'} x^k_{i_c j}$	k delivers c $\sum_{j \in N'} x^k_{j_c j}$	k unloads c $\sum_{\tau \in \mathcal{T}} \sum_{j \in N'} x^k_{j_{c\tau} j}$	k loads c $\sum_{\tau \in \mathcal{T}} \sum_{j \in N'} x^k_{i_{c\tau} j}$
0	0	0	0
0	1	0	1
1	0	1	0
1	1	0	0
1	1	1	1

The time and capacity variables are updated in restrictions (10) and (11). Constraints (12) ensure that vehicles do not exceed their capacity at all times. The correct chronological order of picking up, unloading, loading and delivering is enforced by restrictions (13) and (14). Constraints (13) are tightened by constraints (14), as a vehicle must finish its unloading tasks before another vehicle can start loading. The time-windows on the geo-nodes are taken care of in restrictions (15). Finally, all variables are defined as either binary or non-negative continuous in restrictions (16) and (17).

Since the traveling salesman problem is a special case of the RPDPT, the RPDPT is NP-hard.

4 Exact solution methods

In this section, we describe the research avenues we took in order to arrive at an exact solution of the RPDPT. Exact solutions of mid-size to large RPDPT's would allow us to assess the quality of heuristic solution methods. We tried solving different MILP-formulations directly in Gurobi, as well as a more advanced column generation procedure.

4.1 MIP models

A direct implementation in Gurobi of the network flow model described in section 3.3 could only solve very small problem instances in a reasonable time. With only one vehicle and one transshipment point, solution times already exploded with just ten requests. These long solution times are among others caused by the bad LP-relaxation of the network flow formulation. We tried to overcome this problem by altering the formulation, inspired by Baldacci et al. [2004]. In this two-commodity flow formulation, each arc is replaced by two new arcs, one representing the capacity left on a vehicle, and one representing the free capacity on a vehicle. The new set of variables introduced represents the amount of flow on each arc. Despite the stronger LP-relaxation of this formulation, it was not able to solve larger instances of the RPDPT.

4.2 Column Generation

In our problem, we can make a distinction between intra-route constraints and inter-route constraints. Intra-route constraints affect only single routes and can be evaluated locally. Examples are restrictions on vehicle capacities, time windows and customer sequencing. Inter-route constraints affect multiple routes. Examples are the requirements that each request should be picked up and delivered and that a request can only be loaded at a transshipment point if it is unloaded there before.

An intuitive exact solution approach would be to relax these inter-route constraints in the network flow formulation using Lagrangian Relaxation or a Dantzig-Wolfe decomposition. The resulting subproblems deal with the intra-route constraints and can be modeled as Elementary Shortest Path Problems with Resource Constraints (ESPPRC) and linear node cost for which reasonably well exact or heuristic solution methods exist. However, we expect a bad performance of these methods for two main reasons. The first reason is that some big-M constraints in the network flow formulation will enter the objective function, leading to bad lower bounds. The second reason is the potential symmetry problems which may arise for problem instances with a lot of vehicles of the same type. The solutions generated by the subproblems corresponding to these vehicles will be the same. This will lead to slow convergence and makes it also hard to construct tight upper bounds.

To circumvent these problems, we use the set covering formulation instead of the network flow formulation. Generally, the LP-relaxation of a set covering formulation is quite strong. In this formulation, all inter-route constraints are modeled explicitly, while the intra-route constraints are implicitly taken care of in the variables.

Since the set covering formulation has exponentially many variables, we use a column generation framework to solve the LP-relaxation. Initially, only a small subset of all variables are included in the formulation, and iteratively, promising new variables, which represent routes, are included. These variables are found by solving pricing problems that find variables with negative reduced

cost. The pricing problems boil down to an ESPPRC with linear node costs, which is solved by using a dynamic programming labeling algorithm.

The speed of the column generation approach depends heavily on the ability to discard dominated labels in the labeling algorithm. One of the dominance criteria for labels at a certain node, is that for each vehicle the set of requests which should still be delivered is exactly the same. Due to the negative dual variables on delivery locations and the presence of transshipment locations, the triangle inequality does not hold in the auxiliary graph. Therefore, we are not able to say whether the requirement to perform certain requests is desirable or not. This leads to a very weak dominance criterion, which is why we can only potentially discard labels that handle precisely the same requests in a different order.

Consequently, the NP-hard pricing problem can only be solved to optimality by a process that is close to enumeration. Since the pricing problem should be solved to optimality at least once to provide valid lower bounds on the set covering formulation, we expect that the column generation approach will not be able to solve mid-size RPDPT instances in a reasonable amount of time.

Our findings are in line with those of Qu and Bard [2012], as they could only solve a similar problem with one vehicle, one transshipment point and up to six requests to optimality.

5 Heuristic solution method

In section 4, we saw that exact approaches to the RPDPT do not seem very promising. Therefore, in this section, we develop a heuristic solution method that is able to find good solutions to RPDPT-instances of 25 requests in a reasonable amount of time. Section 5.1 states the general idea of the heuristic method. The remaining sections work out each part of the heuristic in more detail.

5.1 General overview

The heuristic used to solve the RPDPT is a variation on the Greedy Randomized Adaptive Search Procedure (GRASP). The main idea of a GRASP algorithm is to generate multiple start solutions and to subsequently improve each of them by applying local search. We complement the GRASP by path relinking, a procedure that creates a new solution based on two existing solutions. Furthermore, our GRASP is a reactive algorithm, which means that parameters are updated each iteration to bias the search towards promising solutions. We refer to our algorithm as a Reactive GRASP with Path Relinking (RGPR). Each step of our algorithm is depicted in figure 2. We now describe the general idea of each step.

Construct multiple random solutions greedily. In the RPDPT, there is a vast number of restrictions on time, capacity, capability and node sequencing. Due to the constrained nature of our problem, it is likely that parts of the feasible region will be *disconnected*, that is, we cannot transform a feasible solution from

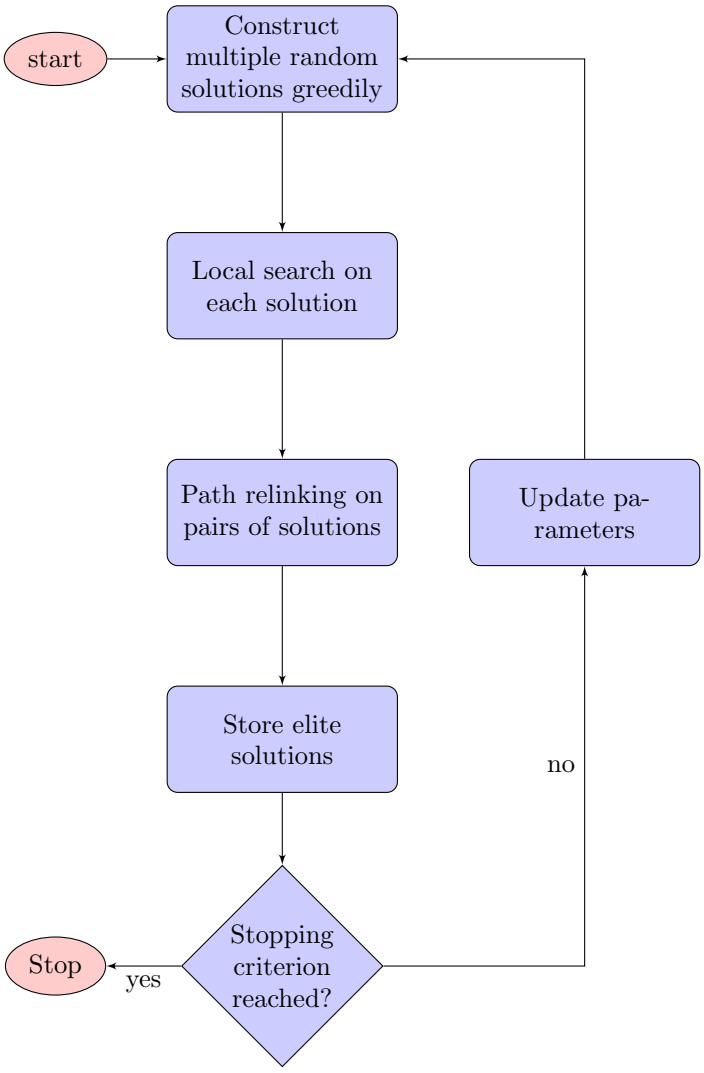

Figure 2: Outline of reactive GRASP with path relinking

one region to a feasible solution from the other region by a series of moves from small neighborhoods, where all intermediate solutions are feasible. This means that local search procedures will quickly get stuck in local optima. Therefore, it is important for highly constrained problems to heavily diversify the search in the early stages of the algorithm. In this step, RGPR constructs multiple solutions. It creates each solution by iteratively inserting requests in routes until all requests are handled. For each request, all feasible insertion positions are evaluated based on the cost of insertion. The algorithm does not choose the cheapest insertion move, but randomly chooses an insertion move from a restricted list of the top-ranked moves. In this way, the average solution quality decreases, but the search is diversified which leads to a higher probability of finding good solutions.

Local search on each solution. In the previous step, all solutions were created greedily, which means that it is likely that local search methods can improve each individual solution. We define multiple ruin and recreate neighborhoods. In each iteration, the algorithm randomly chooses a neighborhood in which the algorithm tries to find a move that improves the incumbent. The size of the neighborhood increases if the algorithm was not able to improve the incumbent.

Path relinking on pairs of solutions. Path relinking is used as an intensification strategy that combines the features of two different solutions. Each solution that results from the local search phase is matched with an elite solution, which is one of the already found solutions with the lowest cost. The path relinking procedure transforms one solution in another solution by making a sequence of small changes. The best intermediate solution found in this process is returned by the algorithm.

Store elite solutions. After each RGPR iteration, a list with the best solutions is updated. This solution pool is contains the solutions with a low cost, but is also diversified in order to make the path relinking procedure more effective.

Stopping criterion reached? The algorithm terminates if the maximum number of iterations is reached.

Update parameters. Our RGPR algorithm is reactive and embedded in an ant colony optimization framework, what means that current solutions are used to alter the search for new solutions in the future. Based on the current solution pool, we search for solution characteristics that are present in good solutions and absent in bad solutions. Based on these characteristics, the cost function that is used to evaluate (partial) solutions in the algorithm is altered in favor of those characteristics. This reactive cost function biases the algorithm towards favorable patterns in the solution, while keeping the search diversified by the randomness present in the algorithm. Additionally, the probability of investigating a neighborhood in the local search phase is altered based on the relative success of each neighborhood to find a cost reducing move.

5.2 Construction phase

In this section, we give a detailed description of the construction phase of the algorithm, which is mainly inspired by Qu and Bard [2012].

5.2.1 Route representation

A (partial) solution S of a RPDPT is fully represented by the following information:

- A set of routes R, where route $r_k \in R$ of vehicle k is represented by a sequence of nodes, starting with b_k and ending with b'_k. If b_k and b'_k are the only nodes in route r_k, we refer to the route as being null or empty.
- For each node i in a route, t_i represents the arrival time at that node, σ_i^f the time spent on fixed (un)loading at i, σ_i^v the time spent on variable loading at i and q_i represents the remaining capacity of the vehicle just before arriving at this node.
- The set of unsatisfied requests $U \subseteq C$, which contains all requests that are not in any of the routes in K.

5.2.2 Request insertion

The basis for the construction phase is the sequential insertion of requests $i_c \rightarrow j_c$ for customer c into routes of the partial solution S. Due to the presence of transshipment points, the pick-up node can be either i_c or $i_{c\tau}$ and the delivery node can be either j_c or $j_{c\tau}$. There are two types of insertion operations. The first one is *single insertion*, where request $i \rightarrow j$ for customer c is handled by inserting the nodes i_c and j_c in route r_{k_1}. Both nodes must be inserted after node b_{k_1} and before node b'_{k_1}. Moreover, node j_c must be inserted after node i_c, as the request can only be delivered if it is picked up first.

The second type of insertion is *double insertion*, whereby the request is shipped via a transshipment location. The request $i \rightarrow j$ for customer c is split up in the two requests $i_c \rightarrow j_{c\tau}$ and $i_{c\tau} \rightarrow j_c$ for some $\tau \in \mathcal{T}$. Request $i_c \rightarrow j_{c\tau}$ is inserted into route r_{k_1} and request $i_{c\tau} \rightarrow j_c$ is inserted into route r_{k_2}. Possibly, r_{k_1} and r_{k_2} are the same routes. Again, the nodes must be inserted after the start node end before the end node of the route. Furthermore, i_c must be inserted before $j_{c\tau}$ and $i_{c\tau}$ must be inserted before j_c.

Algorithm 1 describes the basic insertion procedure. The algorithm tries all possible insertion positions of both single and double insertions, and returns the n_C^{max} cheapest ones, according to some cost function $Q()$, which is further specified in equation (33). For step 1, all insertion combinations are tried for $|K|$ routes. Since the CheckFeasible procedure runs in $\mathcal{O}(|N|)$ time, step 1 has a time complexity of $\mathcal{O}\left(|K||N|^3\right)$. Step 2 evaluates for each pair of routes $\left(\mathcal{O}\left(|K|^2\right)\right)$, for each transshipment location $(\mathcal{O}(|T|))$ and each possible insertion position $\left(\mathcal{O}\left(|N|^4\right)\right)$ the feasibility $(\mathcal{O}(|N|))$, resulting in a computational complexity of $\mathcal{O}\left(|T||K|^2|N|^5\right)$. The complexity of the BasicInsertion algorithm is therefore $\mathcal{O}\left(|T||K|^2|N|^5\right)$.

Algorithm 1 Basic insertion

Input: Request c, partial solution S, maximum number of solution n_C^{max}
Output: Set of solutions SL resulting from the best n_C^{max} possible insertions of c in S

function BasicInsertion(c, S, n_C^{max})
//Step 0: Initialization
  $SL \leftarrow \emptyset$
//Step 1: single insertions
  for all $r \in R$ **do**
    for all possible insertions of $i_c \rightarrow j_c$ in r **do**
      S_0 is the solution with the inserted nodes and $NL = \{i_c, j_c\}$
      if CheckFeasible(S_0, NL) **then**
        Remove c from open shipment pool U in S_0
        if $|SL| < n_C^{max}$ **then**
          $SL \leftarrow SL \cup \{S_0\}$
        else
          $S_{worst} = \text{argmax}_{S \in SL}(Q(S))$
          if $Q(S_{worst}) > Q(S_0)$ **then**
            Replace S_{worst} with S_0 in SL
          end if
        end if
      end if
    end for
  end for
//Step 2: double insertions
  for all $r_1 \in R$ **do**
    for all $r_2 \in R$ **do**
      for all $\tau \in \mathcal{T}$ **do**
        for all insertions of $i_c \rightarrow j_{c\tau}$ in r_1 and $i_{c\tau} \rightarrow j_c$ in r_2 **do**
          Set $S_0 = S$ and $NL = \{i_c, j_{c\tau}, i_{c\tau}, j_c\}$
          if CheckFeasible(S, NL) **then**
            Remove c from open shipment pool U in S_0
            if $|SL| < n_C^{max}$ **then**
              $SL \leftarrow SL \cup \{S_0\}$
            else
              $S_{worst} = \text{argmax}_{S \in SL}(Q(S))$
              if $Q(S_{worst}) > Q(S_0)$ **then**
                Replace S_{worst} with S_0 in SL
              end if
            end if
          end if
        end for
      end for
    end for
  end for
  return SL
end function

5.2.3 Solution update and feasibility check

After each single or double insertion into S, the route, time and capacity information contained in S must be updated accordingly. Also, it must be checked if the performed insertion is indeed feasible. If new nodes are inserted in S, not all nodes in S are affected. For efficiency reasons, the updating algorithm therefore only checks affected nodes, which are stored in the list NL. After a single insertion of $i \rightarrow j$, the nodes i and j are added to NL. After a double insertion of $i_1 \rightarrow j_1$ and $i_2 \rightarrow j_2$, these four nodes are added to NL. When the request $i \rightarrow j$ is deleted, the nodes right after i and j in their original routes are added to NL. The algorithm takes the original solution and NL as input, and returns whether the insertion or deletion results in a feasible solution. If that is the case, it also returns the solution with the updated time and capacity information. The algorithm for the solution update and feasibility check is given in algorithm 2.

We will clarify some of the steps in algorithm 2. The first and computationally most inexpensive feasibility check is performed in step 2. Requests can only be inserted in routes that are performed by vehicles that are capable to handle that request.

In step 3, the time and capacity information of the nodes is updated, making use of algorithm 3. In line (3)-(9) of algorithm 3, it is calculated whether the node incurs fixed and variable time cost. This information is needed for the calculations in line (10) and (18). Note for example, that fixed (un)loading cost are not included when the previous node had the same geographical location. In line (10), the arrival time at the node is calculated, based on the arrival time at the previous node and the time windows. In line (18), the arrival time is adjusted for transshipment pick-up nodes, such that it can start its variable loading tasks only after the unloading of the request has been finished.

In step 4 of algorithm 2, the nodes that are affected by the changes in the current node are identified. The affected nodes are the nodes right after the current node, as time and capacity values have changed. Secondly, it should be checked if deliveries are not planned before pick-ups. However, with these propagation rules, infinite loops may occur. Suppose for example that the vehicle of route r_1 loads request c_1 and subsequently unloads request c_2. Furthermore, vehicle r_2 unloads request c_1 and loads request c_2. Such a cross-synchronization cannot be feasible and will cause infinite looping of the algorithm. In line (5), cases of cross-synchronization are recognized, and the solution is marked as infeasible.

In order to speed up the algorithm, the list FL should always be sorted in such a way that if node j is visited before node i, j should be listed higher than i. In the opposite scenario, all nodes after i would first be updated. After that, they would need updating again, since all nodes after j would be updated. The proposed list sorting speeds up the algorithm by avoiding double updating of nodes.

Algorithm 2 Solution update and feasibility check

Input: Partial solution S, set of affected nodes NL
Output: Updated partial solution S', feasibility indicator *feasible*

function CHECKFEASIBLE(S, NL)
//Step 0: initialization
  $FL \leftarrow NL$
//Step 1: check affected nodes
  if $FL = \emptyset$ **then**
    return S', *feasible* = true
  else if FL contains a node that is located anywhere in a route before a transshipment delivery node in NL **then**
    return *feasible* = false
  else
    $i = FL(0)$, k is the associated vehicle, c the associated request
    $FL \leftarrow FL \setminus \{i\}$
  end if
//Step 2: check capability
  if $k \notin K(i)$ **then**
    return *feasible* = false
  end if
//Step 3: update node information
  $S' =$UPDATENODEINFORMATION(S, i)
  if $(t_i > b_i \wedge \mathcal{G}(i) \neq \mathcal{G}(i-1)) \vee q_i > Q_k$ **then**
    return *feasible* = false
  end if
//Step 4: Propagation
  Find if there is a node j visited after i
  $FL \leftarrow FL \cup \{j\}$
  if node i is a transshipment delivery node **then**
    Find the corresponding transshipment pick-up node
    $FL \leftarrow FL \cup \{j\}$
  end if
  Go to step 1
end function

Algorithm 3 Update node information

Input: Partial solution S, node i
Output: Updated partial solution S'

function UpdateNodeInformation(S, i)
  //Step 1: update node information based on route path
    Let j be the node visited right before i
    if $\mathcal{G}(i) = \mathcal{G}(j)$ **then**
      $\sigma_i^f = 0, \sigma_i^v = \sigma_k^v \cdot L_c$
    else if $i = b'_k$ **then**
      $\sigma_i^f = 0, \sigma_i^v = 0$
    else
      $\sigma_i^f = \sigma_k^f, \sigma_i^v = \sigma_k^v \cdot L_c$
    end if
    $t_i = \max\{t_j + \sigma_j^f + \sigma_j^v + T_{ji}, a_i\}$
    if node j is pick-up node for customer c' **then**
      $q_i = q_j + L_{c'}$
    else if node j is delivery node for customer c' **then**
      $q_i = q_j - L_{c'}$
    end if
  //Step 2: update node information based on shipment path
    if i is a transshipment pick-up node **then**
      Find the corresponding transshipment delivery node j
      $t_i = \max\{t_i, t_j + \sigma_j^f + \sigma_j^v - \sigma_i^f\}$
    end if
    return updated partial solution S'
end function

Performing step 1 to 4 in algorithm 2 has a computational complexity of $\mathcal{O}(1)$. In the worst case scenario, the algorithm loops over all nodes in $N \backslash \{b_k | k \in K\}$. The computational complexity of algorithm 2 is therefore $\mathcal{O}(|N|)$.

5.2.4 Solution construction

Algorithm 4 states how a solution is constructed from scratch. First, a list CL is created that contains the best n_C^{max} insertions, aggregated over all requests. Randomly, one of these insertions is chosen based on the probability function $PROB$. This process is repeated until all requests are routed, or until no feasible insertion can be found. Steps 1 and 2 are executed $|C|$ times. Step 1 loops over all requests and in each loop, the BasicInsertion algorithm is called. Therefore, the SolutionConstruction algorithm has a computational complexity of $\mathcal{O}\left(|T||K|^2|N|^5|C|^2\right)$.

5.2.5 Computation time improvement

As mentioned in section 5.2.4, the theoretical computational complexity of the construction phase is $\mathcal{O}\left(|T||K|^2|N|^5|C|^2\right)$. Although polynomial, in practice the computation times turn out to be too high for larger problems. Since a lower theoretical computation complexity can only be achieved at the cost of the quality of the construction phase, we aim to lower computation times by diminishing the practical computation time.

Reduce combinations double insertion
A closer look at the construction algorithm can reveal where the main computation burden is located. Note that the values of $|T|$, $|K|$ and $|C|$ are relatively small in comparison with $|N|$. This $|N|$ is raised to the fifth power since we evaluate all insertion position combinations for requests that make use of double insertion. Request c for transshipment node τ is split up in request $i_c \rightarrow j_{c\tau}$ for route 1 and request $i_{c\tau} \rightarrow j_c$ for route 2. This operation requires checking $\mathcal{O}\left(|N|^4\right)$ configurations in combination with a feasibility check of $\mathcal{O}(|N|)$ for each combination.

We develop a faster way to evaluate all these combinations. Therefore, we take the following three steps:

1. Try all insertion positions of $i_c \rightarrow j_{c\tau}$ in route 1 and store all feasible positions in combination with the resulting cost of the solution in list L_1. Subsequently, do the same for request $i_{c\tau} \rightarrow j_c$ for route 2 and store the results in list L_2. In this step we evaluate $\mathcal{O}\left(|N|^2\right)$ different combinations and do perform a feasibility check for each. The complexity therefore equals $\mathcal{O}\left(|N|^3\right)$.

2. Sort L_1 and L_2 in ascending order based on their cost. This step has a computational complexity of $\mathcal{O}(|N| \log |N|))$.

Algorithm 4 Solution construction

Input: Set of requests U, set of vehicles K, maximum number of solutions n_C^{max}
Output: Solution S with all requests handled

function SolutionConstruction(U, K, n_C^{max})
Step 0: Initialization
    Initialize all routes in S as empty
//Step 1: build candidate list CL
    $CL \leftarrow \emptyset$
    for $u \in U$ **do**
        $SL =$ BasicInsertion (u, S, n_C^{max})
        for $S_0 \in SL$ **do**
            if $|CL| < n_C^{max}$ **then**
                $CL \leftarrow CL \cup \{S_0\}$
            else
                $S_{worst} = \text{argmax}_{S \in CL}\,(Q(S))$
                if $Q\,(S_{worst}) > Q\,(S_0)$ **then**
                    Replace S_{worst} with S_0 in CL
                end if
            end if
        end for
    end for
    if $CL = \emptyset$ **then**
        No feasible solution can be found. Stop.
    end if
//Step 2: choose new solution
    Select new solution S randomly from CL with probability $\frac{1}{|CL|}$
    if $U = \emptyset$ **then**
        return S
    end if
    Go to step 1
end function

3. Pick the insertion positions at the top of the list L_1 and L_2 and insert $i_c \to j_{c\tau}$ and $i_{c\tau} \to j_c$ at these positions in the original solution. If this results in a feasible solution, we stop and return the solution. Otherwise, we take the next least cost combination from list L_1 and L_2 and repeat step 3. If we do never find a feasible solution, such a solution does not exist.

Note that the theoretical complexity of step 3 still equals $\mathcal{O}\left(|N|^5\right)$ as we have to iterate through all possible combinations of insertion positions in L_1 and L_2 in the worst case. However, since the algorithm terminates in step 3 as soon as we find a feasible solution, the practical computation time will decrease drastically. Note that the returned solution is guaranteed to be the least cost solution regarding the double insertion of c into the solution.

Anticipate infeasibility
Suppose that in the BasicInsertion algorithm we insert request $i \to j$ in a route r. We now check all possible insertion combinations of i and j and check their feasibility. However, if an infeasibility occurs due to the position of i in the route, we do not need to check all other (i, j)-combinations with i at that specific position. But we can only skip these combinations provided that the infeasibility was not caused by the position of j in the route. Since we update the node information in the direction from the start node to the end node, this is only true if the infeasibility was detected at a node 1) in r before j, or 2) in a route that is being updated due to the change of node information of a node before j.

Arriving too late
If the insertion of i at position x in route r is infeasible due to the fact that we do not arrive within the time-window of node i, we also do not have to evaluate all insertion combinations with i in r at position $x+1, x+2, \ldots$. We would also arrive too late at these positions, as visiting more nodes before i only can delay our arrival at i.

Capabilities
If the insertion of i in route r is infeasible due to capabilities, all insertion combination with i in r are also infeasible and do not need to be checked.

5.3 Improvement phase

In the local search phase of the RGPR algorithm, we try to improve each individual solution that resulted from the construction phase. Many heuristic approaches in vehicle routing make use of local search, in which small alterations of the incumbent solution are evaluated. A relatively new approach is the ruin and recreate method, in which a relatively large part of the solution is first destroyed and then rebuilt. Such an approach is particularly useful for complex problems that have to meet a lot of constraints. Neighbor solutions

are often infeasible, which causes local search methods to be quickly stuck in local optima. A ruin and recreate approach defines a larger neighborhood and it thus more likely to find admissible and better solutions.

A general outline of the ruin and recreate phase is given in algorithm 5. In each iteration, a ruin and a recreate method are selected in line (5). According to these methods, the solution is destroyed and subsequently rebuilt (line (6)-(7)). The algorithm starts by exploring a small neighborhood, removing and inserting only one request. If after $iter_u$ iterations this does not result in an improvement of the incumbent solution, the neighborhood is expanded, up to removing and inserting u_{max} requests. If this also does not improve the incumbent solution, the solution is allowed to drift (line (8)-(18)). If this results in a new global optimum, this solution is stored in line (9). After all the solutions are handled by the ruin and recreate procedure, the weights which influence the likelihood of a heuristic to be chosen are updated, as further described in section 5.3.3.

Algorithm 5 Ruin and recreate

Input: Set of solutions S_{set}, maximum number of removed requests u_{max}, number of iterations $iter_{rr}$ and $iter_u$
Output: Set of improved solutions S^*_{set}

1: **function** RuinAndRecreate($S, u, iter_{rr}, iter_u$)
2: **for** $S \in S_{set}$ **do**
3: $S^* \leftarrow S$, $u \leftarrow 1$, $S_{init} \leftarrow S$
4: **for** $iter_{rr}$ times **do**
5: Randomly choose a ruin and a recreate method based on weight
6: $S \leftarrow Ruin(S_{init}, u)$
7: $S \leftarrow Recreate(S, u)$
8: **if** $Q(S) < Q(S^*)$ **then**
9: $S^* \leftarrow S$, $S_{init} \leftarrow S$, $u \leftarrow 1$, $count_u \leftarrow 1$
10: **else**
11: $count_u \leftarrow count_u + 1$
12: **end if**
13: **if** $count_u = iter_u$ **then**
14: $u \leftarrow u + 1$, $count_u \leftarrow 0$
15: **end if**
16: **if** $u = u_{max}$ **then**
17: $S_{init} \leftarrow S$, $count_u \leftarrow 0$
18: **end if**
19: **end for**
20: $S^*_{set} = S^*_{set} \cup S^*$
21: **end for**
22: Update heuristic weights
23: **return** S^*_{set}
24: **end function**

5.3.1 Ruin methods

In this section, we develop several methods that destroy the current solution. Each ruin method removes u requests from the solution and stores them in the set U. When a request is removed from a solution, the pick-up and delivery node are taken out, plus the transshipment nodes if applicable. We make a distinction between methods that focus on the transshipment points and methods that are not specifically designed for transshipment problems.

General ruin methods
We make use of several general ruin methods, each with their own advantages.

Random removal This method randomly removes u requests with equal probability. This method assures diversification of the ruin and recreate procedure, which allows us to avoid getting stuck in local optima.

Shaw removal The Shaw removal heuristic aims to remove requests that are in some sense similar to each other. This heuristic was first proposed in Shaw [1997], and we adapt it to our problem. Especially in highly constrained problems, it is more likely that a request can be inserted in a different place if similar requests are removed. The similarity of two requests depends on the load, time windows, pick-up location, delivery location and vehicle capabilities. Equation (18) defines the relatedness measure for request c_1 and c_2.

$$\begin{aligned} R_{c_1,c_2} =& \frac{\theta_1}{L^*}|L_{c_1} - L_{c_2}| + \frac{\theta_2}{D^*}\left(D_{i_{c_1},i_{c_2}} + D_{j_{c_1},j_{c_2}}\right) \\ &+ \frac{\theta_3}{T^*}\left(|a_{i_{c_1}} - a_{i_{c_2}}| + |b_{i_{c_1}} - b_{i_{c_2}}| + |a_{j_{c_1}} - a_{j_{c_2}}| + |b_{j_{c_1}} - b_{j_{c_2}}|\right) \\ &+ \frac{\theta_4}{K^*}\left(\frac{1}{2} - \frac{|K(i_{c_1}) \cap K(i_{c_2})| + |K(j_{c_1}) \cap K(j_{c_2})|}{|K(i_{c_1})| + |K(i_{c_2})| + |K(j_{c_1})| + |K(j_{c_2})|}\right) \end{aligned} \tag{18}$$

The constants in equation (18) are calculated as follows:

$$C^* = \frac{1}{2|C|(|C|-1)} \tag{19}$$

$$L^* = C^* \sum_{c_1 \in C} \sum_{c_2 \in C} |L_{c_1} - L_{c_2}| \tag{20}$$

$$D^* = C^* \sum_{c_1 \in C} \sum_{c_2 \in C} \left(D_{i_{c_1},i_{c_2}} + D_{j_{c_1},j_{c_2}}\right) \tag{21}$$

$$T^* = C^* \sum_{c_1 \in C} \sum_{c_2 \in C} \left(|a_{i_{c_1}} - a_{i_{c_2}}| + |b_{i_{c_1}} - b_{i_{c_2}}| + |a_{j_{c_1}} - a_{j_{c_2}}| + |b_{j_{c_1}} - b_{j_{c_2}}|\right) \tag{22}$$

$$K^* = C^* \sum_{c_1 \in C} \sum_{c_2 \in C} \left(\frac{1}{2} - \frac{|K(i_{c_1}) \cap K(i_{c_2})| + |K(j_{c_1}) \cap K(j_{c_2})|}{|K(i_{c_1})| + |K(i_{c_2})| + |K(j_{c_1})| + |K(j_{c_2})|}\right) \tag{23}$$

According to equation (18), if request c_1 and c_2 are more related, the value of R_{c_1,c_2} becomes smaller. Requests are more related if their load is more similar and if the distances between their respective pick-up locations and delivery locations are smaller. Additionally, requests are more similar if their time windows overlap more. The fourth term decreases if more vehicles can handle the pick-ups or deliveries of both requests, and increases in the number of vehicles that can handle at least one task of these two requests. Note that $R_{c_1,c_2} \geq 0$ and $R_{c_1,c_1} = 0$ for all c_1 and c_2.

Each term is weighed with a positive parameter $\theta_1, \theta_2, \theta_3$ or θ_4. In order to arrive at meaningful parameter values for a variety of instances, we scale each parameter with an instance specific value as defined in equations (19)-(23). These scaling values are calculated by averaging the corresponding term over all request combinations.

The Shaw removal heuristic executes the following steps:

1. Randomly pick a request c_1 and remove it from the solution.
2. Sort all the requests c_2 that remain in the solution in list L in ascending order based on their relatedness R_{c_1,c_2}.
3. Choose a random number $p \in (0, 1)$ and remove the request on position $\lceil p^{\alpha_s}|L| \rceil$ from the solution and from L. Repeat step 3 until u requests are removed.

The value of the parameter $\alpha_s \in [1, \infty)$ determines the degree of randomness in choosing related requests. If $\alpha_s = 1$, the heuristic is the same as the random removal heuristic. If $\alpha_s = \infty$, the $u - 1$ most similar requests are chosen. The value of α_s is fixed, and set by making use of tuning instances.

Route removal This heuristic randomly picks a route and removes all the requests which are handled in that route. We keep removing routes until a route removal would result into more than u requests being removed. In that case, the random removal heuristic is applied to that route to bring the amount of deleted requests up to u. It can happen that a route only contains a segment of a request, that is, it handles either the pick-up or the delivery, but not both. In that case, the other segment is also removed from the current solution. The aim of this ruin method is to allow for new route structures to arise, by freeing up a vehicle.

Ruin methods dedicated to transshipment

We develop two more ruin methods, that are specifically designed to take advantage of the transshipment option.

Transshipment removal The transshipment removal heuristic randomly chooses a transshipment point and removes all requests that make use of the transshipment point in the current solution. We keep selecting new transshipment points, until the removal of all related requests would lead to a removal of

more than u requests. In that case, we remove a random subset of these requests to bring the number of deleted requests op to u. If the total number of requests that are routed via a transshipment point is less than u, only these requests are removed. This heuristic aims to reroute requests via other transshipment points.

Cluster removal Requests of which the pick-up locations or the delivery locations are geographically close to each other are more likely to be handled by the same vehicle in the optimal solution. For example, one vehicle might pick-up all of these clustered requests and deliver them to a transshipment point, from which multiple vehicles deliver them to their geographically scattered delivery locations. The cluster removal heuristic therefore takes the following steps:

1. Randomly pick a request c_1 and remove it from the solution. Also, randomly decide whether we consider pick-up or delivery locations.

2. Sort all the requests c_2 that remain in the solution in list L in ascending order based on either the distance $D_{i_{c_1},i_{c_2}}$ or $D_{j_{c_1},j_{c_2}}$.

3. Choose a random number $p \in (0,1)$ and remove the request on position $\lceil p^{\alpha_c}|L| \rceil$ from the solution and from L. Repeat step 3 until u requests are removed.

If the value of the parameter $\alpha_c \in [1,\infty)$ increases, the degree of randomness in choosing nearby requests also increases. In the case that $\alpha_c = 1$, the heuristic is equal to the random removal heuristic. Alternatively, if $\alpha_c = \infty$, the $u-1$ nearest requests are chosen. The value of α_c is fixed, and determined by using tuning instances.

5.3.2 Recreate methods

After the ruin phase, a partial solution and u or more unhandled requests remain. The goal of the recreate methods is to insert these unhandled requests back in the partial solution against the lowest cost. It is important to emphasize that the recreate methods only determine the order in which the requests are inserted. The insertion place in the solution for each request is determined in the same way as in the construction phase, which comes down to inserting a request at that position in the solution which leads to the smallest cost increase.

Greedy insertion The greedy insertion method is identical to the insertion method in the construction phase described in algorithm 1. All possible insertion moves for all requests are evaluated, and a move is randomly selected from a list of moves that lead to the smallest cost increase. The solution is updated, and the insertion process is repeated with the unhandled requests.

Since we want to add more intelligence to the improvement phase, the order of insertion can also be chosen with more advanced methods than the greedy method.

Regret k-insertion The regret k-insertion method first inserts the requests which would lead to the largest regret if they were not inserted first. For every request c, the k best insertion positions in the solution are evaluated, with associated cost $f_c^1, \ldots, f_c^k$. Then, request c^* is inserted, with

$$c^* = \underset{c \in U}{\operatorname{argmax}} \sum_{j=2}^{k} \left(f_c^j - f_c^1 \right) \tag{24}$$

Then, c^* is inserted at its most profitable position and removed from U, and the process is repeated until all requests are inserted. The intuition behind this insertion heuristic is that we should first insert requests which have only a few low-cost insertion positions, and many high-cost insertion positions.

Random insertion The random insertion methods randomly determines the order of insertion. This method assures diversification in the improvement phase.

5.3.3 Method selection

The effectiveness of ruin and recreate methods may vary per method and per problem instance. In order to speed up the algorithm, we would like to favor the successful heuristics over the less effective ones. Therefore, we choose each method based on their weight using the roulette wheel principle, as proposed by Ropke and Pisinger [2006]. If we have k heuristics with weight $w_i, i \in \{1, \ldots, k\}$, we choose heuristic j with probability

$$p_j = \frac{w_j}{\sum_{i=1}^{k} w_i} \tag{25}$$

We let the weights reflect the effectiveness of each heuristic by adapting these weights based on the capability of the heuristic to find both better and new solutions. When entering the improvement phase of the algorithm, all scores π_i of heuristic i are set to zero. Each time heuristic i is used in the improvement phase, this score can be increased. The score is incremented by three if the heuristic was able to find a new global optimum. The score is incremented by two if the heuristic found a new solution that also improved the incumbent. The score is incremented by one of the heuristic did find a new solution. In all other cases, the score stays level.

The weights of the heuristics are updated at the end of the improvement phase. If we have completed the improvement phase for the j-th time, the weight $w_{i,j+1}$ of heuristic i for the next improvement phase is equal to

$$w_{i,j+1} = w_{i,j}(1 - r) + r\frac{\pi_i}{\theta_i} \tag{26}$$

The value of $r \in [0, 1]$ in equation (26) determines how quickly the weights change based on recent performance. θ_i counts how many times heuristic i is used. At the start of the RGPR, we do not know which heuristics will be most effective, so we set all $w_{i,1}$ equal to each other.

5.4 Path relinking

The purpose of the path relinking procedure is to explore paths between pairs of elite solutions of the RPDPT. The first solution is referred to as the initial solution, and the second solution is called the guiding solution. Path relinking gradually transforms the initial solution into the guiding solution by choosing moves from its neighborhood. Therefore, the solutions that are explored along the way contain characteristics of both the initial and the guiding solution, which constitutes the main power of the path relinking procedure.

In this chapter, we first describe how the path relinking procedure works on a given pair of solutions. Secondly, we show how the path relinking procedure is embedded within the RGPR.

5.4.1 Solution representation

When transforming the initial solution into the guiding solution, we do only evaluate solutions in terms of the requests that each vehicle handles, and the order in which it does so. The capacity and time information is not relevant in the definition of path relinking neighborhoods and distance measures. All the solution information that is required for these steps can therefore be accessed by using the following two functions:

- The function $F_1(S_1, S_2, k)$ returns for solution S_1 and S_2 and vehicle $k \in K$ the ordered set that contains all the nodes in the route handled by vehicle k in solution S_1 that are also handled by k in solution S_2.
- The function $F_2(S, k)$ returns for solution S and vehicle $k \in K$ the ordered set that contains all the request legs that are handled by k.

A request leg is defined as the combination of a pick-up node and a delivery node of a request. For example, request c can either be handled in one request leg $i_c \rightarrow j_c$ or in two request legs $i_c \rightarrow j_{c\tau}$ and $i_{c\tau} \rightarrow j_c$.

5.4.2 Neighborhoods

The path relinking procedure transforms the initial solution into the guiding solution by iteratively selecting moves from neighborhoods. In this section, we define these neighborhoods. A neighborhood needs to satisfy two conditions. Firstly, all requests in C must be present in the solution. Secondly, the order of nodes in the solution must be feasible with respect to the pick-up and delivery constraints, i.e., pick-ups have to happen before deliveries. We use the following two neighborhoods:

- Replace node: remove a node from a route and insert it at a different position in the same route, while satisfying the pick-up and delivery constraints.

- Relocate request: delete all request legs that are associated with a request and insert the new request legs at a different route. The insertion positions of the nodes must be such that the *replace node* neighborhood does not have to be applied on these nodes in order to arrive at the guiding solution.

By exclusively making use from moves of these two neighborhoods, each initial solution can be transformed to every guiding solution.

5.4.3 Solution distance

The path relinking distance $PRD(S_a, S_b)$ between two solutions S_a and S_b is defined as a weighted minimum number of neighborhood moves that is needed to transform solution S_a into solution S_b. This distance can be seperated in two parts: the weighed minimum number of *relocate request* moves $RRD(S_a, S_b)$ that is needed and the minimum number of *replace node* moves $RND(S_a, S_b)$ that is needed. Hence,

$$PRD(S_a, S_b) = RRD(S_a, S_b) + RND(S_a, S_b) \tag{27}$$

The correctness of equation (27) follows directly from the observation that a number of *relocate request* moves cannot lead to the same result as a number of *replace node* moves and vice versa. $RRD(S_a, S_b)$ is calculated as in equation (28).

$$\begin{aligned} RRD(S_a, S_b) = &\sum_{k \in K} \left(|F_2\left(S_a, k\right)| - |F_2\left(S_a, k\right) \cap F_2\left(S_b, k\right)| \right) \\ &+ \sum_{k \in K} \left(|F_2\left(S_b, k\right)| - |F_2\left(S_a, k\right) \cap F_2\left(S_b, k\right)| \right) \end{aligned} \tag{28}$$

The first term of equation (28) calculates the number of request legs to delete, and the second term of equation (28) calculates the number of request legs to insert. Note that the weight of a *relocate request* move depends on the number of request legs associated with this move. For example, the deletion of request legs $i_c \rightarrow j_{c\tau}$ and $i_{c\tau} \rightarrow j_c$ from one route and the insertion of $i_c \rightarrow j_c$ in another, has a weight of 3, while a relocation of request $i_c \rightarrow j_c$ to another route has a weight of 2.

Now we need to define $RND(S_a, S_b)$, the minimum number of *replace node* moves needed. First, we observe that moves in the *relocate request* neighborhood already insert the corresponding nodes at such positions that they do not need to be relocated anymore. Therefore, to calculate $RND(S_a, S_b)$, it suffices to look for each vehicle $k \in K$ at the partial routes $S_{a,k}^{RND} = F_1(S_a, S_b, k)$ and $S_{b,k}^{RND} = F_1(S_b, S_a, k)$, that is, to look at the sequence of nodes corresponding with all requests that will not be subject to a *relocate request* move.

Now, we can calculate the number of moves needed as in equation (29).

$$\begin{aligned} RND(S_a, S_b) &= RND(S_a^{RND}, S_b^{RND}) \\ &= \sum_{k \in K} \left(|S_{a,k}^{RND}| - SS(S_{a,k}^{RND}, S_{b,k}^{RND}) \right) \end{aligned} \tag{29}$$

Here, the function $SS(S_{a,k}^{RND}, S_{b,k}^{RND})$ determines the length of the longest subsequence in $S_{a,k}^{RND}$ that is ordered as in $S_{b,k}^{RND}$. For example, $SS((1,5,3,2,6,4), (1,2,5,3,4,6))$ equals 4, as the sequence $(1,5,3,6)$ is present in both sequences. The minimum number of moves is therefore $6-4=2$: only the numbers 2 and 4 do need to be replaced.

Note that the distance measure $PRD(S_a, S_b)$ is symmetric, so $PRD(S_a, S_b) = PRD(S_b, S_a)$.

Symmetry problems
In the RPDPT, a subset of the vehicles might be homogeneous. In the above definition of the distance measure this may cause very similar solutions to have a large distance. For example, if in one solution vehicle A executes the same route as the identical vehicle B in the second solution, the path relinking procedure will unnecessarily try to transfer all requests from vehicle A to vehicle B.

Therefore, we want to pair vehicles in the one solution with vehicles in the other solution in such a way that the distance measure is minimized. As the number of possible assignments is factorial in the size of the homogenous subset of vehicles, we rely on a greedy heuristic to make the assignment. A random vehicle from one solution is selected and paired with the vehicle of the other solution which has the most shared nodes in its route. In order to guarantee the finiteness of the path relinking algorithm, the same vehicle assignment is kept until the guiding solution is reached.

5.4.4 Path relinking algorithm

Algorithm 6 describes the procedure for path relinking. In line (2), the best found solution is initialized. Line (3) initializes the current solution, which is gradually transformed from S_a to S_b. The algorithm starts in line (5) with considering all possible moves on the current solution. This can be either *relocate request* nodes or *replace node* moves. The moves that result in a solution which is closer to the guiding solution than the current solution are stored. In line (6) the lowest cost solution is picked from this list.

Since the RPDPT is often heavily constrained, it is likely that a move would result in a solution which is infeasible with respect to the time and capacity constraints. Therefore, we try to improve each solution in step (7). This step executes the RuinAndRecreate algorithm with one key difference. If a request is handled by direct delivery, the recreate methods will never assign the request to a transshipment point. Also, if a request is handled via transshipment point τ, the recreate methods will always send the request via the same transshipment point. The reason for this adaption is that the main strength of the path relinking algorithm is that it combines the solution structures of two different solutions. Since the decisions whether requests are sent via a transshipment point, and which one, are defining for the structure of the problem, we do not allow the AugmentedRuinAndRecreate algorithm to change this structure.

Algorithm 6 Path relinking

Input: Initial solution S_a and guiding solution S_b
Output: Improved solution S^*
1: **function** PathRelinking(S_a, S_b)
2: $S^* \leftarrow S_a$
3: $S_c \leftarrow S_a$
4: **while** $S_c \neq S_b$ **do**
5: Store all solutions S_m in the neighborhood of S_c in list L, for which $PRD(S_m, S_b) < PRD(S_c, S_b)$
6: $S_c \leftarrow$ the feasible solution S_m in L with the lowest $Q(S_m)$. If there are no feasible solutions, the one with the lowest $Q(S_m)$
7: $S_c^* \leftarrow$AugmentedRuinAndRecreate(S_c)
8: **if** $Q(S_c^*) < Q(S^*)$ **then**
9: $S^* \leftarrow S_c^*$
10: **end if**
11: **end while**
12: **return** S^*
13: **end function**

In line (8), the best solution found is stored. If the path from the initial solution to the guiding solution is completed, the best intermediate solution is returned in line (12)

5.5 Ant colony optimization framework

The ant colony optimization technique is a metaheuristic that finds it inspiration in the natural behavior of ants. When ants need to find the shortest path between their colony and a food source, they start by randomly choosing a path. Along the way, they all drop a chemical substance called pheromone, which gives off a smell that decreases in strength over time. When ants choose a path randomly, they choose paths with a higher level of pheromone with a higher probability. In this way, the probability that a short path is chosen increases over time, as the pheromone on short paths has less time to decrease than the pheromone on long paths. Eventually, all the ants will choose the shortest path with probability 1.

Our ant colony optimization technique loosely mimics this process. In the first iteration of the RGPR algorithm, we start with a probabilistic procedure that constructs and improves a batch of solutions. The quality of these solutions is evaluated solely on the real cost. However, at the end of the first RGPR iteration, we might discover that good quality solutions exhibit certain characteristics that bad quality solutions do not have and vice versa. We exploit this information by making it more likely that solutions which are constructed and improved in the future, will contain the same characteristics as the good quality solutions. According to the same principle, it will be less likely that future solutions will contain characteristics that are present in bad quality solutions.

5.5.1 Solution characteristics

In this section, we state which solution features will be regarded by the ant colony framework. The set of solution characteristics should have two properties. First of all, it should fully define the solution, that is, there is exactly one solution that satisfies a given set of solution characteristics. This property is needed in order to effectively steer the algorithm in making good quality solutions. Secondly, if a solution characteristic is present in a certain partial solution, it should also be present in the solution if more requests are added to this solution. This property ensures that the algorithm does not become short-sighted in focusing on partial solutions, but aims at finding good quality complete solutions.

We will use the following solution characteristics variables:

$$\xi^1_{nmS} = \begin{cases} 1 \text{ if node } m \text{ is handled after node } n \text{ by the same vehicle in solution } S \\ 0 \text{ else;} \end{cases}$$

$$\xi^2_{nkS} = \begin{cases} 1 \text{ if node } n \text{ is handled by a vehicle with the same vehicle type as} \\ \quad \text{vehicle } k \text{ in solution } S \\ 0 \text{ else;} \end{cases}$$

This set of solution characteristics satisfies both properties. Note that in the definition of ξ^1_{nmS}, this variable is also set to 1 if there are some nodes between n and m: m does not have to be *directly* after node n, as this would not satisfy the second property.

5.5.2 Pheromone update

At the final stage of a RGPR iteration, the pheromone levels ξ^1_{nm} and ξ^2_{nk} associated with each solution characteristic are updated in the following way:

$$\xi^1_{nm} \leftarrow (1-\rho)\xi^1_{nm} + \frac{H}{|\S|}\sum_{S\in\S}\frac{\xi^1_{nmS}}{COST(S)} \tag{30}$$

$$\xi^2_{nk} \leftarrow (1-\rho)\xi^2_{nk} + \frac{H}{|\S|}\sum_{S\in\S}\frac{\xi^2_{nkS}}{COST(S)} \tag{31}$$

Here, COST() is the functions that evaluates the real cost of a solution, § is the set of the newly generated solutions in the current RGPR iteration and $\rho \in [0,1]$ is the pheromone evaporation coefficient that determines the degree of influence of previous iterations on the next one. H is a constant scaling parameter. From equations (30) and (31), it becomes apparent that characteristics exhibited by low-cost solutions are awarded a higher amount of pheromone than characteristics exhibited by high-cost solutions.

5.5.3 Solution quality evaluation

The total amount of pheromone ξ_S associated with a solution S can straightforwardly be calculated as

$$\xi_S = \sum_{n \in N} \sum_{m \in N} \xi^1_{nm} \xi^1_{nmS} + \sum_{n \in N} \sum_{k \in K} \xi^2_{nk} \xi^2_{nkS}. \tag{32}$$

The function that evaluates the quality of a solution is given in equation (33).

$$Q(S) = \frac{(COST(S))^{\alpha}}{(\xi_S)^{\beta}} \tag{33}$$

The quality function $Q(S)$ is decreasing in the real cost of the solution S and is increasing in the amount of pheromone associated with the solution. The values of the parameters α and β jointly define the influence of both factors on the solution quality. If $\alpha \gg \beta$, the ant colony part of the RGPR is turned off. If $\beta \gg \alpha$, the real costs are not considered anymore and we would quickly get stuck in local optima.

6 Results

In this section, we describe the obtained results of the RGPR algorithm. The algorithm was implemented in JAVA and executed on a computer with a 2.7GHz Intel Core Processor and 16 GB of RAM. First, we give a description of the used data and secondly, we evaluate the performance of the algorithm and the influence of different cross-docking policies on the costs.

6.1 Data

6.1.1 Data from literature

As our problem is relatively new in literature, there is only one dataset publicly available that closely resembles our problem structure. This is the dataset of Qu and Bard [2012], which these authors made available on request. These problem instances lack the presence of a heterogeneous fleet, fixed and variable costs and loading or unloading times. However, these factors do not heavily define the problem structure. The most important aspect which makes this dataset suitable for us is that there is no obligated order of picking-up, delivering, loading or unloading operations, just like in our problem description.

Qu and Bard [2012] were only able to solve instances with six requests and two vehicles to proven optimality. As this instance size is generally too small to test the quality of an algorithm, they expended the solved instances to instances with 25 requests in such a way that the optimal solution remained the same. This is done by adding requests along the optimal route, with feasible time windows and loads.

Although this method is a creative way of getting optimal solutions to large instances, there are some drawbacks to the resulting dataset. Firstly, it turns out that it is quite hard to find a feasible solution for these instances. Since the locations of the extra nineteen suppliers and customers and the loads of the associated requests are all derived from the optimal solution of the instance with six requests, it becomes harder to satisfy these requests in a suboptimal route. Secondly, if a solution is feasible, it is likely that the objective value is close to the optimum, which biases the results of solved instances towards good solutions. In their paper, Qu and Bard [2012] report that they could not find a feasible solution for a large part of their instances.

Another drawback of this dataset is that the number of cross-docking operations in the optimal solutions is low. Typically, only one or two of the 25 requests are cross-docked, which makes the dataset unsuitable to evaluate our algorithm that focuses on optimizing cross-docking operations.

There are other datasets in literature with reported optima, such as the dataset of Wen et al. [2009] and Dondo and Cerdá [2013]. However, their problem instances differ from our problem description in two important ways. Firstly, they require that all pick-ups should happen before deliveries. Secondly, they require that each vehicle visits a cross-dock in between and only once. Santos et al. [2013] drop the second requirement, and reports the resulting optima of instances that are based on those of Wen et al. [2009]. Therefore, their dataset is the closest to our problem description after those of Qu and Bard [2012]. However, for some instances, our algorithm did find feasible solutions with lower cost than their reported optima. We thoroughly verified the feasibility and cost of our solutions, and can only conclude that at least some of the reported optima of Santos et al. [2013] are incorrect. This makes their dataset unsuitable for benchmarking our algorithm.

In summary, there are no suitable instances in literature that can be used to compare the results of the RGPR-algorithm with optima. Therefore, we focus on analyzing the performance of our algorithm in relation to the computation time. Also, we investigate how different routing policies influence the cost.

The dataset of Santos et al. [2013] is used for these purposes. The data is generated from a real dataset of Transvision, a Danish logistics company. There are 200 supplier-customers pairs which all require a number of pallets. The suppliers are mostly located in Zealand, the eastern part Denmark. The customers are situated in Jutland, which is the western part of Denmark. Both customers and suppliers have time windows which are generally two hours long. There is one cross-dock, situated in the western part of Jutland, near Copenhagen. All vehicles start at this cross-dock and should be back there at the end of the day.

Santos et al. [2013] extracted part of these supplier-customer pairs, resulting in instances with $10, 15, 20, 25$ and 30 requests with respectively $4, 6, 7, 9$ and 10 homogeneous vehicles. For each size, five different instances are created. We refer to each instance with a code that is the number of requests, followed by a letter from the alphabet. For example, we refer to the third instance of the instances with 25 requests with the code 25c.

6.1.2 Data from ORTEC

The dataset from ORTEC comes from a Chinese distribution company. In order to test improvement to case-related software improvements, they use a test case that is based on the data of this company. Each request has to be picked up at one of the 23 suppliers, and has to be delivered to a final DC. Furthermore, there are three cross-docks which can be used. The geographical distribution of all locations is such that all three cross-docks are situated between the pick-up and delivery location. The company has access to a diverse heterogeneous fleet. It has eight vehicles with a capacity of two tonne, four vehicles with a capacity of five ton and six vehicles with a ten ton capacity. Each of these vehicles starts at one of the 23 pick-up points and should be returned to the same location. Furthermore, there are three fifteen and three thirty ton vehicles, dispersed over the three cross-docks.

In the original case, loads and capacities are three-dimensional. Since our algorithm can only handle one-dimensional loads, the three-dimensional loads are mapped to a one-dimensional load, in such a way that a solution which was provided for this case is still feasible in terms of capacity. Furthermore, different variable costs are associated with each vehicle type. All the precise case data is available on request.

In this case, the distribution company has additional requirements for the vehicle routing process. The small trucks of two, five and ten ton can only be used for local pick-ups. If used, these vehicles have to deliver their requests to a cross-dock. Only the big trucks of fifteen and thirty ton can do deliveries. Additionally, the big trucks can do at most three stops besides the start and end location. Direct delivery by a big truck is a possibility.

Most importantly for our analysis purposes, the company uses static cross-docking, meaning that it has pre-assigned each request to one of the three cross-docks. This implies that each request either has to be handled by a direct delivery, or is handled via its designated cross-dock. Each request is assigned to that cross-dock for which the difference between the travel distance for direct delivery and the travel distance with cross-docking is the smallest. Our algorithm can deal with these constraints by marking all solutions that do not adhere the constraints as infeasible. Section 6.9 analyses the gains from using a dynamic cross-docking policy instead of the described static cross-docking policy.

6.2 Parameter settings

In this section, we describe how the parameter values were chosen. Section 6.2.1 states the parameters that need to be tuned and section 6.2.2 describes the tuning process.

6.2.1 Parameters

The parameters that need to be tuned are the size of the option list n_C^{max} for each greedy insertion, the maximum number of removed requests u_{max} in the

ruin methods, the parameters $\theta_1, \theta_2, \theta_3$ and θ_4 that determine the weight of each component of the relatedness measure, the number of evaluated insertion positions k in the regret-k insertion heuristic, the parameters α_c and α_s that determine the degree of randomness in respectively the cluster removal and the Shaw removal heuristic, the rate of change r in the weight updates of the heuristics and the parameters H, ρ, α and β that shape the function that evaluates the quality of a solution in the ant colony framework.

Furthermore, a trade-off between solution quality and computation time has to be made in choosing the number of improvement iterations per solution $iter_{rr}$, the number of unsuccessful improvement iterations before increasing u, $iter_u$, the number of RGPR iterations $iter_{rg}$, the number of improvement iterations in the path relinking procedure $iter_{pr}$ and the batch size b.

6.2.2 Parameter tuning

We use a two stage strategy to determine the parameter values. First of all, a parameter setting is obtained by an ad-hoc trial-and-error phase. These parameter values are found while developing and testing the heuristic. In the second phase, these parameter values are improved by changing one parameter value, while keeping all the others fixed. We start by changing the value of the first parameter. The heuristic is applied on the instances $10a, 15a, 20a$ and $25a$. For each instance, the average value of the best 10 solutions found by the heuristic under the new parameter value is compared with the average value of the best 10 solutions found by the heuristic under the initial parameter value. The percentual deviation of these values is averaged over the four instances. The parameter value which leads to the best results is chosen.

Given this new value for the first parameter, the second parameter undergoes the same process, and so on until all parameter values are handled. We repeat this process two more times in order to arrive close to a local optimum.

The parameters $iter_{rr}, iter_u, iter_{rg}, iter_{pr}$ and b do not go through the second phase, as these parameter values should be evaluated in terms of solution quality versus computation time.

The parameter tuning results in the parameter vector $(n_C^{max}, u_{max}, \theta_1, \theta_2, \theta_3, \theta_4, \alpha_c, \alpha_s, r, iter_{rr}, iter_u, iter_{rg}, iter_{pr}, b, H, \rho, \alpha, \beta) = (4, 0.25, 2, 5, 3, 1, 5, 4, 0.2, 100, 5, 6, 5, 25, -, -, -, -)$. Furthermore, the solution quality appeared to be insensitive to the parameter values of $\theta_1, \theta_2, \theta_3, \theta_4$ and α_s. No satisfying values were found for the parameters H, ρ, α and β, as further discussed in section 6.4.

6.3 Effectiveness of GRASP

The GRASP construction phase chooses an insertion randomly from the best n_C^{max} insertions. Therefore, it is a hybrid algorithm that positions itself between two extremes. The first extreme is a purely random multi-start procedure, that chooses an insertion randomly out of all possible insertions ($n_C^{max} = \infty$). This algorithm produces bad, but diverse solutions. The second extreme is a greedy algorithm, that chooses always the least-cost insertion ($n_C^{max} = 1$). This

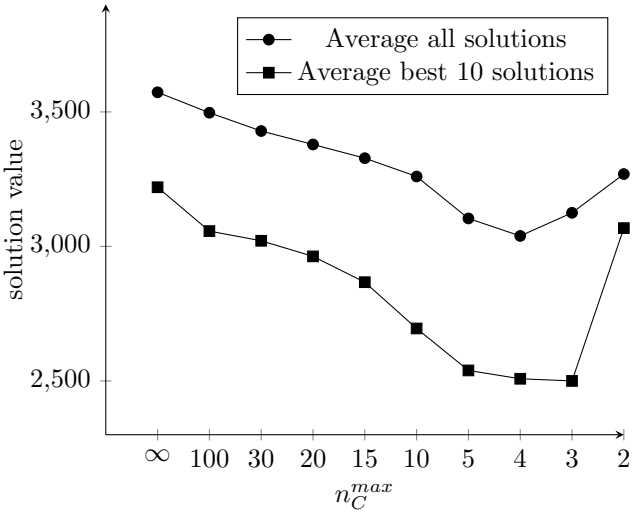

Figure 3: Influence of n_C^{max} on effectiveness GRASP

algorithm produces only one solution, albeit of better quality. In this section, we evaluate the relation between the value of n_C^{max} and the effectiveness of GRASP.

To show the influence of this parameter on GRASP, we run the construction phase 500 times on instance $15a$ for different values of n_C^{max}, ranging from infinity to two. Figure 3 shows the average objective value over these 500 solutions, together with the average objective value of the best 10 solutions for each parameter setting. As expected, the average objective value declines as the construction phase becomes less random. However, the average objective value increases again after $n_C^{max} = 4$, which indicates that for these values the construction phase is not very diverse and gets stuck in local optima. This presumption is strengthened by the observation that the average solution value of the best 10 solutions sharply increases if n_C^{max} is changed from 3 to 2. This also indicates that the degree of randomness becomes too low to avoid local optima. Figure 3 shows that a value of $n_C^{max} = 4$ results in both good average solutions, as well in enough randomness to provide a diverse batch of solutions to the improvement phase.

6.4 Effectiveness of the ant colony framework

The ant colony framework aims to alter the quality evaluation function for each solution in such a way that solutions with a low value are more likely to be created by the construction or improvement algorithm. If the framework is effective, we would see that the average solution value of a solution batch would decrease each RGPR iteration. However, we are not able to find parameter settings for which this is the case, proving that the ant colony framework is not an effective addition to the RGPR algorithm. In this section, we will give a

possible explanation why this is the case.

The main reason for the poor performance of the ant colony framework is that solutions which are very different can have a relatively large amount of the same solution characteristics. Consider for example the routes $A = (1, 2, 3, 4, 5, 6, 7, 8)$, $B = (2, 1, 4, 3, 6, 5, 8, 7)$ and $C = (1, 2, 3, 4, 5, 6, 7, 9)$. Not a single leg of route A is present in route B, but they still share 24 of the 28 solution characteristic variables ξ^1. In contrast, route A and C are very similar, but they share only 21 of the 28 solution characteristic variables, less than A shares with B.

The proposed ant colony framework therefore cannot effectively recognize solution traits. This makes the steering mechanism inadequate. The remaining results in this section are given for parameter values $\alpha = 1$ and $\beta = 0$, effectively disabling the ant colony framework.

6.5 Computation times

6.5.1 Influence of time improvements

In this section, we analyze the reduction in computation time that results from the improvements described in section 5.2.5. In order to gain more insight, a new instance 35a is created by duplicating 5 random requests from instance 30a. Since all the proposed improvements are applied in the construction phase, we compare only the computation times for this phase. For the instances 10a, 15a, 20a, 25a, 30a and 35a the construction phase is run a 100 times, and the average computation time is calculated. Testing showed that the variance of the average computation time is negligible over a 100 runs. This process is repeated, but then the construction phase is executed without enabling the improvements. Figure 4 shows the resulting computation times.

We should note that the results of figure 4 are very much dependent on the specific problem instance. In general, we expect the computational gain factor to be larger if the instance is more tightly constrained. In this case, it is more likely that we can anticipate infeasibility or that we arrive too late at a node, which would result in a reduction in the number of the possible insertion positions that need to be checked. Also, we expect the computational gain factor to be larger for instances with a higher request-to-vehicle ratio. In this case, it is more likely that we can cut down the number of checked possible double insertion combinations. Since our instances are only moderately tight constrained, and since a vehicle handles on average only around three requests, the computational gain factor is relatively small and around 8. We also came across instances with a computational gain of a factor 60.

Since the two lines in figure 4 are more or less parallel, we conclude that the computational gain factor is independent of the instance size. Following the reasoning in the previous paragraph, this makes sense, as the instances do not differ in how tightly constrained they are and in their request-to-vehicle ratio. For these instances, the performance improvements allow us to solve instances with around 10 requests more in the same amount of time.

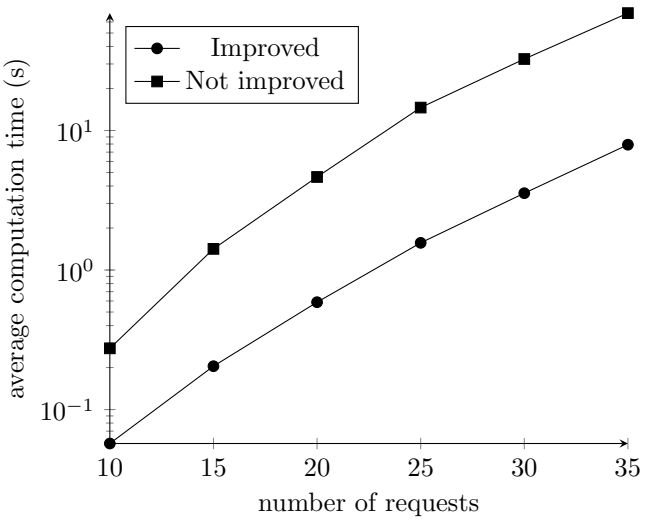

Figure 4: Computation time comparison with and without performance improvement

6.5.2 Time consumption per phase

Table 2 shows the computation time averaged over five instances. Clearly, the time needed for the construction phase is only a fraction of the total time needed. Since the recreate procedure in the improvement phase uses the construction procedure, the time efficiency of this procedure mainly becomes apparent in the computation times of the improvement phase.

Table 2: Average computation time per phase

	Number of requests			
	10	15	20	25
Construction (s)	0.8	4.5	16.0	38.7
Improvement (s)	48.1	142.6	308.6	529.4
Path relinking (s)	13.8	92.3	501.5	1610.2

The time needed for path relinking rapidly increases in the size of the instance. Note that the augmented improvement phase is called every time a new neighbor solution is accepted. If we compare the increase in computation time of the improvement phase with the increase in time of the path relinking phase, we see that both stages need more time due to the increased instance size, which makes the (augmented) improvement phase more complex. However, in the path relinking phase, the number of times that this procedure is called also increases, due to the larger distance between solutions. This leads to the fact that computation times for the path relinking phase increase faster.

6.6 Effectiveness per improvement heuristic

The quality of each improvement heuristic relative to each other can be deduced from the roulette wheel probabilities in equation (25). Due to the way the weights are calculated, if at the end of the algorithm a heuristic has a high probability of being chosen, it means that it was often able to improve the incumbent solution in the improvement phase. Since the resulting probabilities are similar for each instance, table 3 displays these probabilities, averaged over all instances.

Table 3: Effectiveness per improvement heuristic

Heuristic	Probability	Heuristic	Probability
Random removal	0.21	Random insertion	0.28
Shaw removal	0.21	Greedy insertion	0.33
Route removal	0.20	Regret insertion	0.39
Transshipment removal	0.18		
Cluster removal	0.20		

Table 3 shows that the ruin methods perform equally well, while the recreate methods differ in performance. Apparently, it matters more to build a solution in a smart way, than it matters to destroy the solution smartly. We see that the greedy insertion heuristic outperforms the random insertion heuristic, which corresponds with the results in section 6.3. Furthermore, the regret insertion heuristic performs best. Because of the look-ahead feature of this heuristic, it is more likely to produce feasible and low-cost solutions.

6.7 Relative effectiveness of RGPR phases

On average, the improvement phase reduces the average solution value with 12.5%. Furthermore, executing the path relinking procedure after the improvement phase reduces the average solution value with an additional 2.3%. The path relinking procedure is more effective in the later batches than in the early batches. This is due to the fact that the average quality of the elite pool increases each batch. Each solution is paired up with a better elite solution, which increases the quality.

6.8 Comparison routing policies

The routing policy studied in this thesis is more complex than that of the most commonly studied PDPCD's. The main complex feature is that direct deliveries are allowed and that there is no prescribed order of pick-ups, deliveries and cross-docking. In this section, we analyze the gain of incorporating these complex features by comparing best solutions. In the unconstrained version, we solve each instance with the problem structure as defined in section 3. The constrained version requires that each vehicle visits the cross-dock once, and

that all pick-ups happen before the deliveries. Table 4 shows the difference between the best solutions found for each instance.

Table 4: Best solutions for constrained and unconstrained instances

	Instance				
	10a	10b	10c	10d	10e
Constrained	1415	2108	1922	1513	1879
Unconstrained	1415	2063	1902	1513	1879

	Instance				
	15a	15b	15c	15d	15e
Constrained	2421	2931	2561	2478	2851
Unconstrained	2392	2895	2516	2413	2805

	Instance				
	20a	20b	20c	20d	20e
Constrained	3257	3434	3209	3454	2997
Unconstrained	3057	3342	3126	3411	2914

	Instance				
	25a	25b	25c	25d	25e
Constrained	4128	4498	3641	4219	4084
Unconstrained	4034	4311	3545	4107	4001

Three of the five small instances with 10 requests have the same solution under both routing policies. For all other instances, a more liberal routing policy results in a cost reduction. Note that this reduction is larger for the bigger instances. The average gain of the instances with 10 requests is 0.6%, while the average gain of the 25-request instances is 2.84%.

These gains are sizable, especially when we consider the data structure of the instances. The instances originate from a case where the pick-up locations are clustered east from the cross-dock, while the delivery locations are clustered west from the cross-dock. This makes that the optimal solution under the unconstrained routing policy automatically will contain routes that are also feasible under the constrained routing policy. However, even with this data structure, routing costs can be decreased by deviating from the most commonly used constrained routing policy.

6.9 Comparison of static and dynamic cross-docking

A comparison between static and dynamic cross-docking is made by solving the case of the Chinese distribution company. First, we run the algorithm for the case of static cross-docking: solutions in which a request is handled by a non-assigned cross-dock are considered infeasible. Then, we run the algorithm again without this restriction. The algorithm itself determines the best cross-

dock location for each request. When we use dynamic cross-docking instead of static cross-docking, the structure of the solution changes. Only thirteen of the 23 request routes remain unchanged. Furthermore, less vehicles are required. In both solutions, four big trucks are used, but the number of small trucks used declines from six to three. Small trucks can pick their pick-up routes more efficiently since they are not forced to visit certain cross-docks. Most importantly, when using dynamic cross-docking, the costs decrease with 2.7% from 607.7 to 591.0.

This cost decrease is remarkably large when we take a closer look at the problem structure of this distribution company. When each request has a different pick-up and delivery location, cross-docking also has the function of consolidating requests with distant pick-up locations and close delivery locations and vice versa. Each cross-dock then serves its own geographical area. However, since in our case all the requests have the same delivery locations, this functionality of cross-docks becomes less important. This means that the gains of using dynamic cross-docking instead of static cross-docking are expected to be smaller when there is only a single delivery location. Since in this case, there are still gains of 2.7%, we expect that gains in cases with dispersed delivery locations will be even larger.

7 Conclusions

In this thesis, we studied a new complex problem that has its roots in the PDPCD literature. We clearly define this problem and give a mathematical formulation for it, which is our first contribution. Our main contribution is that to solve this problem, we developed an heuristic that combined a constructive GRASP algorithm with a ruin and recreate procedure and path relinking and embedded this in an ant colony optimization framework.

We applied our heuristic to 20 different instances, and found that GRASP, ruin and recreate and path relinking are an effective combination. The ant colony framework was not effective, as it failed to harmonize with the other parts of the algorithm. Furthermore, we showed that sizable gains can be achieved by having a more liberal routing policy than the one most used in literature, even for problem instances that seem to be tailored for those conservative policies.

Using dynamic instead of static cross-docking lowers cost with 2.7%. We deem that to be a conservative estimate of the average savings companies will obtain by applying this dynamic assignment of cross-docks.

For further research, the main step would be to benchmark our heuristic against optimal solutions, or against the performance of other heuristics. Currently, there is no suitable benchmark data available. Such benchmarking would provide more insight in the quality of the heuristic.

To our knowledge, the ant colony optimization method has never been applied to cross-docking problems. We attempted to incorporate ant colony optimization by making use of solution characteristics that fully define a solution. Due to the fact that the heuristic makes abundantly use of insertion opera-

tions, this method fails. This removes the reactive part from our algorithm. Further research could investigate whether more simple solution characteristics that focus on only a few features of the solution will be more effective.

Our heuristic only makes a few assumptions about the structure of problem instances. On the one hand, this makes that our heuristic can be applied to a wide variety of problems. On the other hand, the quality of the heuristic may decrease as it makes use of general methods that cannot make use of many data structures. Therefore, it could be useful to tailor our heuristic for different sets of problem instances, and see if the performance improves.

References

Dwi Agustina, CKM Lee, and Rajesh Piplani. A review: Mathematical modles for cross docking planning. *International Journal of Engineering Business Management*, 2(2):47–54, 2010.

Roberto Baldacci, Eleni Hadjiconstantinou, and Aristide Mingozzi. An exact algorithm for the capacitated vehicle routing problem based on a two-commodity network flow formulation. *Operations research*, 52(5):723–738, 2004.

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

Rodolfo Dondo and Jaime Cerdá. A sweep-heuristic based formulation for the vehicle routing problem with cross-docking. *Computers & Chemical Engineering*, 48:293–311, 2013.

Thomas A Feo and Mauricio GC Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

Fred Glover, Manuel Laguna, and Rafael Martí. Fundamentals of scatter search and path relinking. *Control and cybernetics*, 29(3):653–684, 2000.

Gianfranco Guastaroba, Maria Grazia Speranza, and Daniele Vigo. Intermediate facilities in freight transportation planning: A survey. *Transportation Science, page (to appear)*, 2015.

Bahar Y Kara and Mehmet R Taner. Hub location problems: The location of interacting facilities. In *Foundations of location analysis*, pages 273–288. Springer, 2011.

Jiyin Liu, Chung-Lun Li, and Chun-Yan Chan. Mixed truck delivery systems with both hub-and-spoke and direct shipment. *Transportation Research Part E: Logistics and Transportation Review*, 39(4):325–339, 2003.

Vinicius WC Morais, Geraldo R Mateus, and Thiago F Noronha. Iterated local search heuristics for the vehicle routing problem with cross-docking. *Expert Systems with Applications*, 41(16):7495–7506, 2014.

Viet-Phuong Nguyen, Christian Prins, and Caroline Prodhon. Solving the two-echelon location routing problem by a grasp reinforced by a learning process and path relinking. *European Journal of Operational Research*, 216(1):113–126, 2012.

Yuan Qu and Jonathan F Bard. A grasp with adaptive large neighborhood search for pickup and delivery problems with transshipment. *Computers & Operations Research*, 39(10):2439–2456, 2012.

Mauricio GC Resendel and Celso C Ribeiro. Grasp with path-relinking: Recent advances and applications. In *Metaheuristics: progress as real problem solvers*, pages 29–63. Springer, 2005.

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.

SJ Sadjadi, Mostafa Jafari, and T Amini. A new mathematical modeling and a genetic algorithm search for milk run problem (an auto industry supply chain case study). *The International Journal of Advanced Manufacturing Technology*, 44(1-2):194–200, 2009.

Fernando Afonso Santos, Geraldo Robson Mateus, and Alexandre Salles Da Cunha. A novel column generation algorithm for the vehicle routing problem with cross-docking. In *Network Optimization*, pages 412–425. Springer, 2011.

Fernando Afonso Santos, Geraldo Robson Mateus, and Alexandre Salles Da Cunha. The pickup and delivery problem with cross-docking. *Computers & Operations Research*, 40(4):1085–1093, 2013.

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 SteadieSeifi, Nico P Dellaert, W Nuijten, Tom Van Woensel, and R Raoufi. Multimodal freight transportation planning: A literature review. *European journal of operational research*, 233(1):1–15, 2014.

Christos D Tarantilis. Adaptive multi-restart tabu search algorithm for the vehicle routing problem with cross-docking. *Optimization letters*, 7(7):1583–1596, 2013.

Min Wen, Jesper Larsen, Jens Clausen, Jean-François Cordeau, and Gilbert Laporte. Vehicle routing with cross-docking. *Journal of the Operational Research Society*, 60(12):1708–1718, 2009.