Journal Title
XX(X):1–13
©The Author(s) 0000
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Rubik Tables and Object Rearrangement

Mario Szegedy and Jingjin Yu[1]

Abstract

A great number of robotics applications demand the rearrangement of many mobile objects, e.g., organizing products on store shelves, shuffling containers at shipping ports, reconfiguring fleets of mobile robots, and so on. To boost the efficiency/throughput in systems designed for solving these rearrangement problems, it is essential to minimize the number of atomic operations that are involved, e.g., the pick-n-places of individual objects. However, this optimization task poses a rather difficult challenge due to the complex inter-dependency between the objects, especially when they are tightly packed together.

In this work, in tackling the aforementioned challenges, we have developed a novel algorithmic tool, called Rubik Tables, that provides a clean abstraction of object rearrangement problems as the proxy problem of shuffling items stored in a table or lattice. In its basic form, a Rubik Table is an $n \times n$ table containing n^2 items. We show that the reconfiguration of items in such a Rubik Table can be achieved using at most n column and n row shuffles in the partially labeled setting, where each column (resp., row) shuffle may arbitrarily permute the items stored in a column (resp., row) of the table. When items are fully distinguishable, additional n shuffles are needed. Rubik Tables allow many generalizations, e.g., adding an additional depth dimension or extending to higher dimensions.

Using Rubik Table results, we have designed a first constant-factor optimal algorithm for stack rearrangement problems where items are stored in stacks, accessible only from the top. We show that, for nd items stored in n stacks of depth d each, using one empty stack as the swap space, $O(nd)$ stack pop-push operations are sufficient for an arbitrary reconfiguration of the stacks where $d \le n^{\frac{m}{2}}$ for arbitrary fixed $m > 0$. Rubik Table results also allow the development of constant-factor optimal solutions for solving multi-robot motion planning problems under extreme robot density. These algorithms based on Rubik Table results run in low-polynomial time.

Keywords

Rubik Tables, Stack Rearrangement, Multi-Robot Motion Planning

1 Introduction

In a broad range of real-world applications, items are arranged in *stacks* to balance between efficient space usage and the ease of storage and retrieval (see Fig. 1). In a stack based storage solution, only the item on the top of an non-empty stack can be accessed instantaneously. If other stored items are to be retrieved, additional items must be moved beforehand. Such an approach, while preventing the direct random access of an arbitrary item, allows more economical utilization of the associated storage space, which is always limited. A prime example is the stacking of containers at shipping ports Borgman et al. (2010); Dayama et al. (2014), where stacks of containers may need to be rearranged (shuffled) for retrieval in a specific order, which contains a stack rearrangement component. Similar scenarios also appear frequently elsewhere, e.g., parking yards during busy hours in New York City, the re-ordering of misplaced grocery items on supermarket shelves Han et al. (2018a), the rearrangement of goods in warehouses Christofides and Colloff (1973), and so on. In all these application scenarios, the overall efficiency of the system critically depends on minimizing the number of item storage and retrieval operations. However, the tightly packed items, combined with the stack-based access, induce complex dependencies that make the rearrangement tasks a challenge to optimize.

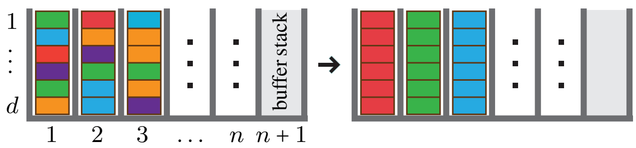


Figure 1. An illustration of the colored stack rearrangement problem with an initially empty buffer stack. From an initial arrangement of the items shown on the left, we would like to sort the items into the ordered target arrangement, by item color or type, on the right. Items may assume unique identifiers in the labeled version of the problem.

Similarly, in many large-scale multi-robot applications, e.g., warehouse automation Wurman et al. (2008); Li et al. (2020), it is desirable to operate as many robots as possible in a confined workspace: a small workspace requires less

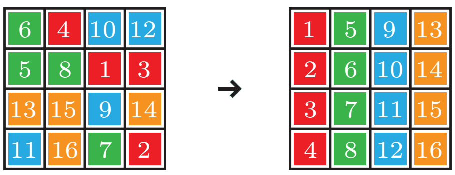
Department of Computer Science, Rutgers University

Corresponding author:

Jingjin Yu, 110 Frelinghuysen Road, Piscataway, NJ, 08854-8019 USA.
Email: jingjin.yu@cs.rutgers.edu

travel time for robots between tasks and more robots can execute more tasks simultaneously. At the same time, the efficient routing of a large number of robots in limited space again requires minimizing redundant movement of objects, in this case the robots. Such problems are well-known to be computationally intractable Yu and LaValle (2013); Solovey and Halperin (2016) due to the curse of dimensionality. As such, whereas tractable solutions have been proposed at least a few decades ago Kornhauser et al. (1984), polynomial time algorithms with good optimality guarantees have been illusive until recently, e.g., Yu (2018), due partly to this study.

Motivated by the array of practical scenarios, we have performed a systematic study of multi-object rearrangement problems where the objects are either movable items or robots that could move on their own. As we pay close attention to the structure of these apparently diverse set of robotics problems (e.g., rearranging items in stacks and multi-robot motion planning), we observe that, as the density of the objects become sufficiently high, they can be effectively viewed as a problem of *shuffling items stored in tables or lattices*. This leads us to the development of the *Rubik Table* abstraction (see Fig. 2). In a basic two-dimensional setting, a Rubik Table is an $n \times n$ table storing n^2 items. These items may either have n colors (or types) with each color containing n items, or they may all have unique labels. The *Rubik Table problem* asks for the sorting of the items by type or by label using the least number of column and row shuffles, where a column (resp., row) shuffle allows the arbitrary permutation of the items in a column (resp., row). Intuitively, solving the color-based Rubik Table sorting problem appears to require at least about $2n$ shuffles; it seems that each row and column should be worked on at least once. Surprisingly, the reconfiguration of the Rubik Table by color can be achieved using at most $2n$ column and row shuffles, through a careful application of a version of the Hall's marriage theorem Hall (2009). As a consequence, sorting by label is achievable using $3n$ shuffles.



(a) (b)

Figure 2. An instance of a 4×4 Rubik Table problem where we want to rearrange the items to go from the left configuration to the right configuration, sorted by color or by label.

The Rubik Table results, with the extensions, turn out to be fairly powerful in helping boost the asymptotic efficiency of algorithms for solving object rearrangement tasks. In the stack rearrangement problem (Fig. 1), nd items are stored in n stacks of capacity d each. With a buffer stack of depth d, we are to sort the items by type (n of these, d each) or by their unique labels using stack pop-push operations. We use CSR (resp., LSR) as shorthand for the colored (resp., labeled) stack rearrangement problem. For CSR and LSR, through simulating column and row shuffles using a series of specially crafted stack pop-push operations, it is possible to apply a version of the Rubik Table results to derive an algorithm that requires only $O(nd)$ number of pop-pushes when $d \le n^{\frac{m}{2}}$ for an arbitrary fixed $m > 0$. This suggests that the polynomial-time algorithm computes $O(1)$-optimal solutions for these settings. Previously, the best upper bound Han et al. (2018a) incurs an additional logarithmic factor.

In a similar vein, but to an apparently different problem, the Rubik Table theorems also apply to improving polynomial time multi-robot motion planning (MRMP) algorithms. In the context of the current work, *motion* refers to discrete time synchronous motion of many robots in a grid-like environment. To our knowledge, no polynomial time algorithms exist prior to Yu (2018); Demaine et al. (2019) that provide guaranteed solution makespan optimality for MRMP. Through simulating column/row shuffles using more rudimentary multi-robot motion primitives, a version of the Rubik Table result gives rise to an expected $O(1)$-optimal solution for arbitrary fixed dimensions, computed in low polynomial time Yu (2018). Using this expected $O(1)$-optimal solution, an $O(1)$-optimal solution can be constructed as well. In this study, we highlight the application of Rubik Table to MRMP and describe a new and improved expected $O(1)$-optimal algorithm for MRMP.

In the robotics domain, our study relates to multi-object rearrangement tasks, which may be carried out using mobile robots Ben-Shahar and Rivlin (1998); Garrett et al. (2015); Havur et al. (2014) or fixed robot arms Krontiris and Bekris (2015, 2016); Han et al. (2018b); Huang et al. (2019). Clearly a challenging task and motion planning (TAMP) problem in the general setting Huang et al. (2019), even the combinatorial aspect of object rearrangement is shown to be computationally hard in multiple problems in seemingly simple setups Han et al. (2018b). A multi-arm rearrangement problem is recently explored Shome et al. (2021). In a more abstract setting, multi-object rearrangement has also been studied under the PushPush line of problems Demaine and Hoffmann (2001); Demaine et al. (2000). More broadly, object rearrangement problems are connected to multi-robot motion planning problems Erdmann and Lozano-Perez (1987); Kornhauser et al. (1984); Solovey and Halperin (2016); Yu (2018); Demaine et al. (2019) and the problem of navigation among movable obstacles Wilfong (1991); Stilman and Kuffner (2008); Van Den Berg et al. (2009). Lastly, as a sorting problem, our study shares some similarities with sorting networks Ajtai et al. (1983); West (1993). Indeed, results on sorting networks can be applied to solve object rearrangement problems.

The stack rearrangement problem, which we study to some depth in this work, was first formally studied in the stated form in Han et al. (2018a). They established an $O(nd(\log n + \log d))$ algorithmic upper bound. Heuristics-based search methods are also developed that can compute the optimal solution for stack rearrangement problems involving tens of items. A closely related problem is the Hanoi tower problem Brousseau (1980); Szegedy (1999); Grigorchuk and Šunik (2006), which has additional constraints limiting the relative order of items in a stack during the rearrangement process.

The main algorithmic contributions of this work, beside building a novel structural connection between the abstract class of Rubik Table problems and challenging object rearrangement problems, are:

- For an $n \times n$ Rubik Table, sorting by the n types can be achieved using at most $2n$ column and/or row shuffles; if the items are all uniquely labeled, sorting can be done using at most $3n$ shuffles (Theorem 3.1). The algorithm for computing the required shuffle operations takes $O(n^2 \log n)$ expected time or $O(n^3)$ deterministic time. The Rubik Table results generalize to including a depth dimension (Theorem 3.2) and to higher dimensions (Theorem 3.3).
- For any fixed $m > 0$, LSR (and therefore, CSR) with $d \le n^{\frac{m}{2}}$ can be solved using $O(nd)$ pop-push operations. If m is an input parameter instead, LSR with $d = n^{\frac{m}{2}}$ can be solved using $O(3^m nd)$ pop-pushes (Theorem 4.1). Therefore, for an arbitrary fixed real number c, LSR may be solved using $O(nd)$ pop-pushes for $d \le \lceil cn \rceil$ (Theorem 4.2).
- The MRMP problem on an $m_1 \times m_2$ grid can be solved using a makespane of $O(m_1 + m_2)$ through a single application of the Rubik Table shuffle algorithm.

This paper builds on the conference publication Szegedy and Yu (2021). As we continue to develop the related research, this archive version adapts a new structure starting with the Rubik Table results and then develops their applications on stack rearrangement and multi-robot motion planning, which we believe is a more proper presentation of the research. In synchronization with the effort, we have developed several new results including a lower bound on the required number of shuffles needed for solving Rubik Table problems (Proposition 2.1), a new upper bound for the stack rearrangement problem with the constant in the big O notation given explicitly (Proposition 4.1 and Corollary 4.1), and a new expected $O(1)$ makespan-optimal algorithm for multi-robot motion planning (Proposition 5.1). We also expanded the discussion of open problems, which provides several concrete future directions for readers to explore.

The rest of the paper is organized as follows. In Sec. 2, Rubik Table problems, stack rearrangement problems, and a version of the multi-robot motion planning problem are formally defined; straightforward lower bounds for the Rubik Table problem and the stack rearrangement problem are also provided. In Sec. 3, we present results on several Rubik Table problems. In Sec. 4, refined upper bounds are established for CSR and LSR, demonstrating the utility of the Rubik Table abstraction. We further expose the application of Rubik Table to multi-robot motion planning in Sec. 5. We conclude in Sec. 6 with a brief discussion of many interesting open questions.

2 Preliminaries

2.1 Rubik Table Problems

The *Rubik Table problem* (see, e.g., Fig. 2) formalizes the task of carrying out globally coordinated token swapping operations in lattices. We associate it with the name *Rubik* as it shares some similarity with the Rubik's Cube toy. The basic setting deals with a planar table.

Problem 2.1. (Rubik Table problem). *Let M be an $n \times n$ table containing n^2 items, one in each table cell. The n^2 items are of n types with each type having a multiplicity of n. In a shuffle operation, the items in a single column or a single row of M may be permuted in an arbitrary manner. Given an arbitrary configuration X_I of the items, find a sequence of shuffles that take M from X_I to the configuration where column i, $1 \le i \le n$, contains only items of type i.*

Intuitively, $\Omega(n)$ shuffles are required for solving the Rubik Table problem, assuming that items are randomly distributed. We formally establish this lower bound here.

Proposition 2.1. (Lower bound for the Rubik Table problem). *A random Rubik Table problem instance requires at least $n-1$ shuffles to solve, in expectation.*

Proof. We may consider the probability that the content of cell (i, i) must participate in a shuffle (of either column i or row i). The probability of cell (i, i) containing items with type i is $\frac{1}{n}$, suggesting that the probability that the cell must be shuffled is $\frac{n-1}{n}$. The expected number of shuffles required to place an item of type i at cell (i, i) is then at least $\frac{n-1}{n}$. Summing over all i, $1 \le i \le n$, and by the linearity of expectation, yields that at least $n - 1$ shuffles are required to solve the Rubik Table problem, in expectation. □

One may further require that the n^2 items be uniquely labeled $1, \ldots, n^2$, and ask for an arbitrary reconfiguration of the items. This is equivalent to sorting the items by label, i.e., going from Fig. 2(a) to Fig. 2(b), ignoring the colors. We call this the *labeled Rubik Table problem*.

Problem 2.2. (Labeled Rubik Table problem). *Let the n^2 items in an $n \times n$ Rubik Table M have unique labels from $1, \ldots, n^2$. Given an arbitrary configuration X_I of the items, find a sequence of shuffles that takes M from X_I to the column-major sequential ordering of the n^2 items.*

We may allow a Rubik Table M to have a "depth" K. In each column (resp., row) shuffle of such a "fat" Rubik Table, we allow the arbitrary permutation of a "fat" column (resp., row) which contains nK items.

Problem 2.3. (Fat Rubik Table problem). *Let M be an $n \times n \times K$ (row × column × depth) table containing n^2K items, one in each cell of the table. These items are of n types, with each type having a multiplicity of nK. In a shuffle operation, the items in a single fat column (i.e., items with indices in $\{1, \ldots, n\} \times \{i\} \times \{1, \ldots, K\}$ for $1 \le i \le n$) or a single fat row (i.e., items with indices in $\{i\} \times \{1, \ldots, n\} \times \{1, \ldots, K\}$ for $1 \le i \le n$) of M may be permuted in an arbitrary manner. Given a configuration X_I of the items, find a sequence of shuffles that takes the table M from X_I to the configuration where each fat column i, $1 \le i \le n$, only contains items of type i.*

Similarly, a labeled version of the fat Rubik Table problem may be defined. We omit the straightforward definition.

The Rubik Table problem may be further generalized to arbitrary dimensions.

Problem 2.4. (Rubik R-D Table problem). *Let M be an $\underbrace{n \times \ldots \times n}_{R}$ table, $R \ge 2$, filled with n^R unique items. Assuming that any $(R-1)$-dimensional column can be arbitrarily shuffled, given two arbitrary configurations of the items, X_I and X_G, find a sequence of shuffles that takes M from X_I to X_G.*

2.2 Stack Rearrangement

In a *stack rearrangement* problem (see, e.g., Fig. 1), there are n stacks (i.e., LIFO queues), each filled to capacity with d items. In the *labeled* version, or LSR (labeled stack rearrangement), the items in the stacks are uniquely labeled $1, \ldots, nd$. Given an arbitrary initial arrangement of the items, we would like to rearrange them to follow lexicographic order, in which the k^{th} stack, $1 \le k \le n$, contains items labeled $(k-1)d+1$ to kd, with numbers decreasing (or increasing) monotonically from the top of the stack to the bottom of the stack. In a single *pop-push* stack operation, an item can be popped off from any non-empty stack and immediately pushed onto a stack which is not filled to its capacity d. To allow the rearrangement of items, we assume that there is an empty buffer stack with capacity d. During the moves the buffer can hold items but it must be emptied by the end. We seek to *minimize* the number of pop-pushes to take the stacks from an arbitrary initial arrangement to the specified target arrangement, which is equivalent to having an arbitrary goal arrangement.

In a *colored* version, or CSR (colored stack rearrangement), we still require that items labeled $(k-1)d+1, \ldots, kd$ go into the k^{th} stack but do not require these items take a specific order within the stack. This is equivalent to saying that we would like to sort nd items with n types of d each so that the k^{th} stack contains only items of type k.

It takes at least $\Omega(nd)$ pop-pushes to solve the stack rearrangement problem for a typical input instance, because most items must move at least once to get into place. Here, we prove a stronger lower bound. We mention that similar bounds are described in Han et al. (2018a). We provide a more accurate bound for LSR here with a proof counting the number of bits required to describe an algorithm. A bound for CSR is also included for completeness.

Lemma 2.1. (Lower bound for LSR). *An average LSR instance requires $\Omega(nd + nd\frac{\log d}{\log n})$ pop-pushes to resolve.*

Proof. The proof is by a counting argument. Any correct algorithm must follow different paths for all of the $(nd)!$ initial arrangements, since two different initial arrangements followed by identical moves would lead to different final arrangements. A step (corresponding to one pop-push operation) of the algorithm can be described with $2\lceil\log(n+1)\rceil$ bits: (from where, to where). Therefore, the two-based logarithm of the number of possible sequences of at most t steps is upper bounded by $2t\lceil\log(n+1)\rceil$. So as long as it holds that

$$2t\lceil\log(n+1)\rceil \le \log\left(0.01 \cdot (nd)!\right) = \Omega(nd\log nd),$$

i.e. when $t = o(nd + nd\frac{\log d}{logn})$, the initial arrangements that can be solved with t steps constitute only a small minority of all arrangements. The counter-positive of this gives the lemma. □

Lemma 2.2. (Lower bound for CSR). *Any algorithm for CSR must take at least $\Omega(nd)$ pop-pushes for an average input.*

Proof. Me may view the generation of a random instance as selecting from n types of items with replacement d for up to nd rounds. Therefore, there are $(\Theta(n))^{\Theta(nd)}$ initial configurations. Following the same argument from the proof of Lemma 2.1, $\Omega(nd)$ pop-pushes are necessary. □

2.3 Multi-Robot Motion Planning

For multi-robot motion planning, we focus on a grid-based setting. Let m_i, $1 \le i \le R$ be the span of the i^{th} dimension of an R-dimensional grid. That is, we work with an $m_1 \times \ldots \times m_R$ grid. There are $\prod_i m_i$ robots labeled $1, \ldots, \prod_i m_i$, each occupying a vertex of the grid. As to the motion primitive, in each time step, robots on any set of mutually disjoint, non-self-intersecting cycles of the grid may synchronously move across one edge on these cycles in the same direction, i.e., robots may rotate along these cycles (see e.g., Fig. 3).

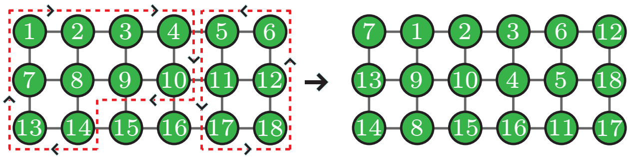


Figure 3. Illustration of a single time step move of multiple robots along two disjoint cycles of a 3×6 grid.

The makespan-optimal multi-robot motion planning (MRMP) problem seeks a routing plan that takes the robots between two arbitrary configurations X_I and X_G with the minimum makespan. This is an NP-hard problem Yu (2015); Banfi et al. (2017). In this study, we will illustrate how Rubik Table facilitates solving MRMP to $O(1)$ makespan-optimal.

3 Algorithms for Rubik Table Problems

Given that no less than $n-1$ shuffles are required for solving the Rubik Table problem in expectation, it is somewhat surprising that only $2n$ shuffles can get the job done.

Theorem 3.1. (Linear shuffle algorithm for Rubik Table and labeled Rubik Table problems). *A Rubik Table problem is solvable using n column shuffles followed by n row shuffles. Additional n column shuffles then solve the labeled Rubik Table problem.*

We call the result a linear shuffle algorithm because each item may be moved a (small) constant number of times, even though the number of column and row shuffles is the square root of the number of items which is sub-linear. Before presenting the proof of Theorem 3.1, we introduce a Kőnig-Hall type matching theorem Hall (2009) with parallel edges.

Lemma 3.1. (Hall's Matching theorem with parallel edges). *Let B be a d-regular ($d > 0$) bipartite graph on $n+n$ nodes, possibly with parallel edges. Then B has a perfect matching.*

Proof. Let the vertex set of B be $L \uplus R$, where L is the left partite set of B and R is the right partite set; the $\uplus$ symbol denotes the disjoint union of two sets. Consider a maximal matching M in B. We show that M meets all of the vertices of B, so it is perfect. Assume that M is not incident to some vertex $v \in L$. Consider all nodes of B reachable by an alternating path from v, that is, a path that starts in v, goes to R along some edge, then goes back to L along an edge of M (if such an edge exists), then goes along an arbitrary edge to R, an so on, always alternating between edges of

M and non-edges of M. We stop whenever we want. If any such path P ends up in a point w of R not matched in M, we could make M bigger as follows: We discard from M its intersection with P and add $P \backslash M$ to it, creating

$$M' = M \triangle P = (M \backslash (M \cap P)) \cup (P \backslash M).$$

It is easy to see that M' is a matching and $|M'| = |M| + 1$, contradicting the maximality of M. Otherwise, let $L' \subseteq L$ be the subset of L reachable via an alternating path from v (which includes v too), and let $R' \subseteq R$ be the set of nodes reachable with alternating path from v. Then $|L'| > |R'|$, since every vertex in R' has a matching partner in L' through M, and in addition L' contains v, which is not a partner of any node in R'. Furthermore, all neighbors of the nodes in L' must be in R', otherwise we could find a neighbor w of a node t in L', which is reachable via an alternating path from v (formed by adding edge (t, w) to the alternating path from v to t), but unmatched in M. Since the nodes in L' have a total of $d|L'|$ edges incident to them (counted with multiplicities), which is more than the number $d|R'|$ of edges incident to R' (counted with multiplicities), we again have a contradiction. □

Proof of Theorem 3.1. The $n + n + n$ shuffles to construct an arbitrary permutation π for solving the labeled Rubik Table problem are outlined in Table 1.

1.	Preparation:	By appropriately permuting the items within each column we reach the situation where the n items destined to go to any fixed column will end up in n different rows.
2.	Column fitting:	By appropriately permuting the items within each row we reach the situation where the n items destined to go to any fixed column goes to that column.
3.	Row fitting:	By appropriately permuting the items within each column we move each item into its final destination.

Table 1. A three-phase shuffle plan for rearranging labeled items in an $n \times n$ Rubik Table. The first $n + n$ shuffles solve the type/color based Rubik Table problem.

The preparation phase is necessary for the column fitting phase. We need to prove that we can permute the items only within every column (i.e. such that no item changes its column coordinate) with the effect that the n items destined to go to any fixed column end up in n different rows. This comes from Lemma 3.2, which shows the feasibility of the preparation phase and therefore, the entire algorithm. □

Lemma 3.2. *Let M be an $n \times n$ table filled with items of n different types. The number of items of type i is exactly n for $1 \le i \le n$. Then we can permute the items within each column of M separately such that in the resulting new arrangement all of the n items of any fixed type i (for $1 \le i \le n$) go into separate rows.*

Proof. We begin by creating a bipartite graph $B(T, C)$ on $n + n$ nodes such that the left partite set, T, stands for all the types $\{1, \ldots n\}$, and the right partite set, C, stands for all the columns of M (also, see Fig. 4). We draw k edges between type j and column i, if column i contains k items of type j. Notice that B is n-regular from both sides with parallel edges. Lemma 3.1 implies that graph B contains a perfect matching M_1. Label the edges of this matching with the number 1, and take it out of B. We obtain an $(n - 1)$-regular bipartite graph on which Lemma 3.1 may be applied again. We keep creating matchings M_2, M_3, …, in this fashion and label their edges with 2, 3, …, until we arrive at M_n, when we stop. Notice that now each type $j \in T$ is connected to edges labeled with 1 through n, and that each column C_i is connected to all n types of edges as well (in both cases exactly one from each type). For every $1 \le i \le n$ we rearrange the items in column C_i such that the item corresponding to an edge labeled with i goes into the i^{th} row. There will be no collisions by construction and we have arrived at the desired arrangement. □

From the algorithmic perspective, each matching step in Lemma 3.2 can be computed in expected $n \log n$ time Goel et al. (2013). Alternatively, if a deterministic algorithm is desirable, a matching can be computed in $O(n^2)$ time Cole et al. (2001). The n matchings can then be completed in $O(n^2 \log n)$ expected time or $O(n^3)$ deterministic time. This is the dominant part of the algorithm's time complexity.

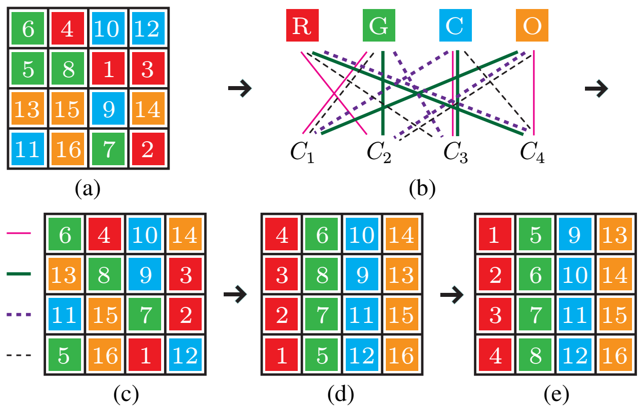


Figure 4. Illustration of applying the $n + n + n$ shuffles as described in the proof of Theorem 3.1. (a) The initial 4×4 table with a random arrangement of 16 items that are also colored (red, green, cyan, orange). (b) The bipartite graph constructed from the table and a possible set of 4 perfect matchings, where $C_i, 1 \le i \le 4$ are the columns. As an example, green appears twice in the first column of the table in (a) so there are two edges between green and C_1. Each matching is marked with a unique line type (thin, thick, thick dashed, thin dashed). (c) Permuting each column according to the matching results in each row containing each item type exactly once. (d) Permuting each row of (c) then sorts all columns. (e) A final set of n column shuffles fully sorts the items.

To provide some intuition of the shuffle algorithm, Fig. 4 illustrates an application of the procedure used in proving Theorem 3.1 on a 4×4 Rubik Table containing 16 items that also fall into four colors (types). Fig. 4(b) shows the constructed 4-regular bipartite graph based on the configuration from Fig. 4(a). Four matchings are shown in different line patterns (thin solid, thick solid, thick dashed,

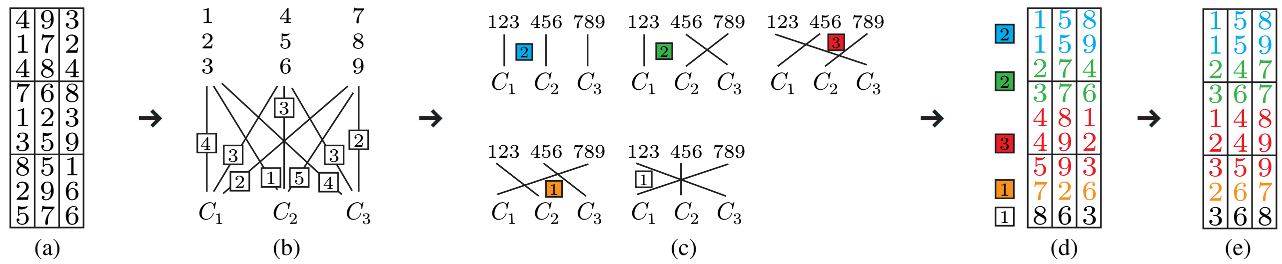


Figure 5. Illustration of applying the first two shuffle phases from the proof for Theorem 3.2. (a) An initial $3 \times 3 \times 3$ fat Rubik Table with a random arrangement of 9 types of items. (b) The (weighted) regular bipartite graph from the setup in (a). The numbers on edges denote the weight/multiplicity of the edges. (c) The 5 sets of (weighted) perfect matching extracted from (b). (d) The fat-column permutations based on the matchings. Notice that the numbers in the columns remain the same between (a) and (d). (e) The following fat-row permutations which correctly sort the columns. With one more round of fat-column permutations, we can sort the table so that each cell contains a single item type. Additional distinguishability within a cell is allowed as well.

and thin dashed) and colors (pink, green, purple, and black). Based on the matchings, an intermediate Rubik Table is constructed as given in Fig. 4(c), where each column is shuffled as compared with Fig. 4(a). Note that each row now contains each type exactly once. After n row shuffles, sorting by type is achieved as shown in Fig. 4(d). With one more round of n column shuffles, Fig. 4(d) can be sorted to reach the fully sorted column-major configuration in Fig. 4(e).

It can be readily verified that Theorem 3.1 can be generalized to tables that are not squares.

Corollary 3.1. (Linear shuffle algorithm for the Rubik Rectangle problem). *Let M be an $n \times m$ table filled with nm unique items, $n \leq m$. In n row shuffles and $2m$ column shuffles, items in M can be sorted arbitrarily.*

The matching-based shuffle routine is quite flexible, allowing many extensions to the Rubik Table problem. A particular useful extension is one for the fat Rubik Table problem as stated in Problem 2.3 and its labeled version, which adds a *depth* to each table cell. With some relatively minor modifications, $2n$ or $3n$ shuffles are again sufficient.

Theorem 3.2. (Linear shuffle algorithm for fat Rubik Table problems). *The fat Rubik Table problem and the labeled fat Rubik Table problem may be solved using $2n$ shuffles and $3n$ shuffles, respectively.*

Proof. The proof of Theorem 3.1 can be adapted with relatively minor changes. A similar three-phase procedure will be followed; again, the crucial part is the proof of the preparation phase, in which we show that we can permute the items within each fat column to reach the situation where the nK items destined to go to any fixed fat column will end up in nK positions, that are different when we project them to the first and third coordinates. The needed procedure for doing this is provided in Lemma 3.3. □

Lemma 3.3. *Let M be an $n \times n \times K$ table (row × column × depth) filled with items of n different types. The number of items of type j is exactly nK for $1 \leq j \leq n$. Then we can permute the items within each fat column $(*, i, *)$ of M $(1 \leq i \leq n)$ such that for any fixed type j $(1 \leq j \leq n)$, if we look at the nK items of type j, they occupy distinct (row, depth) values when we project the triplet representing their new positions to the pair of row and depth coordinates.*

Proof. The proof of the lemma is again based on applying Lemma 3.1 on an $n + n$ bipartite graph. The nodes on the left are n different types and the nodes on the right represent the fat columns. The edges correspond to the items, and we have K parallel edges between right node i and left node j as long as K items need to go from fat column i to fat column j. The only difference is that now the graph is nK-regular rather than n-regular. Again, we can iteratively decompose the edge-set of this bipartite graph into nK perfect matchings, which gives the solution we are looking for. □

In terms of running time, the fat Rubik Table problems introduces an additional multiplicative factor K, yielding $O(n^2 K \log n)$ expected or $O(n^3 K)$ deterministic time.

Again, to provide some intuition of the somewhat non-intuitive procedure, Fig. 5 illustrates an application of Theorem 3.2 to derive the first two sets of permutations for restoring order to a $3 \times 3 \times 3$ fat Rubik Table. In applying Lemma 3.3, type j corresponds to items numbered $(j-1) * 3 + 1$ to $j * 3$. For example, all items numbered $1 - 3$ are treated as type 1.

As for the Rubik Table problem, we mention that non-square fat Rubik Tables can be supported and leave it to the readers to fill in the details.

Lastly, we examine a high-dimensional version of the Rubik Table problem. A fat version adding additional depth is again possible, which we do not further detail in this exposition.

Theorem 3.3. (Rubik R-D Table). *Let M be an $\underbrace{n \times \ldots \times n}_{R}$ table, $R \geq 2$, filled with n^R unique items. Assuming that any $(R-1)$-dimensional column can be arbitrarily shuffled, then M can be arbitrarily sorted in $(2^R - 1)n^{R-1}$ shuffles.*

Proof. Let $F(n, R)$ be the number of shuffles for given n and R. We prove the claimed bound on $F(n, R)$ by induction on R. We can do the 2 dimensional case in $3n$ shuffles by Theorem 3.1. For $R > 2$, select out the first two dimensions and treat the remaining $R - 2$ dimensions as the depth of a fat Rubik Table. By the induction hypothesis we can permute any fat column of M any way we want in $F(n, R-1)$ shuffles (by the induction hypothesis, $F(n, R-1) = (2^{R-1} - 1)n^{R-2}$). In the preparation phase we must do n of these. Then we do n^{R-1} row operations and finally we do again permutations on the fat columns, which cost $nF(n, R-1)$.

Altogether, we have

$$\begin{aligned} F(n,R) &= 2nF(n,R-1) + n^{R-1} \\ &= 2n(2^{R-1}-1)n^{R-2} + n^{R-1} \\ &= (2^R-1)n^{R-1}. \end{aligned}$$

□

4 Application to Stack Rearrangement

In Han et al. (2018a), an $O(nd(\log n + \log d))$ upper bound is established for LSR. Their method is sequential: it first sorts items into the correct stacks and then sort each stack individually. In Sec. 4.1, we start with describing a more efficient approach that improves the bound for $n \geq d$ to $O(nd \log d)$ with the constant in the big $O(\cdot)$ notation given explicitly. This is achieved interleaving inter- and intra-stack sorting. This result also uses ideas that enable the Rubik Table results but in a more direct manner. As such, the proof techniques may be of independent interest. Then, in Sec. 4.2 and Sec. 4.3, we show how the Rubik Table results allow us to further lower the upper bound to $O(nd)$ for $d \leq n^{\frac{m}{2}}$ with $m > 0$ being an arbitrary constant. We briefly discuss the cases of having constant n or d in Sec. 4.4.

4.1 A $7nd(1 + \log d)$ Upper Bound for $n \geq d$

As stated, this subsection seeks to establish the following.

Proposition 4.1. (A $7nd(1 + \log d)$ upper bound for CSR with $n \geq d$). *For arbitrary $n \geq d$ where $d = 2^{\lceil \log d \rceil}$, a CSR instance can be solved using $7nd(1 + \log d)$ pop-pushes.*

For simplicity, assume that d is a two-power, i.e., $d = 2^{\lceil \log d \rceil}$. If d is not a two-power, we lose at most a factor of 2 in terms of efficiency. The algorithm is recursive, and we recurse on $\log d$. When $d = 1$, there is a simple solution that uses less than $\frac{3}{2}n$ pop-pushes. Let $F(n, d)$ be the number of pop-push operations used by our algorithm, we will prove the following recurrence:

$$F(n,d) = 2F(n, \frac{d}{2}) + 7nd.$$

This will clearly give the claimed efficiency because $7nd(1 + \log d) = 7nd + 2(7n\frac{d}{2})(1 + \log \frac{d}{2})$. We denote the content of a stack S_i with $S_i[1], \ldots, S_i[d]$, where $S_i[d]$ refers to the top of stack S_i. The buffer stack is denoted as $S_\varnothing$. The recurrence will be proved using two intermediate lemmas. We will need the notion of a balanced arrangement.

Definition 4.1. (Balanced arrangement). An arrangement, where each stack $S_i, 1 \leq i \leq n$, holds d items, is *balanced* if for all types $j \in \{1, \ldots, n\}$, the number of items of type j elements held in $S_1[1..\frac{d}{2}] \cup \ldots \cup S_n[1..\frac{d}{2}]$ is exactly $\frac{d}{2}$.

Lemma 4.1. *Assume we have an arbitrary arrangement of the items in the stacks $S_1, \ldots, S_n$, where each S_i holds exactly d items. Then, we can permute the contents of each stack individually such that the arrangement becomes balanced.*

Proof. Create a bipartite graph $B(T, S)$ on $n + n$ nodes such that the left side, T, stands for all the types $\{1, \ldots, n\}$, and the right side, S, stands for all the stacks $S_1, \ldots, S_n$. We draw k edges between type j and stack i if stack i contains k elements of type j. Notice that B is d-regular from both sides. Hall's theorem implies that graph B contains a perfect matchings M_1. We remove M_1 and repeat the process to get $\frac{d}{2}$ matchings $M_1, \ldots, M_{\frac{d}{2}}$. Color these matchings blue. Notice that now in B, each type T_j is connected to exactly $\frac{d}{2}$ blue edges, and that each stack S_i is connected to $\frac{d}{2}$ blue edges as well. For every $1 \leq i \leq n$ rearrange the items in stack S_i such that the items corresponding to the blue edges occupy $S_i[1..\frac{d}{2}]$. We have arrived at a balanced arrangement. □

Next, we show how a balanced arrangement can be realized efficiently.

Lemma 4.2. *For $n \geq d$, starting from an arbitrary arrangement, we can achieve a balanced arrangement using at most $4nd$ pop-pushes.*

Proof. It is sufficient to show that any permutation of stack i can be implemented in $4d$ pop-pushes:

- Select $d - 1$ stacks other than S_i, and put their top elements (temporarily) into $S_\varnothing$. At the end of this step, $S_\varnothing$ holds $d - 1$ items and have one more space available.
- Put the elements of S_i into the thus freed tops of the stacks, including one on the top of $S_\varnothing$.
- Put the elements back to S_i in the desired order.
- Restore the tops of the stacks from $S_\varnothing$.

Each of the phases above takes d pop-pushes to execute, altogether $4d$ pop-pushes. □

With Lemma 4.2, we are ready to outline the algorithm that backs up Proposition 4.1, given in Alg. 1.

Algorithm 1: SR(n, d)

1 Rearrange the stacks to achieve a balanced partition using Lemma 4.2.
2 Forget about the bottom $\frac{d}{2}$ elements of each stack and solve the problem on the top with $SR(n, \frac{d}{2})$.
3 Swap the top $\frac{d}{2}$ elements with the bottom $\frac{d}{2}$ elements in each S_i.
4 Forget about the bottom $\frac{d}{2}$ elements of each stack and solve the problem on the top with $SR(n, \frac{d}{2})$.

It is clear that the algorithm returns the correct solution. The number of pop-pushes used, line by line, is:

$$\begin{aligned} F(n,d) &= \underbrace{4nd}_{line\ 1} + \underbrace{F(n, \frac{d}{2})}_{line\ 2} + \underbrace{3nd}_{line\ 3} + \underbrace{F(n, \frac{d}{2})}_{line\ 4} \\ &= 2F(n, \frac{d}{2}) + 7nd. \end{aligned}$$

Because each stack can be sorted by label in $d \log d$ pop-pushes (see Lemma IV.3. in Han et al. (2018a)), adding another $nd \log d$ pop-pushes can solve LSR.

Corollary 4.1. (A $7nd + 8nd \log d$ upper bound for LSR with $n \geq d$). *For arbitrary $n \geq d$, an LSR instance can be solved using $7nd + 8nd \log d$ pop-pushes.*

4.2 Linear pop-push algorithm for LSR, $d \leq \sqrt{n}$

Results on fat Rubik Table problem lead to significantly improved upper bounds for CSR and LSR that largely match the lower bound (asymptotically), which we establish in this section. The proposed algorithmic approach applies directly to LSR and therefore CSR. The improved upper bounds are obtained through recursive applications of the fat Rubik Table result (Theorem 3.2) through "simulated" fat Rubik Table column and row permutations. The recursion is done based on increasing $2\frac{\log d}{\log n}$. We first address the case of $2\frac{\log d}{\log n} \leq 1$ (i.e., $d \leq \sqrt{n}$), followed by the case $2\frac{\log d}{\log n} \leq 2$ (i.e., $d \leq n$), and finally the general case of $2\frac{\log d}{\log n} \leq m$ (i.e., $d \leq n^{\frac{m}{2}}$).

Lemma 4.3. (Linear pop-push algorithm for LSR, $d = \sqrt{n}$). *LSR with $d = \sqrt{n}$ can be solved using $O(nd)$ pop-pushes.*

Proof. We construct an $n' \times n' \times K$ fat Rubik Table with $n' = K = d = \sqrt{n}$. A depth $K = d$ fat cell of the table with index $(i, j), 1 \leq i, j \leq n' = d$ is identified with the stack indexed $(j - 1) * d + i$ (see Fig. 6 for an example), which ranges between 1 and $n = d^2$.

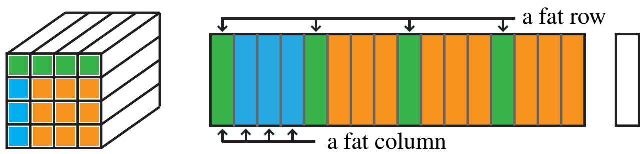


Figure 6. Correspondence between a $d \times d \times d$ fat Rubik Table and the $n = d^2$ stacks of depth d in a stack rearrangement problem instance. Here, $d = 4$.

We first show that we can simulate a single fat column permutation of $n'K = \sqrt{n}d = d^2$ items in $O(d^2)$ pop-pushes, which can be achieved by:

1. Moving the content of $\sqrt{n} = d$ stacks to the top of the n stacks using $O(d^2)$ pop-pushes. For each stack, we may move its content to the top of other stacks using the operations illustrated in the first four figures in Fig. 7, which takes $3d$ pop-pushes. Applying this to d stacks requires $3d^2$ pop-pushes, resulting the configuration shown in the fifth figure (bottom left) of Fig. 7.

2. Sort the d^2 elements on top of the stacks arbitrarily, which takes $O(d^2)$ pop-pushes. This requires using the buffer stack to hold at most one item temporarily. This happens in the fifth figure of Fig. 7.

3. Revert the first step above to return the sorted d^2 items to the d stacks of current interest. This corresponds to going from the fifth figure to the last figure in Fig. 7.

Following the same procedure, a fat row permutation can also be carried out in $O(d^2)$ pop-pushes. To apply Theorem 3.2, we partition all $nd = d^3$ items into d types

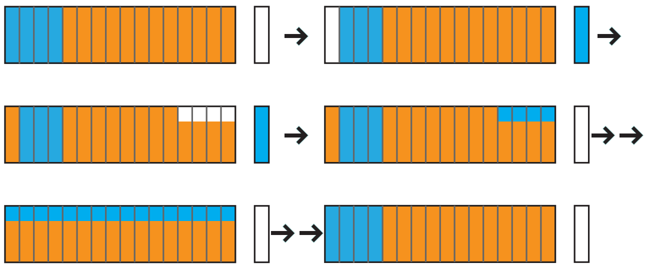


Figure 7. Illustration of the steps for realizing a simulated fat column permutation in $O(n'K) = O(d^2)$ pop-pushes. The cyan stacks (left four stacks in the first figure) are the d stacks of interest. The first step (indicated by the arrow) illustrates emptying the leftmost stack to the buffer. Then, the top of some stacks not of current interest (the orange ones) can be moved to the emptied stack (second step). Subsequently, the buffer content can be put on the top of stacks (third step). After this is done for all stacks of current interest, the contents of these stacks are moved to the top of the d^2 stacks (fourth step, marked with double arrows "→→"). After rearranging these items as needed, they can then be returned (fifth step, marked with double arrows "→→"). The "simulated" fat-column shuffle mirrors the step of permuting the first column of Fig. 5(a) to the first column of Fig. 5(d).

where items of type t, $1 \leq t \leq d$, have destinations in stack $(t-1)d + 1$ to stack td. By Theorem 3.2, using d fat column permutations and d fat row permutations, all items of type $t, 1 \leq t \leq d$ can be moved to fat column t. Then, applying a fat column permutation to a fat column t can sort items in the fat column arbitrarily. This solves the LSR problem (and therefore, a CSR problem).

Tallying the number of pop-pushes, we have done $3d$ fat column/row permutations, each of which takes $O(d^2)$ stack pop-pushes. The total is then $O(d^3) = O(nd)$ (with more careful counting, we can conclude that the number of pop-pushes is bounded by $27nd$). □

It is straightforward to see that Lemma 4.3 readily generalizes to $d < \sqrt{n}$. If n is a square, then the corollary directly applies. For n that is not a square, e.g., $n = q^2 + p$ where q^2 is the largest square less than n, we can partition the n stacks into two groups of q^2 stacks each with $q^2 - p$ of the stacks overlapping between the two groups (we can assume that n is sufficiently large so that $q^2 - p > p$; otherwise n can be treated as a constant). Focusing on the first group of q^2 stacks, we can then apply Lemma 4.3 (note that m satisfies $\sqrt{n} > q > \lceil\sqrt{n}\rceil - 1 \geq d$) to "concentrate" items that should go to the rest p stacks in the $q^2 - p$ stacks shared between the two groups. Then, Lemma 4.3 can be applied again to the second group of q^2 stacks in a similar fashion, followed by one last application to the first group of q^2 stacks, which solves the entire problem. We have proved

Corollary 4.2. (Linear pop-push algorithm for LSR, $d \leq \sqrt{n}$). *LSR with $d \leq \sqrt{n}$ can be solved using $O(nd)$ pop-pushes.*

Another consequence of Theorem 3.2 is that, if we allow $b = \lceil\sqrt{n}\rceil$ empty buffer stacks (instead of a single buffer stack) of depth d each, CSR with arbitrary n and d can be solved using $O(nd)$ pop-pushes. This is true because a constrained (items are distinguishable by types but do not have individual labels) fat column permutation can be readily executed in $2\sqrt{n}d$ pop-pushes using $\lceil\sqrt{n}\rceil$ buffer stacks.

Corollary 4.3. (Linear pop-push algorithm for CSR with extra buffers). *Given $b = \lceil\sqrt{n}\rceil$ buffer stacks, CSR with arbitrary but sufficiently large n and d can be solved using $O(nd)$ pop-pushes.*

If n is a perfect square, then the number of required pop-pushes is bounded by $6nd$. It is not clear that having $\lceil\sqrt{n}\rceil$ buffers help with solving LSR in $O(nd)$ time for arbitrary n and d; we leave this as an open question.

4.3 Linear pop-push algorithm for LSR, $d < n^{\frac{m}{2}}$ and Constant $m > 0$

We continue to look at the case where $2\frac{\log d}{\log n} > 1$, starting with $n = d = k^2$ for some integer k. The algorithm for doing so will invoke Lemma 4.3 repeatedly, which uses the top k rows of the stacks.

Lemma 4.4. (Linear pop-push algorithm for LSR, $d = n$). *For $n = d = k^2$, LSR can be solved in $O(nd)$ pop-pushes.*

Proof. Similar to how Lemma 4.3 is proven, we will simulate column and row permutations on a fat Rubik Table mapped to the stack rearrangement instance. To do the mapping, we simply identify stacks $(i-1)k+1, \ldots ik$ with the i^{th} fat column of the fat Rubik Table. The $j, j+k, j+2k, \ldots, j+(d-1)k$ stacks are identified with the j^{th} fat row. It is clear that, if we can simulate fat column/row permutations using $O(k^3)$ pop-pushes, then the statement of the lemma holds.

To simulate a fat column/row permutation, we note that the content of any k stacks can be flipped with the contents of the top k rows of the k^2 stacks, using the buffer stack. This takes $O(k^3)$ pop-pushes and is illustrated in Fig. 8(a) → Fig. 8(e), which is similar to the procedure illustrated in Fig. 7 (if we "compress" k consecutive items in a stack into a single item). Once the contents of the selected k stacks (corresponding to a fat column/row) occupy the top k rows of the k^2 stacks, Lemma 4.3 may be applied to rearrange the items in them arbitrarily, which takes $O(k^3)$ time as well. A reversal of the first step then completes a simulated fat column/row permutation. The total number of pop-push operations used is $O(k^3)$. □

It is clear that Lemma 4.4 continues to apply when $\sqrt{n} < d < n$, following the same argument used for establishing Corollary 4.3. That is,

Proposition 4.2. (Linear pop-push algorithm for LSR, $d \le n$). *LSR with $d \le n$ can be solved using $O(nd)$ pop-pushes.*

The condition $d = n$ in Lemma 4.4 may be viewed as $\frac{\log d}{\log n} = 1$ or $d = n^{\frac{m}{2}}$ with $m = 2$. Taking a closer look at the proof for Lemma 4.4, it is straightforward to see that the same argument directly extends to show that the LSR case of $d = k^3$ and $n = k^2$ ($\frac{\log d}{\log n} = \frac{3}{2}$) can be solved using $O(nd)$ pop-pushes for any positive integer k. In proving Lemma 4.4, the top k rows of the stacks are used as a *swap space* for applying Lemma 4.3, simulating a fat column/row permutation. In a similar fashion, for $d = k^3$ and $n = k^2$, the

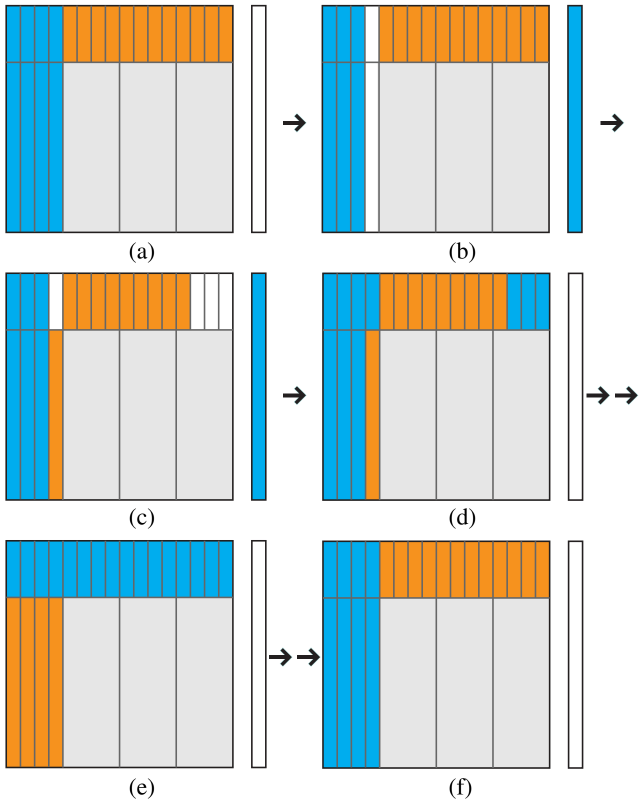


Figure 8. Illustration of a simulated fat column permutation over n stacks of depth d, with $n = d = k^2$. (a) The cyan colored stacks (left four in the first figure) map to a fat column of a fat Rubik Table. (b)-(e) In $O(k^3)$ stack pop-pushes, a fat column (or a fat row) can be moved to the top k rows of the n stacks, over which Lemma 4.3 can be applied to perform arbitrary sorting in $O(k^3)$ pop-pushes. (f) After rearrangement, the fat column (or fat row) can be restored using the same procedure in $O(k^3)$ pop-pushes, completing the shuffle operation.

top k^2 rows can be used as the swap space, which allows us to work with a total of $k^2 \cdot k^2 = k^4$ items. Once the swap space is properly set up, the k^4 items can be rearranged arbitrarily by Lemma 4.4 using $O(k^4)$ pop-pushes. So LSR with $d = n^{\frac{m}{2}}$ for $m = 3$ can be solved in $O(nd)$ pop-pushes. Corollary 4.3 then generalizes to apply to all cases where $\frac{\log d}{\log n} \le \frac{3}{2}$.

Recursively, Lemma 4.4 may be generalized to arbitrary $m \ge 2$. For $m = 3$, the procedure will call the $m = 2$ case $3k$ times. If the $n = d$ case requires $cnd = ck^4$ pop-pushes for some constant c, then the $m = 3$ case will need $3ck^5$ pop-pushes. Recursively, for general m, the recursive procedure will require about $3^m cnd$ pop-pushes for $d = n^{\frac{m}{2}}$. We have proved

Theorem 4.1. (Algorithm for LSR with $d \le n^{\frac{m}{2}}$ and $m \ge 0$). *LSR with $d \le n^{\frac{m}{2}}$ for $m \ge 0$ can be solved using $O(3^m nd)$ pop-pushes.*

For any fixed $m \ge 2$, it is clear that LSR can be solved in $O(nd)$ pop-pushes for $n^{\frac{m-1}{2}} < d < n^{\frac{m}{2}}$, possibly with a larger constant than the $d = n^{\frac{m}{2}}$ case. For fixed m, 3^m is also a constant. Summarizing the results on the upper bounds obtained so far, we have

Theorem 4.2. (Linear pop-push algorithm for LSR, $d \leq \lceil cn \rceil$). *For arbitrary fixed real number $c > 0$, LSR with $d \leq \lceil cn \rceil$ can be solved using $O(nd)$ pop-pushes.*

For CSR with $d = n = k^2$, with additional care in carrying out the recursive procedure, we only need to make $2k$ calls to Lemma 4.3 instead of $3k$ as required in proving Lemma 4.4. This gives us that CSR with $d \leq n^{\frac{m}{2}}$ for $m \geq 2$ can be solved using $O(2^m nd)$ pop-pushes instead of the $O(3^m nd)$ stated in Theorem 4.1. We omit the very involved procedure, which boils down to doing a mixed column and row permutation.

4.4 Constant n or d

Lastly, we briefly discuss what happens when n or d is a constant. An $O(nd \log n)$ algorithm for CSR is provided in Han et al. (2018a) for arbitrary n and d, using divide and conquer over the number of stacks n. This implies that for constant n, $O(d)$ pop-pushes is sufficient, matching the $\Omega(nd)$ lower bound. For constant d, each stack can be sorted in $O(1)$ pop-pushes by first moving all type k items to the top of the stacks they are at (for a stack i that contains type k item, this can be done by first moving the top item from some d stacks to the buffer, moving items in stack i to the empty d top spots, and then moving them back to stack i so that type k items stay on the top). Then type k items can be all moved to the buffer stack and followed by emptying stack k, then to stack k. This yields an $O(n)$-pop-push algorithm, also matching the lower bound.

5 Application to Optimal Multi-Robot Motion Planning

The built-in global coordination perspective of Rubik Table problems naturally extends them to applications toward optimal multi-robot motion planning (MRMP) as well. In Yu (2018), an $O(1)$ makespan-optimal algorithm, *partition and flow* (PAF), is proposed for solving makespan-optimal MRMP for the grid setting illustrated in Fig. 9. PAF in turn utilizes an expected $O(1)$-optimal algorithm, *split and group* (SAG), that recursively splits the problem into smaller pieces. SAG uses (non-labeled) Rubik Table results. In Sec. 5.1, we summarize how Rubik Table results are applied to enable SAG as is done in Yu (2018). Then, in Sec. 5.2, we describe a new and more direct expected $O(1)$-optimal algorithm as an application of the labeled Rubik Table result. We mention that, whereas our illustration will focus on the 2D setting, the methods generalize to higher dimensions. Because the generalization is fairly straightforward, we do not further elaborate here.

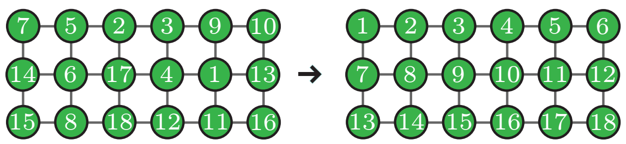


Figure 9. An example MRMP instance on a 3×6 grid where 18 robots in an arbitrary configuration on the left must be routed to take the row-major ordering on the right, which is equivalent to moving between two arbitrary configurations.

5.1 Recursive Split and Group

For a random MRMP instance (e.g., Fig. 9), the expected makespan is equal to the sum of the grid's dimensions. For 2D, this is $O(m_1 + m_2)$ (see Sec. 2.3 for the definition of MRMP on grids). To compute a routing solution with expected $O(1)$-optimal makespan, the total number of steps must be bounded by $O(m_1 + m_2)$. To achieve this, the *split and group* or SAG algorithm from Yu (2018) *splits* the $m_1 \times m_2$ grid along a longer dimension into two equal (or roughly equal) sized grids, and then route the robots so that they move to the half that they belong to in the goal configuration. This induces a *grouping* problem where each robot must either stay in the current half grid it is in or move to the other half grid. For the instance given in Fig. 9, the grouping problem is given in Fig. 10.

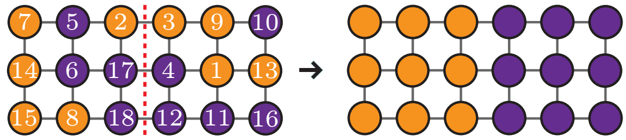


Figure 10. Grid splitting induces a grouping problem where robots must be routed to one of the two half grids. Here, the labels do not matter in the goal configuration; only the group (color) matters. That is, the goal configuration for this induced problem is not unique.

We claim this grouping operation can be achieved in $O(m_1 + m_2)$ steps (or makespan). Then, assuming without loss of generality that $m_1 \leq m_2$, after the first iteration, two half grids of dimensions about $\frac{m_1}{2} \times m_2$ are obtained. Invoking SAG on each of the two sub-problems, the second iteration takes $O(\frac{m_1}{2} + m_2)$ steps because the two sub-problems can be solved in parallel. The overall recursion takes $O(m_1 + m_2) + O(\frac{m_1}{2} + m_2) + O(\frac{m_1}{2} + \frac{m_2}{2}) + \ldots = O(m_1 + m_2)$ steps, as desired.

To carry out the $O(m_1 + m_2)$-step grouping operation in the first iteration as just claimed (e.g., Fig. 10), motion primitives are first built, starting from a 3-step pairwise swaps on a 2×3 grid, as shown in Fig. 11.

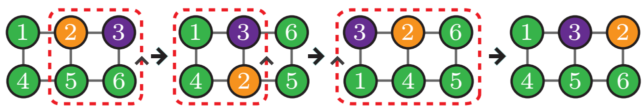


Figure 11. In three steps, robots 2 and 3 can be "swapped".

Then, multiple of these pairwise swaps can be combined, with parallel executions, to enable the swapping of two groups of robots of equal number on a path of length ℓ embedded in a grid (see, e.g., Fig. 12). This can be completed in $O(\ell)$ steps. Note that these two groups to be exchanged may overlap; let us denote the group swapping procedure on an embedded path as *line swap*. Multiple line swaps can be carried out in parallel on parallel paths.

Using line swaps, to realize the grouping procedure (e.g., Fig. 10), we may simply convert it to a Rubik Table (more precisely, Rubik Rectangle) problem by relabeling the robots to form m_2 types. One possible relabeling is given in Fig. 13. The Rubik Table results then readily apply to produce three

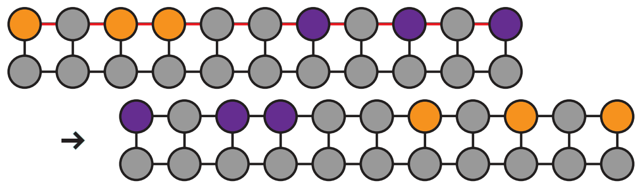


Figure 12. The orange and purple robots on the red path of length ℓ can be exchanged in $O(\ell)$ steps.

sets of column and row shuffles that can be simulated using line swaps, with each set executed in parallel, guaranteeing the $O(m_1 + m_2)$-step bound.

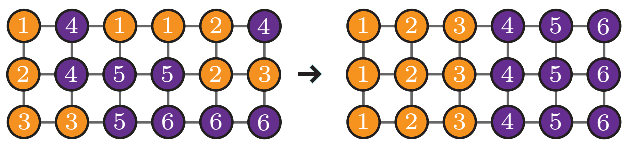


Figure 13. Converting the grouping problem from Fig. 10 to a Rubik Rectangle problem with six "types".

5.2 A New Expected $O(1)$-Optimal Algorithm

In the i^{th} iteration of the SAG algorithm, 2^i calls of the Rubik Table results must be invoked. Here, we describe a simplified algorithm that only requires a single call to the labeled Rubik Table result. As a preparation step, we first update the line swap motion primitive to apply to fully labeled robots instead of just two groups of robots.

Lemma 5.1. (Labeled line swap algorithm). *The arbitrary reconfiguration of labeled robots on a path of length ℓ embedded in an $m_1 \times m_2$ grid with $m_1, m_2 \geq 2$ and $m_1 m_2 \geq 6$ can be realized in $O(\ell)$ steps.*

Proof. The labeled line swap can be realized by recursive and parallel calls to the line swap procedure. For the first iteration, the path is split in the middle, which induces a partition of the robots; some must have goals in the current half they are on and the others must have goals on the opposite half. This leads to a problem that is exactly the same as that illustrated in Fig. 12, which can be solved by the line swap algorithm in $O(\ell)$ steps, yielding two sub-problems on two paths of length about $\frac{\ell}{2}$ each, that can be solved in parallel recursively. All together, the required number of steps for solving the entire problem is then $O(\ell) + O(\frac{\ell}{2}) + O(\frac{\ell}{4}) + \ldots = O(\ell)$. □

The labeled line swap routine allows the simulation of both column and row shuffles as required for solving a labeled Rubik Table problem. Given an MRMP problem on an $n \times n$ square grid, Theorem 3.1 can readily compute the $n + n + n$ shuffles for solving the MRMP problem as a Rubik Table problem. Each set of n shuffles can be executed as labeled line swaps in parallel, requiring a makespan of $O(n)$. Clearly, the same process works on an $m_1 \times m_2$ grid, yielding:

Proposition 5.1. (Expected $O(1)$-optimal algorithm for MRMP via Rubik Table). *MRMP on an $m_1 \times m_2$ grid with $m_1, m_2 \geq 2$ and $m_1 m_2 \geq 6$ can be solved in $O(m_1 + m_2)$ steps, using a single call to the labeled Rubik Table algorithm.*

6 Conclusion and Discussion

In studying decision-making problems for many mobile objects including stack rearrangement and multi-robot motion planning, we propose an abstract problem involving the shuffling of columns and rows of an $n \times n$ table containing n^2 items, which we call the Rubik Table problem. We show that, surprisingly, the Rubik Table problems can be solved using a minimal number of column and row shuffle operations. Then, through simulating these column and row shuffle operations, more efficient methods are developed for problems involving moving many objects. Specifically, for stack rearrangement problems, the fat Rubik Table result applies recursively to yield an $O(nd)$ pop-push algorithm for all $d \leq n^{\frac{m}{2}}$ where $m > 0$ is an arbitrary constant, which meets the lower bound. For multi-robot motion planning, Rubik Table results facilitate the iterative split and group of two groups of robots, leading to an *expected* $O(1)$ makespan-optimal algorithm, which can be used to drive an $O(1)$ makespan-optimal algorithm for the same. This work further provides an updated expected $O(1)$-optimal algorithm using a single call to the labeled Rubik Table algorithm.

We conclude the work by discussing several interesting open questions for the readers to explore further.

Solving Rubik Table problems with fewer shuffles. It is conceivable that a typical Rubik Table problem may not need $2n$ shuffles to solve. Given its general applicability, it is interesting to seek algorithms for optimally solving Rubik Table problems minimizing the total number of shuffles. Another directly related question to ponder here is the computational complexity of optimizing the number of shuffles; the problem appears to be intractable. The same questions can be asked on the labeled Rubik Table problems and other generalizations.

Bound gap. Whereas we know that it is not possible to reach $O(nd)$ for LSR for arbitrary n and d, we do not know whether the same is true for CSR. In our algorithmic solution, though we achieve $O(nd)$ for arbitrarily large but fixed $\frac{\log d}{\log n}$, we have not fully closed the gap for CSR. In the approach that we have used, the issue is caused by the $3k$ recursive calls. The 3 there is where the 3^m factor (in the $O(3^m nd)$ complexity stated in Theorem 4.1) comes from. For CSR, we were able to further drop the required number of moves to $O(2^m nd)$ with a much more involved argument than repeatedly applying Rubik Table results. Reducing the number of recursive calls may get us closer to closing the small remaining gap between the lower and upper bounds.

Hardness of stack rearrangement. The question of whether polynomial time algorithms can be designed for optimally solving CSR and LSR remains open. We conjecture that both CSR and LSR are NP-hard. In this regard, it may be interesting to study the case of constant d. Whereas the case of $d = 1$ can be readily solved, larger d appears to be challenging.

Utility of multiple buffer stacks In the current study, we have mainly examined the case of using a single buffer stack. We also show that using $\sqrt{n}$ empty buffer stacks allow the resolution of CSR in $O(nd)$ pop-pushes without any additional conditions imposed between n and d as required by, e.g., Theorem 4.2. A natural question to ask is for what values of $b \in [1, \sqrt{n})$, b empty buffer stacks would enable solving CSR in $O(nd)$ pop-pushes. As have been discussed, it is not clear that $\sqrt{n}$ buffer stacks are sufficient for solving LSR in $O(nd)$ pop-pushes for arbitrary n and d, which also warrants further examination.

Other queuing models As generalizations to the current problem, it could be interesting to study a two-dimensional stack setting, e.g., items may be accessed both from the top or from the left side. Does such a setting, which provides similar storage capacity as stacks, allow more access flexibility? One may also replace a stack with a queue that may be accessed from both ends. Many additional settings similar to these two could be examined.

Acknowledgment

The work is supported in part by NSF awards IIS-1734419, IIS-1845888 and CCF-1934924.

References

Ajtai M, Komlós J and Szemerédi E (1983) An 0 (n log n) sorting network. In: *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. pp. 1–9.

Banfi J, Basilico N and Amigoni F (2017) Intractability of time-optimal multirobot path planning on 2d grid graphs with holes. *IEEE Robotics and Automation Letters* 2(4): 1941–1947.

Ben-Shahar O and Rivlin E (1998) Practical pushing planning for rearrangement tasks. *IEEE Transactions on Robotics and Automation* 14(4): 549–565.

Borgman B, van Asperen E and Dekker R (2010) Online rules for container stacking. *OR spectrum* 32(3): 687–716.

Brousseau BA (1980) Tower of hanoi with more pegs. *J. Recreational Mathematics* 8.

Christofides N and Colloff I (1973) The rearrangement of items in a warehouse. *Operations Research* 21(2): 577–589.

Cole R, Ost K and Schirra S (2001) Edge-coloring bipartite multigraphs in o (e logd) time. *Combinatorica* 21(1): 5–12.

Dayama NR, Krishnamoorthy M, Ernst A, Narayanan V and Rangaraj N (2014) Approaches for solving the container stacking problem with route distance minimization and stack rearrangement considerations. *Computers & Operations Research* 52: 68–83.

Demaine ED, Demaine ML and O'Rourke J (2000) Pushpush and push-1 are np-hard in 2d. *arXiv preprint cs/0007021* .

Demaine ED, Fekete SP, Keldenich P, Meijer H and Scheffer C (2019) Coordinated motion planning: Reconfiguring a swarm of labeled robots with bounded stretch. *SIAM Journal on Computing* 48(6): 1727–1762.

Demaine ED and Hoffmann M (2001) Pushing blocks is np-complete for noncrossing solution paths. In: *Proc. 13th Canad. Conf. Comput. Geom.* Citeseer, pp. 1–5.

Erdmann M and Lozano-Perez T (1987) On multiple moving objects. *Algorithmica* 2(1-4): 477.

Garrett CR, Lozano-Pérez T and Kaelbling LP (2015) Ffrob: An efficient heuristic for task and motion planning. In: *Algorithmic Foundations of Robotics XI*. Springer, pp. 179–195.

Goel A, Kapralov M and Khanna S (2013) Perfect matchings in o(nlogn) time in regular bipartite graphs. *SIAM Journal on Computing* 42(3): 1392–1404.

Grigorchuk R and Šunik Z (2006) Asymptotic aspects of schreier graphs and hanoi towers groups. *Comptes Rendus Mathematique* 342(8): 545–550.

Hall P (2009) On representatives of subsets. In: *Classic Papers in Combinatorics*. Springer, pp. 58–62.

Han SD, Stiffler NM, Bekris KE and Yu J (2018a) Efficient, high-quality stack rearrangement. *IEEE Robotics and Automation Letters* 3(3): 1608–1615. Note: presented at ICRA 2018.

Han SD, Stiffler NM, Krontiris A, Bekris KE and Yu J (2018b) Complexity results and fast methods for optimal tabletop rearrangement with overhand grasps. *The International Journal of Robotics Research* 37(13-14): 1775–1795.

Havur G, Ozbilgin G, Erdem E and Patoglu V (2014) Geometric rearrangement of multiple movable objects on cluttered surfaces: A hybrid reasoning approach. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 445–452.

Huang E, Jia Z and Mason MT (2019) Large-scale multi-object rearrangement. In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 211–218.

Kornhauser D, Miller G and Spirakis P (1984) Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In: *25th Annual Symposium on Foundations of Computer Science, 1984*. IEEE, pp. 241–250.

Krontiris A and Bekris KE (2015) Dealing with difficult instances of object rearrangement. In: *Robotics: Science and Systems*. pp. 1–8.

Krontiris A and Bekris KE (2016) Efficiently solving general rearrangement tasks: A fast extension primitive for an incremental sampling-based planner. In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 3924–3931.

Li J, Tinka A, Kiesel S, Durham JW, Kumar TS and Koenig S (2020) Lifelong multi-agent path finding in large-scale warehouses. In: *Proceedings International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. pp. 1898–1900.

Shome R, Solovey K, Yu J, Bekris K and Halperin D (2021) Fast, high-quality two-arm rearrangement in synchronous, monotone tabletop setups. *IEEE Transactions on Automation Science and Engineering* 18(3): 888–901. DOI:10.1109/TASE.2021.3055144.

Solovey K and Halperin D (2016) On the hardness of unlabeled multi-robot motion planning. *The International Journal of Robotics Research* 35(14): 1750–1759.

Stilman M and Kuffner J (2008) Planning among movable obstacles with artificial constraints. *The International Journal of Robotics Research* 27(11-12): 1295–1307.

Szegedy M (1999) In how many steps the k peg version of the towers of hanoi game can be solved? In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, pp. 356–361.

Szegedy M and Yu J (2021) On rearrangement of items stored in stacks. In: LaValle SM, Lin M, Ojala T, Shell D and Yu J (eds.) *Algorithmic Foundations of Robotics XIV*. Cham: Springer International Publishing. ISBN 978-3-030-66723-8, pp. 518–533.

Van Den Berg J, Stilman M, Kuffner J, Lin M and Manocha D (2009) Path planning among movable obstacles: a probabilistically complete approach. In: *Algorithmic Foundation of Robotics VIII*. Springer, pp. 599–614.

West J (1993) Sorting twice through a stack. *Theoretical Computer Science* 117(1-2): 303–313.

Wilfong G (1991) Motion planning in the presence of movable obstacles. *Annals of Mathematics and Artificial Intelligence* 3(1): 131–150.

Wurman PR, D'Andrea R and Mountz M (2008) Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine* 29(1): 9–9.

Yu J (2015) Intractability of optimal multirobot path planning on planar graphs. *IEEE Robotics and Automation Letters* 1(1): 33–40.

Yu J (2018) Constant factor time optimal multi-robot routing on high-dimensional grid. In: *Robotics: Science and Systems (RSS)*. pp. 1–8.

Yu J and LaValle S (2013) Structure and intractability of optimal multi-robot path planning on graphs. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. pp. 1–7.