Simple quantum algorithm to efficiently prepare sparse states

Debora Ramacciotti,[*] Andreea I. Lefterovici,[†] and Antonio F. Rotundo[‡]
Institut für Theoretische Physik, Leibniz Universität Hannover, 30167 Hannover, Germany

(Received 31 January 2024; revised 10 July 2024; accepted 12 August 2024; published 10 September 2024)

State preparation is a fundamental routine in quantum computation, for which many algorithms have been proposed. Among them, perhaps the simplest one is the Grover-Rudolph algorithm. In this paper we analyze the performance of this algorithm when the state to prepare is sparse. We show that the gate complexity is linear in the number of nonzero amplitudes in the state and quadratic in the number of qubits. We then introduce a simple modification of the algorithm, which makes the dependence on the number of qubits also linear. This is competitive with the best known algorithms for sparse state preparation.

DOI: 10.1103/PhysRevA.110.032609

I. INTRODUCTION

Given a classical vector $\psi \in \mathbb{C}^N$, the goal of state preparation is to build a unitary U_ψ such that $U_\psi|0\rangle = |\psi\rangle$, where $|\psi\rangle$ is a quantum state whose amplitudes are given by ψ. This is the first step of many algorithms, such as the quantum simulation of physical systems [1,2], quantum machine learning [3], and quantum linear solvers [4,5]. For this reason, state preparation is a subroutine of fundamental importance in quantum computing, and it is an object of ongoing research.

Early state-preparation algorithms are described in [1,6,7]. The basic idea of these algorithms is the same: It was independently introduced in [6,7] and had already been present in earlier works such as [1,8]. Following what is now the standard notation, we collectively refer to these algorithms as Grover-Rudolph algorithms.

In recent years, several works have tried to design new state-preparation algorithms with better worst-case asymptotic scaling (e.g., [9–12]) and have uncovered an interesting trade-off between space and time complexity in state preparation. In this paper we take a more practical point of view. We focus on sparse vectors, i.e., vectors with only a few nonzero elements. This is a special class of vectors, which often appears in practical applications, such as quantum linear solvers [4,5]. Recent works that have considered state-preparation algorithms tailored for sparse vectors are [13–16]. These algorithms have a complexity linear in both the sparsity of the vector and the number of qubits.

The number of gates required to prepare a generic state with the Grover-Rudolph algorithm scales exponentially in the number of qubits, so this algorithm is sometimes overlooked as an option to prepare sparse states. Our first contribution is to explicitly show that the Grover-Rudolph algorithm is able to prepare sparse vectors with a number of gates linear in the sparsity and quadratic in the number of qubits. This is a simple result, but it is not clearly stated and proved in the literature. We then introduce a small modification of the Grover-Rudolph algorithm, which reduces the complexity of preparing sparse vectors to linear in both the sparsity and the number of qubits. This shows that the Grover-Rudolph algorithm is a competitive algorithm for preparing sparse states.

A much better complexity of order $\log(nd)$ can be achieved at the cost of introducing a large $O(nd \log d)$ number of ancillary qubits [17]. However, the algorithms considered in this paper require only $O(n)$ ancillary qubits, so we will compare them only to other algorithms that require a similar number of ancillary qubits.

The rest of the paper is organized as follows. In Sec. II we summarize the Grover-Rudolph algorithm. In Sec. III we specialize to sparse states and analyze the number of gates the Grover-Rudolph algorithm requires for their preparation. In Sec. IV we introduce a simple variation of the Grover-Rudolph algorithm, which we call the permutation Grover-Rudolph algorithm, and show that it has the same complexity as more recent algorithms designed for sparse vectors [13–16].

II. GROVER-RUDOLPH ALGORITHM

In this section we describe the Grover-Rudolph algorithm [6] for state preparation.[1]

Letting $\psi \in \mathbb{C}^N$ be a classical vector, we want to implement a unitary U_ψ such that $U_\psi|0\rangle = |\psi\rangle$, where $|\psi\rangle$ is a quantum state with amplitudes equal to ψ. For simplicity, we assume that $N = 2^n$ so that we can encode the vector ψ in an n-qubit state.[2] More precisely, we want that

$$U_\psi|0\rangle = \frac{e^{i\theta}}{\|\psi\|} \sum_{i_1\cdots i_n} \psi_{i_1\cdots i_n} |i_1 \cdots i_n\rangle, \tag{1}$$

[*]Contact author: debora.ramacciotti@itp.uni-hannover.de
[†]Contact author: andreea.lefterovici@itp.uni-hannover.de
[‡]Contact author: af.rotundo@gmail.com

[1]One can consider several versions of the Grover-Rudolph algorithm. The one we present here is similar to the one of [9], except for the use of phase gates, in place of R_Z rotation, and for skipping an optimization step. See the text for further explanation.

[2]If this is not the case, one can pad ψ with zeros until this condition is met.

where the indices i_k take values in $\{0, 1\}$, $\|\psi\|$ is the 2-norm of the vector, and $\theta \in [0, 2\pi)$ is some irrelevant global phase. We use a standard binary representation of integers, with the most significant bit on the left. The strategy of the Grover-Rudolph algorithm is to construct a series of coarse-grained versions of ψ and prepare them recursively using controlled rotations.

More precisely, let $\psi^{(k)}$, for $k = 1, \dots, n-1$, be the following coarse-grained states with components:

$$\psi^{(k)}_{i_1\cdots i_k} = e^{i\arg(\psi^{(k+1)}_{i_1\cdots i_k 0})}\sqrt{\left|\psi^{(k+1)}_{i_1\cdots i_k 0}\right|^2 + \left|\psi^{(k+1)}_{i_1\cdots i_k 1}\right|^2}. \tag{2}$$

The superscript (k) keeps track of the number of qubits required to encode $\psi^{(k)}$. For notational convenience, we also introduce $\psi^{(0)} \equiv 1$ and $\psi^{(n)} \equiv \psi$. We prepare states $|\psi^{(k)}\rangle$, whose amplitudes are given by $\psi^{(k)}$, by recursively appending a qubit in state $|0\rangle$ and performing the transformation

$$\begin{aligned}\psi^{(k)}_{i_1\cdots i_k}|i_1\cdots i_k\rangle|0\rangle \to \psi^{(k)}_{i_1\cdots i_k}|i_1\cdots i_k\rangle\big(&\cos\theta^{(k)}_{i_1\cdots i_k}|0\rangle \\ &+ e^{i\phi^{(k)}_{i_1\cdots i_k}}\sin\theta^{(k)}_{i_1\cdots i_k}|1\rangle\big).\end{aligned} \tag{3}$$

The angles and phases should be chosen so that the new state is $|\psi^{(k+1)}\rangle$, i.e., such that the term on the right-hand side of (3) is equal to $\sum_j \psi^{(k+1)}_{i_1\dots i_k j}|i_1\cdots i_k j\rangle$. A short calculation shows that this requires

$$\begin{aligned}\theta^{(k)}_{i_1\cdots i_k} &= 2\arccos\frac{\left|\psi^{(k+1)}_{i_1\cdots i_k 0}\right|}{\left|\psi^{(k)}_{i_1\cdots i_k}\right|}, \\ \phi^{(k)}_{i_1\cdots i_k} &= \arg\left(\psi^{(k+1)}_{i_1\cdots i_k 1}\right) - \arg\left(\psi^{(k+1)}_{i_1\cdots i_k 0}\right),\end{aligned} \tag{4}$$

where $\arg(z)$ is the phase of a complex number z, and when $\psi^{(k)}_{i_1\cdots i_k} = 0$ one should pick $\theta^{(k)}_{i_1\cdots i_k} = 0$. For $k = 0$, there are no controlling qubits, so we are simply performing a one-qubit gate. The transformation (3) can be implemented by applying a y rotation $R_y(\theta^{(k)}_{i_1\cdots i_k})$ and a phase shift gate $P(\phi^{(k)}_{i_1\cdots i_k})$,[3] both controlled on the state of the first k qubits being $|i_1\cdots i_k\rangle$,

$$\begin{aligned}U_k = \sum_{i_1\cdots i_k} |i_1\cdots i_k\rangle\langle i_1\cdots i_k| \otimes \left[P\left(\phi^{(k)}_{i_1\cdots i_k}\right)R_y\left(\theta^{(k)}_{i_1\cdots i_k}\right)\right], \\ k = 0, \dots, n-1.\end{aligned} \tag{5}$$

Note that the superscripts of the angles and phases indicate how many qubits control the transformation and the subscripts which value the controls should have. For instance, $\theta^{(2)}_{11}$ means that the rotation and phase gates are applied when the first two qubits are in state $|11\rangle$.

The steps we have just explained are summarized in Algorithm 1 (GroverRudolph). The algorithm takes as input the angles and phases needed to implement the U_k's. We decide to store these in a list of dictionaries L_k for $k = 0, \dots, n-1$. The entries in the dictionary L_k are given by {key: value} pairs of the form $\{(i_1, \dots, i_k)\colon (\theta^{(k)}_{i_1\cdots i_k}, \phi^{(k)}_{i_1\cdots i_k})\}$. For the special case $k = 0$, we set $L_0 = \{1\colon (\theta^{(0)}, \phi^{(0)})\}$. This dictionary can be computed using Algorithm 2 (FindAngles).

[3]The y rotation acts on the computational basis as $R_y(\theta)|0\rangle = \cos(\theta/2)|0\rangle + \sin(\theta/2)|1\rangle$ and $R_y(\theta)|1\rangle = \cos(\theta/2)|1\rangle - \sin(\theta/2)|0\rangle$. The phase shift gate acts on the computational basis as $P(\phi)|0\rangle = |0\rangle$ and $P(\phi)|1\rangle = e^{i\phi}|1\rangle$.

ALGORITHM 1. GroverRudolph.

```
1: function GROVERRUDOLPH(angle and phase dictionaries $L_k$)
2:     $|\psi\rangle \leftarrow P(\phi^{(0)})R_y(\theta^{(0)})|0\rangle$
3:     for $k = 1, \dots, n-1$ do
4:         $|a\rangle \leftarrow |0\rangle$
5:         for $(i_1, \dots, i_k), (\theta^{(k)}_{i_1\cdots i_k}, \phi^{(k)}_{i_1\cdots i_k})$ in $L_k$ do   ▷Implement $U_k$ as in Eq. (5)
6:             if $|\psi\rangle = |i_1, \dots, i_k\rangle$ then
7:                 $|a\rangle \leftarrow P(\phi^{(k)}_{i_1\cdots i_k})R_y(\theta^{(k)}_{i_1\cdots i_k})|a\rangle$
8:             end if
9:         end for
10:        $|\psi\rangle \leftarrow |\psi\rangle \otimes |a\rangle$
11:    end for
12:    return $|\psi\rangle$
12: end function
```

We now analyze the complexity of both Algorithm 1 (GroverRudolph) and Algorithm 2 (FindAngles). Let us begin with the quantum part, Algorithm 1 (GroverRudolph). The algorithm consists of n unitaries U_k for $k = 0, 1, \dots, n-1$. Each unitary involves $k+1$ qubits and is made out of R_y rotations and phase gates controlled on k qubits (U_0 is not controlled). In the worst case, the kth unitary is made out of $O(2^k)$ controlled gates. For each controlled gate, we need $O(k)$ Toffoli gates. Summing over all k, we arrive at a worst-case asymptotic scaling of $O(n2^n)$. The complexity of the classical preprocessing required to determine the rotation angles and the phases is $O(2^n)$. To see this, consider the function FINDANGLES in Algorithm 2 (FindAngles). To find the angles, we need first to find the coarse-grained states $\psi^{(k)}$. In the worst case, $\psi^{(k)}$ has 2^k nonzero components, so to find the next coarse-grained state we need $O(2^k)$ operations. In total, we need $O(2^n)$ operations to find all coarse-grained states. To find the angles, we need the same number of operations.

Note that one could use the efficient gate decomposition from [9] to reduce the worst-case gate complexity to $O(2^n)$. We do not do this because when specializing to sparse vectors, most angles and phases are zero. The number of multicontrolled one-qubit gates is then much smaller than in the

ALGORITHM 2. FindAngles.

```
1:  function FINDANGLES($\psi \in \mathbb{C}^N$ with $N = 2^n$)
2:      $\psi^{(n)} \leftarrow \psi$
3:      for $k = n-1, n-2, \dots, 1$ do
4:          $L_k \leftarrow$ empty dictionary
5:          Compute $\psi^{(k)}$ using Eq. (2)
6:          for $(i_1, \dots, i_k) \in \{0, 1\}^k$
7:              Compute $\theta^{(k)}_{i_1\cdots i_k}$ and $\phi^{(k)}_{i_1\cdots i_k}$ using Eq. (4)
8:              $L_k[(i_1, \dots, i_k)] \leftarrow (\theta^{(k)}_{i_1\cdots i_k}, \phi^{(k)}_{i_1\cdots i_k})$
9:          end for
10:     end for
11:     return $\{L_k\}$
12: end function
```

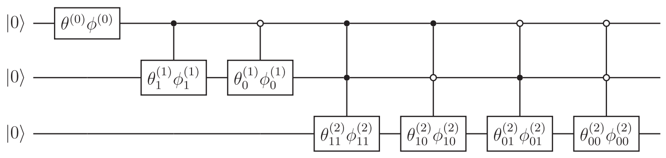

$|0\rangle$
$|0\rangle$
$|0\rangle$
$\theta^{(0)}\phi^{(0)}$
$\theta^{(1)}_1\phi^{(1)}_1$
$\theta^{(1)}_0\phi^{(1)}_0$
$\theta^{(2)}_{11}\phi^{(2)}_{11}$
$\theta^{(2)}_{10}\phi^{(2)}_{10}$
$\theta^{(2)}_{01}\phi^{(2)}_{01}$
$\theta^{(2)}_{00}\phi^{(2)}_{00}$

FIG. 1. General Grover-Rudolph circuit for preparing a three-qubit state.

worst-case scenario, and the decomposition of [9] would in fact lead to much deeper circuits.[4]

A simple example

Before continuing, we illustrate the Grover-Rudolph algorithm in a simple example. Consider a vector with eight positive components

$$\psi = \begin{bmatrix} 0 & \sqrt{\tfrac{1}{3}} & 0 & 0 & 0 & 0 & \sqrt{\tfrac{2}{3}} & 0 \end{bmatrix}. \tag{6}$$

Our goal is to prepare the corresponding quantum state

$$|\psi\rangle = \sqrt{\tfrac{1}{3}}|001\rangle + \sqrt{\tfrac{2}{3}}|110\rangle. \tag{7}$$

The most general circuit implementing Algorithm 1 (Grover-Rudolph) for three qubits is depicted in Fig. 1. Note that to simplify the notation, we have denoted the rotation gates by θ instead of $R_y(\theta)$ and phase gates by ϕ instead of $P(\phi)$.

To find the angles and phases, we first need to compute the coarse-grained vectors, as in Eq. (2). Since the vector ψ is positive, we can interpret its entries as the square root of a probability and visualize the coarse-graining procedure as in Fig. 2. The coarse-grained vectors $\psi^{(k)}$ are obtained by iteratively binning together the probabilities in pairs and summing them.

The Grover-Rudolph procedure starts by preparing the first coarse-grained state

$$|\psi^{(1)}\rangle = \sqrt{\tfrac{1}{3}}|0\rangle + \sqrt{\tfrac{2}{3}}|1\rangle \tag{8}$$

by applying a one-qubit rotation to $|0\rangle$. This can be done by applying $R_y(\theta^{(0)})$ with $\theta^{(0)} = 2\arccos(1/\sqrt{3})$. We can then prepare the next coarse-grained state

$$|\psi^{(2)}\rangle = \sqrt{\tfrac{1}{3}}|00\rangle + \sqrt{\tfrac{2}{3}}|11\rangle \tag{9}$$

by appending a qubit in state $|0\rangle$ and rotating it depending on the state of the first qubit. More specifically, when the first qubit is in state $|1\rangle$, we need to rotate the second to $|1\rangle$; when the first qubit is in state $|0\rangle$, we should leave the second qubit in $|0\rangle$. This can be done by picking $\theta^{(1)}_0 = 0$ and $\theta^{(1)}_1 = \pi$. The last step is performed similarly. We need to pick angles

[4]To be more explicit, consider Figs. 1 and 2 from [9]. The decomposition proposed there works by replacing the multicontrolled rotations with 2^k controlled-NOT (CNOT) gates and 2^k one-qubit rotations, with angles defined by Eq. (3) in [9]. For sparse vectors, it turns out that most angles on the right-hand side of Eq. (3) are zero. The angles on the left-hand side, on the other hand, are typically all different from zero. So for sparse vectors, we actually end up with a less efficient circuit.

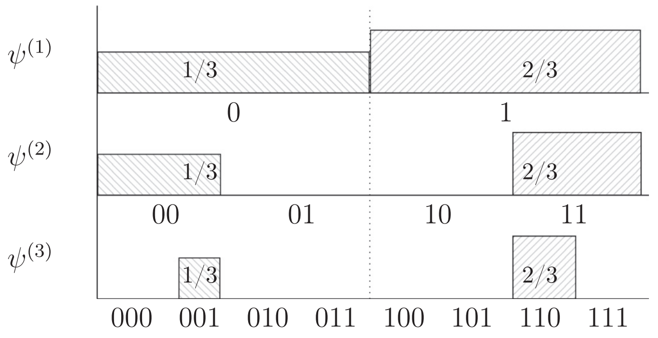


FIG. 2. Visualization of the coarse-graining procedure from Eq. (2). The bars in the histograms correspond to probabilities, i.e., amplitudes squared. Note that $\psi^{(3)} \equiv \psi$.

$\theta^{(2)}_{00} = \pi$ and $\theta^{(2)}_{11} = 0$. The end of this process yields the desired state $|\psi\rangle$.

III. GROVER-RUDOLPH ALGORITHM FOR SPARSE VECTORS

In this section we analyze how well the Grover-Rudolph algorithm performs for preparing sparse vectors. We consider vectors $\psi \in \mathbb{C}^N$ which have only $d \ll N$ nonzero elements. Above we saw that the worst-case complexity of the Grover-Rudolph algorithm is exponential in the number of qubits. However, we intuitively expect that for sparse vectors it should be possible to prepare the state with only $O(d)$ gates, as the vector has only d degrees of freedom. Below we explicitly show this and find that the Grover-Rudolph algorithm can prepare sparse states with $O(dn^2)$ gates.

We assume that we know the number of nonzero elements in ψ and their locations, namely, that we have access to ψ as a tuple of vectors (λ, ϕ). The vector λ contains the locations of the nonzero entries of ψ, and ϕ contains their values. The length of both vectors is d. Without loss of generality, we assume that the elements of λ are arranged in increasing order.

The coarse-graining procedure from Eq. (2) does not increase the number of nonzero elements; hence all $\psi^{(k)}$ have sparsity $d_k \leqslant d$. Let $(\lambda^{(k)}, \phi^{(k)})$ be the pair of vectors defining $\psi^{(k)}$. Equation (2) needs to be applied only when two consecutive elements of $\psi^{(k)}$ are nonzero. From this it follows that we can find each coarse-grained vector using at worst $O(d)$ operations, for a total of $O(dn)$ classical operations. Once we have these vectors, we can find the angles and phases with additional $O(dn)$ operations.

We can find the angles efficiently, exploiting the sparsity of the vector, as follows. Letting $(\lambda^{(n)}, \phi^{(n)}) \equiv (\lambda, \phi)$, we compute the coarse-grained vectors $(\lambda^{(k)}, \phi^{(k)})$ and layers of angles for $k = n-1, \ldots, 1$ by repeating the following steps. We initialize two empty lists $\lambda^{(k)}$ and $\phi^{(k)}$ for the coarse-grained vector and an empty dictionary L_k for the angles and phases. We then loop over the list $\lambda^{(k+1)}$ that was found at the previous iteration (in the first step, it is given by λ); let l be the current location in $\lambda^{(k+1)}$. Recall that we need to apply Eq. (2) only when $\psi^{(k+1)}$ has two consecutive nonzero entries, the first one of which is at an even location. So we check if $\lambda^{(k+1)}_l$ is even and if $\lambda^{(k+1)}_{l+1} = \lambda^{(k+1)}_l + 1$. If these two conditions are true, we compute Eq. (2) and append the

ALGORITHM 3. FindSparseAngles.

```
1:  function FindSparseAngles(nonzero location and values $(\lambda, \phi)$)
2:    $(\lambda^{(n)}, \phi^{(n)}) \leftarrow (\lambda, \phi)$
3:    for $k = n-1, n-2, \dots, 1$
4:      $\lambda^{(k)} \leftarrow [\,], \phi^{(k)} \leftarrow [\,]$
5:      $L_k \leftarrow \{\,\}$   ▷Empty dictionary for angles and phases of unitary $U_k$
6:      for $l = 0, 1, \dots, \mathrm{len}(\lambda^{(k+1)})$ do
7:        if $\lambda_l^{(k+1)}$ is even and $\lambda_{l+1}^{(k+1)} = \lambda_l^{(k+1)} + 1$ then
8:          $x \leftarrow$ evaluate Eq. (2)   ▷Use $\psi_{i_1\cdots i_k j}^{(k+1)} \leftarrow \phi_{l+j}^{(k+1)}$
9:          Append $x$ to $\phi^{(k)}$
10:         Append $\lambda_l^{(k+1)}/2$ to $\lambda^{(k)}$
11:         $l \leftarrow l+1$   ▷Skip one iteration
12:       else
13:         Append $\phi_l^{(k+1)}$ to $\phi^{(k)}$
14:         Append $\lfloor \lambda_l^{(k+1)}/2 \rfloor$ to $\lambda^{(k)}$
15:       end if
16:       Compute $\theta_{i_1\cdots i_k}^{(k)}$ and $\phi_{i_1\cdots i_k}^{(k)}$ using Eq. (4)   ▷Use $\psi_{i_1\cdots i_k}^{(k)} \leftarrow x$
17:       $L_k[(i_1, \dots, i_k)] \leftarrow (\theta_{i_1\cdots i_k}^{(k)}, \phi_{i_1\cdots i_k}^{(k)})$   ▷$(i_1, \dots, i_k)$ bit representation of $\lfloor \lambda_l^{(k+1)}/2 \rfloor$
18:     end for
19:   end for
20:   return $\{L_k\}$
21: end function
```

result to $\phi^{(k)}$. Otherwise, we simply copy $\phi_l^{(k+1)}$ to $\phi^{(k)}$. In any case, since the coarse-graining halves the length of the vector, the location of the new entry in $\phi^{(k)}$ is at $\lfloor \lambda_l^{(k+1)}/2 \rfloor$, so we append this value to $\lambda^{(k)}$. When we use Eq. (2), we should skip the next element in $\psi^{(k+1)}$, as this has already been merged, so we increase l by 1. Having found $\psi_l^{(k)}$, we can use Eq. (4) to find $\theta_{i_1\cdots i_k}^{(k)}$ and $\phi_{i_1\cdots i_k}^{(k)}$, where $(i_1, \dots, i_k)$ is the bit representation of $\lfloor \lambda_l^{(k+1)}/2 \rfloor$, and append them to L_k. Finally, we return all the dictionaries L_k. We summarize these steps in Algorithm 3 (FindSparseAngles). Note that this is the only thing we need to change to take advantage of sparsity; Algorithm 1 (GroverRudolph) remains unchanged.

Since each $\psi^{(k)}$ has at most d nonzero elements, we find that each U_k has at most d nonzero angles and phases. From this it follows that each U_k can be implemented using $O(kd)$ gates (d from the number of nonzero angles and k from the number of controlling qubits). Summing over k, we arrive at an overall gate complexity of $O(dn^2)$. As expected, we conclude that Algorithm 1 (GroverRudolph) performs on sparse vectors much better than the worst-case complexity would suggest.

On practical instances, the performance of Grover-Rudolph might very well be better than the worst case. So we estimate the typical complexity of the algorithm by sampling random sparse vectors and explicitly counting the gates needed to prepare them. We compute the cost of the algorithm by counting the number of Toffoli, CNOT, and one-qubit gates needed to implement it. We use standard constructions for controlled gates (see, e.g., [18]). To implement a one-qubit gate controlled on $k \geqslant 2$ qubits being in a state given by the bit string x, we use $k-1$ ancilla qubits, $2(k-1)$ Toffoli gates, two CNOT gates, and $4 + 2(n - |x|)$ one-qubit gates. Here $|x|$ is the Hamming weight of the bit string x.

In more detail, we consider 100 random complex vectors, for various values of d and n, and we study how the gate count scales as a function of d and n. The results are displayed in Fig. 3. Note that we display only the count of Toffoli gates, as the plots for the counts of CNOT and one-qubit gates are very similar. By inspecting the diagram, we find that also the average-case complexity of the algorithm scales linearly in d and quadratically in n.

In this section we have shown that the Grover-Rudolph algorithm works well on sparse vectors. However, the scaling we have found, $O(dn^2)$, does not quite match the best known algorithms for preparing sparse vectors [13–16], which have a worst-case complexity of $O(dn)$. Note that n is only logarithmic in the size of the vector, so it is possible that optimizing the Grover-Rudolph circuit might overcome this small overhead in practical applications. In Appendix A we consider a simple optimization procedure that reduces the number of needed gates, at the cost of a small classical overhead. However, we find that this reduction is not significant enough. Therefore, in the next section we introduce a small variant of the Grover-Rudolph algorithm for which we can prove a worst-case scaling of $O(dn)$.

IV. PERMUTATION OF THE GROVER-RUDOLPH ALGORITHM

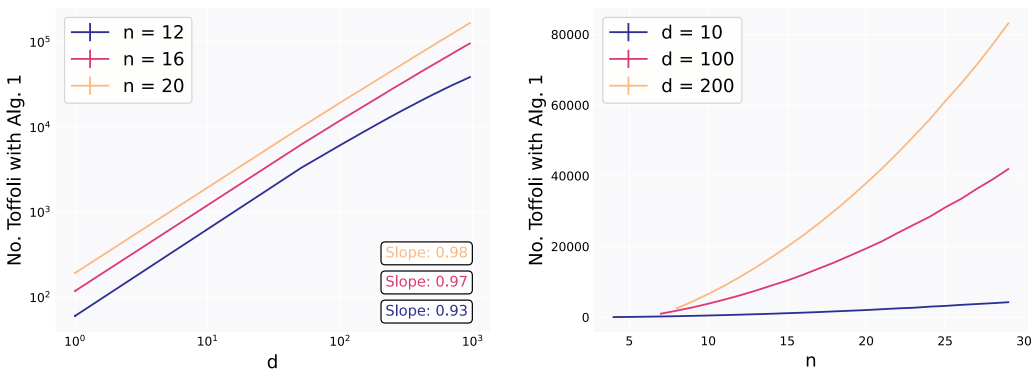
We introduce a simple variation of the Grover-Rudolph algorithm, for which we can prove the worst-case complexity of $O(dn)$. The idea of this variant is as follows. First, we prepare a dense state whose amplitudes are given by the nonzero entries of ψ. To prepare this state, we only need $\lceil \log d \rceil$ qubits, and we can use Algorithm 1 (GroverRudolph). We then append a sufficient number of qubits, such that the total dimension of the Hilbert space becomes N, and apply a permutation unitary which maps the nonzero amplitudes to their correct location. We show that this permutation can be efficiently implemented. The idea of preparing a dense state with all the nonzero entries and then permuting the basis states has already been used in [14]. Our algorithm differs in the implementation of the permutation and, as we discuss further below, has a better classical complexity.

Similarly to the preceding section, we assume we have access to ψ as a tuple of vectors (λ, ϕ). Each vector has size d, with $\lambda_i \in \{0, \dots, N\}$ being the location of the ith nonzero element of ψ and ϕ_i being its value. We assume without loss of generality that the elements of λ are arranged in increasing order. As a first step, we prepare a dense vector

$$|\tilde{\psi}\rangle = \sum_{i=0}^{d-1} \phi_i |i\rangle, \tag{10}$$

using the standard Grover-Rudolph algorithm. We then add a sufficient number of qubits initialized in $|0\rangle$ such that the total size of the Hilbert space becomes N. Finally, we apply a permutation unitary that maps $|i\rangle \to |\lambda_i\rangle$.

There are of course many permutations, mapping i to λ_i. We build one, directly decomposed in cycles, as follows. Let



(a) (b)

FIG. 3. Gate count for preparing random states using Algorithm 1 (GroverRudolph) (a) as a function of d at fixed n and (b) as a function of n at fixed d using (a) a log-log scale and (b) a lin-lin scale.

$i \in I$, with $I = \{0, 1, \dots, d-1\}$, and for simplicity consider $i = 0$. We initialize a cycle with elements $(0, \lambda_0)$. If $\lambda_0 \geqslant d$, we can close the cycle, add it to our permutation, and go to the next available i. If $\lambda_0 < d$, we set $j = \lambda_0$, we remove j from I, and we add λ_j to the cycle. We then repeat the steps above until we find a λ_j larger than d. The steps are summarized in Algorithm 4 (SparsePermutation).

An example can be useful to understand how SparsePermutation works. Consider $d = n = 4$ and $\lambda = (0, 3, 12, 15)$. We begin by initializing $S = (1, 1, 1, 1)$, which keeps track of which elements of λ have not already been included in a cycle, and an empty list P, where we will store the cycles. We then loop over $i = 0, 1, 2, 3$.

(i) For $i = 0$, we have $\lambda_0 = 0$, so we do not need to permute anything and we can continue to the next i [there is no need to add to P the trivial cycle (0)].

(ii) For $i = 1$, we have $S_1 = 1$ and $\lambda_1 = 3 \neq 1$, so we initialize the cycle c to $(1, 3)$. Since $\lambda_1 < d$, we also append $\lambda_3 = 15$ to c and we set $S_3 = 0$. Now $\lambda_3 \geqslant d$, so we can stop here and append $c = (1, 3, 15)$ to P.

(iii) For $i = 2$, we have $S_2 = 1$ and $\lambda_2 = 12 \neq 2$, so we set $c = (2, 12)$. Since $\lambda_2 \geqslant d$, we can stop and append c to P.

(iv) For $i = 3$, since $S_3 = 0$ [see step (ii)], we can continue to the next i.

Finally, we return $P = \{(1, 3, 15), (2, 12)\}$.

To understand how the complexity of Algorithm 4 (SparsePermutation) scales, let $P = \{c_0, c_1, \dots, c_{n_c-1}\}$ be the list of cycles returned by the algorithm. We denote by M_k the length of the kth cycle in the list, with $k = 0, 1, \dots, n_c - 1$. To generate this list, the algorithm loops over $i = 0, 1, \dots, d-1$ and does three blocks of operations: an if statement, an initialization, and a while loop. The first two operations take $O(n)$ time. The if statement is run every iteration and the initialization step is run $n_c \leqslant d$ times. So they both contribute $O(dn)$ to the classical complexity. The while loop is run only for cycles with more than two entries, with each iteration taking $O(n)$ time. Hence, we find a contribution of $O(n \sum_k \max(M_k - 2, 0))$. We can upper bound the sum by $\sum_k M_k$, which is the total length of the cycles in P. In the worst case, we have $\sum_i M_i = 2d$, which happens for a permutation made of d 2-cycles. This happens when $\lambda_i \geqslant d$ for all i. To see this, let $\lambda_j < d$ for some value of j. Then in the permutation we replace two cycles of length 2 with one cycle of length 3, and $\sum_i M_i$ decreases. We conclude that Algorithm 4 (SparsePermutation) has complexity of $O(nd)$.

ALGORITHM 4. SparsePermutation.

```
1:  function SPARSEPERM(vector of nonzero locations $\lambda$)
2:      $P \leftarrow \{\}$                                                  ▷Initialize empty permutation
3:      $S = \{S_0, S_1, \dots, S_{d-1}\}$ with $S_i = 1$, for $i = 0, \dots, d-1$     ▷Track valid starting values for cycles
4:      for $i = 0, 1, \dots, d-1$ do
5:          if $S_i = 0$ or $\lambda_i = i$ then
6:              continue                      ▷Skip iteration if $i$ already present in a cycle or if cycle is trivial
7:          end if
8:          $j \leftarrow \lambda_i, c \leftarrow \{i, j\}$
9:          while $j < d$ do
10:             $S_j \leftarrow 0, j \leftarrow \lambda_j$
11:             append $j$ to $c$
12:         end while
13:         append $c$ to $P$                                          ▷Add cycle to permutation
14:     end for
15:     return $P$                                                    ▷Permutation decomposed in cycles
16: end function
```

ALGORITHM 5. PermutationGroverRudolph.

```
1: function PERMGR(sparse vector $\{(x_0, \psi_0), \dots, (x_{d-1}, \psi_{d-1})\}$, number of qubits $n$)
2:     Apply Algorithm 1 (GroverRudolph) to prepare $|\tilde{\psi}\rangle = \sum_{i=0}^{d-1} \psi_i |i\rangle$
3:     Append $n - \lceil \log_2 d \rceil$ qubits in state $|0\rangle$          ▷Add qubits until there are $n$
4:     $P \leftarrow$ SPARSEPERM($\{x_i\}$)
5:     for $c \in P$
6:         Apply CYCLE($c, n$)          ▷See Algorithm 7 (Cycle)
7:     end for
8: end function
```

Once we have decomposed the permutation in cycles, we can use Algorithm 7 (Cycle) (see Appendix B) to implement it. Algorithm 7 (Cycle) implements a cycle of length M with $O(Mn)$ classical operations. Note that since, as we have argued above, $\sum_i M_i = O(d)$, the overall classical complexity is still of $O(nd)$. Putting everything together, we find Algorithm 5 (PermutationGroverRudolph) for preparing sparse states.

The quantum complexity of this algorithm is given by the cost of the Grover-Rudolph step needed to prepare $\tilde{\psi}$ and the cost of implementing the permutation $|i\rangle \rightarrow |\lambda_i\rangle$. As explained in Sec. II, the first is given by $O(n2^n)$, where n is the number of qubits. For the Grover-Rudolph step in this case, the number of qubits is only $n = O(\log d)$; hence we find $O(d \log d)$. For the second one, we can use Eq. (B1), which states that the complexity of one cycle scales like $O(Mn)$, where n is the number of qubits and M is the cycle length. The cost of the permutation scales like the sum of all cycles lengths times the number of qubits. In the worst case, we have d cycles of length 2, obtaining that the complexity of the permutation is $O(dn)$. Putting everything together, we find that the worst-case complexity of the algorithm scales as $O(dn)$. Note that we could consider better algorithms for the Grover-Rudolph step, e.g., the algorithm of [9], which would scale as $O(d)$ instead of $O(d \log d)$. Since the complexity of the algorithm is dominated by the permutation step, we do not think this would make a significant difference. It would be interesting to consider better ways to implement the permutation.

Similarly to what we did in Sec. III, we numerically estimate the average-case complexity of Algorithm 5 (PermutationGroverRudolph). We consider random complex sparse vectors, for various values of d and n, and compute the number of gates required by Algorithm 5 (PermutationGroverRudolph) to prepare them. The results are shown in Fig. 4. We find that scaling is linear in both d and n, the same as in the worst-case analysis.

From our analysis, we know that the worst-case cost of Algorithm 5 (PermutationGroverRudolph) scales better than that of Algorithm 1 (GroverRudolph). However, this speedup might fail to appear for vectors of reasonable size and sparsity. For this reason, we study empirically the relative costs of these two algorithms. In Fig. 5 we plot the ratio between the number of Toffoli gates required by Algorithm 5 (PermutationGroverRudolph) and Algorithm 1 (GroverRudolph) [subjected to the optimization strategy in Algorithm 6 (OptimizeAngles)] to prepare the same random vectors used to generate the plots in Fig. 4. We find that Algorithm 5 (PermutationGroverRudolph) performs better than the optimized version of Algorithm 1 (GroverRudolph) already at moderate values of n and starting at densities d/N between 10^{-3} and 10^{-2}. Unsurprisingly, for larger values of n the transition happens sooner, i.e., at larger densities.

V. CONCLUSION

In this paper we have studied the performance of the Grover-Rudolph algorithm for preparing sparse states. We have found that the usual version of the algorithm [see Algorithm 1 (GroverRudolph)] has a gate complexity of $O(dn^2)$. Here n is the number of qubits needed to encode the vector we want to prepare, ψ, and d is the number of nonzero

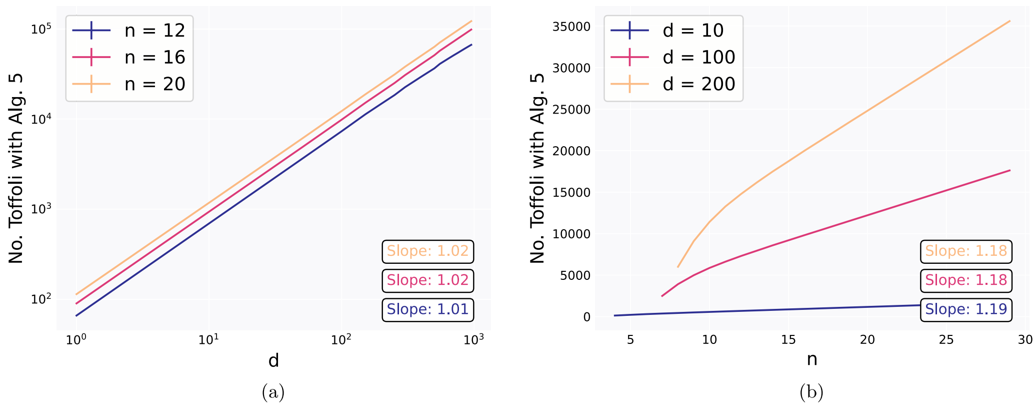
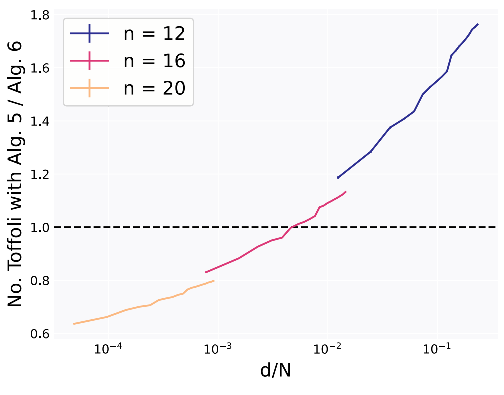
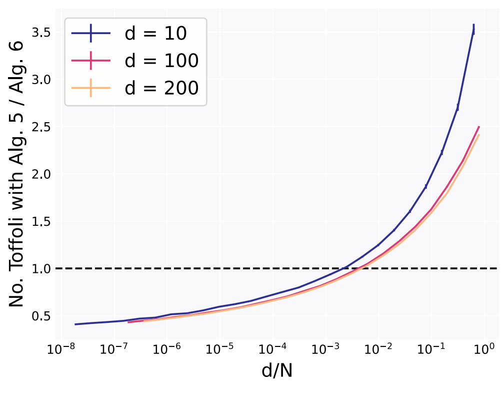


FIG. 4. Gate count for preparing random states using Algorithm 5 (PermutationGroverRudolph) (a) as a function of d at fixed n and (b) as a function of $N = 2^n$ at fixed d using (a) a log-log scale and (b) a lin-lin scale.



(a)



(b)

FIG. 5. Ratio between the number of gates required by Algorithm 5 (PermutationGroverRudolph) and the optimized gate count given by Algorithm 1 (GroverRudolph) in concert with Algorithm 6 (OptimizeAngles) to prepare some random states, as a function of the density d/N at (a) fixed n and (b) fixed d. We use a logarithmic scale on the abscissa.

entries in ψ. Moreover, we have introduced a simple modification of the algorithm which has a gate complexity of $O(dn)$ [Algorithm 5 (PermutationGroverRudolph)]. This is competitive with the best known algorithms for preparing sparse vectors [13–16]. The classical complexity of both Algorithm 1 (GroverRudolph) and Algorithm 5 (PermutationGroverRudolph) is $O(dn)$. This is better than those of [13,14], which are $O(nd^2 \log d)$ and $O(nd^2)$, respectively, and it is equal to that of [15,16]. Ultimately, the decision of which algorithm to use depends on the specific properties of the vectors to prepare. The main point of this work is that, when considering sparse vectors, the Grover-Rudolph algorithm should also be considered as an option.

We point out that in both Algorithm 1 (GroverRudolph) and Algorithm 5 (PermutationGroverRudolph) there is space for improvements. In particular, it would be interesting to consider optimization procedures to reduce the number of controlled rotations and Toffoli gates in Algorithm 1 (GroverRudolph). We consider one such optimization procedure in Appendix A, which shows promising results, for real vectors. In Algorithm 5 (PermutationGroverRudolph) it would be interesting to try to improve the permutation step, both at the level of the classical preprocessing, i.e., finding a different suitable permutation, and at the level of the quantum circuit needed to implement the permutation. However, it should be noted that these improvements will not change the asymptotic scaling that in all cases is $O(dn)$.

Finally, we point out an interesting possible future direction of research. In a recent work [19] an approximate state-preparation algorithm was developed that significantly reduces the cost of preparing a state when a small error is allowed. It would be interesting to understand whether this technique can be adapted to sparse states.

All the results were obtained using Python. The code is available on Ref. [20].

ACKNOWLEDGMENTS

This work was supported by the Quantum Valley Lower Saxony and by the BMBF project QuBRA. Helpful correspondence and discussions with J. Ammermann, L. Binkowski, T. Bittner, D. Eichhorn, D. Incalza, A. Loţan, T. J. Osborne, A. Peduri, S. Wilkening, and H. Wilming are gratefully acknowledged.

APPENDIX A: OPTIMIZING THE GATES

We consider a simple optimization strategy in which we merge consecutive gates that have the same rotation angles and phases and have controls differing only by one bit flip. For example, consider the situation depicted in Fig. 6. The first gate is conditioned on 11 and performs a rotation with an angle θ, while the second gate is conditioned on 10 and executes a rotation with the same angle θ. Given the gates differ in only one control and share the same rotation angles, they can be combined into a single gate, controlled on the first qubit being in 1. Note that this reduction not only affects the total gate count but also leads to gates with one fewer controlling qubit.

To explain how to perform the merge, we assume again that the angles and phases needed to implement the unitaries U_k are stored in dictionaries L_k as $\{(i_1, \dots, i_k) \colon (\theta^{(k)}_{i_1 \dots i_k}, \phi^{(k)}_{i_1 \dots i_k})\}$. For every control $(i_1, \dots, i_k)$ in dictionary L_k, we loop over its neighbors and check whether the corresponding angles, if present, are the same. Here the neighbors are found by flipping one bit, i.e., they are neighbors in Hamming distance. When the angle is the same, we merge the two controls into one. To do this, we replace the one bit on which the original controls disagreed with e. This simply helps to keep track of which qubits control the rotation. For example, 010 and 000 would be merged in $0e0$. We repeat this procedure until no new merging is possible. Note that to find neighbors, we do

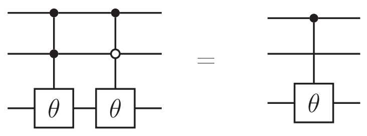


FIG. 6. Example of the optimization procedure.

ALGORITHM 6. OptimizeAngles.

```
function OPTIMIZEANGLES (dictionary$D$)
    Merging_success $\leftarrow$ True                    ▷Flag to mark merging success
    while (Merging_success = True) & (len($D$) > 1) do
        Merging_success $\leftarrow$ MERGEABLE($D$)
    end while
    return $D$
end function

function MERGEABLE(dictionary $D$)
    for $k, \theta$ in $D$
        for $i = 0, \dots, \mathrm{len}(k)$
            if $k[i] = e$ then continue
            end if
            $k' \leftarrow$ copy of $k$ with $i$th entry flipped
            $\theta' \leftarrow D[k']$
            if $\theta = \theta'$
                Remove $k, k'$ from $D$
                $k'' \leftarrow$ copy of $k$ with $i$th entry set to $e$
                Add $\{k'' : \theta\}$ to $D$
                return True
            end if
        end for
    end for
    return False
end function
```

not consider the entries set to e. These steps are summarized in Algorithm 6 (OptimizeAngles).

The complexity of Algorithm 6 (OptimizeAngles) is $O(dn^2 \log d)$. To see this, consider first the function MERGEABLE. This has complexity of $O(dn^2)$, as can it be seen by considering the structure of the two nested for loops. The first for loop iterates over all the items in the dictionary, whose number is upper bounded by d. The second for loop iterates over all bits in a key, whose number is upper bounded by n. Finally, the operations inside the inner for loop have complexity $O(n)$. Next we consider the while loop. This takes at most $\log d$ repetitions, as can be seen by noticing that at any iterations of the while loop, only angles which have been merged in the previous iteration can be further merged. Hence, the number of mergeable angles decreases by at least a factor 2 at every repetition of the while loop. Therefore, we have at most $O(\log d)$ iterations. Putting everything together, we arrive at a complexity of $O(dn^2 \log d)$.

It is difficult to understand theoretically how much this optimization reduces the number of required gates. Therefore, we rely on numerics. We consider the same random vectors used to generate Fig. 3, optimize the angles using Algorithm 6 (OptimizeAngles), and compute the ratio between the gates needed to prepare the state before and after optimization. The results are shown in Figs. 7(a) and 7(b). We apply the same strategy further for both real and uniform random vectors and we show the results in Figs. 7(c) and 7(d) and Figs. 7(e) and 7(f), respectively.

We find that the optimization of random vectors is only relevant at intermediate values of d. Empirically, the best improvement we observe is around 10%. As we have seen, this comes at a classical cost, which is $O(n^2 \log d)$ worse than the one required to find the angles. In the case of real vectors, we observe that at densities $d/N \approx 0.1$ the improvement in the gate count ranges from 20% to 25%, respectively, making them suitable candidates for this particular type of optimization. In the case of uniform vectors, the improvement in gate count at moderate values of d ranges from 10% to 35%. For fixed d, our optimization approach showcases improvements of up to 40%.

In the near future, quantum costs will be the bottleneck of quantum calculations; hence it is still useful to incur a higher classical complexity cost that will reduce the gate total count, even by a modest amount.

APPENDIX B: IMPLEMENTING PERMUTATION MATRICES

We present a simple way to implement permutation matrices.

Given a permutation σ of N elements, we want to implement a unitary, which we also denote by σ, that acts as $\sigma|i\rangle = |\sigma(i)\rangle$. For simplicity, we assume that $N = 2^n$ for some integer n, but this condition can be easily relaxed. First, we classically decompose the permutation in cycles, $\sigma = c_0 c_1 \cdots c_{n_c-1}$. Here n_c is the number of required cycles.

Each cycle is then simple to implement. Let $c = (x_0 x_1 \cdots x_{M-1})$ be a cycle of length M, where x_k are n-bit strings, and let $|a\rangle$ be an ancilla register that we initially prepare in $|0\rangle$. The cycle can be implemented by repeating for $k = 0, 1, \dots, M-1$ the following two operations.

(i) We flip the ancilla conditioned on the state of the first n qubits being $|x_k\rangle$.

(ii) Conditioned on the ancilla being in state $|1\rangle$, we map the first n qubits to $|x_{k+1}\rangle$.

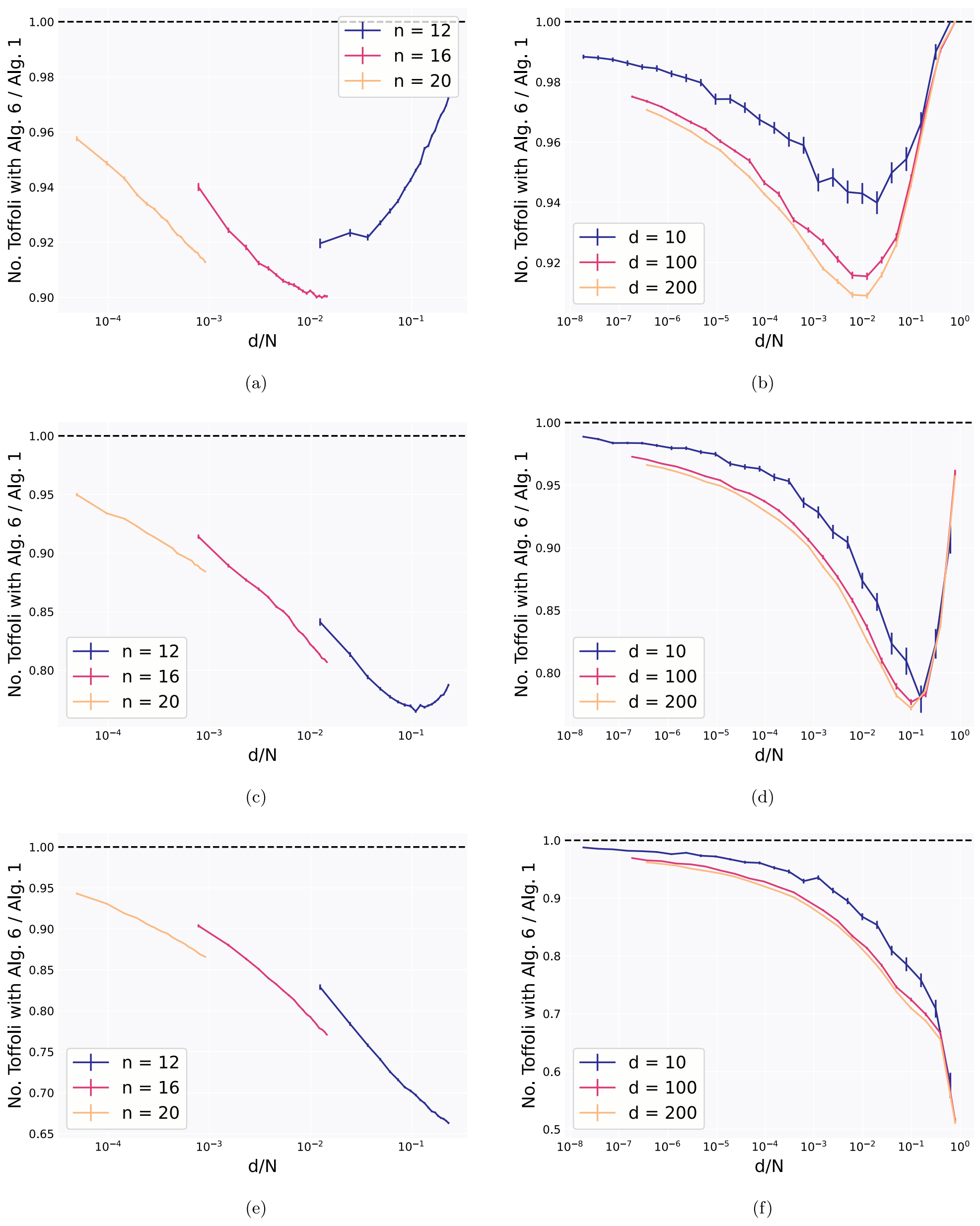


FIG. 7. Ratio of the number of Toffoli gates needed to run Algorithm 1 (GroverRudolph) after and before optimizing the angles using Algorithm 6 (OptimizeAngles) to prepare (a) and (b) random states, (c) and (d) random real states, and (e) and (f) uniform states as a function of the density d/N at fixed n and at fixed d. We use a logarithmic scale on the abscissa.

To map $|x_k\rangle$ to $|x_{k+1}\rangle$ we can use $\bigotimes_{l=0}^{n-1} X^{\Delta_l}$, where $\Delta = x_k \oplus x_{k+1}$ is the bitwise difference between x_k and x_{k+1}. Note that we are setting $x_M \equiv x_0$. For example, in Fig. 8 we show the circuit obtained for cycle $c = (0, 1, 2)$ and $N = 8$. The steps needed to implement a cycle are summarized in Algorithm 7 (Cycle).

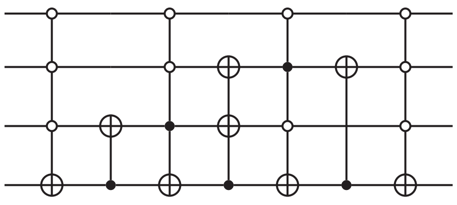


FIG. 8. Circuit for $N = 8$ and $\sigma = (0, 1, 2)$.

Finally, we provide a count for the number of gates required to run this algorithm. The ancilla flip is a generalized Toffoli gate with n controls; hence the cost of each is $2(n-1)\mathcal{C}_T + 4\mathcal{C}_1 + 2\mathcal{C}_{\text{CNOT}}$. We also need to include two X gates for every qubit controlled on 0, resulting in adding $n - |x_k|$ gates. There will be $M + 1$ of these terms. Then each state flip is determined by the number of bits that need to be swapped. More precisely, we need $|x_k \oplus x_{k+1}|$ CNOT gates for each of the M elements of a cycle.

ALGORITHM 7. Cycle.

```
1:  function CYCLE(cycle $c = (x_0x_1\cdots x_{M-1})$, $n$-qubit basis state $|y\rangle$)
2:      $x_M \leftarrow x_0$
3:      Add a qubit ancilla in state $|a\rangle = |0\rangle$
4:      for $k = 0, 1, \ldots, M-1$ do
5:          if $y = x_k$ then
6:              Flip ancilla, $a \to a \oplus 1$
7:          end if
8:          $\Delta \leftarrow x_k \oplus x_{k+1}$
9:          if $a = 1$ then
10:             $|y\rangle \leftarrow (X^{\Delta_0} \otimes X^{\Delta_1} \otimes \cdots \otimes X^{\Delta_{n-1}})|y\rangle$      ▷ Map $|x_k\rangle$ to $|x_{k+1}\rangle$
11:         end if
12:     end for
13:     return $|y\rangle$
14: end function
```

The cost of this algorithm is

$$\begin{aligned}\mathcal{C}[\text{cycle}(c)] &= 2(M+1)((n-1)\mathcal{C}_T + 2\mathcal{C}_1 + \mathcal{C}_{\text{CNOT}}) + 2\sum_{k=0}^{M}(n-|x_k|)\mathcal{C}_1 + 4\sum_{k=0}^{M-1}|x_k \oplus x_{k+1}|\mathcal{C}_1 + 2\sum_{k=0}^{M-1}|x_k \oplus x_{k+1}|\mathcal{C}_{\text{CNOT}}\\ &= O(Mn),\end{aligned} \tag{B1}$$

where $\mathcal{C}_T$, $\mathcal{C}_{\text{CNOT}}$, are C_1 are the costs of Toffoli, CNOT, and one-qubit gates, respectively, $c = (x_0x_1\cdots x_{M-1})$, $|\cdot|$ denotes the Hamming weight of the bit string, and again $x_M \equiv x_0$. Note that the algorithm also requires $O(Mn)$ classical operations.

[1] C. Zalka, Simulating quantum systems on a quantum computer, Proc. R. Soc. London Ser. A **454**, 313 (1998).

[2] I. M. Georgescu, S. Ashhab, and F. Nori, Quantum simulation, Rev. Mod. Phys. **86**, 153 (2014).

[3] S. Lloyd, M. Mohseni, and P. Rebentrost, Quantum algorithms for supervised and unsupervised machine learning, arXiv:1307.0411.

[4] A. W. Harrow, A. Hassidim, and S. Lloyd, Quantum algorithm for linear systems of equations, Phys. Rev. Lett. **103**, 150502 (2009).

[5] A. M. Childs, R. Kothari, and R. D. Somma, Quantum algorithm for systems of linear equations with exponentially improved dependence on precision, SIAM J. Comput. **46**, 1920 (2017).

[6] L. Grover and T. Rudolph, Creating superpositions that correspond to efficiently integrable probability distributions, arXiv:quant-ph/0208112.

[7] P. Kaye and M. Mosca, in *Proceedings of the International Conference on Quantum Information, Rochester* (Optica, Washington, DC, 2001), paper PB28.

[8] E. Knill, Approximation by quantum circuits, arXiv:quant-ph/9508006.

[9] M. Mottonen, J. J. Vartiainen, V. Bergholm, and M. M. Salomaa, Transformation of quantum states using uniformly controlled rotations, Quantum Inf. Comput. **5**, 467 (2005).

[10] X. Sun, G. Tian, S. Yang, P. Yuan, and S. Zhang, in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (IEEE, Piscataway, NJ, 2023).

[11] M. Plesch and Č. Brukner, Quantum-state preparation with universal gate decompositions, Phys. Rev. A **83**, 032302 (2011).

[12] J. Gonzalez-Conde, T. W. Watts, P. Rodriguez-Grasa, and M. Sanz, Efficient quantum amplitude encoding of polynomial functions, Quantum **8**, 1297 (2024).

[13] N. Gleinig and T. Hoefler, An efficient algorithm for sparse quantum state preparation, in *Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA* (IEEE, Piscataway, NJ, 2021), pp. 433–438.

[14] E. Malvetti, R. Iten, and R. Colbeck, Quantum circuits for sparse isometries, Quantum **5**, 412 (2021).

[15] T. M. L. de Veras, L. D. da Silva, and A. J. da Silva, Double sparse quantum state preparation, Quantum Inf. Process. **21**, 204 (2022).

[16] F. Mozafari, G. D. Micheli, and Y. Yang, Efficient deterministic preparation of quantum states using decision diagrams, Phys. Rev. A **106**, 022617 (2022).

[17] X.-M. Zhang, T. Li, and X. Yuan, Quantum state preparation with optimal circuit depth: Implementations and applications, Phys. Rev. Lett. **129**, 230504 (2022).

[18] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press, Cambridge, 2010).

[19] G. Marin-Sanchez, J. Gonzalez-Conde, and M. Sanz, Quantum algorithms for approximate function loading, Phys. Rev. Res. **5**, 033114 (2023).

[20] https://github.com/qubrabench/grover-rudolph.