

# Scheduling of container storage and retrieval

Iris F.A. Vis

VU University Amsterdam, Faculty of Economics and Business Administration,

De Boelelaan 1105, Room 3A-31, 1081 HV Amsterdam, The Netherlands

Kees Jan Roodbergen

RSM Erasmus University, P.O. box 1738, 3000 DR Rotterdam, The Netherlands

**Please refer to this article as:**

**Vis, I.F.A., and Roodbergen, K.J. (2009), Scheduling of container storage and retrieval, *Operations Research* 57(2), 456-467.**

Subject classifications: Transportation: freight/materials handling; Transportation: scheduling;

Facilities/equipment planning; scheduling

Area of review: Transportation

## **Abstract**

We consider the problem of scheduling the storage and retrieval of containers in the storage area of a container terminal. Some arcs in the underlying directed network must be visited; other arcs may - but need not be - visited. We can, therefore, consider this problem to be a special case of the directed Rural Postman Problem. We show that this problem can be reformulated as an asymmetric Steiner Traveling Salesman Problem. This reformulation can be efficiently solved to optimality by a combination of optimal assignments in bipartite networks for parts of the problem and dynamic programming for the connections between those parts.

# 1 Introduction

The main purpose of a container terminal is to transfer containers from one mode of transportation to another. Especially sea-going vessels put a large strain on the terminals since they may have to load or unload up to 10,000 containers per visit. In order to deal with these large ships adequately, there is an ever increasing pressure to improve the speed and efficiency of terminal operations. One of the activities at a container terminal is the temporary storage of containers at the stack, where containers wait for further transport.

A typical stack consists of a number of parallel rows where containers can be stored. Often container terminals have two sets of vehicles. One set of vehicles operates exclusively in the stack; the other set of vehicles is used for transporting containers between the stack and other areas. Containers can be delivered to and picked up from both sides of each row in the stack. For example, a container that arrives by sea and leaves at a later date by road is typically delivered to the stack on one side ("seaside") by a transport vehicle and picked up from the other side ("landside") by a truck. In this paper we will focus on the sequencing of storage and retrieval requests within the stack for a single straddle carrier (a general-purpose vehicle for transporting and storing containers). The objective is to route the straddle carrier efficiently through a number of parallel rows to handle all requests.

Our problem is part of a more general class of problems, which consists of planning and scheduling for the transport of unit loads. Unit-load transport means that each load requires its own vehicle for transportation. The Vehicle Routing Problem with Unit Loads (VRPUL) consists of two components: (1) assignment of transport requests to vehicles, and (2) scheduling of the requests for each vehicle. Scheduling per vehicle should be, due to the unit-load principle, such that all assigned requests for the vehicle are executed sequentially. This means that the next load can only be picked up for transport after the previous load has been delivered to its destination. A vehicle's route therefore consists of an alternating sequence of loaded rides to execute transport requests,

and empty rides to make connections to the starting point of the next request. The VRPUL can be seen as a variant of the classical Vehicle Routing Problem.

In the VRPUL, a directed network is required and there is a set of required arcs to traverse (each required arc is the path from a pick-up location to the corresponding drop-off location). Typical examples include internal transport of pallets in a warehouse (De Koster, Le-Anh and Van der Meer, 2004), full truck load movements between cities (Arunapuram, Mathur and Solow, 2003), collection of full skips and delivery of empty skips for waste collection, which is similar to the rollon-rolloff problem (De Meulemeester *et al.*, 1997, and Bodin *et al.*, 2000), and scheduling of automated storage and retrieval systems in warehouses (Van den Berg and Gademann, 1999).

Compared to the existing literature on VRPUL, our problem and solution method are interesting for a number of reasons. First, in our setting of container terminals, an algorithm that solves the scheduling problem for a single vehicle, also solves the complete VRPUL for most practical purposes. This is due to the fact that it is common in practice to dedicate straddle carriers to specific quay cranes working on a ship to unload and load specified parts of a ship. As a result, it is known exactly which storage and retrieval requests need to be handled by that specific straddle carrier (see Kim and Kim, 1999a). Furthermore, the stack is often divided into a number of sub areas, each of which is served by a single straddle carrier, which avoids time-consuming interference among straddle carriers (see Kim *et al.*, 2003). The assignment of straddle carriers to quay cranes and the number of rows assigned to each straddle carrier is typically based on requirements concerning the speed of unloading and loading ships (10,000 containers in 24 hours for a large sea-going vessel).

The second point where our research differs from previous work is that our scheduling problem is relatively complex compared to some of the other instances. Consider for example, the collection of full skips and delivery of empty skips for waste collection. Typically, all full skips must be delivered to a central facility and all empty skips must be retrieved from this location. Thus, all arcs have either their origin or their destination at the central facility. Furthermore, most customers typically

receive an empty skip at the same time when a full skip is collected, which means that these requests do not have to be taken into account for the scheduling. In our problem, origins and destinations may be different for all arcs, i.e. the vehicle does not need to return to the same central facility after every retrieval and before every storage.

Thirdly, our algorithm is capable of finding an optimal solution for the scheduling problem in a container stack in polynomial time. The general VRPUL is NP-hard. Therefore solution methods mainly focus on enumerative procedures, such as branch-and bound (Arunapuram, Mathur and Solow, 2003, and De Meulemeester *et al.*, 1997) and/or heuristics (Bodin *et al.*, 2000, De Koster, Le-Anh and Van der Meer, 2004, and De Meulemeester *et al.*, 1997). Also the subproblem of scheduling a given list of requests for a vehicle, is generally a hard problem. A few efficiently solvable cases for the scheduling problem are known. Kim and Kim (1999b) studied the problem of crane routing in container terminals, which has the same practical background as our problem. An optimal solution to their problem can be found efficiently. However, their assumptions of considering only retrievals and only one side of the stack limit the practical applicability. We present an algorithm that can deal with storage and retrieval requests at both sides of the stack as is common in practice. A solution method that allows for retrieval requests and storage requests on opposite sides is presented by Van den Berg and Gademann (1999) for an automated storage and retrieval system (AS/RS) in warehouses. However, this method works only for a single aisle and one *input point* for retrieval requests and one *output point* for storage requests. Contrary to Van den Berg and Gademann (1999), our approach can be used for multiple rows with both an input and an output point on each side of each row.

In the next section we will show that our problem can be formulated as a special case of the Rural Postman Problem (see e.g. Lawler *et al.* 1985). The aim in the Rural Postman Problem (RPP) is to find a shortest tour that traverses a specified subset of the edges in the network. A well-known special case of the Rural Postman Problem is the Chinese Postman Problem, for which

the subnetwork of required edges is strongly connected. The Chinese Postman Problem (directed or undirected) is solvable in polynomial time (see e.g. Ahuja, Magnanti and Orlin, 1993), whereas the more general Rural Postman Problem is known to be NP-hard (Lenstra and Rinnooy Kan, 1976).

The Rural Postman formulation of our problem can subsequently be transformed into a Steiner Traveling Salesman Problem, which provides a convenient structure to solve the problem efficiently to optimality. The Steiner Traveling Salesman Problem (STSP) looks for a shortest tour such that a given subset of the vertices is visited at least once. The Steiner Traveling Salesman Problem is only solvable in polynomial time for some situations (see e.g. Cornuéjols, Fonlupt and Naddef, 1985). An interesting example of a solvable case of the Steiner Traveling Salesman Problem is the problem of routing order pickers in a warehouse (Ratliff and Rosenthal, 1983). In this problem a shortest tour has to be found to retrieve a number of products from specified locations in a warehouse with a number of parallel aisles (partly comparable with the rows in a container stack). However, this method is only applicable for undirected networks and does not incorporate storage requests.

We develop an algorithm that solves the scheduling problem of container storage and retrieval requests in multiple rows with an input and output point at each end of each row to optimality. The main body of the paper presents the base case algorithm, which solves the problem under a number of restrictions. Relaxations are given in the online supplement to this article. In Section 2 we give a directed network representation of this sequencing problem, which conforms to the definition of the Rural Postman Problem. Next, we show how to transform the Rural Postman formulation into a Steiner Traveling Salesman Problem. We present a special type of dynamic programming algorithm in Section 3 that can construct a shortest tour for a straddle carrier in multiple rows. One of the steps in the dynamic programming algorithm requires prior knowledge on shortest tours within a single row for given start and end points. This subproblem is presented separately in Section 4. It is shown in Section 5 that a solution to the scheduling problem can be found in polynomial time. Sections 6 and 7 explain how the algorithm can be applied and give the results of some numerical

experiments to demonstrate the usefulness of our method. Conclusions are presented in Section 8.

## 2 Network formulation and reformulation

We consider the following situation. A container stack (or a pre-specified subsection of the stack) consists of  $n$  rows with container storage locations and an input/output point ( $I/O$  point) on each side (head and tail) of each row. In total  $\sigma$  containers have to be stored and  $\rho$  containers have to be retrieved from these  $n$  rows. The goal is to minimize the total travel time required to perform  $m$  storage and retrieval requests, where  $m = \sigma + \rho$ . A container that must be stored at a location in row  $i$  ( $1 \leq i \leq n$ ) is available from either the  $I/O$ -point at the head or at the tail of that row, where it has already been deposited by another transport vehicle. A container that must be retrieved, is deposited at either the head or the tail of its row, as specified beforehand. The  $I/O$ -point at the head and tail of a row will be denoted by  $I/O_1$  and  $I/O_2$ . Note that each row has its own  $I/O_1$  and  $I/O_2$ , however, the row number has been suppressed in the notation for ease of reading.

Before a tour of the straddle carrier can be calculated, first all pick-up and drop-off locations for the involved containers must be known. In practice, there is very limited freedom in determining pick-up and drop-off locations. For a retrieval request, the pick-up location is simply the location where that container is currently stored and the drop-off location is the row's  $I/O$ -point at either the sea-side or the land-side of the stack, depending on the container's destination. For a storage request, the pick-up location is the  $I/O$ -point where the container has been delivered, which in turn depends on whether the container arrived by sea or by land. The only locations that do provide some freedom of choice for the terminal's software are the drop-off locations for storage requests. This freedom could potentially be used to reduce travel times for the straddle carrier, but commonly priority is given in practice to quick accessibility of containers for future retrieval and to the support of a fast unloading process for sea-going vessels by using storage locations near the seaside of the stack. Especially the unloading and loading process of sea-going vessels is given priority since the

quality and speed of this process can be directly assessed by and is very costly for the terminal's customers. We therefore assume all pick-up and drop-off locations to be known before invoking our tour optimization algorithm.

In one tour, we cannot have both a retrieval request and a storage request at the same location, because all pick-up and drop-off locations are fixed before optimizing the tour. If we would allow a pick-up and a drop-off to be linked to the same location, then we could get solutions where a container must be deposited at a certain location before that location is actually available (i.e. before the previous container was retrieved). Potentially, efficiency could be improved by making retrieval locations available for storage in the same tour. There are, however, two reasons why the impact of this issue will be very small on overall efficiency. First, even though our scheduling approach is described here as a static process, it would likely be implemented dynamically in practice, by adopting the *block sequencing approach* (see e.g. Han *et al.*, 1987). In this approach a set (i.e. block) of the most urgent storage and retrieval requests is selected from the available requests. The selected requests are then sequenced. When the straddle carrier is done, a new set is again selected, sequenced and executed. Thus, a tour consists of only a limited number of container moves, especially when compared to the total number of (empty) locations. Furthermore, locations that became available due to a retrieval, can be used already in the next tour. Secondly, as long as the utilization of the stack (i.e. the percentage of locations that are occupied by a container) is not too high, there is likely an empty location near any of the retrieval locations. Typically, stacks operated by straddle carriers have an utilization in practice of about 60% (Saanen, 2004).

The tour of a straddle carrier starts and ends at the same, given position. This assumption is made to facilitate the explanation of the algorithm; in Appendix D we show how the problem can also be solved without this restriction. A tour consists of handling several storage and retrieval requests. A straddle carrier travels over a single row of containers to store or retrieve a container. As explained earlier, the transport capacity of a straddle carrier is one container at a time. The

straddle carrier can crossover to another row of containers at the cross aisles positioned at the head and tail of the rows. Figure 1a illustrates a straddle carrier operating in a stack with 6 rows,  $I/O$ -points at both ends of each row, the start and end point of the tour (i.e. depot) and several storage and retrieval requests with respectively their destination or origin that need to be handled.

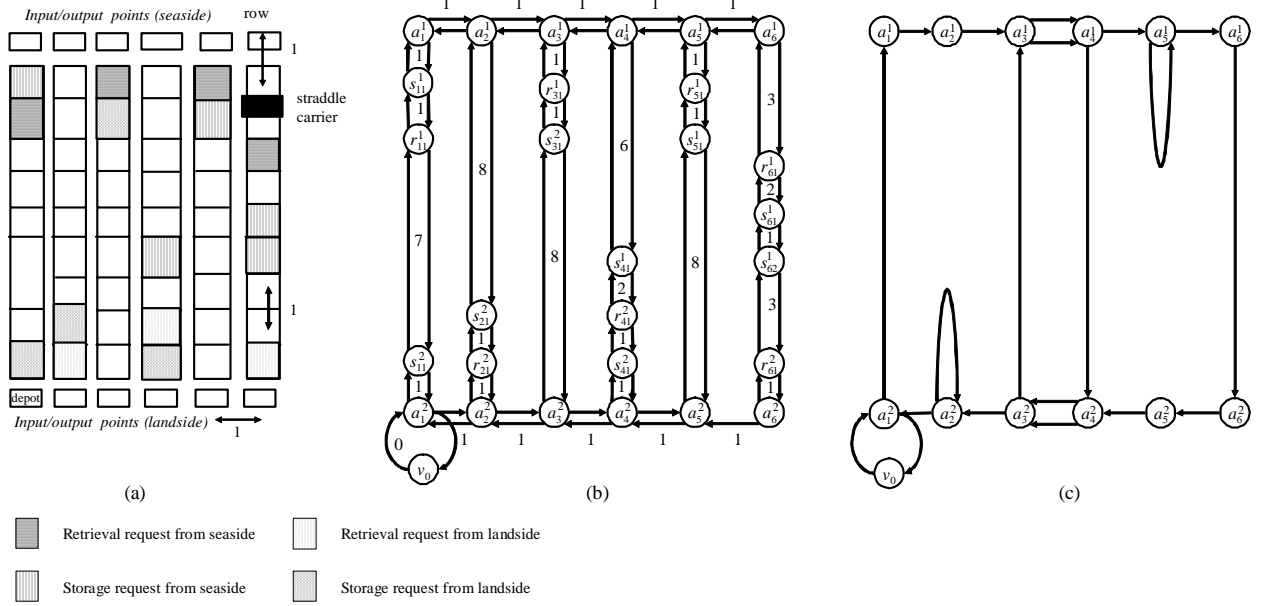


Figure 1: (a) Example of a stack with 6 rows,  $I/O$ -points at both land- and seaside of each row and several storage and retrieval requests. (b) Directed network representation of (a). To keep the figure simple, we incorporated only arcs between two adjacent nodes. (c) Shortest tour resulting from our algorithm.

We use the following notation for each row  $i$  ( $1 \leq i \leq n$ ):

$\sigma_i^1$  number of storages in row  $i$  with origin at  $I/O_1$ .

$\sigma_i^2$  number of storages in row  $i$  with origin at  $I/O_2$ , with  $\sigma = \sum_{i=1}^n \sum_{k=1}^2 \sigma_i^k$

$\rho_i^1$  number of retrievals in row  $i$  with destination at  $I/O_1$ .

$\rho_i^2$  number of retrievals in row  $i$  with destination at  $I/O$ , with  $\rho = \sum_{i=1}^n \sum_{k=1}^2 \rho_i^k$ .

There are no restrictions on the sequence of storage and retrieval requests within a given row.

However, once a row is entered, all requests in that row must be handled before the straddle carrier can continue to the next row. This assumption is quite reasonable in practice due to the large time needed to change rows. Compared to equipment that spans multiple rows (e.g. automated stacking cranes), a straddle carriers needs a relatively large amount of space to switch between rows due to its required width (even though the distance between two rows is only about 2.4 meters, a straddle carrier drives about 14.4 meters due to its radius and positioning requirements). Note that this assumption restricts the number of times rows are entered and left; it does not put a direct restriction on the travel distances between rows, nor does it force a fixed sequence to visit the rows. If beneficial, a row may be initially skipped and only visited at a later stage. In Appendix E we demonstrate how this assumption can be relaxed. With these data a directed network can be constructed. We define the set of nodes  $V$  in the directed network as follows:

$$\begin{aligned}
V &= R^1 \cup R^2 \cup S^1 \cup S^2 \cup V', \\
R^1 &= \{r_{ij}^1, 1 \leq i \leq n \text{ and } 1 \leq j \leq \rho_i^1\}, \\
R^2 &= \{r_{ij}^2, 1 \leq i \leq n \text{ and } 1 \leq j \leq \rho_i^2\}, \\
S^1 &= \{s_{ij}^1, 1 \leq i \leq n \text{ and } 1 \leq j \leq \sigma_i^1\}, \\
S^2 &= \{s_{ij}^2, 1 \leq i \leq n \text{ and } 1 \leq j \leq \sigma_i^2\}, \\
V' &= \{v_0\} \cup \{a_i^k, 1 \leq i \leq n \text{ and } k = 1, 2\}.
\end{aligned}$$

The nodes of  $S^1$  and  $S^2$  correspond to the locations, where containers must be stored. The nodes of  $R^1$  and  $R^2$  correspond to the locations where containers must be picked up. The superscript indicates whether the storages (retrievals) originate from (end at) either input/output point  $I/O_1$  or  $I/O_2$ . Node  $v_0$  corresponds to start point of the tour. Nodes  $a_i^1$  and  $a_i^2$  correspond to the locations of respectively  $I/O_1$  and  $I/O_2$  for each row  $i$ .

Next, we need to define the arcs in the network. For any pair of nodes  $x, y$  in the same row we introduce two arcs:  $(x, y)$  and  $(y, x)$ . Some of these arcs must be traversed, i.e. are required.

First of all we have the required arcs  $(r_{ij}^k, a_i^k)$  for  $1 \leq i \leq n$ ,  $k = 1, 2$  and  $1 \leq j \leq \rho_i^k$ . These arcs are necessary to force the straddle carrier to bring the container from  $r_{ij}^k$  to the appropriate  $I/O$  point after picking it up and before continuing with another request. Similarly, we define the required arcs  $(a_i^k, s_{ij}^k)$  for  $1 \leq i \leq n$ ,  $k = 1, 2$  and  $1 \leq j \leq \sigma_i^k$  to enforce the storage of containers. Furthermore, we introduce an unlimited number of copies of the arcs  $(a_i^k, a_{i+1}^k)$  and  $(a_{i+1}^k, a_i^k)$  for  $i = 1, 2, \dots, n - 1$  to make the connections between the rows. The length of an arc  $(x, y)$  is denoted as  $\bar{d}(x, y)$  and simply equals the physical distance between the nodes  $x$  and  $y$ . Figure 1b gives the directed network representation of the example in Figure 1a. Only directed arcs between two adjacent nodes have been drawn to keep the figure simple. For example, also the arc  $(a_1^1, s_{11}^2)$  exists with distance 9. Figure 2a shows row 1 of Figure 1b, with only the required arcs drawn. That is, the arc from a retrieval to the corresponding  $I/O$ -point, and the arcs from the appropriate  $I/O$ -point to the storages have been drawn. These arcs must be traversed, other arc traversals are optional, but some must be included to obtain a tour. A complete graph, based on the RPP formulation, for only the first row is given in Figure 2b.

We now have a directed network that conforms with the definition of the Rural Postman Problem (RPP). However, the nodes of the network are not freely positioned in Euclidian space, but restricted to a limited number of parallel lines (the rows of the stack). Connections between the rows can only occur at the head and tail of the rows. Furthermore, all required arcs in any row  $i$  either originate or end in  $a_i^k$  for either  $k = 1$  or  $2$ . We can exploit this special structure to solve this problem efficiently. The first step is to transform the current network formulation into a network formulation that classifies as a Steiner Traveling Salesman Problem (STSP). To achieve this, we need to change the RPP network formulation such that we no longer have required arcs, but are still able to obtain a valid solution for the original problem. The network  $G$  for our STSP formulation of the problem has exactly the same set of nodes as our RPP formulation. The nodes  $r_{ij}^k$  and  $s_{ij}^k$  are required nodes; the nodes in  $V'$  may, but do not have to be, visited. All arcs lengths  $d(x, y)$  in

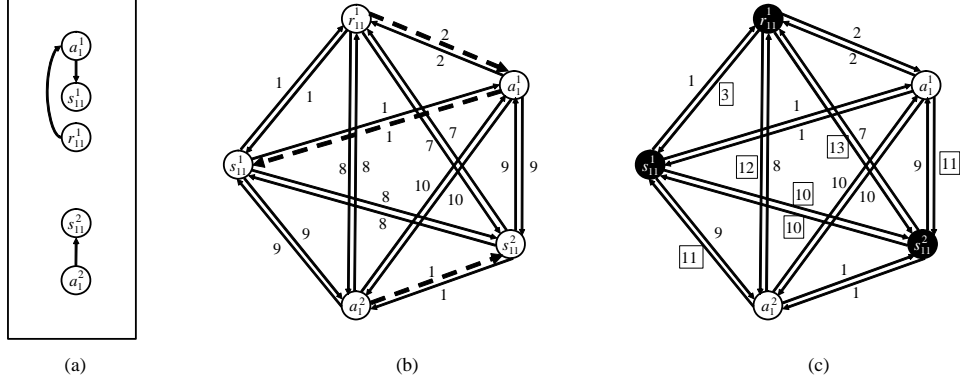


Figure 2: (a) Representation of row 1 of Figure 1b with only the required arcs drawn. (b) Graph of the RPP formulation with dashed lines representing the required arcs. (c) Graph of the reformulation as STSP. Required nodes are solid black. Arcs lengths that differ from their RPP counterparts are highlighted.

$G$  are equal to their RPP counterparts  $\bar{d}(x, y)$ , with the following exceptions:

$$\begin{aligned}
 d(r_{ij}^k, x) &= \bar{d}(r_{ij}^k, a_i^k) + d(a_i^k, x), \quad r_{ij}^k \in R^1 \cup R^2, x \in V \\
 d(x, s_{ij}^k) &= \bar{d}(x, a_i^k) + d(a_i^k, s_{ij}^k), \quad s_{ij}^k \in S^1 \cup S^2, x \in V.
 \end{aligned} \tag{1}$$

For example, the arc length  $d(r_{11}^1, s_{11}^2)$  in row 1 in Figure 1b becomes:  $d(r_{11}^1, s_{11}^2) = \bar{d}(r_{11}^1, a_1^1) + d(a_1^1, s_{11}^2) = \bar{d}(r_{11}^1, a_1^1) + \bar{d}(a_1^1, a_1^2) + d(a_1^2, s_{11}^2) = 2 + 10 + 1 = 13$ . In words, before we can travel from the retrieval location  $r_{11}^1$  to the storage location  $s_{11}^2$ , the straddle carrier first needs to deliver the container from  $r_{11}^1$  to  $I/O_1$ . Thereafter, the straddle carrier needs to travel from  $I/O_1$  to the origin of the storage request,  $I/O_2$ , and then with the container to storage location  $s_{11}^2$ . A graph for the STSP reformulation of the RPP graph in Figure 2b is given in Figure 2c.

It is fairly straightforward to show that any sequence of nodes has exactly the same length in the RPP formulation as in the STSP formulation, provided that each node  $r_{ij}^k$  and  $s_{ij}^k$  ( $1 \leq i \leq n$ ,  $k = 1, 2$ ,  $1 \leq j \leq \rho_i^k$  and  $1 \leq j \leq \sigma_i^k$ ) is visited only once. If any of the required nodes is visited more than once in the STSP, then the resulting tour length will be longer than the equivalent tour

in the RPP formulation. Now, if we can show that required nodes are not repeated in an optimal solution of the STSP formulation, then it is demonstrated that this solution will indeed give an optimal solution for the original RPP formulation as well. Each node in a row is directly connected to every other node in that row in both directions. For any nodes  $x, y, z$ , the triangle inequality  $\bar{d}(x, z) \leq \bar{d}(x, y) + \bar{d}(y, z)$  holds because the arc lengths correspond to physical distances. As a consequence, the triangle inequality also holds for the STSP formulation, since if  $d(x, z) = \bar{d}(x, z) + \epsilon$ , with  $\epsilon > 0$ , then due to equation 1 either  $d(x, y) = \bar{d}(x, y) + \epsilon$  or  $d(y, z) = \bar{d}(y, z) + \epsilon$ .

We now have a network formulation that can serve as an input for our solution method. Shortest tours in the network will be determined by means of dynamic programming. The essence of a dynamic programming method consists of three components: the potential *states*, the possible *transitions* between states, and the *costs* involved in such a transition. By consistently adding arcs (transitions) to partial tours (states), the algorithm gradually builds towards a complete tour. The dynamic programming algorithm is presented in Section 3. There are two types of transitions in the algorithm. The first transition type consists of adding connections between two consecutive rows. The second type of transition consists of adding arcs within a single row. Given the fact that a row may be visited only once (a relaxation of this assumption is given in Appendix E), there are just five useful arc configurations that can lead to a shortest feasible tour. These five arc configurations are, however, not always trivial to obtain. Therefore, we will separately present a method for determining all five arc configurations for any given row in Section 4. An example of an optimal solution can be found in Figure 1c.

### 3 The main algorithm

In this section, we describe a special type of dynamic programming to determine a shortest tour through the network  $G$  of our STSP formulation that visits all required nodes at least once. First of all, it is important to realize the aim of the reformulation as presented in the previous section.

The reformulation as STSP essentially allows us to search for a solution without having to explicitly consider any of the required arcs. Having found an optimal solution for the STSP formulation, however, we have also found an optimal solution for the original formulation, including all required arcs to execute all jobs. Therefore, we can focus on solving the STSP formulation of the problem in this section.

One of the key foundations of dynamic programming is the "principle of optimality", which is the property that pieces of optimal solutions are optimal themselves. As an example of dynamic programming consider an acyclic digraph where nodes  $i, j \in V$  are ordered such that  $i < j$  for all arcs  $(i, j)$  in the graph. Define  $d(v)$  to denote the length of a shortest path from  $s$  to  $v$  and  $c_{iv}$  to denote the length of arc  $(i, v)$ . Then the shortest path from the source  $s$  to the sink  $t$  can be found by applying the recurrence  $d(v) = \min_{i \in V - \{v\}} \{d(i) + c_{iv}\}$  for  $v = 2, 3, \dots, n$ . Or in words, if we know the length of the shortest paths from  $s$  to every predecessor of  $v$ , then we can find the length of the shortest path from  $s$  to  $v$ . By repeatedly applying this relation, we find the shortest path from  $s$  to  $t$  (Wolsey, 1998).

Our problem is quite different from the shortest path problem, which makes it necessary to use a somewhat different methodology and notation. One of the reasons for this is that we need to construct a tour instead of a path. Furthermore, we do not have the fixed ordering of nodes as in the shortest path problem; all nodes can be visited in any sequence. Nevertheless, it is possible to formulate a recursive procedure that rest on the principle of optimality. We will use the common notation and terminology for this class of problems as originally introduced by Ratliff and Rosenthal (1983). The main text in this section will be used to convey the ideas behind the algorithm; theorems and proofs used to derive the results can be found in Appendix B.

We can consider any tour through our network  $G$  as a special kind of subgraph of  $G$ , which we will call a *tour subgraph* (a formal definition is given in Theorem B.1). Even though strictly speaking there is a difference between a tour and a tour subgraph, it is fairly straightforward to

obtain a tour from a given tour subgraph (see Ratliff and Rosenthal, 1983). Therefore, the problem of finding a shortest tour can be solved by finding a shortest tour subgraph. If we divide the network  $G$  into two pieces by drawing a vertical line, we obtain a left part and a right part. The left part of  $G$  will be denoted by  $L_i^+$  if the left part includes all nodes and arcs up to and including those of row  $i$ . We will use  $L_{i+1}^-$  to denote the left part of  $G$  when it includes  $L_i^+$  plus all connecting arcs between row  $i$  and row  $i + 1$ . Furthermore, the left part of a tour subgraph will be referred to as a *partial tour subgraph* and the right part as its *completion*. The principle of optimality tells us that if the entire tour subgraph is optimal, then both the partial tour subgraph as well as the completion must be optimal. We use this knowledge while building towards the final solution.

The algorithm starts with an empty partial tour subgraph containing no arcs. Next, some arcs of row 1 are added to the empty partial tour subgraph. If we consider the rows vertically oriented (see Figure 1) so that we can refer to a “north” and “south” end, then there are just four ways to perform the work in each row: a north-to-south traversal, a south-to-north traversal, a north-to-north traversal and a south-to-south traversal. A fifth possibility is not to enter the row at all, which obviously is only feasible if there is no work to do in the row. These possibilities are graphically depicted in Figure 3 and an algorithm to calculate the related distances (costs) for each of them is presented in Section 4. We now either have one empty partial tour subgraph (if there is no work to be done in row 1) or four partial tour subgraphs. We will refer to partial tour subgraphs that contain only arcs from row 1 as  $L_1^+$  *partial tour subgraphs* (a formal definition is given in Theorem B.2).

The next step is to make the connection to row 2. Intuitively, it is quite clear how these connections should look like. For example, a  $L_1^+$  partial tour subgraph with a south-to-south traversal (see Figure 3, *iv*) in row 1 must be connected by two arcs at the south position. Any other option would either not result in a feasible tour or not be optimal. Partial tour subgraphs containing arcs in row 1 and connecting arcs between row 1 and row 2 will be called  $L_2^-$  partial tour

subgraphs. Costs are counted in each step by simply adding the lengths of the arcs that are added. The next step is to determine all possible  $L_2^+$  partial tour subgraphs by adding arcs in row 2. And so on.

The structure we described so far, could serve as a fairly straightforward branch-and-bound procedure to find an optimal solution. With every addition of arcs from subsequent rows and their connections, the number of partial tour subgraphs increases. The strength of our approach – and the reason why our method classifies as dynamic programming instead of branch-and-bound – is that we can actually limit the number of partial tour subgraphs at each stage. Moreover, we can describe the partial tour subgraphs in such a way that we have a fixed, pre-determined set that can be used at every stage.

As explained before, we divide tour subgraphs into two pieces; the left part is a partial tour subgraph and the right part is its completion. Obviously, at any stage in the algorithm we have the partial tour subgraph, but not the completion. Thus, while executing the algorithm, multiple completions may exist that will result in different tour subgraphs for any given partial tour subgraph. Due to the principle of optimality, we know that for an optimal tour subgraph both the partial tour subgraph and the completion must be optimal. Therefore, if the set of possible completions for one partial tour subgraph is identical to the set of possible completions for another partial tour subgraph, we only need to retain the shortest of the two partial tour subgraphs. The other partial tour subgraph can never result in a shortest tour since the left part is longer and the available set of options for the right part is identical. It now can be seen that partial tour subgraphs are best classified according to their feasible completions.

The feasibility of connecting a completion to a partial tour subgraph mainly depends on the arcs at the right boundary of the partial tour subgraph. For example, if the north side of a row has two incoming arcs, then the completion that connects to the same point must have at least two outgoing arcs to ensure a balance of zero. More arcs in the completion are possible, but for every

additional incoming arc there must also be an extra outgoing arc.

If any completion of one partial tour subgraph is also a completion for the other, then we call these two partial tour subgraphs *equivalent* (a formal definition is given in Theorem B.3). A set of equivalent partial tour subgraphs is referred to as an *equivalence class*. Note that in the algorithm we only need to remember the shortest partial tour subgraph of each equivalence class, because the others can never lead to the optimal solution. Since equivalence is only dependent on the set of completions, we can almost uniquely describe equivalence classes by the possible connections to the right-most nodes of the partial tour subgraph. Furthermore, one extra piece of information must be added to ensure feasible tours (see Theorem B.3). If the partial tour subgraph consists of two unconnected pieces, then any completion must consist of one piece and connect to both the north and south side. If this were not the case, then the final solution may have two unconnected subcycles instead of forming one connected tour. If the partial tour subgraph consist of one piece, then the connection to the completion can be made at the north side, the south side or both as long as the number of incoming and outgoing arcs are balanced.

After obtaining an optimal tour subgraph, we need to translate it back into the original RPP format by reversing what we did in equation 1. Note that this only impacts arcs within rows, since there are no required arcs between rows in the RPP formulation. An example of this reverse translation will be given in Section 4 for one row. Taking all information together and using theorems from Appendix B, we can now define the equivalence classes and the transitions as follows.

## Equivalence classes

The classes of equivalent  $L_i$  partial tour subgraphs can be characterized by the following five features:

(degree parity of  $a_i^1$ , degree parity of  $a_i^2$ , balance of  $a_i^1$ , balance of  $a_i^2$ , connectivity)

Here  $a_i^1$  refers to the right-most northern node of the partial tour subgraph, and  $a_i^2$  refers to the right-most southern node. The *degree parity* of a node in a directed graph equals the sum of the

number of incoming arcs (i.e. *indegree*) and number of outgoing arcs (i.e. *outdegree*) in the node. The *balance* of a node equals the indegree of that node minus the outdegree of that node. The degree parity is denoted by  $U$  if the degree is odd ("uneven"), by  $E$  if the degree is even, and by 0 if the degree is zero. The use of degree parities is almost superfluous. The degree parity follows directly from the value of the balance. If the balance of a node is odd, then the degree of this node is odd. The degree of a node is even if the value of the balance of this node is even. Only if the balance is zero, then the degree parity may either be even or zero. Therefore, the degree parity can and will be omitted from our notation except if the balance of  $a_i^1$  and  $a_i^2$  equals 0. The *connectivity* indicates the number of components in the graph. The connectivity equals 0, 1 or 2 (see Theorem B.2.c).

The number of equivalence classes depends on the number of rows in the problem. From Theorem B.2 it directly follows that the number of incoming arcs in a node  $a_i^k$  ( $i = 1, \dots, n$ ,  $k = 1, 2$ ) is at most  $\lfloor n/2 \rfloor$ . As a result, the balance of  $a_i^1$  and  $a_i^2$  can have the values  $-\lfloor \frac{n}{2} \rfloor, \dots, -1, 0, 1, \dots, \lfloor \frac{n}{2} \rfloor$ . Consequently, there exist  $(n + 1)$  possible values for the balance of  $a_i^1$  and  $a_i^2$ . However, in the proof of Theorem B.2 it is noted that if the balance of  $a_i^1$  or the balance of  $a_i^2$  is unequal to zero, then *balance*  $a_i^1 = -\text{balance } a_i^2$ . Therefore, it can be shown (using the Theorems in Appendix B) that the following equivalence classes should be used in the algorithm:

$$\begin{aligned} & (0, 0, 0), (E, E, 0, 0, 1), (E, E, 0, 0, 2), (E, 0, 0, 0, 1), (0, E, 0, 0, 1) \\ & (k, -k, 1) \text{ for } -\lfloor \frac{n}{2} \rfloor \leq k \leq \lfloor \frac{n}{2} \rfloor. \end{aligned} \tag{2}$$

The equivalence class  $(0, 0, 0)$  is only possible if none of the rows that have already been considered by the algorithm contain a request. The equivalence class  $(0, 0, 1)$  is possible only if none of the rows that still need to be considered by the algorithm contain a request. Note that not all equivalence classes exist in all stages of the algorithm (notably the equivalence classes with a high value of  $k$  only exist in the middle region of the container stack). The number of equivalence classes equals  $6 + 2 * \lfloor \frac{n}{2} \rfloor$ .

From Theorem B.1 it follows that after considering row  $n$  in the algorithm, a minimum length directed multiple row storage and retrieval tour is the shortest of the partial tour subgraphs in the equivalence classes  $(0, 0, 1)$ ,  $(E, E, 0, 0, 1)$ ,  $(E, 0, 0, 0, 1)$  and  $(0, E, 0, 0, 1)$ .

## Transitions

In the algorithm two types of connections (transitions) are made. To expand a  $L_i^-$  partial tour subgraph to a  $L_i^+$  partial tour subgraph, arcs within row  $i$  must be added. To expand a  $L_i^+$  partial tour subgraph to a  $L_{i+1}^-$  partial tour subgraph, arcs must be added to make a proper connection to the next row. By starting with an empty tour subgraph and repeatedly adding arcs within a row or arcs between two rows, we gradually build towards the final solution. Note that at every stage typically several arc configurations can be added to a partial tour subgraph. Furthermore, at any time several  $L_i$  partial tour subgraphs will exist which all are to be expanded at every stage. We will discuss these two types of transitions in more detail.

### *Transition to add arcs within a row*

It is far from trivial to determine adequate arc configurations within a single given row. A separate optimization procedure is required to find a shortest sequence of jobs within a row for any given starting and ending point. At this point, we limit ourselves to the observation that there are just five possible configurations as depicted in Figure 3. This is a direct consequence of our assumption that each row can only be visited once. If more than one visit to a row is allowed, then more configurations are feasible. Appendix E contains the required changes to the algorithm to deal with situations in which multiple visits per row are possible. A polynomial-time algorithm to determine the distance of a directed path for any of the configurations in Figure 3 is presented in Section 4.

In possibilities  $(i)$  and  $(ii)$  the directed path through the row starts at one side of the row and ends in the other one meanwhile executing all requests in the row. This means that the nodes  $a_i^1$  and

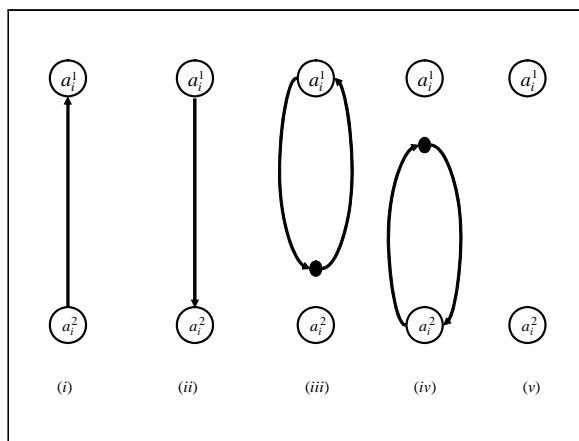


Figure 3: Five ways to visit row  $i$ .

$a_i^2$  may be visited multiple times in such a directed path. Transitions (iii) and (iv) start and end at the same side of the row. However, the other row endnode may still be visited while executing requests in the row. Transition (v) is only allowed if there are no requests in row  $i$ . The sequence in which the requests in a row are handled for a given transition, is determined by the algorithm presented in Section 4. Table 1 indicates the resulting equivalence classes for  $L_i^+$  if we add the transitions from Figure 3 to the  $L_i^-$  minimum length partial tour subgraphs for each equivalence class.

	(i)	(ii)	(iii)	(iv)	(v)
(0,0,0)	(1,-1,1)	(-1,1,1)	(E,0,0,0,1)	(0,E,0,0,1)	(0,0,0)
(0,0,1)	-	-	-	-	(0,0,1)
(E,E,0,0,1)	-	-	-	-	(E,E,0,0,1)
(E,E,0,0,2)	(1,-1,1)	(-1,1,1)	(E,E,0,0,2)	(E,E,0,0,2)	(E,E,0,0,2)
(E,0,0,0,1)	(1,-1,1)	(-1,1,1)	(E,0,0,0,1)	(E,E,0,0,2)	(E,0,0,0,1)
(0,E,0,0,1)	(1,-1,1)	(-1,1,1)	(E,E,0,0,2)	(0,E,0,0,1)	(0,E,0,0,1)
(-k,k,1)	(-k+1,k-1,1)	(-k-1,k+1,1)	(-k,k,1)	(-k,k,1)	(-k,k,1)
(k,-k,1)	(k+1,-k-1,1)	(k-1,-k+1,1)	(k,-k,1)	(k,-k,1)	(k,-k,1)
(-1,1,1)	(E,E,0,0,1)	(-2,2,1)	(-1,1,1)	(-1,1,1)	(-1,1,1)
(1,-1,1)	(2,-2,1)	(E,E,0,0,1)	(1,-1,1)	(1,-1,1)	(1,-1,1)

Table 1: Equivalence classes resulting from performing the transition to add arcs within a row. Note that  $k$  can be any integer value with  $k \in \{-\lfloor \frac{n}{2} \rfloor, \dots, -2, 2, \dots, \lfloor \frac{n}{2} \rfloor\}$ . A dash (-) indicates that no feasible solution can be obtained after performing this transition.

	(0a)	(0b)	(0c)	(0d)	ka	kb
(0,0,0)	(0,0,0)	-	-	-	-	-
(0,0,1)	(0,0,1)	-	-	-	-	-
(E,E,0,0,1)	(0,0,1)	(E,0,0,0,1)	(0,E,0,0,1)	-	-	-
(E,E,0,0,2)	-	-	-	(E,E,0,0,2)	-	-
(E,0,0,0,1)	(0,0,1)	(E,0,0,0,1)	-	-	-	-
(0,E,0,0,1)	(0,0,1)	-	(0,E,0,0,1)	-	-	-
(-k,k,1)	-	-	-	-	-	(-k,k,1)
(k,-k,1)	-	-	-	-	(k,-k,1)	-

Table 2: Equivalence classes  $L_{i+1}^+$  resulting from performing transitions to add arcs between rows  $i$  and  $i + 1$ . Note that  $k$  can be any integer value with  $k \in \{-\lfloor \frac{n}{2} \rfloor, \dots, -1, 1, \dots, \lfloor \frac{n}{2} \rfloor\}$ . A dash (-) indicates that no feasible

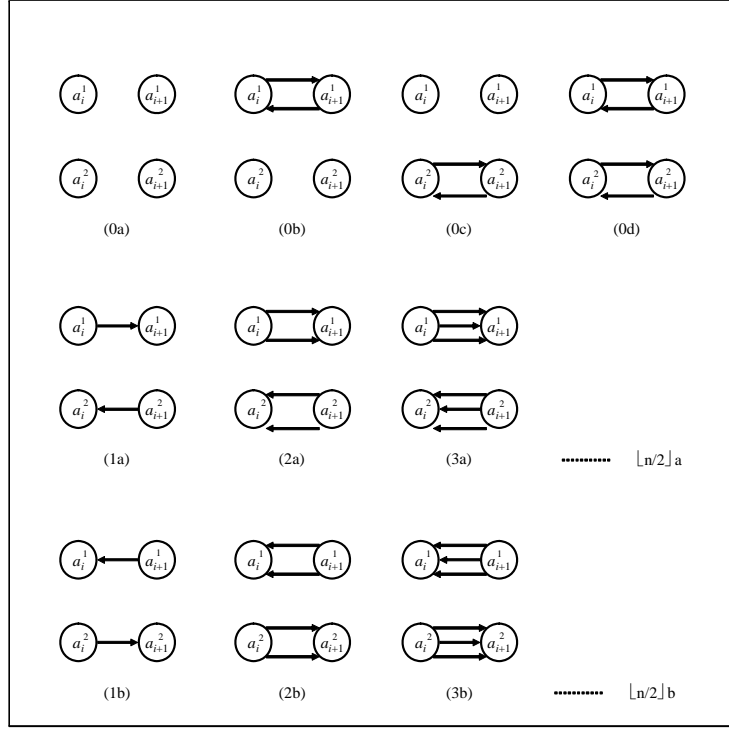


Figure 4:  $4 + 2 * \lfloor \frac{n}{2} \rfloor$  possible transitions to make the crossover from row  $i$  to row  $i + 1$ .

solution is obtained by performing this transition. For values between  $(0d)$  and  $ka$  no feasible solutions exist.

### ***Transition to add arcs between rows***

By adding the transitions given in Figure 4, the crossover between row  $i$  and row  $i + 1$  can be made. Other transitions will never lead to an optimal solution. As explained, the maximum number of incoming arcs for a node  $a_i^k$  equals  $\lfloor \frac{n}{2} \rfloor$ . As a direct consequence, the maximum number of outgoing arcs also equals  $\lfloor \frac{n}{2} \rfloor$ . This results in the transitions  $(1a), \dots, \lfloor \frac{n}{2} \rfloor$  of Figure 4. Furthermore, the transitions in  $(0a)$ ,  $(0b)$ ,  $(0c)$  and  $(0d)$  might exist (see also the proof of Theorem B.2). Consequently, the total number of transitions equals  $4 + 2 * \lfloor \frac{n}{2} \rfloor$ . Table 2 indicates the equivalence classes for  $L_{i+1}^-$  if we add the minimum length configurations from Figure 4 to the  $L_i^+$  minimum length partial tour subgraphs for each equivalence class.

## 4 Storage and retrieval in a single row

In this section, we describe a method to determine the sequence of storage and retrieval requests in a single row  $i$  such that the tour begins and ends at specified  $I/O$  points. This is needed to obtain the precise arc configurations for the transitions that add the arcs within a row (as depicted in Figure 3). The objective is to minimize the travel distances of the straddle carrier in this row for a given start point and end point. We denote the start point of the tour in row  $i$  by  $a_i^s$ , with  $a_i^s \in \{a_i^1, a_i^2\}$  and the end point by  $a_i^e$ , with  $a_i^e \in \{a_i^1, a_i^2\}$ . We give the following algorithm to determine a shortest tour in a single row  $i$  with given start point  $a_i^s$  and end point  $a_i^e$ .

### THEOREM 1

The below mentioned algorithm returns a shortest tour for a single row, for given starting and ending locations, in  $O(m_i \log m_i)$  time, with  $m_i = \rho_i^1 + \rho_i^2 + \sigma_i^1 + \sigma_i^2$ .

### PROOF

See Appendix A.

### Algorithm to determine an optimal tour in a single row

1. For row  $i$  we define a bipartite network  $G_i = (N_1 \cup N_2, A)$  with a copy of each node  $s_{ij}^k$  (for  $k = 1, 2$  and  $1 \leq j \leq \sigma_i^k$ ) and of each node  $r_{ij}^k$  (for  $k = 1, 2$  and  $1 \leq j \leq \rho_i^k$ ) in both  $N_1$  and  $N_2$ . Furthermore, the start point  $a_i^s$  is included in  $N_1$  and the end point  $a_i^e$  is included in  $N_2$ . The set of arcs  $A$  consists of all arcs  $(x, y)$  for  $x \in N_1$  and  $y \in N_2$  and  $x \neq y$ . Arc costs are determined by the arc lengths as defined in the STSP formulation in Section 2.
2. Solve the assignment problem for the bipartite network of step 1. This can be accomplished with an existing method such as the *Hungarian Method* (see e.g. Papadimitriou and Steiglitz, 1982), but a more efficient method is possible (see proof of Theorem 1).
3. Check the solution from step 2 to see if there are any unconnected subtours. If not, then the solution from step 2 gives an optimal solution. However, chances are that the solution con-

tains several unconnected subtours that must be 'patched' (a general description of subtours patching is given in e.g. Burkard *et al.*, 1998, section 4). Some subtours that appear unconnected in this STSP formulation, are actually connected in the original RPP formulation. We need to 'undo' the network reformulation to show this. Remember that we obtained our STSP formulation from the original RPP formulation as follows: any required arc  $(r_{ij}^k, a_i^k)$  in the RPP formulation corresponds to a single node  $r_{ij}^k$  in the STSP formulation. A similar transformation holds for required arcs  $(a_i^k, s_{ij}^k)$ . Arc lengths in both formulations are related according to equation 1. The result of 'undoing' the transformation is that a number of occurrences of the nodes  $a_i^1$  and  $a_i^2$  will appear in the subtours. Any two subcycles resulting from step 2 can easily be merged – without extra costs – if they both contain the same node  $a_i^k$  in the RPP formulation.

4. Check the solution from step 3 to see if there are any unconnected subtours left. If not, then the solution from step 3 gives an optimal solution. If there still are unconnected subtours, then – due to step 3 – there are exactly two subtours (one subtour does not contain  $a_i^1$  and the other subtour does not contain  $a_i^2$ ). In terms of the container terminal, this corresponds to a situation where one subtour contains all request related to  $I/O_1$  and the other subtour contains all requests related to  $I/O_2$ . For any pair of arcs  $(w, x)$  and  $(y, z)$  with  $(w, x)$  in one subtour and  $(y, z)$  in the other subtour, determine  $c_{x,y} = d(w, z) + d(x, y) - d(w, x) - d(y, z)$ , which is the additional distance that needs to be traveled if the two subtours are 'patched' by replacing  $(w, x)$  and  $(y, z)$  by  $(w, z)$  and  $(y, x)$ . Determine  $\min_{x,y} c_{x,y}$  and perform the corresponding arc replacement.

### Example of an optimal tour in a single row

As an illustration, we apply the algorithm to row 1 of Figure 1b with both start and endpoint for the straddle carrier in  $I/O_2$ . Thus,  $a_1^s = a_1^e = a_1^2$ . Figure 5a gives the bipartite network representation as defined in step 1 of the algorithm, based on the STSP graph in Figure 2c. Next, we solve the

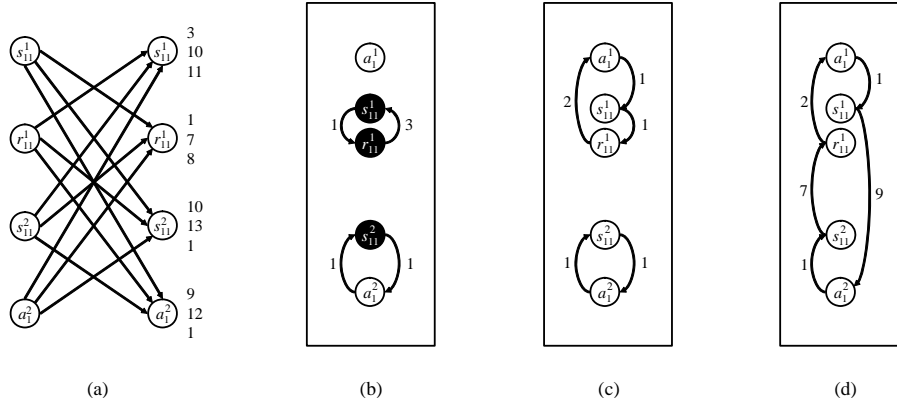


Figure 5: **(a)** Bipartite network  $G_1$  created with step 1 of the single row algorithm based on the STSP graph of Figure 2c. **(b)** Solution of the assignment problem determined in step 2 of the single row algorithm drawn in the STSP graph. **(c)** Translation of the solution back to the RPP formulation. **(d)** Connecting the unconnected subtours according to step 4 of the single row algorithm at minimum costs.

assignment problem for  $G_1$  (see step 2). The subtours resulting from an optimal assignment in  $G_1$  are shown in Figure 5b and equal  $(r_{11}^1, s_{11}^1, r_{11}^1)$  and  $(a_1^2, s_{11}^2, a_1^2)$  with a total travel distance of 6. Note that this solution is given in the STSP formulation. The translation back to the original RPP formulation (see step 3 of the algorithm) is given in Figure 5c. There are two unconnected subtours, which cannot be merged without extra costs, due to the fact that one subtour does not contain  $a_1^1$  and the other subtour does not contain  $a_1^2$ . Therefore, we need to use the procedure from step 4 to patch both subtours, which is depicted Figure 5d. That is, we replace the arcs  $(s_{11}^2, a_1^2)$  and  $(s_{11}^1, r_{11}^1)$  by  $(s_{11}^2, r_{11}^1)$  and  $(s_{11}^1, a_1^2)$  against an extra travel distance of 14. The result  $(a_1^2, s_{11}^2, r_{11}^1, a_1^1, s_{11}^1, a_1^2)$  is a shortest tour for row 1 with starting and ending point  $I/O_2$  and a total travel distance of 20. Note that this is only a shortest tour for row 1 conditional on that fact that the tour starts and ends at  $I/O_2$ . The dynamic programming method, as described in the previous section, will enable us to adequately choose from various ways to traverse the rows. All these ways can be determined similar to this example. In the shortest complete tour for the example of Figure 1b, with all rows being visited, row 1 will eventually be entered at  $a_1^2$  and left at  $a_1^1$  (see Figure 1c).

## 5 Complexity of the algorithm

By applying our dynamic programming algorithm, a shortest directed multiple row storage and retrieval tour can be found in polynomial time. Specifically, the running time is of the order  $O(n^3 + n^2m \log m)$ , where  $n$  equals the number of rows and  $m$  the total number of requests. A proof is given in Appendix C.

## 6 Illustrative example

In this section, we apply our algorithm to the example of Figure 1b. For ease of presentation we will first give the distances that need to be traveled within rows for each of the five possible transitions (as depicted in Figure 3). The algorithm from Section 4 is applied to obtain these distances. Table 3 summarizes the results.

The maximum number of incoming and outgoing arcs in nodes  $a_i^1$  and  $a_i^2$  equals  $\lfloor 6/2 \rfloor = 3$ . Therefore, we have the following equivalence classes (see Equation 2):  $(0, 0, 0)$ ,  $(0, 0, 1)$ ,  $(E, E, 0, 0, 1)$ ,  $(E, E, 0, 0, 2)$ ,  $(E, 0, 0, 0, 1)$ ,  $(0, E, 0, 0, 1)$ ,  $(-3, 3, 1)$ ,  $(-2, 2, 1)$ ,  $(-1, 1, 1)$ ,  $(1, -1, 1)$ ,  $(2, -2, 1)$ ,  $(3, -3, 1)$ .

Tables 1 and 2 can be reformulated for this special example.

transition	row 1	row 2	row 3	row 4	row 5	row 6
$(i)$	12	12	10	26	14	34
$(ii)$	16	14	28	12	12	20
$(iii)$	20	20	20	20	4	30
$(iv)$	20	4	20	20	20	30
$(v)$	-	-	-	-	-	-

Table 3: Travel distances to handle all requests in row  $i$  of the example problem presented in Figure 1b using transition  $i$ ,  $ii$ ,  $iii$ ,  $iv$  and  $v$ .

Equivalence class	row 1		row 2		row 3	
	$L_1^-$	$L_1^+$	$L_2^-$	$L_2^+$	$L_3^-$	$L_3^+$
1 (0,0,0)	-	-	-	-	-	-
2 (0,0,1)	-	-	20/6/0a	-	28/3/0a	-
3 (E,E,0,0,1)	-	-	-	28/10/ii	-	34/7/i
4 (E,E,0,0,2)	-	-	-	26/5/iv	30/4/0d	48/6/iii
5 (E,0,0,0,1)	-	20/-/iii	22/5/0b	42/5/iii	30/3/0b	50/5/iii
6 (0,E,0,0,1)	-	20/-/iv	22/6/0c	26/6/iv	28/6/0c	48/6/iv
7 (-1,1,1)	-	16/-/ii	18/7/1b	22/7/iv	24/7/1b	44/7/iii
8 (-2,2,1)	-	-	-	32/7/ii	36/8/2b	52/7/ii
9 (-3,3,1)	-	-	-	-	-	64/8/ii
10 (1,-1,1)	-	<u>12/-/i</u>	<u>14/10/1a</u>	<u>18/10/iv</u>	<u>20/10/1a</u>	38/6/i
11 (2,-2,1)	-	-	-	26/10/i	30/11/2a	<u>30/10/i</u>
12 (3,-3,1)	-	-	-	-	-	40/11/i

Table 4 (part 1): Solution to the example problem presented in Figure 1b.

Equivalence class	row 4		row 5		row 6	
	$L_4^-$	$L_4^+$	$L_5^-$	$L_5^+$	$L_6^-$	$L_6^+$
1 (0,0,0)	-	-	-	-	-	-
2 (0,0,1)	34/3/0a	-	52/3/0a	-	60/3/0a	-
3 (E,E,0,0,1)	-	52/10/ii	-	60/10/ii	-	<u>74/10/ii</u>
4 (E,E,0,0,2)	52/4/0d	56/5/iv	60/4/0d	58/6/iii	62/4/0d	90/5/iv
5 (E,0,0,0,1)	36/3/0b	56/5/iii	54/3/0b	58/5/iii	60/5/0b	90/5/iii
6 (0,E,0,0,1)	36/3/0c	56/6/iv	54/3/0c	74/6/iv	62/3/0c	92/6/iv
7 (-1,1,1)	46/7/1b	48/5/ii	50/7/1b	54/7/iii	56/7/1b	80/5/ii
8 (-2,2,1)	56/8/2b	58/7/ii	62/8/2b	62/7/ii	66/8/2b	76/7/ii
9 (-3,3,1)	70/9/3b	68/8/ii	74/9/3b	74/8/ii	80/9/3b	86/8/ii
10 (1,-1,1)	40/10/1a	<u>46/11/ii</u>	<u>48/10/1a</u>	<u>52/10/iii</u>	<u>54/10/1a</u>	84/10/iii
11 (2,-2,1)	<u>34/11/2a</u>	54/11/iii	58/11/2a	62/11/iii	66/11/2a	88/10/i
12 (3,-3,1)	46/12/3a	60/11/i	66/12/3a	70/12/iii	76/12/3a	100/11/i

Table 4 (part 2): Solution to the example problem presented in Figure 1b.

In Table 4 we present all information regarding the minimum length partial subtours. The numbers in each cell indicate respectively, the minimum length, the equivalence class of the predecessor and the chosen arc configuration. We have numbered the equivalence classes according to the first column of Table 4. The values for  $L_1^+$  are obtained by adding the mentioned arc configuration to a null graph. We obtain  $L_2^-$  by adding transitions from Table 2. The minimum length among all values corresponding to a specific equivalence class is chosen for each equivalence class.

By adding transitions from Table 1, we obtain  $L_2^+$ . Again we take the minimum value for each equivalence class. For example, the equivalence class  $(E, E, 0, 0, 1)$  in  $L_2^+$  can be obtained in two different ways. First, by performing transition  $(i)$  to  $(-1, 1, 1)$ , which results in a length of  $18 + 12 = 30$ . Secondly, by performing transition  $(ii)$  to  $(1, -1, 1)$  we also obtain equivalence class

$(E, E, 0, 0, 1)$ . The length in this case equals  $14 + 14 = 28$ .

Clearly, the minimum length equals 28. The process is continued in this way for each row. The solution to the problem is indicated by the underscored values. Starting in row 6, we travel backwards through Table 4 to obtain, the arc configurations which should be used. A shortest directed multiple row storage and retrieval tour related to the data from Table 4 is given in Figure 1c. The length of the total travel distance equals 74.

## 7 Numerical experiments

A simulation study has been performed in Borland Delphi to show the effectiveness of our approach compared to the common policy of first-come-first-served (FCFS). We consider a stack as presented in Figure 1a with six rows and 40 storage locations of each 1 TEU (twenty feet equivalent unit, i.e. a container of 6 meters long). The distances to be traveled between two adjacent rows typically equal about 14.4 meters. We apply a block-scheduling approach where a straddle carrier handles between 5 and 10 containers in a single trip. On average, this corresponds to a workload of 20 to 40 minutes. For each container we randomly generate the corresponding row  $i$  ( $1 \leq i \leq 6$ ), the corresponding location  $\ell$  ( $1 \leq \ell \leq 40$ ), the type of request (storage or retrieval) and an  $I/O$ -point.

Multiple experiments have been performed in which we vary the number of requests to be handled and the scheduling policy. Average total travel distances have been calculated for each of the experiments. To obtain a valid estimation of the total travel distances, we need to determine the minimum number of replications required. According to Law and Kelton (2000) an approximation for the minimum number of replications  $i$ , such that the relative error is smaller than  $\gamma$  ( $0 < \gamma < 1$ ) with a probability of  $1 - \alpha$  equals  $i \geq S^2(i)[z_{1-\alpha/2}/\gamma'\overline{X}(i)]^2$ , where  $S^2(i)$  is the sample variance,  $z_{1-\alpha/2}$  the  $1 - \alpha/2$  percentile of the normal distribution,  $\overline{X}(i)$  the sample mean and  $\gamma' = \gamma/(1 + \gamma)$ . To obtain a relative error  $\gamma$  smaller than 2% with a probability of 95%, a replication size of 2500 is sufficient for all the experiments in this paper. The time required to calculate a tour on a desktop

computer is typically less than a second.

Table 5 presents total travel distances for a straddle carrier handling  $m$  ( $5 \leq m \leq 10$ ) requests while either the FCFS policy or the optimal policy is applied. Differences in travel distances between the two policies are also indicated. The results clearly show that the optimal policy strongly outperforms the FCFS rule. If 10 containers need to be handled, the travel distances for the FCFS policy are more than 50% over the optimal travel distances. Or stated in a different way: a straddle carrier can handle about 9 requests with the optimal policy in the same time it takes to handle 6 requests with the FCFS policy.

$M$	FCFS	optimal	difference
5	1626.36	1174.74	38.4%
6	1916.94	1345.86	42.4%
7	2222.82	1524.18	45.8%
8	2517.06	1697.64	48.3%
9	2805.36	1865.76	50.4%
10	3110.22	2027.28	53.4%

Table 5: Total travel distances in meters for a straddle carrier handling 5 to 10 requests when the FCFS policy and the optimal policy are applied.

## 8 Conclusions

In this paper we have studied the problem of sequencing storage and retrieval requests at a container terminal. Each row has two input/output stations located at its head and tail. It is shown that this problem conforms to the definition of a directed Rural Postman Problem. The related directed network can be transformed in such a way that it classifies as an asymmetric Steiner Traveling Salesman Problem. We have developed an algorithm to construct a shortest *directed multiple row storage and retrieval tour* to execute all requests. Optimal sequences in the rows are used as an

input for a dynamic programming algorithm to optimally connect the rows. The total procedure can be executed in polynomial time. With a simulation study we have shown that this optimal approach outperforms the common rule of first-come-first-served.

## References

- AHUJA, R.K., T.L.MAGNANTI, AND J.B. ORLIN. 1993. Network Flows, Theory, Algorithms, and Applications. Prentice Hall, New Jersey.
- ARUNAPURAM, S., K. MATHUR, AND D. SOLOW. 2003. Vehicle routing and scheduling with full truckloads. *Transportation Science*. 37(2), 170-182.
- BODIN, L., A. MINGOZZI, R. BALDACCI, AND M. BALL. 2000. The Rollon-Rolloff vehicle routing problem. *Transportation Science*. 34(3), 271-288.
- BURKARD, R.E., V.G. DEINEKO, R. VAN DAL, J.A.A. VAN DER VEEN, and G.J. WOEGINGER. 1998. Well-solvable special cases of the traveling salesman problem: a survey. *SIAM review*. 40(3), 496-546.
- CORNUÉJOLS, G., J. FONLUPT, and D. NADDEF. 1985. The traveling salesman problem on a graph and some related integer polyhedra. *Mathematical Programming*. 33, 1-27.
- DE KOSTER, M.B.M., T. LE-ANH, and J.R. VAN DER MEER. 2004. Testing and classifying vehicle dispatching rules in three real-world settings. *Journal of Operations Management*. 22(4), 369-386.
- DE KOSTER, R., and E. VAN DER POORT. 1998. Routing orderpickers in a warehouse: a comparison between optimal and heuristic solutions. *IIE Transactions* 30, 469-480.
- DE MEULEMEESTER, L., G. LAPORTE, F.V. LOUVEAUX and F. SEMET. 1997. Optimal sequencing of skip collections and deliveries. *Journal of the Operational Research Society*. 48(1), 57-64.
- GILMORE, P.C., and R.E. GOMORY. 1964. Sequencing a one state-variable machine: a solvable

- case of the traveling salesman problem. *Operations Research*. 12(5), 655-679.
- HAN, M.H., L.F. MCGINNIS, J.S. SHIEH, and J.A. WHITE. 1987. On sequencing retrievals in an automated storage/retrieval system. *IIE Transactions*. 19(1), 56-66.
- KIM, K.H. and K.Y. KIM. 1999a. Routing straddle carriers for the loading operation of containers using a beam search algorithm. *Computers & Industrial Engineering* 36, 109-136.
- KIM, K.H. and K.Y. KIM. 1999b. An optimal routing algorithm for a transfer crane in port container terminals. *Transportation Science*. 33(1), 17-33.
- KIM, K.H., K.M. LEE, and H. HWANG. 2003. Sequencing delivery and receiving operations for yard cranes in port container terminals. *International Journal of Production Economics* 84, 283-292.
- LAW, A.M., and W.D. KELTON. 2000. *Simulation Modeling and Analysis*, third edition. McGraw-Hill, New York.
- LAWLER, E.L., J.K. LENSTRA, A.H.G. RINNOOY KAN, and D.B. SHMOYS. 1985. *The Traveling Salesman Problem, a Guided Tour of Combinatorial Optimization*. John Wiley & Sons, Chichester.
- LENSTRA, J.K., and A.H.G. RINNOOY KAN. 1976. On general routing problems. *Networks*. 6, 273-280.
- PAPADIMITRIOU, C.H., and K. STEIGLITZ. 1982. *Combinatorial Optimization, Algorithms and Complexity*. Prentice-Hall, Inc., Englewood Cliffs.
- RATLIFF, H.D., and A.S. ROSENTHAL. 1983. Orderpicking in a rectangular warehouse: a solvable case of the traveling salesman problem. *Operations Research*. 31(3), 507-521.
- SAANEN, Y.A. 2004. An approach for designing robotized marine container terminals, Ph.D. thesis. Technical University Delft, The Netherlands.
- VAN DEN BERG, J.P., and A.J.R.M. GADEMANN. 1999. Optimal routing in an automated storage/retrieval system with dedicated storage. *IIE Transactions*. 31, 407-415.
- WOLSEY, L.A. 1998. *Integer Programming*. John Wiley & Sons, Inc., New York.

## Online Appendix A: proof of Theorem 1

Define nodes  $A_\lambda$  ( $\lambda = 1, 2, \dots, m_i$ ) as the destination of each required arc in the RPP formulation ( $s_{ij}^k$  for arcs  $(a_i^k, s_{ij}^k)$  and  $a_i^k$  for arcs  $(r_{ij}^k, a_i^k)$ ). Similarly, define  $B_\mu$  ( $\mu = 1, 2, \dots, m_i$ ) as the origin of each required arc in the RPP formulation. Create a cost matrix  $\Gamma$  with

$$\gamma_{\lambda\mu} = d(A_\lambda, B_\mu). \quad (3)$$

This cost matrix is a so-called *permuted Monge matrix*. To see this, we renumber (permute) the rows and columns of  $\Gamma$  such that

$$d(A_\lambda, a_i^1) \leq d(A_{\lambda+1}, a_i^1) \text{ and } d(B_\mu, a_i^1) \leq d(B_{\mu+1}, a_i^1) \quad \forall \lambda, \mu \quad (4)$$

We denote this new matrix by  $C = (c_{\lambda\mu})$ . If  $\pi$  denotes the permutation we performed on the rows and  $\phi$  the permutation on the columns, then  $c_{\lambda\mu} = \gamma_{\pi(\lambda), \phi(\mu)}$ .

That is, we put rows and columns in increasing order of the distance of the corresponding node to  $I/O_1$ . Clearly, we can easily do this because all nodes are positioned on a straight line.

A cost matrix is, by definition, a *Monge matrix* if

$$c_{\lambda_1\mu_1} + c_{\lambda_2\mu_2} \leq c_{\lambda_1\mu_2} + c_{\lambda_2\mu_1} \text{ for } \lambda_1 < \lambda_2 \text{ and } \mu_1 < \mu_2.$$

Now, due to the fact that all nodes are positioned on a straight line and due to equation 4

$$\begin{aligned} c_{\lambda_1\mu_1} + c_{\lambda_2\mu_2} &= d(A_{\lambda_1}, B_{\mu_1}) + d(A_{\lambda_2}, B_{\mu_2}) \\ &= |d(A_{\lambda_1}, a_i^1) - d(B_{\mu_1}, a_i^1)| + |d(A_{\lambda_2}, a_i^1) - d(B_{\mu_2}, a_i^1)| \\ &\leq |d(A_{\lambda_1}, a_i^1) - d(B_{\mu_2}, a_i^1)| + |d(A_{\lambda_2}, a_i^1) - d(B_{\mu_1}, a_i^1)| \\ &= d(A_{\lambda_1}, B_{\mu_2}) + d(A_{\lambda_2}, B_{\mu_1}) \\ &= c_{\lambda_1\mu_2} + c_{\lambda_2\mu_1} \end{aligned}$$

Thus, the permuted version of our matrix is a Monge matrix. It is a known result for Monge matrices that an optimal assignment equals the identity permutation. So all arcs  $(A_\lambda, B_\mu)$  are in the optimal assignment if  $\pi(\lambda) = \phi(\mu)$ . This theory provides a fast method to determine step 2, namely simply sorting the rows and columns, which requires  $O(m_i \log m_i)$  time (see e.g. Burkard *et al.*, 1998).

To prove that our problem can indeed be solved by first solving the assignment problem and consecutive subtour patching, consider the following. A cost matrix  $C$  is called a Gilmore-Gomory matrix if

$$c_{\lambda\mu} = \begin{cases} \int_{A_\lambda}^{B_\mu} h_1(x) dx & \text{if } B_\mu \geq A_\lambda \\ \int_{B_\mu}^{A_\lambda} h_2(x) dx & \text{if } B_\mu < A_\lambda \end{cases}.$$

Since equation 3 defines  $c_{\lambda\mu}$  simply as a distance on a line, this condition is satisfied. It is a known result for Gilmore-Gomory matrices, that an optimal tour can be found by consecutively solving the assignment problem and subtour matching. Since in our situation there are at most two subtours to match, it is just a matter of 'patching' the two with minimum costs. Step 4 exactly identifies the minimum cost patching. This step can actually be performed in  $O(m_i)$  time by using the structure of the Monge matrix (see Gilmore and Gomory, 1964). ■

## Online Appendix B

### THEOREM B.1

A directed subgraph  $T \subset G$  is a directed tour subgraph if and only if:

- (a) all nodes in  $\{v_0\}, R^1, R^2, S^1, S^2$  have positive degree in  $T$ ,
- (b) excluding nodes with zero degree (i.e. nodes  $a_i^k$  which are not visited),  $T$  is a strongly connected directed graph,
- (c) each node in  $T$  has equal out- and indegrees (balance equals 0).

## PROOF

The proof is similar to the proof of Theorem 1 in Ratliff and Rosenthal (1983)

We define  $L_i^-$  as the directed subgraph of  $G$  consisting of the nodes  $a_i^1$  and  $a_i^2$  together with all arcs and nodes to the left of  $a_i^1$  and  $a_i^2$ . Let  $M_i$  be the directed subgraph of  $G$  consisting of nodes  $a_i^1$  and  $a_i^2$  together with all nodes and arcs in  $G$  between  $a_i^1$  and  $a_i^2$ . Then define  $L_i^+$  as  $L_i^+ = L_i^- \cup M_i$ . We use the notation  $L_i$  if a result holds for both  $L_i = L_i^-$  and  $L_i = L_i^+$ . For any directed subgraph  $L_i \subset G$  a subgraph  $T_i \subset L_i$  is called a  $L_i$  *partial directed tour subgraph* if there exists a *completion*  $C_i$  for  $T_i$  ( $C_i \subset G - L_i$ ) such that  $T_i \cup C_i$  is a directed tour subgraph of  $G$

## THEOREM B.2

Necessary and sufficient conditions for  $T_i \subset L_i$  to be a  $L_i$  *partial tour subgraph* are:

- (a) the degree of all nodes  $r_{ij}^k \in L_i$  and of all nodes  $s_{ij}^k \in L_i$  is positive in  $T_i$ ,
- (b) every node in  $T_i$ , except possibly for  $a_i^1$  and  $a_i^2$ , has equal out- and indegrees (i.e. the balance equals zero),
- (c) excluding nodes with zero degree  $T_i$  has either no connected component, a single connected component containing at least one of the nodes  $a_i^1$  and  $a_i^2$  or two connected components with  $a_i^1$  in one component and  $a_i^2$  in the other.

## PROOF

Note: The sum of the balances of all vertices in  $T_i$  is zero. Hence, if condition (b) is satisfied, if one of the nodes  $a_i^1$  or  $a_i^2$  has a balance unequal to zero then both have a balance unequal to zero and *balance*  $a_i^1 = -(\text{balance } a_i^2)$  to ensure that the sum of all balances equals zero.

At most  $n$  visits are made to the  $n$  rows in which a straddle carrier enters from head or tail or from both sides. A row can not be entered multiple times from one side. These  $n$  rows are connected by  $n$  paths through the cross aisles. Therefore, a maximum of  $n$  arcs between any two rows  $i$  and  $i + 1$  exist. These  $n$  arcs between rows  $i$  and  $i + 1$  will be divided in at most  $\lfloor \frac{n}{2} \rfloor$  arcs

from  $a_i^1$  to  $a_{i+1}^1$  and at most  $\lfloor \frac{n}{2} \rfloor$  arcs from  $a_{i+1}^2$  to  $a_i^2$ . Namely, if we allow more than  $\lfloor \frac{n}{2} \rfloor$  arcs at one side, we create superfluous traffic between rows which will, therefore, not lead to an optimal tour. Even though the actual upper bound is  $\lfloor \frac{n}{2} \rfloor$ , we will give the proof for an upperbound of  $n$  for ease of calculations.

To prove sufficiency, we first consider the case where  $\text{balance } a_i^1 = -(\text{balance } a_i^2) \neq 0$  in  $T_i$ . We use induction to the value of the balance to prove that  $T_i \cup C_i$  is a directed tour subgraph.

Firstly, suppose  $\text{balance } a_i^1 = 1$ . Consequently,  $\text{balance } a_i^2 = -1$ . It will be clear that the below mentioned proof also holds if  $\text{balance } a_i^2 = 1$  and  $\text{balance } a_i^1 = -1$ . Let  $C_i$  be the subgraph of  $G$  consisting of all nodes in  $G - L_i$ , incoming arcs and outgoing arcs in each node in  $M_k$  such that each one is visited in a directed path starting and ending in  $a_i^1$  or starting and ending in  $a_i^2$ , for  $k = i, i + 1, \dots, n - 1$  (if  $L_i = L_i^+$ ,  $k = i + 1, \dots, n - 1$ ). Furthermore,  $C_i$  contains incoming and outgoing arcs in each node in  $M_n$  such that each one is visited in a directed path starting in  $a_i^1$  and ending in  $a_i^2$ , and one outgoing arc from  $a_k^1$  to  $a_{k+1}^1$  and one incoming arc from  $a_{k+1}^2$  to  $a_k^2$ ,  $k = i, i + 1, \dots, n - 1$ .  $T_i \cup C_i$  then satisfies the conditions of Theorem B.1.

Next, assume that  $T_i \cup C_i$  for some  $C_i$  is a directed tour subgraph if  $\text{balance } a_i^1 = 2, \dots, n - 1$  and consider the case that  $\text{balance } a_i^1 = n$  and  $\text{balance } a_i^2 = -n$ .  $D_i$  is a subgraph of  $G$  consisting of nodes  $a_i^1$  and  $a_i^2$  and all nodes between  $a_i^1$  and  $a_i^2$ , incoming and outgoing arcs in each node in  $M_i$  such that each one is visited in a directed path starting in  $a_i^1$  and ending in  $a_i^2$  and  $(n - 1)$  outgoing arcs from  $a_i^1$  to  $a_{i+1}^1$  and  $(n - 1)$  incoming arcs from  $a_{i+1}^2$  to  $a_i^2$ .  $T_{i+1} = T_i \cup D_i$ . Consequently,  $\text{balance } a_{i+1}^1 = n - 1$  and  $\text{balance } a_{i+1}^2 = -(n - 1)$ . The proof holds for the case in which the balance equals  $(n - 1)$ . Therefore, there is a completion  $C_{i+1}$  of  $T_{i+1}$ .  $D_i \cup C_{i+1}$  is a completion of  $T_i$ . As a result,  $T_i \cup D_i \cup C_{i+1}$  is a directed tour subgraph if  $\text{balance } a_i^1 = n$  and  $\text{balance } a_i^2 = -n$ . Consequently,  $T_i \cup C_i$  is a directed tour subgraph if  $\text{balance } a_i^1 = -\text{balance } a_i^2 \neq 0$ .

Next, consider the case where both  $a_i^1$  and  $a_i^2$  have a balance equal to zero in  $T_i$ . If both nodes  $a_i^1$  and  $a_i^2$  have a degree of zero in  $T_i$  than there are no nodes in  $L_i$ . If the  $\text{balance } a_i^1 = 0$  and  $\text{degree}$

$a_i^1 > 0$ , than one incoming arc from the left arrives at  $a_i^1$  and one outgoing arc to the left begins at  $a_i^1$ . We first consider the case in which the degree of  $a_i^2$  equals zero. Let  $D_i$  be the subgraph containing  $a_i^1$  and  $a_i^2$  and all nodes between  $a_i^1$  and  $a_i^2$ , incoming and outgoing arcs in each node in  $M_i$  such that each node is visited in a directed path starting in  $a_i^2$  and ending in  $a_i^1$ , and an outgoing arc from  $a_i^1$  to  $a_{i+1}^1$  and an incoming arc from  $a_{i+1}^2$  to  $a_i^2$ . *balance*  $a_{i+1}^1 = 1$  and *balance*  $a_{i+1}^2 = -1$ . Let  $C_{i+1}$  be the same directed subgraph of  $G$  described above for the case in which *balance*  $a_i^1 = 1$  and *balance*  $a_i^2 = -1$ .  $D_i \cup C_{i+1}$  is a completion of  $T_i$ . As a result,  $T_i \cup D_i \cup C_{i+1}$  satisfies the conditions of Theorem B.1.. The same proof holds for the case in which the degree of  $a_i^1$  is zero and one incoming arc from the left arrives at  $a_i^2$  and one outgoing arc to the left begins at  $a_i^2$ . Secondly, we consider the case in which the degrees of  $a_i^1$  and  $a_i^2$  are larger than zero and the balance of both nodes equals zero. Then both nodes have one incoming arc from the left and one outgoing arc to the left.  $D_i$  and  $C_{i+1}$  can be constructed in the same way as in the first case. Consequently,  $T_i \cup C_i$  is a directed tour subgraph if *balance*  $a_i^1 = \text{balance } a_i^2 = 0$ .

The proof for necessity is the same as the proof for necessity in Theorem 2 of Ratliff and Rosenthal (1983). Namely, let, for some directed tour subgraph  $T$ ,  $T_i$  be  $T_i = T - T \cap (G - L_i)$ . Note that, except for possibly  $a_i^1$  and  $a_i^2$ , no arcs in  $T \cap (G - L_i)$  are incident to nodes in  $T_i$ . Therefore, since conditions (a) and (b) must hold in  $T$  according to Theorem B.1, they also hold in  $T_i$ . Excluding nodes with zero degree,  $T$  is strongly connected. Hence,  $T_i$  cannot have a connected component which does not contain either  $a_i^1$  or  $a_i^2$ . ■

### **THEOREM B.3**

Two  $L_i$  partial tour subgraphs  $T_i^1$  and  $T_i^2$  are equivalent if:

- (a) the balance of  $a_i^1$  and  $a_i^2$  is the same for both
- (b) excluding nodes with zero degree, both  $T_i^1$  and  $T_i^2$  have no connected components, both have a single connected component containing at least one of  $a_i^1$  and  $a_i^2$  or both have two connected components with  $a_i^1$  in one component and  $a_i^2$  in the other.

## PROOF

The proof is similar to the proof of Theorem 3 in Ratliff and Rosenthal (1983).

## Online Appendix C: time complexity

To bound the running time of the complete algorithm, we first bound the running time of each of its operations. The running time of an operation can be bounded by the total number of steps associated with this operation. The bound of the running time of the algorithm equals the product of the bound on the number of steps per operation and the bound on the number of operations (see Ahuja, Magnanti and Orlin,1993).

In this algorithm an operation consists of performing transitions in row  $i$  (Figure 3) combined with transitions for the crossover from row  $i$  to  $i + 1$  (Figure 4). The total number of rows equals  $n$  and therefore the total number of operations is bounded by  $n$ .

The worst case of the procedure to sequence requests in a single row occurs if all requests  $m$  are located in this single row. As a result,  $O(m \log m)$  is the upper bound of the time to perform this part of the operation (see Theorem 1). All  $6 + 2 * \lfloor \frac{n}{2} \rfloor$  equivalence classes need to be searched for the smallest value. In each equivalence class a constant number of values needs to be searched. Therefore, this operation has a time complexity of  $O(n)$ . To perform the operation in an row the upper bound of time equals  $O(m \log m + n)$ .

There exist  $4 + 2 * \lfloor \frac{n}{2} \rfloor$  possible ways to make the crossover from row  $i$  to  $i + 1$ . Therefore, the upper bound of this part of the operation equals  $n$ .

In the complete operation all transitions in an row are combined with all transitions between rows. Again,  $n$  equivalence classes need to be searched for a minimum value among a constant number of values. Therefore, the upper bound of an operation equals  $O(n(m \log m + n))$ . As a result, the complexity of the algorithm is  $O(n^2(m \log m + n)) = O(n^3 + n^2m \log m)$ . ■

## Online Appendix D: varying start and endpoint of a tour

If straddle carriers receive their instructions via a wireless computer system, there is no need for them to return to a central location after each tour to pick up new instructions. Therefore it is interesting for at least the larger container terminals to adapt the algorithm such that a new tour can be started where the last activity of the previous tour occurred. A comparable problem was considered for warehouse order picking by De Koster and Van der Poort (1998). They altered the original algorithm of Ratliff and Rosenthal (1983) for "decentralized depositing", i.e. for situations where picked products can be deposited on a conveyor that runs along the front cross aisle. The required alterations to adapt the Ratliff and Rosenthal (1983) algorithm for decentralized depositing in undirected networks are largely the same as those that have to be made to remove the depot location from our algorithm. Therefore we only briefly describe the required steps.

The adapted algorithm would need to allow tours to start at any given *I/O*-point. Furthermore, we allow the algorithm to freely determine the endpoints of the tours, which will further reduce travel times. The combination of these two requirements will allow the vehicle to start the next tour exactly at the point where it finished the previous tour. Clearly, a further improvement would be possible by optimizing the location where the vehicle changes from one tour to the next. However, the advance information needed for this is usually not available, because that would require a perfect prediction of arrival times of ships and of completion times of requests at the terminal.

For each row we need to decide if a tour will end at this row and if so, at which of the two *I/O*-points. In the network of Figure 1b, we change the role of the "depot node"  $v_0$ . The new depot node no longer represent a physical location, but merely serves as virtual node that will allow us to keep working with round trips in the algorithm. We include a directed arc from node  $v_0$  to the (given) startpoint of the tour with costs 0. Since the endpoint needs to be determined by the algorithm, we include a directed arc from each node  $a_i^x$  (for  $i = 1, \dots, n; x = 1, 2$ ) to  $v_0$  with zero cost.

The structure of the dynamic programming algorithm remains largely the same. The transitions within a row remain as presented in Figure 3, and the distances can still be calculated with the algorithm presented in Section 4. After considering the transitions within a row, a new stage must be added to the dynamic programming algorithm that considers the connection to the depot node  $v_0$ . At this stage we have four possible transitions as presented in Figure 6. Clearly, this step is bounded by a constant number of operations which do not depend on  $n$  or  $m$ . As a result, this extra stage will not influence the time-complexity of the algorithm. To ensure the depot node  $v_0$  is visited only once, we need to remember whether this decision has already been made. This can be achieved by including an extra feature to the equivalence classes, namely "number of directed arcs to  $v_0$ " with possible values 0 and 1.

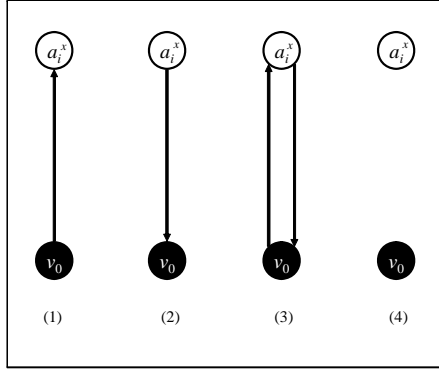


Figure 6: Four transitions to connect the virtual depot  $v_0$  with a head or tail of row  $i$  if starting and ending point of a tour are allowed to be different.

To ensure that in the final solution the balance of each node  $a_i^1$  and  $a_i^2$  equals zero, extra transitions between for  $L_i^+$  to  $L_{i+1}^-$  need to be considered. All of the transitions in Figure 4 are still applicable. However, two copies of each of these transitions must be added, namely a copy with one arc deleted between  $a_i^1$  and  $a_{i+1}^1$ , and a copy with one arc deleted between  $a_i^2$  and  $a_{i+1}^2$ . These extra transitions will be used if the startpoint of the tour is at a different location than the endpoint. As a result, the number of transitions between rows equals  $8 + 6 * \lfloor \frac{n}{2} \rfloor$ . The extra transitions result in

roughly a similar number of extra equivalence classes to be considered.

Concluding, the resulting time-complexity remains similar to the time-complexity of the original situation. The number of operations, which depend on the number of equivalence classes and number of transitions, to be executed within each step of the algorithm increases considerably. However, they are still bounded by the number of rows  $n$ . The complexity of the algorithm itself clearly increases due to the extra number of transitions and equivalence classes that need to be considered.

## Online Appendix E: multiple visits to the same row

In this appendix, we show how the assumption that every row is visited at most once, can be relaxed. Several adaptations need to be made to the algorithm to obtain solutions in which we allow a straddle carrier to enter a row multiple times. The basic structure of the dynamic programming algorithm remains the same as described in the introduction of Section 3. Each row  $i$  is considered sequentially and the procedure ends if row  $n$  has been considered. As a result, the total number of operations in the algorithm is still bounded by  $n$ . We will discuss each of the adaptations in more detail.

First, we study all transitions from  $L_i^-$  to  $L_i^+$ . Compared to Figure 3 several extra transitions are required to enable a row to be entered multiple times. Figure 7 presents the new set of transitions. Rows can now be entirely traversed several times with all traversals occurring in the same direction (transitions  $i'$  and  $ii'$ ). Furthermore, rows can be entered twice from opposite sides (transition  $v'$ ). Other combinations are not needed to find an optimal solution. As an example of an unnecessary transition, consider a transition consisting of two full traversals from  $I/O_2$  to  $I/O_1$  and one traversal from  $I/O_1$  to  $I/O_2$ . This transition can be replaced by a transition consisting one full traversal from  $I/O_2$  to  $I/O_1$  (transition  $i'_a$ ). This follows directly from the fact that any transition actually only defines start and endpoints. Thus, transition  $i'_a$  also covers the possibility of traversing the row up and down several times, provided that the start is at  $I/O_2$  and the end at  $I/O_1$ . Note that transition ( $v'$ ) is only an interesting option if costs are required to patch the two cycles. If patching

can occur without costs, then transition  $(v')$  is equivalent to transition  $(iii')$  or  $(iv')$ .

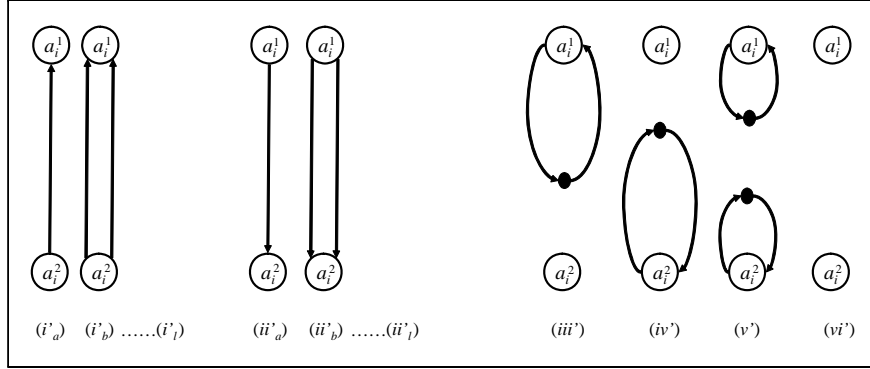


Figure 7:  $4 + 2 * \ell$  transitions to visit row  $i$  where  $\ell = \min(m_i, \lfloor \frac{m}{2} \rfloor)$  if multiple visits to row  $i$  are allowed.

The number of visits to a row is bounded from above by the sum of the number of storage and retrieval requests in a row ( $m_i = \rho_i^1 + \rho_i^2 + \sigma_i^1 + \sigma_i^2$ ). Furthermore, multiple visits to a single row only occur in combination with visits to other rows (otherwise there would not be a need for multiple visits). Therefore the number of visits to any row is also bounded from above by  $\lfloor \frac{m}{2} \rfloor$ . The total number of transitions in row  $i$  therefore is  $4 + 2 * \ell$ , where  $\ell = \min(m_i, \lfloor \frac{m}{2} \rfloor)$ .

Secondly, we study all transitions from  $L_i^+$  to  $L_{i+1}^-$ . Previously, the number of possible transitions between two rows was related to the number of rows  $n$ . In this altered situation, the number of transitions will depend on the number of times a row can be visited. It is known that no more than  $\lfloor \frac{m}{2} \rfloor$  visits to a row will be made. As a direct consequence, the maximum number of outgoing arcs from either  $a_i^1$  or  $a_i^2$  also equals  $\lfloor \frac{m}{2} \rfloor$ . Consequently, the total number of transitions between rows equals  $4 + 2 * \lfloor \frac{m}{2} \rfloor$ . By replacing  $n$  by  $m$  in Figure 4, no new figure needs to be presented.

As a direct result of altering both the transitions within rows and between rows, the equivalence classes used in the algorithm will change. Compared to equation (2)  $k$  can be redefined as  $-\lfloor \frac{m}{2} \rfloor \leq k \leq \lfloor \frac{m}{2} \rfloor$ . Similar to Tables 1 and 2, it would now be possible to derive equivalence classes resulting from performing transitions  $L_i^-$  to  $L_i^+$  and  $L_i^+$  to  $L_{i+1}^-$ . The corresponding costs related to all possible transitions within rows can still be determined with the algorithm presented in Section 4.

For transitions  $(i'_b, \dots, i'_\ell)$  and transitions  $(ii'_b, \dots, ii'_\ell)$  we just need to add an adequate number of copies of the  $I/O$ -points to the bipartite network (step 1 of the algorithm).

The complexity of the algorithm can be derived similar to the proof in Appendix C. The upper bound of sequencing requests in a single row equals  $O(m^2 \log m + m)$ , which depends on the maximum number of equivalence classes  $(6 + 2 * \lfloor \frac{m}{2} \rfloor)$ , the number of transitions in a row, and the upper bound of the related procedure (i.e.  $O(m \log m)$ ). The upper bound for making the crossover from row  $i$  to row  $i + 1$  equals  $m$ . As a result, the complexity of the algorithm equals  $O(nm^2 + nm^3 \log m)$ .