Efficient, High-Quality Stack Rearrangement

Shuai D. Han[1], Nicholas M. Stiffler[1], Kostas E. Bekris[1], and Jingjin Yu[1]

***Abstract*—This work studies rearrangement problems involving the sorting of robots or objects in stack-like containers, which can be accessed only from one side. Two scenarios are considered: one where every robot or object needs to reach a particular stack, and a setting in which each robot has a distinct position within a stack. In both cases, the goal is to minimize the number of stack removals that need to be performed. Stack rearrangement is shown to be intimately connected to pebble motion problems, a useful abstraction in multi-robot path planning. Through this connection, feasibility of stack rearrangement can be readily addressed. Lower and upper bounds on optimality are established, which differ only by a logarithmic factor, in terms of stack removals. An algorithmic solution is then developed that produces suboptimal paths much quicker than a pebble motion solver. Furthermore, informed search-based methods are proposed for finding high-quality solutions. The efficiency and desirable scalability of the methods is demonstrated in simulation.**

***Index Terms*—Foundations of Automation, Planning, Scheduling and Coordination, Inventory Management, Manipulation Planning, Task Planning**

I. Introduction

MANY robotic applications involve the handling of multiple stacks. For instance, spatial restrictions in growing urban areas already motivate stackable parking lots for vehicles, based on robotic technology[1], as in Fig. 1(a). Similarly, products in convenience stores are frequently arranged in "gravity flow" shelves depending on their type, as in Fig. 1(b). Such stacked products arise in the industry where a robot is able to interact with the foremost object and perform operations similar to a "pop" or a "push" of a stack.

In the above stack rearrangement setups, the objective may be to remove a specific object from the stack (e.g., a specific car from the stackable parking lot) or to rearrange the objects into a specific arrangement, which specifies the location of each object within a stack (e.g., a Hanoi tower-like setting). High quality solutions are more desirable for applications, which critically depend on reducing the number of stack pop and push operations. Otherwise, an exorbitant amount of time is spent performing redundant actions, which reduces efficiency or appears unnatural to people.

Manuscript received: September 10, 2017; Revised November 22, 2017; Accepted January 7, 2018.
This paper was recommended for publication by Editor Han Ding upon evaluation of the Associate Editor and Reviewers' comments.
This work is supported by NSF awards IIS-1617744, IIS-1451737, IIS-1734419, and CCF-1330789, as well as internal support by Rutgers University. Opinions or findings expressed here do not reflect the views of the sponsor.
[1]S. D. Han, N. M. Stiffler, K. E. Bekris, and J. Yu are with the Department of Computer Science, Rutgers, the State University of New Jersey, Piscataway, NJ, USA. {shuai.han, nick.stiffler, kostas.bekris, jingjin.yu} @ rutgers.edu
Digital Object Identifier (DOI): see top of this page.

1 A stackable parking lot refers to a setup where a robotic lift has access only to the foremost vehicle in a stack of vehicles. Access to vehicles further away from the lift access point necessitates the removal of intermediary vehicles.

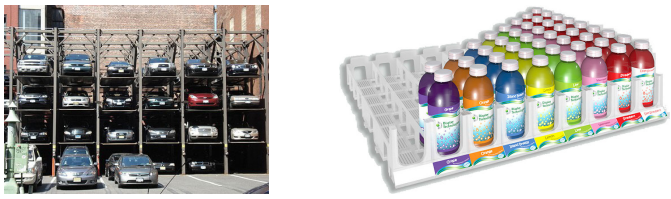


Fig. 1. (a) Stackable parking lots are expected to become even more popular in urban environments with the advent of autonomous cars. (b) Rearranging stacks of objects is a task often encountered in convenience and grocery stores.

Through a reduction to a pebble motion problem, which is well-studied in the multi-robot literature, the feasibility of stack rearrangement can be decided. A naive feasible solution, however, can be far from optimal in minimizing stack removals. Adapting a divide-and-conquer technique, this paper establishes asymptotic lower and upper bounds on this number that differ by a logarithmic factor. Results are provided both for objects that need to be placed in the right stack as well as the case where objects need to acquire a specific stack position. Finally, the paper considers both optimal and sub-optimal informed search methods and proposes effective heuristics for stack rearrangement. This leads to an experimental evaluation of the different methods and heuristics, which suggests a combination that scales nicely with the number of objects.

Related Work: *Multi-body planning* is itself hard. In the continuous case, complete approaches do not scale even though methods try to decrease the effective DOFs [1]. For specific geometries, e.g., unlabeled unit-discs among polygons, optimality is possible [2], even though the unlabeled case is still hard [3]. Given the problem's hardness, decoupled methods, such as priority-based schemes [4] or velocity tuning [5], trade completeness for efficiency.

Recent progress has been achieved for the discrete problem variant, where robots occupy vertices and move along edges of a graph. For this "pebble motion on a graph" problem [6]–[9], feasibility can be answered in linear time and paths can be acquired in polynomial time [10]–[13]. The optimal variation is still hard but optimal solvers with good practical efficiency have been developed [12]–[15]. More recently, $O(1)$-optimal solutions are shown to be efficiently computable in practical environments [16]. The current work is motivated by this progress and aims to show that for stack rearrangement it is possible to come up with practically efficient algorithms.

General rearrangement planning [17]–[19] is also hard, similar to the related "navigation among movable obstacles" (NAMO) task [20]–[24], which can be extended to manipulation among movable obstacles (MAMO) and related challenges [25]–[31]. These efforts focus on feasibility and no solution quality arguments have been provided. A recent work has focused, on high-quality rearrangement solutions but in the context of manipulation challenges in tabletop environments [32].

Variants of the stack rearrangement problem have been studied in the field of operations research. An optimal MLP-based solver was constructed for the scenario where objects need only to be moved to a target stack without specifying their exact position within the stack [33]. There has also been research towards estimating and minimizing the number of relocations of multiple objects in stacks [34], [35]. While this work investigates a similar problem, the emphasis of the current work is on the structural and algorithmic study of problems that relate to robotics applications.

II. Problem Formulation

Assume n objects $\mathcal{O} = \{o_1, \dots, o_n\}$ that occupy $w+1$ last-in-first-out (LIFO) queues, i.e., *stacks*, where $w \geq 2$, since 2-stack rearrangement is impossible. Elements can only be added or removed from one end of the data structure, often referred to as the "top". Furthermore, each stack has an integer *depth* of $d \geq 1$, corresponding to the maximum stack capacity. An object at the top of a stack has a depth of 1.

Modeling many real world problems, the assumption is that objects in a stack always occupy contiguous positions, e.g., if the top object is removed from a stack in Fig. 1(b), the remaining objects will "slide" towards the opening. Similarly, as an object is pushed into a stack, the existing objects will shift backwards by one position. It is straightforward to see that the two versions of the problem, as shown in Fig. 2, are equivalent. The difference denotes whether objects in the stacks gravitate towards the top or bottom. The setup in Fig. 2(a) is used for the remainder of the paper.

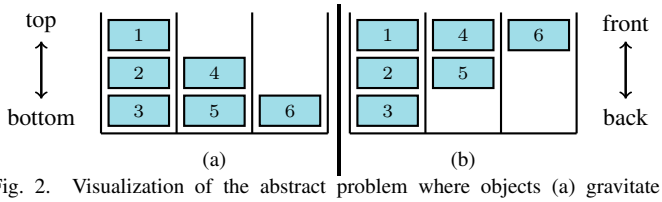


Fig. 2. Visualization of the abstract problem where objects (a) gravitate towards the bottom of the stacks, (b) slide to the opening of the stacks.

Using this setup, an object that currently resides at the top of stack i can be transferred to an arbitrary stack j via a *pop-and-push action*, denoted as a_{ij}. A *permissible pop-and-push action* constrains the above definition by requiring that i be non-empty, and that j not currently be at capacity.

An *arrangement* is an injective mapping $\pi : \mathcal{O} \to \mathbb{N}^2$, $o_i \mapsto \pi(o_i)$. Here, $\pi(o_i)$ is a 2-tuple $(\pi^1(o_i), \pi^2(o_i))$ in which $1 \leq \pi^1(o_i) \leq w+1$ and $1 \leq \pi^2(o_i) \leq d$ are the stack and depth locations of o_i, respectively. The paper primarily focuses on two main problems, defined as follows.

Problem 1. Labeled Stack Rearrangement (`LSR`). *Given $\langle \mathcal{O}, w, d, \pi_I, \pi_G \rangle$, compute a sequence of permissible pop-and-push actions $A = (a_{i_1 j_1}, a_{i_2 j_2}, \dots)$ that move the objects from an initial arrangement π_I to a goal arrangement π_G.*

Problem 2. Column-Labeled Stack Rearrangement (`C-LSR`). *Similar to `LSR`, but the objects are only required to be moved to their goal stacks without a specific depth. That is, π_G^2 is left unspecified for all objects.*

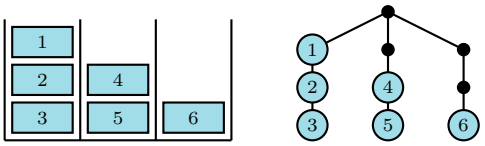
Whereas `C-LSR` appears less general, it has practical incarnations – perhaps more so than `LSR`. For example, in retail, it is almost always the case that a shelf slot holds the same type of product (e.g., Fig. 1(b)). Solving `C-LSR` then corresponds to rearranging an out of order shelf so that each stack holds only a single type of product.

In this paper, the *optimization objective* is to minimize the number of actions taken, i.e. $|A|$. In robotic manipulation, the objective models the required number of grasps by the robotic manipulator, which is frequently the key limiting factor. As such, this work assumes a unit cost to move an object between any two arbitrary stacks and ignores other less significant cost such as the cost of transporting objects between stacks.

Focused on generating high quality solutions, it is assumed that $n \leq wd$, unless specified otherwise. This assumption ensures that `LSR` and `C-LSR` with arbitrary π_I, π_G are always feasible. Details are further discussed in the following section.

III. Structural Analysis

A closely related problem is *Pebble Motion on Graphs* (`PMG`) [6]: suppose an undirected graph $G = (V, E)$ has $p < |V|$ pebbles placed on distinct vertices and which can move sequentially to adjacent empty vertices. Given a `PMG` instance $\langle G, x_I, x_G \rangle$, the goal of `PMG` is to decide if the *configuration* x_G is reachable from x_I, and to subsequently find a sequence of moves to do so when possible. When G is a tree, this problem is referred to as *Pebble Motion on Trees* (`PMT`). The considered versions of `LSR` (and `C-LSR`) are `PMT` problems.



(a) A `LSR` instance (b) A `PMT` instance

Fig. 3. From `LSR` to `PMT`

Proposition III.1. *A `LSR` instance is always reducible to a `PMT` instance. In particular, a solution to the reduced `PMT` instance is also a solution to the initial `LSR` instance.*

Proof. Given a `LSR` instance $\langle \mathcal{O}, w, d, \pi_I, \pi_G \rangle$, as shown in Fig. 3, the tree graph $T = (V, E)$ in the `PMT` instance is obtained by first viewing each stack as a path of length d, and then joining the top vertices of these stacks with a root vertex, which builds the connection between them. This yields $|V| = ((w+1)d) + 1$ vertices. It is clear that object arrangements π_I and π_G directly map to configurations x_I and x_G of a `PMT` instance. Note that a pop-and-push action in the `LSR` solution is equivalent to moving one pebble from a path on T to another path through the root vertex. Similarly, given a solution to the `PMT` instance, a solution to the `LSR` instance can be constructed by treating a pebble passing through the root as a pop-and-push action. □

Given the relationship between `PMT` and `LSR`, and that finding optimal solutions (i.e., a shortest solution sequence) for `PMG` and its variants is NP-hard [36], [37], there is evidence to believe that solving `LSR` optimally (i.e., minimizing the number of actions) is also computationally hard.

In terms of feasibility, the `LSR` problem is always feasible as defined. This is due to the assumption that $n \leq wd$ while the total number of slots in the stacks are $(w+1)d$. This allows to always clear one stack of depth d and then the elements in

the remaining stacks can be arranged with the aid of the empty one. Consider, however, a more general version, called GLSR, that allows for n to exceed wd. So there may be fewer than d buffers available to rearrange objects.

Note that Proposition III.1 still holds for GLSR. The mapping from GLSR to PMT immediately leads to algorithmic solutions for GLSR (and therefore, LSR). By Proposition III.1, a GLSR instance is feasible if and only if the corresponding PMT instance is so. The feasibility test of PMT can be performed in linear time [8], so the same is true for GLSR as the reduction can be performed also in linear time.

For a feasible GLSR, solving the corresponding PMT can be performed in $O(|V|^3)$ running time (and pebble moves) [6]. This translates to a solution for GLSR that runs in $O(w^3d^3)$ time using up to $O(w^3d^3)$ actions. Nevertheless, this upper bound is no longer tight for LSR (i.e., when $n \leq wd$) as a LSR instance is in fact always feasible. *It turns out that* LSR *can be solved with less computational effort and a reduced number of actions than using a* PMT *solver*. This contribution is established in the following proposition.

Proposition III.2. *An arbitrary* LSR *can be solved using* $O(wd^2)$ *pop-and-push actions.*

Proof. Consider a LSR with $n = wd$. Without loss of generality, assume that: $\forall\ o \in \mathcal{O} : \pi_I^1(o) \leq w, \pi_G^1(o) \leq w$, i.e., stack $w+1$ is empty at the start and goal arrangement. It suffices to show that one stack (e.g., the first) can be rearranged in $O(d^2)$ actions and this can be repeated w times. The $O(d^2)$ cost for a stack is because each object can be moved to its destination in $O(d)$ moves and this can be repeated for d times. Consider the object o to be moved to the bottom of the first stack, i.e., $\pi_G(o) = (1, d)$. Without loss of generality, assume that $\pi_I(o) = (x, y)$ with $x \neq 1$. Initially, o will be moved to the top of stack x. If $y = 1$, no action is needed. Otherwise, perform the following moves per Fig. 4: *(i)* move the object at $(1, 1)$ to the buffer stack $(w + 1)$, *(ii)* move objects from $(x, 1)$ to $(x, y - 1)$ to the buffer, *(iii)* move o to $(1, 1)$, *(iv)* move all objects in the buffer except the last to stack x, *(v)* move o to the top of stack x, and *(vi)* move the last object in the buffer to stack 1.

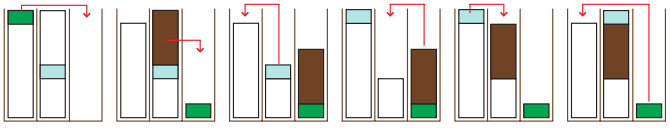


Fig. 4. The cyan object moves to the top of its stack x with $O(d)$ actions.

Using the same $O(d)$ procedure, the object o' at $(1, d)$ can be moved to $(1, 1)$. Using the buffer stack $(w+1)$, o and o' can be swapped in three actions. Then, reverting the sequences, o can be moved to $(1, d)$ for an $O(d)$ total number of actions. So, rearranging a single stack needs $O(d^2)$ actions and the entire solution takes $O(wd^2)$ actions. □

The running time is also bounded by $O(wd^2)$ since the only computational cost is to go through π_I and π_G and recover the solution sequence. For GLSR, which allows $n > wd$, an arbitrary instance may not be feasible. This paper focuses on the optimal number of actions for rearrangement problems, so GLSR is not considered further.

IV. Fundamental Bounds on Optimality

This section provides an analysis on the structural properties of LSR, focusing on the fundamental optimality bounds and polynomial time algorithms for computing them. The analysis assumes the hardest case of LSR where n equals wd. Without loss of generality, it is assumed that stack $(w + 1)$ is empty at the initial and goal arrangement, serving as a *buffer*. First, consider the lower bound on the number of required actions.

Proposition IV.1. *In the average case,* $\Omega(wd)$ *actions are required for solving* LSR.

Proof. First consider a worst case scenario, i.e., that the deepest objects $o_i \in \mathcal{O}$ in each stack k, i.e., $\pi_I(o_i) = (k, d)$, must be moved to the next stack $k + 1$ modulo w, i.e., $\pi_G(o_i) = ((k + 1) \mod w, d)$. To move each of these objects, at least d actions are needed because $d - 1$ objects are blocking the way to them. Therefore, the total number of required actions is $\Omega(wd)$.

In the average case (assuming π_I and π_G are both uniformly randomly generated), each object o has probability $(w-1)/w$ to have $\pi_I^1(o) \neq \pi_G^1(o)$. That is, with probability $1/w$, o will stay in its initial stack and with probability $(w - 1)/w$ it must be moved to a different stack. Because moving o will require on average $d/2$ actions, the expected cost of moving it is then $(w - 1)d/(2w)$. For all w stacks, this is then $\Omega((w - 1)wd/(2w)) = \Omega(wd)$. □

Then, the following lemma holds.

Lemma IV.1. *A lower bound on the number of moves for solving* LSR *is* $\Omega(wd \log d/ \log w)$.

Proof. The bound is established by counting the possible LSR problems for fixed w and d, i.e., for the case $n = wd$. Given a fixed π_I, there are $n!$ possible π_G, so there are at least $n! = (wd)!$ different LSR instances. With each action, one object at the top of a stack ($w + 1$ of these) can be moved to any other stack (w of these). Therefore, each action can create at most $w(w + 1) < (w + 1)^2$ new arrangements. In order to solve all possible LSR instances, it must then be the case that the required number of moves, defined as $\min\{|A|\}$, must satisfy $[(w + 1)^2]^{\min\{|A|\}} \geq (wd)!$. Then, by Stirling's approximation, $\min\{|A|\} = \Omega(wd\frac{\log d}{\log w})$. □

Lemma IV.1 implies Proposition IV.1 as well but does so in a less direct way. Interestingly for the case of $w \ll d$, Lemma IV.1 immediately implies the following better lower bounds.

Corollary IV.1. *For a* LSR *with* $w = e^{\sqrt{\log d}}$, *on average it requires* $\Omega(wd\sqrt{\log d})$ *actions to solve.*

Corollary IV.2. *For a* LSR *with* w *being a constant, on average it requires* $\Omega(d \log d)$ *actions to solve.*

The focus now shifts towards upper bounds on optimality where polynomial time algorithms are presented for computing them. Recall that a trivial upper bound of $O(wd^2)$ is given by Proposition III.2. Comparing the $O(wd^2)$ upper bound with the lower bound, which ranges between $\Omega(wd)$ and $\Omega(d \log d)$ (for constant w), there remains a sizable gap. Forthcoming algorithms illustrate how to significantly reduce, and in certain cases eliminate this gap.

Lemma IV.2. *An arbitrary instance of* C−LSR *can be solved using* $O(wd \log w)$ *actions.*

Proof. A recursive algorithm is outlined for solving C−LSR. The $\log w$ factor in $O(wd \log w)$ is a result of a divide-and-conquer approach, similar to the one that appears in popular sorting algorithms like quicksort. In the first iteration, partition all wd objects into two sets based on π_G^1. For an object $o \in \mathcal{O}$, if $\pi_G^1(o) \leq \lceil w/2 \rceil$, then it is assigned to the *left* set. Otherwise it is assigned to *right* set. The goal of the first iteration is to sort objects so that the left set resides in stacks 1 to $\lceil w/2 \rceil$, as illustrated in Fig. 5.

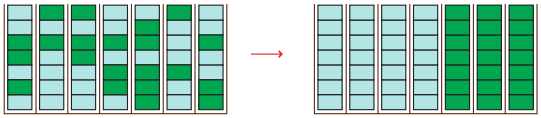


Fig. 5. The goal of the first iteration in solving an C−LSR instance with $w = 7$ and $d = 7$. The empty stack is not drawn.

In the first iteration, begin with the first stack and sort it into two contiguous sections belonging to the left set and the right set. The process involves using another occupied stack and the buffer stack (using $O(d)$ moves). Note that the content of the other occupied stack is irrelevant. These three stacks are illustrated in the first figure in Fig. 6. Assume that ℓ objects of stack 1 belong to the left set (in the example, $\ell = 4$). To begin, the top ℓ objects of the last stack is moved to the buffer. This allows the sorting of the first stack into two contiguous blocks of left only and right only objects, which can then be returned to the first stack. Note that the order of the two blocks can be reversed using the same procedure; this will be used shortly. The procedure is then applied to all stacks. The procedure and the end result are illustrated in Fig. 6. It is clear that the total actions required is $O(wd)$.

Fig. 6. The left four figures illustrate the process of sorting a single stack into two contiguous blocks. The last figure is the end result of applying the procedure to all stacks.

The next step involves the first two stacks and attempts to *consolidate* the sets. If any stack is already fully occupied by either the left or the right set, then that stack can be skipped; suppose not. Let these two stacks be i-th and j-th stacks and let ℓ_i and ℓ_j be the number of objects belonging to the left set in the i-th and j-th stacks, respectively. If $\ell_i + \ell_j \geq d$, then using the buffer stack, stack i can be forced to contain only objects belonging to the left set. Fig. 7 illustrates applying the procedure to the left most two stacks to the running example and the result.

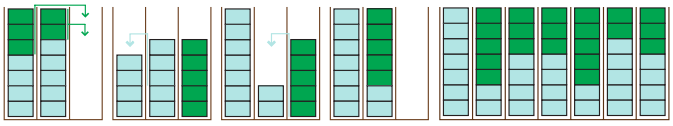


Fig. 7. Consolidating the first two stacks.

If $\ell_i + \ell_j < d$, then stack i is processed so that the ℓ_i objects belonging to the left set are on the top (using the block reverse procedure mentioned earlier in this proof). Then, a similar consolidation routine can be applied. Fig. 8 illustrates the application of the procedure to stacks 2 and 3 of the right most figure of Fig. 7. With these two variations, all stacks can be sorted so that each stack contains only objects from either the left set or the right set.

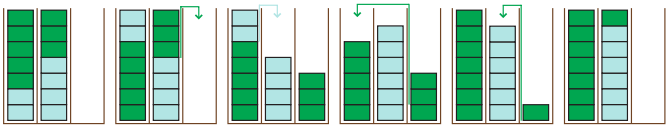


Fig. 8. Consolidating the first two stacks that requires reversing the two contiguous blocks one of the two stacks.

At this point, using the buffer stack, entire stacks can be readily swapped to complete the first iteration. The total number of actions used is $O(wd)$ per iteration. Applying the same iterative procedure to the left and the right sets of objects, the full C−LSR problem can then be solved with $O(wd \log w)$ actions. □

After solving C−LSR, each stack needs to be sorted again to fully solve the original LSR problem, which can be performed using $O(d \log d)$ actions.

Lemma IV.3. *After solving the* C−LSR *portion of a* LSR *instance, a stack can be fully sorted using another stack and the buffer stack with* $O(d \log d)$ *actions.*

Proof. The sorting is done recursively. Suppose stack i is to be sorted using stack j and the buffer stack. Assume without loss of generality that $d = 2^k$ for some k. To start, move half of the objects in stack j to the buffer stack. This creates two buffers of size 2^{k-1}. Using these two buffers, stack i can be sorted into a top half and a bottom half. As these two halves are restored to stack i, the top and bottom halves are separated. Iteratively applying the same procedure can then sort stack i fully in $\log d$ iterations. The total number of required actions is then $O(d \log d)$. Fig. 9 provides an illustrative example sorting sequence for $k = 3$. □

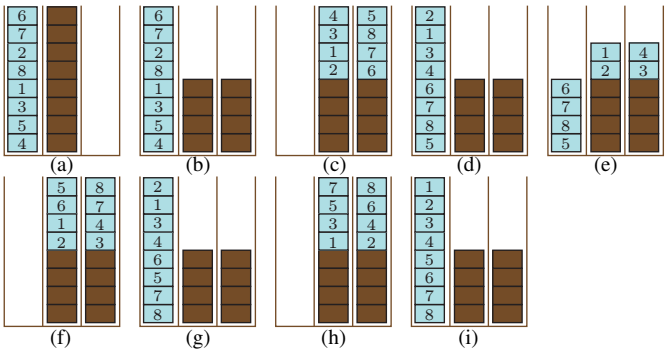


Fig. 9. An example of sorting a stack with $d = 2^3$.

Lemma IV.2 and Lemma IV.3 suggest that the gap between the lower and upper bounds for LSR can be completely eliminated for constant w.

Theorem IV.1. *For constant* w, LSR *can be solved using* $O(d \log d)$ *actions, agreeing with the* $\Omega(d \log d)$ *lower bound.*

Proof. By Lemma IV.2, when w is a constant, a corresponding C−LSR problem can be solved using $O(wd \log w) = O(d)$ actions. Lemma IV.3 then applies to do the final sorting in $O(wd \log d) = O(d \log d)$ actions. The total number of

required actions is then $O(d \log d)$. The lower bound is given by Corollary IV.2. □

The following provides a tighter upper bound for non-constant w.

Theorem IV.2. *An arbitrary* LSR *instance can be solved using* $O(wd \max\{\log d, \log w\})$ *actions.*

Proof. Applying Lemma IV.2 to the LSR problem yields a stack sorted instance of LSR using $O(wd \log w)$ actions. Since sorting each of the stacks afterward takes $O(d \log d)$ time, the full LSR problem can be solved using $O(wd \max\{\log d, \log w\})$ actions. □

The greatly improved and general upper bound is now fairly close to the general lower bound $\Omega(wd)$ within only a logarithmic factor.

Algorithm 1: Poly-LSR(π_I, π_G)

```
1  Poly-C-LSR(1, w)
2  for c_i ∈ [1, w] do Sort stack c_i
3  return
4  Function Poly-C-LSR(l, r)
5      if l = r then return
6      i ← l, j ← r, m ← ⌊(l + r)/2⌋
7      for o_i ∈ O do
8          if l ≤ π_G^1(o_i) ≤ m then label o_i: left
9          else if m + 1 ≤ π_G^1(o_i) ≤ r then label o_i: right
10     for c_i ∈ [l, r] do Reorder stack c_i
11     for c_i ∈ [l, r] do Consolidate stack c_i
12     Poly-C-LSR(l, m)
13     Poly-C-LSR(m + 1, r)
14     return
```

The algorithmic process (Poly-LSR) for the method introduced above is shown in Alg. 1. It contains the subroutine for solving C−LSR (Poly-C-LSR, in line 4-14), which iteratively separates the left and right sets in each stack (line 12), and consolidate the stacks (line 13). Details are already mentioned in Lemma IV.2. LSR is solved by first call Poly-C-LSR and then sort all the stacks using the routine in Lemma IV.3 (line 2). The overall time complexity is $O(wd \max\{\log d, \log w\})$, which is equivalent to the number of actions in the solution.

V. Optimal and Suboptimal LSR Solvers

The polynomial algorithms (Sections III and IV) provide a balance between computational complexity and solution quality. These algorithms are more appropriate for solving large-scale problems quickly with a bounded sub-optimality guarantee. Nevertheless, given that the computational burden is relatively manageable for smaller problem instances, optimal solutions can be computed via alternative search-based solvers, which are introduced in this section. Appropriate heuristics are developed to guide these solvers, which helps in making them tractable for larger problem instances.

LSR can be reduced to a *Shortest Path Problem*, which searches for a minimum weight path between two nodes in an undirected graph. Here a node simply denotes an arrangement π. The neighbors of this node are all the arrangements reachable from π via a single pop-and-push action. The edge weights between connected nodes are uniform.

The A* graph search algorithm [38] is a common tool for solving such a problem optimally. The branching factor is $(w+1)w$ since a pop-and-push action picks an object from one of $(w+1)$ stacks, and places it in one of the other w stacks. Several heuristic functions are designed to guide the search:

Depth Based Heuristic (DBH). This heuristic returns an admissible number of pop-and-push actions needed to move a single object $o_i \in \mathcal{O}$ to its goal. The detailed process appears in Alg. 2. It initially checks if o_i is at its goal position (line 1), and simply returns 0 if this statement is true. Lines 2 and 3 calculate the number of objects in front of o_i in π_C (resp. π_G), and denote it as n_c (resp. n_g). At this point (line 4), if the object is currently in its goal stack, DBH computes the estimated number of moves via the following process: (1) take o_i out of $\pi_G^1(o_i)$, (2) make the goal pose reachable by inserting/removing intermediary objects from $\pi_G^1(o_i)$, and (3) place o_i back into $\pi_G^1(o_i)$. If the object is not in its goal stack, then one of the following apply: If $\pi_G(o_i)$ is reachable from $\pi_C(o_i)$ solely by removing intermediary objects in front of both positions, then the object can be moved to $\pi_G(o_i)$ in $n_c + n_g + 1$ steps; Otherwise, o_i needs to be moved to an intermediate stack and this induces an extra move.

Algorithm 2: DBH(o_i, π_C, π_G)

```
1  if π_C(o_i) = π_G(o_i) then return 0
2  n_c ← |{o_j | o_j ∈ O, π_C^1(o_j) = π_G^1(o_i), π_C^2(o_j) < π_G^2(o_i)}|
3  n_g ← |{o_j | o_j ∈ O, π_C^1(o_j) = π_G^1(o_i), π_C^2(o_j) < π_G^2(o_i)}|
4  if π_C^1(o_i) = π_G^1(o_i) then
5      if n_c > n_g then return 2n_c − n_g + 2
6      else return n_g + 2
7  else
8      if π_C^2(o_i) + π_G^2(o_i) − 1 ≤ (w + 1)d − n then
9          return n_c + n_g + 1
10     else return n_c + n_g + 2
```

The following variants of DBH deal with multiple objects:

1) DBH1: admissible, takes the maximum DBH value over all objects: $h_{\text{DBH1}} = \max_{o \in \mathcal{O}} \text{DBH}(o, \pi_C, \pi_G)$.
2) DBHn: inadmissible, takes the summation of DBH values: $h_{\text{DBHn}} = \sum_{o \in \mathcal{O}} \text{DBH}(o, \pi_C, \pi_G)$.

Column Based Heuristic (CBH). Described in Alg. 3, CBH counts the summation of the minimum number of actions necessary to move each object to its goal stack. As opposed to DBH which seeks a tight estimate for a single object, CBH considers all objects.

The detailed process is as follows. For every object $o_i \in \mathcal{O}$, CBH first determines whether $\pi_C^1(o_i) = \pi_G^1(o_i)$ (line 3). If $\pi_C^1(o_i) = \pi_G^1(o_i)$, the heuristic value h remains unchanged if the objects behind o_i are all at their goals. Otherwise, there exists either an object currently deeper than o_i that needs to be evacuated, or an object in another stack that needs to be inserted to $\pi_G^1(o_i)$ at a depth deeper than $\pi_G^2(o_i)$. Thus o_i must be taken out of its goal stack and placed back afterwards. This requires 2 actions on o_i (line 4).

If $\pi_C^1(o_i) \neq \pi_G^1(o_i)$, it takes at least 1 action for o_i to be moved to $\pi_G^1(o_i)$ (line 7). However if the empty locations in the stacks other than $\pi_C^1(o_i)$ and $\pi_G^1(o_i)$ cannot contain all the objects in front of $\pi_C(o_i)$ and $\pi_G(o_i)$, o_i must be moved to an intermediate stack. This requires 2 actions (line 6).

An example of DBH and CBH calculation is shown in Fig. 10. The running time for DBH is $O(d)$, so totally $O(nd)$ for both DBH1 and DBHn. CBH runs in $O(n)$ time when dealing objects in each stack from the bottom. The admissible

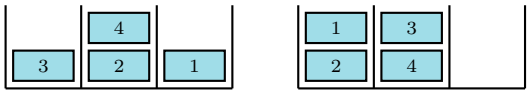


Fig. 10. An example for heuristic calculation. The left figure denotes π_C, while the right one denotes π_G. The heuristic values are $h_{\text{DBH1}} = \max\{3,1,3,4\} = 4$, $h_{\text{DBHn}} = 3{+}1{+}3{+}4 = 11$, $h_{\text{CBH}} = 2{+}1{+}2{+}2 = 7$, The optimal solution for this problem costs 9 steps.

Algorithm 3: CBH(π_C, π_G)

```
h ← 0
for o_i ∈ O do
    if π^1_C(o_i) = π^1_G(o_i) then
        if ∃o_j ∈ O, π^1_C(o_j) = π^1_C(o_i), π^2_C(o_j) >
          π^2_C(o_i), π_C(o_j) ≠ π_G(o_j) then h ← h + 2
    else
        if (w+1)d − n < π^2_C(o_i) + π^2_G(o_i) − 1 then h ← h + 2
        else h ← h + 1
return h
```

heuristics CBH and DBH1 are also consistent. Proofs are omitted due to the lack of space.

An alternate optimal solver is *bidirectional heuristic search* (BHPA) [39]. It runs two A* searches simultaneously: One starts from π_I and searches for π_G; the other starts from π_G and searches for π_I. BHPA terminates when it finds a path with cost $\mu \leq \max\{f_I, f_G\}$. Here f_I and f_G denote the minimum f-values in the two search fringes, respectively.

By multiplying the heuristic value with a weight $\omega > 1$, *weighted A* search* [40] generates ω-approximate solutions, and runs significantly faster than A* search. Weighted A* is denoted as A*(ω) and weighted BHPA as BHPA(ω).

Remark. Other algorithms, including, but not limited to, ALT [41], ID [42], CBS [43], ILP [15], although efficient in solving search or PMG problems, are expected to underperform on LSR because of the high density and lack of parallel movements. Details are omitted due to the lack of space.

VI. Experimental Results

This section presents experimental validation for the algorithms introduced in this paper. All experiments were executed on a Intel® Core™ i7-6900K CPU with 32GB RAM at 2133MHz.

Both the *success rate* (Fig. 11-12) and *average cost* (Table I) are evaluated for each problem setup. The success rate is the percentage of instances that generated a solution before a five second timeout occurred. The quality of solutions is presented as the average number of actions $|A|$. The experiments were conducted with varying values for w, d, and n. For each problem setup (w, d, n), 100 random instances are generated and the average solution cost is reported. To observe the robustness of the algorithms to various parameter values, three sets of experiments were conducted that fixed certain parameters while varying others. The following table shows the position of the experiments as they appear relative to Table I and the Figures 11-12 along with the accompanying parameter values.

Position	w	d	n
top	varies	3	wd
middle	2	varies	wd
bottom	5	5	varies

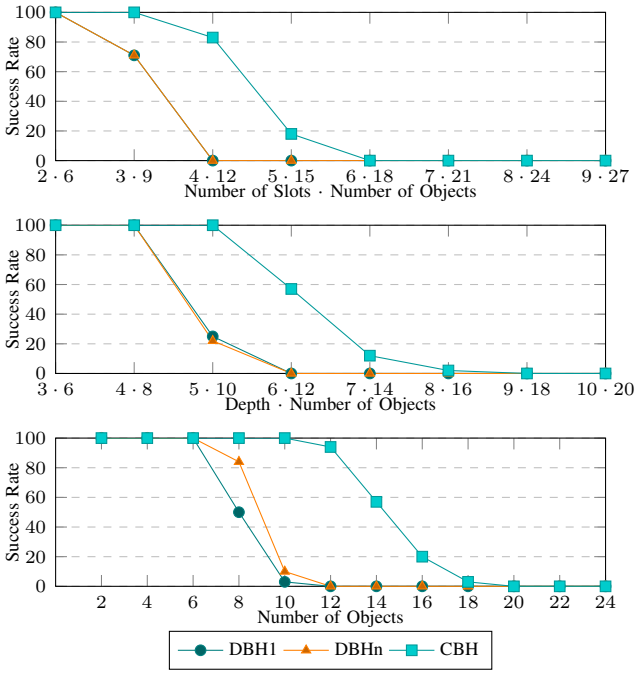


Fig. 11. Success rate of heuristics when applied to the A* algorithm.

Polynomial algorithms are implemented with basic post-processing, which removes back-and-forth actions (i.e., transfer an object to another stack and immediately move it back). Poly-D is the simple $O(wd^2)$ algorithm in Section III, while Poly-LSR is the $O(wd\max\{\log d, \log w\})$ algorithm in Section IV. The yellow columns in Table I show the average solution cost of Poly-D and Poly-LSR across all simulated instances. Poly-D generates solutions with a lower average cost than Poly-LSR when n is small. The performance flips when there is more than 1000 objects. For example, when $w = 50, d = 40, n = 2000$, Poly-D uses $\sim 63,000$ steps to solve a problem, while Poly-LSR uses $\sim 50,000$ steps. Both the Poly-D and Poly-LSR algorithms are able to solve LSR problems with 1000 objects in 1 second, which is already beyond a practical number.

The heuristics described in Section V are tested with the A* algorithm. As evidenced in Fig. 11, the admissible heuristic CBH has a higher success rate than the alternatives. The average solution cost across all scenarios that employ an A* heuristic appear in the brown and red columns of Table I. Fig. 12 shows that with the help of CBH, A* has a much higher success rate than *Breadth First Search* (BFS) and its bidirectional version Bi-BFS. As expected, A* also beats BHPA [38].

The weighted search algorithms (red columns in Table I) generate solutions close to the optimal (green column in Table I). BHPA(2) has a higher success rate than A*(2) (see Fig. 12), and also generates solutions with lower cost. This is because as the heuristic becomes inadmissible, the termination criterion of BHPA is more easily to be satisfied.

The key difference between the optimal search-based methods and the suboptimal methods centers around the relaxation of the objective function. By forgoing optimality, the suboptimal methods gain increased scalability, which often generate near-optimal feasible solutions to problem instances that are

TABLE I
AVERAGE SOLUTION COST OF THE ALGORITHMS

w	d	n	Opt.Val.	Poly-D	Poly-LSR	BFS	Bi-BFS	A*, with Heuristic			BHPA	A*(2)	BHPA(2)
								DBH1	DBHn	CBH	CBH	CBH	CBH
2	3	6	12.11	14.07	31.14	12.11	12.11	12.11	12.84	12.11	12.11	13.12	13.49
3	3	9	17.16	24.08	53.95	11.17*	17.16	16.13*	17.58*	17.16	17.16	19.8	19.78
4	3	12	21.95	33.78	83.88	NA	16.4*	NA	NA	21.52*	21.1*	25.35	25.2
5	3	15	*	44.35	114.85	NA	NA	NA	NA	24.72*	24.94*	31.85	31.32
6	3	18	*	54.34	146.53	NA	NA	NA	NA	NA	NA	37.28	36.77
7	3	21	*	64.52	178.49	NA	NA	NA	NA	NA	NA	44.43*	43.98*
8	3	24	*	75.93	217.77	NA	NA	NA	NA	NA	NA	49.73*	49.82*
9	3	27	*	86.99	256.47	NA	NA	NA	NA	NA	NA	56.71*	56.15*
2	3	6	12.11	14.07	31.14	12.11	12.11	12.11	12.84	12.11	12.11	13.12	13.49
2	4	8	18.17	23.97	58.74	18.17	18.17	18.17	20.47	18.17	18.17	20.57	20.27
2	5	10	25.08	35.43	78.96	NA	25.02*	21.48*	25.23*	25.08	25.08	28.62	28.4
2	6	12	32.57	48.90	101.2	NA	25.0*	NA	NA	30.23*	29.57*	37.2	37.41
2	7	14	*	63.95	124.61	NA	NA	NA	NA	35.0*	34.29*	46.21*	46.38
2	8	16	*	79.57	147.63	NA	NA	NA	NA	36.5*	NA	53.88*	54.94*
2	9	18	*	98.54	172.42	NA	NA	NA	NA	NA	NA	61.6*	61.51*
2	10	20	*	117.47	197.3	NA	NA	NA	NA	NA	NA	68.25*	69.45*
5	5	2	1.74	1.86	15.35	1.74	1.74	1.74	1.74	1.74	1.74	1.74	1.74
5	5	4	4.2	5.04	30.73	4.2	4.2	4.2	4.38	4.2	4.2	4.29	4.26
5	5	6	6.87	9.49	45.82	6.65*	6.87	6.87	7.37	6.87	6.87	7.12	7.18
5	5	8	9.62	15.89	61.31	NA	9.6*	8.56*	10.35*	9.62	9.62	10.3	10.28
5	5	10	13.01	22.63	76.93	NA	10.2*	9.33*	11.9*	13.01	13.01	14.69	14.44
5	5	12	16.01	30.42	92.08	NA	NA	NA	NA	15.87*	15.86*	18.49	18.32
5	5	14	19.74	38.86	107.13	NA	NA	NA	NA	18.86*	18.85*	22.79	22.69
5	5	16	*	48.1	122.74	NA	NA	NA	NA	21.5*	21.65*	27.54	27.26
5	5	18	*	57.53	138.08	NA	NA	NA	NA	23.33*	23.25*	33.42	32.92
5	5	20	*	68.17	153.61	NA	NA	NA	NA	NA	NA	38.23*	37.88*
5	5	22	*	80.04	168.41	NA	NA	NA	NA	NA	NA	44.16*	44.12*
5	5	24	*	97.57	183.97	NA	NA	NA	NA	NA	NA	55.69*	55.59*

* Failed instances are not involved in calculating the average cost, which makes the data point less informative.
NA: all test cases failed.
Content explanation:
- The leftmost 3 columns denote different setups of `LSR`.
- Green column: optimal costs achieved by running the A* algorithm with CBH heuristic with 300 seconds timeout. The memory requirements of the problem are a bottleneck for these instances.
- Yellow, blue, brown, and red columns denote results for polynomial algorithms, BFS, optimal search-based solvers, and suboptimal search-based solvers, respectively.

intractable for traditional search-based methods.

The accompanying video[2] provides additional experiments. These experiments illustrate the effectiveness of the proposed methods in application domains, such as shelf stocking via a robot manipulator and automated vehicle parking.

VII. CONCLUSION

This paper describes a novel approach to the object rearrangement problem where objects are stored in stack-like containers. Fundamental optimality bounds are provided by modeling these challenges as pebble motion on a graph problems. While optimal solvers exist to tackle pebble motion on a graph problems, these methods are ill-suited for stack object rearrangement due to the approaches' poor scalability. To overcome this shortcoming, an algorithmic solution is presented, which is faster than optimal solvers, albeit producing suboptimal solutions. The utility of the proposed method is validated experimentally.

The current work assumes a single manipulator, which forces all stack operations to be sequential. A future line of research is to consider parallel operations. A starting point may be the parallelization of the stack operations with multiple manipulators where each manipulator is in charge of a number of stacks. From an algorithmic perspective, the divide-and-conquer algorithm proposed here will naturally benefit from such a scheme. Further parallelization efforts, however, must be considered jointly with the design of the underlying system since carrying out multiple stack operations simultaneously will require careful coordination between the manipulators.

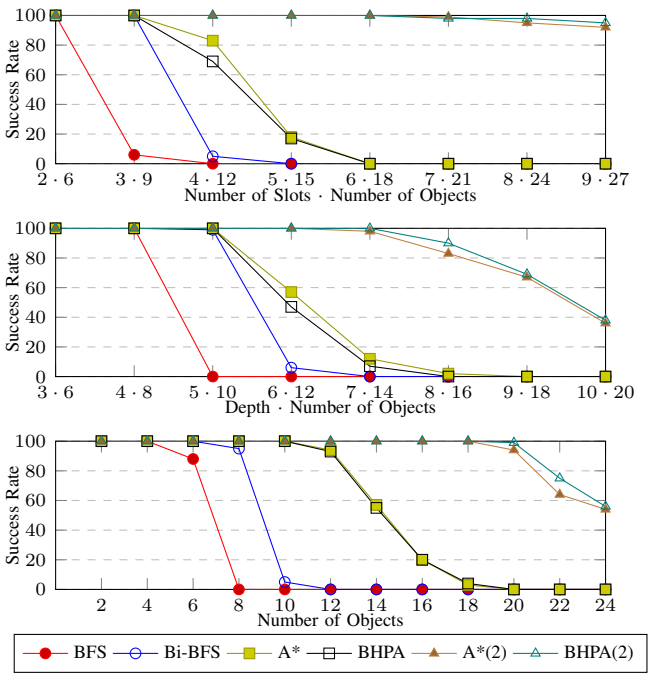


Fig. 12. Success rate of algorithms with the CBH heuristic.

The cost function in this paper focuses on the number of pop and push actions given the high real-world cost of grasping objects in stacks. It can be expanded, however, to reflect the distance between entrances of different stacks. Developing a

2 Avaliable at https://youtu.be/qOIT-lq5S6w

polynomial time algorithm with tight optimality guarantees will be more challenging but also useful for physical systems where the energy expense and execution time are affected by the distance over which an object is transported.

REFERENCES

[1] B. Aronov, M. de Berg, A. F. van den Stappen, P. Švestka, and J. Vleugels, "Motion planning for multiple robots," *Discrete and Computational Geometry*, vol. 22, no. 4, pp. 505–525, 1999.

[2] K. Solovey, J. Yu, O. Zamir, and D. Halperin, "Motion planning for unlabeled discs with optimality guarantees," in *Proc. Robotics: Science and Systems*, Rome, Italy, Jul. 2015.

[3] K. Solovey and D. Halperin, "On the hardness of unlabeled multi-robot motion planning," in *Proc. Robotics: Science and Systems*, Rome, Italy, Jul. 2015.

[4] J. van den Berg and M. Overmars, "Prioritized motion planning for multiple robots," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, Edmonton, AB, Canada, Aug. 2005, pp. 2217–2222.

[5] S. Leroy, J.-P. Laumond, and T. Siméon, "Multiple path coordination for mobile robots: A geometric algorithm," in *Proc. International Joint Conferences on Artificial Intelligence*, Stockholm, Sweden, Jul. 1999, pp. 1118–1123.

[6] D. Kornhauser, G. Miller, and P. Spirakis, "Coordinating pebble motion on graphs, the diameter of permutation groups, and applications," in *Proc. IEEE Symposium on Foundations of Computer Science*, Singer Island, FL, USA, 1984, pp. 241–250.

[7] G. Calinescu, A. Dumitrescu, and J. Pach, "Reconfigurations in graphs and grids," *SIAM Journal on Discrete Mathematics*, vol. 22, no. 1, pp. 124–138, 2008.

[8] V. Auletta, A. Monti, D. Parente, and G. Persiano, "A linear time algorithm for the feasibility of pebble motion on trees," *Algorthmica*, vol. 23, pp. 223–245, 1999.

[9] G. Goraly and R. Hassin, "Multi-color pebble motion on graphs," *Algorthmica*, vol. 58, no. 3, pp. 610–636, 2010.

[10] A. Krontiris, R. Luna, and K. E. Bekris, "From feasibility tests to path planners for multi-agent pathfinding," in *Proc. International Symposium on Combinatorial Search*, Seattle, WA, Jul. 2013.

[11] R. Luna and K. E. Bekris, "Efficient and complete centralized multi-robot path planning," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Francisco, CA, Sep 2011.

[12] G. Wagner, M. Kang, and H. Choset, "Probabilistic path planning for multiple robots with subdimensional expansion," in *Proc. IEEE International Conference on Robotics and Automation*, Minneapolis, MN USA, May 2012.

[13] J. Yu and S. M. LaValle, "Multi-agent path planning and network flow," in *Algorithmic Foundations of Robotics X*. Springer, 2013, pp. 157–173.

[14] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," in *Artificial Intelligence*, no. 219, 2015, pp. 40–66.

[15] J. Yu and S. M. LaValle, "Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics," *IEEE Transactions on Robotics*, vol. 32, no. 5, pp. 1163–1177, 2016.

[16] J. Yu, "Expected constant factor optimal multi-robot path planning in well-connected environments," in *International Symposium on Multi-Robot and Multi-Agent Systems*, 2017.

[17] O. Ben-Shahar and E. Rivlin, "Practical pushing planning for rearrangement tasks," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 4, Aug. 1998.

[18] J. Ota, "Rearrangement planning of multiple movable objects by using real-time search methodology," in *Proc. IEEE International Conference on Robotics and Automation*, vol. 1. Washington DC, USA: IEEE, May 2002, pp. 947–953.

[19] M. Christofides and C. I, "The rearrangement of items in a warehouse," *Operations Research*, vol. 21, no. 2, pp. 577–589, 1973.

[20] G. Wilfong, "Motion planning in the presence of movable obstacles," in *Annals of Mathematics and Artificial Intelligence*, 1991, pp. 131–150.

[21] P. C. Chen and Y. K. Hwang, "Practical path planning among movable obstacles," in *Proc. IEEE International Conference on Robotics and Automation*, Sacramento, CA, USA, Apr. 1991, pp. 444–449.

[22] E. Demaine, J. O'Rourke, and M. L. Demaine, "Pushpush and push-1 are np-hard in 2d," in *Proc. Candadian Conference on Computational Geometry*, 2000, pp. 211–219.

[23] D. Nieuwenhuisen, A. F. van der Stappen, and M. H. Overmars, "An effective framework for path planning amidst movable obstacles," in *Proc. Workshop on the Algorithmic Foundations of Robotics*, New York City, USA, Jul. 2006.

[24] J. van den Berg, M. Stilman, J. J. Kuffner, M. Lin, and D. Manocha, "Path planning among movable obstacles: A probabilistically complete approach," in *Proc. Workshop on the Algorithmic Foundations of Robotics*, Guanajuato, Mxico, Dec. 2008.

[25] M. Stilman, J. Schamburek, J. J. Kuffner, and T. Asfour, "Manipulation planning among movable obstacles," in *Proc. IEEE International Conference on Robotics and Automation*, Rome, Italy, Apr. 2007.

[26] G. Havur, G. Ozbilgin, E. Erdem, and V. Patoglu, "Geometric rearrangement of multiple moveable objects on cluttered surfaces: A hybrid reasoning approach," in *Proc. IEEE International Conference on Robotics and Automation*, Hong Kong, China, May 2014, pp. 445–452.

[27] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *Proc. IEEE International Conference on Robotics and Automation*, Hong Kong, China, May 2014.

[28] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Ffrob: An efficient heuristic for task and motion planning," in *Proc. Workshop on the Algorithmic Foundations of Robotics*, Istanbul, Turkey, Sep. 2014.

[29] A. Krontiris, R. Shome, A. Dobson, A. Kimmel, and K. E. Bekris, "Rearranging similar objects with a manipulator using pebble graphs," in *Proc. IEEE International Conference on Humanoid Robotics*, Madrid, Spain, Nov. 2014.

[30] A. Krontiris and K. E. Bekris, "Dealing with difficult instances of object rearrangement," in *Proc. Robotics: Science and Systems*, Rome, Italy, Jul. 2015.

[31] ——, "Efficiently solving general rearrangement tasks:a fast extension primitive for an incremental sampling-based planner," in *Proc. IEEE International Conference on Robotics and Automation*, Stockholm, Sweden, May 2016.

[32] S. Han, N. M. Stiffler, A. Krontiris, K. E. Bekris, and J. Yu, "High-quality tabletop rearrangement with overhand grasps: Hardness results and fast methods," in *Proc. Robotics: Science and Systems*, Cambridge, MA, Jul. 2017.

[33] N. R. Dayama, M. Krishnamoorthy, A. Ernst, V. Narayanan, and N. Rangaraj, "Approaches for solving the container stacking problem with route distance minimization and stack rearrangement considerations," *Computers & Operations Research*, vol. 52, pp. 68–83, 2014.

[34] K. H. Kim, "Evaluation of the number of rehandles in container yards," *Computers & Industrial Engineering*, vol. 32, no. 4, pp. 701–711, 1997.

[35] M. Caserta, S. Voß, and M. Sniedovich, "Applying the corridor method to a blocks relocation problem," *OR spectrum*, vol. 33, no. 4, pp. 915–929, 2011.

[36] O. Goldreich, "Studies in complexity and cryptography," O. Goldreich, Ed. Berlin, Heidelberg: Springer-Verlag, 2011, ch. Finding the Shortest Move-sequence in the Graph-generalized 15-puzzle is NP-hard, pp. 1–5.

[37] D. Ratner and M. Warmuth, "The (n^2-1)-puzzle and related relocation problems," *Journal of Symbolic Computation*, vol. 10, no. 2, pp. 111–137, 1990.

[38] E. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 2, pp. 100–107, 1968.

[39] I. Pohl, "Bi-directional and heuristic search in path problems," Ph.D. dissertation, Stanford University, Dept. of Computer Science, 1969.

[40] J. Pearl, "Heuristics: intelligent search strategies for computer problem solving," 1984.

[41] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A* search meets graph theory," in *Proc. ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2005, pp. 156–165.

[42] T. Standley and R. Korf, "Complete algorithms for cooperative pathfinding problems," in *Proc. International Joint Conferences on Artificial Intelligence*, Barcelona, Spain, Jul. 2011, pp. 668–673.

[43] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015.