Master Thesis

Scheduling and Sorting Algorithms for Robotic Warehousing Systems

Author(s):
Graf, Daniel

Publication Date:
2015

Permanent Link:
https://doi.org/10.3929/ethz-a-010510176

Rights / License:
In Copyright - Non-Commercial Use Permitted
Scheduling and Sorting Algorithms for Robotic Warehousing Systems

Master Thesis
Daniel Graf
July 18, 2015

Advisors: Prof. Dr. P. Widmayer and K. Böhmová
Department of Computer Science, ETH Zürich
Abstract

Automated storage and retrieval systems are used in many warehouses for time and space efficient stocking and order fulfilment. In this thesis, we study such automated systems in the context of storing bicycles at a train station, where commuters bring their bicycles in the morning and reclaim them in the evening. The commuters place their bicycles in a box at a door, where a robot then moves it underground for safe storage.

In the first part of the thesis, we investigate how the robot best assigns boxes to the underground storage slots. The goal is to find a schedule which allows the robot to meet all the customers at the door without causing any waiting time. For a single door, we devise an efficient algorithm that finds such an optimal schedule under certain conditions. If there is a second door, or arriving and departing customers are heavily interleaved, we show that it is \( \mathcal{NP} \)-complete for the robot to find a wait-free schedule even if it knows all the requests in advance.

In the second part of the thesis, we focus on the task of rearranging the stored boxes in between customer interactions. We phrase the task as a compelling physical sorting problem: on every vertex of a graph \( G \), we place a box. The \( n \) vertices and \( n \) boxes are each numbered from 1 to \( n \), and initially shuffled according to a permutation \( \pi \). A single robot is given the task to sort these boxes. In every step, the robot can walk along an edge of the graph and can carry at most one box at a time. At a vertex, it may swap the box placed there with the box it is carrying. How many steps does the robot need to sort all the boxes? We present efficient algorithms that construct such a shortest sorting walk if \( G \) is a path, a tree or a cycle, and we show that the problem is \( \mathcal{NP} \)-complete for planar graphs.
Acknowledgements

First and foremost, I would like to express my gratitude to Kateřina Böhmová and Peter Widmayer for advising me during this thesis. I enjoyed the interesting and helpful discussions in our weekly meetings immensely. They encouraged me to keep exploring this fascinating topic throughout these months by always asking the right questions and contributing countless insights.

I would like to thank Armin Wyttenbach of Bikeloft GmbH and Matthias Wilhelm of Nussbaum Technologies GmbH for taking the time to explain their exciting automated bicycle storage systems to me. Special thanks go to Filip Pavetić for pointing me to the Croatian task Kutije and to Sandro Feuz for always listening to and dissecting my half-baked ideas and proofs.

I also got valuable comments from those who read preliminary parts of this thesis. I would like to thank Helene Wolleb, Dennis Komm, Claudia Graf and Stefan Willi.

I am grateful to my parents who supported me throughout my studies.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Overview</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Contributions</td>
<td>3</td>
</tr>
<tr>
<td>2 Request Handling</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Storage System Model</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Arrival Only – the Early Morning Scenario</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 The Always-Return-to-the-Door Robot</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 The Direct Robot</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3 The Box-Swapping Robot</td>
<td>12</td>
</tr>
<tr>
<td>2.2.4 Online Request Arrival</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Arrival Only with Two Doors</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1 Fixed Door Assignment</td>
<td>16</td>
</tr>
<tr>
<td>2.3.2 Hardness of Flexible Door Assignment</td>
<td>17</td>
</tr>
<tr>
<td>2.3.3 Approximation of Flexible Door Assignment</td>
<td>21</td>
</tr>
<tr>
<td>2.4 Arrivals and Departures - the Entire Day</td>
<td>30</td>
</tr>
<tr>
<td>2.4.1 Lunch Break with a Full Warehouse</td>
<td>31</td>
</tr>
<tr>
<td>2.4.2 Single Customer per Slot</td>
<td>32</td>
</tr>
<tr>
<td>2.4.3 Hardness for Reusable Slots</td>
<td>34</td>
</tr>
<tr>
<td>2.5 Related Work</td>
<td>37</td>
</tr>
<tr>
<td>2.5.1 Similar Problems on Finding Permutations</td>
<td>38</td>
</tr>
<tr>
<td>2.5.2 Interval Graph Colouring</td>
<td>39</td>
</tr>
<tr>
<td>3 Rearrangement</td>
<td>41</td>
</tr>
<tr>
<td>3.1 Problem Description and Notation</td>
<td>42</td>
</tr>
<tr>
<td>3.2 Minimizing the Travel Time</td>
<td>44</td>
</tr>
<tr>
<td>3.2.1 General Bounds</td>
<td>44</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Dozens or even hundreds of bicycles chaotically parked around train stations is a common phenomenon. Daily commuters on their way to catch a train want to get rid of their two-wheeler as quickly as possible. Some store it in bike lockers but many also just leave it along or on the pedestrian walkway, where they are vulnerable to theft and vandalism, and might obstruct passageways.

In the last few years, various automated bicycle storage systems were designed and developed for this reason. Customers hand their bikes over to a robotic system that stores them at a safe location. The robot is in charge of managing the available storage space and ensures that no one else but the owner can reclaim a bicycle. Once the customer returns to the locker, the robot retrieves the right bicycle as quickly as possible.

One such system is called Bike Loft and was developed by Armin Wyttbach and tested as a prototype in Winterthur in 2014 [60]. In this prototype, a single robot drives along a linear track. Each bicycle is stored in a separate cuboid box so that commuters can also store helmets or rain jackets in the box alongside their bicycle. These boxes are then stored along this track in a two-dimensional grid on both the left and right side of the track. A door is located at one end of the track where customers can interact with the robot. Whenever a customer arrives with his bicycle, the robot brings an empty box to this door. After the customer stores his belongings inside the box, the door closes and the robot carries the box to one of the empty storage slots along the track. When the customer reclaims his bicycle later on, the robot fetches his box from the corresponding storage slot and brings it to the door. An illustration of this system is shown in Figure 1.1.
This thesis aims at the algorithmic scheduling challenges of such a robotic warehousing system. Where should the robot store a filled box? Should it store the box close to the door because it knows that the next customer arrives only twenty seconds later? Or more to the back because it predicts that this box will only be reclaimed late at night? These questions will lead us to a variety of different challenges. If the robot stores a box close to the door, it does not need to move much and is ready quickly to serve the next customer. So why not always use the free spot that is closest to the door? Well, it might be that these close-by slots are needed for customers arriving during rush hour later on and all other boxes should stay out of the way. Another approach might be to quickly drop the box close to the door first and then later during a less busy time of the day pick it up again and put it to some other place along the track. But how should we decide where the box should go and what is the most efficient way to rearrange the boxes?

These scheduling and sorting questions are the problems we will investigate in this thesis. In particular, we look at how such a robot can serve the customers without letting them wait for their bicycles and how it can efficiently rearrange the stored bicycles internally.

All of the models that we consider immediately abstract from the fact that we are dealing with bicycles. Therefore many of our results can be applied to any automated storage system, no matter whether it organizes spare parts in a factory, books in a library or cars in a parking garage.
1.2 Overview

We focus on two problems that arise from modelling a robotic warehousing system. The first problem, which we discuss in Chapter 2, is to make sure that the robot can serve the customers without letting them wait. We show that the difficulty of this scheduling problem heavily depends on the way we model the robot, the customers and the warehouse. In some cases it is efficiently solvable, in others it will be $\mathcal{NP}$-hard to decide whether we can serve all customers without letting any of them wait.

The second problem deals with rearranging the items in storage. We assume that there is a period of time during which no customer arrives and the robot wants to rearrange the bicycles according to some given permutation. This physical sorting problem is outlined and discussed in Chapter 3. We show that we can efficiently find the shortest way for the robot to sort the stored boxes as long as the track system is laid out as a path, tree or cycle but that it is $\mathcal{NP}$-hard for general graphs.

In Chapter 4, we take a look at other approaches for automated bicycle parking systems and how their abstract models might look like. Finally, we summarize our results and point to future work in Chapter 5.

1.3 Contributions

Here, we give a succinct overview of our results as a quick reference for the reader. A detailed summary of this thesis is given in Section 5.1.

**Request Handling Problem:** Section 2.1 introduces our model of a robot that drives along a line consisting of \( n \) boxes and a single, customer-facing door at one end. The robot can carry at most one box at a time and knows all customer requests up front. For all but the last result, we assume that customers only arrive to fill their boxes but never reclaim them.

- We start with Theorem 2.5, where we show that it is $\mathcal{NP}$-hard to decide whether a wait-free schedule exists if the robot always has to return to the door between storing one box and fetching the next box. We see this using a reduction from the task of finding a permutation that has to satisfy given upper bounds on the sums of consecutive entries.

- More importantly however, we show in Theorem 2.11 that if the robot can go directly from one box to another, we can find a feasible schedule efficiently. The same efficient algorithm can also be applied if the robot can instantaneously swap a box with the one currently loaded.

- Additionally, Lemmas 2.13 to 2.16 extend these efficient algorithms to an online setting where the robot only knows a few requests in advance.
1. Introduction

- Surprisingly, once we add a second door, finding a feasible schedule that assigns each customer to a door and a storage slot is \( \mathcal{NP} \)-complete. We show this in Theorem 2.23 using a non-trivial reduction from 3SAT.

- Finally, if we allow the customers to also reclaim their goods and interleave arriving and departing customers, we can use the same slot for multiple customers. As we show in Theorem 2.59, this makes finding a feasible schedule \( \mathcal{NP} \)-complete even with a single door (using a reduction from circular arc colouring).

Rearrangement Problem: Given a graph \( G \) with \( n \) labelled boxes that are shuffled on the \( n \) vertices of \( G \), a robot should sort them as quickly as possible. At any vertex the robot can carry either the box at this vertex or the one it was carrying before to one of the neighbouring vertices. We want to sort the boxes while travelling along as few edges as possible.

- As a first case we let \( G \) be a path graph and let the robot start at its border. In Theorem 3.7, we devise an algorithm to construct a shortest sorting walk using induction on the cycles of the permutation. The modification in Theorem 3.10 even allows us to use only a constant amount of memory and learn the shuffling of the boxes online as we walk around.

- Most interestingly, we can extend this to trees in Theorem 3.15. On trees, we use an auxiliary data structure that we call cycle anchor tree, which tracks the cost of connecting the sorting process of different cycles of the permutation. After applying Edmond’s algorithm for optimum branchings, we can efficiently construct the shortest sorting walk on any tree. This is probably the nicest result of the thesis.

- We can also find the shortest sorting walk on a circle by applying the previous results after a careful case distinction (Theorem 3.35).

- In contrast, we show in Theorem 3.36 that for general graphs and even planar graphs it is \( \mathcal{NP} \)-complete to find the shortest sorting walk (using a reduction from the Hamiltonian circuit problem for grid graphs).

If we count the number of times that the robot picks up a box and minimize that as a first priority before minimizing the number of steps, we can prove:

- On one hand, Theorem 3.40 shows that on path graphs we can still efficiently find such a swap-optimal sorting walk.

- Astonishingly, it is already \( \mathcal{NP} \)-complete for trees, which we show in Theorem 3.42 using a reduction from 3SAT to a problem similar to the Class Steiner tree problem.
Chapter 2

Request Handling

An important objective of an automated bicycle storage system is to minimize the waiting time of the customers. People at a train station are often in a hurry and want to drop off and reclaim their bicycle as quickly as possible. So ideally, the robot should always be at the door and ready whenever a customer arrives.

In this chapter, we focus on this goal of minimizing the time that the customers need to wait. We study this goal under different assumptions. For instance, we significantly simplify the problem first by focusing on a period of time, typically in the early morning, where bicycles arrive but are not being reclaimed. We model a warehouse where there is only a single door for the customers to interact with the robot. Additionally, we assume that the robot does not reshuffle the bicycles that are in storage, as this is the topic of Chapter 3. We devise an efficient algorithm for this early morning setting in Section 2.2.

In Section 2.3, we add a second door to the warehouse and in Section 2.4 we consider a period of time where customers also reclaim their bicycles. We show how both of these changes confront us with \( \mathcal{NP} \)-complete problems.

2.1 Storage System Model

Let us begin by describing the abstract model of a robotic warehousing system that we will use and refine throughout this thesis. An illustration of the model is given in Figure 2.1.

**Layout** The Bike Loft system that we showed in Figure 1.1 featured two grids of storage slots. In our model, we simplify this to a one-dimensional layout of the slots. Our storage systems consists of \( n \) slots that can each store a single box. These \( n \) slots are spread along a line and are at unit distance
2. Request Handling

Figure 2.1: A linear warehouse with $n$ storage slots and $n$ boxes. Four of the boxes shown already contain a bicycle. Three boxes are still empty. There is no box at slot 2 as this box is currently on the robot at the door so that a customer can store his bicycle. Once the box is filled, the robot can take it back to slot 2. The robot might then fetch another empty box and bring it to the door in order to be ready for the next customer.

from each other. Initially, every slot contains an empty box. All boxes are identical and all customers are able to fit their goods into any of the boxes.

Door To the left of the storage slots, there is a single door. The customers will arrive at this door, where the robot brings a box to the customer for loading and unloading.

Robot The robot is modelled as a carriage that can slide along the line and can carry a single box at a time. The robot is initially located at the door. We assume that it has unit velocity, so it takes for instance five seconds to go from the door to slot 5 and it takes six seconds to get from the door to slot 3 and back. We ignore that a real robot would need to accelerate and decelerate. Based on how the robot can pick up and put down a box, we will distinguish two different robot models later. The first robot will have to find an empty slot to put down the box that it is carrying before it can pick up a new box. The second robot will be able to swap the box that it is carrying with the box stored at its current position.

Customers There are $m$ customers arriving during the course of a day. Every customer has an arrival time $t_{ai}$, where they want to fill their box, and a departure time $t_{di}$, where they want to reclaim the item stored previously. All these request times are known to the robot in advance, so that he can optimize his schedule for the entire day.

No handling time As we focus on the driving time of the robot, we assume that all interactions at the door are instantaneous and that the robot can immediately drop off or load a box once it arrived at its desired location.

No reordering Throughout this chapter, we assume that the robot will not rearrange the boxes once they are placed in storage. This means that the robot picks a slot for every box as soon as the box is placed into stock and then never moves this box until it is reclaimed by the customer.
2.2 Arrival Only – the Early Morning Scenario

We first study the setting where customers are only storing items but never retrieving them. In the context of a bicycle storage system, this roughly corresponds to a period in the early morning in a commuter town where people only arrive to store their bicycles on their way to work and will not return to claim them until the evening.

2.2.1 The Always-Return-to-the-Door Robot

First, we consider a simple scheduling routine. We require the robot to always return to its initial position at the door in between different requests. We refer to this robot model as \textsc{Return}.

Let \( n = m \), thus there are as many customers as there are storage slots in the warehouse. As we are not allowing stored items to be rearranged, the schedule of the robot can be fully described by a permutation \( \pi \in S_n \) that assigns each customer its unique storage slot.

If customer \( i \) gets assigned to slot \( \pi(i) \), the \textsc{Return} robot needs \( 2\pi(i) \) seconds immediately before \( t_a^i \) to fetch the empty box from slot \( \pi(i) \) and then takes again \( 2\pi(i) \) seconds immediately after \( t_a^i \) to bring the now filled box to this slot and return back to the door.

Figure 2.2 illustrates the movement of the robot in an example. We see that the robot can serve all the requests without delays as long as the \( n \) intervals of length \( 4\pi(i) \), each centred around \( t_a^i \), do not overlap.

Definition 2.1 (Inter request times) We denote the time periods between consecutive requests as a multiset \( \Delta = \{\Delta_0, \Delta_1, \ldots, \Delta_n\} \) where \( \Delta_i = t_{i+1}^a - t_{i}^a \) for \( 1 \leq i < n \) and \( \Delta_0 = \Delta_n = \infty \).

Finding a permutation that allows a wait-free robot scheduling can now be phrased as the following decision problem. We have to ensure that the time period \( \Delta_i \) between requests \( i \) and \( i + 1 \) is long enough to first store the full box in \( 2\pi(i) \) seconds and then fetch the next empty box in \( 2\pi(i+1) \) seconds.

Definition 2.2 (Always-Return-To-Door Arrival-Only problem) (which we abbreviate as \textsc{ReturnArrival}) Given a multiset of integers \( \Delta = \{\Delta_0, \Delta_1, \ldots, \Delta_n\} \), the question is to decide whether there is a feasible schedule for the \textsc{Return} robot, namely a permutation \( \pi \in S_n \) such that \( 2(\pi(i) + \pi(i+1)) \leq \Delta_i \) for all \( 1 \leq i < n \).

Definition 2.3 (Decision problems on permutations) Given a multiset of integers \( X = \{x_1, \ldots, x_n\} \), we study the following problems:

**SUM:** Is there \( \pi \in S_n \) such that \( \pi(i) + \pi(i + 1) \leq x_i \) for all \( 1 \leq i < n \)?

**TightSUM:** Is there \( \pi \in S_n \) such that \( \pi(i) + \pi(i + 1) = x_i \) for all \( 1 \leq i < n \)?

**Constrained:** Is there \( \pi \in S_n \) such that \( \pi(i) \leq x_i \) for all \( 1 \leq i \leq n \)?
2. Request Handling

Figure 2.2: A small example for the Return robot. Three customers arrive at times \( t_1 = 10 \), \( t_2 = 18 \) and \( t_3 = 25 \) which gives \( \Delta_1 = 8 \) and \( \Delta_2 = 7 \). The only feasible way to assign these customers to storage slots is \( \pi(1) = 3 \), \( \pi(2) = 1 \) and \( \pi(3) = 2 \) so that the robot is always at the door on time and even has a short break from time 20 to 21. The intervals on top denote the time frames needed to handle each request, namely to fetch the box before the customer arrives and to store it afterwards. The plot in the middle tracks the movement of the robot and the cutouts at the bottom show the state of the warehouse at three specific points in time.

Lemma 2.4 ReturnArrival is at least as hard as SUM.

Proof Any instance of the SUM problem can easily be transformed into an equivalent instance of the ReturnArrival problem by setting \( \Delta_i = 2x_i \). □

Theorem 2.5 \[18\] The ReturnArrival problem is strongly \( \mathcal{NP} \)-complete.

Proof Sketch We can easily check whether a given schedule \( \pi \) satisfies the constraints, so ReturnArrival is in \( \mathcal{NP} \). The SUM problem was studied by Couëtoux and Labourel \[18\]. They show its strong \( \mathcal{NP} \)-completeness by constructing a polynomial-time reduction from the Restricted Numerical 3-Dimensional Matching problem \[62\]. By Lemma 2.4, ReturnArrival is at least as hard as SUM. Hence ReturnArrival is strongly \( \mathcal{NP} \)-complete. □

Constant Request Times While the ReturnArrival problem is hard in general, there are some restricted cases where a solution can be found quickly. For instance, let us assume that the time periods between consecutive requests are all of equal length.

Lemma 2.6 Let \( n \geq 2 \). If \( \Delta_1 = \Delta_2 = \Delta_3 = \cdots = \Delta_{n-1} \), then there exists a feasible permutation \( \pi \in S_n \) if and only if \( \Delta_1 \geq 2n + 2 \).

Proof If \( \Delta_1 < 2n + 2 \) there is no solution, as the request that gets assigned to slot \( n \) (requiring \( 2n \) seconds for the robot to reach the slot) leaves not enough time to use any slot for any neighbouring request. If \( \Delta_1 \geq 2n + 2 \), the permutation \( \pi = (n, 1, n - 1, 2, n - 2, 3, n - 3, 4, \ldots) \) satisfies the constraints.
2.2. Arrival Only – the Early Morning Scenario

as \(2(\pi(1) + \pi(2)) = 2n + 2 \leq \Delta_1, 2(\pi(2) + \pi(3)) = 2n \leq \Delta_1\) and in general
\(2(\pi(i) + \pi(i + 1)) = 2n + 2 \leq \Delta_1\) for odd \(i\) and \(2(\pi(i) + \pi(i + 1)) = 2n \leq \Delta_1\)
for even \(i\). □

Busy Robot What if we know a priori that the customers arrive so densely packed that even with the best permutation \(\pi\) the robot will always be busy? In that case, we expect \(2(\pi(i) + \pi(i + 1)) = \Delta_i\) for all \(1 \leq i < n\). This corresponds to the TightSum problem with \(x_i = \lfloor \Delta_i/2 \rfloor\).

Lemma 2.7 TightSum can be solved efficiently, namely in time \(O(n^2)\).

Proof The following algorithm solves the TightSum problem using the insight that once we fix \(\pi(1)\) the entire permutation is determined. Hence we try out all possibilities for the first entry and then check how far we can get without reusing slots while keeping the robot busy. This approach is implemented below. As each of those checks runs in linear time, the running time of this algorithm is in \(O(n^2)\).

TightSum\(X)\
1 \textbf{for } \pi(1) = 1 \ldots n \\
2 \textbf{for } i = 2 \ldots n \\
3 \quad \pi(i) = x_{i-1} - \pi(i-1) \\
4 \quad \textbf{if } \pi(i) \notin [1,n] \text{ or } \pi(i) \in \{\pi(1), \ldots , \pi(i-1)\} \\
5 \quad \textbf{goto } 1 \quad \text{// We continue with the next } \pi(1) \text{ on line 1.} \\
6 \textbf{return Yes } \quad \text{// } \pi \text{ satisfies all the constraints.} \\
7 \textbf{return No } \quad \text{// There is no permutation that satisfies the constraints.}

2.2.2 The Direct Robot

We now show that the hardness of the ReturnArrival problem originates from the restriction that the robot has to return to the door in between requests. If the robot has to store a box far away from the door in one request and then has to fetch another box far away from the door in the next request, the empty drive to the door and back in between is a huge detour and might result in unnecessary delays. If we allow the robot to go straight from dropping a full box to picking up the next empty box, a wait-free slot assignment becomes significantly easier to find. We refer to this second robot model as Direct.

Proposition 2.8 The time needed for the Direct robot to start at the door, store a box at a location \(i\), pick up a new empty box at location \(j\) and return to the door is \(2 \max(i, j)\).

Proof If \(i < j\) the robot can drop off the full box on the way to pick up a new empty box. If \(i > j\) the robot picks up the empty box on the way back
Figure 2.3: If the Direct robot has to store a full box at location $i$ and fetch an empty box from slot $j$ the driving time required is $2 \max(i, j)$, regardless of the order of $i$ and $j$. If $i < j$ as shown on the left, the robot continues driving to the right after dropping the full box at location $i$ to fetch the empty box from slot $j$. If $i > j$ as shown on the right, the robot fetches the empty box on its way back to the door.

from dropping of the full box. So as illustrated in Figure 2.3, the distance travelled only depends on the more remote location in both cases.

Figure 2.4 gives an example of the movement of the robot in this Direct model. Note that the same instance would not allow a wait-free schedule for the Return robot. We see that the robot can now serve all the requests as long as the $n$ intervals of length $4\pi(i)$ centred around $t^a_{i}$ do not extend beyond the point of the previous and next request ($t^a_{i-1}$ and $t^a_{i+1}$). But it is no longer a problem if the intervals overlap. This allows us formulate the decision problem in a way that does not constrain pairs of permutation entries anymore.

**Definition 2.9** *(Direct Arrival-Only problem)* (DirectArrival) Given a multiset of integers $\Delta = \{\Delta_0, \Delta_1, \ldots, \Delta_n\}$, the question is to decide whether there is a feasible schedule for the Direct robot, namely a permutation $\pi \in S_n$ such that $2\pi(i) \leq \Delta_{i-1}$ and $2\pi(i) \leq \Delta_i$ for all $1 \leq i \leq n$.

**Lemma 2.10** **Constrained** is at least as hard as **DirectArrival**.

**Proof** Given an input to the DirectArrival problem, we simply use the constraint $x_i = \min \left( \left\lfloor \frac{\Delta_i}{2} \right\rfloor, \left\lfloor \frac{\Delta_{i-1}}{2} \right\rfloor \right)$ for $1 \leq i \leq n$ to get a single upper bound for each entry of the permutation in the form of the Constrained problem. □

Unlike **Sum**, the Constrained problem can be solved easily.

**Theorem 2.11** The **Constrained** problem can be solved in time $O(n)$.

**Proof** We give two algorithms that solve this problem efficiently. The first one, ConstrainedGreedy, takes one constraint after the other into account by greedily assigning the largest still available number that is not larger than $x_i$ to the next entry $\pi(i)$ of the permutation. That is, it lets the robot always take the rightmost box that it has enough time for. This greedy approach uses the boxes close to the door as late as possible. It can only get stuck at
2.2. Arrival Only – the Early Morning Scenario

Figure 2.4: A small example for the Direct robot. One feasible way to assign the three customers to storage slots is $\pi(1) = 3$, $\pi(2) = 1$ and $\pi(3) = 2$. The time constraints shown as intervals at the top and the position of the robot that is plotted in the middle do no longer correspond one to one in this robot model. For instance, it is not a problem that the intervals for storing the second box and fetching the third box overlap, as the robot can now combine the trip to store a full box and the trip to fetch a new empty box, as shown by the cutouts of seconds 17 and 18. Also note, that the four second drive required to fetch the empty box for the third customer is not performed immediately before this customer arrives at the door. Hence the robot even has a short break from time 20 to 21.

some position $i$ if there exists some threshold $t$ such that $|\{x_j | x_j \leq t, j \leq i\}|$, the number of requests that we considered so far and need to get assigned to the first $t$ slots, is bigger than $t$. In that case there clearly is no feasible permutation.

ConstrainedGreedy($X$)

1. for $i = 1 \ldots n$
2. $S = \{1, \ldots, x_i\} \setminus \{\pi(1), \ldots, \pi(i-1)\}$ // Set of reachable, free slots.
3. if $S = \emptyset$
4. return No // No permutation satisfies all the constraints.
5. $\pi(i) = \max(S)$
6. return Yes // $\pi$ satisfies the constraints.

This algorithm can be implemented efficiently as shown in the listing above. The crucial part is the implementation of lines 3 and 5 that find the rightmost available slot. If for each request a linear search is performed to find this slot, the total runtime will be $\Theta(n^2)$. This can be sped up if we use a union-find data structure [57]. We keep track of the free slots and their neighbouring used slots to the right so that we can quickly find the closest free slot at or to the left of a given $x_i$. This will result in a runtime of $\Theta(n \cdot \alpha(n))$ where $\alpha(n)$ denotes the inverse Ackermann function.
2. Request Handling

A second approach, called ConstrainedSortAndCompare, allows us to drop the \( a(n) \) factor. It sorts the constraints in increasing order and assigns the slots accordingly. The entry with the tightest constraint gets assigned number 1, the second tightest number 2 and so on. If at any point \( t \) in this process the constraint \( x_i \) is too tight, i.e. smaller than \( t \), there clearly is no solution as there are \( t \) requests with constraints below \( t \).

**ConstrainedSortAndCompare**\((X)\)

1. \( \text{for } i = 1 \ldots n \)
2. \( x_i = \min(x_i, n) \)
3. \( \sigma = \text{the permutation that sorts } X \text{ increasingly} \)
4. \( \text{for } i = 1 \ldots n \)
5. \( \text{if } x_{\sigma(i)} > i \)
6. \( \text{return No} \) // No permutation satisfies all the constraints.
7. \( \pi(\sigma(i)) = i. \)
8. \( \text{return Yes} \) // \( \pi \) satisfies the constraints.

As shown in the listing above, we can first clamp the constraints to \( n \) as larger constraints have no effect. This allows us to use distribution counting sort (independently developed by Seward [51] and Feurzig [24], described by Knuth [42] and Cormen et al. [17]) on the range from 1 to \( n \) and therefore to perform all steps of ConstrainedSortAndCompare in linear time. □

2.2.3 The Box-Swapping Robot

Finally, we make the model even simpler mathematically by allowing the robot to swap two boxes instantaneously. Thus if the robot arrives at a slot, it can immediately swap the box located there with the box it is carrying. In particular, whenever it stores a full box it can swap it with the empty box at that slot and be ready for the next request. So we no longer need extra time to prefetch an empty box before the next customer arrives. We additionally assume that the robot initially already carries an empty box. We refer to this robot model as Swap.

Figure 2.5 gives an example of the movement of the robot in this Swap model. Note that the same instance would not allow any wait-free schedules for the Return and Direct robots.

As no empty box needs to be prefetched for this Swap model, the constraints now only extend to the right in the time line. The gaps between the requests \( \Delta_i \) directly correspond to the constraints imposed on the slot assignments.

**Definition 2.12 (Swap Arrival-Only problem) (SwapArrival)** Given a multiset of integers \( \Delta = \{\Delta_0, \Delta_1, \ldots \Delta_n\} \), the question is to decide whether there is a
2.2. Arrival Only – the Early Morning Scenario

Figure 2.5: A small example for the Swap robot. One feasible way to assign the three customers to storage slots is $\pi(1) = 3$, $\pi(2) = 1$ and $\pi(3) = 2$. $\pi' = (2, 1, 3)$ would also be a feasible permutation. The intervals on top now denote the time to store a box. As no prefetching is needed anymore, they only extend to the right. Fetching an empty box is done automatically when the robot swaps the filled box from the previous request at its target position.

A feasible schedule for the Direct robot, namely a permutation $\pi \in S_n$ such that $2\pi(i) \leq \Delta_i$ for all $1 \leq i \leq n$.

As with the Direct Arrival problem, this is equivalent to the Constrained problem and can therefore be answered in linear time. Additionally, note that this model does not rely on the specific layout of our warehouse. It does not matter if there were multiple boxes stored at each slot or if the door would be in the middle of the warehouse or if we would even have an arbitrary topology. All we need to know is the time it takes the robot to get from the door to any given storage slot and back.

2.2.4 Online Request Arrival

Let us discuss our assumption that we know all the requests in advance. While it might be possible to infer a rough estimate of the customer arrival times based on their past behaviour, a more realistic scenario would be if the robot learns the requests only during the day as they occur. In the bicycle storage scenario it might still be possible to get a heads-up immediately before a customer arrives. GPS location of customers’ mobile phones or key cards at the entrance of the warehouse might allow the robot to know of a request some moments before the customer arrives at the door.

In this section, we briefly study this online version of the Direct Arrival and Swap Arrival problems. As these problems are Yes or No questions we will not talk about competitive ratios. There will just be a distinction between as good as offline and non-competitive. Do we need to know all the requests in advance? How far into the future do we need to see to act optimally?
Lemma 2.13 The **SwapArrival** problem can be solved optimally online if the robot knows at time $t_i^a$ the time of the next request $t_{i+1}^a$ or that there is no request in the next $2n$ seconds.

**Proof** We run the **ConstrainedGreedy** algorithm as it can easily run in an online fashion, taking the $\Delta_i$ into account one after the other. To fix $\pi(i)$ at time $t_i^a$ we only need to know $x_i = \min(\Delta_i, 2n)$, namely when the next customer arrives or if we have enough time to drive to every slot.

Lemma 2.14 The **SwapArrival** problem cannot be solved optimally online if the robot at time $t_i^a$ does not know the time of the next request $t_{i+1}^a$.

**Proof** If the robot picks $\pi(1) = 1$, an adversarial strategy will let $\Delta_1$ be big, and then $\Delta_2 = 2$ so that $\pi(2)$ would need to be 1. If the robot picks $\pi(1) > 1$, the adversary will let $\Delta_1 = 2$ so that the robot should have chosen $\pi(1) = 1$. No matter what the robot does, it will get stuck after the first request even though an offline solution might exist. Hence if the robot does not know how much time it has to serve a request, it cannot be competitive.

Lemma 2.15 The **DirectArrival** problem can be solved optimally online if the robot knows at time $t_i^a$ the time of the next two request $t_{i+1}^a$ and $t_{i+2}^a$.

**Proof** Knowing the next two requests in advance allows us to get $\Delta_i$ and $\Delta_{i+1}$ at time $t_i^a$. We thus have all the constraints on $\pi(i) \leq x_i = \min(\Delta_{i-1}, \Delta_i)$ and $\pi(i+1) \leq x_{i+1} = \min(\Delta_i, \Delta_{i+1})$. Hence we can also use the **ConstrainedGreedy** algorithm to decide which empty box should get picked up between $t_i^a$ and $t_{i+1}^a$.

Lemma 2.16 The **SwapArrival** problem cannot be solved optimally online if the robot at time $t_i^a$ does only know the time of the next request $t_{i+1}^a$.

**Proof** Whatever we take as $\pi(2)$, that is the slot that we fetch the empty box from between $t_i^a$ and $t_{i+2}^a$, the adversary can then try to pick $\Delta_2 < 2\pi(2)$, so that we will not have enough time to return that box afterwards. The adversary can always do this unless we pick $\pi(2) = 1$. But then the adversary can keep $\Delta_2$ large and pick $\Delta_3 = 4$, so that any feasible permutation requires $\{\pi(3), \pi(4)\} = \{1, 2\}$ and $\pi(2) > 2$. Hence if the robot only knows one request in advance, it cannot be competitive.
2.3 Arrival Only with Two Doors

We now add a second door to our warehouse located at position $n + 1$, at the other end of the warehouse. We consider the Swap robot model. It is initially located at the left door and carrying an empty box. We still consider costumers that only want to store their items but never retrieve them, so we can assume that there is always either a full box on the robot and it is on its way to store it or that there is an empty box that needs to go to one of the two doors. Figure 2.6 illustrates this setting.

Let $[x]$ denote the set of integers from 1 to $x$, so $[x] = \{1, \ldots, x\}$. In addition to the injective slot assignment function $\pi: [m] \to [n]$, we now also need to find a door assignment function $d: [m] \to \{l, r\}$ that specifies for each customer at which door he is served.

**Definition 2.17 (Feasible two-door schedule)** Given $n$, $m$ and a multiset of inter request arrival times $\Delta$, we call $(\pi, d)$ a feasible two-door schedule if and only if $d: [m] \to \{l, r\}$ and $\pi: [m] \to [n]$ satisfy that $d(1) = l$, $\pi$ is injective and for all $1 \leq i < m$

$$
\Delta_i \geq \begin{cases} 
2\pi(i) & \text{if } d(i) = d(i+1) = l \\
2(n + 1 - \pi(i)) & \text{if } d(i) = d(i+1) = r \\
n + 1 & \text{otherwise.}
\end{cases} \quad (2.1)
$$

Depending on $\Delta_i$, the time until the next customer arrives, we distinguish three kinds of requests.

- **short** requests with $\Delta_i \leq n$. This does constrain us to $d(i) = d(i+1)$, i.e., to serving the next customer at the same door as the current one, since the robot does not have enough time to get from one door to the other. Additionally we get the constraint $\pi(i) \leq \frac{\Delta_i}{2}$.

- **medium** requests with $n < \Delta_i < 2n$. This allows the robot to serve any slot if $d(i) \neq d(i+1)$. If it has to return to the same door, only the slots $\leq \frac{\Delta_i}{2}$ are reachable.

- **large** requests with $2n \leq \Delta_i$. This is enough time for the robot to reach any box and return to any door.
2. Request Handling

![Diagram of a warehouse with doors and slots]

Figure 2.7: An instance of the Fixed2Doors problem with \(m = 6, n = 6, \Delta = \{4, 8, 5, 7, 6\}\) and \(d = (l, l, l, l, r, r)\). The upper plot shows the maximal tour that the robot could take with these constraints. The plot at the bottom gives the six intervals created by the EarliestDeadlineFirstScheduling algorithm in sorted order, so that the entries of \(\pi\) can be assigned from left to right.

2.3.1 Fixed Door Assignment

In this section, we assume that the customers can choose at which door they want to interact with the warehouse. Thus, from the robot’s perspective, the door assignment \(d\) is already fixed.

**Definition 2.18 (Fixed2Doors)** Given \(n, m\), a multiset \(\Delta = \{\Delta_1, \ldots, \Delta_{m-1}\}\) and function \(d: [m] \to \{l, r\}\), the Fixed2Doors problem is to find \(\pi: [m] \to [n]\) such that \((\pi, d)\) is a feasible two-door schedule.

**Theorem 2.19** Fixed2Doors can be solved in time \(O(n)\).

**Proof** As the door assignment is fixed, we can determine for every request the set of slots that it could potentially be assigned to. If the robot switches to the other door during this requests, all slots can be used. If the robot goes back to the same door, then a certain number of slots that are closest to that door can be used. In either case, the set of reachable boxes forms a
2.3. Arrival Only with Two Doors

consecutive interval of slots. Hence, we can specify an interval \( I_i \) for each entry of \( \pi \), so that \( \pi_i \in I_i \) has to hold for all \( i \). Given these \( n \) intervals, we can then go through them from left to right in the order of increasing right borders and fix \( \pi \) always using the leftmost possible slot. This *earliest deadline first* greedy scheduling was first proposed by Liu and Layland [44] as the *deadline driven scheduling algorithm*. We implement this approach below and we illustrate it in Figure 2.7.

```
E
c
Earliest
a
EarliestDeadlineFirstScheduling(\(\Delta, d\))
1 // Determine the interval of possible slots for every request.
2 for i = 1 . . . m - 1
3 if \(d(i) = d(i + 1) = l\)
4 \(I_i = [1, \lfloor \frac{\Delta}{2} \rfloor]\)
5 if \(d(i) = d(i + 1) = r\)
6 \(I_i = [n + 1 - \lceil \frac{\Delta}{2} \rceil, n]\)
7 if \(d(i) \neq d(i + 1)\)
8 if \(\Delta_i \leq n\)
9 return No // Not enough time to reach the other door.
10 \(I_m = [1, n]\) // The last request can be served anywhere.
11 \(\sigma = \) the permutation that sorts the intervals \(I_1, \ldots, I_m\) increasingly in their right border and breaks ties increasingly in the left border.\(^a\)
12 // Assign the slots from left to right in this sorted order \(\sigma\).
13 for i = 1 . . . m
14 \(S = I_{\sigma(i)} \cap ([n] \setminus \{\pi_{\sigma(1)}, \ldots, \pi_{\sigma(i-1)}\})\) // Set of free, reachable slots.
15 if \(S = \emptyset\)
16 return No // Request \(\sigma(i)\) cannot be served in time.
17 \(\pi(\sigma(i)) = \min(S)\) // Take the leftmost, free slot.
18 return Yes // \(\pi\) satisfies the constraints.
```

\(^a\)As all the intervals that we generate in our setting start at 1 or end at \(n\) (or both), this sorting order is equivalent to having first all left-aligned intervals in the order of increasing endpoints and then all right-aligned intervals in the order of increasing startpoints, as illustrated in Figure 2.7.

We can use distribution count sort on line 12, hence all steps of *EarliestDeadlineFirstScheduling* can be done in linear time. \(\square\)

### 2.3.2 Hardness of Flexible Door Assignment

We make the warehouse slightly more powerful by allowing it to pick the door for each request. In this section, we show that this change makes it \(\mathcal{NP}\)-complete to decide whether a feasible schedule exists.
2. Request Handling

Definition 2.20 (2Doors) Given \( n, m \) and a multiset \( \Delta = \{\Delta_1, \ldots, \Delta_{m-1}\} \), the 2Doors problem is to find \( \pi : [m] \to [n] \) and \( d : [m] \to \{l, r\} \) such that \((\pi, d)\) is a feasible two-door schedule.

We show the hardness of 2Doors by mapping every formula in 3-conjunctive normal form to an instance of a warehouse with two doors.

Definition 2.21 (3SAT) Given a boolean formula \( F = \{C_1, \ldots, C_M\} \) of \( M \) clauses. Each clause consists of exactly three literals \( C_i = \{l_i, l_{i2}, l_{i3}\} \) over a set of \( N \) variables \( x_1, \ldots, x_N \) and their negations \( \bar{x}_1, \ldots, \bar{x}_N \). Is there a boolean assignment \( \alpha : \{x_1, \ldots, x_N\} \to \{True, False\} \) that satisfies all \( M \) clauses?

Cook [16] and Levin [43] independently showed that 3SAT is \( \mathcal{NP} \)-complete.

Theorem 2.22 [16], [43] 3SAT is \( \mathcal{NP} \)-complete.

Theorem 2.23 2Doors is at least as hard as 3SAT.

Proof of Theorem 2.23 (Construction) We prove this theorem in two parts. In this first part, we describe a reduction \( \varphi \) that maps every formula \( F \) in 3-conjunctive normal form to a multiset \( \Delta = \varphi(F) \) for the 2Doors problem. After stating a sequence of lemmas, we will show the correctness of this construction in the second part of the proof.

We start with some observations on the 2Doors problem. First, note that successive short requests all have to get assigned to the same door. We call such a consecutive sequence of short requests a request block. Our reduction will only use short and long requests, none of medium size. In fact, we will use a sequence of blocks of short requests and interleave these blocks with single long requests. This way, each block can be independently assigned to the left or to the right door and all long requests in between can get assigned to any slot regardless of these decisions.

Given \( F \), we create a warehouse with \( n = 2N + 6M + E \) many storage slots, where \( N \) is the number of variables, \( M \) is the number of clauses, and \( E \) is a number that we will fix later. We label each of these slots as illustrated in Figure 2.8: The first \( N \) slots are labelled \( x_1 \) to \( x_N \), followed by \( 3M \) slots that are labelled \( C_1 \) to \( C_M \) with three consecutive slots of the same label for each clause. In the centre, there are \( E \) many slots labelled \( E \). The remaining \( N + 3M \) boxes are mirrored from the left side, so first \( 3M \) slots for the clauses and then \( N \) slots for the variables, both in decreasing order. From now on, we call the left door the false door and the right door the true door.

Let us go through the blocks of short requests that \( \varphi \) creates. We wrote such a block \( B \) as a list of slot labels. The function \( f \) maps each slot label in this list to a short request that is just long enough to reach all the corresponding slots with that label when the robot starts at the closer door. So \( f(x_i) = 2i \) and \( f(C_i) = 2(N + 3i) \). As an example, the block \( B = \{C_1, x_4, C_5\} \) maps
to the request sequence \( f(B) = \{ f(C_1), f(x_4), f(C_5) \} = \{ 2(N + 3 \cdot 1), 2 \cdot 4, 2(N + 3 \cdot 5) \} \).

At first, \( \varphi(F) \) generates a prefill block \( B_{\text{prefill}} = \{ C_1, C_2, \ldots, C_M \} \). It then generates \( 2N \) literal blocks. For every possible literal, we create one block that contains this literal and all the clauses that contain it. Hence, \( B_{x_i} = \{ x_i \} \cup \{ C \in F \mid x_i \in C \} \) and \( B_{\bar{x}_i} = \{ \bar{x}_i \} \cup \{ C \in F \mid \bar{x}_i \in C \} \). As each clause contains exactly three literals, every clause will appear in exactly three literal blocks. Therefore, these \( 2N \) literal blocks contain \( 2N + 3M \) requests in total. The end of \( \varphi(F) \) consists of \( 2M \) fillup blocks, two for each clause, that contain a single request each: \( B_{C_i} = \{ C_i \} \). So the total sequence of blocks is \( B_{\text{prefill}}, B_{x_1}, B_{\bar{x}_1}, \ldots, B_{x_N}, B_{\bar{x}_N}, B_{C_1}, B_{C_2}, \ldots, B_{C_M}, B_{C_M} \). We interleave all of these blocks with single long requests \( \Delta_m = N + 1 \).

\[
\varphi(F) = \{ f(B_{\text{prefill}}), \Delta_\leftrightarrow, f(B_{x_1}), \Delta_\leftrightarrow, f(B_{\bar{x}_1}), \Delta_\leftrightarrow, \ldots, \}
\]
\[
f(B_{x_N}), \Delta_\leftrightarrow, f(B_{\bar{x}_N}), \Delta_\leftrightarrow, f(B_{C_1}), \Delta_\leftrightarrow, f(B_{C_2}), \Delta_\leftrightarrow, \ldots \}
\]
\[
f(B_{C_M}), \Delta_\leftrightarrow, f(B_{C_M}) \}
\]

As we have \( |\varphi(F)| = 2N + 6M + (2N + 2M) \) many requests in total, we set \( E = 1 + 2N + 2M \) such that there are as many slots as requests, \( n = m \). Recall that \( |\Delta| = |\varphi(F)| = m - 1 \) and not \( m \) because the last request can be served at any slot and so no \( \Delta_m \) has to be specified. This concludes the first part of our proof and we will now state some lemmas that will allow us to show the correctness of this construction in the second part of the proof.

**Lemma 2.24** All long requests \( \Delta_\leftrightarrow \) have to get assigned to slots labelled \( E \).

**Proof** \( n = m \) implies that no slot can be left unused. All other requests are short requests of length at most \( 2(N + 3M) \) and therefore cannot reach any box labelled \( E \). There are as many long requests as there are boxes labelled \( E \) which concludes the proof. \( \square \)

**Lemma 2.25** Every request gets assigned to a slot with the same label.

**Proof** For every label, there are equally many requests and slots. Lemma 2.24 takes care of all long requests and it remains to show the claim for the short requests. Recall that for these requests, the robot has to return to the same door. While we could assign a request to a slot that is closer to the door.
than the slot corresponding to the request’s label, this, by pigeonhole principle, would cause another request to be assigned even closer to the door. Repeating this argument leads to a request that cannot be served at all, as we end up with more requests than slots for the closest label $x_1$. Hence the only way to serve all the requests is to assign them each to a slot with the corresponding label. $\square$

Lemma 2.26 The corresponding blocks $B_{x_i}$ and $B_{\bar{x}_i}$ get assigned to opposite doors.

Proof Only the blocks $B_{x_i}$ and $B_{\bar{x}_i}$ contain requests labelled $x_i$. As there is only one slot with this label at each door, by Lemma 2.25 the two blocks have to get assigned to different doors. $\square$

Lemma 2.27 For every clause $C_i$, at least one of the three literal blocks that $C_i$ appears in has to get assigned to the slot near the true door.

Proof As the prefill block is the very first block, all its requests have to get assigned on the side of the false door. So by the time the literal blocks start, one out of three slots for each clause near the false door are already occupied. Recall that every clause $C_i = \{l_{i_1}, l_{i_2}, l_{i_3}\}$ appears in exactly three literal blocks, namely $B_{l_{i_1}}, B_{l_{i_2}}, B_{l_{i_3}}$. Hence by Lemma 2.25, at least one of these three literal blocks has to get assigned to the slot near the true door. $\square$

Proof of Theorem 2.23 (Correctness) It remains to show that $\varphi(F)$ has a feasible two-door schedule $(\pi, d)$ if and only if $F$ is satisfiable.

$\Rightarrow$: Given a feasible two-door schedule, we can read off the $F$-satisfying assignment. For every variable $x_i$, we set it to true if and only if the corresponding literal block $B_{x_i}$ is assigned to the true door. As by Lemma 2.27, every clause contains at least one literal whose block gets assigned to the true door, all clauses are satisfied.

$\Leftarrow$: Given an $F$-satisfying assignment $\alpha$, we can easily find a feasible two-door schedule for $\varphi(F)$. We assign $B_{\text{prefill}}$ to the false door. The literal blocks $B_{x_1}$ to $B_{x_N}$ get set as in the assignment, so $B_{x_i}$ on the true side if and only if $\alpha(x_i) = \text{true}$. The $2M$ fillup blocks are then just assigned to fill the remaining literal slots. For a clause $C_i$, as many fillup blocks $B_{C_i}$ get assigned to the false door as there are literals in $C_i$ that evaluate to false. This allows us to fix the door assignment $d$. The slot assignment $\pi$ can then be generated using EarliestDeadlineFirstScheduling.

As the size of the 2Doors instance $\varphi(F)$ only grows by a constant factor compared to $F$, the reduction runs in polynomial time. $\square$

Corollary 2.28 2Doors is $NP$-complete.

Proof 2Doors is in $NP$, as it is trivial to verify whether a given $\pi$ and $d$ form a feasible two-door schedule according to Definition 2.17. By Theorem 2.22 and Theorem 2.23, 2Doors is $NP$-complete. $\square$
2.3.3 Approximation of Flexible Door Assignment

For hard problems, it is natural to think of approximation algorithms that try to come as close as possible to the optimal solution but run in polynomial time. For the 2Doors problem, an interesting aspect would be to consider delays, meaning that the robot can arrive at the door late and we want to minimize the accumulated wait time of the customers. But since Theorem 2.23 showed that it is hard to distinguish instances with and without inevitable wait-times, we cannot expect any relative performance guarantee in such a wait-time-minimization model.

Instead we want to give the robot the power to skip certain requests of its choice if it does not find enough time to serve all of them. This means that we want to find a largest possible subset of the requests that can be served without causing any delays to those customers.

**Definition 2.29** A function $\pi: [m] \rightarrow [n] \cup \{\diamond\}$ is called a partial permutation if it is injective with respect to $[n]$, meaning that $\forall i \neq j: \pi(i) = \pi(j) \Rightarrow \pi(i) = \diamond$. (2.5)

The constant function $\pi: [m] \rightarrow \{\diamond\}$ is called the empty permutation and is denoted with $\pi_\diamond$.

To model request skipping, we define a partial slot assignment as a partial permutation $\pi: [m] \rightarrow [n] \cup \{\diamond\}$, where $\pi(i) = \diamond$ denotes that the $i$th request gets skipped. We denote the number of served requests by $|\pi| = |\{i \mid \pi(i) \neq \diamond\}|$. For simplicity, we still require the robot to meet all customers at the door, even those that do not get served. This allows us to only slightly adapt Definition 2.17 of a feasible schedule to this new setting of partial slot assignments. All we add is a distinction on whether we serve a request. If we do, the definition stays unchanged. If we do not, then we simply check whether the robot has enough time to switch doors if necessary.

**Definition 2.30 (Feasible partial two-door schedule)** Given a multiset of inter request arrival times $\Delta$, we call $(\pi, d)$ a feasible partial two-door schedule if and only if the function $d: [m] \rightarrow \{l, r\}$ and partial permutation $\pi: [m] \rightarrow [n] \cup \{\diamond\}$ satisfy that $d(1) = l$ and for all $1 \leq i < m$

- if $\pi(i) \neq \diamond$

$$
\Delta_i \geq \begin{cases} 
2\pi(i) & \text{if } d(i) = d(i + 1) = l \\
2(n + 1 - \pi(i)) & \text{if } d(i) = d(i + 1) = r \\
n + 1 & \text{otherwise.}
\end{cases}
$$

(2.6)
• if \( \pi(i) = \diamond \)

\[
\Delta_i \geq \begin{cases} 
0 & \text{if } d(i) = d(i+1) \\
n + 1 & \text{otherwise.}
\end{cases}
\]  

(2.7)

**Definition 2.31 (Approx2Doors)** Given \( n, m \) and a multiset \( \Delta = \{ \Delta_1, \ldots, \Delta_{m-1} \} \) that allows a feasible two-door schedule \( (\pi^*, d^*) \), the **Approx2Doors** problem is to find \( \pi: [m] \rightarrow [n] \cup \{ \diamond \} \) and \( d: [m] \rightarrow \{ l, r \} \) such that \( (\pi, d) \) is a feasible partial two-door schedule and \(|\pi|\) is as large as possible.

**Definition 2.32 (Approximation Algorithm)** An algorithm \( A \) is called an \( \alpha \)-approximation for **Approx2Doors** if for every multiset \( \Delta \) that allows a feasible two-door schedule, the schedule \( (\pi, d) = A(\Delta) \) serves at least \( \alpha \cdot m \) customers.

**Deterministic Approximation Algorithms**

As we have seen in the previous sections, the hard part of the 2Doors problem is assigning the requests to the doors. Once \( d \) is fixed, the **Earliest-Deadline-First-Scheduling** can maximize \(|\pi|\). All we need to change in the implementation we have given is line 17 from "return No" to "\( \pi(\sigma(i)) = \diamond \)" in order to just skip requests that cannot be assigned to a free slot and not break prematurely. Therefore, we can fully describe an approximation algorithm by specifying how it determines the door assignment. Let us start with a very simple approximation algorithm.

**Definition 2.33** Let **Never** be the algorithm that never switches to the right door. This means \( d(i) = l \) for all \( 1 \leq i \leq m \).

**Lemma 2.34** **Never** is a \( \frac{1}{2} \)-approximation for **Approx2Doors**.

**Proof** Given an instance \( \Delta \), let \( (\pi^*, d^*) \) be the optimal schedule that serves all \( m \) requests. Let \( (\pi, d) \) be the feasible partial schedule produced by **Never**. We now check whether \( \pi^* \) serves more requests on the left or on the right half of the warehouse. For this, let \( \pi^*_{\text{left}} \) be the partial permutation of the requests assigned to the left half, so

\[
\pi^*_{\text{left}}(i) = \begin{cases} 
\pi^*(i) & \text{if } \pi^*(i) \leq \left\lceil \frac{n}{2} \right\rceil \\
\diamond & \text{otherwise}
\end{cases}
\]

(2.8)

and let \( \pi^*_{\text{right}} \) be the partial permutation of the requests assigned to the right half, mirrored to the left half. This means that for every request that \( \pi^* \) assigns to the right half, we let \( \pi^*_{\text{right}} \) take the mirrored position on the left half for that request. Formally,

\[
\pi^*_{\text{right}}(i) = \begin{cases} 
\pi^*(i) - \left\lceil \frac{n}{2} \right\rceil & \text{if } \pi^*(i) > \left\lceil \frac{n}{2} \right\rceil \\
\diamond & \text{otherwise.}
\end{cases}
\]

(2.9)
2.3. Arrival Only with Two Doors

As \(|\pi^*| \leq |\pi^*_{\text{left}}| + |\pi^*_{\text{right}}|\) we have \(\max(|\pi^*_{\text{left}}|, |\pi^*_{\text{right}}|) \geq \frac{1}{2}|\pi^*|\). Thanks to the mirroring in \(\pi^*_{\text{right}}\), both of these schedules only use slots in the left half. Therefore, a door assignment that always stays at the left door can incorporate these schedules. So both \((\pi^*_{\text{left}}, d^*)\) and \((\pi^*_{\text{right}}, d^*)\) are feasible partial schedules. But as \((\pi, d)\) is a maximal schedule among all those with the fixed door assignment \(d\), we have

\[
|\pi| \geq \max(|\pi^*_{\text{left}}|, |\pi^*_{\text{right}}|) \geq \frac{1}{2}|\pi^*|.
\]

(2.10)

Figure 2.9 gives an example of these partial two-door schedules.

**Lemma 2.35** The approximation ratio \(\frac{1}{2}\) for Never is tight.

**Proof** For any even \(n\), let \(\Delta = \{2, 4, 6, \ldots, n - 2, 2n, 2, 4, 6, \ldots, n - 2\}\). So \(n = m\) and \(d^*\) clearly assigns the first half of all requests to the left door and the second half to the right. This allows an optimal algorithm to satisfy all \(n\) requests whereas Never can only satisfy \(\frac{n}{2} + 1\) requests. For \(n \to \infty\) the ratio of served requests goes to \(\lim_{n \to \infty} \frac{\frac{n}{2} + 1}{n} = \frac{1}{2}\). This is illustrated in Figure 2.10. \(\square\)

We now generalize this slightly to see that every reasonable approximation algorithm has an approximation ratio of \(\geq \frac{1}{2}\).

**Definition 2.36** A door assignment \(d: [m] \to \{l, r\}\) is called feasible with respect to an instance \(\Delta\) if and only if it forms a feasible partial two-door schedule with the empty permutation \(\pi^*\).

**Theorem 2.37** Every algorithm that fixes any feasible door assignment \(d\) and then applies EarliestDeadlineFirstScheduling is an \(\frac{1}{2}\)-approximation.

**Proof** Intuitively speaking, never switching to the other door, like Never, seems to be the most restricting choice. So any other feasible door assignment should allow to serve at least that many requests. We can formalize
this by looking at the two partial permutations $\pi_{\text{left}}^*$ and $\pi_{\text{right}}^*$ of the optimal schedule $(\pi^*, d)$ again as in the proof of Lemma 2.34. We let $\pi_{\text{max}}^*$ be the larger of the two and show that we can adopt this slot assignment to fit with any feasible door assignment $d$. For this we define $\pi_{\text{max}, d}^*$ as

$$
\pi_{\text{max}, d}^*(i) = \begin{cases} 
\pi_{\text{max}}^*(i) & \text{if } d(i) = l \\
 n + 1 - \pi_{\text{max}}^*(i) & \text{if } d(i) = r 
\end{cases}
$$

(2.11)

As $\pi_{\text{max}}^*$ only used slots in the left half of the warehouse, $\pi_{\text{max}, d}^*$ will not use any slot twice and is still a partial permutation. Additionally, since $(\pi^*, d^*)$ was feasible and every request gets assigned to a slot with the exact same distance to the door in $\pi_{\text{max}, d}^*$, we know that also $(\pi_{\text{max}, d'}^*, d)$ is a feasible partial two-door schedule. Finally $|\pi_{\text{max}, d}^*| = |\pi_{\text{max}}^*| \geq \frac{1}{2} |\pi^*|$ gives the approximation guarantee. □

Taking this general lower bound, let us look at some other approximation strategies.

**Definition 2.38** Let **Always** be the algorithm that always switches to the other door if the request time allows. Formally, $d: [m] \to \{l, r\}$ such that

$$
d(i) = \begin{cases} 
l & \text{if } i = 1 \\
d(i - 1) & \text{if } i > 1 \text{ and } \Delta_i \leq n \\
-d(i - 1) & \text{otherwise},
\end{cases}
$$

(2.12)

where $\neg$ denotes negation, so $\neg l = r$ and $\neg r = l$.

**Definition 2.39** Let **Balance** be the algorithm that for every block of short requests greedily puts it on that side that has less requests served so far. If both sides are equivalent, put it on the left side.
Definition 2.40 Let \textsc{BlockGreedy} be the algorithm that for every block of short requests greedily puts it on that side that increases the number of served requests after an \textsc{EarliestDeadlineFirstScheduling}. If both sides are equivalent, put it on the left side.

Definition 2.41 Let \textsc{Local} be the algorithm that starts with all requests assigned to the left door. It then iteratively puts a block of short requests to the opposite door if this step increases the number of requests served after an \textsc{EarliestDeadlineFirstScheduling} the most. As long as there is such a block whose door-flip increases the number of requests served, we flip the door assignment of such a block.

Note that all of these algorithms run in polynomial time. \textsc{Local} terminates after at most \( m \) local improvement steps, which we call block flips. Each block flip requires a linear number of runs of the \textsc{EarliestDeadlineFirstScheduling} procedure.

As with \textsc{Never} in Lemma 2.35, we now give an infinite sequence of instances that establishes an upper bound on the approximation ratio for each of these new algorithms. As in Lemma 2.35, these examples will only achieve the bounds after we scale the warehouse and duplicate the short requests to make the few large requests eventually irrelevant in comparison to the huge blocks of short requests. We formalize this idea of scaling and multiplying the short requests in order to make the large requests irrelevant, and can give very small examples afterwards. But first we have to specify for which algorithms this idea will work: for those algorithms that do not change the block-to-door assignments after scaling. Figure 2.11 illustrates this idea of scaling-unawareness.

Definition 2.42 (Scaling-unawareness) We call a two-door approximation algorithms \( A \) scaling-unaware if scaling the warehouse by some factor \( k \) and replacing all short requests by \( k \) copies does not change its door assignment for every block of short requests.

Proposition 2.43 All our algorithms \textsc{Never}, \textsc{Always}, \textsc{Balance}, \textsc{BlockGreedy} and \textsc{Local} are clearly scaling-unaware as they all just compare relative block sizes.

To denote the number of served short requests in a partial two-door schedule, we let \( |\pi|_\Delta = |\{i \mid \pi(i) \neq \diamond \text{ and } \Delta_i \leq n\}|. \)

Lemma 2.44 Let \( n, m \) and \( \Delta \) be an instance with optimal schedule \((\pi^*, d^*)\) and let \( A \) be a scaling-unaware algorithm that produces the partial schedule \((\pi, d)\), then \( A \) is at most a \( |\pi|_\Delta / |\pi^*|_\Delta \)-approximation.

Proof For any integer \( k \), let \( n_k = n \cdot k, m_k = m + (k - 1) \cdot |\pi^*|_\Delta \) and \( \Delta_k \) be a copy of \( \Delta \) where every short request gets repeated \( k \) times and all requests are scaled by a factor of \( k. \)
2. Request Handling

Figure 2.11: Example of scaling the warehouse by a factor 3 and copying the short requests 3 times. There are two blocks of short requests both before and after the scaling. For scaling-unaware approximation algorithms that assign the two blocks the same way before and after the scaling, the two cases depicted are possible. At the top we show the door assignment, where the first block is assigned to the left door and the second block is assigned to the right door. These assignments allow all requests to be satisfied, before and after the scaling. In the assignment at the bottom, both blocks are assigned to the left door. Before the scaling, one out of five requests could not be satisfied. After the scaling, it is three out of eleven. For bigger and bigger scaling factors, the two long requests would eventually be irrelevant, leaving one third of the requests unserved in the limit.

Since a scaling-unaware algorithm simply extends its door assignment to these scaled instances we will have $|\pi_k| = |\pi| + (k - 1) \cdot |\pi|_\Delta$ and $|\pi^*_k| = m_k = |\pi^*| + (k - 1) \cdot |\pi^*|_\Delta$. Letting $k$ grow leads to the claim

$$\lim_{k \to \infty} \frac{|\pi_k|}{|\pi^*_k|} = \lim_{k \to \infty} \frac{|\pi| + (k - 1) \cdot |\pi|_\Delta}{|\pi^*| + (k - 1) \cdot |\pi^*|_\Delta} = \frac{|\pi|_\Delta}{|\pi^*|_\Delta} \quad (2.13)$$

With this lemma, we can now easily give the following upper bounds. Figure 2.12 illustrates these upper bounds for the four algorithms. These bounds were found after an automatic randomized search for hard examples.

**Lemma 2.45** The approximation ratio $\frac{1}{2}$ for **Always** is tight.

**Proof** The block sequence $\{2\}, \{\}, \{2\}$ lets $|\pi|_\Delta = 1$ and $|\pi^*|_\Delta = 2$ and establishes the bound by Lemma 2.44.

**Lemma 2.46** **Balance** is at most a $\frac{3}{4}$-approximation.
Figure 2.12: Upper bounds for the approximation ratio of four deterministic algorithms corresponding to the examples given in Lemmas 2.45 to 2.48. For every algorithm we show the optimal block-to-door assignment on the left and the one generated by the algorithm on the right.

**Proof** The block sequence \{4\}, \{4\}, \{2, 4\} lets \(|\pi|_\Delta = 3|\) and \(|\pi^*|_\Delta = 4|\) and establishes the bound by Lemma 2.44.

**Lemma 2.47** BlockGreedy is at most a \(\frac{3}{4}\)-approximation.

**Proof** The block sequence \{4\}, \{4\}, \{2\}, \{2\} lets \(|\pi|_\Delta = 3|\) and \(|\pi^*|_\Delta = 4|\) and establishes the bound by Lemma 2.44.

**Lemma 2.48** Local is at most a \(\frac{4}{5}\)-approximation.

**Proof** The block sequence \{8\}, \{10, 10\}, \{2, 10\}, \{2\}, \{4\}, \{6\}, \{4, 6\} lets \(|\pi|_\Delta = 8|\) and \(|\pi^*|_\Delta = 10|\) and establishes the bound by Lemma 2.44.

**Randomized Approximation Algorithms**

As we could not show any lower bounds above \(\frac{1}{2}\) for any of the deterministic approximation algorithms, we started looking at randomized approximation algorithms. We will now see that a very simple randomized algorithm achieves an approximation ratio of \(\frac{3}{4}\) in expectation. For this section, we assume that we are given instances that consist only of short and long requests, so no medium requests. This is just for the simplicity of the argument as
medium request would make argument below more technical. Our randomized algorithm should be as simple as possible: for every new block, so whenever we could reach the other door with \( \Delta_i > n \), we toss a coin and with equal probability either stay at the same door as before or switch to the other side.

**Definition 2.49** Let \( \text{Random} \) be the randomized algorithm that picks a door uniformly at random for every block. Formally, \( d: [m] \to \{l, r\} \) such that

\[
d(i) = \begin{cases} 
  l & \text{if } i = 1 \\
  d(i - 1) & \text{if } i > 1 \text{ and } \Delta_i \leq n \\
  \in_{u.a.r.} \{l, r\} & \text{otherwise}
\end{cases}
\]  

(2.14)

**Theorem 2.50** \( \text{Random} \) is a \( \frac{3}{4} \)-approximation in expectation for \text{Approx2Doors}.

**Proof** Let us consider an optimal schedule \((\pi^*, d^*)\). Let \( d \) be the random door assignment generated by our algorithm. Since the large requests can get served regardless of the door assignment, we focus on the short requests. The main idea will be to look at the pairs of entries of \( \pi^* \) that get assigned to the slots at equal distance from the two doors, so \((1, n), (2, n - 1), (3, n - 2), (4, n - 3)\) and so on. For request \( i \), we denote the request that is served at the position mirrored of \( \pi^*(i) \) as \( \text{inv}(i) \) with \( \text{inv}(i) = \pi^* - 1(n + 1 - \pi^*(i)) \) and call this is inverse request. So if no request is served at the slot opposite of request \( i \), we have \( \text{inv}(i) = \emptyset \). See Figure 2.13 for an illustration.

To argue that we will serve in expectation at least three requests for every four requests that the optimal solution serves, let us define a new partial permutation \( \pi^*_d \). This permutation \( \pi^*_d \) will form a feasible partial schedule when combined with the random door assignment \( d \) and will only serve a subset of the requests served in the optimal solution \( \pi^* \). So if \( \pi^*(i) = \emptyset \), we will also have \( \pi^*_d(i) = \emptyset \). For a request \( i \) that does get served in \( \pi^* \) we distinguish whether \( d^* \) and \( d \) both assign this request to the same door or not. If they are served at the same door, we just serve the request at the same slot and let \( \pi^*_d(i) = \pi^*(i) \). If they are served at opposite doors, we
check what happens to the inverse request of \( i \). If there is no inverse request, so \( \text{inv}(i) = \emptyset \), the slot opposite of \( \pi^*(i) \) is free and we can take it, so \( \pi_d^*(i) = n + 1 - \pi^*(i) \). If there is an inverse request but the door assignment for this inverse request has also changed, we can assign request \( i \) to the slot opposite of \( \pi^*(i) \), so \( \pi_d^*(i) = n + 1 - \pi^*(\text{inv}(i)) \). Only if the door assignment for the inverse request is unchanged, we give way to this inverse request and will not serve request \( i \), so \( \pi_d^*(i) = \emptyset \). This case distinction is summarized as follows:

\[
\pi_d^*(i) = \begin{cases} 
\emptyset & \text{if } \pi^*(d) = \emptyset \\
\pi^*(i) & \text{if } d(i) = d^*(i) \\
 n + 1 - \pi^*(i) & \text{if } d(i) \neq d^*(i) \text{ and } \text{inv}(i) = \emptyset \text{ or } d(\text{inv}(i)) \neq d^*(\text{inv}(i)) \\
\emptyset & \text{otherwise}
\end{cases}
\]  

(2.15)

We can now see that for every pair of opposite requests, we expect to keep at least \( \frac{3}{4} \) of them. If \( \pi^*(i) = \emptyset \) or \( \pi^*(\text{inv}(i)) = \emptyset \), so at least one of the slots in the pair is not used in the optimal solution, equally many slots will be used in \( \pi_d^* \).

The interesting case, illustrated in Figure 2.14, is if both slots are used in the optimal solution, so \( \pi^*(i) \neq \emptyset \) and \( \pi^*(\text{inv}(i)) \neq \emptyset \). If the requests are both assigned to the same door in \( \pi_d^* \), only one of them can and will be kept. If they are assigned to different doors, then both can be kept, but their assignment might get flipped. The two requests \( i \) and \( \text{inv}(i) \) must be in different blocks of short requests as they are assigned to different halves of the warehouse in the optimal assignment \( \pi^* \). Therefore, their door assignment in \( d \) is independently uniformly at random. With probability \( \frac{1}{2} \), they end up in the same half and with probability \( \frac{1}{2} \) in different halves. If they end up in different halves, both can be served. If they end up in the same half, only one of them can be served. Therefore the expected number of request we serve in \( \pi_d^* \) is

\[
\mathbb{E}[|\{\pi_d^*(i), \pi_d^*(\text{inv}(i))\} \setminus \{\emptyset\}|] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 = \frac{3}{2} = \frac{3}{4} \cdot 2.
\]  

(2.16)

By linearity of expectation across all the requests, we get \( \mathbb{E}[|\pi_d^*|] = \frac{3}{4} |\pi^*| \).

Since \( (\pi_d^*, d) \) is a feasible partial schedule, it is also a lower bound for the schedule \( (\pi, d) \) that RANDOM generates since RANDOM is scheduling optimally for the given door assignment \( d \).

\( \square \)

**Lemma 2.51** The expected approximation ratio of \( \frac{3}{4} \) for RANDOM is tight.
2. Request Handling

setting in the optimal assignment
\[ |\{\pi^*(i), \pi^*(\text{inv}(i))\} \setminus \{\diamond\}| = 2 \]

\[
\begin{array}{c}
\text{(}\pi^*, d^*) \\
\text{i} \quad \cdots \quad \text{inv(i)}
\end{array}
\]

the four possible cases in the randomized assignment
\[
\begin{array}{c|c}
|\{\pi^*_d(i), \pi^*_d(\text{inv}(i))\} \setminus \{\diamond\}| = 2 & |\{\pi^*_d(i), \pi^*_d(\text{inv}(i))\} \setminus \{\diamond\}| = 1 \\
\begin{array}{c}
\text{(}\pi^*_d, d) \\
\text{i} \quad \cdots \quad \text{inv(i)}
\end{array} & \begin{array}{c}
\text{(}\pi^*_d, d) \\
\text{i} \quad \cdots \quad \text{inv(i)}
\end{array} \\
d(i) = d^*(i) & d(\text{inv}(i)) = d^*(\text{inv}(i)) \\
\begin{array}{c}
\text{(}\pi^*_d, d) \\
\text{inv(i)} \quad \cdots \quad \text{i}
\end{array} & \begin{array}{c}
\text{(}\pi^*_d, d) \\
\text{inv(i)} \quad \cdots \quad \text{i}
\end{array} \\
d(\text{inv}(i)) \neq d^*(\text{inv}(i)) & d(\text{inv}(i)) \neq d^*(\text{inv}(i))
\end{array}
\]

Figure 2.14: Illustration of what happens to a pair of opposite requests in our randomized block assignment. In two of the four cases, we can still serve both requests. In the other two cases, we can only serve one request.

Proof RANDOM is clearly scaling unaware. Hence we can use Lemma 2.44 and the block sequence \{2\}, \{2\} that leads to \(|\pi^*|_A = 2\) and \(E[|\pi|] = \frac{3}{2}\), as illustrated in Figure 2.15, to show that the bound of \(\frac{3}{4}\) is tight. \(\square\)

2.4 Arrivals and Departures - the Entire Day

In this section, we allow for both arriving and departing customers. We will do this change from the previous arrival-only setting as gradually as possible to see how far the results from the previous section transfer to this more general setting. Throughout this section we assume that we are working with a SWAP robot, even though some results also apply to the DIRECT and RETURN robot models. For simplicity, we also still assume that there are as many slots as customers, so \(n = m\).

First, in Section 2.4.1, we assume that all arrivals occur before all departures and that there is enough time in between for the robot to reorder the stored boxes. Second, in Section 2.4.2, we allow that the arrivals and departures are
2.4. Arrivals and Departures - the Entire Day

Interleaved arbitrarily but assume that every customer has his own personal slot. This means that a slot cannot get reused to serve another customer that arrives at the warehouse after the previous one has already departed. We will see that our results from the arrival-only problem can still be applied in both of these settings.

Finally, in Section 2.4.3 we allow for slot reuse. This means that the same slot can be used for multiple time-disjoint customers. We will show that even for a single Swap robot with a single door this problem is \(\text{NP}\)-complete.

### 2.4.1 Lunch Break with a Full Warehouse

In this section, we want to investigate how much it can help the robot if it can reorder the stored boxes at some point to optimize for future requests. To do this, we assume that there is some time frame, we call lunchtime, during which no customer arrives at the door. We further assume that all customers brought there goods before lunch, in the morning, and will reclaim it only afterwards, in the afternoon. Formally, we assume that there exists \(t_{\text{lunch}}\) such that \(\forall i \in [m]: t_i^d < t_{\text{lunch}} < t_i^d\).

If we assume that the robot can perform an arbitrary permutation \(\pi\) during lunchtime, we can compute the two optimal schedules, \(\pi_A\) for the morning and \(\pi_D\) for the afternoon, completely independent of each other and then just let \(\pi = \pi_A^{-1} \circ \pi_D\).

How long it takes a single Swap robot to rearrange the boxes according to \(\pi\) will be the topic of the entire next Chapter 3. For now we just assume that the lunchtime is long enough so that this sorting is isolated from the request serving for any permutation.

What remains is the question of determining a valid schedule for the Swap robot in the afternoon. As the robot now needs to fetch a specific box, the robot needs to allocate time before the customer arrives at the door. The empty box can then be stored anywhere, for instance at the location where

---

**Figure 2.15:** An instance with two blocks each with a single short request. As RANDOM can only serve one of the two requests in two of the four cases, the bound of \(\frac{3}{4}\) on the expected approximation ratio is tight.
Figure 2.16: A small example for the Swap robot in the afternoon setting. The intervals on top denote the time frames that need to be allocated to handle each request, namely to fetch the box before the customer arrives. The plot in the middle tracks the movement of the robot and the cutouts at the bottom show the state of the warehouse at two specific points in time. One feasible way to assign the three customers to storage slots is \( \pi(1) = 3 \), \( \pi(2) = 2 \) and \( \pi(3) = 1 \). \( \pi' = (2, 3, 1) \) would also be a feasible permutation. As the robot now needs to fetch the customer’s box before he arrives, the time requirement extend to the left. There is no longer a constraint to the right, as the empty box can just be stored at the next slot requested (unlike in the morning setting in Figure 2.5).

the next box is fetched from. This is complementary to the arrival-only problem we studied in Section 2.2.3. Formally, this just results in an index shift by one:

**Definition 2.52 (Swap Departure-Only problem)** (Swap Departure) Given a multiset of integers \( \Delta = \{\Delta_0, \Delta_1, \ldots, \Delta_n\} \), denoting the time intervals between subsequent departure requests, the question is to decide whether there is a feasible schedule for the Direct robot, namely a permutation \( \pi \in S_n \) such that \( 2\pi(i) \leq \Delta_{i-1} \) for all \( 1 \leq i \leq n \).

As in the morning setting, this is an instance of the Constrained problem and hence we already know how to find \( \pi \) efficiently. Figure 2.16 illustrates this for the same instance as in Figure 2.5.

### 2.4.2 Single Customer per Slot

We now allow for arrivals and departures to be arbitrarily mixed but assume that the schedule of the robot can still be described by a permutation that maps customers to slots. This means that every slot can be used for at most one customer. So even if two customers arrive and depart in two disjoint time intervals, the robot will not use the same slot for both of them.
2.4. Arrivals and Departures - the Entire Day

<table>
<thead>
<tr>
<th>Arrival</th>
<th>Arrival</th>
<th>Departure</th>
<th>Departure</th>
</tr>
</thead>
<tbody>
<tr>
<td>analogous to SWAPARRIVAL</td>
<td>analogous to SWAPDEPARTURE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Departure</td>
<td>Arrival</td>
<td></td>
<td></td>
</tr>
<tr>
<td>simply wait at the door between the requests</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arrival</td>
<td>Departure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>crucial case analogous to DIRECTARRIVAL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.17: All four pairs of two consecutive request types. In either case the movement of the robot in between the requests either only depends on one of the two slot assignments or on the maximum of them.

Note that we still consider the boxes to be identical and indistinguishable. Because the boxes are not coupled to any fixed slot, we might use the same box to serve multiple requests as we will see later on.

We still assume that there are as many slots as requests \( n = m \), now as there are twice as many requests, we introduce some new notation.

**Definition 2.53 (Ordered request times)** Let us denote the ordered sequence of all request times by \( \tau = \{t_1, \ldots, t_j, \ldots, t_{2m}\} \). The inter request times are then \( \Delta = \{\Delta_0, \Delta_1, \ldots, \Delta_{2m}\} \) with \( \Delta_j = t_{j+1} - t_j \) for \( 1 \leq j < n \) and \( \Delta_0 = \Delta_n = \infty \). We let \( i_a \) and \( i_d \) denote the index of the arrival and departure of the \( i \)-th customer in the order of all requests.

Using this definition, \( \Delta_{i_a} \) denotes the time between the arrival of customer \( i \) and the next request and \( \Delta_{i_d-1} \) corresponds to the request-free time period before the departure of customer \( i \) for instance.

We claim that the interleaving of both request types does not make it much more complicated for the robot to find a valid schedule as it was in the separate morning and afternoon settings. All it needs is enough time to store the full box after an arrival and fetch an empty box before the departure.

To see that consecutive requests do not interfere any more than before, let us consider all possible pairs of consecutive request types. Figure 2.17 illustrates these four cases.

If an arrival is followed by another arrival, the robot only has to consider the first request when thinking about what to do in between those two requests. The same applies if a departure is preceded by another departure. In between a departure and an arrival the robot can just wait at the door and reuse the same box for the next request.
The crucial case is an arrival \( t^a_i \) immediately followed by a departure \( t^d_j \). In between these two requests the robot has to first store a full box. With the empty box he swapped out from there it can then drive straight to the slot for the departure requests, leave its empty box there and return with the full box to the door. The time required for these steps depends only on the more remote of the two slot, as \( 2 \cdot \max(\pi(t^a_i), \pi(t^d_j)) \) is required. Hence the scheduling constraints separate nicely between the requests, just as they did for the Direct robot in Section 2.2.2. For customer \( i \) we require \( 2\pi(i) \leq \Delta_{iA} \) to store the box and \( 2\pi(i) \leq \Delta_{iD} - 1 \) to fetch it.

**Definition 2.54 (Swap All-Day Personal-Slot problem) (SwapPERSONALSLOT)**

Given the request times \( \tau \) and derived from it \( \Delta = \{\Delta_0, \Delta_1, \ldots, \Delta_{2m}\} \), the question is to decide whether there is a feasible schedule for the Swap robot, namely a permutation \( \pi \in S_n \) such that \( 2\pi(i) \leq \min(\Delta_{iA}, \Delta_{iD} - 1) \) for all \( 1 \leq i \leq n \).

This is an instance of the CONSTRAINED problem, so we can find \( \pi \) efficiently.

### 2.4.3 Hardness for Reusable Slots

In this final section we allow for the same slot to be used to serve multiple customers as long as these customers do not use the warehouse at the same time.

**Definition 2.55 (Time Conflict Set)** Given \( \tau \) and \( \Delta \), we define the set of conflicting requests as \( \mathcal{C}(\tau) = \{(i, j) \mid [t^a_i, t^d_i] \cap [t^a_j, t^d_j] \neq \emptyset\} \) and we say that customers \( i \) and \( j \) overlap if \( (i, j) \in \mathcal{C}(\tau) \).

Until now, we always assumed that there are equally many slots as customers \( (n = m) \) because any instance with more customers than slots \( (n < m) \) would clearly not have enough slots to serve all requests and having fewer customers \( (n > m) \) would have made that problem just easier. But now we can potentially serve more customers than there are storage slots if they do not all use the storage system at the same time.

**Definition 2.56 (Swap All-Day Slot-Reuse problem) (SwapSlotREUSE)** Given the number of slots \( n \), the request times \( \tau \), and derived from it \( \Delta = \{\Delta_0, \Delta_1, \ldots, \Delta_{2m}\} \), the question is to decide whether there is a feasible schedule for the Swap robot, namely a function \( \pi : [m] \to [n] \) such that \( 2\pi(i) \leq \min(\Delta_{iA}, \Delta_{iD} - 1) \) for all \( 1 \leq i \leq n \) and \( \pi(i) \neq \pi(j) \) for all \( (i, j) \in \mathcal{C}(\tau) \).

Figure 2.18 gives an instance with a valid schedule for the SwapSlotREUSE problem where the slot reuse is really necessary\(^1\), meaning that there is no

---

\(^1\)Note that in this slot reuse setting, a robot could technically assign also slightly concurrent requests to the same slot. If a customer reclaims his box that was previously stored at slot 3 and a new customer arrives 6 seconds before that old customer, the robot could take the box of the new customer and swap it directly with the box of the old customer. We do not consider such optimizations here.
2.4. Arrivals and Departures - the Entire Day

Figure 2.18: An example of SwapSlotReuse with 3 customers. Looking at the request times we get the constraints: \( \pi(1) \leq 2, \pi(2) \leq 2 \) and \( \pi(3) \leq 3 \). Note that this instance would clearly not have a feasible schedule in the SwapPersonalSlot setting since a potential third slot is not reachable for any customer. Also note that our greedy approach from ConstrainedGreedy would also fail, as it would start with \( \pi(1) = 2, \pi(2) = 1 \) and then get stuck for the third customer. Looking at the interleaving of the requests we get the conflict set \( C = \{(1,2), (2,3)\} \) meaning that customer 2 needs to get a different slot than customers 1 and 3. Therefore, the only feasible schedule is \( \pi(1) = 1, \pi(2) = 2, \pi(3) = 1 \), which is illustrated above. The curve shows the position of the robot over time with the black circles marking swaps and the white circles marking interactions with a customer at the door.

valid schedule for this instance in the SwapPersonalSlot setting.

We will now show that this greater flexibility of the SwapSlotReuse problem makes it hard to decide whether a feasible schedule exists.

**Definition 2.57 (CircularArcColouring)** Given an integer \( k \) and a set of \( l \) arcs \( I = \{(a_1,b_1), \ldots, (a_l,b_l)\} \) over the unit circle \([0, 2\pi]\), the question is to decide whether we can assign one of \( k \) colors to every arc such that no two overlapping arcs get the same color. In other words: Is the intersection graph \( G \) corresponding to \( I \) \( k \)-colourable?

Unlike colouring intervals on a path, which can be done in polynomial time using an earliest deadline first scheduling, colouring intervals on a circle turned out to be a hard problem. This was shown by Garey et al. [27].

**Theorem 2.58** [27] CircularArcColouring is \( \mathcal{NP} \)-complete.

**Theorem 2.59** SwapSlotReuse is at least as hard as CircularArcColouring.

**Proof** We give a reduction \( \rho \) which transforms every instance of CircularArcColouring into an equivalent instance of SwapSlotReuse. Figure 2.19 will illustrate this reduction.

Given \( k \) and \( I \) let \( I_0 \) be the set of arcs that contain the point 0. Let \( k' = \|I_0\| \). If \( k' > k \), \( I \) is clearly not \( k \) colourable and we let \( \rho(I) \) be any instance of SwapSlotReuse without a feasible schedule.
2. Request Handling

Figure 2.19: The transformation $\rho$ of an instance of CircularArcColouring into an instance of SwapSlotReuse. We have six arcs in $\mathcal{I}$ and want to color them with $k = 3$ colors. The point $0$ is covered by the arcs $I_0 = \{I_1, I_2\}$, hence $k' = 2$. The requests for the corresponding SwapSlotReuse on the right are spaced such that customers $c_{1,\alpha}$ and $c_{1,\beta}$ both only can use slot 1. The customers $c_{2,\alpha}$ and $c_{2,\beta}$ can both use the first two slots. As these two customers overlap with either $c_{1,\alpha}$ or $c_{1,\beta}$, they both have to take slot 2. Hence we ensure that the assignment is consistent across the cut of the circle. All other requests are spaced $k = 3$ seconds apart from each other such that any free slot can be reached by the robot and all requests appear in the order of the starting and ending points of the arcs on the circle.

If $k' \leq k$ we create the following instance of $2l + 2k'$ requests. Every arc $I$ of $\mathcal{I} \setminus \mathcal{I}_0$ will correspond to one customer $C_i$ of the warehouse and these customers will arrive at the door in the order that their intervals start and end on the circle in clockwise-order. The arc $I$ in $\mathcal{I}_0$ will get split into two customers $c_{1,\alpha}$ and $c_{1,\beta}$, one in the early morning and one in the late evening. We will use the timing constraints to ensure that those two customers always get assigned the same color.

Let us formalize this idea. For this let $f(x)$ denote the zero-based index of the event (start or end) at position $x$ on the circle in clockwise order when starting at position 0. Without loss of generality, we can assume that no two arcs in $\mathcal{I}$ start or end at the exact same place on the circle because changing them slightly in the right direction has no effect on the intersection graph $G$. Therefore $f$ is a total order of all the interval endpoints and assigns each of them a number from 0 to $2l - 1$. Also let $o(I)$ denote an arbitrary 1-based order of the $k'$ intervals in $\mathcal{I}_0$.

Our mapping $\rho$ generates $2l$ requests that correspond to the events on the circle in the same order as $f$. These requests have a time period of $k$ between each other and we set the first request to time 0 and the last request to time 36.
2.5. Related Work

Automated storage and retrieval systems were studied in a variety of work in operational and theoretical research. Much work was done in the context

\[ k \cdot (2l - 1) \]

We call this the circle time segment. Additionally, \( \rho \) generates \( k' \) requests on both the left and the right end of the circle time segment. On each side, the \( k' \) requests are at distance 1, 2, 3 up to \( k' \) from each other and the circle time segment. These \( 2k' \) requests will allow us to enforce a consistent assignment of the requests corresponding to the \( k' \) intervals in \( I_0 \). Both of these consistency time segments are of duration \( \sum_{i=1}^{k'} i \).

For an arc \( I = (a, b) \), we now specify \( \rho(I) \), its mapping to request times. If \( I \) does not contain position 0, we map it to a single request pair in the circle time segment, namely a customer \( c_I \) that arrives at time \( k \cdot f(a) \) and departs at time \( k \cdot f(b) \). If \( I \) does contain position 0, so if \( I \in I_0 \), we map this arc to two customers \( c_{I, \alpha} \) and \( c_{I, \beta} \). \( c_{I, \alpha} \) arrives in the first consistency time segment as the \( o(I) \)-th request in that segment and departs as the \( f(a) \)-th request in the circle time segment. Symmetrically, \( c_{I, \beta} \) arrives as the \( f(b) \)-th request in the circle time segment and departs as the \( o(I) \)-th request in the second consistency time segment. Formally,

\[
\rho(I) = \rho((a, b)) = \begin{cases} 
(c_I = (k \cdot f(a), k \cdot f(b)) & \text{if } I \notin I_0 \\
(c_{I, \alpha} = (\sum_{i=0}^{k'} i, k \cdot f(b)) \text{ and } (k \cdot f(a), (2l - 1) + \sum_{i=1}^{o(I)} i)) & \text{if } I \in I_0.
\end{cases}
\]

(2.17)

We let \( \tau \) be the sorted union of all these request times and set \( n = k \) to get an instance of the SwapSlotReuse problem.

It is clear that the set of constraints \( C(\rho(I)) \) corresponds to the intersection graph \( G \). The colors/slots 1 to \( k' \) have to be assigned to the intervals in \( I_0 / \text{customers in } \rho(I_0) \) in the arbitrary order \( o \). For all other requests the request times generate by our reduction \( \rho \) only enforces that at most slot \( k \) is used. So a \( k \)-colouring for \( I \) directly corresponds to a feasible schedule for \( \rho(I) \) and vice versa.

**Corollary 2.60** SwapSlotReuse is \( NP \)-complete.

**Proof** SwapSlotReuse is in \( NP \), as it is easy to verify whether a given \( \pi \) form a feasible schedule as in Definition 2.56. As SwapSlotReuse is at least as hard as CircularArcColouring by Theorem 2.59 and CircularArcColouring is \( NP \)-complete by Theorem 2.58, we have that SwapSlotReuse is also \( NP \)-complete. \( \square \)

2.5 Related Work

Automated storage and retrieval systems were studied in a variety of work in operational and theoretical research. Much work was done in the context
of container terminals where cargo vessels need to get allocated to berths and quay cranes want to move the containers efficiently. We refer to Bierwirth and Meisel [9] for an overview. Another line of research considers operation planning in warehouses which was reviewed by Gu et al. [32].

2.5.1 Similar Problems on Finding Permutations

A particularly interesting problem from warehouse operations research is order picking. When given the layout of a warehouse and a list of items to collect, a worker wants to use the fastest way to pick this order, so to visit the locations of all the items on the list. The lecture notes of Bartholdi and Hackman [6] give a nice introduction and De Koster et al. review relevant publications [19]. Formulated as a graph theoretical problem, order picking translates to the Steiner travelling salesman problem. Ratliff and Rosenthal [49] give an efficient algorithm for typical warehouse layouts that consist of a sequence of aisles with crossovers at both ends of these aisles, which correspond to subdivisions of ladder graphs. Compared to the models we studied in this chapter, these results correspond to a departure-only setting on a non-linear layout with a robot that can carry multiple boxes and should bring a given set of boxes to the door.

For a single on a line, an interesting problem is the travelling repairman problem. Given the positions of \( n \) customers on this line, we want to find a permutation of the customers that minimizes the total wait time if the robot visits the customers in this order. The robot is initially placed in between these customers and has to decide whether to first go to the left or to the right. Afrati et al. [3] give a quadratic dynamic programming solution and show that the problem gets \( \mathcal{NP} \)-complete once we add an individual deadline for each customer by which he has to get served. Our arrival-only problem differs in two aspects; The order of the requests is given but the robot is free to choose the storage slots he visits for each of these requests. Furthermore, the robot always has to return to the special location of the door where all the customers arrive.

Searching permutations with certain properties was studied for many different constraints. A similar formulation to what we studied in the arrival-only setting in Section 2.2 is the problem by De Biasi [8] of reconstructing a permutation when given the absolute differences of consecutive entries. This corresponds to our TightSum problem with the sum replaced by the absolute difference and can hence be formulated as the following decision problem.

**Definition 2.61 (AbsoluteDifference)** Given a sequence of integers \( x_1, \ldots, x_{n-1} \), is there \( \pi \in S_n \) such that \( |\pi(i) - \pi(i+1)| \leq x_i \) for all \( 1 \leq i < n \)?
De Biasi showed that this problem is \( \mathcal{NP} \)-complete by giving a polynomial time reduction from the Hamiltonian path problem on grid graphs [36].

### 2.5.2 Interval Graph Colouring

Related to our \textsc{SwapSlotReuse} problem in Section 2.4.3, where we allow arbitrarily interleaved arrivals and departures, is the topic of colouring interval graphs. Every customer corresponds to an interval and every slot corresponds to a colour. Overlapping customers should get assigned to different slots. We want a valid colouring of the corresponding interval graph under the additional constraints given in Definition 2.56, which ensure that the robot is always back at the door in time.

If the robot had infinite velocity, so he could store and retrieve boxes instantaneously, the minimal number of slots needed would just correspond to the chromatic number of the interval graph. For interval graphs, the chromatic number equals its clique number and an optimal colouring can be efficiently determined using a greedy colouring in the \textit{earliest start time first} order.

There are several \( \mathcal{NP} \)-hard variations of the interval colouring problem. We present three such variations which all were proven \( \mathcal{NP} \)-hard using a reduction from \textsc{CircularArcColouring} [27]. All of these proofs use the same idea we used in Theorem 2.59 of cutting the circle by splitting all intervals across a certain point into two parts.

- Biró et al. [10] showed the \( \mathcal{NP} \)-completeness of \textit{precolouring extension}, where a subset of the intervals already has a fixed colour assigned. They show that it is hard even if we fix at most two intervals for each color, since after cutting the circle some pairs of two intervals need to be forced to stay consistent.

- Bonomo et al. [12] showed the \( \mathcal{NP} \)-hardness of \( \mu \)-colouring on interval graphs, where each interval has an individual upper bound on the colour it can take. This problem is most similar to our setting where the time between two request imposes an upper bound on the slot assignment of the arrival request before and the departure request after this time period.

- Marx [48] gave a simpler proof than the earlier proof by Szkaliczki [54] for the \( \mathcal{NP} \)-hardness of \textit{sum colouring} where the sum of the assigned colors (as integers from 1 to \( k \)) should be minimized. In our setting, this would correspond to a cost function where the cost of assigning a customer to a certain slot is proportional to the distance of the slot to the door, so for instance if we want to minimize the distance travelled by an infinitely fast robot.
Chapter 3

Rearrangement

In Chapter 2, we saw that being able to rearrange the boxes in storage can significantly simplify the scheduling of arrival and departure requests. In this chapter, we teach our robot to rearrange the boxes efficiently when faced with such a physical sorting task.

In contrast to standard sorting algorithms, the robot does not have constant time access to the stored objects. It might need to travel for a significant amount of time before fetching the object in question, and then moving it to its desired location also takes time. We want to look at the problem of finding the most efficient route for the robot that allows it to permute the stored objects.

We assume that all the boxes are already in storage and no customers interact with the storage system for the time required for this sorting process. We generalize the layout of the warehouse from a path to general graphs.

Overview  Consider a graph $G$ with $n$ vertices. On each vertex, we place a box. These $n$ vertices and $n$ boxes are both numbered from 1 to $n$ and initially shuffled according to a permutation $\pi \in S_n$. We introduce a sorting problem for a single robot: In every step, the robot can walk along an edge of $G$ and can carry at most one box at a time. At a vertex, it may swap the box placed there with the box it is carrying. How many steps does the robot need to sort all the boxes?

We present an algorithm that produces a shortest possible sorting walk for such a robot if $G$ is a tree. The algorithm runs in time $O(n^2)$ and can be simplified further if $G$ is a path. We show that for planar graphs the problem of finding a shortest possible sorting walk is $\mathcal{NP}$-complete. We also show that if we want to minimize the number of swaps as a first priority the problem is $\mathcal{NP}$-complete even for tree graphs.
Organization Section 3.1 formally defines the problem and introduces some terminology. We then study the problem of minimizing the length of the shortest sorting walk. In Section 3.2.1, we show some first lower and upper bounds on general graphs. Section 3.2.2 shows a way of finding shortest sorting walks on path graphs where the robot starts at one of the ends of the path. Our main result is given in Section 3.2.3, where we efficiently construct shortest sorting walks on arbitrary trees with arbitrary starting position. We apply these results in Section 3.2.4 to find shortest sorting walks on cycle graphs. In Section 3.2.5, we show that it is $\mathcal{NP}$-complete to find a shortest sorting walk for planar graphs. In Section 3.3, we look at the problem of minimizing the number of box handling interactions as a first priority while still minimizing the travel time. We see that this sorting problem remains solvable on path graphs but is $\mathcal{NP}$-complete on tree graphs. Section 3.4 summarizes related work on physical sorting. We give solutions to two problems of recent programming competitions that have a very similar flavour.

3.1 Problem Description and Notation

We consider the following model throughout this chapter. Our warehouse holds $n$ boxes. Each box is unique in its content but all the boxes have the same dimensions and can be handled the same way. The storage locations and aisles of the warehouse are represented by a connected graph $G = (V, E)$, where $n = |V|$ and $m = |E|$. Every vertex $v \in V$ represents a location that can hold a single box. Every edge $e = (u, v) \in E$ represents a bidirectional aisle between two locations. We assume that our warehouse is full, meaning that at each location there is exactly one box stored initially. The boxes and locations are numbered from 1 to $n$ and are initially shuffled according to some permutation $\pi \in S_n$, representing that the box at vertex $i$ should get moved to vertex $\pi(i)$. The robot is initially placed at a vertex $r$ and does not carry a box. In every step, the robot can move along a single edge. It can carry at most one box with it at any time. When arriving at a vertex, it can either

- put down the box it was travelling with if there is no box at this vertex.
- pick up the box from the current vertex if it arrived without carrying a box.
- swap the box it was carrying with the box at this vertex if there is one.
- or do nothing.

We refer to each travelled edge of the robot as a step of the sorting process. A sequence of steps that lets the robot sort all the boxes according to $\pi$ and return to $r$ is called a sorting walk. We measure the length of a sorting walk as the number of edges that the robot travels along. Therefore, we assume
that all aisles are of equal length and that all of the box-handling actions (pickup, swap, putdown) only take a negligible amount of time compared to the time spent travelling along the edges. In Section 3.2, we look for the shortest sorting walk. To address the fact that the box-handling actions are not negligible in practice, we also count the number of times a box is loaded or unloaded onto the robot. In Section 3.3, we minimize the number of these actions as a first priority while still minimizing the length of the sorting walk as a second priority.

**Example**  Figure 3.1 shows an example of a warehouse where $G$ is a tree consisting of 8 vertices. It is not obvious how we can find a short walk that allows the robot to sort these 8 boxes. We will see an efficient algorithm that produces such a sorting walk and we will prove that this sorting walk has minimum length.

**Notation**  Formally, we describe the state $\tau$ of the warehouse by a triple $(v, b, \sigma)$ where $v \in V$ is the current position of the robot, $b \in \{1, \ldots, n\} \cup \{\Box\}$ is the number of the box that the robot is currently travelling with or $\Box$ if it is travelling without a box, and $\sigma$ is the current mapping from vertices to boxes. If there is no box at some vertex $i$, we will have $\sigma(i) = \Box$. At any point, there will always be at most one vertex without a box, thus at most one number will not appear in $\{\sigma(i) \mid i \in \{1, \ldots, n\}\}$. In other words: Looking at $\sigma$ and $b$ together will at all times be a permutation of $\{1, \ldots, n\} \cup \{\Box\}$. Given the current state, the next step $s$ of the robot can be specified by the pair $(p, b)$, if the robot moves to $p \in V$ with box $b \in \{1, \ldots, n\} \cup \{\Box\}$.

We start with $\tau_0 = (r, \Box, \pi)$, so the robot is at the starting position and is not carrying a box. Applying a step $s_t = (p, b)$ to a state $\tau_{t-1} = (v_{t-1}, b_{t-1}, \sigma_{t-1})$
transforms it into the state $\tau_t = (v_t, b_t, \sigma_t)$ with $v_t = p, b_t = b$. $\sigma_t$ only differs from $\sigma_{t-1}$ if a swap was performed, so if $b_{t-1} \neq b$, in which case we set $\sigma_t(v_{t-1}) = b_{t-1}$. In order to get $\sigma = id$ in the end, we let the robot put its box down whenever it moves into an empty location. Thus if $\sigma_{t-1}(p) = \square$, we let $b_t = \square$ and $\sigma_t(p) = b$.

Step $s_t$ is valid only if $(v_{t-1}, p) \in E$ and $b \in \{b_{t-1}, \sigma_{t-1}(v_{t-1})\}$, enforcing that the robot moves along an edge of $G$ and carries either the same box as before or the box that was located at the previous vertex. Thus after putting down a box at an empty location, the robot can either immediately pick it up again or continue without carrying a box. A sequence of steps $S = (s_1, \ldots, s_t)$ is a sorting walk of length $l$ if we start with $\tau_0$, all steps are valid, and we end in $\tau_l = (r, \square, id)$. We are looking for the minimum $l$ such that a sorting walk of length $l$ exists.

**Definition 3.1 (GraphSort)** Given a graph $G = (V, E)$, a starting vertex $r \in V$, and a permutation $\pi$ on $V$, we let GraphSort denote the problem of finding a shortest sorting walk to sort $\pi$ on $G$ with the robot starting at vertex $r$.

We denote the set of cycles of the permutation $\pi$ by $C = \{C_1, \ldots, C_{||C||}\}$, where each cycle $C_i$ is an ordered list of vertices $C_i = (v_{i, 1}, \ldots, v_{i, ||C||})$ such that $\pi(v_{i, j}) = v_{i, j+1}$ for all $j < ||C_i||$ and $\pi(v_{i, ||C||}) = v_{i, 1}$. In the example shown in Figure 3.1, we have $C = \{(1, 4), (2), (3, 7, 5), (6, 8)\}$. As cycles of length one represent boxes that are placed correctly from the beginning, we usually ignore such trivial cycles and let $\overline{C} = \{C \in C \mid ||C|| > 1\}$ be the set of non-trivial cycles.

Let $d(u, v)$ denote the distance (length of the shortest path) from $u$ to $v$ in $G$. So if the robot wants to move a box from vertex $u$ to vertex $v$, it needs at least $d(u, v)$ steps for that. By $d(C)$, we denote the sum of distances between all pairwise neighbours in the cycle $C$ and by $d(\pi)$ the sum of all such cycle distances for all cycles in $\pi$, i.e., $d(\pi) = \sum_{C \in \overline{C}} d(C) = \sum_{v \in V} d(v, \pi(v))$. In the example shown in Figure 3.1, we have

$$d(\pi) = d((1, 4)) + d((2)) + d((3, 7, 5)) + d((6, 8)) = d(1, 4) + d(4, 1) + d(2, 2) + d(3, 7) + d(7, 5) + d(5, 3) + d(6, 8) + d(8, 6) = 2 + 2 + 0 + 2 + 5 + 3 + 1 + 1 = 16.$$

### 3.2 Minimizing the Travel Time

#### 3.2.1 General Bounds

We distinguish two kinds of steps in a sorting walk: essential and non-essential steps.
### 3.2. Minimizing the Travel Time

**Definition 3.2 (Essential steps)** A step \( s = (p, b) \) is essential if it brings box \( b \) one step closer to its target position than it was in any of the previous states, so if \( d(p, b) \) is smaller than ever before. We say that such a step is essential for a cycle \( C \) if \( b \in C \).

A single step can be essential for at most one cycle, as at most one box is moved in a step and each box belongs to exactly one cycle. In the example in Figure 3.1 for instance, the first step was essential for cycle \((1, 4)\). Overall, 16 steps (all but \( s_2 \) and \( s_{15} \)) were essential. This corresponds to the sum of distances of all boxes to their targets \( d(\pi) \), which we formalize as follows.

**Lemma 3.3 (Lower bound by counting essential steps)** Every sorting walk for a permutation \( \pi \) on a graph \( G \) has length at least \( d(\pi) = \sum_{b \in \{1, \ldots, n\}} d(b, \pi(b)) \).

**Proof** Throughout any sorting walk, there will be exactly \( d(b, \pi(b)) \) essential steps that move box \( b \). As the robot cannot move more than one box at a time, the sum of distances between all boxes and their target positions can decrease by at most 1 in each step. Therefore, there will be \( d(\pi) = \sum_{b \in \{1, \ldots, n\}} d(b, \pi(b)) \) essential steps in every sorting walk and at least as many steps overall.

The remaining challenge is to minimize the number of non-essential steps. In case that \( \pi \) consists only of a single cycle, the shortest solution is easy to find. We just pick up the box at \( r \) and bring it to its target position \( \pi(r) \) in \( d(r, \pi(r)) \) steps. We continue with the box at \( \pi(r) \), bring it to \( \pi(\pi(r)) \) and so on until we return to \( r \) and close the cycle. Therefore, by just following this cycle, the robot can sort these boxes in \( d(\pi) \) steps without any non-essential steps. As it brings one box one step closer to its target position in every step, by Lemma 3.3 no other sorting walk can be shorter.

But what if there is more than one cycle? One idea could be to sort each cycle individually one after the other. This might not give a shortest possible sorting walk, but it might give a reasonable upper bound. So the robot picks up the box at \( r \), brings it to its target, swaps it there, continues with that box and repeats this until it closes the cycle. After that, the robot moves to any box \( b \) that is not placed at its correct position yet. These steps will be non-essential as the robot does not carry a box during these steps from \( r \) to \( b \). Once it arrives at \( b \), it sorts the cycle in which \( b \) is contained. In this way, it sorts cycle after cycle and finally returns to \( r \). The number of non-essential steps in this process depends on the order in which the cycles are processed and which vertices get picked to start the cycles. The following lemma shows that a linear amount of non-essential steps will always suffice.

**Lemma 3.4 (Upper bound from traversal)** There is a sorting walk of length at most \( d(\pi) + 2 \cdot (n - 1) \) for a permutation \( \pi \) on a graph \( G \).
3. Rearrangement

**Proof** We let the robot do a depth-first search traversal of \( G \) while not carrying a box. Whenever we encounter a box that is not placed correctly yet, we sort its entire cycle. As the robot returns to the same vertex at the end of the cycle, we can continue the traversal at the place where we interrupted it. Recall that \( G \) is connected, so during the traversal we will visit each vertex at least once and at the end all boxes will be at their target position. The number of non-essential steps is now given by the number of steps in the traversal which is twice the number of edges of the spanning tree produced by the traversal. \( \square \)

We can see that these sorting walks might not be optimal, for instance in the example shown in Figure 3.1. Every sorting walk that sorts only one cycle at a time will have length at least 20, while the optimal solution consists of only 18 steps. Hence it might be possible to reduce the number of non-essential steps by interleaving the sorting of several cycles.

As \( d(\pi) \) can grow quadratically in \( n \), the linear gap between the upper and lower bound might already be considered negligible. However, for the rest of this paper we want to find sorting walks that are as short as possible.

### 3.2.2 Sorting on a Path

We now look at the case where \( G \) is the path graph \( P = (V, E) \). Imagine that the vertices \( v_1 \) to \( v_n \) are ordered on a line from left to right and every vertex is connected to its left and right neighbour, thus \( E = \{ \{v_i, v_{i+1}\} \mid i \in \{1, \ldots, n-1\} \} \). We further assume that the robot is initially placed at one of the ends of the path, so let \( r = v_1 \).

**Definition 3.5 (PathSort)** We let \( \text{PathSort} \) denote the instances of \( \text{GraphSort} \) where the graph \( G \) is a path.

**Definition 3.6 (BorderPathSort)** We let \( \text{BorderPathSort} \) denote the instances of \( \text{PathSort} \) where the robot starts at one of the ends of the path.

By \( I(C) = [l(C), r(C)] \), we denote the interval of \( P \) covered by the cycle \( C \), where \( l(C) = \min_{v_i \in C} i \) and \( r(C) = \max_{v_i \in C} i \). We say that two cycles \( C_1 \) and \( C_2 \) intersect if their intervals intersect. Now let \( \mathcal{I} = (\mathcal{C}, \mathcal{E}) \) be the intersection graph of the non-trivial cycles, so \( \mathcal{E} = \{ \{C_1, C_2\} \mid C_1, C_2 \in \mathcal{C} \text{ s.t. } I(C_1) \cap I(C_2) \neq \emptyset \} \). We then use \( \mathcal{D} = \{D_1, \ldots, D_{||\mathcal{D}||}\} \) to represent the partition of \( \mathcal{C} \) into the connected components of this intersection graph \( \mathcal{I} \). Two cycles \( C_1 \) and \( C_2 \) are in the same connected component \( D_i \in \mathcal{D} \) if and only if there exists a sequence of pairwise-intersecting cycles that starts with \( C_1 \) and ends with \( C_2 \). We let \( l(D) = \min_{C \in D} l(C) \) and \( r(D) = \max_{C \in D} r(C) \) be the boundary vertices of the connected component \( D \). We index the cycles and components from left to right according to their leftmost vertex, so that \( l(C_i) < l(C_j) \) and \( l(D_i) < l(D_j) \) whenever \( i < j \).
3.2. Minimizing the Travel Time

Theorem 3.7 (Shortest sorting walk for BorderPathSort) The shortest sorting walk on a path \( P \) with permutation \( \pi \) and starting position \( r = v_1 \) can be constructed in time \( \Theta(n^2) \) and has length

\[
d(\pi) + 2 \cdot \left( l(D_1) - 1 + \sum_{i=1}^{\|P\|-1} (l(D_{i+1}) - r(D_i)) \right).
\]  

(3.1)

Proof We claim that the number of non-essential steps that are needed is twice the number of edges that are not covered by any cycle interval, and lie between \( r \) and the rightmost box that needs to be moved.

We prove the claim by induction on the number of non-trivial cycles of \( \pi \). We already saw how we can find a minimum sorting walk if \( \pi \) consists of a single cycle only. If there are several cycles but only one of them is non-trivial, so \( \|C\| > 1 \) but \( \|\overline{C}\| = 1 \), the shortest sorting walk is also easy to find: we walk to the right until we encounter the leftmost box of this non-trivial cycle \( C \), then we sort \( C \) and return to \( r \). The number of steps is \( d(\pi) + 2 \cdot (l(C) - 1) \) and is clearly optimal. Figure 3.2 (left) gives an example of such a case.

Now let us look at the case where \( \pi \) consists of exactly two non-trivial cycles \( C_1 \) and \( C_2 \). If \( C_1 \) and \( C_2 \) intersect, we can interleave the sorting of the two cycles without any non-essential steps. We start sorting \( C_1 \) until we first encounter a box that belongs to \( C_2 \), so until the first step \((p, b)\) where \( p \in C_2 \). This will happen eventually, as we assumed that \( C_1 \) and \( C_2 \) intersect. We then leave box \( b \) at position \( p \) in order to sort \( C_2 \). After sorting \( C_2 \), we will be back at position \( p \) and can finish sorting \( C_1 \), continuing with box \( b \). As we will end in \( l(C_1) \) and then return to \( v_1 \), we found a minimum walk of length \( d(\pi) + 2 \cdot (l(C_1) - 1) \). Figure 3.2 (centre) gives an example of such a case.

Let us assume that \( C_1 \) and \( C_2 \) do not intersect. This implies that there is no box that has to go from the left of \( r(C_1) \) to the right of \( l(C_2) \) and vice versa. But the robot still has to visit the vertices of \( C_2 \) at some point and then get back to the starting position. So each of the edges between the two cycles will be used for at least two non-essential steps. We construct a sorting walk that achieves this bound of \( d(\pi) + 2 \cdot (l(C_1) - 1 + l(C_2) - r(C_1)) \). We start by sorting \( C_1 \) until we get to \( r(C_1) \). We then take the box \( \pi(r(C_1)) \) from there and walk with it to \( l(C_2) \). From there we can sort \( C_2 \) starting with box \( \pi(l(C_2)) \). We again end at \( l(C_2) \), where we can pick up box \( \pi(r(C_1)) \) again and take it back to position \( r(C_1) \). From there, we finish sorting \( C_1 \) and return back to \( v_1 \). Figure 3.2 (right) gives an example of such a case.

Next, let us assume that we have three or more non-trivial cycles. We look at these cycles from left to right and we assume that by induction we already found a minimum sorting walk \( S_i \) for sorting the boxes of the first \( i \) cycles.
3. Rearrangement

![Diagram showing cycles and sorting walks](image-url)

**Figure 3.2:** (left) An example with a single non-trivial cycle. A shortest sorting walk $S$ with $\|S\| = d(\pi) + 2 \cdot l(D_1) = 8 + 2 \cdot (2 - 1) = 10$ is $(2, \square, (3, 5), (4, 5), (5, 5), (4, 3), (3, 3), (4, 4), (3, 2), (2, 2), (1, \square))$. (centre) An example with two intersecting cycles. A shortest sorting walk $S$ with $\|S\| = d(\pi) = 10$ is $(2, 3), (3, 5), (4, 5), (4, 4), (3, 2), (2, 2), (3, 3), (2, 1), (1, 1))$. (right) An example with two non-intersecting cycles. A shortest sorting walk $S$ with $\|S\| = d(\pi) + 2 \cdot (l(D_2) - r(D_1)) = 4 + 2 \cdot (4 - 2) = 8$ is $(2, 2), (3, 1), (4, 1), (5, 5), (4, 4), (3, 1), (2, 1), (1, 1))$.

For the next cycle $C_{i+1}$ we now distinguish two cases: If $C_{i+1}$ intersects any cycle $C^* \in \{C_1, \ldots, C_i\}$ (which does not necessarily need to be $C_i$), we can easily insert the essential sorting steps for $C_{i+1}$ into $S_i$ at the point where $S_i$ first walks onto $l(C_{i+1})$ while sorting $C^*$. As we only add essential steps, this new walk $S_{i+1}$ will still be optimal if $S_i$ was optimal. We have $\|S_{i+1}\| = \|S_i\| + d(C_{i+1}) = \|S_i\| + \sum_{b \in C_{i+1}} d(b, \pi(b))$. In the other case, $C_i$ does not intersect any of the previous cycles. We then know that any sorting walk uses all the edges between $\max_{j \in \{1, \ldots, i\}} r(C_j)$ and $l(C_{i+1})$ for at least two non-essential steps. So if we interrupt $S_i$ at the step where it visits $\max_{j \in \{1, \ldots, i\}} r(C_j)$ to insert non-essential steps to $l(C_{i+1})$, essential steps to sort $C_{i+1}$ and non-essential steps to get back to $\max_{j \in \{1, \ldots, i\}} r(C_j)$ we get a minimum walk $S_{i+1}$. This case occurs whenever $C_{i+1}$ lies in another connected component than all the previous cycles. So if $C_i$ is the first cycle in some component $D_j$, we have $\|S_{i+1}\| = \|S_i\| + d(C_{i+1}) + 2 \cdot (l(D_j) - r(D_{j-1}))$, and so we get exactly the extra steps claimed in the theorem. □

**Algorithmic Construction**

The proof of Theorem 3.7 immediately tells us how we can construct a minimum sorting walk efficiently. Given $P$ and $\pi$ we first extract the cycles of $\pi$ and order them according to their leftmost box, which can easily be done in linear time. We then build our sorting walk $S$ in the form of a linked list of steps inductively, starting with an empty walk. While adding cycle after cycle we keep for every vertex $v$ of $P$ a reference to the earliest step of the current walk that arrives at $v$. We also keep track of the step $s_{\max}$ that reaches the rightmost vertex visited so far.

When adding a new cycle $C$ to the walk, we check whether we stored a step for $l(C)$. If yes, we simply insert the steps to sort $C$ into the walk and update the vertex-references of all the vertices we encounter while sorting $C$ (except for $l(C)$, as there is already an earlier step in the walk that arrives at $l(C)$). If
3.2. Minimizing the Travel Time

If \( l(C) \) was not visited by the walk so far, we insert the necessary non-essential steps into the walk to get from \( s_{\text{max}} \) to \( l(C) \) and back after sorting \( C \). In either case we update \( s_{\text{max}} \) if necessary. The runtime of adding a new cycle to the walk is linear in the number of steps we add. Overall our construction runs in time \( \Theta(n + \|S\|) \subseteq \Theta(n^2) \), so it is linear in the combined size of the input and output and at most quadratic in the size of the warehouse.

**Online Algorithm with Constant Memory**

Before we move on to more complex graphs and arbitrary starting position, we want to answer two more questions about shortest sorting walks on path graphs.

- Does the robot really need to know the entire permutation in advance or can it do equally well if it learns the shuffling of the boxes while it is driving around?

- Does the robot really need to figure out and memorize the entire sorting walk of quadratic size before it even starts moving or can it use a different algorithm that requires him to remember only very little information?

We show that in both questions the latter is the case. Hence also a robot that does not know the shuffling up front and can only store a constant number of box locations is able to sort the permutation almost equally fast.

**Definition 3.8 (Online Sorting on Paths)** We say that a sorting algorithm on a path is working online if it does not know the permutation \( \pi \) up front, but only learns \( \pi(i) \) once it is on vertex \( i \). We still assume that \( n \), the length of the path, and \( r \), the starting position, are known to the robot.

**Definition 3.9 (Constant Memory)** A sorting algorithm uses constant memory if the robot at every state of the sorting process remembers only a constant amount of information in order to decide on its next step to perform.

We will now state an online algorithm that finds the shortest possible sorting walk that we can hope for. It will only use more non-essential steps than an offline algorithm if some boxes nearest to the door opposite of the starting position are already sorted. An offline algorithm can just ignore these boxes, but an online algorithm has to go to the end of the path to ensure that it does not miss a non-trivial cycle\(^1\).

**Theorem 3.10** There is an online algorithm that uses constant memory and finds a sorting walk that is at most \( 2 \cdot (n - r(D_{\|D\|})) \) steps longer than a shortest possible sorting walk.

---

\(^1\)Technically, one could figure out the box on the very last vertex of the path by keeping track of what we see while walking over the other vertices. If we conclude that \( \pi(n) = n \), even an online algorithm would not need to go to this last vertex.
Proof Our algorithm works in two phases: a left-to-right phase and a right-to-left phase. The robot only keeps track of his current position, the largest box number that it encountered so far and the number of correctly placed boxes at the right end of the path that it is aware of. As we will see, this is all the robot needs to remember and therefore it achieves constant memory usage.

In the left-to-right phase, we let the robot walk once from the first to the last vertex of the path and always carry with it the box with the biggest number it has seen so far. After this phase, the robot is at the right end carrying box $n$. We claim that out of the $n - 1$ steps in this phase, all steps inside a connected component of cycle intervals were essential. When going to the right, moving the largest box seen so far is an essential step as long as this box has not passed its target position yet. But this only happens after the robot has reached the right end of a connected component of cycle intervals. It will then move a correctly placed box to the right until it reaches the next connected component or the end of the path. This way, the robot can learn about the permutation while only performing as many non-essential left-to-right steps as the optimum solution from Theorem 3.7 plus some extra non-essential steps for the suffix of the path that was already sorted from the beginning.

After having placed box $n$ correctly, the right-to-left phase starts. In this phase, the robot will find box $n - 1$, put it where it belongs, find box $n - 2$, place it correctly and so on. Due to its memory constraints the robot does not remember where box $n - 1$ is and needs to be careful not to spend unnecessary non-essential steps when walking to the left to find it. The robot can do this by carrying with it in every step to the left the smallest possible box available. Once box $n - 1$ is found it then takes this box and moves it to right to vertex $n - 1$. This minimum-carrying search to the left is then repeated for $n - 2$, $n - 3$ and so on until the robot returns to the start with box 1. Figure 3.4 showcases a run of this constant-memory online algorithm.

Why does this strategy work and does not end up with countless non-essential steps? Clearly, all left-to-right steps in this second phase are essential as they bring boxes straight to their target position. As long as we never carry a box past its target, all right-to-left steps in this phase are also bringing one box one step closer to their target position. As we will see, always carrying the smallest available box with us ensures that it never happens that we take a box beyond its target position. Assume otherwise, so consider the following scenario: While looking for a box $x$ and carrying box $y$ (with $y < x$ since all boxes to the right of $x$ are already sorted) we are reaching slot $y$. If we have not found box $x$ up to this point and $y$ is the smallest box we have encountered on our search so far, we would continue carrying $y$ to left and remove it from its target position again. This can only
3.2. Minimizing the Travel Time

Figure 3.3: Illustration of the argument in Theorem 3.10. The $n - y + 2$ boxes shown in grey would all need to be labelled with a number $\geq y$ which is impossible.

happen if all the boxes at the slots from $y$ to $x$ are all larger than $y$ and none of them is box $x$. But as all boxes to the right of slot $x$ are all already sorted and the only empty slot is slot 1, this is impossible as there are only $x - y$ different boxes remaining for these $x - y + 1$ many slots. Figure 3.3 illustrates this contradiction.

So all right-to-left steps in this second phase will be essential except for those that undo a non-essential step from the left-to-right phase (even though they bring a box closer to its target, these steps are not considered essential as these boxes were equally close to their target before). Therefore, there are equally many non-essential steps in the both phases of our algorithm and the total number of steps is what we claimed

$$d(\pi) + 2 \cdot \left( (l(D_1) - 1) + \sum_{i=1}^{\|D\| - 1} (l(D_{i+1}) - r(D_i)) + (n - r(D_{\|D\|})) \right). \quad (3.2)$$

3.2.3 Sorting on a Tree

So far, we assumed that the robot works on a path and starts at an endpoint of that path. What if the robot starts at an inner vertex of the path? Even with this minimal change from BORDERTORESORT to PATHSORT the problem gets more involved, as it is no longer greedily decidable whether the first move of the robot should go to the left or to the right. Instead of going into the details of this scenario, we now study the more general problem of arbitrary trees with arbitrary starting positions.

Let $T = (V, E)$ be the underlying tree that the warehouse is based upon, let $r \in V$ be the starting vertex and let $T$ be rooted at $r$.

**Definition 3.11 (TreeSort)** We let TREESORT denote the instances of GRAPH-SORT where the graph $G$ is a tree.

**Definition 3.12 (Cycle hitting and covering vertices)** For any cycle $C$ of $\pi$ we say that it hits a vertex $v$ if the box initially placed on $v$ belongs to the cycle $C$. We denote by $V(C)$ the set of vertices hit by $C$. We let $T(C)$ denote the minimum subtree of $T$ that contains all vertices hit by $C$ and we say $C$ covers $v$ for every $v \in T(C)$. 

51
In Figure 3.1 for example, we have $T((3, 5, 7)) = \{1, 2, 3, 5, 6, 7\}$.

Before describing our solution, we will first derive a lower bound on the length of any sorting walk on $T$. We describe how we map each sorting walk to an auxiliary structure called cycle anchor tree that reflects how the cycles of $\pi$ are interleaved in the sorting walk. We then bound the length of the sorting walk only knowing its cycle anchor tree. We give an explicit construction of a sorting walk that shows that this bound is tight. In order to find an optimal solution we first find a cycle anchor tree with the minimum possible bound and then apply the tight construction to get a shortest possible sorting walk.

**Cycle Anchor Trees**

**Definition 3.13** A cycle anchor tree $\tilde{T}$ is a directed, rooted tree that contains one vertex $\tilde{v}_C$ for every non-trivial cycle $C$ of $\pi$ and an extra root vertex $\tilde{r}$.

Given a sorting walk $S$ we construct from it a cycle anchor tree $\tilde{T}$ as follows: We start with $\tilde{T}$ only containing $\tilde{r}$. We go through the essential steps in $S$. If step $s$ is the first essential step for some cycle $C$, we create a vertex $\tilde{v}_C$ in $\tilde{T}$. To determine the parent node of $\tilde{v}_C$ in $\tilde{T}$ we look for the last essential step $s'$.
3.2. Minimizing the Travel Time

Figure 3.5: The two pairs of dashed arrows symbolize boxes that need to be swapped. A shortest path from any \(v \in T(C_1)\) to any \(w \in V(C_2)\) is shown with continuous arrows, three of them being down-steps, so \(c((\tilde{v}_{C_1}, \tilde{v}_{C_2})) = 3\). The anchor vertex \(a\) is the vertex immediately before the first down step. Note that \(c\) is not symmetric as \(c((\tilde{v}_{C_2}, \tilde{v}_{C_1})) = 2\).

in \(S\) before \(s\) and its corresponding cycle \(C'\). We now say that \(C\) is anchored at \(C'\) and add an edge \((\tilde{v}_{C'}, \tilde{v}_C)\) to \(\tilde{T}\). If no such step \(s'\) exists (which only happens for the very first essential step in \(S\)) we use the root \(\tilde{r}\) as the parent of \(\tilde{v}_C\).

We also assign an integer cost to each edge of a cycle anchor tree. For this we call a sorting step a down-step if the robot moves away from the root and an up-step otherwise. The cost \(c\) for an edge between two nodes of \(\tilde{T}\) is now defined as follows: Let \(c((\tilde{v}_{C_1}, \tilde{v}_{C_2}))\) be the minimum number of down-steps on the path from any vertex \(v \in T(C_1)\) to any vertex \(w \in V(C_2)\). Let us fix one such path that minimizes the number of down-steps and let \(v\) and \(w\) be its endpoints. This path, conceptually, consists of two parts: some up-steps towards the root and then some down-steps away from the root. However, note that we never walk down and then up again, as this would correspond to traversing the same edge twice. Let \(a\) be the vertex where this path switches from up-steps to down-steps, also known as the lowest common ancestor of \(v\) and \(w\). We say that \(a\) is an anchor vertex for anchoring \(C_2\) at \(C_1\). For the single edge incident to the root, we have \(c((\tilde{r}, \tilde{v}_C))\) being the minimum number of down-steps on the path from the root to any vertex \(v \in V(C)\). The cost \(c(\tilde{T})\) of an entire cycle anchor tree \(\tilde{T}\) is simply the sum of its edge costs. Figure 3.5 illustrates the definitions and Figure 3.6 gives an example of the transformation from a sorting walk to a weighted cycle anchor tree.

**Theorem 3.14 (Lower bound for trees)** Any sorting walk \(S\) that sorts a permutation \(\pi\) on a tree \(T\) and corresponds to a cycle anchor tree \(\tilde{T}\) has length at least \(d(\pi) + 2 \cdot c(\tilde{T})\).

**Proof** We partition the steps of \(S\) into three sets: essential steps \(S_e\), non-essential down-steps \(S_{n,d}\) and non-essential up-steps \(S_{n,u}\). From Lemma 3.3 we have \(\|S_e\| = d(\pi)\). We argue that \(S\) contains at least \(c(\tilde{T})\) many non-essential down-steps. To do this we look at the segments of \(S\) that were
relevant when we described how we derive $\overline{T}$ from $S$. For an edge $(\overline{v}_{C_1}, \overline{v}_{C_2})$ of $\overline{T}$, we look for the segment $S_{C_1,C_2}$ of $S$ between the first essential step $s_2$ of $C_2$ and its most recent preceding essential step $s_1$ for some other cycle $C_1$. What do we know about $S_{C_1,C_2}$? First of all, we know that $s_1$ is essential for $C_1$, so $s_1$ ends at a vertex covered by $C_1$ and $S_{C_1,C_2}$ starts somewhere in $T(C_1)$. Next, $s_2$ is the first essential step that moves a box of $C_2$. Note that some or even all of the boxes of $C_2$ might have been moved in non-essential steps before $s_2$, putting them further away from their target position. But as we are on a tree (where there is only a single path between any pair of points), the first time a box gets moved closer to its target position than it was originally is a move away from its initial position, which means that $s_2$ starts at a vertex hit by $C_2$. So $S_{C_1,C_2}$ ends somewhere in $V(C_2)$. By definition of $c(\overline{v}_{C_1}, \overline{v}_{C_2})$, there are at least $c(\overline{v}_{C_1}, \overline{v}_{C_2})$ many down-steps in $S_{C_1,C_2}$. The same holds for the initial segment $S_{r,d}$. As all these segments of the sorting walk are disjoint, we get that $\|S_{n,d}\| \geq c(\overline{T})$.

Finally we argue that $\|S_{u,l}\| = \|S_{n,d}\|$ to conclude the proof. Consider any edge $e$ of $T$ and count all steps of $S$ that go along $e$. Regardless of whether the steps are essential or non-essential, we know that there must be equally many up-steps and down-steps along $e$, as $S$ is a closed sorting walk and $T$ has no cycles. So for every time we walk down along an edge, we also have to walk up along it once. We see that this equality also holds for the essential up-steps and down-steps along $e$. Along $e$ there will be as many essential up-steps as there are boxes in the subtree below $e$ whose target is in the tree above $e$. As $\pi$ is a permutation, there are equally many boxes that are initially placed above $e$ and have their target in the subtree below $e$. So as the overall number of steps match and the essential number of steps match, also the number of non-essential up-steps and down-steps must be equal along $e$. As this holds for any edge $e$, it also holds for the entire sorting walk. \(\square\)
Note that we did not say anything about where these non-essential up-steps are on S, just that there are as many as there are non-essential down-steps.

Reconstructing a Sorting Walk

We now give a tight construction of a sorting walk of the length of this lower bound.

**Theorem 3.15 (Tight construction)** Given T, π and cycle anchor tree ˜T, we can find a sorting walk of length \( d(\pi) + 2 \cdot c(\tilde{T}) \).

**Proof** We perform a depth-first search traversal of ˜T, starting at ˜r and iteratively insert steps into an initially empty sorting walk S. At any point of the traversal, S is a closed sorting walk that sorts all the visited cycles of the anchor tree. For traversing a new edge of ˜T from ˜v C to ˜v C′, we do the following: Let \( v \) ∈ T(C) and \( w \) ∈ V(C′) be the two vertices that have the minimum number of down-steps between them, as in the definition of the edge weights of ˜T. Let \( a \) denote the anchor vertex on the path from \( v \) to \( w \). Furthermore, let \( s = (a, b) \) be the first step of S that ends in \( a \). Note that such a step has to exist, as \( a \) either lies in \( T(C) \) or on the path from \( v \) to the root and all of these vertices already have been visited by \( S \) if \( S \) sorts \( C \). We now build a sequence \( S_{C'} \), which consists of three parts: We first take the box \( b \) from \( a \) to \( w \), then sort \( C' \) starting at \( w \) and finally bring \( b \) back from \( w \) to \( a \). \( S_{C'} \) will contain exactly \( c(\tilde{v}_C, \tilde{v}_{C'}) \) down-steps in the first part, then \( d(C') \) steps to sort \( C' \), and finally \( c(\tilde{v}_C, \tilde{v}_{C'}) \) up-steps. We insert \( S_{C'} \) into \( S \) immediately after \( s \), making sure that \( S \) now also sorts \( C' \) and is still a valid sorting walk. After the traversal of all cycles in the anchor tree, \( S \) will sort \( \pi \) and be of length \( d(\pi) + 2 \cdot c(\tilde{T}) \). □

Note that the sorting walk \( S \) constructed this way does not necessarily map back to \( \tilde{T} \), but its corresponding cycle anchor tree has the same weight as \( \tilde{T} \).

Finding a Cheapest Cycle Anchor Tree

Let \( S^* \) denote a shortest sorting walk for T and \( \pi \). Using Theorem 3.15 to find \( S^* \) (or another equally long sorting walk), all we need is its corresponding cycle anchor tree \( \tilde{T}^* \). It suffices to find any cycle anchor tree with cost at most \( c(\tilde{T}^*) \). Especially, it suffices to find a cheapest cycle anchor tree \( \tilde{T}_{\text{min}} \) among all possible cycle anchor trees. We then use Theorem 3.15 to get a sorting walk \( S_{\text{min}} \) from \( \tilde{T}_{\text{min}} \). As \( c(\tilde{T}_{\text{min}}) \leq c(\tilde{T}^*) \) we get

\[
\|S_{\text{min}}\| = d(\pi) + 2 \cdot c(\tilde{T}_{\text{min}}) \leq d(\pi) + 2 \cdot c(\tilde{T}^*) \leq \|S^*\| \tag{3.3}
\]

and therefore \( S_{\text{min}} \) is a shortest sorting walk.

To find this cheapest cycle anchor tree, we build the complete directed graph \( G \) of potential anchor tree edges. Note that the weights of these edges only depend on T and \( \pi \) but not on a sorting walk.
3. **Rearrangement**

**Optimum Branching**  Given this complete weighted directed graph $\tilde{G}$ we find its minimum directed spanning tree rooted at $\tilde{r}$ using Edmond’s algorithm for optimum branchings [22]. A great introduction to this algorithm, its correctness proof by Karp [40] and its efficient implementation by Dijkstra [58] can be found in the lecture notes of Zwick [63]. Combining these results with Theorem 3.15 will now allow us to find shortest sorting walks in polynomial time.

**Theorem 3.16 (Efficient solution for TreeSort)**  For any sorting problem on a tree $T$ with permutation $\pi$, we can find a minimum sorting walk in time $O(n^2)$.

**Proof**  We first extract all the cycles in linear time. We then precompute the weights of all potential cycle anchor tree edges between any pair of cycles or the root. For this we run breadth-first search (BFS) $|\tilde{C}| + 1$ times, starting once with $r$ and once with $T(C)$ for every $C \in \tilde{C}$ and count the number of down-steps along these BFS trees. We also precompute all the anchor points. As we run $O(n)$ many BFS traversals, this precomputation takes time $O(n^2)$.

As an efficient implementation of Edmond’s algorithm allows us to find $\tilde{T}_\text{min}$ in time $O(n^2)$, we can find $S_{\text{min}}$ in time $O(n^2)$ time overall. In every step of the construction in Theorem 3.15, we can find step $s$ in constant time, if we keep track of the first step of $S$ visiting each vertex of $T$. We build $S$ as a linked list of steps in time linear to its length. Thus, as on the path (Theorem 3.7), we can construct $S_{\text{min}}$ in time $Θ(n + ||S||)$ from $\tilde{T}_{\text{min}}$.

Combining these three steps gives an algorithm that runs in time $O(n^2)$.

3.2.4 **Sorting on a Circle**

The logical next step after knowing how to sort any permutation on arbitrary trees is to look at graphs containing cycles. In this section, we study the case where the warehouse is a single cycle, so the circular graph $C_n$ on $n$ vertices. To avoid confusion with the cycles of the permutation $\pi$, we consistently refer to the graph as the *circle graph* $C_n$, even though $C_n$ is often called the *cycle graph* in the literature.\(^2\)

**Definition 3.17 (CircleSort)**  We let **CircleSort** denote the instances of **GraphSort** where the graph $G$ is a circle.

The main goal of this section is to develop an algorithm that efficiently computes a shortest sorting walk on circles. We will do multiple case distinctions which will lead us to instances of **PathSort**, the problem of sorting on paths with non-border starting positions. We will then reuse our algorithm for sorting on trees of Theorem 3.16 to solve these individual cases.

\(^2\)We abuse the notation a bit and use $C_3$ for both the circle graph on 3 vertices and the third cycle in the set of cycles of the permutation $\pi$. Whenever it is not clear from the context, we state it explicitly.
3.2. Minimizing the Travel Time

Figure 3.7: (left) An instance on the circle $C_5$ with two non-trivial cycles in $\pi = (4, 3, 2, 1, 5)$ and $d(\pi) = 6$. The dashed lines illustrate the shortest path for each box. (centre) A non-optimal sorting walk $S_1$ that performs the 6 essential steps along the shortest paths. The continuous line marks the walk of the robot. Every small dot marks a position where a box is swapped. Two non-essential steps between slots 1 and 2 are needed to connect the two cycles. (right) An optimum sorting walk $S_2$ with only one non-essential step. By taking box 1 along the longer part of the circle, the step from vertex 4 to 3 is the only non-essential step and already sufficient to connect $C_1$ to $C_2$.

Even vertex number If $n$ were even, boxes whose targets are on the opposite side of the circle could reach them in both clockwise or counterclockwise direction in $\frac{n}{2}$ steps each. Let us assume that $n$ is odd. This way, we still have unique shortest paths between any pair of vertices. Uniqueness of shortest paths is something that we heavily relied on in our algorithms for paths and trees. We will see at the end, in Remark 3.34, that the ambiguity can easily be handled and that we can also find shortest sorting walks on circles of even length. But for now, we assume that our circle is of odd length.

Example Having unique shortest paths does not guarantee however that the essential steps of each box are performed along this shortest path, not even in an optimum sorting walk. Consider the example in Figure 3.7 to see that an optimum sorting walk might still use the longer path to bring a box to its target if this allows including other cycles using fewer non-essential steps. Note that the optimum sorting walk uses each edge more often in counterclockwise than clockwise direction. Therefore, our up/down essential/non-essential parity counting argument which was used in Theorem 3.14 cannot easily be adapted to a clockwise/counter-clockwise version on circles.

Preliminaries

We start with some definitions and observations. Let the vertices on the circle $C_n$ be numbered in clockwise order. We denote by $[a, b]$ the subpath of the circle from $a$ to $b$ in clockwise order. By $(a, b)$ we mean the path $[a, b]$ without its endpoints, so the path graph that is missing the first and last
3. Rearrangement

vertex of \([a, b]\). This way \([a, b]\) and \((b, a)\) conveniently partition the vertex set of \(C_n\).

In order to have same vocabulary to talk about how a cycle lies on the circle and how it gets sorted during a sorting walk, we introduce the following three terms.

**Definition 3.18 (Circular cycles)** Given a permutation \(\pi\) on \(C_n\), we say that a cycle \(C\) of \(\pi\) is circular if its shortest paths (the union of all shortest paths from \(i\) to \(\pi(i)\) for \(i \in C\)) cover all \(n\) vertices. Otherwise \(C\) is non-circular.

**Definition 3.19 (Sorting in a circular way)** Given a sorting walk \(S\), we say that a cycle \(C\) is sorted in a circular way if the set of vertices that the boxes of \(C\) reach during \(S\) consists of all the vertices of the circle.

Note that this does not mean that every edge of the circle was used while sorting the cycle. There might be one edge of the circle that was not used, as illustrated on the right in Figure 3.8.

**Definition 3.20 (Extending a cycle)** If a cycle \(C\) is non-circular but was sorted in a circular way in a given sorting walk \(S\), we say that \(C\) was extended to be sorted in a circular way.

In the example in Figure 3.7, no cycle is circular, but in the sorting walk \(S_2\) the cycle \(C_1\) is sorted in a circular way. An important distinction that we will make later on will be whether or not the shortest sorting walk extends a cycle to be sorted in a circular way. But to get there, we need some more definitions and machinery.

We use \(I(C)\) to denote the subpath of the circle that is covered by the shortest paths of a cycle. If \(C\) is a circular cycle, \(I(C)\) will be the full circle. Otherwise, \(I(C)\) denotes a subpath of the cycle and we use \(l(C)\) and \(r(C)\) to denote the first and last vertex of \(I(C)\) in clockwise order. \(\|I(C)\|\) denotes the number of vertices covered by \(C\). We let \(V(C)\) denote the vertices hit by \(C\).

Recall from the previous sections that whenever we are sorting a cycle \(C_a\), we can also sort a cycle \(C_b\) without any additional non-essential steps if at least one vertex hit by \(C_b\) is also covered by \(C_a\). We formalize this notion of a cost-free interleaving of the sorting process of two different cycles.

**Definition 3.21 (Linking cycles)** We say that we can link a cycle \(C_b\) into the sorting of the cycle \(C_a\) if \(\exists b \in V(C_b): b \in I(C_a)\). We denote this by \(C_a \leadsto C_b\). We also say that \(C_a\) leads to \(C_b\).

Furthermore, we say that we can link \(C_b\) into \(C_a\) indirectly if there exists a sequence of cycles \(C_1, C_2, \ldots, C_n\) such that \(C_a \leadsto C_1 \leadsto C_2 \leadsto \cdots \leadsto C_n \leadsto C_b\). We also say that \(C_a\) indirectly leads to \(C_b\).

For a sorting walk \(S\), we denote the number of essential steps as \(\|S\|_e\) and non-essential steps as \(\|S\|_n\). Recall that \(\|S\| = \|S\|_e + \|S\|_n\).
3.2. Minimizing the Travel Time

**Observations** Let $S^*$ denote an optimum sorting walk.

- If we assume that a cycle $C$ is sorted in a circular way in $S^*$, all we need to find is a sorting walk that minimizes the non-essential steps until we first reach any box of $C$. Once we start working on $C$, we can interleave all other cycles without any more non-essential steps.

- If there is a circular cycle $C$ in $\pi$, $S^*$ will sort it in a circular way. Sorting $C$ in a non-circular way would mean that we would add non-essential steps which would only make sense if that would allow us to reach new cycles. As circular cycles can already reach all other cycles, this will not happen. So for circular cycles, we will always use the shortest path for every box.

**How to Extend a Cycle**

Let us assume that the optimum sorting walk extends some cycle $C$ to be sorted in a circular way. We will now introduce two tools that allow us to determine the number of non-essential steps needed in a minimum sorting walk under this assumption.

Some non-essential steps will be needed to make sure that we visit all the vertices of the circle while sorting $C$. We call this the *extension cost*. But before we can even start sorting $C$ in a circular way, the robot has to get from his starting position to some box of $C$. We let the *reachability cost* be the smallest number of non-essential steps needed to reach a box of $C$.

**Definition 3.22 (Extension Cost)** For any cycle $C$, let $\text{extend}(C)$ denote the minimum number of non-essential steps needed to sort $C$ in a circular way when the robot starts at any box of $C$.

**Lemma 3.23**

$$\text{extend}(C) = \min_{b \in C}(n - 2 \cdot d(b, \pi(b)), 2 \cdot (n - \|I(C)\|)) \quad (3.4)$$

**Proof** There are two ways of extending a cycle such that it gets sorted in a circular way. Both ways are illustrated in Figure 3.8.

- We can take any box $b \in C$ and move it along the longer path from $b$ to $\pi(b)$. This longer path from $b$ to $\pi(b)$ consists of $n - d(b, \pi(b))$ steps instead of $d(b, \pi(b))$ steps and so the difference in lengths is $(n - d(b, \pi(b)) - d(b, \pi(b))$. So we need $(n - 2 \cdot d(b, \pi(b))$ non-essential steps for that.

- We can also fill in the uncovered area of the circle by adding a detour for one of the two boxes at the border of $I(C)$. Before we bring the box to its target, we move it across the $n - \|I(C)\|$ vertices not covered by $I(C)$ and then take it $n - \|I(C)\|$ steps back to its starting point which accumulates to $2 \cdot (n - \|I(C)\|)$ non-essential steps.
3. Rearrangement

Figure 3.8: Two examples on the circle $C_{11}$ with a single cycle $C$ shown with dashed arcs. The vertices in $V(C)$ and the path $I(C)$ are marked in grey. (left) The cycle gets extended by moving box $b$ the longer way (shown in bold). This way, the box $b$ takes six instead of five steps to reach its target. The first step with $b$ in counter-clockwise direction is therefore non-essential and so $\text{extend}(C) = 1$. (right) As every box only needs to move for at most three essential steps, taking one of them the longer way would result in at least five non-essential steps. Since $I(C)$ covers all but two vertices, it is faster to make a detour of four steps with box $b$ before following its shortest path. Hence $\text{extend}(C) = 4$ in this case.

The cost to extend $C$ is the cheapest of these options.

How to Reach a Cycle

Now that we know how to best extend a cycle, it remains to show how we get there, so to formalize the aforementioned idea of reachability costs. We will define $\text{reach}(C)$ as the minimum number of non-essential steps that we need in any sorting walk that sorts $C$ in a circular way, in addition to the $\text{extend}(C)$ many non-essential steps. How and where these $\text{reach}(C)$ many non-essential steps will be spent is not obvious. Maybe, we should just walk straight from $r$ to the closest box in $V(C)$ and back. But maybe the other cycles of $\pi$ can help us to get to a box in $V(C)$ with fewer non-essential steps than that. If we sort some of the other cycles partly before and partly after we extend $C$, so start sorting them to get to a box in $V(C)$ and then finish sorting them to get back to $r$, we might only need very few non-essential steps in addition to $\text{extend}(C)$. Consider Figure 3.9 for some examples where this is the case.

What we cannot do however is to start sorting some cycles of $\pi$ to get to a box in $V(C)$ cheaply and then not finish sorting these cycles on the way back to $r$, where we account for the rest of $\text{reach}(C)$. Because even when we are sorting $C$ in a circular way and can link all other cycles into $C$, we would still need to spend non-essential steps to finish sorting these cycles that we
started. So let us formalize this notion of sorting walks that sort each cycle of $\pi$ either completely or not at all.

**Definition 3.24 (Partial sorting walk)** A sequence of steps $S' = (s_1, \ldots, s_l)$ is a partial sorting walk if we start in a state $\tau_0 = (r, \square, \pi)$, all steps are valid, and we end in a state $(r, \square, \pi')$ such that the set of non-trivial cycles $\mathcal{C}'$ in $\pi'$ is a subset of the set of non-trivial cycles $\mathcal{C}$ in $\pi$.

This means that $S'$ can sort some of the cycles, but not necessarily any or all of them. So especially the empty sequence that sorts no cycle is a partial sorting walk, as well as any sorting walk that sorts all the cycles. This enforces that whenever $S'$ moves some box, $S'$ has to move that box to its target and also has to bring all other boxes of the same cycle to their targets.

**Definition 3.25 (Reachability Cost)** For any non-trivial cycle $C$, we look at all partial sorting walks $S'$ that visit at least one vertex hit by $C$. We let $\text{reach}(C)$ denote the minimum number of non-essential steps in any such walk $S'$.

Out of all boxes in $C$, only two boxes are candidates for being the ones we reach first in $S'$: the box closest to $r$ in clockwise and the one closest in counter-clockwise direction. We denote them by $\text{cw}(C)$ and $\text{ccw}(C)$. Because $C$ is non-trivial and therefore contains at least two boxes, we have $\text{cw}(C) \neq \text{ccw}(C)$.

As illustrated in Figure 3.9, it is possible that the $S'$ that is optimum according to Definition 3.25 sorts none, one, or even several of the cycles of $\pi$.

How do we compute $\text{reach}(C)$? We could try out all subsets of cycles and then try to find the best partial sorting walk for that subset. But there can easily be exponentially many such subsets and hence such an approach would not be efficient. The problem of finding $S'$ sounds a bit like sorting only the boxes on the path $(\text{ccw}(C), \text{cw}(C))$ between to two boxes of $C$ that are closest to $r$. Reducing it to such a problem on a path is exactly what we will do, but we have to be careful to get the details right.

So why can we not just take that path $(\text{ccw}(C), \text{cw}(C))$? Well, there might be some box $b$ on that path whose target position $\pi(b)$ is not on that path, so $\pi(b) \notin (\text{cw}(C), \text{ccw}(C))$. So we do not necessarily get a valid sub-permutation if we restrict $\pi$ to the vertices of $(\text{cw}(C), \text{ccw}(C))$. Instead, we will take the largest path between $\text{cw}(C)$ and $\text{ccw}(C)$ that contains $r$ and forms a valid sub-permutation. We call this the complement of $C$.

**Definition 3.26 (Complement of a cycle)** For any non-trivial cycle $C$, we let $\xi(C)$ be the largest $r$-containing subpath of the circle that does not contain any vertex in $(\text{cw}(C), \text{ccw}(C))$ and does not contain a box $b$ with $\pi(b) \notin \xi(C)$. We call $\xi(C)$ the complement of $C$. 61
3. Rearrangement

Figure 3.9: Three examples of the reachability cost for some cycle $C$ whose boxes and covered subpath $I(C)$ are marked in grey. The dashed arcs denote the cycles of the permutation. Everything printed in bold is part of $S'$, the partial sorting walk that reaches $C$ (according to Definition 3.25). The bold continuous arcs mark the non-essential steps in $S'$. (left) $C$ is the only non-trivial cycle. Therefore $S'$ is the walk from $r$ to ccw($C$) and back without sorting any cycle. (centre) The additional cycle $C_1$ makes it cheaper to take the path to cw($C$). Out of the eight steps in $S'$ (printed in bold), only four are non-essential (printed as continuous, bold arcs) because the four other steps are essential for $C_1$ (printed as dashed, bold arcs). Hence we have reach($C$) = 4. (right) Out of the cycles $C_1$, $C_2$, and $C_3$, $S'$ chooses to sort only $C_2$ and $C_3$ but not $C_1$. Linking $C_3$ into $C_2$ allows $S'$ to reach ccw($C$) with only two non-essential steps (the two continuous, bold arcs from $r$ to its clockwise neighbour and back).

Note that $\xi(C)$ might be empty, for instance if $r$ is part of a cycle that can be linked to $C$. Figure 3.10 shows a few examples of such cycle complements.

The cycle complement $\xi(C)$ leaves us with a proper instance of the PathSort problem, but it does not entirely capture what we would like to achieve: we are not interested in sorting all the cycles in $\xi(C)$. We only want to find a partial sorting walk that contains one of the end points of $\xi(C)$ so that we can eventually lead the robot to cw($C$) or ccw($C$). We do not mind if we potentially leave some cycles in $\xi(C)$ unsorted. To model this, we add two extra vertices to $\xi(C)$. A vertex $v_l$ on one side and a vertex $v_r$ on the other side. We swap the boxes on these two vertices so that they form a cycle that covers the entire path and leads us to all other cycles. This is the PathSort instance that we will now define in a formal way and use later on.

**Definition 3.27 (Cycle complement problem)** For any cycle $C$ with non-empty $\xi(C)$, let $P_{\xi(C)}$ be the path $\xi(C)$ extended with an extra vertex $v_l$ on the left and a vertex $v_r$ on the right. Let $\pi_{\xi(C)}$ be the permutation on $\{v_l\} \cup \xi(C) \cup \{v_r\}$ with

$$\pi_{\xi(C)}(v) = \begin{cases} 
  v_r & \text{if } v = v_l \\
  \pi(v) & \text{if } v \in \xi(C) \\
  v_l & \text{if } v = v_r
\end{cases}$$

(3.5)
3.2. Minimizing the Travel Time

Figure 3.10: Three instances of CircleSort and their cycle complement problems. (top row) The upper grey area marks the cycle complements for the three examples from Figure 3.9. Note that the two instances on the left and in the centre have the same $\xi(C)$ as the additional cycle $C_1$ in the centre is completely contained in $\xi(C)$. In the example on the right, no more than two vertices can be kept on the left side of $r$ because $I(C_3)$ overlaps with $V^*(C)$ and so no vertex of $C_3$ can be in $\xi(C)$. As the left vertex of $C_2$ is in $I(C_3)$, also this $I(C_2)$ cannot be in $\xi(C)$ and so the right vertex of $C_2$ limits $\xi(C)$ in clockwise direction. (bottom row) The cycle complement problems for these instances with the added $v_i$ and $v_r$ at the ends of the paths. $\|S_{\xi(C)}^r\|$ denotes the length of the shortest sorting walk that solves these PathSort instances, with $\|S_{\xi(C)}^r\|_e$ and $\|S_{\xi(C)}^r\|_n$ being the number of essential and non-essential steps in that walk. Note how the number of non-essential steps equals the reach($C$) costs from Figure 3.9. We will prove in Lemma 3.28 that this is always the case.

By definition of $\xi(C)$, all boxes on $P_{\xi(C)}$ have to be moved to vertices that are also in $P_{\xi(C)}$ and so $\pi_{\xi(C)}$ is guaranteed to be a valid permutation. Therefore sorting $\pi_{\xi(C)}$ on $P_{\xi(C)}$ starting from $r$ is a proper instance of the sorting problem on a path. We refer to Figure 3.10 for a few such examples.

It remains to show that it is sufficient to solve the cycle complement problem in order to compute reach($C$) for any cycle $C$, as suggested by Figure 3.10.

**Lemma 3.28** Let $S_{\xi(C)}^*$ be a minimum sorting walk to sort $\pi_{\xi(C)}$ on $P_{\xi(C)}$. Then $\|S_{\xi(C)}^*\|_n = \text{reach}(C)$.

**Proof** We prove this equality by constructing a partial sorting walk according to the definition of reach($C$) from a given minimum sorting walk for the
cycle complement problem and vice versa. As these two constructions will preserve the number of non-essential steps, we can show the equality as two inequalities.

First inequality: \( \|S^*_\xi(C)\|_n \geq \text{reach}(C) \). Let us assume that we are given a solution \( S^*_\xi(C) \) to the cycle complement problem on the path. We will construct from it a partial sorting walk \( S \) on the circle such that \( S \) reaches a vertex hit by \( C \) and the number of non-essential steps is the same as in \( S^*_\xi(C) \).

We start by following \( S^*_\xi(C) \) until it first reaches \( v_l \) or \( v_r \). Let us denote this part of \( S^*_\xi(C) \) by \( S^*_\xi(C),\text{start} \). We claim that \( \|S^*_\xi(C),\text{start}\|_n = \frac{1}{2}\|S^*_\xi(C)\|_n \), meaning that exactly half of all non-essential steps are in this first part of the walk. This holds because once \( S^*_\xi(C) \) reaches \( v_l \) or \( v_r \), all cycles that have not been partially sorted yet can be sorted without extra non-essential steps as they can all be linked into the sorting of the cycle \( (v_l, v_r) \). So the only non-essential steps in the remainder of \( S^*_\xi(C) \) will be the ones of \( S^*_\xi(C),\text{start} \) performed in reverse direction when the cycles that were partially sorted in \( S^*_\xi(C),\text{start} \) get completed on the way back to \( r \).

Note that \( S^*_\xi(C),\text{start} \) is also a valid sequence of steps when we apply it to the entire circle and permutation \( \pi \). Hence we let \( S \), the partial sorting walk on the circle that we are constructing, begin with \( S^*_\xi(C),\text{start} \).

If the end of \( S^*_\xi(C),\text{start} \) corresponds to a vertex in \( V(C) \), so we immediately reach \( C \) as on the left and in the centre of Figure 3.10, we let \( S \) conclude by finishing the sorting process of all the cycles that were partially sorted in \( S^*_\xi(C),\text{start} \) with the same amount of non-essential steps on the way back to \( r \).

If the end of \( S^*_\xi(C),\text{start} \) does not correspond to a vertex in \( V(C) \), then by definition of the complement problem \( \xi(C) \), that vertex must be part of a cycle \( C' \) that (indirectly) leads to \( C \) and does not have any box in \( \xi(C) \), as in Figure 3.10 (right). As \( V(C') \) is disjoint of \( \xi(C) \), no box of \( C' \) has been moved in \( S^*_\xi(C),\text{start} \) and so we can continue \( S \) on the circle by sorting the cycles needed to reach cw(\( C) \) or ccw(\( C) \). As this is possible by linking cycles, it will not require any more non-essential steps, so all the non-essential steps of \( S \) will be carried out inside \( \xi(C) \).

In either case, the \( S \) that we constructed is a partial sorting walk that visits a vertex hit by \( C \) and has \( \|S\|_n = 2\cdot\|S^*_\xi(C),\text{start}\|_n \). This gives the desired upper bound on \( \text{reach}(C) = \|S'\|_n \leq \|S\|_n = \|S^*_\xi(C)\|_n \).

---

3We let \( v_l \) and \( v_r \) correspond to their natural extensions of \( \xi(C) \) on the circle (i.e., \( v_l \) corresponds to the left neighbour of the leftmost vertex in \( \xi(C) \) and \( v_r \) corresponds to the right neighbour of the rightmost vertex in \( \xi(C) \)).
Second inequality: \( \|S^*_\xi(C)\|_n \leq \text{reach}(C) \). Let us assume that we are given an optimum partial sorting walk \( S' \) on the circle according to the definition of \( \text{reach}(C) \). We will construct from it a sorting walk \( S'_\xi(C) \) on the path \( P_\xi(C) \) such that \( S'_\xi(C) \) sorts all cycles of the permutation \( \pi_\xi(C) \) and the number of non-essential steps is the same as in \( S' \).

Let \( S'_\text{start} \) be the beginning of \( S' \) that stays inside of \( P_\xi(C) \). We know that all non-essential steps of \( S' \) are in \( P_\xi(C) \) because we have seen in the first part that it is possible to reach \( \text{cw}(C) \) or \( \text{ccw}(C) \) without any more non-essential steps once we reach the border of \( P_\xi(C) \).

To build \( S'_\xi(C) \) for the path problem, we start with \( S'_\text{start} \) and then sort the cycle of \((v_l, v_r)\). As this cycle covers the entire path, we can link into it all other cycles that have not been partially sorted yet. Back at the border of the path, we wrap up \( S'_\xi(C) \) by finishing the sorting process of all partially sorted cycles with the same amount of non-essential steps as in \( S'_\text{start} \) on this way back to \( r \).

So even if \( S' \) only sorts some but not all of the cycles in \( \xi(C) \), we can transform it so that we sort all of \( P_\xi(C) \) without additional non-essential steps, thanks to the \((v_l, v_r)\) cycle. In conclusion, the shortest sorting walk for \( \pi_\xi(C) \) on path \( P_\xi(C) \) cannot have more than \( \text{reach}(C) \) many non-essential steps. □

Lemma 3.29 If a cycle \( C \) has an empty cycle complement \( \xi(C) \), we have \( \text{reach}(C) = 0 \).

Proof \( \xi(C) \) is only empty if the box at the starting vertex \( r \) is part of a cycle \( C' \) that (indirectly) leads to \( C \). Therefore it is possible to reach \( C \) without any non-essential steps. □

Finally, we are at a point where we can put all these definitions into action.

Shortest Sorting Walk on a Circle under Different Assumptions

Case Distinction We now do the following complete case distinction on specific properties of the minimum sorting walk for the entire circle \( S^* \) and the permutation \( \pi \). We will study these five cases:

- \( S^* \) extends a single cycle to be sorted in a circular way. \( \rightarrow \) Case a)
- \( S^* \) does not extend a cycle to be sorted in a circular way and
  - \( \pi \) contains a circular cycle. \( \rightarrow \) Case b)
  - \( \pi \) does not contain a circular cycle and
    - all edges of \( C_n \) are used by a non-essential step in \( S^* \). \( \rightarrow \) Case c)
    - some edge in \( C_n \) is not used by a non-essential step in \( S^* \) and
      - there is an edge in \( C_n \) that is not used by an essential step in \( S^* \). \( \rightarrow \) Case d)
      - all edges of \( C_n \) are used by an essential step in \( S^* \). \( \rightarrow \) Case e)
3. Rearrangement

So, we first make a distinction on whether $S^*$ extends a cycle or if not if $\pi$ contains a circular cycle. If neither is the case, we separate three more cases based on whether the essential and non-essential steps of $S^*$ cover the entire circle or not. To see that this case distinction really is complete and that we do not need to say at least one cycle in Case a), we argue that it is never optimal to extend more than one cycle to sort them in a circular way.

**Lemma 3.30** In $S^*$, at most one cycle is extended to be sorted in a circular way.

**Proof** Assume that both $C_1$ and $C_2$ are extended. We then claim that

$$\|S^*\| \geq d(\pi) + \text{extend}(C_1) + \text{extend}(C_2) + \min(\text{reach}(C_1), \text{reach}(C_2))$$

because at least one box of $C_1$ or $C_2$ has to be reached before all other cycles can be sorted with essential steps. Now let $S_1$ be the sorting walk that only extends $C_1$ and hence has length $\|S_1\| = d(\pi) + \text{extend}(C_1) + \text{reach}(C_1)$. Analogously $S_2$ only extends $C_2$ with $\|S_2\| = d(\pi) + \text{extend}(C_2) + \text{reach}(C_2)$. As $\text{extend}(C_1) > 0$ and $\text{extend}(C_2) > 0$, we have $\|S^*\| > \min(\|S_1\|, \|S_2\|)$ and reach a contradiction. □

We now go through the five cases and explain how we find the shortest sorting walk under each of these assumptions.

**Case a) Extending a single cycle to be sorted in a circular way** As we assume that the shortest sorting walk $S^*$ only extends a single cycle, we can find a sorting walk $S_{\text{min}}$ of equal length by just trying to extend each non-trivial cycle individually.

For every $C \in \mathcal{C}$, we compute $\text{extend}(C)$ and $\text{reach}(C)$ by solving the cycle complement problem and then we use the cycle $C_{\text{min}}$ with

$$C_{\text{min}} = \arg \min_{C \in \mathcal{C}}(\text{extend}(C) + \text{reach}(C))$$

to generate $S_{\text{min}}$ as described in the proof of Lemma 3.28.

For the remaining cases, we can now assume that no cycle gets extended, which means that all essential steps are performed along the shortest paths of their boxes.

**Case b) No cycle is extended but $\pi$ contains a circular cycle** If there is a circular cycle $C$, we have the nice property that once we reach a box of this cycle, we can link all other cycles into it. So if there is a circular cycle, the interesting part of the walk is the one until we reach a box of $C$. An optimal walk for this part can be found using the computation of $\text{reach}(C)$ in the exact same way we did for non-circular cycles in $C$. In fact, we already included the circular cycles in the search for $C_{\text{min}}$ in Case a). Therefore no additional work has to be done in this case. Circular cycles will just have $\text{extend}(C) = 0$. 66
3.2. Minimizing the Travel Time

Figure 3.11: Example for Case c). An instance on $C_{15}$ with three non-trivial cycles. The shortest sorting walk is shown by the continuous arrow, the small dots mark the endpoints of the six essential steps. In this example, it is optimal to perform a full circle of 15 non-essential steps. All other options produce longer sorting walks: Extending one of the cycles would result in at least 6 non-essential steps for reaching and 13 non-essential steps for extending that cycle. Also not using one of the edges of the circle results in a longer walk, namely in at least $2 \cdot 3 \cdot 3 = 18$ non-essential steps.

Case c) $S^*$ covers the full circle with non-essential steps. This implies that $S^*$ contains at least $n$ non-essential steps. No minimum sorting walk on a circle will ever use more than $n$ non-essential steps, as we can always build the following walk $S$: we let the robot walk once around the circle in clockwise order. Whenever it encounters a box that has not been sorted yet, it sorts the entire cycle to which the box belongs. This will result in $\|S\| = d(\pi) + n$. See Figure 3.11 for an example where such a sorting walk is optimal.

Let us recall what we can assume for the last two cases: there is no circular cycle, we do not extend one and we do not perform a full circle of non-essential steps.

Case d) Some edges are not used by essential steps in $S^*$. As we can assume that all essential steps are performed along shortest paths, we have some edges of the circle that are not part of the shortest path of any box. The largest of these gaps between connected components of cycles will not be used by non-essential steps either, because we know that there is at least one edge not used by a non-essential step and it is hence optimal to get rid of the largest gap. Ignoring this largest gap reduces the circle to an instance of the tree sorting problem and can hence be solved with the algorithm from Theorem 3.16. Figure 3.12 shows an example of this transformation.

Case e) All edges covered by shortest paths. If all edges of the circle are covered by shortest paths, there will of course be no such gap that the robot will never cross, as there was in Case d). In this case, all cycles build one
single connected component. But this does not mean that it is trivial to link all cycles into each other because non-essential steps might still be required to get from the starting position to a cycle that can lead to all other cycles. But we will now show that such a cycle always exists under the assumptions of this case.

**Definition 3.31 (master cycle)** A cycle $C \in \mathcal{C}$ is called a master cycle if all other cycles in $\mathcal{C}$ can be indirectly linked into $C$.

**Lemma 3.32** Under the assumptions of Case e), every cycle $C \in \mathcal{C}$ that is not fully contained by any other cycle, so $\not\exists C' \in \mathcal{C}$ such that $I(C) \subset I(C')$, is a master cycle.

**Proof** All cycles $C'$ that overlap $C$, i.e., $I(C) \cup I(C') \neq \emptyset$, can be reached while sorting $C$ and therefore can be directly linked into $C$. For all other cycles, we can go through them in clockwise order, ordered by $I(C)$, their
first box in clockwise order, and argue inductively that we can link all of them indirectly into $C$. Let $\tilde{C}$ be the next such cycle in this order. There must be another cycle $C^*$ that also covers $l(\tilde{C})$, as we assume that all edges of the circle are covered by cycles and $\tilde{C}$ only covers one of the two edges incident to $l(\tilde{C})$. Because we know that no cycle fully includes $I(C)$ and $l(\tilde{C}) \in I(C^*)$, we know that $C^*$ cannot extend to the left of $l(C)$ and so we have $l(C) < l(C^*) < l(\tilde{C})$ in clockwise order. Therefore $C^*$ is either a cycle overlapping with $C$ or a cycle that we looked at already and by induction already know how to link it into $C$. As $l(\tilde{C}) \in I(C^*)$ we can therefore also indirectly link $\tilde{C}$ into $C$. Hence $C$ is a master cycle. See Figure 3.13 for an illustration of this argument. 

\begin{corollary} 
Every setting of Case e) contains a master cycle that we can find efficiently.
\end{corollary}

\begin{proof} 
Since interval-inclusion ($I(C_1) \subset I(C_2)$) induces a partial order of the cycles, there is always a maximum element, i.e., a cycle that is not contained in any other cycle. By Lemma 3.32 every such cycle is a master cycle. A simple pair-wise comparison allows us to easily find such a cycle in time $O(n^2)$. 
\end{proof}

Once we have found such a master cycle $C$, we know that the length of the optimum sorting walk under the assumptions of this case is $d(\pi) + \text{reach}(C)$. We compute this walk solving the cycle complement problem of $C$.

\section*{Combining all the Cases}

\begin{remark}
The same case distinction also applies for circles with even $n$.
\end{remark}

We assumed $n$ odd above, such that the shortest path is always unique. If $n$ is even, the shortest path is not unique for all the boxes that have to go to the directly opposite vertex. Deciding whether we move such a side-swapping box $b$ in a cycle $C$ in clockwise or counter-clockwise direction to $\pi(b)$ could potentially have an influence on how easy it is to link other cycles into the sorting walk. But this is not the case: it is easy to see that for every such cycle $C$ we can always pick one of the two shortest paths for $b$ such that $C$ is sorted in a circular way, and therefore all other cycles can be linked directly into $C$. To see this look at the remainder of $C$, a sequence of shortest paths that starts at $\pi(b)$ and ends at $b$. At least one of the half circles between $b$ and $\pi(b)$ is already fully covered by these shortest paths (even if there are other side-swapping boxes in $C$, no matter which shortest path we pick for them). So for $b$ we can always pick the one shortest path which ensures that $C$ is sorted in a circular way.

\begin{theorem} \textbf{(Efficient solution for CircleSort)} \ For any sorting problem on a circle with permutation $\pi$, we can find a minimum sorting walk in time $O(n^3)$.
\end{theorem}
3. Rearrangement

Figure 3.14: (left) An input to the Hamiltonian circuit problem on a grid graph. (right) The corresponding input to the sorting problem.

**Proof** The case distinction above is complete. In each of the five cases we can either efficiently generate the sorting walk under these assumptions directly or have to solve at most \( n \) different problems of sorting on a line, which by Theorem 3.16 can done in time \( O(n^2) \) each. We can find a minimum sorting walk by just taking the shortest of all those different walks. □

3.2.5 Sorting on General Graphs

In this section, we show that our efficient algorithms for paths, trees, and circles cannot be easily extended to solve the general GraphSort problem. In fact, no efficient algorithm for general graphs can be found unless \( P = NP \).

**Theorem 3.36 (NP-completeness for planar graphs)** Finding a shortest sorting walk for a planar graph \( G = (V, E) \) and permutation \( \pi \) is \( NP \)-complete.

**Proof** We show a reduction from the problem of finding Hamiltonian circuits for grid graphs. This problem was shown to be \( NP \)-complete by Itai et al. in Theorem 2.1 of [36]. The main idea is to replace each vertex of the grid by a pair of neighbouring vertices with swapped boxes. Given a grid graph \( \hat{G} = (\hat{V}, \hat{E}) \), we build the following input for the sorting problem:

\[
V = \{v \mid \forall v \in \hat{V}\} \cup \{v' \mid \forall v \in \hat{V}\} \quad (3.6)
\]

\[
E = \{(v, v') \mid \forall v \in \hat{V}\} \cup \hat{E} \quad (3.7)
\]

\[
\pi(v) = v' \forall v \in \hat{V} \text{ and } \pi(v') = v \forall v \in \hat{V} \quad (3.8)
\]

We can use any vertex in \( V \) as the starting vertex for the robot. An illustration of this transformation is given in Figure 3.14.

Let \( n = \|\hat{V}\| \). We now claim that \( \hat{G} \) has a Hamiltonian circuit if and only if \( \pi \) on \( G \) can be sorted in exactly \( 3n \) steps. \( d(\pi) = 2n \), as \( \pi \) contains \( n \) pairs of neighbouring boxes that have been swapped, so any sorting walk
3.3. Minimizing the Box Handling Time

will contain exactly $2n$ essential steps. As all essential steps will move along edges in $E \setminus \hat{E}$, only non-essential steps can be used to move from cycle to cycle and so at least $n$ non-essential steps are needed to complete the sorting walk. If the sorting walk contains exactly $3n$ steps, the $n$ non-essential steps will only move along edges in $\hat{E}$, visiting all vertices in $\hat{V}$ and therefore build a Hamiltonian circuit for $\hat{G}$. So finding a shortest sorting walk is at least as hard as determining Hamiltonicity of grid graphs. Checking a given sorting walk of length $3n$ is easy, so the problem is clearly in $\mathcal{NP}$ and therefore $\mathcal{NP}$-complete. The fact that all grid graphs are planar concludes the proof. □

3.2.6 Implementation and Visualization

We provide an implementation of our two algorithms for BFrontPathSort and for TreeSort on our website: http://dgraf.ch/treesort. To find the shortest sorting walk on paths with $r = v_1$, we perform the algorithm of Theorem 3.7 that links the cycles from left to right. For arbitrary trees, we compute the optimum branching using Edmond’s algorithm [22] to find a cheapest cycle anchor tree and generate a corresponding sorting walk as in Theorem 3.16. A text-based visualization allows the animation of shortest sorting walks and can be used to perform the sorting steps interactively. The interested reader can find a detailed usage tutorial on the website.

3.3 Minimizing the Box Handling Time

In practice, the time to load a box onto the robot is not negligible. The Bike Loft system [60] takes roughly 5 seconds to load or unload a box and can move at the speed of roughly $3m/s$. So especially in small or medium size systems with only a few dozen or hundred storage slots, this box handling time can have a significant effect on the overall performance.

Ideally, we would try to minimize a cost function that models the combined handling and driving time. As a step in this direction, in this section, we study the problem of finding sorting walks with minimal number of box handling operations. As a second priority, we still want to minimize the number of steps in the sorting walk.

Definition 3.37 (Swap count) The swap count $\langle S \rangle$ of a sorting walk $S$ denotes the number of boxes picked up during the walk. Formally

$$\langle S \rangle = |\{s_i = (p_i, b_i) \in S \mid b_i \neq \square \text{ and } (i = 1 \text{ or } b_{i-1} \neq b_i)\}| \quad (3.9)$$

Recall that $b_i = \square$ denotes a step where the robot does not carry a box and so $\langle S \rangle$ counts the number of steps where the robot carries a different box than in the step before.
Definition 3.38 (Swap-optimal sorting walk) A sorting walk $S$ for a permutation $\pi$ on a graph $G$ is called swap-optimal if among all sorting walks it has minimum swap count $\langle S \rangle$ and has minimum length among all sorting walks with minimum swap count. Formally

$\langle S \rangle$ is swap-optimal $\iff \exists$ a sorting walk $S'$ s.t. $\langle S' \rangle < \langle S \rangle$ or $(\langle S' \rangle = \langle S \rangle$ and $\|S'\| < \|S\|)$. 

(3.10)

We denote the arising swap-optimal sorting problems by GraphSwapSort, BorderPathSwapSort, PathSwapSort and TreeSwapSort.

3.3.1 Sorting on a Path

Theorem 3.39 (Swap-optimal border-starting sorting walk on paths) A swap-optimal sorting walk on a path $P$ with $r = v_1$ and permutation $\pi$ can be constructed in time $\Theta(n^2)$ and has length

$$d(\pi) + 2 \cdot \left(\max_{C \in \mathcal{C}} l(C) - 1\right).$$

(3.11)

Proof If we want to minimize the number of swaps, the robot can pick up each box at most once. This way, the number of swaps will be $n - \|C\| + \|\mathcal{T}\|$, namely the number of boxes that are not placed correctly from the beginning. This means that every box has to be brought straight to its target position once it is loaded onto the robot. Therefore, the robot is forced to sort the permutation cycle by cycle. Once he loads a box of cycle $C$, he cannot do anything else before all boxes in $C$ are sorted.

As we can no longer interleave the sorting of different cycles, as we did for shortest sorting walks in Theorem 3.7, the overlap components $D$ are not relevant here. We just need to reach one box of each cycle as quickly as possible. The steps to connect the cycles will all be non-essential, no matter in which order we sort the cycles.

Hence what we are looking for is a minimum connected subgraph $P'$ that contains the starting vertex and at least one vertex of every non-trivial cycle. $P'$ then represents the non-essential part of the sorting walk. As the robot starts at the left end of the path, all it has to do is walk to the right until it encounters the leftmost box of every cycle. On its way back, it can weave in the sorting of all the cycles.

Since we can easily find $P'$ by checking the leftmost box of all non-trivial cycles and since $d(\pi) \in \Theta(n^2)$, the claim follows. $\square$

In contrast to the sorting walks of minimal length, where having a non-border starting position made things much more complicated, a swap-optimal sorting walk can also be found easily if the robot starts anywhere on the path.
3.3. Minimizing the Box Handling Time

Figure 3.15: (left) Example for the subgraph $P^\prime$ of non-essential steps for a swap-optimal sorting walk. $C_3$ is the cycle with its leftmost box furthest to the right. (right) Example where the robot starts at a non-border position. $P^\prime = [4,7]$ is the only connected subgraph of at most 3 edges that contains the starting vertex 5 and a vertex of every non-trivial cycle.

**Theorem 3.40 (Efficient solution for PathSwapSort)** A swap-optimal sorting walk on a path $P$ with any $r$ and permutation $\pi$ can be constructed in time $\Theta(n^2)$.

**Proof** As before, we are looking for a connected $r$-containing subgraph $P^\prime$ that contains at least one vertex of every non-trivial cycle. There are only $O(n^2)$ many such subgraphs and we can easily check for each of them in amortized constant time whether they contain a box of every cycle. $P^\prime$ then allows the robot to reach and sort all the cycles with minimum number of swaps and minimum number of non-essential steps. \hfill \blacksquare

Figure 3.15 gives two examples of such minimal subgraphs that describe the non-essential steps in swap-optimal sorting walks.

### 3.3.2 Sorting on a Tree

We now look at a tree $T = (V, E)$ as the underlying structure of our warehouse. As on paths, all we need to find is a connected subgraph $T'$ that contains both the root and a vertex of every non-trivial cycle. But as we will see, this is already a hard problem. This problem is known as the **Class Steiner Tree** problem, see Section 3.4 for related work. As in Theorem 2.23, we use a reduction from the problem of finding a satisfying assignment for a formula in 3-conjunctive normal form.

**Definition 3.41 (3SAT)** Given a boolean formula $F = \{C_1, \ldots, C_M\}$ of $M$ clauses. Each clause consists of exactly three literals $C_i = \{l_{i1}, l_{i2}, l_{i3}\}$ over a set of $N$ variables $x_1, \ldots, x_N$ and their negations $\bar{x}_1, \ldots, \bar{x}_N$. Is there a boolean assignment $\alpha: \{x_1, \ldots, x_N\} \rightarrow \{\text{True, False}\}$ that satisfies all $M$ clauses?

**Theorem 3.42 ($NP$-completeness for trees)** Finding a swap-optimal sorting walk on a tree $T$ and a permutation $\pi$ is $NP$-complete.

**Proof** We describe a reduction $\varphi$ that maps every formula $F$ in 3-conjunctive normal form to a rooted tree $T$ and a permutation $\pi$ for the problem of finding a swap-optimal sorting walk.

73
3. Rearrangement

The main idea is to build a spider graph where each leg represents a literal. At the end of the $x_i$-leg there is a box labelled $\bar{x}_i$ and vice versa, so that the two endpoints of the $x_i$-leg and the $\bar{x}_i$-leg form a cycle and the robot has to include at least one of those legs into his subgraph $T'$ of non-essential steps. Along each leg, we have one box per clause and for every clause, we permute the three boxes on those three legs that correspond to the literals in that clause. This way, the connected subgraph $T'$ needs to contain at least one of the tree literal legs involved in every clause.

For a formal definition, let $L = \{x_1, \ldots, x_N\} \cup \{\bar{x}_1, \ldots, \bar{x}_N\}$ denote the set of all literals on $N$ variables. We create a tree $T = (V, E)$ with

\begin{align}
V &= \{r\} \cup \{v_{l,C} \mid l \in L \text{ and } C \in F\} \cup \{v_l \mid l \in L\} \quad (3.12) \\
E &= \{(r, v_{l,C}) \mid l \in L\} \cup \{(v_{l,C}, v_{l,C+1}) \mid l \in L, i \in [M-1]\} \cup \{(v_{l,M}, v_l) \mid l \in L\} \quad (3.13)
\end{align}

and a permutation $\pi \in S_n$ and

\begin{align}
\pi(v_{l,C}) &= v_{i,C} \text{ for } C_i = (l_1, l_2, l_3) \in F \quad (3.14) \\
\pi(v_{i,C}) &= v_{i,C} \text{ for } C_i = (l_1, l_2, l_3) \in F \quad (3.15) \\
\pi(v_{i,C}) &= v_{i,C} \text{ for } C_i = (l_1, l_2, l_3) \in F \quad (3.16) \\
\pi(v_l) &= v_l \text{ for } l \in L \quad (3.17) \\
\pi(v) &= v \text{ otherwise} \quad (3.18)
\end{align}

We call the cycles of length 3 clause cycles and the cycles of length 2 variable cycles. An illustration of such an instance is given in Figure 3.16.

For every variable $x_i$, $T'$ needs to contain either $v_{x_i}$ or $v_{\bar{x}_i}$. Otherwise $T'$ would not cover all variable cycles. Therefore, at least $N$ legs of the spider need to be fully contained in $T'$, so $|E(T')| \geq N \cdot (M+1)$. It remains to show that this inequality is only tight if $F$ can be satisfied.

**Claim** $F$ is satisfiable if and only if $(T, \pi) = \varphi(F)$ contains a subtree $T'$ that contains $r$ and reaches all cycles of $\pi$ and has size $|E(T')| = N \cdot (M+1)$.

$\Rightarrow$: Given a satisfying assignment $a$, we let $T'$ be the tree of the $N$ legs that correspond to literals which evaluate to true under $a$. Clearly all variable cycles are covered this way. Since every clause needs to have at least one satisfied literal under $a$, also all clause cycles are covered this way.

$\Leftarrow$: The variable cycles enforce that whenever we have $T' = N \cdot (M+1)$, we know that $T'$ consists of $N$ complete legs and no partial legs of the spider. As exactly one leg per variable has to be in $T'$ (either $(r, v_{x_i})$ or $(r, v_{\bar{x}_i})$) we can read off an assignment $a$ with $a(x_i) = true$ if and only if the $(r, v_{x_i})$-leg is in $T'$. As $T'$ covers all clause cycles, $F$ is satisfied by $a$. \hfill \square

In conclusion, the sorting problems TreeSwapSort and hence also the general GraphSwapSort are $\mathcal{NP}$-complete.
3.4 Related Work

Efficient algorithms for sorting physical objects were studied in countless different models. We give an overview of these models and also refer to some work related to the techniques that we used in our two hardness proofs (Theorem 3.36 and Theorem 3.42). In Sections 3.4.3 and 3.4.4, we introduce in detail two physical sorting problems which appeared in algorithmic programming competitions recently and are interesting variations of our problem of finding shortest sorting walks on paths.

3.4.1 Physical Sorting Algorithms

We distinguish physical sorting algorithms by the type of operation that the methods can perform and the kind of additional resources they have.

For instance, sorting a permutation by repeatedly reversing parts of it was studied in various flavours. The restricted version by Gates and Papadimitriou [28], where only prefixes of the permutation can be flipped, is motivated by the problem of sorting a pile of pancakes using a spatula. Bulteau et al. [13] recently showed that it is $\mathcal{NP}$-hard to find a shortest possible sequence of such prefix reversals that sorts a given permutation. Kaplan et al. [39] studied the extension to signed permutations, where each operation reverses a part of the permutation and also flips the sign of that part. One applica-
3. Rearrangement

tion is in computational biology where the reversal distance between two chromosomes can measure their evolutionary distance.

Bender et al. [7] proposed a method called library sort. It is based on the idea that if a librarian would put all her books into a long shelf in alphabetic order, she would not squash them all at the beginning of the shelf but spread them out evenly, leaving an empty spot here and there. This way, a new acquisition that has to go into the middle of the shelf only requires a few books to be moved and not half of them. The goal of this variation of insertion sort is to keep gaps between the array entries in order to prevent frequent expensive moves when adding new books to the library.

Another way one could sort physical items is by a process Elizalde and Winkler called homing [23]. If you know the final order of the objects, a homing operation is the following: take one object and place it at its final position while shifting the other objects by one position as necessary. They show that for any sequence of homing steps the permutation eventually becomes sorted but it can take exponentially many steps in the worst case. It is easy to see though that it is possible to sort any permutation in \( n - 1 \) homing steps by homing the objects from left to right in their final order.

Sorting streams of objects was studied for instance by Knuth [42], where we can use an additional stack to buffer objects for rearrangement. These results were later generalized for sorting permutations with a sequence or entire network of stacks and queues, first by Tarjan [56] and then by many others. We refer to Bóna for a survey [11]. A typical area for applications and extensions of these algorithms is the design of efficient railway switchyards. There railway cars need to be sorted with as few shunting steps as possible. Gatto et al. [29] give a nice introduction into this topic of shunting.

An algorithm with the goal of minimizing the number of writes when sorting an array is called cycle sort by Haddon [33]. It works by fixing one cycle of the permutation after the other, the same way we sort the boxes on a path when minimizing the box handling time.

The more general problem of sorting \( n \) objects on a graph of \( n \) vertices using as few swaps of objects on neighbouring vertices as possible was studied for various graphs in the past.

- For complete graphs, which allow to swap any pair of objects, this corresponds to the setting of cycle sort and was already known by Cayley in 1849 [15].

- For path graphs, which allow adjacent transpositions, the shortest sequence of swaps can be generated by bubble sort and the number of swaps needed is the inversion number of the permutation, as shown by Knuth [42].
3.4. Related Work

- Jerrum [37] gives a solution for the case where the underlying graph is a cycle, so also the first and the last element can be swapped. This turned out to be substantially more complicated than on the path and required solving an integer program.

- Katsuhisa et al. [61] recently studied this problem on trees. They give a 2-approximation for tree graphs by simulating cycle sort.

Compared to our setting, these models do not require that successive swaps are applied to nearby vertices and hence do not have to care about the non-essential steps between objects that need to be moved, like we do. We could interpret this as a robot that is able to teleport whenever it is not carrying a box.

Sliding tokens on graphs were also studied by Demaine et al. [20] in a context not related to sorting. In their model, the tokens form an independent set and we want to decide whether it is possible to transform one independent set into another independent set by sliding one token at a time so that all intermediate states form independent sets as well. For planar graphs, this problem is PSPACE-complete, but they give an efficient algorithm for trees.

Sliding physical objects also appear in many popular puzzle games like the Dad’s puzzle, the 15 puzzle or the board games Rush Hour or Ricochet Robots. They were studied in the context of their computational complexity and a lot of them have been proven to be PSPACE-complete. We refer to Hearn [34] for an overview.

3.4.2 Hard Problems for Travelling on Graphs

Our hardness proof for finding shortest sorting walks on planar graphs is based on the proof by Itai et al. [36] that showed that the Hamiltonian path problem and Hamiltonian circuit problem on grid graphs are $\text{NP}$-complete. It is interesting to note that the hardness requires the fact that these grid graphs can contain holes. Umans [59] showed that finding Hamiltonian cycles on grid graphs without holes is in $\text{P}$.

When minimizing the box handling time, we faced the problem of finding a minimal connected subgraph that contains at least one vertex for every non-trivial cycle. Since the order of the boxes inside each non-trivial cycle does not matter for our search of the swap-optimal sorting walk, we can abstract this to the following problem: we are given a family of subsets of vertices and we want the smallest connected subgraph that contains at least one vertex from every set of vertices. This problem is known as the Class Steiner Tree or Group Steiner Tree problem\(^4\). This problem was

\(^4\)Note that in our problem every vertex can be contained in at most one non-trivial cycle.
3. Rearrangement

first introduced by Reich and Widmayer [50]. It is in \( \mathcal{P} \) if the underlying graph is a path, as we showed in Theorem 3.39. The \( \mathcal{NP} \)-completeness for the case where the graph is restricted to a tree was shown by Ihler et al. [35]. Like we do in Theorem 3.42, they also used a reduction from 3SAT but their construction is different than ours as their graph model allows weighted edges, specifically zero-weight edges.

3.4.3 Croatian Task Kutije

The following task was given as part of the Croatian National Competition in Informatics in 2006 [21]. The task is called Kutije (boxes) and was written by Luka Kalinović (see [38] for the original problem statement).

**Problem Statement**  A sequence of \( n \) boxes is given, numbered from left to right. Each box contains exactly two marbles. Every marble is either black or white. A robot, initially placed above the leftmost box, can perform the following steps:

- move one box to the left or to the right
- pick up one of the marbles in the box below it
- drop one of the marbles it is carrying into the box below it

The robot can carry at most two marbles at a time and every box is big enough to contain an arbitrary amount of marbles.

The robot’s task is to sort the marbles so that

- each box contains exactly two marbles of the same color,
- between any two boxes containing white marbles there is no box containing black marbles and
- between any two boxes containing black marbles there is no box containing white marbles.

In other words: the robot should sort the marbles so that all white marbles are on one side and all black marbles on the other side. It is guaranteed that this is always possible, meaning that the amount of marbles of each colour is even. What is the shortest sequence of steps the robot can take?

**Solution Sketch**  We compute the shortest sequence of steps to reach both a black-then-white and a white-then-black configuration and take the shorter of the two. Let us assume that we target the white-then-black configuration.

One might come up with the following approach: Let the robot repeatedly take the leftmost two black marbles and swap them with the rightmost two

---

In the [Class Steiner Tree](#) problem a vertex can be contained in multiple classes.
First Greedy Approach: Total of 65 steps (49 moves and 16 marble handling)

Our Optimal Solution: Total of 56 steps (40 moves and 16 marble handling)

Figure 3.17: Example for the task Kutije. Repeatedly swapping the outermost two misplaced marbles of each colour, as shown at the top, does not lead to an optimal solution. An optimal solution generated by our approach is shown at the bottom.

white marbles until we achieved the intended order. However, this approach is not optimal as we illustrate in Figure 3.17. As in that example, the robot might end up walking more steps during which he carries no marbles than is really necessary. In the terms of our GRAPHSORT problem, we would say that this approach does not minimize the number of non-essential steps.

Instead we propose the following solution for this task, which contains a crucial modification of the greedy approach mentioned above:

1. Move to the right and pick up the two leftmost black marbles along the way.

2. Move to the right until you encounter the rightmost two white marbles and swap them (by first putting a black marble down and then picking a white marble up).

3. Go back to the leftmost two black marbles and swap them with the two white marbles you are carrying.
3. REARRANGEMENT

4. Repeat steps 2. and 3. as long as there are at least two black and two white marbles not placed as in the target state. If there is only one such marble of each colour left in either of the steps, perform the steps with this one marble.

5. You will end up carrying two white marbles. Go back to the position(s) where you picked up the first two black marbles in the very beginning and place your white marbles there. If there was only a single misplaced black marble to begin with, just go to the box where it was.

This procedure clearly achieves that the marbles get sorted, but is it fastest possible? The number of marble handling steps is clearly minimal since every marble that is not placed correctly from the beginning is picked up and put down exactly once. So we need to argue that the number of left and right moves is also minimal. Their optimality can be seen using a similar sum-of-target-distances argument as we used in our lower bound of Lemma 3.3. Compared to our setting on the path in Section 3.2.2, the robot in this task can transport two items at the same time and there are multiple slots where every item could end up at. The sum of target distances (which was \(d(\pi)\) in our setting) is independent of which black marble from the left side goes to which white spot on the right side and vice versa. In this setting, we call a step fully-essential if it moves two marbles one step closer to its target than they were ever before. Whenever a step only moves one marble closer towards its target, we call it semi-essential. If a step is not essential for any marble, we call it non-essential.

Ideally, our algorithm would only perform fully-essential steps. We can argue that the few non-essential and semi-essential steps that our algorithm performs are unavoidable.

- **non-essential steps**: In the very beginning, until we reach the first black marble when walking to the right, we do not move any marble. These non-essential steps are clearly unavoidable (marked \(x\) in Figure 3.18).

- **semi-essential steps**: Let us look at every edge of the path graph corresponding to these \(n\) boxes. If an edge has \(t\) black marbles on its left side that should go to its right side, we clearly need to walk at least \(\left\lceil \frac{t}{2} \right\rceil\) essential steps across this edge from left to right. Hence whenever there is an edge with an odd number of marbles crossing it in either direction, it is unavoidable to spend at least one semi-essential step across this edge in each direction. And this is exactly what our algorithm achieves (marked \(y\) in Figure 3.18).

Figure 3.18 illustrates this algorithm and the optimality argument above. For a more formal proof, one can translate the edge-wise lower bounds on how often each edge has to be travelled into a directed multi-graph as sketched
at the bottom of Figure 3.18. Since the sorting steps of our algorithm correspond to an Eulerian path of this lower-bound-graph, the number of steps in our approach is smallest possible.

### 3.4.4 Tait’s Counter Puzzle

**Problem Statement** The following puzzle goes back to the Scottish mathematical physicist Peter Guthrie Tait who wrote in the late 19th century [55]:

“A few weeks ago, in a railway-train, I saw the following problem proposed: Place four sovereigns and four shillings in close alternate order in a line. Required, in four moves, each of two contiguous pieces (without altering the relative position of the two), to form a continuous line of four sovereigns followed by four shillings.”

According to Martzloff [47] the same puzzle was already known in Japanese sources in the 18th century. A variant of this puzzle was also given by Martin Gardner in his column in the Scientific American in June 1961 as well as in a collection of his favourite puzzles [25].
More recently, namely at the ACM International Collegiate Programming Competition World Finals 2014 in Yekaterinburg [1], Tait’s counter puzzle was posed as task Baggage [2]. The task generalizes the puzzle to $2n$ coins. It is phrased in the context of reordering baggage bins at an airport, but we keep using the original setting with coins of two types. We say that there are $n$ black coins and $n$ white coins that are initially alternating, starting with a black coin. The task still is to find a shortest sequence of moves that rearranges the two types of coins into sorted order, so that all white coins are to the left of all black coins and the $2n$ coins occupy $2n$ consecutive spots. Every move has to move a pair of neighbouring coins and $n$, the number of coins of each colour, is limited to $3 \leq n \leq 100$.

In contrast to our problem on a line or the task Kutije, we only count the number of moves in this problem and we do not care how far each move takes a coin or whether subsequent moves are close to each other or not.

Solution Sketch As solutions for this task are a bit scattered throughout the internet, we briefly state the main ideas here. We summarize the approaches given in the blog post by Maria Keet [41], the forum entry of topcoder user Snapdragon [52] and the video by ICPC analyst David Sturgill [53].

As we will see, it is always possible to sort $2n$ coins in $n$ moves. Using exhaustive search, we can find the solutions for some small $n$. Figure 3.19 gives a shortest sorting sequence for $n$ from 3 to 7. At first sight, these cases all look rather chaotic and not very structured. Notice however that for $n = 4$, $n = 5$, $n = 6$ and $n = 7$ the two positions on the left of the initial coin configuration are the only two positions that are used in addition to the initial locations throughout the sorting process. This allows us to use these four cases as base cases for an inductive solution. For any $n > 7$, we can use the four steps shown in Figure 3.20 as a reduction to the problem with $n - 4$ pairs of coins. Using this step repeatedly, we can construct a sequence of $n$ steps that sorts $2n$ coins for any $n \geq 3$.

It remains to show that it is impossible using less than $n$ steps. Perhaps we would need fewer moves if the sorted order would be at a different position at the end? To see that this is not the case, take any final position of the sorted sequence. Figure 3.21 gives four such potential shifts of the sorted sequence. For every pair of a black and a white coin in the initial state, we mark exactly one of the two coins. Depending on how these two coins compare to the sorted state at these two locations, we will either mark the black coin on the left or the white coin on the right. At least one of the two locations has to be different as there is no black coin to the left of a white coin in the sorted state.

1. If only one of the two coins is different, we mark that coin.
3.4. Related Work

Figure 3.19: Optimal solutions for the smallest instances with $3 \leq n \leq 7$ for the tasks baggage.

Figure 3.20: Illustration of the induction step to sort the coins for any $n > 7$. After two steps we can sort the subproblem for $n - 4$. After another two steps we have sorted all $n$ pairs of coins. So if we can do it in $n - 4$ steps for $n - 4$ pairs of coins, then we can do it in $n$ steps for $n$ pairs of coins.
3. **Rearrangement**

![Figure 3.21: Marking argument for four different shifts of the sorted sequence. The pairs labelled 1 to 4 correspond to examples for each of the cases of our marking scheme. For instance at the pair labelled 2, both locations contain different coins at the end and so we mark the white coin. The $n$ non-neighbouring marks illustrate that at least $n$ moves are required.](image)

2. If both locations contain different coins, so a white and then a black coin, we mark the white coin on the right.

3. If both locations do not contain a coin and the sorted configuration is further to the right, we mark the black coin on the left.

4. If both locations do not contain a coin and the sorted configuration is further to the left, we mark the white coin on the right.

These four rules are illustrated in Figure 3.21. Note how we never mark two neighbouring coins. For all possible shifts of the target configuration, our rules ensure that when we go through the pairs from left to right, we first mark the black coin for some pairs and then mark the white coin for the remaining pairs. So again, this ensures that we never mark two adjacent locations. Since all the marked locations are different in the final configuration, the coin originally placed there has to be moved at least once. As any move can transport at most one marked coins, at least $n$ moves are required.
Chapter 4

Other Bicycle Storage Systems

There are many different interesting automated bicycle storage systems in use all over the world. The Bike Loft system, from which we abstracted our models, is just one of them. In this chapter we want to provide a brief overview of the different ideas that have been explored and give some hints on how they could be expressed in settings similar to ours.

The different ideas can mainly be distinguished on how the slots are laid out and get accessed. The Bike Loft uses a grid of slots with a few rows and many columns. A long, linear track is used for the robot to move horizontally and the robot contains a hoist so that the load handling attachment can be moved up and down to access different rows of the grid. The Bike Loft system can therefore be extended in two dimensions.

Most other systems store the bicycles on vertical layers with a fixed number of slots per layer. We will now present two such types of systems and then also take a brief look at a system that uses a conveyor belt to move all the bicycles at once.

4.1 Cylindrical Storage Systems

Some systems use a cylindrical structure so that the slots in each layer are laid out around a circle. The robot is then in the centre of this circle moving up and down and rotating around its central axis to access all the slots. One such system is the underground bicycle parking system in Tokyo Japan, which is called ECO Cycle and is built by Giken Seisakusho Co. LTD [46]. Each of the 17 layers contains 12 storage slots so a total of 204 bicycles can be stored. These layers extend up to 11 meters below ground so that only the door on top is visible on the street level.

There are also systems that stack the circular layers in a tower above the door, like the Locker Bikestation by LS Logistics [45] or the Bike-Tower by Stone Man-
4. Other Bicycle Storage Systems

ufaktur GmbH in Meckenbeuren Germany [31]. A similar cylindrical tower is also used for automated car parking in the T-Park by Skyline Parking in Zurich Switzerland [4].

Compared to our one-dimensional models in Chapter 2, there are many similarities: There is still a single door and a single robot with the door located at one end of the robot’s linear track. One free slot is required for every arriving bicycle and these systems usually do not rearrange the bicycles once they are put into storage. The main differences are the availability of multiple slots per layer and the box-less storing of objects, i.e., no empty box has to be brought to the door before the customer arrives.

Let us assume that the time required for the rotation is negligible in comparison to the vertical movement. In that case, these systems are quite similar to our Swap robot model, where we also did not need to worry about finding empty boxes, but extended to feature multiple boxes per slot. Our models for SwapArrival, SwapDeparture, SwapPersonalSlot and SwapSlotReuse all did not depend on a single storage slot at each location. All we needed for solving these problems were the distances between the door and all the slots and the assumption that accessing two different locations before returning to the door takes time proportional to the maximal distance of the two locations. The cylindrical systems satisfy both of these properties. Similarly, also the sorting model on paths from Section 3.2.2 could be extended to such a model with multiple slots per vertex.
A slightly different approach was chosen for the Radhaus, a system developed by Nussbaum Technologies GmbH [30] and used by dozens of daily commuters at the train station in Offenburg Germany. Instead of handling the bicycles individually, they are put onto pallets that can hold up to twelve bicycles each. The system consists of 10 such pallets which are stored on five floors of two pallets each. An additional ground floor is used for the interaction with the customers. A central forklift can fetch one of the pallets at a time and can put it on the ground level or it can put a pallet from the ground level into one of the 10 storage locations. At the ground floor, every one of the 12 slots on the pallet is accessible by a separate door, so all slots on the pallet can be accessed simultaneously. It takes roughly two minutes to exchange the pallet on this ground floor, regardless of which one of the pallets is fetched next.

Let us also compare this system to our models in the previous chapters. Unlike most cylindrical systems, the Radhaus uses some kind of box, namely these pallets, which the robot has to bring to the doors before a customer can hand off his bicycle. The main differences are the size of up to 12 bicycles per box/pallet in the Radhaus and the constant amount of time required for swapping two pallets.

Having such *uniform access time* would trivialize our previous arrival-only
problems where we had only a single bicycle per box, like $\text{SwapArrival}$. A wait-free schedule would then be possible whenever no two customers arrive within the time it takes to perform a swap. In the setting of the Radhaus with multiple bicycles per slot, the arrival-only is still interesting though, even with this uniform access time.

Let us assume it takes $t$ seconds to swap a pallet and that each of the $p$ pallets can store up to $k$ bicycles. All customers that arrive less than $t$ seconds after their predecessor would need to store their bicycle on the same pallet as the previous customer. This constraint would cluster the sequence of customers into groups, where all customers of a group need to go to the same pallet. In between such groups, with an interrupt of at least $t$ seconds, we can switch to any other pallet. A wait-free arrival-only schedule is therefore only possible if the customer groups can be partitioned into at most $p$ partitions of at most $k$ customers each. Depending on how we let number or the size of the pallets grow in comparison with the number of costumers, this arrival-only problem can get hard very quickly. For instance, if we are given $p, k$ and the size of each group of costumers as an input, so with the customers already clustered into groups, this is already a generalization of the $\text{NP-complete}$ problems $\text{Partition}$ and $\text{3Partition}$ [26]. However, if the number of costumers is polynomially bounded and we have a constant number of pallets or a constant size for each pallet, interesting options for dynamic programming approaches might arise. Mixing arrival and departures would of course also be interesting for the model of this Radhaus.

### 4.3 Conveyor Belt Systems

Another intriguing approach is the use of a conveyor belt, which, like the Radhaus, allows moving multiple bicycles at once - maybe even all of them. In the Dutch system *Velominck*, engineered by Lo Minck Systemen BV, the bicycles are hooked up to such an overhead conveyor belt [14]. The conveyor belt can carry up to 100 bicycles along the 50 meter long loop. At one designated location a robotic arm can attach and release the bicycles from the loop and take it to a door facing the customer. The conveyor belt itself can either be placed below ground, like in a garage at the main station in Amsterdam, or in a building above ground, like in an installation in Münster Germany.

The arrival-only setting for such a conveyor belt is not very interesting. If we simply move the belt slot by slot to the left, we always have another free spot ready for the next arriving customer in the shortest possible time. In contrast, looking at both arrivals and departures would lead to a very interesting problem where our results from the linear storage systems in Chapter 2 cannot be applied directly.
Also some automated car parking systems were based on conveyor belt techniques. Most often, the cars are placed into boxes that rotate around a vertical loop, similar to a Ferris wheel. This is especially attractive in urban areas, where the small footprint of such a system is an important advantage. A fascinating example of such a paternoster-style system was built in Chicago in 1932. We encourage the reader to take look at the delightful report by Pathé News [5] about this historic car parking machine.
In this thesis, we have introduced formal models and algorithms for request scheduling and sorting operations in robotic warehousing systems. We abstracted an existing automated bicycle storage system to arrive at challenging problems of minimizing the customer waiting time. We studied a novel sorting problem on a variety of different graph classes and surveyed other sorting problems for physical objects and other bicycle warehousing systems.

This chapter summarizes our contributions and points to some interesting open problems.

5.1 Summary of Contributions

In Chapter 2 we introduced a simple model for an automated warehouse. A single robot moves along a line of \( n \) boxes with a customer-facing door at one end. The robot knows all customer requests in advance. We explored the question of finding a customer-to-slot assignment that allows the robot to serve all the customers without causing any waiting time. We first looked at the case where customers only fill their boxes and never come back to empty them. We showed that finding a customer-to-slot assignment is \( \text{NP} \)-hard to find if the robot has to always return to the door in between storing a full box and fetching a new empty box. If we allow the robot to go directly from storing the full box to fetching an empty box we can give an efficient algorithm for finding such a schedule. If the robot is even able to directly swap out the box it is carrying, we can also find a schedule quickly. We also showed how much foresight is needed if we learn about the requests in an online setting.

In Section 2.3, we added a second door on the other end of our warehouse. We were able to adopt our algorithms to this two-door-setting if the cus-
5. Conclusion

tomers pick the door they approach. But using a reduction from 3SAT, we also showed that finding a feasible schedule is $\mathcal{NP}$-hard as soon as the robot can decide for every customer which door he has to use. We gave some basic approximation algorithms for this problem in which we tried to serve as many customers as possible without accumulating any delays.

We then went on to allow the customers to also reclaim their goods in the case of a single door. If all arrivals occur before all departures and the robot has time to rearrange the boxes in between, we can find a good schedule efficiently. The same holds if the arrivals and departures are arbitrarily interleaved but every storage slot is used by at most one customer. If we start reusing slots for multiple customers, finding a wait-free schedule is $\mathcal{NP}$-complete again, which we showed via a reduction from circular arc colouring.

In Chapter 3, we focused on the task of rearranging the stored boxes, ignoring the customer interaction. We introduced the problem of finding shortest sorting walks for a robot that can move at most one box at a time and wants to sort a given permutation of $n$ boxes.

By induction on the cycles of this permutation, we were able to give a nice algorithm to solve this problem for paths where the robot starts at the border. A second algorithm even allowed the robot to sort efficiently if it does not know the permutation beforehand and only has a constant amount of memory available.

We then extended the layout of the warehouse to more general graphs. Our most interesting result is that we can also find shortest sorting walks for any permutation on any tree. We use an auxiliary graph to track how we connect the cycles of the permutation and use an optimum branching on a complete directed graph to find the best way to do so. If the graph is a single circle, a lengthy case distinction allowed us to describe an optimal sorting procedure as well.

In contrast, the shortest sorting walk is $\mathcal{NP}$-hard to find in general, even on planar graphs. A straightforward reduction from the problem of finding a Hamilton tour on a grid graph, allowed us to prove this.

In Section 3.3, we stopped focusing solely on the number of steps that the robot performs while sorting and minimized as a first priority the number of boxes that need to be picked up. For path graphs we still found an efficient algorithm. But on trees it is already $\mathcal{NP}$-complete.

We ended these two chapters with a summary of related work and presented two related sorting puzzles. Chapter 4 surveyed other automated bicycle storage systems and compared them to our models.
5.2 Future Work

At several points in this thesis we came across interesting questions that we could not answer and that might stimulate future research.

It is not known whether the TightSum problem can be solved in time $o(n^2)$. Some first experiments with random permutations suggested that the TightSum algorithm that we gave in Lemma 2.7 might run even in average time $O(n \log n)$.

For the arrival-only two-door problem, it could lead to challenging new problems if one looks at a different approximation model than we did. An interesting one would be to enforce that all requests get served but the accumulated wait time should be minimized. Corollary 2.28 implies that probably no efficient algorithm can achieve any approximation ratio in that model but one might still find bounds on the total amount of delay caused by a certain algorithm. Allowing for resource augmentation, in which multiple boxes can get stored at each slot, would be another compelling model. Knowing that there is a schedule that uses every slot at most once, the goal would then be to minimize the maximal load of each slot. In the request skipping model that we studied in Section 2.3.3, the main open problem is to show lower bounds better than $\frac{1}{2}$ for any of our deterministic approximation algorithms. The fact that our upper bounds came from an automated search for counter examples suggests that the true approximation ratio is probably closer to our current upper bounds.

For the problem of finding the shortest sorting walk, we were able to show optimality of our algorithms on paths, trees and cycles. Some interesting graph classes of yet unknown complexity would be ladder graphs, wheel graphs, hypercubes and other graphs with multiple cycles. All our results on paths, trees and circles easily extend to weighted graphs where each edge has an individual travel time. It is also open whether there are approximation algorithms for general graphs that are better than our simple bound in Lemma 3.4.

Furthermore, it would be intriguing to study variations of the sorting problem. What if the robot could pick up multiple boxes at a time? Could some of the locations of the warehouse just as well be empty? What if $\pi$ is not a permutation but just a general function? How could multiple robots collaborate?

Finally, we noticed when looking at other automated bicycle storage systems that some of them can also lead to enticing algorithmic challenges. We specifically want to mention the partitioning problems caused by the uniform access assumption for the Radhaus and the setting with interleaved arrivals and departures for conveyor belt systems.


[21] DMIH. Dani Mladih Informatičara Hrvatske. [http://www.hsin.hr/ dmih06/](http://www.hsin.hr/ dmih06/).


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

__________________________________________________________________________

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Scheduling and Sorting Algorithms for Robotic Warehousing Systems

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s): Graf

First name(s): Daniel

With my signature I confirm that
− I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
− I have documented all methods, data and processes truthfully.
− I have not manipulated any data.
− I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Zurich, July 17, 2015

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.