Quantum routing with fast reversals

Aniruddha Bapat[1,4], Andrew M. Childs[1,2,3], Alexey V. Gorshkov[1,4], Samuel King[5], Eddie Schoute[1,2,3], and Hrishee Shastri[6]

[1]Joint Center for Quantum Information and Computer Science, NIST/University of Maryland, College Park, Maryland 20742, USA
[2]Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742, USA
[3]Department of Computer Science, University of Maryland, College Park, Maryland 20742, USA
[4]Joint Quantum Institute, NIST/University of Maryland, College Park, Maryland 20742, USA
[5]University of Rochester, Rochester, New York 14627, USA
[6]Reed College, Portland, Oregon 97202, USA

We present methods for implementing arbitrary permutations of qubits under interaction constraints. Our protocols make use of previous methods for rapidly reversing the order of qubits along a path. Given nearest-neighbor interactions on a path of length n, we show that there exists a constant $\epsilon \approx 0.034$ such that the quantum routing time is at most $(1-\epsilon)n$, whereas any SWAP-based protocol needs at least time $n-1$. This represents the first known quantum advantage over SWAP-based routing methods and also gives improved quantum routing times for realistic architectures such as grids. Furthermore, we show that our algorithm approaches a quantum routing time of $2n/3$ in expectation for uniformly random permutations, whereas SWAP-based protocols require time n asymptotically. Additionally, we consider sparse permutations that route $k \leq n$ qubits and give algorithms with quantum routing time at most $n/3 + O(k^2)$ on paths and at most $2r/3 + O(k^2)$ on general graphs with radius r.

1 Introduction

Qubit connectivity limits quantum information transfer, which is a fundamental task for quantum computing. While the common model for quantum computation usually assumes all-to-all connectivity, proposals for scalable quantum architectures do not have this capability [MK13; Mon+14; Bre+16]. Instead, quantum devices arrange qubits in a fixed architecture that fits within engineering and design constraints. For example, the architecture may be grid-like [MG19; Aru+19] or consist of a network of submodules [MK13; Mon+14]. Circuits that assume all-to-all qubit connectivity can be mapped onto these architectures via protocols for *routing* qubits, i.e., permuting them within the architecture using local operations.

Aniruddha Bapat: ani@umd.edu
Andrew M. Childs: amchilds@umd.edu
Alexey V. Gorshkov: gorshkov@umd.edu
Eddie Schoute: eschoute@umd.edu

Long-distance gates can be implemented using SWAP gates along edges of the graph of available interactions. A typical procedure swaps pairs of distant qubits along edges until they are adjacent, at which point the desired two-qubit gate is applied on the target qubits. These swap subroutines can be sped up by parallelism and careful scheduling [SWD11; SSP13; SSP14; PS16; LWD15; Mur+19; ZW19]. Minimizing the SWAP circuit depth corresponds to the ROUTING VIA MATCHINGS problem [ACG94; CSU19]. The minimal SWAP circuit depth to implement any permutation on a graph G is given by its *routing number*, $\mathrm{rt}(G)$ [ACG94]. Deciding $\mathrm{rt}(G)$ is generally NP-hard [BR17], but there exist algorithms for architectures of interest such as grids and other graph products [ACG94; Zha99; CSU19]. Furthermore, one can establish lower bounds on the routing number as a function of graph diameter and other properties.

Routing using SWAP gates does not necessarily give minimal circuit evolution time since it is effectively classical and does not make use of the full power of quantum operations. Indeed, faster protocols are already known for specific permutations in specific qubit geometries such as the path [Rau05; Bap+20]. These protocols tend to be carefully engineered and do not generalize readily to other permutations, leaving open the general question of devising faster-than-SWAP quantum routing. In this paper, we give a positive answer to this question.

Following [Rau05; Bap+20], we consider a continuous-time model of routing, where the protocol is defined by a Hamiltonian that can only include nearest-neighbor interactions. To make consistent comparisons with a gate-based model of routing, we bound the spectral norm of interactions [Bap+20] so that a SWAP gate takes unit time [VHC02], as determined by the canonical form of a two-qubit Hamiltonian [Ben+02]. We suppose that single-qubit operations can be performed arbitrarily fast, a common assumption [VHC02; Ben+02] that is practically well-motivated due to the relative ease of implementing single-qubit rotations.

Rather than directly engineering a quantum routing protocol, we consider a hybrid strategy that leverages a known protocol for quickly performing a specific permutation to implement general quantum routing. Specifically, we consider the reversal operation

$$\rho := \prod_{k=1}^{\lfloor \frac{n}{2} \rfloor} \mathrm{SWAP}_{k,n+1-k} \tag{1}$$

that swaps the positions of qubits about the center of a length-n path. Fast quantum reversal protocols are known in the gate-based [Rau05] and time-independent Hamiltonian [Bap+20] settings. The reversal operation can be implemented in time [Bap+20]

$$T(\rho) \le \frac{\sqrt{(n+1)^2 - p(n)}}{3} \le \frac{n+1}{3}, \tag{2}$$

where $p(n) \in \{0,1\}$ is the parity of n. Both protocols exhibit an asymptotic time scaling of $n/3 + O(1)$, which is asymptotically three times faster than the best possible SWAP-based time of $n-1$ (bounded by the diameter of the graph) [ACG94]. The odd-even sort algorithm provides a nearly tight time upper bound of n [LDM84] and will be our main point of comparison.

The Hamiltonian protocol of [Bap+20] can be understood by looking at the time evolution of the site Majorana operators obtained by a Jordan-Wigner transformation of the spin chain. In this picture, the protocol can be interpreted as the rotation of a fictitious particle of spin $n+1/2$ whose magnetization components are in one-to-one correspondence with the Majoranas on the chain. A reversal corresponds to a rotation of the large spin by an angle of π. The gate-based reversal protocol [Rau05] is a special case of a quantum cellular automaton with a transition function given by the $(n+1)$-fold product of nearest-neighbor controlled-Z (CZ) operations—an operation that can be done 3 times faster than a SWAP gate—and Hadamard operations. In an open spin chain, this process spreads out local Pauli observables at site i over the chain and "refocuses" them at site $n+1-i$ in $n+1$ steps for every i. The ability to spread local observables (which is present in the gate-based and Hamiltonian protocols but not in SWAP-based protocols) may be key to obtaining a speedup over SWAP-based algorithms.

We expect both the gate-based and Hamiltonian protocols to be implementable on near-term quantum devices. The gate-based protocol uses nearest-neighbor CZ gates and Hadamard gates, both of which are widely used on existing quantum platforms. The Hamiltonian protocol involves nearest-neighbor Pauli XX interactions with non-uniform couplings, which is within the capabilities of, e.g., superconducting architectures [Kja+20].

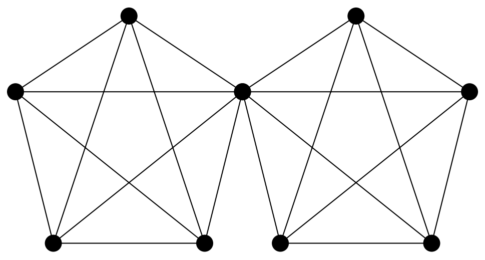
Routing using reversals has been studied extensively due to its applications in comparative genomics (where it is known as *sorting by reversals*) [BP93; KS95]. References [Ben+08; PS02; NNN05] present routing algorithms where, much like in our case, reversals have length-weighted costs. However, these models assume reversals are performed sequentially, while we assume independent reversals can be performed in parallel, where the total cost is given by the evolution time, akin to circuit depth. To our knowledge, results from the sequential case are not easily adaptable to the parallel setting and require a different approach.

Routing on paths is a fundamental building block for routing on more general graphs. For example, a two-dimensional grid graph is the Cartesian product of two path graphs, and the best known routing routine applies a path routing subroutine 3 times [ACG94]. A quantum protocol for routing on the path in time cn, for a constant $c > 0$, would imply a routing time of $3cn$ on the grid. A similar speedup follows for higher-dimensional grids. More generally, routing algorithms for the *generalized hierarchical product* of graphs can take advantage of faster routing of the path base graph [CSU19]. For other graphs, it is open whether fast reversals can be used to give faster routing protocols for general permutations.

In the rest of this paper, we present the following results on quantum routing using fast reversals. In Section 2, we give basic examples of using fast reversals to perform routing on general graphs to indicate the extent of possible speedup over SWAP-based routing, namely a graph for which routing can be sped up by a factor of 3, and another for which no speedup is possible. Section 3 presents algorithms for routing sparse permutations, where few qubits are routed, both for paths and for more general graphs. Here, we obtain the full factor 3 speedup over SWAP-based routing. Then, in Section 4, we prove the main result that there is a quantum routing algorithm for the path with worst-case constant-factor advantage over any SWAP-based routing scheme. Finally, in Section 5, we show that our algorithm has average-case routing time $2n/3 + o(n)$ and any SWAP-based protocol has average-case routing time at least $n - o(n)$.

2 Simple bounds on routing using reversals

Given the ability to implement a fast reversal ρ with cost given by Eq. (2), the largest possible asymptotic speedup of reversal-based routing over SWAP-based routing is a factor of 3. This is because the reversal operation, which is a particular permutation, cannot be performed faster than $n/3 + o(n)$, and can be performed in time n classically using odd-even sort. As we now show, some graphs can saturate the factor of 3 speedup for general permutations, while other graphs do not


(a) Joined graph K_9^*.

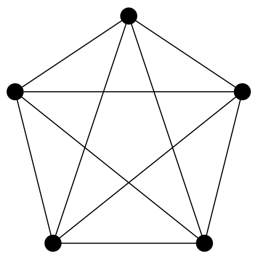

(b) Complete graph K_5.

Figure 1: K_9^* admits the full factor of 3 speedup in the worst case when using reversals over SWAPs, whereas K_5 admits no speedup when using reversals over SWAPs.

admit any speedup over SWAPs.

Maximal speedup: For n odd, let K_n^* denote two complete graphs, each on $(n+1)/2$ vertices, joined at a single "junction" vertex for a total of n vertices (Figure 1a). Consider a permutation on K_n^* in which every vertex is sent to the other complete subgraph, except that the junction vertex is sent to itself. To route with SWAPs, note that each vertex (other than that at the junction) must be moved to the junction at least once, and only one vertex can be moved there at any time. Because there are $(n+1)/2-1$ non-junction vertices on each subgraph, implementing this permutation requires a SWAP-circuit depth of at least $n-1$.

On the other hand, any permutation on K_n^* can be implemented in time $n/3+O(1)$ using reversals. First, perform a reversal on a path that connects all vertices with opposite-side destinations. After this reversal, every vertex is on the side of its destination and the remainder can be routed in at most 2 steps [ACG94]. The total time is at most $(n+1)/3+2$, exhibiting the maximal speedup by an asymptotic factor of 3.

No speedup: Now, consider the complete graph on n vertices, K_n (Figure 1b). Every permutation on K_n can be routed in at most time 2 using SWAPs [ACG94]. Consider implementing a 3-cycle on three vertices of K_n for $n \geq 3$ using reversals. Any reversal sequence that implements this permutation will take at least time 2. Therefore, no speedup is gained over SWAPs in the worst case.

We have shown that there exists a family of graphs that allows a factor of 3 speedup for any permutation when using fast reversals instead of SWAPs, and others where reversals do not grant any improvement. The question remains as to where the path graph lies on this spectrum. Faster routing on the path is especially desirable since this task is fundamental for routing in more complex graphs.

3 An algorithm for sparse permutations

We now consider routing sparse permutations, where only a small number k of qubits are to be moved. For the path, we show that the routing time is at most $n/3+O(k^2)$. More generally, we show that for a graph of radius r, the routing time is at most $2r/3+O(k^2)$. (Recall that the radius of a graph $G=(V,E)$ is $\min_{u\in V}\max_{v\in V}\operatorname{dist}(u,v)$, where $\operatorname{dist}(u,v)$ is the distance between u and v

```
    Input   : $\pi$, a permutation
 1  function MiddleExchange($\pi$):
 2      identify the labels $x_1, \ldots, x_k \in [n]$ to be permuted, with $x_i < x_{i+1}$
 3      let $t$ be the largest index for which $x_t \leq \lfloor n/2 \rfloor$, i.e., the last label $x_t$ left of the median
 4      for $i = 1$ to $t - 1$ :
 5          perform $\rho(x_i - i + 1, x_{i+1} - 1)$
 6      for $j = k$ to $t + 2$ :
 7          perform $\rho(x_j + k - j, x_{j-1} + 1)$
 8      perform $\rho(x_t - t + 1, \lfloor n/2 \rfloor)$
 9      perform $\rho(x_{t+1} + k - t - 1, \lfloor n/2 \rfloor + 1)$
10      $\bar{\rho} \leftarrow$ the sequence of all reversals so far
11      route the labels $x_1, \ldots, x_k$ such that after performing $\bar{\rho}$ in reverse order, each label is at
          its destination
12      perform $\bar{\rho}$ in reverse order
```

Algorithm 3.1: MiddleExchange algorithm to sort sparse permutations on the path graph. We let $\rho(i, j)$ denote a reversal on the segment starting at i and ending at j, inclusive.

in G.) Our approach to routing sparse permutations using reversals is based on the idea of bringing all k qubits to be permuted to the center of the graph, rearranging them, and then sending them to their respective destinations.

3.1 Paths

A description of the algorithm on the path, called `MiddleExchange`, appears in Algorithm 3.1. Figure 2 presents an example of `MiddleExchange` for $k = 6$.

In Theorem 3.1, we prove that Algorithm 3.1 achieves a routing time of asymptotically $n/3$ when implementing a sparse permutation of $k = o(\sqrt{n})$ qubits on the path graph. First, let $\mathcal{S}_n$ denote the set of permutations on $\{1, \ldots, n\}$, so $|\mathcal{S}_n| = n!$. Then, for any permutation $\pi \in \mathcal{S}_n$ that acts on a set of labels $\{1, \ldots, n\}$, let π_i denote the destination of label i under π. We may then write $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$. Let $\bar{\rho}$ denote an ordered series of reversals $\rho_1, \ldots, \rho_m$, and let $\bar{\rho}_1 +\!\!+ \bar{\rho}_2$ be the concatenation of two reversal series. Finally, let $S \cdot \rho$ and $S \cdot \bar{\rho}$ denote the result of applying ρ and $\bar{\rho}$ to a sequence S, respectively, and let $|\rho|$ denote the length of the reversal ρ, i.e., the number of vertices it acts on.

Theorem 3.1. *Let $\pi \in \mathcal{S}_n$ with $k = |\{x \in [n] \mid \pi_x \neq x\}|$ (i.e., k elements are to be permuted, and $n-k$ elements begin at their destination). Then Algorithm 3.1 routes π in time at most $n/3 + O(k^2)$.*

Proof. Algorithm 3.1 consists of three steps: compression (Line 4–Line 9), inner permutation (Line 11), and dilation (Line 12). Notice that compression and dilation are inverses of each other.

Let us first show that Algorithm 3.1 routes π correctly. Just as in the algorithm, let $x_1, \ldots, x_k$ denote the labels $x \in [n]$ with $x_i < x_{i+1}$ such that $\pi_x \neq x$, that is, the elements that do not begin at their destination and need to be permuted. It is easy to see that these elements are permuted correctly: After compression, the inner permutation step routes x_i to the current location of the label π_{x_i} in the middle. Because dilation is the inverse of compression, it will then route every x_i to its correct destination. For the non-permuting labels, notice that they lie in the support of

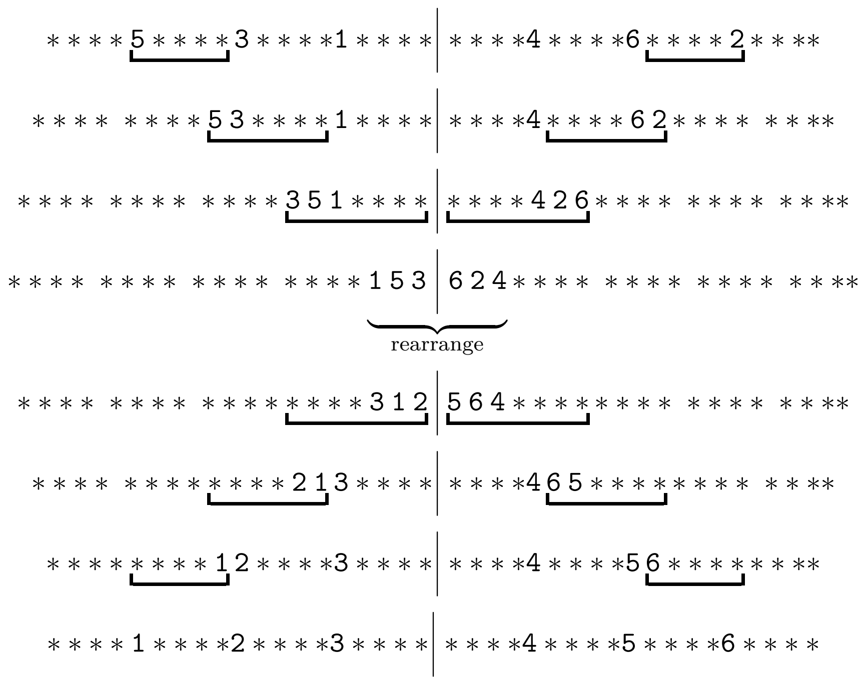


Figure 2: Example of `MiddleExchange` (Algorithm 3.1) on the path for $k = 6$.

either no reversal or exactly two reversals, ρ_1 in the compression step and ρ_2 in the dilation step. Therefore ρ_1 reverses the segment containing the label and ρ_2 re-reverses it back into place (so $\rho_1 = \rho_2$). Therefore, the labels that are not to be permuted end up exactly where they started once the algorithm is complete.

Now we analyze the routing time. Let $d_i = x_{i+1} - x_i - 1$ for $i \in [k-1]$. As in the algorithm, let t be the largest index for which $x_t \le \lfloor n/2 \rfloor$. Then, for $1 \le i \le t-1$, we have $|\rho_i| = d_i + i$, and, for $t+2 \le j \le k$, we have $|\rho_j| = d_{j-1} + k - j$. Moreover, we have $|\rho_t| = \lfloor n/2 \rfloor - x_t - 1 + t$ and $|\rho_{t+1}| = x_{t+1} - \lfloor n/2 \rfloor + k - t$. From all reversals in the first part of Algorithm 3.1, $\bar{\rho}$, consider those that are performed on the left side of the median (position $\lfloor n/2 \rfloor$ of the path). The routing time of these reversals is

$$\begin{aligned}
\frac{1}{3}\sum_{i=1}^{t} |\rho_i| + 1 &= \frac{1}{3}\left(\lfloor n/2 \rfloor - x_t - 1\right) + \frac{1}{3}\sum_{i=1}^{t}\left(d_i + i + 1\right) \\
&= \frac{t(t+1)}{6} + \frac{1}{3}\left(\lfloor n/2 \rfloor - x_t - 1\right) + \sum_{i=1}^{t}\left(x_{i+1} - x_i\right) \\
&= O(t^2) + \frac{1}{3}\left(\lfloor n/2 \rfloor - x_1\right) \\
&\le \frac{n}{6} + O(k^2).
\end{aligned} \tag{3}$$

By a symmetric argument, the same bound holds for the compression step on the right half of the median. Because both sides can be performed in parallel, the total cost for the compression step is at most $n/6 + O(k^2)$. The inner permutation step can be done in time at most k using odd-even

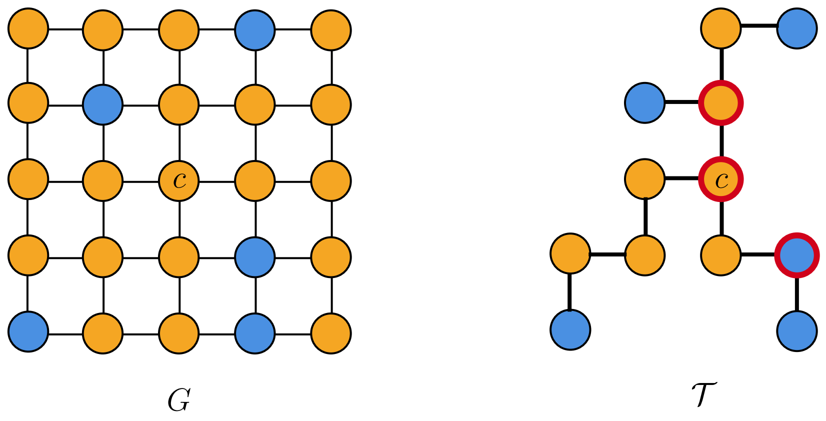


Figure 3: Illustration of the token tree $\mathcal{T}$ in Theorem 3.2 for a case where G is the 5×5 grid graph. Blue circles represent vertices in S and orange circles represent vertices not in S. Vertex c denotes the center of G. Red-outlined circles represent intersection vertices. In particular, note that one of the blue vertices is an intersection because it is the first common vertex on the path to c of two distinct blue vertices.

sort. The cost to perform the dilation step is also at most $n/6+O(k^2)$ because dilation is the inverse of compression. Thus, the total routing time for Algorithm 3.1 is at most $2(n/6+O(k^2))+k = n/3+O(k^2)$. □

It follows that sparse permutations on the path with $k = o(\sqrt{n})$ can be implemented using reversals with a full asymptotic factor of 3 speedup.

3.2 General graphs

We now present a more general result for implementing sparse permutations on an arbitrary graph.

Theorem 3.2. *Let $G = (V, E)$ be a graph with radius r and π a permutation of vertices. Let $S = \{v \in V : \pi_v \neq v\}$. Then π can be routed in time at most $2r/3 + O(|S|^2)$.*

Proof. We route π using a procedure similar to Algorithm 3.1, consisting of the same three steps adapted to work on a spanning tree of G: compression, inner permutation, and dilation. Dilation is the inverse of compression and the inner permutation step can be performed on a subtree consisting of just $k = |S|$ nodes by using the ROUTING VIA MATCHINGS algorithm for trees in $3k/2+O(\log k)$ time [Zha99]. It remains to show that compression can be performed in $r/3 + O(k^2)$ time.

We construct a *token tree* $\mathcal{T}$ that reduces the compression step to routing on a tree. Let c be a vertex in the *center* of G, i.e., a vertex with distance at most r to all vertices. Construct a shortest-path tree $\mathcal{T}'$ of G rooted at c, say, using breadth-first search. We assign a token to each vertex in S. Now $\mathcal{T}$ is the subtree of $\mathcal{T}'$ formed by removing all vertices $v \in V(\mathcal{T}')$ for which the subtree rooted at v does not contain any tokens, as depicted in Figure 3. In $\mathcal{T}$, call the first common vertex between paths to c from two distinct tokens an *intersection* vertex, and let $\mathcal{I}$ be the set of all intersection vertices. Note that if a token t_1 lies on the path from another token t_2 to c, then the vertex on which t_1 lies is also an intersection vertex. Since $\mathcal{T}$ has at most k leaves, $|\mathcal{I}| \leq k-1$.

For any vertex v in $\mathcal{T}$, let the *descendants* of v be the vertices $u \neq v$ in $\mathcal{T}$ whose path on $\mathcal{T}$ to c includes v. Now let $\mathcal{T}_v$ be the subtree of $\mathcal{T}$ rooted at v, i.e., the tree composed of v and all of the

```
Input : A vertex $v$ in token tree $\mathcal{T}$
function MoveUpTo($v$):
    if $\mathcal{T}_v$ contains no intersection vertices other than $v$ then          // Base case
        for each leaf node $u \in V(\mathcal{T}_v)$ :
            if $u$ is the first leaf node then
                Perform reversal from $u$ to $v$.
            else
                Perform reversal from $u$ to $v$, exclusive.
        return
    for each descendant $b$ of $v$ :
        $w :=$ the intersection vertex in $\mathcal{T}_b$ closest to $b$          // may include $b$
        MoveUpTo($w$)
        $m :=$ the number of tokens from $S$ in $\mathcal{T}_b$
        $l(p) :=$ the length of the path $p$ from $w$ to $b$ in $\mathcal{T}_v$
        if $l(p) \geq m$ then          // Enough room on $p$, form a path of tokens at $b$
            Route the $m$ tokens in $\mathcal{T}_b$ to the first $m$ vertices of $p$ using Routing via
              Matchings.
            Perform a reversal on the segment starting at $w$ and ending at $b$.
        else          // Not enough room on $p$, form a tree of tokens rooted at $b$
            Route the $m$ tokens in $\mathcal{T}_b$ as close as possible to $b$ using Routing via
              Matchings.
    if $v$ has no token then          // Put token on root $v$
        Perform a reversal on the segment starting from $v$ and ending at a vertex $u$ in $\mathcal{T}_v$
          with a token such that no descendant of $u$ has a token.
```

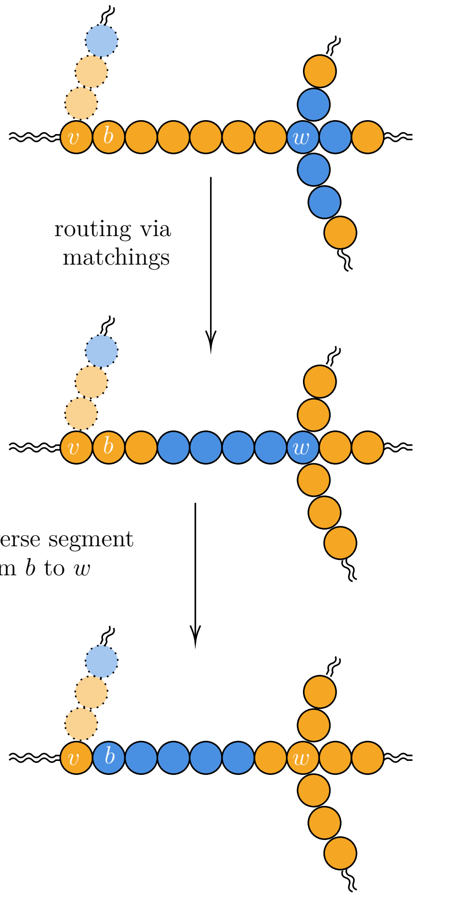
Algorithm 3.2: An algorithm that recursively moves all tokens from S that lie on $\mathcal{T}_v$ up to an intersection vertex v.

descendants of v. We say that all tokens have been *moved up to* a vertex v if for all vertices u in $\mathcal{T}_v$ without a token, T_u also does not contain a token. The compression step can then be described as moving tokens up to c.

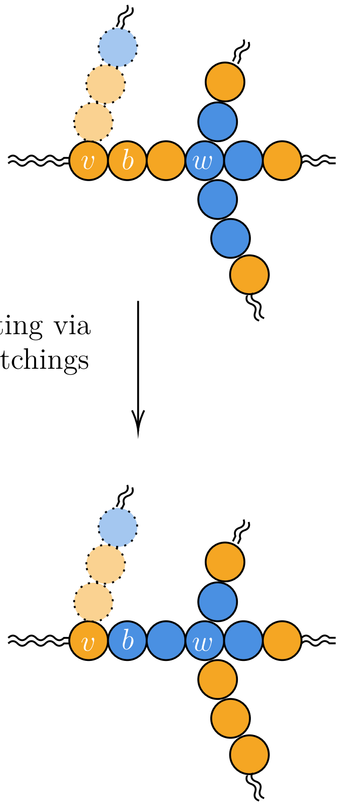
We describe a recursive algorithm for doing so in Algorithm 3.2. The base case considers the trivial case of a subtree with only one token. Otherwise, we move all tokens on the subtrees of descendant b up to the closest intersection w using recursive calls as illustrated in Figure 4. Afterwards, we need to consider whether the path p between v and w has enough room to store all tokens. If it does, we use a Routing via Matchings algorithm for trees to route tokens from w onto p, followed by a reversal to move these tokens up to v. Otherwise, the path is short enough to move all tokens up to v by the same Routing via Matchings algorithm.

We now bound the routing time on $\mathcal{T}_{w_1}$ of `MoveUpTo(`w_1`)`, for any vertex $w_1 \in V(\mathcal{T})$. First note that all operations on subtrees $\mathcal{T}_b$ of $\mathcal{T}_{w_1}$ are independent and can be performed in parallel. Let $w_1, w_2, \ldots, w_t$ be the sequence of intersection vertices that `MoveUpTo(·)` is recursively called on that dominates the routing time of `MoveUpTo(`w_1`)`. Let d_w, for $w \in V(\mathcal{T}_{w_1})$, be the distance of w to the furthest leaf node in $\mathcal{T}_w$. Assuming that the base case on Line 2 has not been reached, we have a routing time of

$$T(w_1) \leq T(w_2) + \frac{d_{w_1} - d_{w_2}}{3} + O(k), \tag{4}$$



(a) When $l(p) \geq m$. In this case, $l(p) = 7 \geq 5 = m$.



(b) When $l(p) < m$. In this case, $l(p) = 3 < 5 = m$.

Figure 4: An example of moving the m tokens in $\mathcal{T}_w$ up to b (Line 14–Line 18 in Algorithm 3.2).

where $O(k)$ bounds the time required to route $m \leq k$ tokens on a tree of size at most $2m$ following the recursive `MoveUpTo(`w_2`)` call [Zha99]. We expand the time cost $T(w_i)$ of recursive calls until we reach the base case of w_t to obtain

$$T(v) \leq T(w_t) + \sum_{i=1}^{t-1} \left(\frac{d_{w_i} - d_{w_{i+1}}}{3} + O(k) \right) = T(w_t) + \frac{d_{w_1} - d_{w_t}}{3} + t \cdot O(k) \leq \frac{d_{w_1}}{3} + (t+1) O(k). \quad (5)$$

Since $d_v \leq r$ and $t \leq k$, this shows that compression can be performed in $r/3 + O(k^2)$ time. $\square$

In general, a graph with radius r and diameter d will have $d/2 \leq r \leq d$. Using Theorem 3.2, this implies that for a graph G and a sparse permutation with $k = o(\sqrt{r})$, the bound for the routing time will be between $d/3 + o(d)$ and $2d/3 + o(d)$. Thus, for such sparse permutations, using reversals will always asymptotically give us a constant-factor worst-case speedup over any SWAP-only protocol since $\mathrm{rt}(G) \geq d$. Furthermore, for graphs with $r = d/2$, we can asymptotically achieve the full factor of 3 speedup.

```
Input   : $\pi$, a permutation of a contiguous subset of $[n]$.
function GenericDivideConquer(BinarySorter, $\pi$):
    if $|\pi| = 1$ then
        return $\emptyset$
    $B :=$ BinaryLabeling($\pi$)
    $\bar{\rho} :=$ BinarySorter($B$)
    $\pi := \pi \cdot \bar{\rho}$
    $\bar{\rho} = \bar{\rho} \mathbin{+\!\!+}$ GenericDivideConquer(BinarySorter, $\pi[0, \lfloor \frac{n}{2} \rfloor]$)
    $\bar{\rho} = \bar{\rho} \mathbin{+\!\!+}$ GenericDivideConquer(BinarySorter, $\pi[\lfloor \frac{n}{2} \rfloor + 1, |\pi|]$)
    return $\bar{\rho}$
```

Algorithm 4.1: Divide-and-conquer algorithm for recursively sorting π. `BinaryLabeling`(π) is a subroutine that uses Eq. (6) to transform π into a bitstring, and `BinarySorter` is a subroutine that takes as input the resulting binary string and returns an ordered reversal sequence $\bar{\rho}$ that sorts it.

4 Algorithms for routing on the path

Our general approach to implementing permutations on the path relies on the divide-and-conquer strategy described in Algorithm 4.1. It uses a correspondence between implementing permutations and sorting binary strings, where the former can be performed at twice the cost of the latter. This approach is inspired by [PS02] and [Ben+08] who use the same method for routing by reversals in the sequential case.

First, we introduce a binary labeling using the indicator function

$$I(v) = \begin{cases} 0 & \text{if } v < n/2, \\ 1 & \text{otherwise.} \end{cases} \tag{6}$$

This function labels any permutation $\pi \in \mathcal{S}_n$ by a binary string $I(\pi) := (I(\pi_1), I(\pi_2), \ldots, I(\pi_n))$. Let π be the target permutation, and σ any permutation such that $I(\pi\sigma^{-1}) = (0^{\lfloor n/2 \rfloor} 1^{\lceil n/2 \rceil})$. Then it follows that σ divides π into permutations π_L, π_R acting only on the left and right halves of the path, respectively, i.e., $\pi = \pi_L \cdot \pi_R \cdot \sigma$. We find and implement σ via a binary sorting subroutine, thereby reducing the problem into two subproblems of length at most $\lceil n/2 \rceil$ that can be solved in parallel on disjoint sections of the path. Proceeding by recursion until all subproblems are on sections of length at most 1, the only possible permutation is the identity and π has been implemented. Because disjoint permutations are implemented in parallel, the total routing time is $T(\pi) = T(\sigma) + \max(T(\pi_L), T(\pi_R))$.

We illustrate Algorithm 4.1 with an example, where the binary labels are indicated below the corresponding destination indices:

$$\begin{tikzcd}
7\,6\,0\,2\,5\,1\,3\,4 \arrow[r, "\text{label}"] & \frac{7\,6\,0\,2\,5\,1\,3\,4}{1\,1\,0\,0\,1\,0\,0\,1} \arrow[r, "\text{sort}"] & \frac{0\,3\,1\,2\,5\,7\,6\,4}{0\,0\,0\,0\,1\,1\,1\,1} \arrow[r, "\text{label}"] & \frac{0\,3\,1\,2\quad 5\,7\,6\,4}{0\,1\,0\,1\quad 0\,1\,1\,0} \arrow[d, "\text{sort}"] \\
& \frac{0\,1\quad 2\,3\quad 4\,5\quad 6\,7}{0\,1\quad 0\,1\quad 0\,1\quad 0\,1} & \frac{0\,1\quad 3\,2\quad 5\,4\quad 6\,7}{0\,1\quad 1\,0\quad 1\,0\quad 0\,1} \arrow[l, "\text{sort}"'] & \frac{0\,1\,3\,2\quad 5\,4\,6\,7}{0\,0\,1\,1\quad 0\,0\,1\,1} \arrow[l, "\text{label}"']
\end{tikzcd} \tag{7}$$

Each labeling and sorting step corresponds to an application of Eq. (6) and `BinarySorter`, respectively, to each subproblem. Specifically, in Eq. (7), we use TBS (Algorithm 4.2) to sort binary strings.

```
   Input  : $B$, a binary string
1  function TripartiteBinarySort($B$):
2      if $|B| = 1$ then
3          return $\emptyset$
4      $m_1 := \left\lfloor \frac{|B|}{3} \right\rfloor$
5      $m_2 := \left\lfloor \frac{2|B|}{3} \right\rfloor$
6      $\bar{\rho} :=$ TripartiteBinarySort($B[0, m_1]$)
7      $\bar{\rho} := \bar{\rho} +\!\!+$ TripartiteBinarySort($B[m_1 + 1, m_2] \oplus$ 11...1)
8      $\bar{\rho} := \bar{\rho} +\!\!+$ TripartiteBinarySort($B[m_2 + 1, |B|]$)
9      $B \leftarrow$ apply reversals in $\bar{\rho}$ to $B$
10     $i :=$ index of first 1 in $B$
11     $j :=$ index of last 0 in $B$
12     return $\bar{\rho} +\!\!+ \rho(i, j)$
```

Algorithm 4.2: Tripartite Binary Sort (TBS). We let $\rho(i, j)$ denote a reversal on the subsequence $S[i, j]$ (inclusive of i and j). In line 7, $\oplus$`11...1` indicates that we flip all the bits, so that we sort the middle third backwards.

We present two algorithms for `BinarySorter`, which perform the work in our sorting algorithm. The first of these binary sorting subroutines is Tripartite Binary Sort (TBS, Algorithm 4.2). TBS works by splitting the binary string into nearly equal (contiguous) thirds, recursively sorting these thirds, and merging the three sorted thirds into one sorted sequence. We sort the outer thirds forwards and the middle third backwards which allows us to merge the three segments using at most one reversal. For example, we can sort a binary string as follows:

$$\begin{array}{c}
\texttt{010011100011010011110111001} \\
\texttt{010011100}\quad\texttt{011010011}\quad\texttt{110111001} \\
\text{TBS}\downarrow \qquad \text{TBS}\downarrow\text{ backwards} \qquad \downarrow\text{TBS} \\
\texttt{000001111}\quad\texttt{111110000}\quad\texttt{000111111} \\
\texttt{00000}\,\underbracket{\texttt{1111111110000000}}\,\texttt{11111} \\
\texttt{00000000000011111111111111},
\end{array} \tag{8}$$

where the arrows with TBS indicate recursive calls to TBS and the bracket indicates the reversal to merge the segments. Let `GDC(TBS)` denote Algorithm 4.1 when using TBS to sort binary strings, where `GDC` stands for `GenericDivideConquer`.

The second algorithm is an adaptive version of TBS (Algorithm 4.3) that, instead of using equal thirds, adaptively chooses the segments' length. Adaptive TBS considers every pair of partition points, $0 \leq i \leq j < n - 1$, that would split the binary sequence into two or three sections: $B[0, i]$, $B[i + 1, j]$, and $B[j + 1, n - 1]$ (where $i = j$ corresponds to no middle section). For each pair, it calculates the minimum cost to recursively sort the sequence using these partition points. Since each section can be sorted in parallel, the total *sorting time* depends on the maximum time needed to sort one of the three sections and the cost of the final merging reversal. Let `GDC(ATBS)` denote Algorithm 4.1 when using Adaptive TBS to sort binary strings.

Notice that the partition points selected by TBS are considered by the Adaptive TBS algorithm and are selected by Adaptive TBS only if no other pair of partition points yields a faster sorting

```
    Input  : $B$, a binary string
 1  function AdaptiveTripartiteBinarySort($B$):
 2  $\overline{\rho} := \emptyset$
 3  for $i = 0$ to $n-2$ :
 4      for $j = i$ to $n-2$ :
 5          $\overline{\rho_0}$ = AdaptiveTripartiteBinarySort($B[0,i]$)
 6          $c_0 := cost(\overline{\rho_0})$
 7          $\overline{\rho_1}$ = AdaptiveTripartiteBinarySort($B[i+1,j]$)
 8          $c_1 := cost(\overline{\rho_1})$
 9          $\overline{\rho_2}$ = AdaptiveTripartiteBinarySort($B[j+1,n-1]$)
10          $c_2 := cost(\overline{\rho_2})$
11          $r :=$ cost of merging reversal using $i$ and $j$ as partition points
12          if $\overline{\rho} = \emptyset$ or $\max\{c_0, c_1, c_2\} + r < cost(\overline{\rho})$ then
13              $\overline{\rho} := \overline{\rho_0} \mathbin{+\!\!+} \overline{\rho_1} \mathbin{+\!\!+} \overline{\rho_2}$
14  return $\overline{\rho}$
```

Algorithm 4.3: Adaptive TBS. For the sake of clarity, we implement an exhaustive search over all possible ways to choose the partition points. However, we note that the optimal partition points can be found in polynomial time by using a dynamic programming method [Ben+08].

time. Thus, for any permutation, the sequence of reversals found by Adaptive TBS costs no more than that found by TBS. However, TBS is simpler to implement and will be faster than Adaptive TBS in finding the sorting sequence of reversals.

4.1 Worst-case bounds

In this section, we prove that all permutations of sufficiently large length n can be sorted in time strictly less than n using reversals. Let $n_x(b)$ denote the number of times character $x \in \{0,1\}$ appears in a binary string b, and let $T(b)$ (resp., $T(\pi)$) denote the best possible sorting time to sort b (resp., implement π) with reversals. Assume all logarithms are base 2 unless specified otherwise.

Lemma 4.1. *Let $b \in \{0,1\}^n$ such that $n_x(b) < cn + O(\log n)$, where $c \in [0, 1/3]$ and $x \in \{0,1\}$. Then, $T(b) \leq (c/3 + 7/18)\, n + O(\log n)$.*

Proof. To achieve this upper bound, we use TBS (Algorithm 4.2). There are $\lfloor \log_3 n \rfloor$ steps in the recursion, which we index by $j \in \{0, 1, \dots, \lfloor \log_3 n \rfloor\}$, with step 0 corresponding to the final merging step. Let $|\rho_j|$ denote the size of the longest reversal in recursive step j that merges the three sorted subsequences of size $n/3^{j+1}$. The size of the final merging reversal ρ_0 can be bounded above by $(c + 2/3)n + O(\log n)$ because $|\rho_0|$ is maximized when every x is contained in the leftmost third if $x = 1$ or the rightmost third if $x = 0$. So we have

$$T(b) \leq \left(\sum_{j=0}^{\log_3 n} \frac{|\rho_j|}{3} \right) + O(\log n) \leq \left(\frac{c}{3} + \frac{2}{9} \right) n + O(\log n) + \left(\sum_{j=1}^{\log_3 n} \frac{|\rho_j|}{3} \right) + O(\log n) \tag{9}$$

$$\leq \left(\frac{c}{3} + \frac{7}{18} \right) n + O(\log n), \tag{10}$$

where we used $|\rho_j| \leq n/3^j$ for $j \geq 1$. ☐

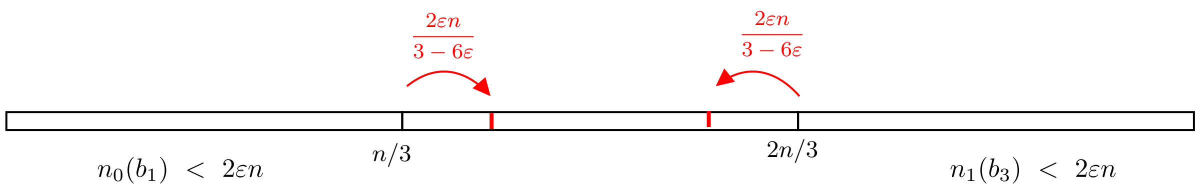

$\frac{2\varepsilon n}{3-6\varepsilon}$
$\frac{2\varepsilon n}{3-6\varepsilon}$
$n/3$
$2n/3$
$n_0(b_1) < 2\varepsilon n$
$n_1(b_3) < 2\varepsilon n$

Figure 5: Case 2 of Theorem 4.2. If there are few zeros and ones in the leftmost and rightmost thirds, respectively, we can shorten the middle section so that it can be sorted quickly. Then, because each of the outer thirds contain far more zeros than ones (or vice versa), they can both can be sorted quickly as well.

Now we can prove a bound on the cost of a sorting series found by Adaptive TBS for any binary string of length n.

Theorem 4.2. *For all bit strings $b \in \{0,1\}^n$ of arbitrary length $n \in \mathbb{N}$, $T(b) \le (1/2-\varepsilon)\,n + O(\log n) \approx 0.483n + O(\log n)$, where $\varepsilon = 1/3 - 1/\sqrt{10}$.*

Proof. Let $b \in \{0,1\}^n$ for some $n \in \mathbb{N}$. Partition b into three sections $b = b_1b_2b_3$ such that $|b_1| = |b_3| = \lfloor n/3 \rfloor$ and $|b_2| = n - 2\lfloor n/3 \rfloor$. Since $\lfloor n/3 \rfloor = n/3 - d$ where $d \in \{0, 1/3, 2/3\}$, we write $|b_1| = |b_2| = |b_3| = n/3 + O(1)$ for the purposes of this proof. Recall that if segments b_1 and b_3 are sorted forwards and segment b_2 is sorted backwards, the resulting segment can be sorted using a single reversal, ρ (see the example in Eq. (8)). Then we have

$$T(b) \le \max(T(b_1), T'(b_2), T(b_3)) + \frac{|\rho|+1}{3}, \tag{11}$$

where $T'(b_2)$ is the time to sort b_2 backwards using reversals.

We proceed by induction on n. For the base case, it suffices to note that every binary string can be sorted using reversals and, for finitely many values of $n \in \mathbb{N}$, any time needed to sort a binary string of length n exceeding $(1/2-\varepsilon)\,n$ can be absorbed into the $O(\log n)$ term. Now assume $T(b) \le (1/2-\varepsilon)\,k + O(\log k)$ for all $k < n$, $b \in \{0,1\}^k$.

Case 1: $n_0(b_1) \ge 2\varepsilon n$ or $n_1(b_3) \ge 2\varepsilon n$. In this case, $|\rho| \le n - 2\varepsilon n$, so

$$T(b) \le \frac{n - 2\varepsilon n + 1}{3} + \max(T(b_1), T'(b_2), T(b_3)) \le \left(\frac{1}{2} - \varepsilon\right) n + O(\log n) \tag{12}$$

by the induction hypothesis.

Case 2: $n_0(b_1) < 2\varepsilon n$ and $n_1(b_3) < 2\varepsilon n$. In this case, adjust the partition such that $|b_1| = |b_3| = n/3 + 2\varepsilon n/(3-6\varepsilon) - O(1)$ and consequently $|b_2| = n/3 - 4\varepsilon n/(3-6\varepsilon) + O(1)$, as depicted in Figure 5. In this adjustment, at most $2\varepsilon n/(3-6\varepsilon)$ zeros are added to the segment b_1 and likewise with ones to b_3. Thus, $n_1(b_3) \le 2\varepsilon n + 2\varepsilon n/(3-6\varepsilon) = (1 + 1/(3-6\varepsilon))\,2\varepsilon n$. Since $n = (3-6\varepsilon)|b_1| - O(1)$, we have

$$n_1(b_3) \le \left(1 + \frac{1}{3-6\varepsilon}\right) 2\varepsilon((3-6\varepsilon)|b_1| - O(1)) = (2-3\varepsilon)4\varepsilon|b_1| - O(1). \tag{13}$$

Let $c = (2-3\varepsilon)4\varepsilon = 2/15$. Applying Lemma 4.1 with this value of c yields

$$T(b_3) \le \left(\frac{2}{45} + \frac{7}{18}\right)|b_1| + O(\log(|b_1|)) = \left(\frac{1}{\sqrt{10}} - \frac{1}{6}\right) n + O(\log n). \tag{14}$$

Since $|b_1| = |b_3|$, we obtain the same bound $T(b_1) \leq (1/\sqrt{10} - 1/6)n + O(\log n)$ by applying Lemma 4.1 with the same value of c.

By the inductive hypothesis, $T'(b_2)$ can be bounded above by

$$T'(b_2) \leq \left(\frac{1}{2} - \varepsilon\right)\left(\frac{n}{3} - \frac{4\varepsilon}{3-6\varepsilon}n + O(1)\right) + O(\log n) = \left(\frac{1}{\sqrt{10}} - \frac{1}{6}\right)n + O(\log n). \tag{15}$$

Using Eq. (11) and the fact that $|\rho| \leq n$, we get the bound

$$T(b) \leq \left(\frac{1}{\sqrt{10}} - \frac{1}{6}\right)n + O(\log n) + \frac{n+1}{3} = \left(\frac{1}{2} - \varepsilon\right)n + O(\log n)$$

as claimed. □

This bound on the cost of a sorting series found by Adaptive TBS for binary sequences can easily be extended to a bound on the minimum sorting sequence for any permutation of length n.

Corollary 4.3. *For a length-n permutation π, $T(\pi) \leq (1/3 + \sqrt{2/5})n + O(\log^2 n) \approx 0.9658n + O(\log^2 n)$.*

Proof. To sort π, we turn it into a binary string b using Eq. (6). Then let $\rho_1, \rho_2, \ldots, \rho_m$ be a sequence of reversals to sort b. If we apply the sequence to get $\pi' = \pi\rho_1\rho_2\cdots\rho_m$, every element of π' will be on the same half as its destination. We can then recursively perform the same procedure on each half of π', continuing down until every pair of elements has been sorted.

This process requires $\lfloor \log n \rfloor$ steps, and at step i, there are 2^i binary strings of length $\frac{n}{2^i}$ being sorted in parallel. This gives us the following bound to implement π:

$$T(\pi) \leq \sum_{i=0}^{\log n} T(b_i), \tag{16}$$

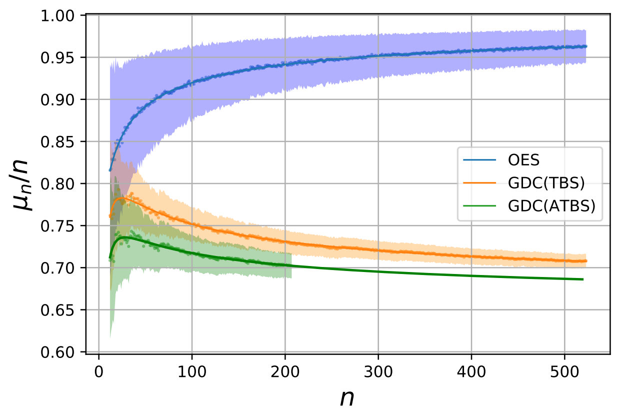
where $b_i \in \{0,1\}^{n/2^i}$. Applying the bound from Theorem 4.2, we obtain

$$T(\pi) \leq \sum_{i=0}^{\log n} T(b_i) \leq \sum_{i=0}^{\log n} \left(\left(\frac{1}{6} + \frac{1}{\sqrt{10}}\right)\frac{n}{2^i} + O(\log(n/2^i))\right) = \left(\frac{1}{3} + \sqrt{\frac{2}{5}}\right)n + O(\log^2 n).$$ □

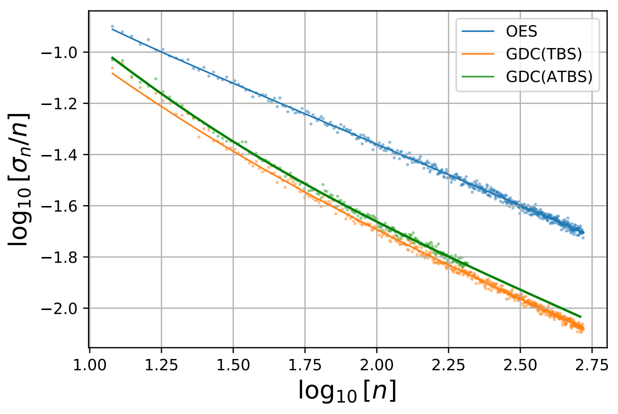
5 Average-case performance

So far we have presented worst-case bounds that provide a theoretical guarantee on the speedup of quantum routing over classical routing. However, the bounds are not known to be tight, and may not accurately capture the performance of the algorithm in practice.

In this section we show better performance for the *average-case* routing time, the expected routing time of the algorithm on a permutation chosen uniformly at random from $\mathcal{S}_n$. We present both theoretical and numerical results on the average routing time of swap-based routing (such as odd-even sort) and quantum routing using TBS and ATBS. We show that on average, `GDC(TBS)` (and `GDC(ATBS)`, whose sorting time on any instance is at least as fast) beats swap-based routing by a constant factor 2/3. We have the following two theorems, whose proofs can be found in Appendices A and B, respectively.



(a) Normalized mean routing time with std. deviation.



(b) Log normalized standard deviation of the routing time.

Figure 6: The mean routing time and fit of the mean routing time for odd-even sort (OES), and routing algorithms using Tripartite Binary Sort (`GDC(TBS)`) and Adaptive TBS (`GDC(ATBS)`). We exhaustively search for $n < 12$ and sample 1000 permutations uniformly at random otherwise. We show data for `GDC(ATBS)` only for $n \leq 207$ because it becomes too slow after that point. We find that the fit function $\mu_n = an + b\sqrt{n} + c$ fits the data with an $R^2 > 99.99\%$ (all three algorithms). For OES, the fit gives $a \approx 0.9999$; for `GDC(TBS)`, $a \approx 0.6599$; and for `GDC(ATBS)`, $a \approx 0.6513$. Similarly, for the standard deviation, we find that the fit function $\sigma_n^2 = an + b\sqrt{n} + c$ fits the data with $R^2 \approx 99\%$ (all three algorithms), suggesting that the normalized deviation of the performance about mean scales as $\sigma_n/n = \Theta(n^{-0.5})$ asymptotically.

Theorem 5.1. *The average routing time of any* SWAP*-based procedure is lower bounded by* $n - o(n)$.

Theorem 5.2. *The average routing time of* `GDC(TBS)` *is* $2n/3 + O(n^{\alpha})$ *for a constant* $\alpha \in (\frac{1}{2}, 1)$.

These theorems provide average-case guarantees, yet do not give information about the non-asymptotic behavior. Therefore, we test our algorithms on random permutations for instances of intermediate size.

Our numerics [KSS21] show that Algorithm 4.1 has an average routing time that is well-approximated by $c \cdot n + o(n)$, where $2/3 \lesssim c < 1$, using TBS or Adaptive TBS as the binary sorting subroutine, for permutations generated uniformly at random. Similarly, the performance of odd-even sort (OES) is well-approximated by $n + o(n)$. Furthermore, the advantage of quantum routing is evident even for fairly short paths. We demonstrate this by sampling 1000 permutations uniformly from $\mathcal{S}_n$ for $n \in [12, 512]$, and running OES and `GDC(TBS)` on each permutation. Due to computational constraints, `GDC(ATBS)` was run on sample permutations for lengths $n \in [12, 206]$. On an Intel i7-6700HQ processor with a clock speed of 2.60 GHz, OES took about 0.04 seconds to implement each permutation of length 512; `GDC(TBS)` took about 0.3 seconds; and, for permutations of length 200, `GDC(ATBS)` took about 6 seconds.

The results of our experiments are summarized in Figure 6. We find that the mean normalized time costs for OES, `GDC(TBS)`, and `GDC(ATBS)` are similar for small n, but the latter two decrease steadily as the lengths of the permutations increase while the former steadily increases. Furthermore, the average costs for `GDC(TBS)` and `GDC(ATBS)` diverge from that of OES rather quickly, suggesting that `GDC(TBS)` and `GDC(ATBS)` perform better on average for somewhat small permutations ($n \approx 50$) as well as asymptotically.

The linear coefficient a of the fit of μ_n for OES is $a \approx 0.9999 \approx 1$, which is consistent with the asymptotic bound proven in Theorems 5.1 and 5.2. For the fit of the mean time costs for `GDC(TBS)` and `GDC(ATBS)`, we have $a \approx 0.6599$ and $a \approx 0.6513$ respectively. The numerics suggest that the algorithm routing times agree with our analytics, and are fast for instances of realistic size. For example, at $n = 100$, `GDC(TBS)` and `GDC(ATBS)` have routing times of $\sim 0.75n$ and $0.72n$, respectively. On the other hand, OES routes in average time $> 0.9n$. For larger instances, the speedup approaches the full factor of $2/3$ monotonically. Moreover, the fits of the standard deviations suggest $\sigma_n/n = \Theta(1/\sqrt{n})$ asymptotically, which implies that as permutation length increases, the distribution of routing times gets relatively tighter for all three algorithms. This suggests that the average-case routing time may indeed be representative of typical performance for our algorithms for permutations selected uniformly at random.

6 Conclusion

We have shown that our algorithm, `GDC(ATBS)` (i.e., Generic Divide-and-Conquer with Adaptive TBS to sort binary strings), uses the fast state reversal primitive to outperform any SWAP-based protocol when routing on the path in the worst and average case. Recent work shows a lower bound on the time to perform a reversal on the path graph of n/α, where $\alpha \approx 4.5$ [Bap+20]. Thus we know that the routing time cannot be improved by more than a factor α over SWAPs, even with new techniques for implementing reversals. However, it remains to understand the fastest possible routing time on the path. Clearly, this is also lower bounded by n/α. Our work could be improved by addressing the following two open questions: (i) how fast can state reversal be implemented, and (ii) what is the fastest way of implementing a general permutation using state reversal?

We believe that the upper bound in Corollary 4.3 can likely be decreased. For example, in the proof of Lemma 4.1, we use a simple bound to show that the reversal sequence found by `GDC(TBS)` sorts binary strings with fewer than cn ones sufficiently fast for our purposes. It is possible that this bound can be decreased if we consider the reversal sequence found by `GDC(ATBS)` instead. Additionally, in the proof of Theorem 4.2, we only consider two pairs of partition points: one pair in each case of the proof. This suggests that the bound in Theorem 4.2 might be decreased if the full power of `GDC(ATBS)` could be analyzed.

Improving the algorithm itself is also a potential avenue to decrease the upper bound in Corollary 4.3. For example, the generic divide-and-conquer approach in Algorithm 4.1 focused on splitting the path exactly in half and recursing. An obvious improvement would be to create an adaptive version of Algorithm 4.1 in a manner similar to `GDC(ATBS)` where instead of splitting the path in half, the partition point would be placed in the optimal spot. It is also possible that by going beyond the divide-and-conquer approach, we could find faster reversal sequences and reduce the upper bound even further.

Our algorithm uses reversals to show the first quantum speedup for unitary quantum routing. It would be interesting to find other ways of implementing fast quantum routing that are not necessarily based on reversals. Other primitives for rapidly routing quantum information might be combined with classical strategies to develop fast general-purpose routing algorithms, possibly with an asymptotic scaling advantage. Such primitives might also take advantage of other resources, such as long-range Hamiltonians or the assistance of entanglement and fast classical communication.

Acknowledgements

We thank William Gasarch for organizing the REU-CAAR program that made this project possible.

A.B. and A.V.G. acknowledge support by the DoE ASCR Quantum Testbed Pathfinder program (award number de-sc0019040), ARO MURI, DoE ASCR Accelerated Research in Quantum Computing program (award number de-sc0020312), U.S. Department of Energy award number de-sc0019449, NSF PFCQC program, AFOSR, and AFOSR MURI. A.M.C. and E.S. acknowledge support by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder program (award number de-sc0019040) and the U.S. Army Research Office (MURI award number W911NF-16-1-0349). S.K. and H.S. acknowledge support from an NSF REU grant, REU-CAAR (CNS-1952352). E.S. acknowledges support from an IBM Ph.D. Fellowship.

References

[ACG94] N. Alon, F. R. K. Chung, and R. L. Graham. "Routing Permutations on Graphs via Matchings". In: *SIAM Journal on Discrete Mathematics* 7.3 (1994), pp. 513–530. DOI: 10.1137/s0895480192236628.

[Aru+19] F. Arute et al. "Quantum supremacy using a programmable superconducting processor". In: *Nature* 574.7779 (2019), pp. 505–510. DOI: 10.1038/s41586-019-1666-5.

[BP93] V. Bafna and P. A. Pevzner. "Genome rearrangements and sorting by reversals". In: *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*. 1993, pp. 148–157. DOI: 10.1137/S0097539793250627.

[BR17] I. Banerjee and D. Richards. "New Results on Routing via Matchings on Graphs". In: *Fundamentals of Computation Theory*. Lecture Notes in Computer Science 10472. Springer, 2017, pp. 69–81. DOI: 10.1007/978-3-662-55751-8_7.

[Bap+20] A. Bapat, E. Schoute, A. V. Gorshkov, and A. M. Childs. *Nearly optimal time-independent reversal of a spin chain*. 2020. arXiv: 2003.02843v1 `[quant-ph]`.

[Ben+08] M. A. Bender, D. Ge, S. He, H. Hu, R. Y. Pinter, S. Skiena, and F. Swidan. "Improved bounds on sorting by length-weighted reversals". In: *Journal of Computer and System Sciences* 74.5 (2008), pp. 744–774. DOI: 10.1016/j.jcss.2007.08.008.

[Ben+02] C. H. Bennett, J. I. Cirac, M. S. Leifer, D. W. Leung, N. Linden, S. Popescu, and G. Vidal. "Optimal simulation of two-qubit Hamiltonians using general local operations". In: *Physical Review A* 66.1 (2002). DOI: 10.1103/physreva.66.012305.

[Bre+16] T. Brecht, W. Pfaff, C. Wang, Y. Chu, L. Frunzio, M. H. Devoret, and R. J. Schoelkopf. "Multilayer microwave integrated quantum circuits for scalable quantum computing". In: *npj Quantum Information* 2.16002 (2016). DOI: 10.1038/npjqi.2016.2.

[CSU19] A. M. Childs, E. Schoute, and C. M. Unsal. "Circuit Transformations for Quantum Architectures". In: *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*. Vol. 135. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 3:1–3:24. DOI: 10.4230/LIPIcs.TQC.2019.3.

[KS95] J. Kececioglu and D. Sankoff. “Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement”. In: *Algorithmica* 13.1-2 (1995), pp. 180–210. DOI: 10.1007/BF01188586.

[KSS21] S. King, E. Schoute, and H. Shastri. *reversal-sort*. 2021. URL: https://gitlab.umiacs.umd.edu/amchilds/reversal-sort.

[Kja+20] M. Kjaergaard, M. E. Schwartz, J. Braumüller, P. Krantz, J. I.-J. Wang, S. Gustavsson, and W. D. Oliver. “Superconducting qubits: Current state of play”. In: *Annual Review of Condensed Matter Physics* 11 (2020), pp. 369–395. DOI: 10.1146/annurev-conmatphys-031119-050605.

[Klø08] T. Kløve. *Spheres of Permutations under the Infinity Norm–Permutations with limited displacement*. Research rep. 376. Department of Informatics, University of Bergen, Norway, 2008. URL: http://www.ii.uib.no/publikasjoner/texrap/pdf/2008-376.pdf.

[LDM84] S. Lakshmivarahan, S. K. Dhall, and L. L. Miller. “Parallel Sorting Algorithms”. In: vol. 23. Advances in Computers. Elsevier, 1984, pp. 321–323. DOI: 10.1016/S0065-2458(08)60467-2.

[LWD15] A. Lye, R. Wille, and R. Drechsler. “Determining the minimal number of swap gates for multi-dimensional nearest neighbor quantum circuits”. In: *The 20th Asia and South Pacific Design Automation Conference*. IEEE, 2015, pp. 178–183. DOI: 10.1109/asp-dac.2015.7059001.

[MG19] D. McClure and J. Gambetta. *Quantum computation center opens*. Tech. rep. IBM, 2019. URL: https://www.ibm.com/blogs/research/2019/09/quantum-computation-center/ (visited on 03/30/2020).

[MK13] C. Monroe and J. Kim. “Scaling the Ion Trap Quantum Processor”. In: *Science* 339.6124 (2013), pp. 1164–1169. DOI: 10.1126/science.1231298.

[Mon+14] C. Monroe, R. Raussendorf, A. Ruthven, K. R. Brown, P. Maunz, L.-M. Duan, and J. Kim. “Large-scale modular quantum-computer architecture with atomic memory and photonic interconnects”. In: *Physical Review A* 89.2 (2014). DOI: 10.1103/physreva.89.022317.

[Mur+19] P. Murali, J. M. Baker, A. J. Abhari, F. T. Chong, and M. Martonosi. “Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers”. In: *ASPLOS ’19. Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. The Association for Computing Machinery, 2019, pp. 1015–1029. DOI: 10.1145/3297858.3304075.

[NNN05] T. C. Nguyen, H. T. Ngo, and N. B. Nguyen. “Sorting by Restricted-Length-Weighted Reversals”. In: *Genomics, Proteomics & Bioinformatics* 3.2 (2005), pp. 120–127. DOI: 10.1016/S1672-0229(05)03016-0.

[PS16] M. Pedram and A. Shafaei. “Layout Optimization for Quantum Circuits with Linear Nearest Neighbor Architectures”. In: *IEEE Circuits and Systems Magazine* 16.2 (2016), pp. 62–74. DOI: 10.1109/MCAS.2016.2549950.

[PS02] R. Pinter and S. Skiena. “Genomic sorting with length-weighted reversals”. In: *Genome informatics. International Conference on Genome Informatics* 13 (2002), pp. 103–11. DOI: 10.11234/gi1990.13.103.

[Rau05] R. Raussendorf. "Quantum computation via translation-invariant operations on a chain of qubits". In: *Physical Review A* 72.5 (2005). DOI: 10.1103/physreva.72.052301.

[Rob55] H. Robbins. "A remark on Stirling's formula". In: *The American Mathematical Monthly* 62.1 (1955), pp. 26–29. DOI: 10.2307/2315957.

[SWD11] M. Saeedi, R. Wille, and R. Drechsler. "Synthesis of quantum circuits for linear nearest neighbor architectures". In: *Quantum Information Processing* 10.3 (2011), pp. 355–377. DOI: 10.1007/s11128-010-0201-2.

[SV17] M. Schwartz and P. O. Vontobel. "Improved Lower Bounds on the Size of Balls Over Permutations With the Infinity Metric". In: *IEEE Transactions on Information Theory* 63.10 (2017), pp. 6227–6239. DOI: 10.1109/TIT.2017.2697423.

[SSP13] A. Shafaei, M. Saeedi, and M. Pedram. "Optimization of Quantum Circuits for Interaction Distance in Linear Nearest Neighbor Architectures". In: *Proceedings of the 50th Annual Design Automation Conference*. DAC '13. ACM, 2013, 41:1–41:6. DOI: 10.1145/2463209.2488785.

[SSP14] A. Shafaei, M. Saeedi, and M. Pedram. "Qubit placement to minimize communication overhead in 2D quantum architectures". In: *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2014. DOI: 10.1109/aspdac.2014.6742940.

[TS10] I. Tamo and M. Schwartz. "Correcting Limited-Magnitude Errors in the Rank-Modulation Scheme". In: *IEEE Transactions on Information Theory* 56.6 (2010), pp. 2551–2560. DOI: 10.1109/TIT.2010.2046241.

[VHC02] G. Vidal, K. Hammerer, and J. I. Cirac. "Interaction Cost of Nonlocal Gates". In: *Physical Review Letters* 88.23 (2002), p. 237902. DOI: 10.1103/PhysRevLett.88.237902.

[Zha99] L. Zhang. "Optimal Bounds for Matching Routing on Trees". In: *SIAM Journal on Discrete Mathematics* 12.1 (1999), pp. 64–77. DOI: 10.1137/s0895480197323159.

[ZW19] A. Zulehner and R. Wille. "Compiling SU(4) quantum circuits to IBM QX architectures". In: *ASP-DAC '19. Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM Press, 2019, pp. 185–190. DOI: 10.1145/3287624.3287704.

A Average routing time using only SWAPs

In this section, we prove Theorem 5.1. First, define the infinity distance $d_\infty \colon \mathcal{S}_n \to \mathbb{N}$ to be $d_\infty(\pi) = \max_{1 \le i \le n} |\pi_i - i|$. Note that $0 \le d_\infty(\pi) \le n-1$. Finally, define the set of permutations of length n with infinity distance at most k to be $B_{k,n} = \{\pi \in \mathcal{S}_n : d_\infty(\pi) \le k\}$.

The infinity distance is crucially tied to the performance of odd-even sort, and indeed, any SWAP-based routing algorithm. For any permutation π of length n, the routing time of any SWAP-based algorithm is bounded below by $d_\infty(\pi)$, since the element furthest from its destination must be swapped at least $d_\infty(\pi)$ times, and each of those SWAPs must occur sequentially. To show that the average routing time of any SWAP-based protocol is asymptotically at least n, we first show that $|B_{(1-\varepsilon)n,n}|/n! \to 0$ for all $0 < \varepsilon \le 1/2$.

Schwartz and Vontobel [SV17] present an upper bound on $|B_{k,n}|$ that was proved in [Klø08] and [TS10]:

Lemma A.1. *For all* $0 < r < 1$, $|B_{rn,n}| \leq \Phi(rn, n)$, *where*

$$\Phi(k,n) = \begin{cases} ((2k+1)!)^{\frac{n-2k}{2k+1}} \prod_{i=k+1}^{2k} (i!)^{2/i} & \text{if} \quad 0 < k/n \leq \frac{1}{2} \\ (n!)^{\frac{2k+2-n}{n}} \prod_{i=k+1}^{n-1} (i!)^{2/i} & \text{if} \quad \frac{1}{2} \leq k/n < 1. \end{cases} \tag{17}$$

Proof. Note that $r = k/n$. For the case of $0 < r \leq 1/2$, refer to [Klø08] for a proof. For the case of $1/2 \leq r < 1$, refer to [TS10] for a proof. □

Lemma A.2.

$$n! = \Theta\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right) \tag{18}$$

Proof. This follows from well-known precise bounds for Stirling's formula:

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n+1}} \leq n! \leq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}} \tag{19}$$

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e \tag{20}$$

(see for example [Rob55]). □

With Lemmas A.1 and A.2 in hand, we proceed with the following theorem:

Theorem A.3. *For all* $0 < \varepsilon \leq 1/2$, $\lim_{n\to\infty} |B_{(1-\varepsilon)n,n}|/n! = 0$. *In other words, the proportion of permutations of length* n *with infinity distance less than* $(1-\epsilon)n$ *vanishes asymptotically.*

Proof. Lemma A.1 implies that $|B_{(1-\varepsilon)n,n}|/n! \leq \Phi((1-\varepsilon)n, n)/n!$. The constraint $0 < \varepsilon \leq 1/2$ stipulates that we are in the regime where $1/2 \leq r < 1$, since $r = 1 - \varepsilon$. Then we use Lemma A.2 to simplify any factorials that appear. Substituting Eq. (17) and simplifying, we have

$$\frac{\Phi((1-\varepsilon)n, n)}{n!} = \frac{\prod_{i=(1-\varepsilon)n+1}^{n-1} (i!)^{2/i}}{(n!)^{2\varepsilon - 2/n}} = O\left(\frac{e^{2\varepsilon n-2}}{n^{2\varepsilon n-2}} \prod_{i=(1-\varepsilon)n+1}^{n-1} \frac{i^{2+1/i}}{e^2}\right). \tag{21}$$

We note that $i^{1/i}$ terms can be bounded by

$$\prod_{i=(1-\varepsilon)n+1}^{n-1} i^{\frac{1}{i}} \leq \prod_{i=(1-\varepsilon)n+1}^{n-1} n^{\frac{1}{(1-\varepsilon)n}} \leq n^{\frac{\varepsilon}{1-\varepsilon}} \leq n \tag{22}$$

since $\varepsilon \leq 1/2$. Now we have

$$O\left(\frac{e^{2\varepsilon n-2}}{n^{2\varepsilon n-2}} \prod_{i=(1-\varepsilon)n+1}^{n-1} \frac{i^{2+1/i}}{e^2}\right) = O\left(\frac{n}{n^{2\varepsilon n-2}} \prod_{i=(1-\varepsilon)n+1}^{n-1} i^2\right) \tag{23}$$

$$= O\left(\frac{n}{n^{2\varepsilon n-2}}\left(\frac{(n-1)!}{((1-\varepsilon)n+1)!}\right)^2\right) \tag{24}$$

$$= O\left(\frac{n}{n^{2\varepsilon n-2}e^{2\varepsilon n}} \frac{(n-1)^{2n-1}}{((1-\varepsilon)n+1)^{2(1-\varepsilon)n+2}}\right) \tag{25}$$

$$= O\left(\frac{n}{n^{2\varepsilon n-2}e^{2\varepsilon n}} \frac{n^{2n}}{((1-\varepsilon)n)^{2(1-\varepsilon)n}}\right) \tag{26}$$

$$= O\left(\frac{n^3}{\exp\left((\ln(1-\varepsilon)(1-\varepsilon)+\varepsilon)2n\right)}\right). \tag{27}$$

Since $\ln(1-\varepsilon)(1-\varepsilon)+\varepsilon > 0$ for $\varepsilon > 0$, this vanishes in the limit of large n. □

Now we prove the theorem.

Proof of Theorem 5.1. Let $\bar{T}$ denote the average routing time of any SWAP-based protocol. Consider a random permutation π drawn uniformly from $\mathcal{S}_n$. Due to Theorem A.3, π will belong in $B_{(1-\varepsilon)n,n}$ with vanishing probability, for all $0 < \varepsilon \leq 1/2$. Therefore, for any fixed $0 < \varepsilon \leq 1/2$ as $n \to \infty$, $(1-\varepsilon)n < \mathbb{E}\left[d_\infty(\pi)\right]$. This translates to an average routing time of at least $n - o(n)$ because we have, asymptotically, $(1-\varepsilon)n \leq \bar{T}$ for all such ε. □

B Average routing time using TBS

In this section, we prove Theorem 5.2, which characterizes the average-case performance of TBS (Algorithm 4.2). This approach consists of two steps: a recursive call on three equal partitions of the path (of length $n/3$ each), and a merge step involving a single reversal.

We denote the uniform distribution over a set S as $\mathcal{U}(S)$. The set of all n-bit strings is denoted $\mathbb{B}^n$, where $\mathbb{B} = \{0,1\}$. Similarly, the set of all n-bit strings with Hamming weight k is denoted $\mathbb{B}^n_k$. For simplicity, assume that n is even. We denote the runtime of TBS on $b \in \mathbb{B}^n$ by $T(b)$.

When running `GDC(TBS)` on a given permutation π, the input bit string for TBS is $b = I(\pi)$, where the indicator function I is defined in Eq. (6). We wish to show that, in expectation over all permutations π, the corresponding bit strings are quick to sort. First, we show that it suffices to consider uniformly random sequences from $\mathbb{B}^n_{n/2}$.

Lemma B.1. *If $\pi \sim \mathcal{U}(\mathcal{S}_n)$, then $I(\pi) \sim \mathcal{U}(\mathbb{B}^n_{n/2})$.*

Proof. We use a counting argument. The number of permutations π such that $I(\pi) \in \mathbb{B}^n_{n/2}$ is $(n/2)!(n/2)!$, since we can freely assign index labels from $\{1,2,\ldots,n/2\}$ to the 0 bits of $I(\pi)$, and from $\{n/2+1,\ldots,n\}$ to the 1 bits of $I(\pi)$. Therefore, for a uniformly random π and arbitrary $b \in \mathbb{B}^n_{n/2}$,

$$\Pr(I(\pi) = b) = \frac{(n/2)!(n/2)!}{n!} = \frac{1}{\binom{n}{n/2}} = \frac{1}{|\mathbb{B}^n_{n/2}|}. \tag{28}$$

Therefore, $I(\pi) \sim \mathcal{U}(\mathbb{B}^n_{n/2})$. □

While $\mathbb{B}^n_{n/2}$ is easier to work with than $\mathcal{S}_n$, the constraint on the Hamming weight still poses an issue when we try to analyze the runtime recursively. To address this, Lemma B.2 below shows that relaxing from $\mathcal{U}(\mathbb{B}^n_{n/2})$ to $\mathcal{U}(\mathbb{B}^n)$ does not affect expectation values significantly.

We give a recursive form for the runtime of TBS. We use the following convention for the substrings of an arbitrary n-bit string a: if a is divided into 3 segments, we label the segments $a_{0.0}, a_{0.1}, a_{0.2}$ from left to right. Subsequent thirds are labeled analogously by ternary fractions. For example, the leftmost third of the middle third is denoted $a_{0.10}$, and so on. Then, the runtime of TBS on string a can be bounded by

$$T(a) \leq \max_{i\in\{0,1,2\}} T(a_{0.i}) + \frac{n_1(a_{0.0}) + n_1(\overline{a_{0.2}}) + n/3 + 1}{3}, \tag{29}$$

where $\overline{a}$ is the bitwise complement of bit string a and $n_1(a)$ denotes the Hamming weight of a. Logically, the first term on the right-hand side is a recursive call to sort the thirds, while the second term is the time taken to merge the sorted subsequences on the thirds using a reversal. Each term $T(a_{0.i})$ can be broken down recursively until all subsequences are of length 1. This yields the general formula

$$T(b) \leq \frac{1}{3}\left(\sum_{r=1}^{\lceil \log_3(n)\rceil} \max_{i\in\{0,1,2\}^{r-1}} \{n_1(a_{0.i0}) + n_1(\overline{a_{0.i2}})\} + n/3^r + 1\right), \tag{30}$$

where $i \in \emptyset$ indicates the empty string.

Lemma B.2. *Let $a \sim \mathcal{U}(\mathbb{B}^n)$ and $b \sim \mathcal{U}(\mathbb{B}^n_{n/2})$. Then*

$$\mathbb{E}[T(b)] \leq \mathbb{E}[T(a)] + \widetilde{O}(n^\alpha) \tag{31}$$

where $\alpha \in (\frac{1}{2}, 1)$ is a constant.

The intuition behind this lemma is that by the law of large numbers, the deviation of the Hamming weight from $n/2$ is subleading in n, and the TBS runtime does not change significantly if the input string is altered in a subleading number of places.

Proof. Consider an arbitrary bit string a, and apply the following transformation. If $n_1(a) = k \geq n/2$, then flip $k - n/2$ ones chosen uniformly randomly to zero. If $k < n/2$, flip $n/2 - k$ zeros to ones. Call this stochastic function $f(a)$. Then, for all a, $f(a) \in \mathbb{B}^n_{n/2}$, and for a random string $a \sim \mathcal{U}(\mathbb{B}^n)$, we claim that $f(a) \sim \mathcal{U}(\mathbb{B}^n_{n/2})$. In other words, f maps the uniform distribution on $\mathbb{B}^n$ to the uniform distribution on $\mathbb{B}^n_{n/2}$.

We show this by calculating the probability $\Pr(f(a) = b)$, for arbitrary $b \in \mathbb{B}^n_{n/2}$. A string a can map to b under f only if a and b disagree in the same direction: if, WLOG, $n_1(a) \geq n_1(b)$, then a must take value 1 wherever a, b disagree (and 0 if $n_1(a) \leq n_1(b)$). We denote this property by $a \succeq b$. The probability of picking a uniformly random a such that $a \succeq b$ with x disagreements between them is $\binom{n/2}{x}$, since $n_0(b) = n/2$. Next, the probability that f maps a to b is $\binom{n/2+x}{x}$.

Combining these, we have

$$\Pr(f(a)=b) = \sum_{x=-n/2}^{n/2} \Pr\Big(a \succeq b \text{ and } n_1(a) = \frac{n}{2}+x\Big)\cdot \Pr\Big(f(a)=b \mid a \succeq b \text{ and } n_1(a)=\frac{n}{2}+x\Big), \tag{32}$$

$$= \sum_{x=-n/2}^{n/2} \frac{\binom{n/2}{|x|}}{2^n}\cdot\frac{1}{\binom{n/2+|x|}{|x|}}, \tag{33}$$

$$= \frac{1}{\binom{n}{n/2}} \sum_{x=-n/2}^{n/2} \frac{\binom{n}{n/2-x}}{2^n}, \tag{34}$$

$$= \frac{1}{\binom{n}{n/2}} = \frac{1}{|\mathbb{B}^n_{n/2}|}. \tag{35}$$

Therefore, $f(a) \sim \mathcal{U}(\mathbb{B}^n_{n/2})$. Thus, f allows us to simulate the uniform distribution on $\mathbb{B}^n_{n/2}$ starting from the uniform distribution on $\mathbb{B}^n$.

Now we bound the runtime of TBS on $f(a)$ in terms of the runtime on a fixed a. Fix some $\alpha \in (\frac{1}{2},1)$. We know that $n_1(f(a)) = n/2$, and suppose $|n_1(a)-n/2| \le n^\alpha$. Since $f(a)$ differs from a in at most n^α places, then at level r of the TBS recursion (see Eq. (30)), the runtimes of a and $f(a)$ differ by at most $1/3\cdot\min\{2n/3^r, n^\alpha\}$. This is because the runtimes can differ by at most two times the length of the subsequence. Therefore, the total runtime difference is bounded by

$$\Delta T \le \frac{1}{3}\sum_{r=1}^{\lceil \log_3(n)\rceil} \min\Big\{\frac{2n}{3^r}, n^\alpha\Big\}, \tag{36}$$

$$= \frac{1}{3}\left(\sum_{r=1}^{\lceil \log_3(2n^{1-\alpha})\rceil} n^\alpha + 2\sum_{r=\lceil \log_3(2n^{1-\alpha})\rceil+1}^{\lceil \log_3(n)\rceil} \frac{n}{3^r}\right), \tag{37}$$

$$= \frac{1}{3}\left(n^\alpha \log(2n^\alpha/3) + 2\sum_{s=0}^{\lfloor \log_3(n^\alpha/2)\rfloor-1} 3^s\right) \tag{38}$$

$$= \frac{1}{3}\left(n^\alpha\log(2n^\alpha/3) + n^\alpha/2 - 1\right) = \widetilde{O}(n^\alpha). \tag{39}$$

On the other hand, if $|n_1(a)-n/2| \ge n^\alpha/2$, we simply bound the runtime by that of OES, which is at most n.

Now consider $a \sim \mathcal{U}(\mathbb{B}^n)$ and $b = f(a) \sim \mathcal{U}(\mathbb{B}^n_{n/2})$. Since $n_1(a)$ has the binomial distribution $\mathcal{B}(n,1/2)$, where $\mathcal{B}(k,p)$ is the sum of k Bernoulli random variables with success probability p, the Chernoff bound shows that deviation from the mean is exponentially suppressed, i.e.,

$$\Pr(|n_1(a)-n/2| \ge n^\alpha) = \exp(-O(n^{2\alpha-1})). \tag{40}$$

Therefore, the deviation in the expectation values is bounded by

$$|\mathbb{E}[T(f(a))] - \mathbb{E}[T(a)]| \le n\exp(-O(n^{2\alpha-1})) + c(1-\exp(-O(n^{2\alpha-1})))n^\alpha\log(n) = \widetilde{O}(n^\alpha), \tag{41}$$

where c is a constant. Finally, we conclude that

$$\mathbb{E}[T(b)] \le \mathbb{E}[T(a)] + \widetilde{O}(n^\alpha) \tag{42}$$

as claimed. □

Next, we prove the main result of this section, namely, that the runtime of `GDC(TBS)` is $2n/3$ up to additive subleading terms.

Proof of Theorem 5.2. We first prove properties for sorting a random n-bit string $a \sim \mathcal{U}(\mathbb{B}^n)$ and then apply this to the problem of sorting $b \sim \mathcal{U}(\mathbb{B}^n_{n/2})$ using Lemmas B.1 and B.2.

The expected runtime for TBS can be calculated using the recursive formula in Eq. (30):

$$\mathbb{E}[T(a)] \leq \frac{1}{3}\left(\sum_{r=1}^{\log_3(n)} \mathbb{E}\left[\max_{i\in\{0,1,2\}^{r-1}}\{n_1(a_{0.i0}) + n_1(\overline{a_{0.i2}})\}\right] + n/3^r + 1\right). \tag{43}$$

The summand contains an expectation of a maximum over Hamming weights of i.i.d. uniformly random substrings of length $n/3^r$, which is equivalent to a binomial distribution $\mathcal{B}(n/3^r, 1/2)$ where we have $n/3^r$ Bernoulli trials with success probability $1/2$. Because of independence, if we sample $X_1, X_2 \sim \mathcal{B}(n/3^r, 1/2)$, then $X_1 + X_2 \sim \mathcal{B}(2n/3^r, 1/2)$. Using Lemma B.3 with $m = 3^{r-1}$, the expected maximum can be bounded by

$$\frac{n}{3^r} + O\left(\sqrt{(n/3^r)\log(3^{r-1}n/3^r)}\right) = \frac{n}{3^r} + \widetilde{O}\left(n^{1/2}\right) \tag{44}$$

since the second term is largest when $r = O(1)$. Therefore,

$$\mathbb{E}[T(a)] \leq \frac{1}{3}\left(\sum_{r=1}^{\log_3(n)} \frac{2n}{3^r}\right) + \widetilde{O}\left(n^{1/2}\right) = \frac{n}{3} + \widetilde{O}\left(n^{1/2}\right). \tag{45}$$

Lemma B.2 then gives $\mathbb{E}[T(b)] \leq \frac{n}{3} + \widetilde{O}(n^{\alpha})$.

The routing algorithm `GDC(TBS)` proceeds by calling TBS on the full path, and then in parallel on the two disjoint sub-paths of length $n/2$. We show that the distributions of the left and right halves are uniform if the input permutation is sampled uniformly as $\pi \sim \mathcal{U}(\mathcal{S}_n)$. There exists a bijective mapping g such that $g(\pi) = (b, \pi_L, \pi_R) \in \mathbb{B}^n_{n/2} \times \mathcal{S}_{n/2} \times \mathcal{S}_{n/2}$ for any $\pi \in \mathcal{S}_n$ since

$$|\mathcal{S}_n| = n! = \binom{n}{n/2}\left(\frac{n}{2}\right)!\left(\frac{n}{2}\right)! = \left|\mathbb{B}^n_{n/2} \times \mathcal{S}_{n/2} \times \mathcal{S}_{n/2}\right|. \tag{46}$$

In particular, g can be defined so that b specifies which entries are taken to the first $n/2$ positions—say, without changing the relative ordering of the entries mapped to the first $n/2$ positions or the entries mapped to the last $n/2$ positions—and π_L and π_R specify the residual permutations on the first and last $n/2$ positions, respectively. Given $g(\pi) = (b, \pi_L, \pi_R)$, TBS only has access to b. After sorting, TBS can only perform deterministic permutations $\mu_L(b), \mu_R(b) \in \mathcal{S}_{n/2}$ on the left and right halves, respectively, that depend only on b. Thus TBS performs the mappings $\pi_L \mapsto \pi_L \circ (\mu_L(b))$ and $\pi_R \mapsto \pi_R \circ (\mu_R(b))$ on the output. Now it is easy to see that when $\pi_L, \pi_R \sim \mathcal{U}(\mathcal{S}_{n/2})$, the output is also uniform because the TBS mapping is independent of the relative permutations on the left and right halves.

More generally, we see that a uniform distribution over permutations $\mathcal{U}(\mathcal{S}_n)$ is mapped to two uniform permutations on the left and right half, respectively. Symbolically, for, $\pi \sim \mathcal{U}(\mathcal{S}_n)$, we have that

$$g(\pi) = (b, \pi_L, \pi_R) \sim \mathcal{U}(\mathbb{B}^n_{n/2} \times \mathcal{S}_{n/2} \times \mathcal{S}_{n/2}) = \mathcal{U}(\mathbb{B}^n_{n/2}) \times \mathcal{U}(\mathcal{S}_{n/2}) \times \mathcal{U}(\mathcal{S}_{n/2}). \tag{47}$$

As shown earlier, given uniform distributions over left and right permutations, the output is also uniform. By induction, all permutations in the recursive steps are uniform.

We therefore get a sum of expected TBS runtime on bit strings of lengths $n/3^r$, i.e.,

$$\sum_{r=1}^{\log_2 n} \mathbb{E}[T(b_r)] \leq \sum_{r=1}^{\log_2 n} \mathbb{E}[T(a_r)] + \widetilde{O}\left(\left(\frac{n}{2^{r-1}}\right)^\alpha\right) \leq \frac{2n}{3} + \widetilde{O}(n^\alpha) \tag{48}$$

where, by Lemma B.1 and the uniformity of permutations in recursive calls, we need only consider $b_r \sim \mathcal{U}(\mathbb{B}^{n/2^r}_{n/2^{r-1}})$ and we bound the expected runtime using Lemma B.2 with $a_r \sim \mathcal{U}(\mathbb{B}^{n/2^{r-1}})$. ☐

We end with a lemma about the order statistics of binomial random variables used in the proof of the main theorem.

Lemma B.3. *Given m i.i.d. samples from the binomial distribution $X_i \sim B(n,p)$ with $i \in [m]$, and $p \in [0,1]$, the maximum $Y = \max_i X_i$ satisfies*

$$\mathbb{E}[Y] < pn + O\left(\sqrt{n \log(mn)}\right). \tag{49}$$

Proof. We use Hoeffding's inequality for the Bernoulli random variable $X \sim \mathcal{B}(n,p)$, which states that

$$\Pr(X \geq (p+\epsilon)n) \leq \exp(-2n\epsilon^2) \quad \forall \varepsilon \geq 0. \tag{50}$$

Pick $\epsilon = \sqrt{\frac{c}{2n}\log(mn)}$, where $c > 0$ is a constant. For this choice, we have

$$\Pr(X_i \geq (p+\epsilon)n) \leq \left(\frac{1}{mn}\right)^c \tag{51}$$

for every $i = 1, \ldots, m$. Then the probability that $Y < (p+\epsilon)n$ is identical to the probability that $X_i < (p+\epsilon)n$ for every i, which for i.i.d X_i is given by

$$\Pr(Y < (p+\epsilon)n) = \Pr(X < (p+\epsilon)n)^m > \left(1 - \frac{1}{(mn)^c}\right)^m. \tag{52}$$

Using Bernoulli's inequality ($(1+x)^r \geq 1 + rx$ for $x \geq -1$), we can simplify the above bound to

$$\Pr(Y < (p+\epsilon)n)^m > 1 - m^{1-c}n^{-c}. \tag{53}$$

Finally, we bound the expected value of Y by an explicit weighted sum over its range:

$$\begin{aligned}
\mathbb{E}[Y] &= \sum_{k=0}^{n} \Pr(Y=k)\cdot k && (54)\\
&= \sum_{k=0}^{\lfloor (p+\epsilon)n \rfloor} \Pr(Y=k)\cdot k + \sum_{k=\lfloor (p+\epsilon)n \rfloor+1}^{n} \Pr(Y=k)\cdot k && (55)\\
&\leq \sum_{k=0}^{\lfloor (p+\epsilon)n \rfloor} \Pr(Y=k))\cdot k + n\cdot \sum_{k=\lfloor (p+\epsilon)n \rfloor+1}^{n} \Pr(Y=k) && (56)\\
&\leq \sum_{k=0}^{\lfloor (p+\epsilon)n \rfloor} \Pr(Y=k)\cdot k + (mn)^{1-c} && (57)\\
&\leq (p+\epsilon)n + (mn)^{1-c}. && (58)
\end{aligned}$$

Since $(mn)^{1-c} < 1$ for $c > 1$,

$$\mathbb{E}[Y] < \left\lceil pn + 1 + \sqrt{\frac{cn}{2}\log(mn)} \right\rceil = pn + O(\sqrt{n\log(mn)}) \tag{59}$$

as claimed. $\square$