An Almost-Optimal Upper Bound on the Push Number of the Torus Puzzle

Matteo Caporrella
Department of Information Engineering, Computer Science, and Mathematics, University of L'Aquila, Italy

Stefano Leucci
Department of Information Engineering, Computer Science, and Mathematics, University of L'Aquila, Italy

Abstract

We study the *Torus Puzzle*, a solitaire game in which the elements of an input $m \times n$ matrix need to be rearranged into a target configuration via a sequence of unit rotations (i.e., circular shifts) of rows and/or columns.

Amano et al. proposed a more permissive variant of the above puzzle, where each row and column rotation can shift the involved elements by any amount of positions. The number of rotations needed to solve the original and the permissive variants of the puzzle are respectively known as the *push number* and the *drag number*, where the latter is always smaller than or equal to the former and admits an existential lower bound of $\Omega(mn)$. While this lower bound is matched by an $O(mn)$ upper bound, the push number is not so well understood. Indeed, to the best of our knowledge, only an $O(mn \cdot \max\{m, n\})$ upper bound is currently known.

In this paper, we provide an algorithm that solves the Torus Puzzle using $O(mn \cdot \log \max\{m, n\})$ unit rotations in a model that is more restricted than that of the original puzzle. This implies a corresponding upper bound on the push number and reduces the gap between the known upper and lower bounds from $\Theta(\max\{m, n\})$ to $\Theta(\log \max\{m, n\})$.

2012 ACM Subject Classification Mathematics of computing → Permutations and combinations; Theory of computation → Design and analysis of algorithms

Keywords and phrases Torus puzzle, Push number, Permutation puzzles

Digital Object Identifier 10.4230/LIPIcs.FUN.2026.11

Related Version *Full Version*: `https://arxiv.org/abs/2601.08989`

Supplementary Material *InteractiveResource*: `https://www.isnphard.com/g/torus-puzzle/`

1 Introduction

The *Torus Puzzle*, also known as *Sliders*, *TwoBik*, *RotSquare*, *Loopover*, or *Sixteen Puzzle*, is a solitaire game played on an $m \times n$ matrix whose entries consist of (all and only) the integers from 1 to mn. These elements are initially scrambled, and the goal is that of rearranging them via a sequence of *unit rotations* of rows and/or columns that causes the resulting arrangement of the elements to end up in the *sorted configuration*, i.e., in increasing order when read from left to right and from top to bottom.[1] More precisely, a unit rotation of a row is specified by the row index and by a direction, which can be either leftward or rightward, and consists of a circular shift of all the elements in the chosen row along the chosen direction. Similarly, a column can be rotated either upward or downward. Figure 1 shows an example of such rotations.

[1] More formally, the element on row i and column j of the sorted configuration is the integer $(i-1)n + j$.

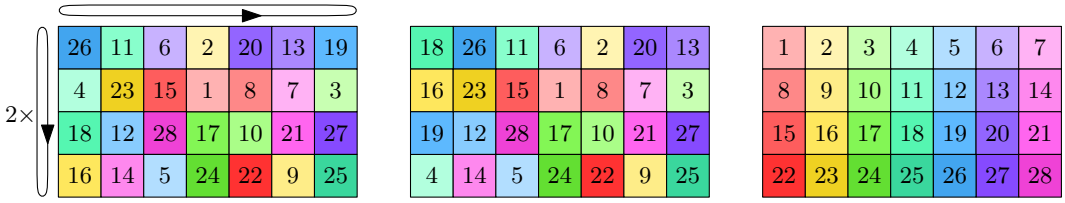

Figure 1 A sortable input instance of the Torus Puzzle (left); the configuration obtained from the input configuration by performing a unit rightward rotation of the first row, followed by two unit downward rotations of the first column (middle); and the target configuration, i.e., the one in which the matrix is sorted (right). Elements belonging to different columns in the target configuration have different hues, while elements belonging to different rows have different lightnesses.

It turns out that not all input matrices are sortable, and it is not hard to derive a characterization of sortable matrices which also serves as a linear-time algorithmic test (see the related works section for a more detailed discussion). This motivated previous research to consider only sortable puzzles and to provide general upper and lower bounds on the number of unit rotations needed to solve the puzzle. This quantity is known as the *push number* and was introduced by Amano et al. [2] along with the related notion of *drag number*, which counts multiple consecutive rotations of the same row or column only once. Equivalently, the drag number can be interpreted as the number of moves needed to solve the Torus Puzzle when each move allows rotating a row or column of choice by an arbitrary amount.[2]

Amano et al. showed a constructive upper bound of $O(mn)$ on the drag number of any sortable $m \times n$ instance of the Torus Puzzle, along with a matching existential lower bound of $\Omega(mn)$, i.e., that there exists a constant $\varepsilon > 0$ such that, for any choice of m and n with $mn > 1$, some sortable $m \times n$ instance has a drag number of at least εmn. Clearly, the push number is no smaller than the drag number, and it can never exceed it by more than a multiplicative factor of $\left\lfloor \frac{\max\{m,n\}}{2} \right\rfloor$ (since a rotation by an arbitrary amount can always be simulated using multiple unit rotations along the "shortest" direction).

Unfortunately, to the best of our knowledge, the best asymptotic lower and upper bounds on the push number, in terms of m and n, are exactly those obtained by combining the $\Omega(mn)$ and $O(mn)$ bounds on the drag number given in [2] with the relations above. That is, there exists an algorithm that solves the Torus Puzzle using $O(mn \cdot \max\{m, n\})$ unit rotations, and any such algorithm needs to perform at least $\Omega(mn)$ rotations in the worst case, leaving a multiplicative gap of $\Theta(\max\{m, n\})$ between the upper and the lower bound.

Our results and techniques. In this paper we make progress towards closing this gap by providing an algorithm solving any sortable $m \times n$ instance of the Torus Puzzle using $O(mn \cdot \log \max\{m, n\})$ *unit rotations*, which is optimal up to polylogarithmic factors in the worst case. We also show that such a sequence of rotations can be found in the same asymptotic worst-case time. The above results hold in the even more restricted model in which the rotation directions cannot be freely chosen by the algorithm, but are independently specified, for each row and column, as part of the input instance.

[2] To be pedantic, [2] only defined the push and drag numbers for a specific sequence of rotations, and then introduced two functions $\mathrm{pn}(A)$ and $\mathrm{dn}(A)$ corresponding to the minimum push and drag numbers among all sequences of rotations that sort the elements of A, respectively. We find it natural to refer to $\mathrm{pn}(A)$ and $\mathrm{dn}(A)$ as the push and drag numbers of instance A, respectively.

While the solution strategy of [2] is based on repeatedly swapping triples of distinct elements, an operation that requires $\Omega(\max\{m, n\})$ unit rotations, our algorithm follows a different approach which relies on three main ingredients. First, we treat all integers that should end up in the first row of the sorted configuration as some sort of *buffer elements*, whose exact positions can be (temporarily) disregarded; we move each remaining element to its intended final column, and the buffer elements to the first row. Next, we sort the non-buffer elements within each column using a strategy akin to that of RadixSort. This requires handling multiple columns in parallel in order to meet the claimed upper bound on the number of moves. Finally, we deal with the buffer elements: since they are all on the first row, the challenge is that of suitably permuting them without affecting the placement of the other elements. As an intermediate step, we show how to simultaneously swap a collection of pairwise-disjoint pairs of buffer elements while causing only controlled side effects to the rest of the matrix. We can then combine multiple parallel swaps in a way that ensures that their side effects cancel out, thus obtaining the sought sorted matrix.

Some historic notes and related works. The earliest report we found of the Torus Puzzle dates back to 1981, when Takeshi Ogihara implemented it on a Mitsubishi MELCOM mainframe using a text-based interface. In 1993, following Ogihara's suggestion while he was involved in upgrading Osaka University's mainframes to NeXT computers, M. Torigoe, a Faculty of Science student, reimplemented the puzzle as a graphical NeXTSTEP application named RotSquare. Ogihara later ported RotSquare to OpenStep, Mac OS X, and iOS (see Figure 2 (left) for a screenshot) [23]. Several successive implementations have been independently produced under various names. A commercial physical realization of a puzzle that appears very similar to a 4×8 Torus puzzle has been reported to exist back in 1987 on the MIT's CubeLovers mailing list (see [18, 19, 25] and Figure 2 (right) for a 1995 photo). Another puzzle involving a toroidal movement mechanic is implemented by the 1992 video game Yoshi's Cookie for the Nintendo Entertainment System, where players must rotate rows and columns of a grid in order to match similar elements *à la* Bejeweled [14, 17].

The first mention of the Torus Puzzle in a mathematical context was in 1988, when Diaconis considered the $n \times n$ version as an easier to analyze variant of the Fifteen Puzzle, showing that $O(n^3)$ random rotations are enough to randomize the position of each individual element and conjecturing that $O(n^3 \cdot \log n)$ rotations suffice to shuffle all elements [10, p. 90]. Diaconis' conjecture is still open to this day, and the best-known upper bound for the *Torus Shuffle* is of $O(n^3 \cdot \log^3 n)$ random rotations [5].

In 1997, mathematician Alexander Bogomolny [6] described the puzzle on his website under the name *Sliders Puzzle*. He proved that every 4×4 matrix is sortable by providing an explicit strategy that swaps two elements of choice without affecting the positions of the other elements. In general, each arrangement of the elements of an $m \times n$ matrix can be seen as a permutation of $\{1, 2, \ldots, mn\}$, with the sorted configuration corresponding to the identity permutation. Bogomolny observed that, in any $n \times n$ puzzle with odd n, all rotations correspond to even permutations, hence all configurations reachable from the initial one must have its same parity. Moreover, all such parity-preserving configurations can actually be reached when n is odd. For even values of n, the elements can always be rearranged into any configuration of choice. Bogomolny's impossibility argument immediately extends to rectangular matrices: an $m \times n$ matrix in which both m and n are odd is sortable only if the permutation induced by the initial configuration of the elements is even.

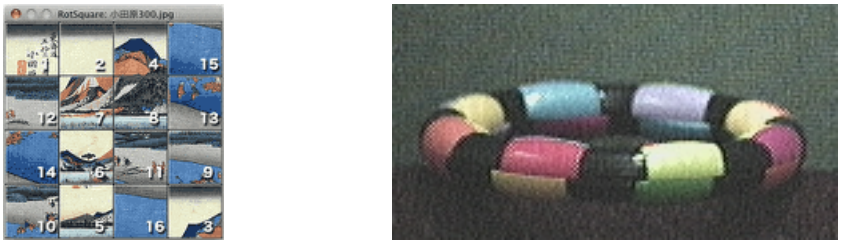

Figure 2 Left: A screenshot of the RotSquare application for Mac OS X by Takeshi Ogihara. Right: a physical puzzle in which eight colored sections of a Torus can rotate in 90-degree increments along the shorter dimension, while the two halves of the torus delimited by a horizontal plane passing through the center can rotate along the long dimension. The puzzle resembles a 4×8 instance of the Torus Puzzle in which some elements are indistinguishable and two pairs of adjacent rows rotate together. The photo was posted in 1995 to MIT's CubeLovers mailing list [18] and is reproduced here with the author's permission.

In 1999, Donald Lee Greenwell gave explicit sequences swapping any two adjacent elements in 6×6 and 8×8 matrices [13], and argued that similar swaps can be performed in any $n \times m$ matrix, as long as at least one of n and m is even. Greenwell also proposed two additional variants of the puzzle, where rows and columns wrap around as if they were laid out on a Klein bottle or on the projective plane.

Later, Amano et al. [2] considered general $m \times n$ matrices and described a sequence of rotations performing a three-element swap: given three distinct elements x_0, x_1, x_2, it is possible to move each x_i to the position of $x_{(i+1) \bmod 3}$ without affecting the placement of the other elements. Then, the authors show how a sequence of $O(mn)$ three-element swaps can sort any $m \times n$ matrix where at least one of m, n, and the initial permutation of the elements is even. Since each three-element swap can be implemented by a sequence consisting of a constant number of *compound rotations*[3] of rows and columns, this implies an upper bound of $O(mn)$ on the drag number. However, the overall number of unit rotations required by a three-element swap can be as large as $\Theta(\max\{m, n\})$, yielding the best-known upper bound of $O(mn \cdot \max\{m, n\})$ on the push number. As observed by [2], a lower bound on the drag number (and on the push number) can be obtained by noticing that there are at least $\frac{(mn)!}{2}$ distinct sortable $m \times n$ matrices and at most $(2mn)^{\ell+1}$ distinct sequences of up to ℓ (possibly compound) rotations, implying that some sortable $m \times n$ matrix requires at least $\log_{2mn} \frac{(mn)!}{2} - 1 = \Omega(mn)$ rotations to be sorted.
To summarize, the results in [2, 6, 13] characterize the sortable instances of the Torus Puzzle:

▶ Remark 1. An $m \times n$ instance A of the Torus Puzzle can be sorted through a sequence of row and column rotations if and only if at least one of n, m, and the permutation of $\{1, 2, \ldots, nm\}$ induced by the arrangement of elements in A is even.

Several heuristic strategies for finding solutions to Torus Puzzle instances have been proposed in [15], and a *colored* variant of the Torus Puzzle has been shown to be NP-Hard in [2]. In this variant, each entry of the matrix consists of one of two colors, and the goal is that of determining whether some target arrangement of colors can be reached.

The Torus Puzzle falls into the general category of permutation puzzles [11, 21]: the input configuration is an element of a suitable permutation group, and the goal is that of deciding whether it belongs to the same connected component of the desired target configuration. If

[3] We use *compound rotation* to refer to multiple consecutive unit rotations of the same row or column.

the answer is affirmative, then one can also look for a sequence of moves transforming the initial configuration into the target one, or even for the shortest such sequence. These two problems correspond to the generalized word problem and to the shortest word problem on the associated permutation group. Some other notable puzzles in this category are the Fifteen Puzzle [3, 9, 12, 24], the Rubik's Cube [7, 8] including a variant played on a torus [1], several other *swapping token* problems (see, e.g., [4, 16, 20, 26] and the references therein), and *rotating puzzles on graphs* [22, 27]. In particular, the latter appear to be close in spirit to the Torus Puzzle: given a graph G with an element on each vertex, the goal is that of rearranging these elements into a target configuration via a sequence of rotations along the cycles of G. An instance of the Torus Puzzle can be seen as a rotating puzzle on an $m \times n$ torus graph, with the further restriction that only some of the cycles can be rotated.

Finally, we mention that the problem of finding lower and upper bounds to the drag and/or push numbers of Torus Puzzle can be cast as the problem of finding corresponding bounds on the diameter of the Cayley graph of the associated permutation group.

Organization of the paper. Section 2 introduces the notation used throughout the paper and recalls some standard definitions. Section 3 presents our sorting algorithm, which solves any sortable instance of the Torus puzzle using $O(mn \cdot \log \max\{m, n\})$ unit rotations. For the sake of simplicity, we assume that the allowed (unit) row rotations are all in the rightward direction, while columns can only be rotated downward. Section 4 discusses how the algorithm can be implemented to run in time $O(mn \cdot \log \max\{m, n\})$, how it can be adapted to handle the case in which each row and column can only be rotated in a direction specified as part of the input, and the problems left open.

A playable version of the puzzle, along with an implementation of our algorithm, is available at `https://www.isnphard.com/g/torus-puzzle/`, which might help the reader visualize the different subroutines discussed in the technical part of the paper.

2 Preliminaries

Matrices and rotations. We use A to refer to the $m \times n$ matrix to be sorted, and we assume that $m, n \geq 2$.[4] For an integer $x \in \{1, \ldots, mn\}$, we denote by t-row(x) and t-col(x) the *target row* and the *target column* in which x should be placed in the sorted configuration, respectively, i.e., we let t-row$(x) = \lceil x/n \rceil$ and t-col$(x) = ((x-1) \bmod n) + 1$, where rows and columns are numbered from 1.

The first row of A will be denoted by R, while $R[j]$ will denote either the position on the j-th column of R or the element contained therein, depending on context. Similarly, the j-th column of A will be denoted by C_j, and the element or position on the i-th row of C_j will be denoted by $C_j[i]$. We refer to $C_j[1]$ and $C_j[m]$ as the *head* and the *tail* of C_j, respectively. The *body of* C_j refers to all positions/elements that are in column C_j but not in C_j's head (notice that the tail of a column is part of its body). Observe that R contains all and only the columns' heads and, more precisely, $C_j[1]$ coincides with $R[j]$.

Since the model considered in the next section only allows for unit rightward rotations of rows, and unit downward rotations of columns, we refer to a row/column notation by the name of the corresponding row/column, and we may omit the terms "unit", "rightward", or "downward". A non-negative integer superscript k denotes k consecutive rotations of the

[4] Given an $m \times 1$ or a $1 \times n$ matrix A, it is easy to design a linear-time algorithm deciding whether A can be sorted and, if that is the case, returning a sequence of $O(nm)$ moves that sorts A.

same row/column, and we may write a sequence of rotations involving multiple rows/columns by juxtaposing row and column names with their respective superscripts. Sequences are evaluated from left to right. For example, $C_1^3R^2C_2^0C_n$ is equivalent to $C_1C_1C_1RRC_n$ and denotes three consecutive downward rotations of C_1, followed by two rightward rotations of R, followed by a downward rotation of C_n.

If p and q are two positions in the matrix, and S is a sequence of rotations, we write $p \xrightarrow{S} q$ to denote that, as a result of applying the sequence S, the element initially placed in p ends up in position q. If $p_1, p_2, \ldots, p_{k+1}$ are positions and $S_1, S_2, \ldots, S_k$ are sequences of rotations such that $p_i \xrightarrow{S_i} p_{i+1}$, we might express this compactly using the shorthand $p_1 \xrightarrow{S_1} p_2 \xrightarrow{S_2} p_3 \xrightarrow{S_3} \ldots \xrightarrow{S_k} p_{k+1}$. Observe that the above implies $p_1 \xrightarrow{S_1S_2\ldots S_k} p_{k+1}$, where $S_1S_2\ldots S_k$ denotes the concatenation of $S_1, S_2, \ldots, S_k$, in this order.

Permutations. Since each rotation can also be interpreted as permutation of the elements in A, we employ the *left-to-right* convention for permutation products. In this way, the configuration of elements resulting from a sequence of rotations corresponds to the action on A of the permutation resulting from their product. Finally, we briefly recall some standard concepts used in the paper: a permutation π can be decomposed into a product of some number k of disjoint *cycles*[5] $C_1, C_2, \ldots, C_k$, i.e., $\pi = C_1C_2\ldots C_k$. A *transposition* is a permutation consisting of a single cycle of length 2, and an *involution* is a permutation in which each cycle has length at most 2, i.e., each cycle is either a *fixed-point* (a cycle of length 1) or a transposition. The number of cycles of length 2 of a permutation π is denoted by $a_2(\pi)$. Each permutation π can be written as either a product of an even number of transpositions, or as a product of an odd number of transpositions, but not both, and the *parity* of π is defined as the parity of the number of such transpositions. If a permutation π of n elements has k cycles (including its fixed-points), then the parity of π is equal to the parity of $n-k$. Moreover, the parity of a permutation π is the same as that of its inverse π^{-1}. The product $\pi\sigma$ of two permutations π and σ is even if and only if the parities of π and σ are the same (i.e., π and σ are both even or both odd).

3 Solving the Torus Puzzle using $O(mn \cdot \log \max\{m,n\})$ unit rotations

In this section we provide an algorithm solving any sortable $m \times n$ instance of the Torus Puzzle using only $O(mn \cdot \log \max\{m,n\})$ rotations. We consider the case in which all rows can only rotate rightward while columns can only rotate downward. In Section 4 we discuss how arbitrary rotation directions can be handled.

We start with some definitions. We say that a column C_j is *near-full* if all elements x in the body of C_j satisfy t-col$(x) = j$, i.e., they are in their target column. If C_j is not near-full, we say that it is *underfull*. A column C_j is *body-full* if it is near-full and its body contains all (and only) the elements x with t-col$(x) = j$ and t-row$(x) > 1$. In other words, the body of C_j contains exactly the same set of elements as that in the sorted configuration, but these elements can appear in an arbitrary order. Finally, a column is *body-sorted* if all elements in its body are in their respective target positions.

The algorithm first makes all columns near-full, then body-full, then sorts their bodies in groups so that all columns become body-sorted, and finally handles sorting the first row. Each of the following subsections describes one of the above steps, in the above order.

[5] A cycle $(x_1\ x_2\ \ldots\ x_\ell)$ of length $\ell \geq 1$ is a permutation γ such that: (i) all elements $x_1, \ldots, x_\ell$ are distinct, (ii) $\gamma(x_i) = x_{i+1}$ for $i = 1, \ldots, \ell-1$, (iii) $\gamma(x_\ell) = x_1$, and (iv) $\gamma(x) = x$ for all $x \notin \{x_1, \ldots, x_\ell\}$.

Algorithm 1 Procedure FillColumns.

Input: An $m \times n$ matrix A with $m \leq n$.
Output (in place): A matrix in which all columns are near-full.

```
1 while there exists an underfull column do
2     for j = 1, 2, ..., n do
3         while t-col(R[j]) = j and C_j is underfull do
4             Rotate C_j;
5     Rotate R;
```

3.1 Making columns near-full

We start by describing a procedure, which we name FillColumns, that receives as input an $m \times n$ matrix A with $m \leq n$ and rearranges the elements in A via a sequence of $O(mn \cdot \log n)$ rotations to ensure that all columns of the resulting configuration are near-full. Intuitively, the algorithm uses the first row as a sort of "sushi belt" from which each column can "pick up" a desirable element while simultaneously "discarding" an undesirable one.

In details, R is repeatedly rotated and, whenever (i) some element x in R ends up in its target column C_j (i.e., t-col$(x) = j$), and (ii) such a column is underfull, then C_j is also rotated. The rotation of C_j causes x to move from the head to the body of C_j, and the tail of C_j to move to the head of C_j (in place of x), i.e., $C_j[1] \xrightarrow{C_j} C_j[2]$ and $C_j[m] \xrightarrow{C_j} C_j[1]$. These column rotations are repeated until either C_j becomes near-full, or the target column of the new head of C_j is no longer C_j (possibly both). When no element x in R satisfies conditions (i) and (ii), the procedure resumes rotating the first row. The procedure stops as soon as all columns are near-full. The pseudocode of FillColumns is shown in Algorithm 1, while Figure 3 shows a possible input, an intermediate configuration, and the resulting output.
We start our analysis by bounding the number of column rotations.

▶ **Lemma 2.** *The number of column rotations performed by FillColumns is $O(mn)$.*

Proof. We prove the claim by focusing on a generic column C_j and showing that, once C_j is rotated $m-1$ times, FillColumns performs no additional rotations on C_j.

Indeed, FillColumns ensures that each rotation of C_j moves some element x having target column C_j into the body of C_j and, specifically, into $C_j[2]$. Since $m-1$ additional downward rotations of C_j are needed for x to leave the body of C_j, we have that after performing any number $h \in \{0, \ldots, m-1\}$ rotations of C_j, the body of C_j contains at least h elements whose target column is C_j. Then, if the $(m-1)$-th rotation of C_j is performed, it must result in a column that is near-full. The claim follows by observing that FillColumns only rotates columns that are not near-full. ◀

Let $A_0 = A$ be the input matrix to FillColumns, and let A_h be the matrix as arranged immediately after the completion of the h-th iteration of the outer while loop of Algorithm 1, if it exists. We associate a *potential* $\Psi(A_h)$ to each A_h, where $\Psi(A_h)$ is defined as the number of elements x in A_h that are in the body of some column other than their target column, i.e., $\Psi(A_h) = |\{(i,j) \in \{2, \ldots, m\} \times \{1, \ldots, n\} : \text{t-col}(C_j[i]) \neq j\}|$. The following lemma shows that each group of n iterations of the outer while loop starting from some configuration A_h decreases the potential by at least half the number of underfull columns in A_h.

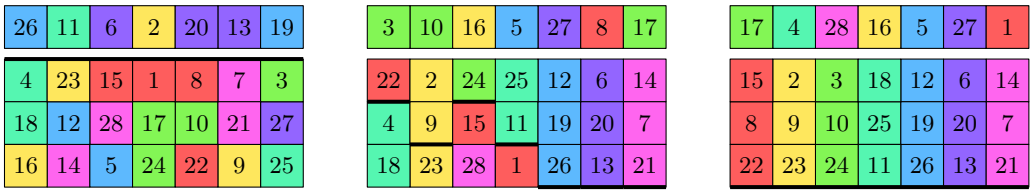

Figure 3 An input instance $A = A_0$ of the Torus Puzzle (left), and the matrices A_9 and A_{17} corresponding the arrangement of the elements in A at end of the ninth and of the last iteration of the outer while loop of FillColumns (middle and right, respectively). Elements are colored according to their target columns. A bold segment on each column C_j indicates the number of rotations of C_j performed by FillColumns until the shown configuration. A small gap has been added under row R for improved visual clarity. Notice how all columns in A_{17} are near-full.

▶ **Lemma 3.** *Consider a configuration A_h with f underfull columns. If A_{h+n} exists, then $\Psi(A_{h+n}) \leq \Psi(A_h) - \frac{f}{2}$.*

Proof. We assume that A_{h+n} exists since otherwise the claim is trivially true. For an underfull column C_j of A_h, we let X_j be the set of elements x in R that have C_j as their target column (i.e., such that t-col$(x) = j$), while for each near-full column C_j we let $X_j = \emptyset$.

Since row R of configuration A_h contains at most one element with target column C_j for each of the $n - f$ near-full columns C_j, there must be at least f distinct elements in R whose target column is underfull. Hence, $\sum_{j=1}^{n} |X_j| \geq f$.

In the rest of the proof we argue that at least $\frac{1}{2}\sum_{j=1}^{n} |X_j|$ elements from $\cup_{j=1}^{n} X_j$ are placed in the bodies of their respective columns in A_{n+h}. Since the only way for FillColumns to move an element x from the head to the body of its target column C_j is that of rotating C_j until some other element $y \neq x$ with t-col$(y) \neq j$ ends up in R, causing $\Psi(\cdot)$ to decrease by 1, this will imply that $\Psi(A_{h+n}) \leq \Psi(A_h) - \frac{1}{2}\sum_{j=1}^{n} |X_j| \leq \Psi(A_h) - \frac{f}{2}$, as claimed.

In particular, we focus on a specific column C_j with $|X_j| \geq 1$, and we show that at least $\max\{1, |X_j| - 1\} \geq \frac{|X_j|}{2}$ of the elements in X_j are in the body of C_j in configuration A_{h+n}. Since row R is rotated n times from configuration A_h to configuration A_{h+n}, and C_j is only rotated if t-col$(C_j[1]) = j$, we have that each element $x \in X_j$ becomes the head of C_j in some (distinct) configuration $A_{\ell(x)}$ with $h \leq \ell(x) < h + n$. Let $x_1, \ldots, x_k$ be the elements in X_j, in increasing order of $\ell(x_i)$. Consider then the $\ell(x_i)$-th iteration of the outer while loop of FillColumns. If $i = 1$ or $i < k$, then C_j is underfull in $A_{\ell(x_i)}$, which means that x is moved to the body of C_j in such iteration, and never leaves it in the following iterations. It follows that at least $\max\{1, k-1\}$ elements are in the body of C_j in configuration A_{h+n}. ◀

The above lemma allows us to upper bound the number of rotations of the first row.

▶ **Lemma 4.** *The number of rotations of R performed by FillColumns is $O(mn \cdot \log n)$.*

Proof. We partition the rotations of R performed by FillColumns into $1 + \lfloor \log n \rfloor$ phases. We say that a configuration A_h belongs to the i-th phase if its number of underfull columns lies in the half-open interval $[\frac{n}{2^{i-1}}, \frac{n}{2^i})$. We also say that a rotation of R belongs to phase i if the configuration A_h from which it is performed belongs to phase i.

By our definition of $\Psi(\cdot)$, we have that if A_h is in phase i, then $\Psi(A_h) \leq \frac{n}{2^{i-1}}(m-1)$. Moreover, Lemma 3 ensures that, if A_{h+n} exists, then $\Psi(A_{h+n}) \leq \Psi(A_h) - \frac{n}{2^{i+1}}$. This shows that there can be at most $\frac{n}{2^{i-1}}(m-1) \cdot \frac{2^{i+1}}{n} = 4(m-1)$ disjoint groups of n rotations of R in phase i. Hence, the number of rotations of R in each phase is at most $(n-1)+4n(m-1) < 4mn$. The claim follows by multiplying $4mn$ by the number of phases. ◀

Algorithm 2 Procedure FloatMinimums.

Input: An $m \times n$ matrix A in which all columns are near-full.
Output (in place): A matrix in which all columns are body-full.

```
1 for n times do
2     for j = 1, ..., n do
3         if t-col(R[j]) = j then
4             while t-row(R[j]) ≠ 1 do
5                 Rotate C_j;
6     Rotate R;
```

The stopping condition of FillColumns, together with Lemma 2 and Lemma 4, imply the following corollary, which summarizes the result of this section.

▶ **Corollary 5.** *Given an $m \times n$ input matrix A with $m \leq n$, FillColumns performs $O(mn \cdot \log n)$ rotations and rearranges the elements in A so that all columns become near-full.*

3.2 Transforming near-full columns into body-full columns

Once all the columns of a configuration A are near-full, it is easy to see that $O(mn)$ rotations suffice to rearrange the elements in A so that all columns become body-full. This is exactly the purpose of our next procedure, which we name FloatMinimums. The corresponding pseudocode is given in Algorithm 2, while Figure 4 shows an example input configuration along with the resulting output.

We start by noticing that the fact that all columns of A are near-full implies that R contains exactly one element x_j with t-col$(x_j) = j$ per column C_j. FloatMinimums uses this fact by performing n rotations of R: before each rotation, it checks whether any element x_j in R is on its target column C_j. Whenever this is the case, column C_j contains all (and only) the elements having j as their target column (in some arbitrary order), and FloatMinimums rotates C_j until its head becomes the unique element x with t-row$(x) = 1$ and t-col$(x) = j$, thus making C_j body-full with at most $m - 1$ rotations of C_j. Column C_j will never be rotated again by the procedure. Hence, we have the following:

▶ **Lemma 6.** *Given an $m \times n$ input matrix A in which all columns are near-full, procedure FloatMinimums performs $O(mn)$ rotations and rearranges the elements in A so that all columns become body-full.*

3.3 Transforming body-full columns into body-sorted columns

In this section, we show how a configuration of an $m \times n$ matrix A, with $m \leq n$, in which all columns are body-full, can be transformed into a configuration in which all columns are body-sorted using $O(mn \cdot \log m)$ rotations.

Sorting the body of a single column. As a warm-up, we start by sketching how a single body-full column C_j can be made body-sorted using $O(n \cdot \log m)$ rotations. The procedure will only rotate R and C_j, hence all elements in the body of columns other than C_j will remain in their initial positions. In particular, any other column that is already body-full (resp. body-sorted) will remain body-full (resp. body-sorted). This also means that the set of elements in R will remain unchanged, although their order might change.

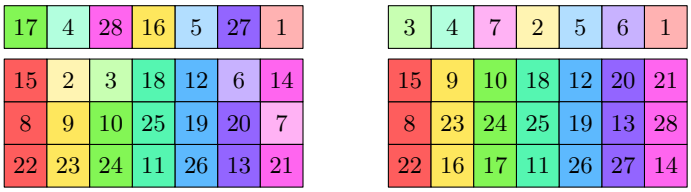

Figure 4 A possible input to FloatMinimums, which corresponds to the output of FillColumns in Figure 3 (left), and the resulting output (right). Elements belonging to different columns in the sorted configuration have different hues, and elements x with t-row$(x) = 1$ are lighter. Notice how all columns of the output are body-full.

The idea is that of simulating the RadixSort algorithm on the elements in the body of C_j. We achieve this by repeatedly *unloading* the body of C_j into R and then re-loading the same elements back into C_j in a suitable order. In particular, the procedure consists of $\lceil \log(m-1) \rceil$ phases, where the ℓ-th phase sorts the elements x of C_j according to the ℓ-th least significant bit of (a suitably shifted version of) t-row(x). This sorting step is stable; hence, after the last phase is completed, column C_j is body-sorted.

In details, a phase is subdivided into three sub-phases. In the first sub-phase we unload all elements of C_j into R with the sequence of rotations $(C_j R)^{m-1} R^{n-m+1}$, which also ensures that the overall number of rotations of R is n. The second sub-phase rotates R n additional times and, before each rotation, we check whether the element x on the head of C_j should be moved to the body of C_j, namely if t-col$(x) = j$, t-row$(x) \geq 2$, and the ℓ-th least significant bit lsb(t-row$(x) - 2, \ell)$ of t-row$(x) - 2$ is set. At this point, R has been rotated $2n$ times, and all the elements that were originally in the body of C_j and have their ℓ-th least significant bit set are now in the body of C_j. The third and final sub-phase is similar to the second one, with the only difference that the condition lsb(t-row$(x) - 2, \ell) = 1$ is replaced with lsb(t-row$(x) - 2, \ell) = 0$. This loads the remaining elements into C_j. Observe that in the second and third sub-phases, the elements inserted in C_j are encountered in the same order in which they were unloaded, hence each phase is stable.

Unfortunately, while sequentially applying the above simulation of RadixSort on each column C_j would render all columns body-full, this would require $\Omega(n^2 \cdot \log m)$ rotations, which is asymptotically larger than our $O(mn \cdot \log n)$ goal whenever $n = \omega(m)$.

Sorting the bodies of up to $\frac{n}{m-1}$ columns. To circumvent this issue, we show how to generalize the above RadixSort-inspired scheme to handle a number k of up to $\frac{n}{m-1}$ columns in parallel. The resulting procedure, which we name RadixSortBodies, performs $O((n + km) \cdot \log m)$ rotations and its pseudocode is given in Algorithm 3.

Let $\mathcal{C} = \{j_1, j_2, \ldots, j_k\}$ be a set of k column indices. RadixSortBodies starts by unloading, in parallel, the bodies of all columns C_j with $j \in \mathcal{C}$ into R. In order to do that, we choose k pairwise-disjoint subsets $I_1, \ldots, I_k$ of $\{1, \ldots, n\}$, each containing $m-1$ elements, e.g., we can choose $I_h = \{(h-1)(m-1)+1, (h-1)(m-1)+2, \ldots, h(m-1)\}$.[6] Intuitively, set I_h is associated to column C_{j_h} and contains the $m-1$ indices i corresponding to the positions $R[i]$ into which RadixSortBodies unloads the elements in the body of C_{j_h}.

[6] The algorithm described previously for sorting the body of a single column corresponds to the case $k = 1$, $\mathcal{C} = \{j_1\}$, and the choice $I_1 = \{1, \ldots, j_1\} \cup \{n - m + j_1 + 2, \ldots, n\}$ (where the second set of the union might be empty).

Algorithm 3 Procedure RadixSortBodies.

Input: An $m \times n$ matrix A with $m \leq n$ such that all columns of A are body-full. A collection $\mathcal{C} = \{j_1, \ldots, j_k\}$ of distinct column indices.
Output (in place): A matrix in which all columns C_j with $j \in \mathcal{C}$ are body-sorted, and the body of each column C_j with $j \notin \mathcal{C}$ is unaffected.

```
1  for ℓ = 1, 2, ..., ⌈log(m − 1)⌉ do
       // First phase
       // Unload the elements in the bodies of the columns in C into R
2      for r = 0, 1, ..., n − 1 do
3          foreach j_h ∈ C do
4              if ρ(j_h, r) ∈ I_h then
5                  Rotate C_{j_h};
6          Rotate R;
       // Second and third phases (with b = 1 and b = 0, respectively)
       // Load the elements back into the bodies of the columns in C
7      for b = 1, 0 do
8          for n times do
9              foreach j ∈ C do
10                 if t-col(R[j]) = j ∧ t-row(R[j]) ≥ 2 ∧ lsb(t-row(R[j]) − 2, ℓ) = b then
11                     Rotate C_j;
12             Rotate R;
```

As before, the procedure works in $\lceil \log(m-1) \rceil$ phases, where the ℓ-th phases is concerned with sorting the elements of interest w.r.t. their ℓ-th least significant bit. Similarly, each phase is also split into three sub-phases.

The first sub-phase unloads all elements in the body of the columns C_j with $j \in \mathcal{C}$ into R. In order to do so, we define $\rho(j, r) = ((j - r - 1) \bmod n) + 1$ so that, as a result of performing r rotations of R, the element that ends up in $R[j]$ is the one that was originally placed in $R[\rho(j, r)]$. In this sub-phase, row R is rotated n times and, before each rotation, we check whether $\rho(j_h, r) \in I_h$ for each $h \in \mathcal{C}$. For all j_h that satisfy the above condition, we rotate C_{j_h}, thus moving the element currently on the tail of C_{j_h} into R.

The second and third sub-phases behave similarly to one another, and handle the elements having their ℓ-th least significant bit set to 1 and 0, respectively. The second (resp. third) sub-phase, rotates R n times and, before each rotation, checks whether any of the elements currently on the heads the columns whose bodies are to be sorted should be loaded into their column's body. More precisely, if x is the head of some column C_j with $j \in \mathcal{C}$, this happens when $\text{t-col}(x) = j$, $\text{t-row}(x) \geq 2$, and $\text{lsb}(\text{t-row}(x) - 2, \ell) = 1$ for the second sub-phase (resp. $\text{lsb}(\text{t-row}(x) - 2, \ell) = 0$ for the third sub-phase). Notice that, within each of the last two sub-phases, the loaded elements are inserted in the bodies of the respective columns in the same order as they were unloaded in the first phase, hence the overall algorithm is stable. Figure 5 shows some configurations encountered during an execution of RadixSortBodies.

Sorting all column bodies. It is now easy to transform any matrix in which each column is body-full into a matrix in which each column is body-sorted using $O(mn \cdot \log n)$ rotations.

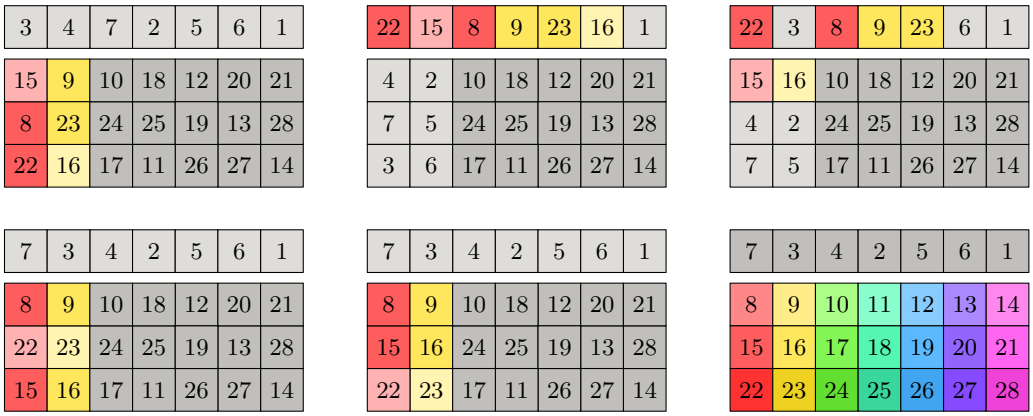

Figure 5 A possible input matrix for an execution RadixSortBodies sorting C_1 and C_2 (top left) which coincides to that output by FloatMinimums in Figure 4, along with the configurations resulting after the first, second, and third sub-phases of the first phase of RadixSortBodies (top middle, top right, and bottom left, respectively). The remaining two configurations are those at the end of the second phase of the algorithm (bottom middle), and at the end of multiple calls to RadixSortBodies which sort all column bodies. The elements x that belong to the bodies of C_1 and C_2 in the input configuration are shown in red and yellow, respectively, while their lightnesses represents whether the relevant bit of t-row$(x) - 2$ is set (a brighter color corresponds to a set bit). The configuration on the bottom right colors elements according to their target position.

Indeed, it suffices to arbitrarily partition the n columns into $k = \left\lceil \frac{n}{\lfloor \frac{n}{m-1} \rfloor} \right\rceil$ groups $G_1, G_2, \ldots, G_k$ of $\lfloor \frac{n}{m-1} \rfloor$ columns each, except possibly for G_k (which has size $n \bmod (m-1)$). Then, we iteratively make the columns in each group G_i body-full by invoking RadixSortBodies on the columns in G_i, which requires $O\big((n + \frac{n}{m-1} \cdot m) \cdot \log m\big) = O(n \cdot \log m)$ rotations. Since the number of groups is $O(m)$, the overall number of rotations is $O(mn \cdot \log m)$.
The following lemma summarizes the discussion in this section.

▶ **Lemma 7.** *Given an $m \times n$ input matrix A with $m \leq n$ and in which all columns are body-full, there exists a procedure that performs $O(mn \cdot \log n)$ rotations and rearranges the elements in A so that all columns become body-sorted.*

3.4 Sorting the first row

We now have a sortable matrix where all columns are body-sorted, which implies that R contains all and only the elements x with t-row$(x) = 1$, and we are left with the task of sorting the elements in R. In particular, the arrangement of elements in R can be interpreted as that resulting from the action of some permutation π_R on the set $\{1, \ldots, n\}$. Then, our goal becomes that of finding a sequence of row and column rotations whose net result is that of applying π_R^{-1} to R. This turns out to be more or less challenging depending on the parity of π_R and on that of n.[7]

In the rest of this section we build up towards this goal by first showing a way to apply an involution to R while keeping side effects under control, and then using such result to handle general permutations.

[7] The elements of R, in order, coincide with those of the one-line representation of π_R^{-1}.

Algorithm 4 Procedure SwapPairs.

Input: An $m \times n$ matrix A. A collection $\{(c_1, c'_1), \ldots, (c_k, c'_k)\}$ of pairs of column indices, all distinct. A direction $d \in \{\texttt{U}, \texttt{D}\}$.

Output (in place): The matrix obtained from A by swapping $R[c_h]$ and $R[c'_h]$ and performing a circular shift of the body of C_h in the direction specified by d, for all $h = 1, \ldots, k$.

```
// First phase
for r = 0, ..., n − 1 do
    for j = 1, ..., k do
        if ρ(j, r) = c_j then
            if d = D then rotate C_j once else rotate C_j m − 1 times;
    Rotate R;
// Second phase
for r = 0, ..., n − 1 do
    for j = 1, ..., k do
        if ρ(j, r) = c'_j then
            if d = D then rotate C_j m − 1 times else rotate C_j once;
    Rotate R;
// Third phase
for r = 0, ..., n − 1 do
    for j = 1, ..., k do
        if ρ(j, r) = c'_j then
            if d = D then rotate C_j once else rotate C_j m − 1 times;
    Rotate R;
```

Swapping pairs of elements in R. We start by describing an auxiliary procedure that takes as input a collection of k pairs $(c_1, c'_1), \ldots, (c_k, c'_k)$ of column indices, where all such indices are distinct, and swaps each element in $R[c_j]$ with that in $R[c'_j]$, for all $j = 1, \ldots, k$. We name this procedure SwapPairs and we give its pseudocode in Algorithm 4.

SwapPairs performs $O(nk)$ rotations but produces some side effects. More precisely, the procedure needs to use a distinct auxiliary column for each pair (c_j, c'_j) which, for the sake of simplicity, we chose to be column C_j. The elements *in the body* of each auxiliary column C_j will undergo a circular shift, which can be either upward (i.e., $C_j[i]$ moves to $C_j[i-1]$ for $i = 3, \ldots, m$, and $C_j[2]$ moves to $C_j[m]$) or downward (i.e., $C_j[i]$ moves to $C_j[i+1]$ for $i = 2, \ldots, m-1$, and $C_j[m]$ moves to $C_j[2]$). The direction of these rotations is controlled by an additional input parameter $d \in \{\texttt{U}, \texttt{D}\}$, where `U` (resp. `D`) denotes an upward (resp. downward) rotation.

We note that these side effects are unavoidable as long as the algorithm needs to work for matrices of arbitrary dimensions. Indeed, it is impossible to swap two (or any even number of pairs of) elements without introducing side effects, as this would allow one to change the parity of the permutation induced by positions of the elements in the matrix (see Remark 1).

The procedure works in three phases. In the first phase, row R is rotated n times. Before the generic r-th rotation, we check for which indices $j = 1, \ldots, k$ the element originally in $R[c_j]$ is now in $R[j]$, i.e., i.e., we check whether $\rho(j, r) = c_j$. For any such index j, we rotate

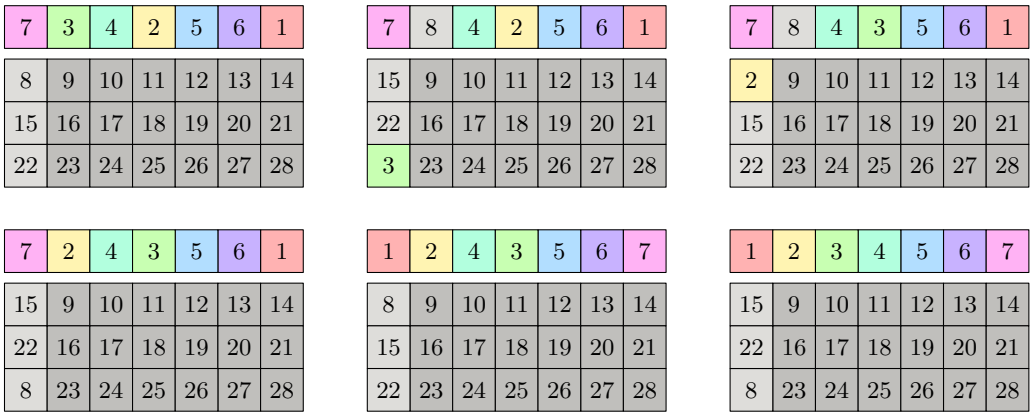

Figure 6 The input configuration to a call to SwapPairs with $d = \texttt{U}$ swapping $R[2]$ and $R[4]$ (top left), which coincides with the last configuration of Figure 5, along with the configurations obtained after the first, second, and third phase (top middle, top right, and bottom left, respectively). Observe how the call causes an upward circular shift of the body of C_1 as a side effect. A further call to SwapPairs with $d = \texttt{D}$ swaps $R[1]$ and $R[7]$ and recovers the original body of C_1 (bottom middle). A final call to SwapPairs with $d = \texttt{U}$ swaps $R[3]$ and $R[4]$, which sorts R but reintroduces the circular shift of the body of C_1 compared to the initial configuration (bottom right). Elements x with t-row$(x) = 1$ are colored according to their target column.

C_j one time if $d = \texttt{D}$ or $m - 1$ times if $d = \texttt{U}$ (effectively simulating an upward rotation of C_j). Denoting with S_1 the sequence of all rotation performed in the first phase, the overall effect of the phase is the following, where $j = 1, \ldots, k$ and all elements that not explicitly handled are unaffected:

- When $d = \texttt{D}$, $R[c_j] \xrightarrow{S_1} C_j[2]$, $C_j[m] \xrightarrow{S_1} R[c_j]$, and $C_j[i] \xrightarrow{S_1} C_j[i+1]$ for $i = 2, \ldots, m-1$;
- When $d = \texttt{U}$, $R[c_j] \xrightarrow{S_1} C_j[m]$, $C_j[2] \xrightarrow{S_1} R[c_j]$, and $C_j[i] \xrightarrow{S_1} C_j[i-1]$ for $i = 3, \ldots, m$.

In the second phase, R is again rotated n times. Before the generic r-th rotation, we find all columns C_j that satisfy $\rho(j, r) = c'_j$, and we rotate each such column $m - 1$ times if $d = \texttt{D}$, or one time if $d = \texttt{U}$. The net effect of the sequence S_2 of rotations performed during the second phase (on the matrix resulting from the first phase) is the following:

- When $d = \texttt{D}$, $R[c'_j] \xrightarrow{S_2} C_j[m]$, $C_j[2] \xrightarrow{S_2} R[c'_j]$, and $C_j[i] \xrightarrow{S_2} C_j[i-1]$ for $i = 3, \ldots, m$.
- When $d = \texttt{U}$, $R[c'_j] \xrightarrow{S_2} C_j[2]$, $C_j[m] \xrightarrow{S_2} R[c'_j]$, and $C_j[i] \xrightarrow{S_2} C_j[i+1]$ for $i = 2, \ldots, m-1$;

The third and final phase is identical to the first one, hence its sequence of rotations S_3 produces the same net effect (on the matrix resulting from the second phase) as that of sequence S_1. Figure 6 shows some example configurations obtained after the different phases, and after successive executions of SwapPairs.

We can now use SwapPairs to apply any involution to R, as the following lemma shows.

▶ **Lemma 8.** *Consider an $m \times n$ matrix A. Given an involution π on $\{1, \ldots, n\}$ and $d \in \{\texttt{U}, \texttt{D}\}$, there exists a procedure that uses $O(a_2(\pi)m + n)$ rotations and produces the following effects: (i) permutes R according to π, and (ii) if $d = \texttt{U}$ (resp. $d = \texttt{D}$) performs a cyclic upward (resp. downward) rotation of the body of each C_j with $j = 1, 2, \ldots, a_2(\pi)$.*

Proof. Let $\tau_1, \tau_2, \ldots, \tau_k$ be the $k = a_2(\pi)$ transpositions of π, where $\tau_j = (c_j\ c'_j)$. We invoke SwapPairs using the same value of d as the one in the statement and with the k pairs of column indices (c_j, c'_j) for $j = 1, \ldots, k$.

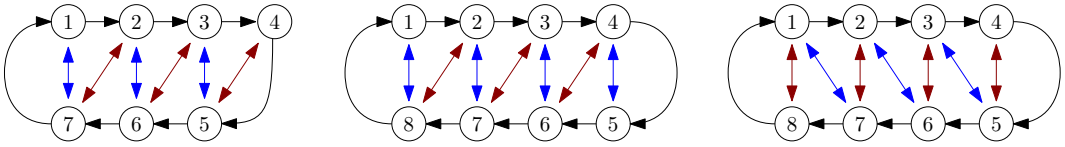

Figure 7 A visualization of the pairs of involutions σ and υ (left), σ^+ and υ^+ (middle), and σ^- and υ^- (right) for even and odd permutation cycles (in black). Involutions σ, σ^+, and σ^- are shown in blue and are to be applied first, while involutions υ, υ^+, and υ^- are shown in red.

We now prove that such a call has the desired effect. To see that each pair of elements $R[c_j], R[c'_j]$ with $j = 1, \ldots, k$ is indeed swapped, observe that:

- When $d = \texttt{D}$, $R[c_j] \xrightarrow{S_1} C_j[2] \xrightarrow{S_2} R[c'_j] \xrightarrow{S_3} R[c'_j]$ and $R[c'_j] \xrightarrow{S_1} R[c'_j] \xrightarrow{S_2} C_j[m] \xrightarrow{S_3} R[c_j]$.
- When $d = \texttt{U}$, $R[c_j] \xrightarrow{S_1} C_j[m] \xrightarrow{S_2} R[c'_j] \xrightarrow{S_3} R[c'_j]$ and $R[c'_j] \xrightarrow{S_1} R[c'_j] \xrightarrow{S_2} C_j[2] \xrightarrow{S_3} R[c_j]$.

To see that the body of each column C_j with $j \in \{1, \ldots, k\}$ undergoes the claimed circular shift, observe that:

- When $d = \texttt{D}$, $C_j[i] \xrightarrow{S_1} C_j[i+1] \xrightarrow{S_2} C_j[i] \xrightarrow{S_3} C_j[i+1]$ for $i = 2, \ldots, m-1$, and $C_j[m] \xrightarrow{S_1} R[c_j] \xrightarrow{S_2} R[c_j] \xrightarrow{S_3} C_j[2]$.
- When $d = \texttt{U}$, $C_j[i] \xrightarrow{S_1} C_j[i-1] \xrightarrow{S_2} C_j[i] \xrightarrow{S_3} C_j[i-1]$ for $i = 3, \ldots, m$, and $C_j[2] \xrightarrow{S_1} R[c_j] \xrightarrow{S_2} R[c_j] \xrightarrow{S_3} C_j[m]$.

To conclude the proof, notice that all elements not explicitly discussed above retain their positions after the rotations in S_1, S_2, and S_3. ◀

Decomposing π_R into involutions and transpositions. We now employ the above result to place all elements in R in their target position, while keeping all columns $C_2, \ldots, C_n$ body sorted, and possibly causing circular shift in the body of C_1.

It is folklore that any permutation π can be written as the product of two involutions (see, e.g., [28]). One can also ensure that, when π is even, these two involutions have the same number of transpositions, while the odd case requires one more transposition. We formally state this result in the following two lemmas, in a way that is convenient for the sequel.

▶ **Lemma 9.** *Let π be a permutation that consists of a single non-trivial cycle. If π is even, then there exists two involutions σ, υ such that $\pi = \sigma\upsilon$ and $a_2(\sigma) = a_2(\upsilon)$. If π is odd, then there exists four involutions σ^+, υ^+, σ^-, υ^- such that $\pi = \sigma^+\upsilon^+ = \sigma^-\upsilon^-$, $a_2(\sigma^+) = a_2(\upsilon^+) + 1$, and $a_2(\sigma^-) = a_2(\upsilon^-) - 1$.*

Proof. Assume w.l.o.g. that $\pi = (1\ 2\ \ldots\ \ell)$. The proof is split in two cases, depending on the parity of π. If π is even, then $\ell = 2k+1$ for some positive integer k. We choose $\sigma = (1\ 2k{+}1)(2\ 2k)(3\ 2k{-}1)\ldots(k\ k{+}2)$, and $\upsilon = (2\ 2k{+}1)(3\ 2k)(4\ 2k{-}1)\ldots(k{+}1\ k{+}2)$. Observe that $a_2(\sigma) = a_2(\upsilon) = k$. An example is shown in Figure 7 (left).

If π is odd, then $\ell = 2k$ for some positive integer k. We choose $\sigma^+ = (1\ 2k)(2\ 2k{-}1)(3\ 2k{-}2)\ldots(k\ k+1)$, and $\upsilon^+ = (2\ 2k)(3\ 2k-1)(4\ 2k-2)\ldots(k\ k+2)$. Observe that $a_2(\sigma^+) = k$ and $a_2(\upsilon^+) = k-1$. An example is shown in Figure 7 (middle). We also choose $\sigma^- = (1\ 2k-1)(2\ 2k-2)(3\ 2k-3)\ldots(k-1\ k{+}1)$ and $\upsilon^- = (1\ 2k)(2\ 2k-1)(3\ 2k-2)\ldots(k\ k{+}1)$. Observe that $a_2(\sigma^-) = k-1$ and $a_2(\upsilon^-) = k$. An example is shown in Figure 7 (right). ◀

▶ **Lemma 10.** *Let π be a permutation. If π is even, then there exist two involutions σ, υ such that $\pi = \sigma\upsilon$, and $a_2(\sigma) = a_2(\upsilon)$. If π is odd, then there exist two involutions σ, υ and a transposition τ such that $\pi = \sigma\upsilon\tau$, and $a_2(\sigma) = a_2(\upsilon)$.*

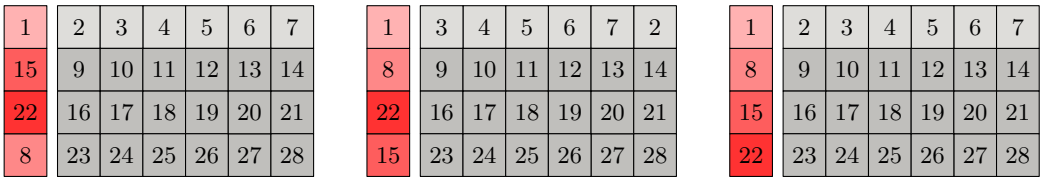

Figure 8 The configuration resulting from applying $\sigma = (2\ 4)$, $\upsilon = (1\ 7)$, and $\tau = (3\ 4)$ to R starting from the configuration in the top left of Figure 6 (top left). Observe how the swaps implementing σ, υ, and τ are exactly those shown in Figure 6, and how the body of C_1 undergoes a circular shift. The correct order of the elements in C_1 is restored by two consecutive calls to SwapPairs (on the transposed matrix) applying involutions $\sigma_1 = (2\ 4)$ (top middle) and $\upsilon_1 = (3\ 4)$ (top right). These latter two calls first introduce and then undo a circular shift of $R[2], R[3], \ldots, R[n]$. The elements in C_1 are colored in red, where different lightnesses correspond to different target rows. A small gap has been added to the right of column C_1 for improved visual clarity.

Proof. Let $C_1^{\text{even}}, C_2^{\text{even}}, \ldots C_h^{\text{even}}$ and $C_1^{\text{odd}}, C_2^{\text{odd}}, \ldots, C_k^{\text{odd}}$ be the non-trivial even and odd cycles of π, respectively. Lemma 9 shows that we can write each even cycle C_i^{even} as $C_i^{\text{even}} = \sigma_i \upsilon_i$, where σ_i and υ_i are two involutions with the same number of transpositions. We let $\sigma^{\text{even}} = \sigma_1\sigma_2\ldots\sigma_h$ and $\upsilon^{\text{even}} = \upsilon_1\upsilon_2\ldots\upsilon_h$. Similarly, Lemma 9 shows that we can write each odd cycle C_i^{odd} as both $C_i^{\text{odd}} = \sigma_i^+\upsilon_i^+$ and $C_i^{\text{odd}} = \sigma_i^-\upsilon_i^-$, where each of σ_i^+, υ_i^+, σ_i^-, and υ_i^- is an involution, $a_2(\sigma_i^+) = a_2(\upsilon_i^+) + 1$, and $a_2(\sigma_i^-) = a_2(\upsilon_i^-) - 1$.

The rest of the proof depends on the parity of π. We consider the case in which π is even first, which implies that k is also even. We let $\sigma^{\text{odd}} = \sigma_1^-\sigma_2^+\sigma_3^-\sigma_4^+\sigma_5^-\ldots\sigma_k^+$ and $\upsilon^{\text{odd}} = \upsilon_1^-\upsilon_2^+\upsilon_3^-\upsilon_4^+\upsilon_5^-\ldots\upsilon_k^+$ (notice the alternating superscripts). The claim follows by choosing $\sigma = \sigma^{\text{even}}\sigma^{\text{odd}}$ and $\upsilon = \upsilon^{\text{even}}\upsilon^{\text{odd}}$.

We now consider the case in which π is odd, which implies that k is also odd. We let τ be an arbitrary transposition of υ_k^-, and we define $\upsilon_k' = \upsilon_k^-\tau$ as the involution obtained by removing τ from υ_k^- (observe that τ is its own inverse), so that $a_2(\sigma_k^-) = a_2(\upsilon_k^-) - 1 = a_2(\upsilon_k')$. We let $\sigma^{\text{odd}} = \sigma_1^-\sigma_2^+\sigma_3^-\sigma_4^+\sigma_5^-\ldots\sigma_{k-1}^+\sigma_k^-$ and $\upsilon^{\text{odd}} = \upsilon_1^-\upsilon_2^+\upsilon_3^-\upsilon_4^+\upsilon_5^-\ldots\upsilon_{k-1}^+\upsilon_k'$ (notice the alternating superscripts). The claim follows by choosing $\sigma = \sigma^{\text{even}}\sigma^{\text{odd}}$, $\upsilon = \upsilon^{\text{even}}\upsilon^{\text{odd}}$, and τ as defined above. ◀

Applying π_R^{-1} to R. We can finally show how the elements in R can be sorted, i.e., permuted according to π_R^{-1}. We start with a simplification: we assume w.l.o.g. that if π_R is odd, then m is even. Indeed, the case in which π_R is odd and m is odd can be avoided by noticing that, whenever this happens, n must be even since otherwise the matrix would not be sortable (see Remark 1). Hence, we can reduce to the case in which π_R is even by rotating R.

We proceed differently depending on the parity of π_R. If π_R is even, we use Lemma 10 to write π_R^{-1} as the product $\sigma\upsilon$ of two involutions σ, υ with $a_2(\sigma) = a_2(\upsilon)$. Then we execute procedure SwapPairs twice: the first call applies σ to R and uses $d = \texttt{U}$, while the second applies υ to the resulting configuration and uses $d = \texttt{D}$. Invoking Lemma 8 twice shows that all rotations caused to the column bodies cancel out, while the elements originally in R get permuted according to $\pi_R^{-1} = \sigma\upsilon$, i.e., they end up in the sorted configuration.

The only remaining case is the one in which π_R is odd and m is even. In this case, we use Lemma 10 to write π_R^{-1} as the product $\sigma\upsilon\tau$ of two involutions σ, υ with $a_2(\sigma) = a_2(\upsilon)$, and a transposition τ. We call procedure SwapPairs five times. The first two calls apply involutions σ and υ to R, in this order, with $d = \texttt{U}$ and $d = \texttt{D}$ respectively. Using Lemma 8 twice, we have that the combined effect of these two calls to SwapPairs leaves the positions

of the elements not in R unaffected. The third call swaps the elements involved in the transposition τ, while causing a circular shift in body of C_1 (the direction of such shift is not important). Since m is even, the number of elements in the body of C_1 is odd, and the permutation π_1 describing their circular shift is even, hence it can be written as the product $\sigma_1\tau_1$ of two involutions $\sigma_1\tau_1$ with the same number of transpositions. Since we can equivalently consider the transposed version A^T of matrix A, we use the fourth and fifth calls to SwapPairs to apply σ_1 and τ_1, in this order, to the first row of A^T, i.e., to column C_1 of A. By using opposite values of d in these two final calls, we ensure any side effects introduced by the fourth call are undone by the fifth and last call. An example is provided in Figure 8. The above discussion is summarized in the following lemma.

▶ **Lemma 11.** *Given an $m \times n$ matrix A in which each column is body-sorted, there exists a procedure that performs $O(mn)$ rotations and sorts A.*

Algorithm 5 shows the final pseudocode for solving the Torus Puzzle when $m \leq n$ (which can be assumed w.l.o.g.), where lines from 6 onward handle sorting R as described above. The main result of this paper follows from combining Corollary 5 and Lemmas 6, 7, and 11.

▶ **Theorem 12.** *There exists an algorithm that, given a sortable $m \times n$ instance of the Torus Puzzle, sorts it by performing $O(mn \cdot \log \max\{m, n\})$ rotations, where each rotation is either a unit downward rotation of a column or unit rightward rotation of a row.*

4 Concluding remarks and open problems

Remarks on the time-complexity of our algorithm. An implementation of the procedures used in our algorithm which closely follows the provided pseudocodes can result in a running time that is larger than our $O(mn \cdot \log \max\{m, n\})$ upper bound on the push number.

Here, we briefly sketch how the algorithm can be implemented to also run in time $O(mn \cdot \log \max\{m, n\})$. We start by observing that, when $m \leq n$, the only row rotated by procedures FillColumns, FloatMinimums, RadixSortBodies, and SwapPairs is R. Then, a data structure that maintains the current configuration of A while allowing constant-time element access, and constant-time (not necessarily unit) rotations of R and $C_1, \ldots, C_j$ can be obtained by storing R and each of the column bodies in circular arrays equipped with pointers to their first elements. Moreover, for a given column C_j and at any point during the execution of FillColumns, FloatMinimums, each (sub-)phase of RadixSortBodies, and each of the three phases of SwapPairs, it is easy to know when the next rotation of C_j will occur. The idea is that of introducing a concept of time, which is determined by the number of rotations of R performed so far, while maintaining a min-heap H storing each of column index j with a key equal to the time of the next rotation of C_j (if any). Since the next rotation of a column (if it exists) is never more than n time-instants away, H can be implemented to support insertions in constant-time, and extractions of the minimum in time proportional to the number of time-instants between the current time and the one given by the key of the extracted element. With these ingredients, all the above procedures can be implemented to run in the same asymptotic time as their respective upper bounds on the number of rotations, and the same follows for the whole sorting algorithm (which, for $m \leq n$, repeatedly invokes these procedure while transposing A at most once).

Reducing the number of distinct rotation types. When our algorithm is executed on an $m \times n$ input matrix with $m \leq n$, all rotations involve R or the columns $C_1, \ldots, C_n$ up until the very end, when SwapPairs might be invoked to apply the permutation π_1 to C_1 (i.e., to the first row of the transposed version of A). This last step may rotate rows other than R.

Algorithm 5 Algorithm for solving the Torus Puzzle.

Input: A sortable $m \times n$ matrix A with $m \leq n$ (consider A^T for $m > n$).
Output (in place): The sorted version of A.

1 Call FillColumns on A;
2 Call FloatMinimums on A;
3 $k \leftarrow \lfloor \frac{n}{m-1} \rfloor$;
4 **for** $h = 0, 1, \ldots, \lceil \frac{n}{k} \rceil - 1$ **do**
5 Call RadixSortBodies on columns $hk+1, hk+2, \ldots, \min\{(h+1)k, n\}$;
6 **if** *n is even $\wedge$ the permutation induced by R is odd* **then**
7 Rotate R;
8 $\pi_R \leftarrow$ permutation induced by R;
9 **if** *π_R is even* **then**
10 $\sigma, \upsilon \leftarrow$ Two involutions s.t. $\pi_R^{-1} = \sigma\upsilon$ and $a_2(\sigma) = a_2(\upsilon)$;
11 Call SwapPairs with $d = \texttt{U}$ to apply σ to R;
12 Call SwapPairs with $d = \texttt{D}$ to apply υ to R;
13 **else**
14 $\sigma, \upsilon, \tau \leftarrow$ Two involutions and a transposition s.t. $\pi_R^{-1} = \sigma\upsilon\tau$ and $a_2(\sigma) = a_2(\upsilon)$;
15 Call SwapPairs with $d = \texttt{U}$ to apply σ to R;
16 Call SwapPairs with $d = \texttt{D}$ to apply υ to R;
17 Call SwapPairs with $d = \texttt{U}$ to apply τ to R;
18 $\pi_1 \leftarrow$ permutation induced by C_1;
19 $\sigma_1, \upsilon_1 \leftarrow$ Two involutions s.t. $\pi_1^{-1} = \sigma_1\upsilon_1$ and $a_2(\sigma_1) = a_2(\upsilon_1)$;
20 Call SwapPairs with $d = \texttt{U}$ to apply σ_1 to C_1;
21 Call SwapPairs with $d = \texttt{D}$ to apply υ_1 to C_1;

One might wonder whether it is possible to avoid rotating these rows, since doing so would result in an algorithm that sorts any sortable matrix using no more than $m + 1$ distinct rotation types, which is also a lower bound.[8] This can be by using the following alternative handing of the case in which π_R is odd (i.e., the "else" branch starting at line 13 of Algorithm 5): after decomposing π_R^{-1} into the product $\sigma\upsilon\tau$, call SwapPairs with alternating choices of d to apply σ, υ, τ, and $(1\ n)$ to R, in this order. This results in a matrix in which every element is in its target position, except for $R[1]$ and $R[n]$, which need to be swapped. In the full version of the paper, we show a sequence of $O(mn)$ unit rotation that performs exactly this task when m is even (as in the case of interest), while only rotating R and C_1.

Handling arbitrary rotation directions. While the description of our algorithm assumes that all columns can only be rotated downwards, it is easy to check that all our procedures work with either no or with minor modifications even when each column can only be rotated in a direction specified as part of the input (where different columns can have different

[8] To see that m rotation types are not enough in general, notice that at least one row and at least one column must necessarily be available for rotations. Then, for any such choice of m rotations types, there is always a row-column pair such that neither the row nor the column can be rotated, which implies that the element on their intersection cannot be moved from its initial position.

directions). This implies that our upper bound holds even when both rows and columns have input-specified rotation directions. The easiest way to see this is that of combining the previous discussion showing that the only row that needs to be rotated is R, with symmetry arguments which allow to equivalently consider a transposed or mirrored version of A.[9]

Open problems. Following our results, the known lower and upper bounds on the push number differ by a multiplicative $\Theta(\log \max\{m, n\})$ factor. It would be interesting to close this gap by providing asymptotically tight bounds.

Moreover, to the best of our knowledge, it is still unknown whether the optimization version of the Torus Puzzle (that asks to return the shortest sequence of rotations sorting the input matrix) is solvable in polynomial time, both for the case unit rotations and for the more permissive case of compound rotations. Likewise, no polynomial-time approximation algorithm with an approximation factor better than the trivial $\widetilde{O}(mn)$ is currently known.

References

1 Jeremy F. Alm, Michael Gramelspacher, and Theodore Rice. Rubik's on the torus. *Am. Math. Mon.*, 120(2):150–160, 2013. doi:10.4169/AMER.MATH.MONTHLY.120.02.150.

2 Kazuyuki Amano, Yuta Kojima, Toshiya Kurabayashi, Keita Kurihara, Masahiro Nakamura, Ayaka Omi, Toshiyuki Tanaka, and Koichi Yamazaki. How to solve the torus puzzle. *Algorithms*, 5(1):18–29, 2012. doi:10.3390/A5010018.

3 Aaron F Archer. A modern treatment of the 15 puzzle. *The American Mathematical Monthly*, 106(9):793–799, 1999.

4 Davide Bilò, Maurizio Fiusco, Luciano Gualà, and Stefano Leucci. Swapping mixed-up beers to keep them cool. In Andrei Z. Broder and Tami Tamir, editors, *12th International Conference on Fun with Algorithms, FUN 2024, Island of La Maddalena, Sardinia, Italy, June 4-8, 2024*, volume 291 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.FUN.2024.5.

5 Olena Blumberg, Ben Morris, and Alto Senda. Mixing time of the torus shuffle. *arXiv preprint arXiv:2411.06006*, 2024.

6 Alex Bogomolny. Cut The Knot! — Slider Puzzles. Webpage, 1997. Accessed on 2025-12-29. URL: https://www.cut-the-knot.org/pythagoras/sliders.shtml.

7 Erik D. Demaine, Martin L. Demaine, Sarah Eisenstat, Anna Lubiw, and Andrew Winslow. Algorithms for Solving Rubik's Cubes. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, volume 6942 of *Lecture Notes in Computer Science*, pages 689–700. Springer, 2011. doi:10.1007/978-3-642-23719-5_58.

8 Erik D. Demaine, Sarah Eisenstat, and Mikhail Rudoy. Solving the Rubik's Cube Optimally is NP-complete. In Rolf Niedermeier and Brigitte Vallée, editors, *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, Caen, France, February 28 - March 3, 2018*, volume 96 of *LIPIcs*, pages 24:1–24:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.STACS.2018.24.

9 Erik D. Demaine and Mikhail Rudoy. A simple proof that the $(n^2 - 1)$-puzzle is hard. *Theor. Comput. Sci.*, 732:80–84, 2018. doi:10.1016/J.TCS.2018.04.031.

10 Persi Diaconis. Group representations in probability and statistics. *Lecture notes-monograph series*, 11:i–192, 1988.

[9] In this way, one can always assume to be dealing with a matrix with at least as many columns as rows, and where unit rightward rotations of the first row are allowed.

11 Sebastian Egner and Markus Püschel. Solving puzzles related to permutation groups. In Volker Weispfenning and Barry M. Trager, editors, *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation, ISSAC '98, Rostock, Germany, August 13-15, 1998*, pages 186–193. ACM, 1998. doi:10.1145/281508.281611.

12 Oded Goldreich. Finding the shortest move-sequence in the graph-generalized 15-puzzle is np-hard. In Oded Goldreich, editor, *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*, volume 6650 of *Lecture Notes in Computer Science*, pages 1–5. Springer, 2011. doi:10.1007/978-3-642-22670-0_1.

13 Donald Lee Greenwell. Cut The Knot! — Sliders, puzzles on graphs. Webpage, 1999. Accessed on 2025-12-29. URL: https://www.cut-the-knot.org/pythagoras/SLIDER.shtml.

14 Luciano Gualà, Stefano Leucci, and Emanuele Natale. Bejeweled, candy crush and other match-three games are (np-)hard. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014, Dortmund, Germany, August 26-29, 2014*, pages 1–8. IEEE, 2014. doi:10.1109/CIG.2014.6932866.

15 Colin Hemphill and Joshua Sheehy. Finding effective search strategies for the twobik puzzle. In Randy K. Smith and Susan V. Vrbsky, editors, *Proceedings of the 50th Annual Southeast Regional Conference, 2012, Tuscaloosa, AL, USA, March 29-31, 2012*, pages 53–58. ACM, 2012. doi:10.1145/2184512.2184526.

16 Sam Hiken and Nicole Wein. Improved hardness-of-approximation for token-swapping. In Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *33rd Annual European Symposium on Algorithms, ESA 2025, Warsaw, Poland, September 15-17, 2025*, volume 351 of *LIPIcs*, pages 57:1–57:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ESA.2025.57.

17 Jesper Juul. Swap adjacent gems to make sets of three: A history of matching tile games. *Artifact: Journal of Design Practice*, 1(4):205–217, 2007.

18 Andrew A. Kinsman. Re: Twist torus [small picture]. Message posted to the MIT's CubeLovers mailing list, December 1995. Posted as "Andy Kinsman 66672"; Message ID: 9512151350.AA20112@newt.PCD1. URL: https://cube20.org/cubelovers/CL18/036.txt.

19 Serge Kornfeld. Re twist torus. Message posted to the MIT's CubeLovers mailing list, December 1995. Message ID: 199512141816.NAA28270@spank.nexen.com. URL: https://cube20.org/cubelovers/CL18/031.txt.

20 Daniel Kornhauser, Gary L. Miller, and Paul G. Spirakis. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*, pages 241–250. IEEE Computer Society, 1984. doi:10.1109/SFCS.1984.715921.

21 Jamie Mulholland. Permutation puzzles: a mathematical perspective. *Departement Of mathematics Simon fraser University*, 54, 2016.

22 Win Hlaing Hlaing Myint, Ryuhei Uehara, and Giovanni Viglietta. Token shifting on graphs. *Int. J. Comput. Math. Comput. Syst. Theory*, 7(4):223–234, 2022. doi:10.1080/23799927.2022.2118622.

23 Takeshi Ogihara. Personal communication, January 2026.

24 Daniel Ratner and Manfred K. Warmuth. Finding a shortest solution for the N × N extension of the 15-puzzle is intractable. In Tom Kehler, editor, *Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, USA, August 11-15, 1986. Volume 1: Science*, pages 168–172. Morgan Kaufmann, 1986. URL: http://www.aaai.org/Library/AAAI/1986/aaai86-027.php.

25 Walter P. Smith. Twist torus. Message posted to the MIT's CubeLovers mailing list, December 1995. Message ID: 199512141525.KAA05516@homer.MCLN.Federal.Unisys.COM. URL: https://cube20.org/cubelovers/CL18/029.txt.

26 Richard M Wilson. Graph puzzles, homotopy, and the alternating group. *Journal of Combinatorial Theory, Series B*, 16(1):86–96, 1974.

27 Chao Yang. Sliding puzzles and rotating puzzles on graphs. *Discret. Math.*, 311(14):1290–1294, 2011. doi:10.1016/J.DISC.2011.03.011.

28 Qingxuan Yang, John A. Ellis, Khalegh Mamakani, and Frank Ruskey. In-place permuting and perfect shuffling using involutions. *Inf. Process. Lett.*, 113(10-11):386–391, 2013. doi:10.1016/J.IPL.2013.02.017.