Noname manuscript No.
(will be inserted by the editor)

How to Sort by Walking and Swapping on Paths and Trees

Daniel Graf

Received: date / Accepted: date

Abstract Consider a graph G with n vertices. On each vertex we place a box. The n vertices and n boxes are each numbered from 1 to n, and initially shuffled according to a permutation π. 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 or a tree, and we show that the problem is $\mathcal{NP}$-complete for planar graphs. If we minimize the number of swaps in addition to the number of walking steps, it is $\mathcal{NP}$-complete even if G is a tree.

Keywords Physical Sorting, Shortest Sorting Walk, Warehouse Reorganization, Robot Scheduling, Robot Transportation Problem, Permutation Properties, Ensemble Motion Planning

1 Introduction

Motivation. Nowadays, many large warehouses are operated by robots. Such automated storage and retrieval systems are used in industrial and retail warehouses, archives and libraries, as well as in automated car or bicycle parking systems. When it needs to rearrange the stored goods, such a robot faces a physical sorting task. In contrast to standard sorting algorithms, it does not have constant time access to the stored objects. It might need to travel for

An extended abstract of this article appeared in the Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15) [17].

D. Graf
ETH Zürich, Department of Computer Science
E-mail: daniel.graf@inf.ethz.ch

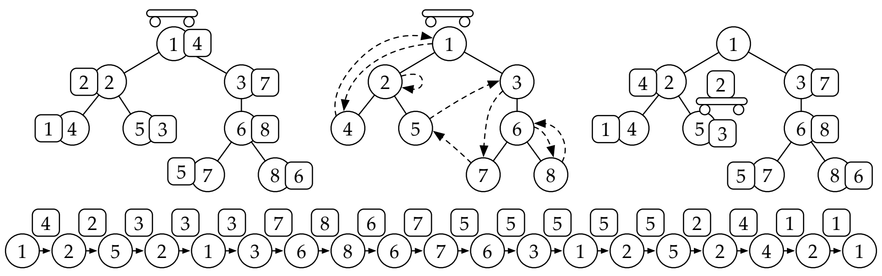

Fig. 1 (left) Initial state of the warehouse with storage locations as circles and boxes as squares. The box at vertex i is labelled with its target vertex $\pi(i)$. (centre) The initial state with π drawn as dashed arcs towards their target vertex instead of numbered boxes. (right) This shows the state of the warehouse after two steps have been performed. First the robot brought box 4 to vertex 2. Then it took box 2 to vertex 5. (bottom) A shortest possible sorting walk consisting of 18 steps.

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.

Our interest in this problem arises from a bike parking system to be built in Switzerland [35]. At a train station, commuters bring their bicycles in the morning and reclaim them in the evening. The commuters place their bicycles in a box at a door of the robotic parking system, where a robot then moves the box underground for safe storage. At some points in time, the robot should rearrange these bike boxes according to the expected pickup times of the customers. An abstraction of this sorting process is what we want to study in this paper. 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.

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?

Example Figure 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 shortest 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.

Organization Section 2 formally defines the problem and introduces some terminology. We then study the problem of minimizing the length of a shortest sorting walk in Section 3. In Section 3.1, we show some lower and upper bounds on general graphs. We then focus on paths in Section 3.2. If the robot starts at one of the ends of the path, there is a simple and elegant way to compute a shortest sorting walk in linear time which we demonstrate in Section 3.2.1. As we show in Section 3.2.4, we can even sort optimally if the robot has only a constant memory size and has to discover the permutation of the boxes in an online fashion. In Section 3.2.5, we allow for an arbitrary starting position of the robot on the path. For this case, we give a quadratic time dynamic programming construction and then use some observations to improve its runtime to linear time. Our main result is given in Section 3.3, where we construct shortest sorting walks on arbitrary trees with arbitrary starting positions in quadratic time. In Section 3.4, we show that it is $\mathcal{NP}$-complete to find a shortest sorting walk for planar graphs. In Section 4, we look at the problem of minimizing the number of box handling interactions as the first priority while still minimizing the travel time. We show that this sorting problem remains solvable on path graphs but is $\mathcal{NP}$-complete on tree graphs. Section 5 points the interested reader to an implementation and visualization of the presented algorithms that we make available online.

Finally, in Section 6, we give a detailed summary of related work. After having published our results in [17] and [18], we found out that many of our findings are independent rediscoveries of results by Frederickson and others. Therefore, we explicitly compare and contrast their results with ours in order to guide the reader through the many different settings that were studied before and to highlight the novelty in our contribution.

2 Problem Description and Notation

We consider the following model throughout this paper. 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, undirected, unweighted 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 π 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, we look for a 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 4, we minimize the number of these box-swapping actions as the first priority while still minimizing the length of the sorting walk as the second priority.

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

We start with $\tau_0 = (r, \square, \pi)$, 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 $v_t = p$, $b_t = b$. σ_t only differs from σ_{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 $S = (s_1, \dots, s_l)$ is a *sorting walk* of length l if we start with τ_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 1 (GraphSort) Given a graph $G = (V, E)$, a starting vertex $r \in V$, and a permutation π on V, we let GraphSort denote the problem of finding a shortest sorting walk to sort π on G with the robot starting at r.

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

3 Minimizing the Travel Time

3.1 General Bounds

We distinguish two kinds of steps in a sorting walk: *essential* and *non-essential* steps.

Definition 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 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 1 (Lower bound by counting essential steps)
Every sorting walk for a permutation π on a graph G is of length at least $d(\pi) = \sum_{b \in \{1,\dots,n\}} d(b, \pi(b))$.

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 $d(\pi) = \sum_{b \in \{1,\dots,n\}} d(b, \pi(b))$ essential steps in every sorting walk and at least as many steps overall. $\square$

The challenge remaining is to minimize the number of non-essential steps. In case that π 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 1 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 vertex v with a 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 v. Once it arrives at v, it sorts the cycle in which v and b are 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 2 (Upper bound from traversal) *There is a sorting walk of length at most $d(\pi) + 2 \cdot (n-1)$ for a permutation π on a graph G.*

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. □

We can see that these sorting walks might not be optimal, for instance in the example shown in Figure 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 section we want to find sorting walks that are as short as possible.

3.2 Sorting on a Path

We now look at the case where G is a path $P = (V, E)$. The vertices v_1 to v_n are ordered on a line from left to right and every vertex is connected to its up to two neighbours, thus $E = \{\{v_i, v_{i+1}\} \mid i \in \{1, \ldots, n-1\}\}$.

Definition 3 (PathSort) We let PathSort denote the instances of GraphSort where the graph G is a 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. Let $\mathcal{I} = (\overline{\mathcal{C}}, \mathcal{E})$ be the intersection graph of the non-trivial cycles, so $\mathcal{E} = \{\{C_1, C_2\} \mid C_1, C_2 \in \overline{\mathcal{C}} \text{ s.t. } I(C_1) \cap I(C_2) \neq \emptyset\}$. We then use $\mathcal{D} = \{D_1, \ldots, D_{|\mathcal{D}|}\}$ to represent the partition of $\overline{\mathcal{C}}$ into the connected components of $\mathcal{I}$. 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 $i < j$.

3.2.1 Border Starting Position

We will first simplify further and assume that the robot is initially placed at one of the ends of the path, so let without loss of generality $r = v_1$. Intuitively, this simplifies the task of the robot as there is no choice between going left or right for its very first step and its last step.

Definition 4 (BorderPathSort) We let BORDERPATHSORT denote the instances of PATHSORT where the robot starts at one of the ends of the path.

Theorem 1 (Shortest sorting walk for BorderPathSort) *A shortest sorting walk on a path P with permutation π and starting position $r = v_1$ can be constructed in time $\Theta(n^2)$ and has length*

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

Proof What we claim is 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 π. We already saw how we can find a minimum sorting walk if π consists of a single cycle only. If there are several cycles but only one of them is non-trivial, so $|\mathcal{C}| > 1$ but $|\overline{\mathcal{C}}| = 1$, a 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 2 (left) gives an example of such a case.

Now let us look at the case where π 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 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

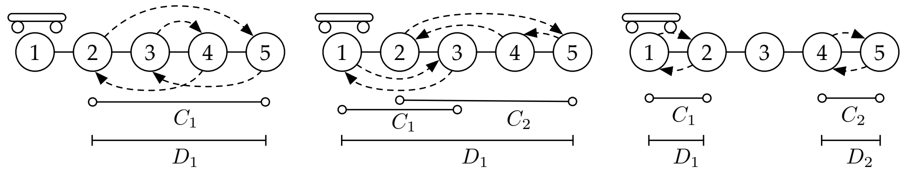

Fig. 2 (left) An example with a single non-trivial cycle. A shortest sorting walk S with $|S| = d(\pi) + 2 \cdot (l(C_1) - 1) = 8 + 2 \cdot (2 - 1) = 10$ is $((2,\Box),(3,5),(4,5),(5,5),(4,3),(3,3),(4,4),(3,2),(2,2),(1,\Box))$. (centre) An example with two intersecting cycles. A shortest sorting walk S with $|S| = d(\pi) = 10$ is $((2,3),(3,5),(4,5),(5,5),(4,4),(3,2),(2,2),(3,3),(2,1),(1,1))$. (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 $((2,2),(3,1),(4,1),(5,5),(4,4),(3,1),(2,1),(1,1))$.

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 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 C_1 to C_i. 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 after 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. □

3.2.2 Algorithmic Construction

The proof of Theorem 1 immediately tells us how we can construct a minimum sorting walk efficiently. Given P and π we first extract the cycles of π 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.

With these references, 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.

3.2.3 Compact Representation of the Sorting Walk

With a slight change in representation, we can describe also the sorting walks of quadratic length in linear size and then also find them in linear time. Moving a box b from u to v takes $d(u,v)$ many consecutive steps that all move the same box b. We will now represent these steps with a single *compact* step $s^c = (b, v)$, which represents that the robot takes box b on the shortest path to vertex v. We denote such a *compact* representation of S as S^c and let $|S^c|$ denote the number of compact steps in S. For example, the compact representation of the 18-step sorting walk in Figure 1 only requires 11 compact steps.

Lemma 3 *The shortest sorting walk for* BorderPathSort *from the algorithm in Theorem 1 has a linear-size compact representation, so $|S^c| \in \mathcal{O}(n)$.*

Proof Every non-correctly placed box is picked up exactly once from its initial position. Its move towards its target position is only interrupted if the sorting of another cycle has to be started (which happens only once per cycle) or if a gap between two connected components of cycles has to be bridged (which happens at most twice per connected component). As the number of boxes and hence also the number cycles and gaps between connected components is linear in n, the box placed on the robot only changes $\mathcal{O}(n)$ many times and therefore only $\mathcal{O}(n)$ many compact steps are required. □

Corollary 1 *The length and the compact represention of a shortest sorting walk for* BorderPathSort *can be computed in time $\mathcal{O}(n)$.*

Proof All the steps in the proof of Theorem 1, especially keeping track of the first step that arrives at any vertex and inserting new steps into the current sorting walk, can be realized in amortized constant time per box when implemented with a compact representation. The length of the sorting walk can then be computed in linear time from its compact representation or by evaluating the formula of Theorem 1.

3.2.4 Online Algorithm for BorderPathSort *with Constant Memory*

Before we move on to other starting positions and other graphs, we want to answer two more questions about shortest sorting walks on paths with $r = v_1$.

– Does it help to know the entire permutation in advance or not?
– Do we need to precompute the entire sorting walk or can we compute it on the fly and remember only very little information at any point?

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

Definition 6 (Constant Memory) A sorting algorithm uses *constant memory* if the robot at every state of the sorting process remembers only constantly many integers of size $\leq n$. This corresponds to $\mathcal{O}(1)$ cells in the word-RAM model or any amount of information that can be representend with $\mathcal{O}(\log n)$ bits. Phrased differently: the robot only distinguishes $\operatorname{poly}(n)$ many internal states.

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 at the end of the path that is opposite of the starting position are already sorted. An offline algorithm can just ignore these presorted 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 2 *There is an online algorithm with constant memory who finds a sorting walk of only $2 \cdot (n - r(D_{|\mathcal{D}|}))$ more steps than a shortest sorting walk.*

Proof Our algorithm works in two phases: a *left-to-right* phase and a *right-to-left* phase. The robot only keeps track of three values: 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 sufficient and therefore 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. Intuitively, this connects all the initial components of cycles which will allow us to easily sort all cycles in the second phase. After the first 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.

In the second phase, the robot will first 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

[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.

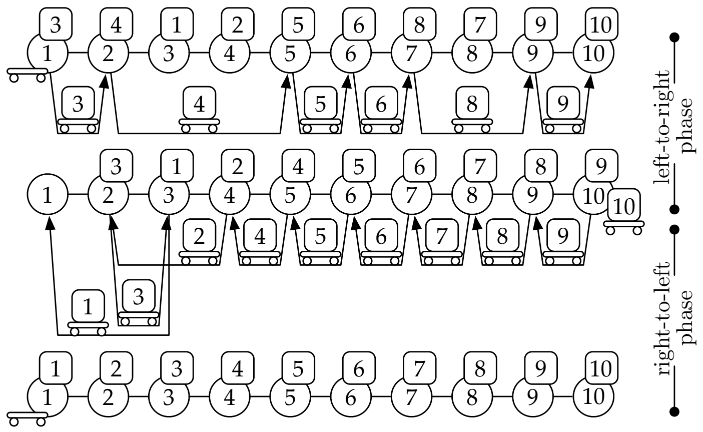

Fig. 3 Example for the $\mathcal{O}(1)$-memory online algorithm of Theorem 2. The first line shows the initial permutation π. Five steps of the left-to-right-phase are non-essential: the step where box 4 is brought to vertex 5, the two steps immediately thereafter and the last two steps. These last two steps of this phase as well as the first two steps of the right-to-left phase would have been prevented by our offline algorithm of Theorem 1 as it would have known without checking that box 9 and box 10 are already placed correctly from the beginning.

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 the 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. See Figure 3 for an example.

Why does this strategy 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 this never happens. Why? Assume otherwise, so consider the following: While looking for a box x and carrying box y, we are reaching slot y. If we have not found box x yet and y is the smallest box encountered so far, we would continue carrying y to left removing it from its target position. This can only happen if all the boxes at the slots y to x are larger than y and none of them is box x. But as the only empty slot is slot 1, this is impossible: there are only $x-y$ boxes for these $x-y+1$ many slots.

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. Therefore, there are equally many non-essential steps in both phases of our algorithm and the total number of steps is as in Theorem 1 plus an extra $2 \cdot (n - r(D_{|\mathcal{D}|})$. $\square$

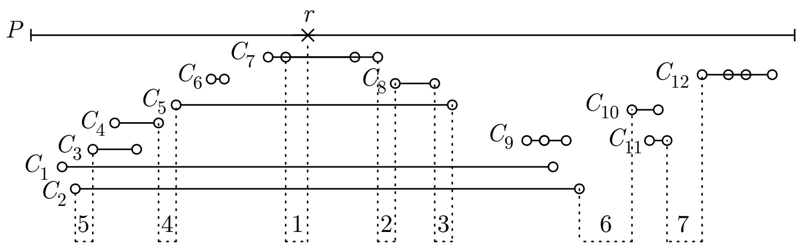

Fig. 4 Example of several nested cycles on a path P with a non-border starting position. Every horizontal segment represents a non-trivial cycle of π with the small circles depicting its boxes. The robot has to decide multiple times whether to spend non-essential steps to the left or to the right to extend its current set of connected cycles. The seven dashed lines on the bottom illustrate a minimal way to connect the cycles. To first reach C_7, it is closest to go to the left until we reach the first box of C_7. To get to C_5, we can sort C_8 along the way so that only a very short parts (dashed lines 2 and 3) have to be travelled non-essentially. Next, we can extend again on the left side so that we can exploit C_4 and C_3 in order to reach C_1, C_2 and C_9 quickly. Finally, we have to spend some more non-essential steps on the right to reach C_{10}, C_{11} and C_{12}.

3.2.5 Non-border Starting Position

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 BorderPathSort 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. In particular, it is no longer sufficient to use non-essential steps exclusively to bridge connected components of overlapping cycles. We might now also need non-essential steps within the components as the robot first needs to reach some box of a cycle that overlaps with a box of another cycle. Inside a connected component, it now matters *how* the cycles are nested and overlap. Figure 4 gives an example of such nested cycles where the robot has to decide multiple times whether to extend its connected region to the left or to the right.

In some respect, this section where we still sort on paths is a special case of our results in Section 3.3 where we will study the problem on trees and will solve it in quadratic time. For paths, however, we can exploit a special structure of the shortest sorting walks, which enables us to improve a first quadratic solution (Lemma 4) to a linear one (Theorem 3).

Lemma 4 (Quadratic Dynamic Programming Approach) *We can compute the length of a shortest sorting walk on a path P with permutation π and arbitrary starting position r in time $\mathcal{O}(n^2)$ using dynamic programming.*

Proof Let $I = [a, b]$ be an interval of the path P containing r that the robot already explored. We call such an interval *self-contained* if and only if no cycle of π partially intersects with I. This means that for any cycle C either all its boxes are in I or none of its boxes are in I. For example, in Figure 4 the

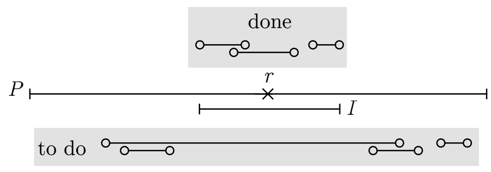

Fig. 5 Example of a self-contained interval I that splits the set $\overline{\mathcal{C}}$ into cycles within I that have been *done* (connected to r) and all others that still need to be connected.

following intervals are self-contained: $I(C_7)$, $I(C_5)$ and $I(C_1) \cup I(C_2)$. But on their own, $I(C_1)$ and $I(C_2)$ are not self-contained. See Figure 5 for illustration.

We now define the function $\texttt{extend}(a, b)$ which returns for any interval $[a, b]$ the smallest self-contained interval $[a', b']$ with $[a, b] \subseteq [a', b']$. Therefore, $\texttt{extend}(a, b)$ is the largest part of the path that we can sort without spending any additional non-essential steps. To compute `extend`, we have to repeatedly add all the cycles C that have boxes both inside and outside of $[a, b]$. So for any such cycle C, we continue with $[a, b] := [a, b] \cup I(C)$ until no more cycles can extend the interval for free.

We start with $[a, b] = \texttt{extend}(r, r)$. Now the fundamental problem remains that we have to somehow decide whether it is cheaper to extend a self-contained interval to the left or to the right. We could try out both options and define the following subproblem:

How many non-essential steps do we still need if we already know how to connect all the cycles in the self-contained interval $[a, b]$*?*

This gives rise to a *dynamic programming* formulation with a state of quadratic size (all intervals containing r). Let $\texttt{combine}(a, b)$ be the cost function for connecting all cycles to the interval $[a, b]$. We can recursively compute it using

$$\begin{aligned}\texttt{combine}(a, b) = 2 + \min(&\texttt{combine}(\texttt{extend}(a - 1, b)),\\ &\texttt{combine}(\texttt{extend}(a, b + 1))).\end{aligned} \tag{2}$$

We need to take care of the border cases (when $a - 1$ or $b + 1$ are outside the path) and initialize with $\texttt{combine}(a^*, b^*) = 0$ for $[a^*, b^*]$ being any interval that contains all non-trivial cycles and r. By implementing `extend` carefully (by lazily only checking if the cycles at a and b extend outside of $[a, b]$), we achieve amortized constant time across all its calls so that we compute the final length $d(\pi) + \texttt{combine}(r, r)$ in quadratic time overall. □

Theorem 3 (Shortest sorting walk for PathSort) *We can compute the length and the compact represention of a shortest sorting walk on a path P with permutation π and arbitrary starting position r in time $\mathcal{O}(n)$.*

Proof To solve the problem in linear time, we have to observe that we can decide locally whether to go left or right without exploring quadratically many states. Imagine that we explored a self-contained interval I and now want to extend it to the left. Let C be the first non-trivial cycle that we encounter this

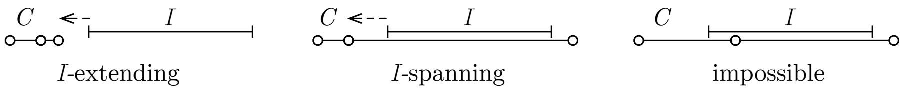

Fig. 6 Distinction between C being I-extending and I-spanning. Note that C cannot have any box within I as we assume that I is self-contained.

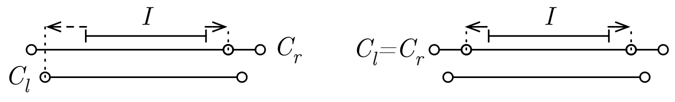

Fig. 7 Two scenarios where the next I-spanning cycles C_l and C_r to the left and right of I are either different or the same.

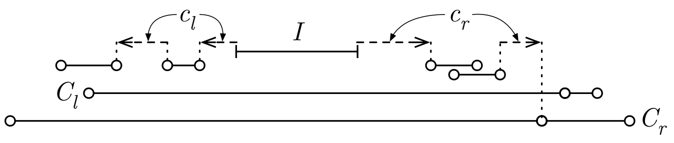

Fig. 8 Example illustrating the costs c_l of extending I to the left to the first I-spanning cycle C_l and analogously the costs c_r to reach C_r.

way. See Figure 6 for an illustration. We now call this cycle either *I-extending* if its interval $I(C)$ lies completely to the left of I or *I-spanning* if it completely contains I. Note that a partial intersection of I and C is not possible as this contradicts I's self-containment.

Let C_l (resp. C_r) be the first I-spanning cycle that we encounter when walking from I to the left (resp. right). Note that C_l and C_r might or might not be the same cycle, but if one of them exists, so does the other. See Figure 7 for an example. Let us assume that C_l and C_r exist for our current interval I. The key insight is now to see that it is optimal to extend I on only one of the two sides until we reach an I-spanning cycle (either C_l or C_r). So we will never need to extend on both sides to reach the next spanning cycle. Why? We know that C_l and C_r overlap (as $I \subseteq I(C_l) \cap I(C_r)$), hence once we are sorting either one of the two cycles, we can also sort the other for free and extend I to the same self-contained interval $\texttt{extend}(I(C_l) \cup I(C_r))$ at no extra cost. As non-essential steps on the right side of I do not help us reaching C_l and vice versa, we can focus on the two sides individually. So all we need to compute is the number of non-essential steps c_l (resp. c_r) required to get form I to a box of C_l (resp. C_r). We greedily take the cheaper of both options and continue with $I' = \texttt{extend}(I(C_l) \cup I(C_r))$. Figure 8 illustrates these two costs.

The cost c_l can be computed as follows. Let v_{C_l} be the first vertex of C_l to the left of $l(I)$. On the way from $l(I)$ to v_{C_l}, we have to spend non-essential steps across the edges that are not covered by any I-extending cycle. Let u_l denote the number of these uncovered edges. u_l can be computed using a linear sweep from $l(I)$ to v_{C_l} in the same way that we compute the gaps between connected components for BorderPathSort instances in Theorem 1.

We then have $c_l = 2u_l$ and analogously $c_r = 2u_r$. This allows us to speed up the binary recursion of (2) to the linear recursion

$$\texttt{combine}(I) = \min(c_l, c_r) + \texttt{combine}(\texttt{extend}(I(C_l) \cup I(C_r))). \tag{3}$$

The sweeps to compute all the u_l and u_r in every step take only linear time overall as we traverse every edge of P at most once across all the sweeps and as we can precompute in advance all the necessary lookup values. The sweeps to compute all the u_l and u_r in every step take only linear time overall as we traverse every edge of P at most once across all the sweeps and as we can precompute in advance all the necessary lookup values.

In the end, once no more I-spanning intervals exist, we might still have to sort some I-extending cycles on either or both sides (as with C_{10}, C_{11} and C_{12} in Figure 4). These are just one or two instances of BorderPathSort. With the same technique of keeping track of the first step that arrives at any vertex as in Corollary 1, we can compute both the length of a shortest sorting walk as well as a compact sorting walk S^c of linear size in linear time. □

3.3 Sorting on a Tree

We now give our main result, the generalization from paths to trees. 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. Recall Figure 1 for an example of sorting on a tree.

Definition 7 (TreeSort) We let TreeSort denote the instances of GraphSort where the graph G is a tree.

Definition 8 (Cycle hitting and covering vertices) For any cycle C of π 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 $v \in T(C)$.

In Figure 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 π 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.

3.3.1 Cycle Anchor Trees

Definition 9 A *cycle anchor tree* $\widetilde{T}$ is a directed, rooted tree that contains one vertex $\widetilde{v}_C$ for every non-trivial cycle C of π and an extra root vertex $\widetilde{r}$.

Given a sorting walk S we construct from it a cycle anchor tree $\widetilde{T}$ as follows: We start with $\widetilde{T}$ only containing $\widetilde{r}$. We go through the essential steps in S. If step s is the first essential step for some cycle C, we create a vertex $\widetilde{v}_C$ in $\widetilde{T}$. To determine the parent node of $\widetilde{v}_C$ in $\widetilde{T}$ we look for the last essential step s' in S before s and its corresponding cycle C'. We now say that C is *anchored* at C' and add an edge $(\widetilde{v}_{C'}, \widetilde{v}_C)$ to $\widetilde{T}$. If no such step s' exists (which only happens for the very first essential step in S) we use the root $\widetilde{r}$ as the parent of $\widetilde{v}_C$.

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 $\widetilde{T}$ is now defined as follows: Let $c((\widetilde{v}_{C_1}, \widetilde{v}_{C_2}))$ 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 $c((\widetilde{r}, \widetilde{v}_C))$ being the minimum number of down-steps on the path from the root to any vertex $v \in V(C)$. The cost $c(\widetilde{T})$ of an entire cycle anchor tree $\widetilde{T}$ is simply the sum of its edge costs. Figure 9 illustrates the definitions and gives an example of the transformation from a sorting walk to a weighted cycle anchor tree.

Theorem 4 (Lower bound for trees) *Any sorting walk S that sorts a permutation π on a tree T and corresponds to a cycle anchor tree $\widetilde{T}$ has length at least $d(\pi) + 2 \cdot c(\widetilde{T})$.*

Proof We partition the steps of S into three sets: essential steps $\mathcal{S}_e$, non-essential down-steps $\mathcal{S}_{n,d}$ and non-essential up-steps $\mathcal{S}_{n,u}$. From Lemma 1 we have $|\mathcal{S}_e| = d(\pi)$. We argue that S contains at least $c(\widetilde{T})$ many non-essential down-steps. To do this we look at the segments of S that were relevant when we described how we derive $\widetilde{T}$ from S. For an edge $(\widetilde{v}_{C_1}, \widetilde{v}_{C_2})$ of $\widetilde{T}$, 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

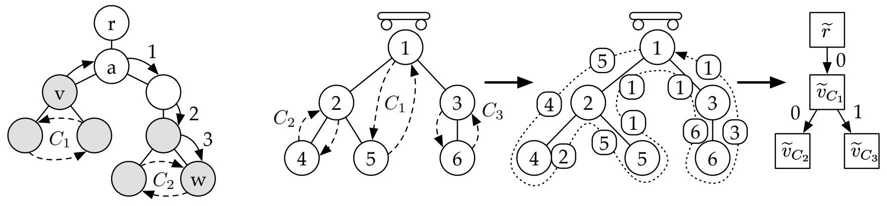

Fig. 9 (first figure on the left) The two pairs of dashed arrows symbolize boxes that need to be swapped. A shortest path from any $v \in T(C_1)$ to any $w \in V(C_2)$ is shown with continuous arrows, three of them being down-steps, so $c((\widetilde{v}_{C_1}, \widetilde{v}_{C_2})) = 3$. The anchor vertex a is the vertex immediately before the first down step. Note that c is not symmetric as $c((\widetilde{v}_{C_2}, \widetilde{v}_{C_1})) = 2$. (the three figures on the right) An example of a sorting walk on a tree with three non-trivial cycles. The dashed arrows on the left show the desired shuffling of the boxes. The dotted arrow in the middle shows a minimum sorting walk of ten steps, where each step is labelled with the box it moves. On the right, the corresponding cycle anchor tree is given. The edge from $\widetilde{v}_{C_1}$ to $\widetilde{v}_{C_3}$ has cost 1 as there is a down-step necessary to get from vertex $1 \in T(C_1)$ to vertex $3 \in V(C_3)$. The edge $(\widetilde{v}_{C_1}, \widetilde{v}_{C_2})$ is free as vertex 2 is both in $T(C_1)$ and $V(C_2)$.

(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 $c(\widetilde{v}_{C_1}, \widetilde{v}_{C_2})$, there are at least $c(\widetilde{v}_{C_1}, \widetilde{v}_{C_2})$ many down-steps in S_{C_1,C_2}. The same holds for the initial segment $S_{r,C}$. As all these segments of the sorting walk are disjoint, we get that $|\mathcal{S}_{n,d}| \geq c(\widetilde{T})$.

Finally we argue that $|\mathcal{S}_{n,d}| = |\mathcal{S}_{n,u}|$ 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 π 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. □

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.

3.3.2 Reconstructing a Sorting Walk

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

Theorem 5 (Tight construction) *Given T, π and cycle anchor tree $\widetilde{T}$, we can find a sorting walk of length $d(\pi) + 2 \cdot c(\widetilde{T})$.*

Proof We perform a depth-first search traversal of $\widetilde{T}$, starting at $\widetilde{r}$ 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 $\widetilde{T}$ from $\widetilde{v}_C$ to $\widetilde{v}_{C'}$, we do the following: Let $v \in T(C)$ and $w \in V(C')$ be the two vertices that have the minimum number of down-steps between them, as in the definition of the edge weights of $\widetilde{T}$. 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 $c(\widetilde{v}_C, \widetilde{v}_{C'})$ down-steps in the first part, then $d(C')$ steps to sort C', and finally $c(\widetilde{v}_C, \widetilde{v}_{C'})$ 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 π and be of length $d(\pi) + 2 \cdot c(\widetilde{T})$. □

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

3.3.3 Finding a Cheapest Cycle Anchor Tree

Let S^* denote a shortest sorting walk for T and π. Using Theorem 5 to find S^* (or another equally long sorting walk), all we need is its corresponding cycle anchor tree $\widetilde{T}^*$. It suffices to find any cycle anchor tree with cost at most $c(\widetilde{T}^*)$. Especially, it suffices to find a cheapest cycle anchor tree $\widetilde{T}_{\min}$ among all possible cycle anchor trees. We then use Theorem 5 to get a sorting walk $S_{\min}$ from $\widetilde{T}_{\min}$. As $c(\widetilde{T}_{\min}) \leq c(\widetilde{T}^*)$ we get

$$|S_{\min}| = d(\pi) + 2 \cdot c(\widetilde{T}_{\min}) \leq d(\pi) + 2 \cdot c(\widetilde{T}^*) \leq |S^*| \quad (4)$$

and therefore $S_{\min}$ is a shortest sorting walk. To find this cheapest cycle anchor tree, we build the complete directed graph $\widetilde{G}$ of potential anchor tree edges. Note that the weights of these edges only depend on T and π but not on a sorting walk.

Optimum Branching Given this complete weighted directed graph $\widetilde{G}$ we find its minimum directed spanning tree rooted at $\widetilde{r}$ using Edmond's algorithm for optimum branchings [8]. A great introduction to this algorithm, its correctness proof by Karp [27] and its efficient implementation by Tarjan [33] can be found in the lecture notes of Zwick [37]. Combining these results with Theorem 5 will now allow us to find shortest sorting walks in polynomial time, so without enumerating all possible cycle anchor trees.

Theorem 6 (Algorithm for TreeSort) *We can find a shortest sorting walk for any tree T with permutation π and starting position r in time $\mathcal{O}(n^2)$.*

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) $|\overline{\mathcal{C}}| + 1$ times, starting once with r and once with $T(C)$ for every $C \in \overline{\mathcal{C}}$ and count the number of down-steps along these BFS trees. We also precompute all the anchor points. As we run $\mathcal{O}(n)$ many BFS traversals, this precomputation takes time $\mathcal{O}(n^2)$.

As an efficient implementation of Edmond's algorithm allows us to find $\widetilde{T}_{\min}$ in time $\mathcal{O}(n^2)$, we can find $S_{\min}$ in time $\mathcal{O}(n^2)$ time overall. In every step of the construction in Theorem 5, 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 we can construct the non-compact shortest sorting walk $S_{\min}$ in time $\Theta(n + |S|)$ from $\widetilde{T}_{\min}$. Combining these three steps gives an algorithm that runs in time $\mathcal{O}(n^2)$. □

Note that unlike on the path, the quadratic runtime is not solely caused by the size of the output. Hence even finding a compact sorting walk takes quadratic time unless we improve the computation of $\widetilde{T}_{\min}$,

3.4 Sorting on General Graphs

In this section, we show that our efficient algorithms for paths and trees can not be easily extended to solve the general GraphSort problem. In fact, no efficient algorithm for general graphs can be found unless $\mathcal{P}$ equals $\mathcal{NP}$.

Theorem 7 ($\mathcal{NP}$-completeness for planar graphs) *Finding a shortest sorting walk for a planar graph $G = (V, E)$ and permutation π is $\mathcal{NP}$-complete.*

Proof We show a reduction from the problem of finding Hamiltonian circuits for grid graphs. This problem was shown to be $\mathcal{NP}$-complete by Itai et al. in Theorem 2.1 of [23]. The main idea is to replace each vertex of the grid by a pair of neighbouring vertices with swapped boxes. Given a grid graph $\widehat{G} = (\widehat{V}, \widehat{E})$, we build the following input for the sorting problem:

$$V = \{v \mid \forall v \in \widehat{V}\} \cup \{v' \mid \forall v \in \widehat{V}\} \tag{5}$$

$$E = \{(v, v') \mid \forall v \in \widehat{V}\} \cup \widehat{E} \tag{6}$$

$$\pi(v) = v' \ \forall v \in \widehat{V} \text{ and } \pi(v') = v \ \forall v \in \widehat{V} \tag{7}$$

We can use any vertex in V as the starting vertex for the robot. An illustration of this transformation is given in Figure 10.

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

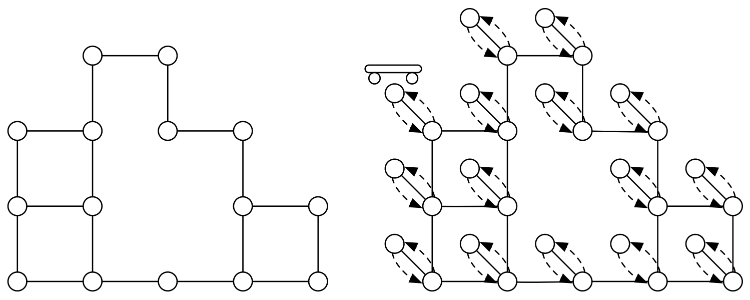

Fig. 10 (left) An input to the Hamiltonian circuit problem on a grid graph. (right) The corresponding input to the sorting problem.

exactly $2n$ essential steps. As all essential steps will move along edges in $E \setminus \widehat{E}$, 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 $\widehat{E}$, visiting all vertices in $\widehat{V}$ and therefore build a Hamiltonian circuit for $\widehat{G}$. 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. □

4 Minimizing the Box Handling Time

So far, we only minimized the time that the robot spends driving and ignored the time needed to load boxes onto the robot. In practice, this box handling time is not negligible. The Bike Loft system [35] that serves as our original motivitation takes roughly 5 seconds to load or unload a box and can move at the speed of roughly $3\frac{\mathrm{m}}{\mathrm{s}}$. 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, we now study the problem of finding sorting walks with minimal number of box handling operations. As a second priority, we still minimize the number of steps in the sorting walk. So among all sorting walks with the minimum number of box-carrying-changes, we want to find the shortest one.

Definition 10 (Swap count) The *swap count* $\langle S \rangle$ of a sorting walk S denotes the number of boxes picked up during the walk. Formally

$$\langle S \rangle = |\{s_i = (p_i, b_i) \in S \mid b_i \neq \square \text{ and } (i = 1 \text{ or } b_{i-1} \neq b_i)\}|. \tag{8}$$

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 box that is different from the one in the step before. Note that this is not exactly the same as the size of the compact walk S^c. $|S^c|$ also counts compact steps that move no box.

Definition 11 (Swap-optimal sorting walk) A sorting walk S for a permutation π on a graph G that starts and ends at r 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

$$\begin{aligned} \langle S \rangle \text{ is swap-optimal } \Leftrightarrow \nexists \text{ sorting walk } S' \text{ s.t. } & (\langle S' \rangle < \langle S \rangle) \text{ or} \\ & (\langle S' \rangle = \langle S \rangle \text{ and } |S'| < |S|). \end{aligned} \tag{9}$$

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

4.1 Sorting on a Path

Theorem 8 (Efficient solution for BorderPathSwapSort) *The length and the compact represention of a swap-optimal sorting walk on a path P with $r = v_1$ and permutation π can be computed in time $\mathcal{O}(n)$ and has length*

$$d(\pi) + 2 \cdot \left(\max_{C \in \overline{\mathcal{C}}} l(C) - 1 \right). \tag{10}$$

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 $n - |\mathcal{C}| + |\overline{\mathcal{C}}|$, 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 1, the overlap components $\mathcal{D}$ 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 r 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 find P' by checking the leftmost box of all non-trivial cycles and can connect the $\mathcal{O}(n)$ many compact steps of the walk in constant time per step, the claim follows. □

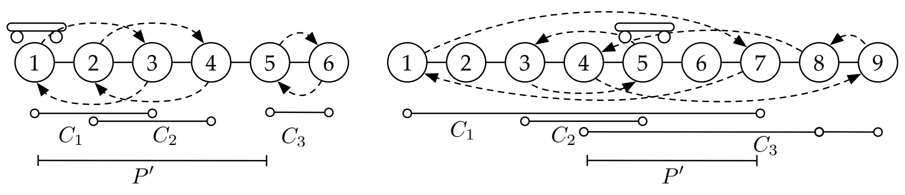

Fig. 11 (left) Example for the subgraph P' of non-essential steps for a swap-optimal sorting walk. C_3 is the cycle with its leftmost box furthest to the right. (right) Example where the robot starts at a non-border position. $P' = [4, 7]$ is the only connected subgraph of at most 3 edges that contains the starting vertex 5 and a vertex of every non-trivial cycle.

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

Theorem 9 (Efficient solution for PathSwapSort) *The length and the compact represention of a swap-optimal sorting walk on a path P with arbitrary r and permutation π can be computed in time $\mathcal{O}(n)$.*

Proof As before, we are looking for a connected r-containing subgraph P' that contains at least one vertex of every non-trivial cycle. We can represent such a subpath P' as an interval $[a, b]$ and can compute in linear time using a single sweep over the path P. We start with $[a, b] = [1, r]$. As long $[a, b]$ does not contain a vertex of every non-trivial cycle we increase b. While doing that we can update in constant time per step which cycles are present in the current interval. Once we have found a feasible interval we increase a and repeat as long as $a \leq r$ and $b \leq n$. Throughout this sweep we keep track of the shortest feasible interval and let that be P'. P' then allows the robot to reach and sort all the cycles with minimum number of swaps and minimum number of non-essential steps. Just as in Theorem 8, the robot walks once non-essentially along P' and we interleave this walk with all the non-essential cycles we encounter along the way. □

Figure 11 gives two examples of such minimal subgraphs that describe the non-essential steps in swap-optimal sorting walks.

4.2 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 closely related to the Class Steiner Tree problem, see Section 6.3 for details. We will show the hardness of our problem using a reduction from the problem of finding a satisfying assignment for a formula in 3-conjunctive normal form (3SAT).

Definition 12 (3SAT) Given a boolean formula $F = \{C_1, \ldots, C_M\}$ of M clauses. Each clause consists of *exactly* three literals $C_i = \{l_{i_1}, l_{i_2}, l_{i_3}\}$ 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 α: $\{x_1, \ldots, x_N\} \to \{\text{True}, \text{False}\}$ that satisfies at least one literal of every clause?

Theorem 10 ($\mathcal{NP}$-completeness for trees) *Finding a swap-optimal sorting walk on a tree T and a permutation π is $\mathcal{NP}$-complete.*

Proof We describe a reduction ϕ that maps every formula F in 3-conjunctive normal form to a rooted tree T and a permutation π describing an instance of the problem of finding a swap-optimal sorting walk on a tree.

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 three 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

$$V = \{r\} \cup \{v_{l,C} \mid l \in L \text{ and } C \in F\} \cup \{v_l \mid l \in L\} \tag{11}$$

$$\begin{aligned} E = &\{(r, v_{l,C_1} \mid l \in L\} \cup \\ &\{(v_{l,C_i}, v_{l,C_{i+1}}) \mid l \in L, i \in [M-1]\} \cup \\ &\{(v_{l,C_M}, v_l) \mid l \in L\} \end{aligned} \tag{12}$$

and a permutation $\pi \in S_n$ with

$$\pi(v_{l_1,C_i}) = v_{l_2,C_i} \text{ for } C_i = (l_1, l_2, l_3) \in F \tag{13}$$

$$\pi(v_{l_2,C_i}) = v_{l_3,C_i} \text{ for } C_i = (l_1, l_2, l_3) \in F \tag{14}$$

$$\pi(v_{l_3,C_i}) = v_{l_1,C_i} \text{ for } C_i = (l_1, l_2, l_3) \in F \tag{15}$$

$$\pi(v_l) = v_{\bar{l}} \text{ for } l \in L \tag{16}$$

$$\pi(v) = v \text{ otherwise} \tag{17}$$

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 12.

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 $(T, \pi) = \phi(F)$ contains a subtree T' that contains r, reaches all cycles of π and has size $|E(T')| = N \cdot (M+1)$.

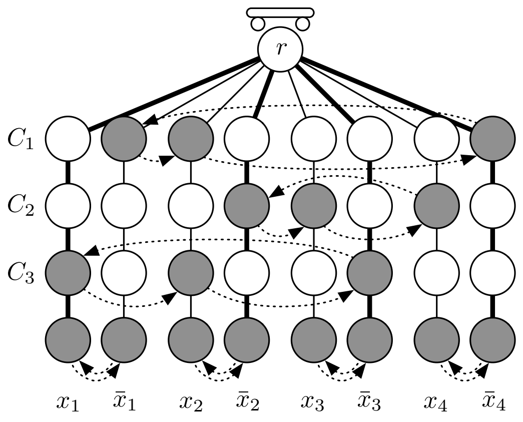

Fig. 12 Illustration of the spider T and permutation π generated for the formula $F = \{\{\bar{x}_1, x_2, \bar{x}_4\}, \{\bar{x}_2, x_3, x_4\}, \{x_1, x_2, \bar{x}_3\}\}$, $N = 4$, $M = 3$. The grey vertices mark the boxes that need to be moved for the three clause cycles and four variable cycles. The four legs printed in bold build the subtree T' that corresponds to the satisfying assignment $\alpha = (x_1, \bar{x}_2, \bar{x}_3, \bar{x}_4)$.

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

$\Leftarrow$: The variable cycles enforce that whenever we have $|E(T')| = N\cdot(M+1)$, 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 α with $\alpha(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 α. □

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

5 Implementation and Visualization

We provide implementations of all our algorithmic results on our website: http://dgraf.ch/treeswapsort. This includes the algorithms from Theorems 1, 3 and 6 to determine the length and compact representation of a shortest sorting walk for BorderPathSort, PathSort and for TreeSort. For trees, we compute the optimum branching using Edmond's algorithm [8] to find a cheapest cycle anchor tree and generate a corresponding sorting walk as in Theorem 6.

The implementation also includes the algorithms from Theorems 8 and 9 for swap-optimal sorting walks for BorderPathSwapSort and PathSwapSort.

A text-based visualization allows the animation of the resulting sorting walks and can be used to perform the sorting steps interactively. The webpage further contains a few stop motion video animations that illustrate the sorting process on small examples as well as a detailed usage tutorial of the implementation.

6 Related Work and Discussion

Efficient algorithms for sorting physical objects were studied in countless different models. We want to survey those that are most relevant to our results in this section. These models have been studied under various names, so finding and comparing the different settings and results is not an easy task. In Section 6.1, we list related work that uses solution techniques that are most similar to ours. Section 6.2 lists models that are still very close in spirit to our setting, but do not share the same solution techniques. Lastly, in Section 6.3, we refer to some work related to our two hardness proofs (Theorem 7 and 10).

6.1 Closely related Problems

We start by listing the problems with similar solution techniques in chronological order. We stumbled upon the cited articles in this section only after having written up and published our results in [17] and [18]. Thus, some of our results are independent rediscoveries of approaches found in the 1970s and 1980s. We will highlight the key similarities and differences and argue that the value of our results is that our algorithms are often significantly simpler to describe, analyze and implement.

6.1.1 Elevator Problem, Bus Problem (1972)

In his volume on Searching and Sorting [29] in section 5.4.8, D. E. Knuth discusses *one-tape sorting*. He cites the thesis of H. B. Demuth [7] to first show that a quadratic number of steps are needed if one can only operate on and move by a bounded part of the tape in every step. This corresponds to an asymptotic version of our lower bound in Lemma 1.

Knuth reformulates one-tape sorting as an *elevator problem*: “What is the fastest way to transport people between floors using a single elevator?”. He then refers to R. M. Karp’s article [26] which gives an optimum algorithm for it. Karp’s setting corresponds to our BorderPathSort problem as his elevator/robot also starts on the bottom floor. In his setting, the robot can carry c boxes at a time and there are exactly b boxes departing and arriving at every vertex, for two constants b and c. So his input is not a single permutation but b permutations. Karp further assumes that all edges of the path are covered by at least one cycle and then presents an algorithm. He does not give a formal proof but the full proof is presented in Knuth’s book [29]. While our algorithm in Theorem 2 in some sense only solves the ($b = 1$, $c = 1$)-case of Karp's setting, our algorithm sorts in a slightly different way and hence it only requires constant memory (Karp's algorithm stores a linear amount), can handle gaps between the connected components of cycle intervals and also works in an online fashion.

The generalization to where the elevator does *not* start and end at the bottom floor (our PathSort solved in Theorem 3) is not discussed. However, Knuth gives the generalization to tree graphs as an exercise, which he also attributes to Karp. The elevator is then called a *bus* that travels through the tree and the exercise is labeled as [M40] (suitable for a term project). According to Frederickson and Guan [13], neither Knuth nor Karp knew an efficient solution to this problem and Frederickson and Guan later showed that this problem (with a robot capacity greater than one) is $\mathcal{NP}$-complete [11,19].

6.1.2 Stacker Crane Problem (1978)

Frederickson, Hecht and Kim presented in [14] the *stacker-crane problem* which they attribute to D. J. Rosenkrantz. Phrased as a modified traveling salesman problem, this stacker-crane problem is roughly a generalization of our problem. The basics are the same, one unit-capacity robot travels along the edges of any given graph. They require that all the boxes are brought straight to their target without any intermediate drops (similar to our swap-minimal setting), they consider weighted graphs and they do not impose a limit on the number of boxes that are placed at each vertex (neither initially, finally, nor during the process). So instead of a permutation π, they get an arbitrary set of arcs on the graph that describe the desired shuffling of the boxes.

They show $\mathcal{NP}$-completeness using a reduction from TSP, with a construction that is different from the ones we presented for our more narrowly defined problems in our hardness proofs (Theorem 7 and 10), and they also give a $\frac{9}{5}$-approximation.

6.1.3 Robot Arm Travel (1987)

Atallah and Kosaraju studied the stacker-crane problem for a *robot arm* that can extend like a telescope and rotate around a fixed pivot point [1]. As it is $\mathcal{NP}$-hard for the combination of both motions, they look at the two cases where only one of the motions is allowed, so for extend-only (which corresponds to a path graph) and for rotate-only (which corresponds to a cycle graph). For both cases, they look at the situation with and without *preemption* (dropping a box before bringing it to its target), which is very similar to how we distinguish between *shortest* and *swap-optimal* sorting walks.

Their algorithm on cycles with preemption runs in linear time and can also be applied to paths. Hence their results are asymptotically equivalent to what we show in Theorem 3. However, their approach needs several nontrivial insights which is why we think that our linear solution to PathSort is significantly easier both to understand and implement. There are also two small differences between their problem and our Pathsort. The first one is that our vertices have a capacity constraint[2] and can only hold one box at any time, while their problem does not have such a constraint. This does not make our problem much more difficult as long as we just minimize the travel time. The only effect is that, unlike in this robot arm setting, our algorithm might need to move a box past its target position and bring it back later, e.g., when linking cycles as in Theorem 1. But this never causes us to make any extra steps and we could always just leave such boxes at their target position if there was no vertex capacity. The second difference is that the desired box-shuffling in the robot arm problem is not bound to be a permutation but allows for a general digraph on V as box-moving-requests. We will show in Section 6.1.5 how we can easily bring any such set of requests into our permutation form. So while having a permutation in our sorting setting is not really a restriction from a theoretical perspective, we think that it significantly simplifies the presentation of our results.

For the non-preemptive setting, where the boxes have to be brought straight to their target, the results do not match ours as closely as in the preemptive setting. For our swap-optimal sorting walks on paths (Theorem 9) we have to sort cycle by cycle of π and cannot sort only parts of the cycle and then move other boxes before finishing it due to the vertex capacity of one. Therefore, the way the cycles can be linked in this non-preemptive setting is different than in ours and so while we can do it in linear time in our setting, the non-preemptive robot arm problem needs time $\mathcal{O}(n\alpha(n))$ on paths and $\mathcal{O}(n \log n)$ on cycles as some minimum spanning trees have to be computed. Frederickson [10] improved the algorithm on cylces to also run in $\mathcal{O}(n\alpha(n))$ and formalized the tight relations to the minimum spanning tree problem.

6.1.4 Ensemble Motion Planning on Trees (1989)

Frederickson and Guan picked up Knuth's and Karp's exercise of a bus traveling through a tree-shaped network and study it for a bus of capacity one. This corresponds to the stacker crane and robot arm problem on a tree graph. They introduce yet another name for this problem: *ensemble motion planning*. Their main results [11] are split into two cases, the non-preemptive case [13] and the preemptive case [12].

In the preemptive case, they show $\mathcal{NP}$-hardness of the problem and give several approximation algorithms. The hardness construction in [12] reduces from the Steiner Tree Problem on bipartite graphs and is more involved than ours. Our hardness reduction for TreeSwapSort in Theorem 10 shows hardness for a more narrowly defined problem (on an unweighted graph with a single permutation) using a simple construction directly from 3SAT. The related Group Steiner Tree problem is discussed in Section 6.3

In the non-preemptive setting, Frederickson and Guan present an algorithm for the equivalent of Treesort that runs in quadratic time in Section 3.2

[2] The elevator problem by Karp also has this constraint.

of [12]. The basic ideas are exactly the same as in our quadratic algorithm in Theorem 6. After some initial balancing, a *directed bridging graph* is computed, which is the complete directed graph $\widetilde{G}$ of potential bridging edges with the same edge cost function that we use. Then, the directed minimum spanning tree of that graph (equivalent to our cycle anchor tree), is a witness on how to optimally link/bridge the cycles. In our setting of the box-shuffling being a permutation, the steps are again a bit shorter and easier to digest. In Section 4 of the paper however, Frederickson and Guan exploit some structure of these bridging graphs and are able to improve the runtime to $\mathcal{O}(n \log n)$. This is asymptotically strictly better than our quadratic result but comes at the prize of being very involved to understand and implement.

6.1.5 Reduction from the stacker crane setting to our permutation setting

An obvious difference between the stacker-crane-like settings (including robot arm travel and ensemble motion planning) and ours is that the stacker crane problem allows an aribtrary set of start and endpoints for the boxes while we are constraining our box-requests to form a permutation graph, which goes back to our original sorting motivation. We argue that this restriction to permutations does not simplify the asymptotical complexity of the task because we can easily transform a general set of box-requests into an equivalent one that forms a permutation – at least on paths and trees where there is always a unique path between any two vertices.

To that end, we first note that all our results easily extend to weighted graphs where each edge has an individual travel time. We then refer to Figure 13 to sketch how we use the *degree-balancing arguments* from [1] and [11] and a subsequent *vertex-splitting-by-degree* step to get an equivalent sorting problem on permutations.

6.2 Other Physical Sorting Algorithms

We now present further interesting models of sorting phyiscal items that are phrased very similar to our problem but significantly differ in their solutions. They all have in common that they play around with the constant time random read and write access that is crucial for many classical sorting algorithms. We can distinguish these physical sorting algorithms by the type of operation that they 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 [15], 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. [4] 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. [25] studied the extension to signed permutations, where each operation reverses a part of the permutation and also flips the sign of that part. One application is in

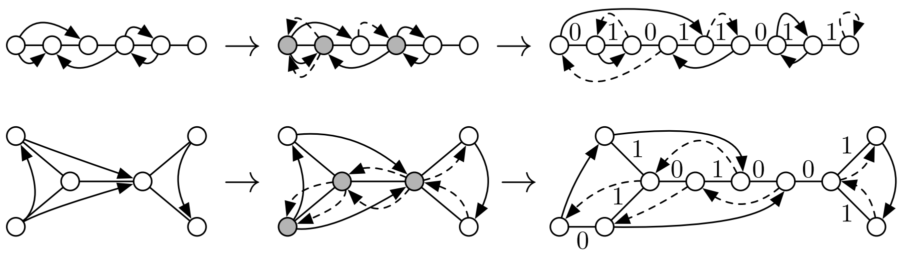

Fig. 13 Sketch of the reduction from an arbitrary set of boxes as in the stacker crane problem (left) to a balanced set that forms a permutation as in our setting (right). The solid lines mark the edges of the path (top row) and tree (bottom row). The solid arrows mark the requested moves. Our transformation now works in two steps. First, we add the unique minimal set of single-edge boxes such that along each edge of the graph the same number of boxes have to travel the two directions (middle). This we depict with the dashed arcs above and corresponds to the *degree-balancing* in [1] and [11]. In the second step, we split up all the vertices with more than one incoming/outgoing box (the ones printed in gray). We connect these copies with zero-cost edges and distribute the incoming and outgoing boxes of the former vertex to the new copies in such a way that every vertex has box-in/outdegree at most one afterwards. If there are any vertices left without boxes (in the example the vertex at the right end of the path), we add a single correctly-placed box on those vertices in order to end with an equivalent permutation-sorting problem.

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

Bender et al. [2] 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 squish 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* [9]. 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 (Section 2.2 in [28]) 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 [32] and then by many others. We refer to Bóna for a survey [3]. 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. [16] 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 [20]. 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 [5].
– 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 (Section 5.2.2 in [29]).
– Jerrum [24] 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.
– Yamanaka et al. [36] studied this problem on trees. By simulating cycle sort, they achieve a 2-approximation.
– Miltzow et al. [30] investigated the complexity for general graphs showing APX-hardness and giving a 4-approximation.

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 view 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. [6] 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 [21] for an overview. It is also a popular topic at competitive programming competitions (see Sections 3.4.3 and 3.4.4 of [18] for a discussion of two recent examples).

6.3 Hard Problems for Travelling on Graphs

Our hardness proof for finding shortest sorting walks on planar graphs (Theorem 7) is based on the proof by Itai et al. [23] that showed that the *Hamil-*

tonian path problem and *Hamiltonian circuit problem* on grid graphs are $\mathcal{NP}$-complete. It is interesting to note that the hardness requires the fact that these grid graphs can contain holes. Umans [34] showed that finding Hamiltonian cycles on grid graphs without holes is in $\mathcal{P}$.

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[3]. This problem was first introduced by Reich and Widmayer [31]. It is in $\mathcal{P}$ if the underlying graph is a path, which we even solved in linear time in Theorem 8. The $\mathcal{NP}$-completeness for the case where the graph is restricted to a tree was shown by Ihler et al. [22]. Like we do in Theorem 10, 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.

7 Conclusion

In this paper, we studied a sorting problem on graphs with the simple cost model of counting the number of edges traveled. We presented several asymptotically tight, linear-time algorithms that find optimum solutions if the graph is a path. Our extension to trees runs in quadratic time. For an extended cost function that minimizes the number of objects loaded, we gave linear time algorithms for paths and showed that it is already $\mathcal{NP}$-complete on trees.

An interesting open problem is whether sorting on trees can also be done in linear time. For many classes of graphs like ladder graphs, wheel graphs and hypercubes, the complexity is still open and it would be interesting to have better approximation algorithms for the hard instances.

Acknowledgements I want to thank Kateřina Böhmová and Peter Widmayer for many interesting and helpful discussions as well as the anonymous reviewers for their comments. I acknowledge the support of SNF project 200021L_156620.

References

1. Atallah, M.J., Kosaraju, S.R.: Efficient solutions to some transportation problems with applications to minimizing robot arm travel. SIAM Journal on Computing **17**(5), 849–869 (1988)
2. Bender, M.A., Farach-Colton, M., Mosteiro, M.A.: Insertion sort is $\mathcal{O}(n \log n)$. Theory of Computing Systems **39**(3), 391–397 (2006)

[3] Note that in our problem every vertex can be contained in at most one non-trivial cycle. In the Class Steiner Tree problem a vertex can be contained in multiple classes.

3. Bóna, M.: A survey of stack-sorting disciplines. Electronic Journal of Combinatorics **9**(2), 16 (2003)
4. Bulteau, L., Fertin, G., Rusu, I.: Pancake flipping is hard. Journal of Computer and System Sciences (2015)
5. Cayley, A.: Note on the theory of permutations. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science **34**(232), 527–529 (1849)
6. Demaine, E.D., Demaine, M.L., Fox-Epstein, E., Hoang, D.A., Ito, T., Ono, H., Otachi, Y., Uehara, R., Yamada, T.: Polynomial-time algorithm for sliding tokens on trees. In: Proceedings of the 25th International Symposium on Algorithms and Computation, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, pp. 389–400. Springer (2014)
7. Demuth, H.B.: Electronic data sorting. Ph.D. thesis, Department of Electrical Engineering, Stanford University (1956)
8. Edmonds, J.: Optimum branchings. Journal of Research of the National Bureau of Standards B **71**(4), 233–240 (1967)
9. Elizalde, S., Winkler, P.: Sorting by placement and shift. In: Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009, pp. 68–75. SIAM (2009)
10. Frederickson, G.N.: A note on the complexity of a simple transportation problem. SIAM Journal on Computing **22**(1), 57–61 (1993)
11. Frederickson, G.N., Guan, D.: Ensemble motion planning in trees. In: Proceedings of the 30th Annual Symposium on Foundations of Computer Science, FOCS 1989, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989, pp. 66–71. IEEE (1989)
12. Frederickson, G.N., Guan, D.: Preemptive ensemble motion planning on a tree. SIAM Journal on Computing **21**(6), 1130–1152 (1992)
13. Frederickson, G.N., Guan, D.: Nonpreemptive ensemble motion planning on a tree. Journal of Algorithms **15**(1), 29–60 (1993)
14. Frederickson, G.N., Hecht, M.S., Kim, C.E.: Approximation algorithms for some routing problems. In: Proceedings of the 17th Annual Symposium on Foundations of Computer Science, FOCS 1976, Houston, Texas, USA, 25-27 October 1976, pp. 216–227. IEEE (1976)
15. Gates, W.H., Papadimitriou, C.H.: Bounds for sorting by prefix reversal. Discrete Mathematics **27**(1), 47–57 (1979)
16. Gatto, M., Maue, J., Mihalák, M., Widmayer, P.: Shunting for dummies: An introductory algorithmic survey. In: R.K. Ahuja, R.H. Möhring, C.D. Zaroliagis (eds.) Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems, Lecture Notes in Computer Science, pp. 310–337. Springer (2009)
17. Graf, D.: How to sort by walking on a tree. In: Proceesings of the 23rd Annual European Symposium on Algorithms, ESA 2015, Patras, Greece, September 14-16, 2015, pp. 643–655. Springer (2015)
18. Graf, D.: Sorting and Scheduling Algorithms for Robotic Warehousing Systems. Master's thesis, ETH Zürich (2015)
19. Guan, D.J.: Routing a vehicle of capacity greater than one. Discrete Applied Mathematics **81**(1), 41–57 (1998)
20. Haddon, B.K.: Cycle-sort: a linear sorting method. The Computer Journal **33**(4), 365–367 (1990)
21. Hearn, R.A.: The complexity of sliding block puzzles and plank puzzles. In: B. Cipra, E.D. Demaine, M.L. Demaine, T. Rodgers (eds.) Tribute to a Mathemagician, pp. 173–183. Wellesley, Mass.: AK Peters (2005)
22. Ihler, E., Reich, G., Widmayer, P.: Class steiner trees and vlsi-design. Discrete Applied Mathematics **90**(1), 173–194 (1999)
23. Itai, A., Papadimitriou, C.H., Szwarcfiter, J.L.: Hamilton paths in grid graphs. SIAM Journal on Computing **11**(4), 676–686 (1982)
24. Jerrum, M.R.: The complexity of finding minimum-length generator sequences. Theoretical Computer Science **36**, 265–289 (1985)
25. Kaplan, H., Shamir, R., Tarjan, R.E.: A faster and simpler algorithm for sorting signed permutations by reversals. SIAM Journal on Computing **29**(3), 880–892 (2000)

26. Karp, R.: Two combinatorial problems associated with external sorting. In: Proceedings of the Courant Computer Science Symposium on Combinatorial Algorithms, pp. 17–29 (1972)
27. Karp, R.M.: A simple derivation of Edmonds' algorithm for optimum branchings. Networks **1**(3), 265–272 (1971)
28. Knuth, D.E.: The art of computer programming: fundamental algorithms, vol. 1. Addison-Wesley, Reading, MA (1973)
29. Knuth, D.E.: The art of computer programming: sorting and searching, vol. 3. Addison-Wesley, Reading, MA (1973)
30. Miltzow, T., Narins, L., Okamoto, Y., Rote, G., Thomas, A., Uno, T.: Approximation and Hardness of Token Swapping. In: Proceedings of the 24th Annual European Symposium on Algorithms, ESA 2016, Aarhus, Denmark, August 22 to 26, 2016, *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 57, pp. 66:1–66:15 (2016)
31. Reich, G., Widmayer, P.: Beyond steiner's problem: A vlsi oriented generalization. In: Proceedings of the 15th International Workshop on Graph-Theoretic Concepts in Computer Science, WG '89, Castle Rolduc, The Netherlands, June 14-16, 1989, pp. 196–210. Springer (1990)
32. Tarjan, R.: Sorting using networks of queues and stacks. Journal of the ACM (JACM) **19**(2), 341–346 (1972)
33. Tarjan, R.E.: Finding optimum branchings. Networks **7**(1), 25–35 (1977)
34. Umans, C.M.: An algorithm for finding hamiltonian cycles in grid graphs without holes. Bachelor's thesis, Williams College, Dept. of Mathematics (1996)
35. Wyttenbach, A.: Bike Loft. `http://bikeloft.ch` (2015)
36. Yamanaka, K., Demaine, E.D., Ito, T., Kawahara, J., Kiyomi, M., Okamoto, Y., Saitoh, T., Suzuki, A., Uchizawa, K., Uno, T.: Swapping labeled tokens on graphs. In: Proceedings of the 7th International Conference on Fun with Algorithms, FUN 2014, Lipari Island, Sicily, Italy, July 1-3, 2014, pp. 364–375. Springer (2014)
37. Zwick, U.: Directed minimum spanning trees (2013). URL `http://www.cs.tau.ac.il/~zwick/grad-algo-13/directed-mst.pdf`