How to Sort by Walking on a Tree

Daniel Graf

ETH Zürich, Department of Computer Science
grafdan@ethz.ch

Abstract. Consider a graph G with n vertices. On each vertex we place a box. These n vertices and n boxes are both numbered from 1 to n and initially shuffled according to a permutation $\pi \in S_n$. We introduce a sorting problem for a single robot: In every step, the robot can walk along an edge of G and can carry at most one box at a time. At a vertex, it may swap the box placed there with the box it is carrying. How many steps does the robot need to sort all the boxes?
We present an algorithm that produces a shortest possible sorting walk for such a robot if G is a tree. The algorithm runs in time $\mathcal{O}(n^2)$ and can be simplified further if G is a path. We show that for planar graphs the problem of finding a shortest possible sorting walk is $\mathcal{NP}$-complete.

Keywords: Physical Sorting, Shortest Sorting Walk, Warehouse Reorganization, Robot Scheduling, Permutation Properties

1 Introduction

Motivation. Nowadays, many large warehouses are operated by robots. Such automated storage and retrieval systems (abbreviated AS/RS) are used in industrial and retail warehouses, archives and libraries, as well as 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 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 Basel, for which bike boxes need to be rearranged according to the expected pickup times of the customers.

Problem Description. 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 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. 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 traveling 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 traveled 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 traveling along the edges. We are looking for the shortest sorting walk.

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 short walk that allows the robot to sort these 8 boxes. We will see an efficient algorithm that produces such a sorting walk and we will prove that this sorting walk has minimum length.

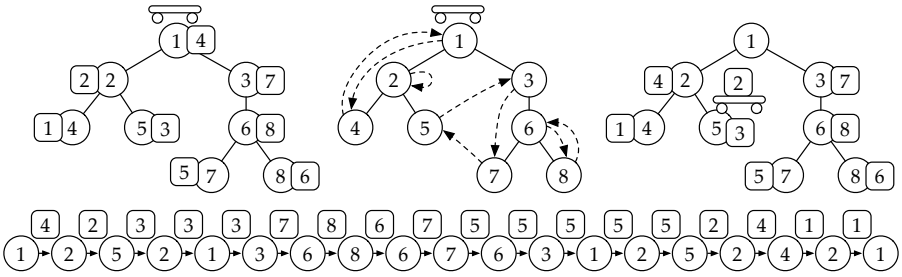

Fig. 1. (left) Initial state of the warehouse with storage locations as circles and boxes as squares. The box at vertex i is labeled with its target vertex $\pi(i)$. (center) The initial state with π drawn as dashed arcs towards their target vertex instead of numbered boxes. (right) This shows the state 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.

Organization. Section 2 introduces some terminology and shows first lower and upper bounds on the length of a shortest sorting walk on general graphs. We then look for shortest walks for certain classes of graphs. Section 3 shows a way of finding shortest sorting walks on path graphs where the robot starts

at one of the ends of the path. Our main result is given in Section 4, where we efficiently construct shortest sorting walks on arbitrary trees with arbitrary starting position. Finally, in Section 5 we show that it is $\mathcal{NP}$-complete to find a shortest sorting walk for planar graphs.

Related Work. Sorting algorithms for physical objects were studied in many different models before. Sorting streams of objects was studied for instance by Knuth [8], where we can use an additional stack to buffer objects for rearrangement. Similar problems were also studied in the context of sorting railway cars, for example by Büsing et al. [1]. Most similar to our solutions is an algorithm called *cycle sort* by Haddon [4] that minimizes the number of writes when sorting an array by looking at the cycles of its permutation. Yamanaka et al. [10] recently studied the process of sorting n tokens on a graph of n vertices using as few swaps of neighboring tokens as possible. For path graphs the number of swaps is minimized by bubble sort. They give a 2-approximation for tree graphs by simulating cycle sort. Compared to our setting, they do not require that successive actions are applied to nearby vertex positions. Sliding physical objects are also studied in the context of the hardness of many different puzzle games. We refer to Hearn [5] for an overview. An extensive overview of the research on storage yard operation can be found in [2].

2 Notation and General Bounds

Before we look at specific types of graphs, we introduce some notation and show some general lower and upper bounds for the length of a shortest sorting walk.

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, $b \in \{1, \ldots, n\} \cup \{\square\}$ is the number of the box that the robot is currently traveling with or $\square$ if it is traveling 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 $\{\sigma(i) \mid i \in \{1, \ldots, n\}\}$. In other words: Looking at σ and b together will at all times be a permutation of $\{1, \ldots, n\} \cup \{\square\}$. 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 $b \in \{1, \ldots, n\} \cup \{\square\}$.

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 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 onto 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 $b \in \{b_{t-1}, \sigma_{t-1}(v_{t-1})\}$, so if the robot moved along an edge of G and carried either the same box as before or the box that was located at the previous vertex. Thus after putting down a box at an empty location, the robot can either immediately pick it up again or continue

without carrying a box. A sequence of steps $S = (s_1, \ldots, s_l)$ 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.

We denote the set of cycles of the permutation π as $\mathcal{C} = \{C_1, \ldots, C_{|\mathcal{C}|}\}$, where each cycle C_i is an ordered list of vertices $C_i = (v_{i,1}, \ldots, v_{i,|C_i|})$ 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 neighbors in the cycle C and by $d(\pi)$ the sum of all such cycle distances for all cycles in π, i.e., $d(\pi) = \sum_{C \in \mathcal{C}} d(C) = \sum_{v \in V} d(v, \pi(v))$.

We distinguish two kinds of steps in a sorting walk: *essential* and *non-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 has length at least $d(\pi) = \sum_{b \in \{1,\ldots,n\}} d(b, \pi(b))$.*

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

The remaining challenge is to minimize the number of non-essential steps. In case that 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 box b

that is not placed at its correct position yet. These steps will be non-essential as the robot does not carry a box during these steps from r to b. Once it arrives at b, it sorts the cycle in which b is contained. In this way, it sorts cycle after cycle and finally returns to r. The number of non-essential steps in this process depends on the order in which the cycles are processed and which vertices get picked to start the cycles. The following lemma shows that a linear amount of non-essential steps will always suffice.

Lemma 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.

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

3 Sorting on Paths

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

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

Theorem 1 (Shortest sorting walk on paths). *The shortest sorting walk on a path P with permutation π can be constructed in time $\Theta(n^2)$ and has length*

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

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

We prove the claim by induction on the number of non-trivial cycles of 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 $|\overline{\mathcal{C}}| = 1$, the shortest sorting walk is also easy to find: We walk to the right until we encounter the leftmost box of this non-trivial cycle C, then we sort C and return to r. The number of steps is $d(\pi) + 2 \cdot (l(C) - 1)$ and is clearly optimal. Figure 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 (center) gives an example of such a case.

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

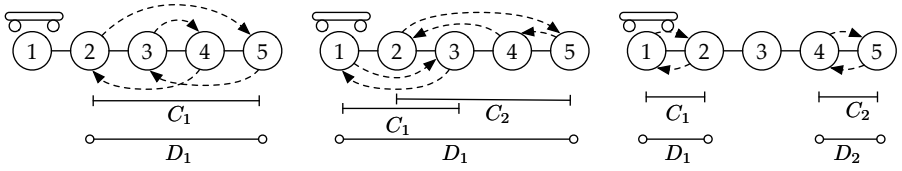

Fig. 2. (left) An example with a single non-trivial cycle. A shortest sorting walk S with $|S| = d(\pi) + 2 \cdot (l(C_1) - 1) = 8 + 2 \cdot (2 - 1) = 10$ is $((2, \square), (3, 5), (4, 5), (5, 5), (4, 3), (3, 3), (4, 4), (3, 2), (2, 2), (1, \square))$. (center) An example with two intersecting cycles.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

any of the previous cycles. We then know that any sorting walk uses all the edges between $\max_{j \in \{1,\dots,i\}} r(C_j)$ and $l(C_{i+1})$ for at least two non-essential steps. So if we interrupt S_i after the step where it visits $\max_{j \in \{1,\dots,i\}} r(C_j)$ 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 $\max_{j \in \{1,\dots,i\}} r(C_j)$ we get a minimum walk S_{i+1}. This case occurs whenever C_{i+1} lies in another connected component than all the previous cycles. So if C_i is the first cycle in some component D_j, we have $|S_{i+1}| = |S_i| + d(C_{i+1}) + 2 \cdot (l(D_j) - r(D_{j-1}))$, and so we get exactly the extra steps claimed in the theorem. □

Algorithmic Construction. The proof of Theorem 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.

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

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? It is not immediately clear whether its first step should go to the left or to the right then.

Instead of going into the details of this scenario, we now study the more general problem of arbitrary trees with arbitrary starting positions.

4 Sorting on Trees

We now want to study the problem of sorting boxes placed on an arbitrary tree. So let $T = (V, E)$ be the underlying tree, let $r \in V$ be the starting vertex and let T be rooted at r. 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.

4.1 Cycle Anchor Trees

Definition. 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 $\widetilde{v}_C$.

Edge Costs. 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 3 illustrates the definitions and gives an example of the transformation from a sorting walk to a weighted cycle anchor tree.

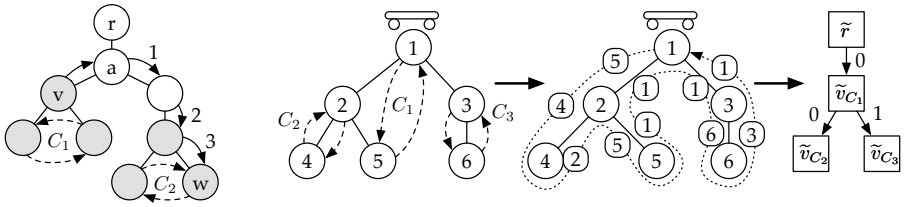

Fig. 3. (first figure on the left) The two pairs of dashed arrows symbolize boxes that need to be swapped. A shortest path from any $v \in T(C_1)$ to any $w \in V(C_2)$ is shown with continuous arrows, three of them being down-steps, so $c((\widetilde{v}_{C_1}, \widetilde{v}_{C_2})) = 3$. The anchor 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 labeled 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 $V(C_2)$.

Theorem 2 (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 (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.

4.2 Reconstructing a Sorting Walk

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

Theorem 3 (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 $\widetilde{T}$.

4.3 Finding a Cheapest Cycle Anchor Tree

Let S^* denote a shortest sorting walk for T and π. Using Theorem 3 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 3 to get a sorting walk $S_{\min}$ from $\widetilde{T}_{\min}$. As $c(\widetilde{T}_{\min}) \leq c(\widetilde{T}^*)$ we get

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

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 [3]. A great introduction to this algorithm, its correctness proof by Karp [7] and its efficient implementation by Dijkstra [9] can be found in the lecture notes of Zwick [11]. Combining these results with Theorem 3 will now allow us to find shortest sorting walks in polynomial time.

Theorem 4 (Efficient solution). *For any sorting problem on a tree T with permutation π, we can find a minimum sorting walk 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) $\left|\overline{\mathcal{C}}\right| + 1$ 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 3, we can find step s in constant time, if we keep track of the first step of S visiting each vertex of T. We build S as a linked list of steps in time linear to its length. Thus, as on the path (Theorem 1), we can construct $S_{\min}$ in time $\Theta(n + |S|)$ from $\widetilde{T}_{\min}$.

Combining these three steps gives an algorithm that runs in time $\mathcal{O}(n^2)$. □

5 Sorting on Other Graphs

Our algorithms for G being a path or a tree rely heavily on having unique paths between any pair of vertices. Therefore, these algorithms cannot be applied to graphs with cycles. In this section, we show that no efficient algorithm for general graphs can be found unless $\mathcal{P}$ equals $\mathcal{NP}$.

Theorem 5 ($\mathcal{NP}$-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 use a reduction from the problem of finding Hamiltonian circuits in grid graphs [6]. We replace each vertex of the grid by a pair of neighboring vertices with swapped boxes. A formal proof is omitted due to the page limitation.

6 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 an efficient algorithm that finds an optimum solution if the graph is a tree, and showed that the problem is hard on general graphs. All our results easily extend to weighted graphs where each edge has an individual travel time. It is open whether there are efficient algorithms for other special kinds of graphs or if there are good approximation algorithms for general graphs.

We provide an implementation of the algorithm for finding shortest sorting walks on paths and trees, as well as an interactive visualization on our website: `http://dgraf.ch/treesort`

Acknowledgments. 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. Büsing, C., Maue, J.: Robust algorithms for sorting railway cars. In: Algorithms–ESA 2010, pp. 350–361. Springer (2010)
2. Carlo, H.J., Vis, I.F., Roodbergen, K.J.: Storage yard operations in container terminals: Literature overview, trends, and research directions. European Journal of Operational Research 235(2), 412–430 (2014)
3. Edmonds, J.: Optimum branchings. Journal of Research of the National Bureau of Standards B 71(4), 233–240 (1967)
4. Haddon, B.K.: Cycle-sort: a linear sorting method. The Computer Journal 33(4), 365–367 (1990)
5. Hearn, R.A.: The complexity of sliding block puzzles and plank puzzles. Tribute to a Mathemagician pp. 173–183 (2005)
6. Itai, A., Papadimitriou, C.H., Szwarcfiter, J.L.: Hamilton paths in grid graphs. SIAM Journal on Computing 11(4), 676–686 (1982)
7. Karp, R.M.: A simple derivation of edmonds' algorithm for optimum branchings. Networks 1(3), 265–272 (1971)
8. Knuth, D.E.: The art of computer programming: sorting and searching, vol. 3. Pearson Education (1998)
9. Tarjan, R.E.: Finding optimum branchings. Networks 7(1), 25–35 (1977)
10. 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: Fun with Algorithms. pp. 364–375. Springer (2014)
11. Zwick, U.: Directed minimum spanning trees. `http://www.cs.tau.ac.il/~zwick/grad-algo-13/directed-mst.pdf` (April 2013)