On Sorting with a Network of Two Stacks

Matúš Mihalák
Department of Data Science and Knowledge Engineering, Maastricht University, The Netherlands
matus.mihalak@maastrichtuniversity.nl

Marc Pont
Department of Data Science and Knowledge Engineering, Maastricht University, The Netherlands
m.pont@student.maastrichtuniversity.nl

Abstract

Sorting with stacks is a collection of problems that deal with sorting a sequence of numbers by pushing and popping the numbers to and from a given set of stacks. Multiple concrete decision or optimization questions are formed by restricting the access to the stacks. The motivation comes, e.g., from shunting train wagons in shunting yards, shunting trams in depots, or in stacking cargo containers on cargo ships or storage yards in transshipment terminals.

We consider the problem of sorting a permutation of n integers $1, 2, \ldots, n$ using $k \geq 2$ stacks. In this problem, elements from the input sequence are pushed one-by-one (in the order of the elements in the sequence) to one of the k stacks. At any time, an element from a stack can be popped and pushed to another stack; such an operation is called a *shuffle*. Also, at any time, an element can be popped from a stack and placed to the output sequence. We can only place the elements to the output in the increasing order of their value such that at the end the output is the ordered sequence of the elements. The problem asks to minimize the number of shuffles in the process.

It is known that for $k \geq 4$, the problem is NP-hard, and that there is no approximation algorithm unless P=NP. For $k \geq 3$, it is known that at most $O(n \log n)$ shuffles are needed for any input sequence. For the case when $k = 2$, there exist input sequences that require $\Omega(n^{2-\varepsilon})$ shuffles, for any $\varepsilon > 0$. Nothing substantially more is known for the case of $k = 2$. In this paper, we study the following variant of the problem with $k = 2$ stacks: no shuffle and no placement to the output sequence can happen before every element is in one of the two stacks. We show that our problem can be seen as the MinUnCut problem by providing a polynomial-time reduction, and thus we show that there exists a randomized $O(\sqrt{\log n})$-approximation algorithm and a deterministic $O(\log n)$-approximation algorithm for our problem.

2012 ACM Subject Classification Mathematics of computing → Approximation algorithms; Theory of computation → Approximation algorithms analysis

Keywords and phrases Sorting, Stacks, Optimization, Algorithms, Reduction, MinUnCut

Digital Object Identifier 10.4230/OASIcs.ATMOS.2019.3

Acknowledgements The authors would like to thank Peter Widmayer and Thomas Erlebach for fruitful discussions on the topic.

1 Introduction

In computer science, a *stack* is a fundamental list-based data structure. Stacks allow item insertions and removals at one end of the list, known as the *last-in, first-out* (LIFO) principle. For stacks, insertions and removals are, respectively, called *push* and *pop* operations. A queue is another fundamental list-based data structure. Item insertions and removals happen at opposite ends in a queue, and this operation modus is known as the *first-in, first-out* (FIFO) principle.

Besides being a fundamental data structure in computer science, both stacks and queues model a wide range of applications in the real world, and in logistics and production planning in particular. An example is reordering of train wagons on a shunting yard, which can be

seen as sorting (reordering) with stacks (rail tracks). Typically, a set of n wagons, each having a unique identifier, need to be coupled to form a train that visits several customers along a fixed route. For every visit, the wagons determined for the respective customer need to be at the very end of the train, so that the wagons can be decoupled from the train and left at the customer. Thus, the n wagons need to be ordered according to a specific order (given by the order of the customers along the fixed route). The ordering of the wagons happens at a railway switching network, commonly known as a shunting yard, switch yard, marshalling yard, or classification yard. Typically, there are several so-called classification tracks, which can be accessed from only one end. The wagons can be pushed from a main track over a network of switches to any of the classification tracks. Subsequently, some of the wagons from a classification track can be pulled back to the main track, and the process can be repeated. Because the tracks can only be accessed from one end, the tracks can be modeled as stacks.

Inspired by such real-world scenarios, Knuth initiated the study of the problem of deciding what input sequences $\pi = (\pi_1, \ldots, \pi_n)$ – permutations of the integers $1, 2, \ldots, n$ – are *sortable* by a single stack or a queue [14]. In his setting, the elements of the input sequence are accessed sequentially (from π_1 to π_n), and are placed to the stack. At any time, either an element from the input sequence is pushed to the stack, or an element from a stack is popped, and then placed to the output sequence. The goal is to provide a sorted sequence at the output. In this setting, some sequences cannot be sorted, and the main focus of Knuth's work was to characterize input sequences that can be sorted. Knuth also touched upon the question of using more than one stack and especially the question of the number of stacks that are needed to sort any input sequence [15].

Subsequently, Even and Itai [6] and Tarjan [21] picked-up from Knuth and studied how to sort an input sequence using several stacks or queues. Tarjan introduced and studied a general model for sorting that contains several stacks and queues [21]. In the model, the stacks and queues are connected by an underlying directed graph that additionally contains the input node s and the output node t, such that s contains the input sequence and t contains the output sequence. Initially, s contains a permutation $\pi = (\pi_1, \ldots, \pi_n)$ of the first n integers, and t contains an empty sequence. Vertex s has no incoming edge, and vertex t has no outgoing edge. At any time, an element (an integer) can be taken from any vertex u (obeying the access rules of the underlying data structure – a stack or a queue), moved along any outgoing edge (u, v) of the graph, and stored to the data structure at v. In this context, the elements from s are obtained in the order $\pi_1, \pi_2, \ldots, \pi_n$. The goal is to decide whether the input sequence can be *sorted*, i.e., whether there is a sequence of moves of the elements along the edges such that the elements arrive at t in the order $1, 2, \ldots, n$. This is not always possible, and the posed question initiated the study of permutation classes, see, e. g. [4]. Much of the work along these questions have focused on structural results, characterizing and counting permutations that are sortable by a given acyclic network of stacks and queues.

However, as Tarjan observes, whenever the underlying graph contains a cycle (and the cycle is reachable from s, and vertex t is reachable from the cycle), any input permutation can be sorted. Thus, for the question "what permutations are sortable?", such underlying graphs are trivial and thus not considered in the research along the question.

Optimization Variant

Much later, the observation of Tarjan was picked up, and two related *optimization* questions were asked for underlying graphs containing cycles [7, 17]: "How many moves do we need to sort a given input sequence?" and "What is the complexity of sorting with a minimum

number of moves?". These questions have mainly been studied in the setting where the underlying graph is a complete graph, with the exception of the vertices s and t, which only have outgoing and incoming edges, respectively.

Felsner and Pergel show that for $k \geq 3$ stacks, any input sequence can be sorted with $O(n \log_{k-1} n)$ many moves [7]. This is asymptotically tight, as the worst-case inputs require at least $\frac{n}{2} \log_k n - \Theta(1)$ moves [7]. This is in strong contrast to the case when $k = 2$, as for this case, Felsner and Pergel show that for any $\varepsilon > 0$ there exists an input sequence which needs $\Omega(n^{2-\varepsilon})$ many moves. Interestingly, their example is also valid if we restrict the movements such that no move to t can be made before all elements from s are in one of the stacks. This restriction is called a *midnight constraint*, as it mimics the situation when trams need to be parked in a depot (tracks in the depot are the stacs in our model) at the end of the day, and can leave the depot only in the morning.

König and Lübbecke study the optimization version of the sorting problem [17]. Naturally, in any solution to the sorting problem, every item needs to move along an arc from s exactly once, and along an arc to t exactly once, so König and Lübbecke study the following optimization problem: sort the input sequence by a minimum number of *shuffles*, where a *shuffle* is a move along an arc that is not incident to s and also not incident to t. For this problem, a ρ-approximation algorithm, for $\rho > 1$, is a polynomial-time algorithm that sorts any input sequence π using at most $\rho \cdot \mathrm{OPT}(\pi)$ many shuffles, where $\mathrm{OPT}(\pi)$ is the minimum number of shuffles needed to sort the sequence π. König and Lübbecke show that it is NP-hard to approximate the minimum number of shuffles within $O(n^{1-\varepsilon})$, for any non-trivial,[1] even constant, $k \geq 4$. Their work is based on the work of Evan and Itai [6], and the relation of the problem of deciding whether a proper k-coloring of a given circle graph exists to the problem of deciding whether the input permutation can be sorted with k-stacks without shuffles. The former problem has been shown NP-complete for $k \geq 4$ by Unger [23].

We note that since for $k \geq 4$ deciding whether one can sort with zero shuffles is NP-hard, it follows that, for $k \geq 4$, there is no approximation algorithm for the problem of minimizing the number of shuffles, unless P=NP.

Optimization with Midnight Constraint

We further note that the optimization problem with $k \geq 4$ stacks remains NP-hard also for the midnight-constraint, since the midnight constraint can be imposed by appending the integer 0 to the input permutation π of integers $1, 2, \ldots, n$, and considering the new problem on the resulting permutation π' of integers $0, 1, \ldots, n$ without midnight constraint. Since the smallest integer 0 comes at the end of the input sequence π', no move to t can happen before 0 is moved from s, and thus before all elements of π' are in one of the k stacks. However, the problem of deciding whether one can sort with zero shuffles becomes polynomial-time solvable for any k in the case of midnight constraint, since this problem is equivalent to the k-coloring of permutation graphs [6], which can be solved in polynomial time. Hence, the inapproximability result for the general case (i.e., no midnight constraint) and $k \geq 4$ does not carry over to the case with the midnight constraint.

Since for $k \geq 3$, at most $O(n \log_k n)$ shuffles are needed [7], it follows that there exists a $O(n \log_k n)$-approximation algorithm for the minimization problem with midnight constraint and $k \geq 3$. For $k = 2$ and midnight constraint, no non-trivial approximation algorithm is known. (It is easy to see that for $k = 2$, no more than $O(n^2)$ shuffles is needed, which gives a trivial $O(n^2)$-approximation algorithm.)

[1] E.g., $k = n$ is trivial, as every item can be placed on a unique stack, and thus no shuffle is required.

The complexity of the minimization problem for $k = 2$ and $k = 3$, however, is an open problem also for the optimization problem with the midnight constraint. Both Felsner and Pergel [7] and König and Lübbecke [17] asked, as an open problem, whether the sorting problem with midnight constraint is computationally easier. (There is a light evidence that the problem with the midnight constraint might be easier: recall that deciding whether one can sort with zero shuffles is NP-hard for $k \geq 4$ in general, but becomes solvable in polynomial-time for the midnight constraint.)

Our Contribution

In this paper, we address the open problems for the case with $k = 2$ stacks and make a progress for a special case of the midnight constraint. We study the minimization problem with the *strong midnight-constraint*, which we define as the midnight constraint, with an additional constraint where no shuffles are allowed before all items are moved away from s.

We show that the problem of minimizing the number of shuffles with the strong midnight-constraint can be seen (by a certain polynomial-time reduction) as the MinUnCut problem on certain graphs, and thus inherits the same approximation-algorithm guarantees. In particular, as a corollary, we show that there exists a randomized $O(\sqrt{\log n})$-approximation algorithm, and a deterministic $O(\log n)$-approximation algorithm for the minimization problem with $k = 2$ stacks and with the strong midnight-constraint.

The result thus substantially improves upon the trivial $O(n^2)$-approximation algorithm for $k = 2$. We note that the $O(n \log n)$-approximation algorithm for $k \geq 3$ and for the midnight constraint carries over also to the case with the strong midnight-constraint. Our result thus gives, for the variant with the strong midnight-constraint, a better approximation algorithm for the case $k = 2$ than for the case $k \geq 3$.

Additionally, we show that in the setting with the strong midnight-constraint and $k = 2$ there exists an input sequence for which every algorithms needs $\Omega(n^2)$ shuffles. This improves upon the lower bound of $\Omega(n^{2-\varepsilon})$ shuffles that holds for any $\varepsilon > 0$, and was shown by Felsner and Pergl for the (normal) midnight constraint and $k = 2$, and that also applies to the setting with the strong midnight constraint [7].

We note that relating the number of shuffles to the number of edges that need to be deleted such that some auxiliary graph becomes bipartite (as is the case of the MinUnCut problem) is not new. Motivated by the result of Evan and Itai [6] which states that one can sort with zero shuffles using k stacks if and only if there exists a k-coloring of the underlying circle graph, König and Lübbecke consider the question whether the number of shuffles is equal to the number of monochromatic edges in a k-coloring of the underlying circle graph, and provided an example demonstrating that this is not case [17]. In our work, we use a different auxiliary graph than the circle graph.

1.1 Further Related Work to Stack Sorting

Albeit Tarjan defined the problem of sorting with a network of stacks and queues for any underlying graph [21], mainly the following two graph classes have been studied: (i) a directed path from s to t; this case is also referred to as *sorting with stacks/queues in series*, and (ii) graphs where every node other than s and t is connected only to s (in the direction from s) and to t (in the direction to t); this setting is also known as *sorting with stacks/queues in parallel*. Observe that the question "can we sort the input sequence with zero shuffles" where the underlying graph induced by the stacks is a complete graph is equivalent to the setting with stacks in parallel. Even and Itai study the setting with stacks in parallel and with

queues in parallel [6]. Besides the general setting, Even and Itai also study the variant where no movement to t can precede any movement from s. In their setting, this is equivalent to both the strong midnight-constraint and the midnight constraint.

As mentioned before, much of the previous work focused on structural results concerning permutations that can be sorted by stacks/queues in series/parallel. Bóna surveys much of the progress made till about 2003 [4]. Among the newer research that is of algorithmic flavor is the study by Smith that compares two greedy algorithms for sorting with stacks in series [20], the paper by Pierrot and Rossin that shows that deciding whether a given permutation is sortable by two stacks in series in polynomial-time solvable [18], and the paper by Biedl et al. [2] which studies the question of how the number of stacks influences the number of shuffles needed to sort any input sequence.

1.2 Related Work in the Application Domain

Over the years, optimization theory literature has mentioned numerous problems related to sorting with stacks. A few examples are: assigning trains, trams or buses to positions in a depot [3, 11, 9]; storing integrated steel slabs in order of processing [16]; sorting car bodies for paint processing [12]; and storage yard operations in container terminals [5, 22]. Each of these problems require items to be placed on stacks such that they can be retrieved in a desired order with minimum effort or shuffles. In practice it is not uncommon that additional constraints exist such as stack height or item placement.

2 The Setting and Preliminaries

We study the RestrictedBi-StackSorting problem, which is defined as follows, and illustrated in Figure 1. We are given a directed graph $G = (V, E)$ on four vertices and an *input sequence* π, which is a permutation of the numbers $1, ..., n$. Each number in the permutation is also called an *element* of π. Vertex set V represents the four possible locations for items $\pi_1, ..., \pi_n$, which are a *source* vertex s, two stack vertices v_1 and v_2, and a *target* vertex t. The two stacks v_1 and v_2 exhibit the LIFO behavior. Edge set E consists of directed edges (i, j) to represent actions that move the first available item from vertex i to vertex j. These actions are *push*, represented by edge (s, v_1) and by edge (s, v_2), *shuffle*, represented by edge (v_1, v_2) and by edge (v_2, v_1), and *pop*, represented by edge (v_1, t) and edge (v_2, t). There are no other edges in E. Items $(\pi_1, ..., \pi_n)$ arrive sequentially from s and may only traverse edges in E along the direction of the edge. The problem asks to move the elements in π from s to t such that at any time, only one item traverses along an edge, the items at v_1 and v_2 respect the LIFO behavior, the items leave s in the given order by π, and the items arrive at t in increasing order of value. Furthermore, we require that while there are items in s, no shuffle *and* no pop appears. We call this the *strong midnight-constraint*. The goal of RestrictedBi-StackSorting is to minimize the number of shuffles along the process. The number of made shuffles by an algorithm is called the *shuffle count* (of the algorithm).

2.1 Useful State Representation

At any time of the sorting procedure, the state is fully determined by the remaining elements in s, and by the content of the two stacks in vertices v_1 and v_2. We now describe an alternative description of a state, which is the crucial element in showing our main result.

We view any sorting procedure as an iterative process, which at any time step $k = 1, 2, \ldots$, moves an element along an edge of G. A state S_k at time step k describes the situation

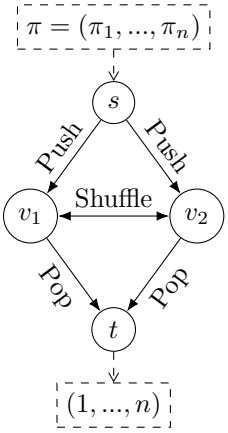

Figure 1 Illustration of the underlying graph for sorting with two (fully connected) stacks.

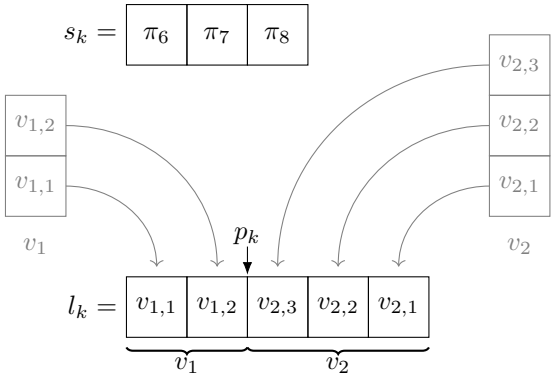

Figure 2 Creating state representation $S_k = (s_k, l_k, p_k)$.

after the k-th move. State S_k is a tuple (s_k, l_k, p_k), where s_k is the remaining input at vertex s, l_k is a list (array) that is the union of the two stacks, and p_k is a pointer to the list l_k, called *shuffler indexer*, which specifies the tops of the two stacks in the list l_k. To create list l_k, we concatenate the stacks $v_1 = (v_{1,1}, ..., v_{1,|v_1|})$ and $v_2 = (v_{2,1}, ..., v_{2,|v_2|})$ as list $l_k = (v_{1,1}, ..., v_{1,|v_1|}, v_{2,|v_2|}, ..., v_{2,1})$, such that $v_{i,1}$ is the bottom-most item of stack v_i and $v_{i,|v_i|}$ is the top-most item of stack v_i. Alternatively, $l_k = v_1 v_2^R$, where v_2^R is the reverse of v_2. Finally, we set shuffler indexer p_k at location $p_k = |v_1|$, representing the top of both of our stacks. Figure 2 depicts our alternative state representation for RestrictedBi-StackSorting. Figure 3 provides an example of how our state representation transitions during a push, a pop, and a shuffle action. State S_0 describes the situation before any move is made, and thus $S_0 = (\pi, (), 0)$, where () denotes the empty list.

2.2 Relation of State with Shuffle Count

Recall that RestrictedBi-StackSorting requires that all elements are first pushed from s before any shuffle or pop operation can happen. Observe that after all n elements are pushed from s, the shuffle count is determined: The first element that needs to be popped is 1. For this, all elements that are above element 1 need to be shuffled to the other stack.

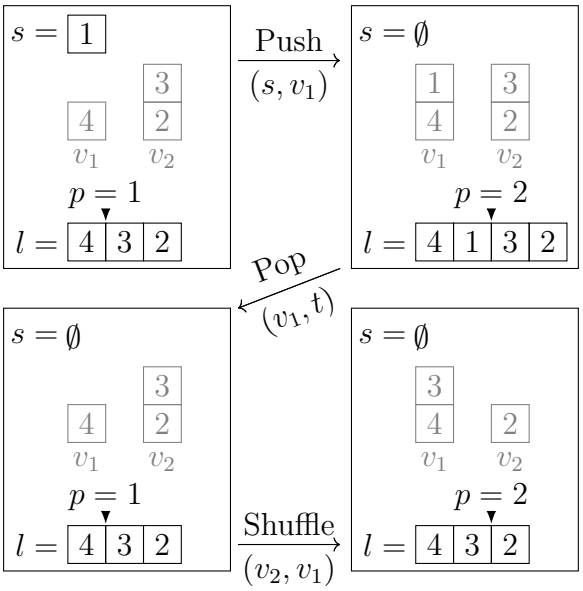

Figure 3 State transitions when applying a push, pop, and shuffle action.

After that, 1 can be popped, and the same procedure continues with element 2: all elements that are above 2 in the same stack need to be shuffled, and then element 2 can be popped. After that, the same procedure continues with elements 3, 4, ..., n.

Using our alternative state representation, we can determine the shuffle count from the list l_n of state S_n, i.e., the state after all n elements have been pushed from s, as follows: We place an auxiliary value 0 at the position p_n in list l_n. Then, observe that to pop element 1, we need to shuffle all elements that appear in l_n between element 0 and element 1. Similarly, after we have popped element $i \geq 1$, we need to shuffle all elements that are the elements that lie in l_n between i and $i+1$ *and that are larger than* $i+1$. This follows because: (i) when element i is popped to t, it is on top of a stack, and to pop the next element $i+1$, elements that lie above $i+1$ need to be shuffled, and these lie between i and $i+1$ in l_n; (ii) any element in l_n that is smaller than $i+1$ has been popped to t and thus does not need to be shuffled.

Let $sc(i, i+1)$ denote the number of elements that lie in l_n between elements i and $i+1$, and that are larger than $i+1$. We have thus showed the following.

▶ **Lemma 1.** *The shuffle count is equal to* $\sum_{i=0}^{n-1} sc(i, i+1)$.

2.3 Worst-Case Number of Shuffles

We now show that the worst-case number of shuffles for RestrictedBi-StackSorting is $\Omega(n^2)$. For the strong midnight-constraint, this further strengthens the lower bound of $\Omega(n^{2-\varepsilon})$, for any constant $\varepsilon > 0$, of Felsner and Pergel [7], which also holds for the (normal) midnight constraint.

▶ **Theorem 2.** *There exists an input sequence* π *for* RestrictedBi-StackSorting *for which every algorithm has a shuffle count of* $\Omega(n^2)$.

Proof. Consider the input sequence $\pi^\star = (2, 4, 6, ..., n, n-1, n-3, ..., 5, 3, 1)$ for any even n. Notice that $\pi^\star$ consists of two sub-sequences: $\pi^\star_{1,\frac{n}{2}}$ of all even values and $\pi^\star_{\frac{n}{2}+1,n}$ of all odd values.

After $\pi^\star_{1,\frac{n}{2}}$ is pushed to the stacks, one of the stacks will contain the majority of the items of $\pi^\star_{1,\frac{n}{2}}$, and thus at least $\frac{n}{4}$ many items. Without loss of generality, assume that stack v_1 contains at least $\frac{n}{4}$ items from $\pi^\star_{1,\frac{n}{2}}$.

Observe that every item in v_1 needs to be popped in the order from bottom to top. Also, observe that for every even value e that we pop from v_1, we then need to pop the odd value $e+1$ from $\pi^\star_{\frac{n}{2}+1,n}$. Now, regardless of how the items in $\pi^\star_{\frac{n}{2}+1,n}$ have been pushed to the stacks, for every bottom value e of v_1, we need to shuffle at least the remaining even values on stack v_1 in order to pop value $e+1$. Given that we have at least $\frac{n}{4}$ even values on stack v_1, we see that the shuffle count is at least $\sum_{i=1}^{\frac{n}{4}}(\frac{n}{4}-i) = \frac{n^2-4n}{32} = \Omega(n^2)$. Thus, any algorithm for RestrictedBi-StackSorting uses at least $\Omega(n^2)$ shuffles. ◀

3 Relation to MinUnCut

We will now show that RestrictedBi-StackSorting can be seen as MinUnCut. The problem MinUnCut is given by an undirected graph $H = (V_H, E_H)$, and asks for a partition (S,T) of the vertices in V_H such that the number of edges with endpoints from the same part is minimized. The problem is the complement of the more famous MaxCut problem, that asks for (S,T) such that the number of edges in the cut, i.e., edges with one endpoint in S and one endpoint in T, is maximized. It can be easily seen that an optimum solution to MaxCut is also an optimum solution to MinUnCut. MaxCut was among the first computational problems shown to be NP-complete [13].

The two problems differ with respect to approximability. While for MaxCut there exists a ρ-approximation algorithm where ρ is roughly 0.878 [10], the best approximation algorithm for MinUnCut is a randomized $O(\sqrt{\log n})$-approximation algorithm by Agarwal et al. [1] and a deterministic $O(\log n)$-approximation algorithm by Garg et al. [8].

Our main technical result is stated in the following theorem.

▶ **Theorem 3.** *We can reduce any input instance of* RestrictedBi-StackSorting *to an instance of* MinUnCut *in polynomial time, such that the number of edges with endpoints in the same part of a solution (S,T) in the created instance of* MinUnCut *is at most the shuffle count of a corresponding solution to the instance of* RestrictedBi-StackSorting*, and the corresponding solution to* RestrictedBi-StackSorting *can be computed in polynomial time.*

Proof. We create graph $H = (V_H, E_H)$ for MinUnCut as follows. We put all vertices $1, 2 \ldots, n$ to V_H, corresponding to the elements (integers) in π. Placing vertex i to S will correspond to pushing the element i to stack v_1, and placing vertex i to T will correspond to pushing the element i to stack v_2.

Recall that before any shuffle, we need to push all elements from s to the stacks. This will lead to the state $S_n = (s_n = \emptyset, l_n, p_n)$. Referring to Lemma 1, and especially to the count $sc(i, i+1)$ for $i = 0, 1, \ldots, n-1$, we want to create edges between vertices in H that express the shuffles in $sc(i, i+1)$.

Recall that $sc(i, i+1)$ is the number of elements between i and $i+1$ in list l_n that are larger than $i+1$. Let x be any element that is larger than $i+1$. Let us investigate the positions of the elements x, i, and $i+1$ as they appear in the permutation π, and how these positions influence $sc(i, i+1)$. **First**, observe that whenever x appears before i and $i+1$ in π, then for any state l_n, x will never be between i and $i+1$, and thus x will never be counted in $sc(i, i+1)$. In this case, we do not create any edge between x and i nor any edge between x and $i+1$ in graph H (as there should be no cost in the MinUnCut problem). **Second**, assume now that x appears between elements i and $i+1$ in the input permutation

π. In this case, either element i appears before x, or element $i+1$ appears before x in π. Among i and $i+1$, let z be the element that appears before x in π. Observe now that x appears between i and $i+1$ in l_n (and thus contributes one to $sc(i, i+1)$), if and only if x is placed on the same stack as z. Thus, in this case, we create an edge between z and x to account for the cost of one in every solution that places z and x to the same part in a partition (S, T). **Third**, assume now that x appears after both elements i and $i+1$ in the input sequence π. Observe now that x appears between i and $i+1$ in the list l_n if and only if i and $i+1$ are placed on two different stacks. For this reason, in this case, we create an auxiliary vertex $v(x, i, i+1)$ and connect it to vertex i and to vertex $i+1$ (and to no other vertex). Observe now that whenever vertices i and $i+1$ are placed such that one is in part S and one is in part T, it does not matter to which part we place vertex $v(x, i, i+1)$: any placement will incur cost exactly one, which exactly corresponds for x appearing between i and $i+1$ in the list l_n (and thus corresponds for one count in $sc(i, i+1)$). Also, observe that whenever i and $i+1$ are placed to the same stack, there is no shuffle of x encountered in the cost $sc(i, i+1)$, and this can be reflected by placing the vertex $v(x, i, i+1)$ to the opposite part in which vertices i and $i+1$ are, which leads to zero count for MinUnCut.

Now, to finish the proof, consider an instance of RestrictedBi-StackSorting and the corresponding graph H created by the reduction above. Consider a solution (S, T) to MinUnCut given by instance H. If in the solution there are vertices i, $i+1$, and $v(x, i, i+1)$ placed in the same part, we modify the solution by moving $v(x, i, i+1)$ to the other part. This decreases the cost of the solution for MinUnCut by two, and only affects the two edges incident to $v(x, i, i+1)$. We repeat this process until no such three vertices i, $i+1$, and $v(x, i, i+1)$ can be found. This takes at most $O(n^2)$ many steps (one for every value i and every value $x > i$), and results in a new solution (S', T') to MinUnCut of cost not larger than the original solution (S, T) to MinUnCut. We can create a solution to the RestrictedBi-StackSorting as follows: push element i to stack v_1 if and only if i is in S' (and otherwise, when i is in T', push it to stack v_2). Observe now that every edge in the new solution (S', T') that has both endpoints in the same part corresponds to exactly one shuffle in the corresponding solution to RestrictedBi-StackSorting, and thus the number of shuffles in the created solution for RestrictedBi-StackSorting is the cost of the solution (S', T') for MinUnCut. ◀

The construction from the proof is illustrated in Figure 4. The existence of approximation algorithms for RestrictedBi-StackSorting with the claimed approximation ratios now comes as a direct corollary.

▶ **Corollary 4.** *There exists a randomized $O(\sqrt{\log n})$-approximation algorithm for the problem* RestrictedBi-StackSorting. *There exists a deterministic $O(\log n)$-approximation algorithm for the problem* RestrictedBi-StackSorting.

3.1 Beyond Two Stacks

It is a natural question to try to adapt our approach also to the cases $k \geq 3$. However, this is not possible (in a direct way). Our approach crucially depends on the alternative state representation introduced in Section 2.2 and on its relation to the shuffle count as expressed by Lemma 1. In our alternative state representation, we heavily used the fact that two stacks after step k can be merged into a linear list l_k. For three or more stacks, it is not clear how such a list could be created. Clearly, it is one of the interesting open problems to provide better approximation algorithms for the case $k = 3$.

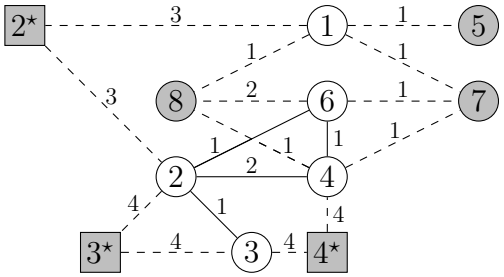

Figure 4 Graph H for the input $\pi = (2, 4, 3, 6, 1, 8, 7, 5)$. The circles denote the vertices of the permutation, the squares denote the auxiliary vertices $v(*, i, i + 1)$, and the edge labels denote the multiplicity of the edges between the vertices. The white vertices are in part S and grey vertices are in part T. Such a partition induces an uncut of size 5, and thus a solution to the sorting problem with 5 shuffles. This is optimum for this permutation.

3.2 Shuffling Before Midnight

Clearly, for $k = 2$, one can use our alternative state representation not also for the strong midnight-constraint, but also for the (normal) midnight constraint, where one can shuffle even if not all elements are pushed from s to the two stacks. For the strong midnight-constraint, the first n moves are only pushes from s, which results in some state S_n, which then uniquely induces what the algorithm does and the shuffle count the algorithm requires. In some sense, for the strong midnight-constraint, the only decisions to be made are to which stack shall we push element i, $i = \pi_1, \pi_2, \ldots, \pi_n$. These decisions can be reflected by the auxiliary graph H that we create in the proof of Theorem 3. However, if we allow to shuffle before all n elements are pushed from s to the stacks, it is not clear how to create an auxiliary graph H which would reflect (via the MinUnCut problem) the shuffle count for this case. In some sense, allowing shuffles before all elements from s are pushed would move the position of the shuffler index while the push operations are made, possibly changing the shuffle count impact of past and future push operations. To reflect this in H, it seems that we would need to update, add, and remove labeled edges in graph H. It is not clear that we can create H that would reflect such a dynamic behavior. A more extensive discussion on this topic can be found in the Master thesis of Pont [19]. We leave it a prominent open problem to settle the complexity of the sorting problem with $k = 2$ stacks and midnight constraint.

4 Conclusions

Motivated by the open problem of addressing the complexity of minimizing the shuffles when sorting with two stacks with or without the midnight constraint, we introduced the *restricted midnight-constraint* and studied the resulting RestrictedBi-Stack Sorting problem. We showed that our problem is closely related to the MinUnCut problem. This shows that the problem admits non-trivial approximation algorithms, which is in strong contrast to known approximability and inapproximability results to the other variants of the optimization problems that have been considered so far.

There are several open problems left by our paper. One of the most important ones is to settle whether the problem is NP-hard. Beyond the topic of this paper, i.e., sorting with two stacks with the strong midnight-constraint, one of the most interesting open problems is to investigate whether non-trivial approximation algorithms exist for the general, unrestricted case of $k = 2$ stacks.

References

1 Amit Agarwal, Moses Charikar, Konstantin Makarychev, and Yury Makarychev. $\mathcal{O}(\sqrt{\log n})$ Approximation Algorithms for min UnCut, min 2CNF Deletion, and Directed Cut Problems. In *Proc. Annual ACM Symposium on Theory of Computing (STOC)*, pages 573–581. ACM, 2005.

2 Therese Biedl, Alexander Golynski, Angèle M. Hamel, Alejandro López-Ortiz, and J. Ian Munro. Sorting with networks of data structures. *Discrete Applied Mathematics*, 158(15):1579–1586, 2010. doi:10.1016/j.dam.2010.06.007.

3 Ulrich Blasum, Michael R Bussieck, Winfried Hochstättler, Christoph Moll, Hans-Helmut Scheel, and Thomas Winter. Scheduling Trams in the Morning. *Mathematical Methods of Operations Research*, 49(1):137–148, 1999.

4 Miklós Bóna. A survey of stack-sorting disciplines. *The Electronic Journal of Combinatorics*, 9(2):A1.1–A1.16, 2003. URL: http://eudml.org/doc/123106.

5 Héctor J Carlo, Iris FA Vis, and Kees Jan Roodbergen. Storage Yard Operations in Container Terminals: Literature Overview, Trends, and Research Directions. *European journal of operational research*, 235(2):412–430, 2014.

6 S. Even and A. Itai. Queues, stacks, and graphs. In *Proc. International Symposium on the Theory of Machines and Computations*, pages 71–86. Academic Press, 1971.

7 Stefan Felsner and Martin Pergel. The Complexity of Sorting With Networks of Stacks and Queues. In *Proc. European Symposium on Algorithms (ESA)*, pages 417–429. Springer, 2008.

8 Naveen Garg, Vijay V Vazirani, and Mihalis Yannakakis. Approximate max-flow min-(multi) cut theorems and their applications. *SIAM Journal on Computing*, 25(2):235–251, 1996.

9 Brady Gilg, Torsten Klug, Rosemarie Martienssen, Joseph Paat, Thomas Schlechte, Christof Schulz, Senan Seymen, and Alexander Tesch. Conflict-Free Railway Track Assignment at Depots. *Journal of rail transport planning & management*, 8(1):16–28, 2018.

10 Michel X. Goemans and David P. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. *J. ACM*, 42(6):1115–1145, 1995. doi:10.1145/227683.227684.

11 Mohamed Hamdouni, Guy Desaulniers, Odile Marcotte, François Soumis, and Marianne Van Putten. Dispatching Buses in a Depot Using Block Patterns. *Transportation Science*, 40(3):364–377, 2006.

12 Stephan A Hartmann and Thomas A Runkler. Online Optimization of a Color Sorting Assembly Buffer Using Ant Colony Optimization. In *Operations Research Proceedings 2007*, pages 415–420. Springer, 2008.

13 Richard M. Karp. Reducibility Among Combinatorial Problems. In *Proc. Symposium on the Complexity of Computer Computations*, pages 85–103. Springer, 1972. doi:10.1007/978-1-4684-2001-2_9.

14 Donald Ervin Knuth. *The Art of Computer Programming*, volume 1, chapter 2.2.1, pages 238–243. Addison-Wesley, 2nd edition, 1987.

15 Donald Ervin Knuth. *The Art of Computer Programming*, volume 3, chapter 5.2.3, page 168. Addison-Wesley, 2nd edition, 1998.

16 Felix G König, Macro Lübbecke, Rolf Möhring, Guido Schäfer, and Ines Spenke. Solutions to Real-World Instances of PSPACE-Complete Stacking. In *Proc. European Symposium on Algorithms (ESA)*, pages 729–740. Springer, 2007.

17 Felix G König and Marco E Lübbecke. Sorting With Complete Networks of Stacks. In *Proc. International Symposium on Algorithms and Computation (ISAAC)*, pages 895–906. Springer, 2008.

18 Adeline Pierrot and Dominique Rossin. 2-Stack Sorting is Polynomial. *Theory of Computing Systems*, 60(3):552–579, 2017. doi:10.1007/s00224-016-9743-8.

19 Marc Pont. The Bi-Stack Sorting Problem. Master's thesis, Department of Data Science and Knowledge Engineering, Maastricht University, 2019.

20 Rebecca Smith. Comparing Algorithms for Sorting with t Stacks in Series. *Annals of Combinatorics*, 8(1):113–121, 2004. doi:10.1007/s00026-004-0209-3.

21 Robert Tarjan. Sorting Using Networks of Queues and Stacks. *J. ACM*, 19(2):341–346, 1972.

22 Kevin Tierney, Dario Pacino, and Rune Møller Jensen. On the Complexity of Container Stowage Planning Problems. *Discrete Applied Mathematics*, 169:225–230, 2014.

23 W. Unger. On the k-colouring of circle graphs. In *Proc. International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 61–72. Springer, 1988.