Quantum Linear Algorithm for Edit Distance using the Word QRAM Model

Massimo Equi
Arianne Meijer - van de Griend
Veli Mäkinen

Department of Computer Science
University of Helsinki, Finland

April 2022

Abstract

Many problems that can be solved in quadratic time have bit-parallel speed-ups with factor $w$, where $w$ is the computer word size. For example, edit distance of two strings of length $n$ can be solved in $O(n^2/w)$ time. In a reasonable classical model of computation, one can assume $w = \Theta(\log n)$. There are conditional lower bounds for such problems stating that speed-ups with factor $n^\epsilon$ for any $\epsilon > 0$ would lead to breakthroughs in complexity theory. However, these conditional lower bounds do not cover quantum models of computing. Indeed, Boroujeni et al. (J. ACM, 2021) showed that edit distance can be approximated within a factor 3 in sub-quadratic time $O(n^{1.81})$ using quantum computing. They also showed that, in their chosen model of quantum computing, the approximation factor cannot be improved using sub-quadratic time.

To break through the aforementioned classical conditional lower bounds and this latest quantum lower bound, we enrich the model of computation with a quantum random access memory (QRAM), obtaining what we call the word QRAM model. Under this model, we show how to convert the bit-parallelism of quadratic time solvable problems into quantum algorithms that attain speed-ups with factor $n$. The technique we use is simple and general enough to apply to many bit-parallel algorithms that use Boolean logics and bit-shifts. To apply it to edit distance, we first show that the famous $O(n^2/w)$ time bit-parallel algorithm of Myers (J. ACM, 1999) can be adjusted to work without arithmetic $+$ operations. As a direct consequence of applying our technique to this variant, we obtain linear time edit distance algorithm under the word QRAM model for constant alphabet. We give further results on a restricted variant of the word QRAM model to give more insights to the limits of the model.

1 Introduction

The edit distance between two strings $A$ and $B$ is the minimum number of character insertions, deletions and substitutions needed to turn $A$ into $B$. Computing the edit distance between two strings is arguably one of the most fundamental problem of combinatorial pattern matching. A classic quadratic solution based on dynamic programming solves the problem in time $O(nm)$, where $n = |A|$ and $m = |B|$, and thus $O(n^2)$ when $n = m$. This is likely to be optimal, since a celebrated conditional lower bound [2] states that no algorithm can compute the edit distance in time $O(n^{2-\epsilon})$, for any $\epsilon > 0$, unless the orthogonal vectors hypothesis (OVH) and thus the strong exponential time hypothesis (SETH) fail. These results are clearly dependent on the model of
computation that we assume, and the quadratic algorithm and lower bound are a consequence of assuming a random access memory (RAM) model. In different models of computation, different results could be achieved. For instance, in the word RAM model, bit-parallelism allows to solve the problem in time $O(|A|/w|B|)$ [11], where $w$ is the size of a memory word. However, in this model we typically assume $w = O(\log n)$, so that we properly reflect how real general purpose computers work. Unfortunately, since $\log n = o(n^{\epsilon})$, the algorithm remains quadratic in worst case.

Quantum computing offers a radically different model of computation, that looks promising for finding true subquadratic solutions. In quantum computing, the unit of information is the qubit (quantum bit), which has the ability of being in state $|0\rangle$, or in $|1\rangle$, or in a linear combination of the two $\alpha|0\rangle + \beta|1\rangle$, called superposition. Coefficients $\alpha, \beta \in \mathbb{C}$ are called amplitudes and are such that $|\alpha|^2 + |\beta|^2 = 1$. As we will show, the use of superpositions dramatically impacts the time complexity of algorithms. Recently, Boroujeni et al. [3] proposed a quantum algorithm that solves edit distance in time $O(n^{1.810})$ within an approximation factor of 3, or in time $O(n^{1.708})$ within a larger constant factor. Moreover, with a reduction from the parity problem [5], they prove that a quantum algorithm cannot compute edit distance in truly subquadratic time within a constant factor smaller than 3. To the best of our knowledge, this is the only work providing some form of subquadratic time algorithm for edit distance using quantum computing.

In our work, we break through the aforementioned classical SETH lower bound and this latest quantum lower bound by enriching our model of computation with a quantum random access memory (QRAM), obtaining what we call the word QRAM model. Under this model, we prove the following result, exploiting a similarity between bit-parallel techniques and quantum parallelism. Notice that for constant alphabets, this algorithm is linear.

**Theorem 1.** Let $A[1..m]$ and $B[1..n]$ be two strings from an ordered alphabet of size $\sigma$, where $\sigma \leq m \leq n$. The computation of the unit cost edit distance of $A$ and $B$ can be solved in $O(n \log \sigma)$ time using $O(n + m)$ QRAM qubits and $O(\log n)$ working qubits, under read-write word QRAM model with word size $O(\log n)$.

The strategy that we use in proving this theorem consists of two steps. First, we propose a new bit-parallel algorithm for edit distance, that computes the dynamic programming matrix by visiting the counter-diagonals relying solely on Boolean operations. Second, we translate this algorithm into a quantum algorithm, using a general techniques that can be applies also to other bit-parallel algorithms. The advantage of computing the edit distance matrix in a counter-diagonal order is that there are no dependencies within the same counter-diagonal, and we note that this approach has already been adopted [10,19]. Nevertheless, to the best of our knowledge, we are the first to propose an arithmetic-free bit-parallel algorithm for edit distance.

The existence of quantum computers using QRAM has already been deemed possible [7], and has already been used as a model of computation for solving some string problems in sublinear time [6]. In this work, we consider mainly two different models of QRAM, a read-write QRAM and a read-only QRAM. The above result on edit distance holds in the read-write model, and features some characteristics, not common among quantum algorithms.

- Our algorithm for edit distance does not use Grover’s Search [8]; instead, it deterministically retrieves the correct result at the end of the computation.
- The correctness of the algorithm does not depend on the values of the amplitudes in the generated superpositions, as long as they all remain different from zero. This is very relevant in practice, as quantum computers often suffer from environmental noise that makes the superposition unstable.
• The techniques that we propose can be used as a framework for converting bit-parallel algorithms into quantum algorithms. From this perspective, quantum algorithms could be seen as bit-parallel algorithms where the memory word can be as large as we need it to be. Thus, the effect on the time complexity is to reduce any $\lceil m/w \rceil$ factor to a constant, as we show to be the case for the edit distance.

• Although we need linear space to store and access the input strings in QRAM, we use only $O(\log n)$ working qubits.

For the read-only model, we show that there are SETH-hard problems that still admit a quantum linear algorithm. Here, we consider exact pattern matching on a graph, that is, deciding if a pattern string $P$ equals a labeled path in a graph $G = (V, E)$, where labels are placed on the nodes. This problem is quadratic in the classical model under SETH \cite{4}, even if restricted to level DAGs, that is DAGs in which, for every two nodes $v$ and $w$, every path from $v$ to $w$ has the same length. However, this problem admits a linear time algorithm in the read-only QRAM model.

**Theorem 2.** The problem of SMLG on level DAG $G = (V, E)$ and pattern $P$ can be solved in $O(|E| + \sqrt{|P|})$ time using $O(|P|)$ QRAM qubits and $O(\max L)$ working qubits, in the read-only word QRAM model.

Here, given any node $s$ with no in-neighbours, we call level (or depth) the length $d(v)$ of any path from $s$ to $v$; the level $d(G)$ (or depth) of DAG $G$ is then $d(G) = \max\{d(v) : v \in V\} = \max L$.

As a final remark, we stress the fact that the assumptions made in the read-write word QRAM model are not always stronger than other quantum computational models. For instance, consider exact string matching, that is the classic problem of finding an occurrence of a string $P$ as a substring of text $T$, that has a linear time classic solution \cite{9}. There exist quantum algorithms to solve this problem in sublinear time, the best of which run in time $O(\sqrt{|T|}(\log^2 |T| + \log |P|))$ \cite{14}. However, such algorithms assume that a quantum register can store the entire text in one quantum register and the pattern into another, which is analogous of assuming that the text and the pattern fit into a memory word in classic computing. Moreover, these approaches implicitly assume to be able to generate a superposition containing all the substrings of text $T$ of length $|P|$ \cite{14, 16} in sublinear time, or they explicitly assume that all such substrings are part of the input \cite{18}. These assumptions are dropped in our model, that assume the quantum register to be of size $\log n$, and thus able to store at most one character each. We find our model to be more in line with what it is possible to do with memory words in classic computing, and we show that, under our constraints, there is a straightforward algorithm for exact string matching that runs time $O(|P| + \sqrt{|T|})$, thus linear in the pattern length.

The paper is structured as follows. We start by introducing the bit-parallel operations and then derive an arithmetic-free variant of Myers’ algorithm \cite{11}. Then we introduce the read-write word QRAM model and show how to simulate each of the required bit-vector operations with it. This suffices to conclude the main result of the paper. The latter half of the paper delves into the assumptions behind the word QRAM model and studies exact string matching algorithms on a more restricted version of it, that is the read-only word QRAM model.

In what follows, we assume the reader is familiar with the basic notions in quantum computing as covered in textbooks \cite{13}.

# 2 Preliminaries

An *alphabet* $\Sigma$ is a set of *characters*. Throughout the paper we assume $\Sigma$ is ordered, i.e., for each $a, b \in \Sigma$ we can decide if $a < b$. A sequence $A \in \Sigma^n$ is called a *string* and its length is denoted
n = |A|. We denote integers $i, i + 1, \ldots, j$ as interval $[i..j]$ and represent a string $A$ as an array $A[1..n]$, where $A[i] \in \Sigma$ for $1 \leq i \leq n$. String $A[i..j]$ is called a substring and string $A[1..i]$ a prefix of $A$. With bit-vectors discussed next, we use 0-based indexing.

Let $X$ be a $w$-bit integer interpreted as string $X[0..w-1]$ from alphabet $\{0, 1\}$ such that $X = \sum_{i=0}^{w-1} X[i] \cdot 2^i$. We call $X$ a bit-vector. Given two bit-vectors $X$ and $Y$, we define the following Boolean operations $A = X \land Y$, $O = X \lor Y$, and $N = \neg X$ as follows: $A[i] = 1$ iff $X[i] = Y[i] = 1$, $O[i] = 1$ iff $X[i] = 1$ or $Y[i] = 1$, and $N[i] = 1$ iff $X[i] = 0$. When bit-vector content is visualized, we list the most significant bit first, i.e., $X[w-1]X[w-2] \cdots X[0]$. With this in mind, we define the left-shifts $L = X \ll k$ and right-shifts $R = X \gg k$ as follows: $L[i+k] = X[i]$ and $R[i] = X[i+k]$. Here values out of the domain of the bit-vectors are assumed to be 0. Logarithms are assumed to be in base two: $\log n = \log_2 n$.

3 Edit distance

In the unit cost edit distance problem, the goal is to compute the minimum amount of insertions, deletions, and substitutions to convert one string into another. Using bit-parallelism, this problem can be solved in $O(n[m/w])$ time [11], where $m$ and $n$ are the lengths of the input strings.

At first sight, converting Myers’ algorithm [11] into its quantum counterpart appears challenging: the algorithm exploits arithmetic operations (namely, $+$) to break global dependencies inside its bit-vectors. Our conversion technique does not support such operations. Thus, in order to use our technique, we first show that Myers’ algorithm can be adjusted to work using only Boolean logics and bit-shifts. This adjustment is simple: instead of columnwise bit-vectors, we can use counter-diagonally bitvectors. In fact, Wright’s bit-parallel approach [19] already uses the same idea achieving $O(n[m/w] \log \sigma)$ running time. However, that algorithm uses arithmetic operations. That is, our approach can be seen as combining the two earlier bit-parallel algorithms and dropping the arithmetic parts while doing their merge.

3.1 Arithmetic-free bit-parallel edit distance

Consider the edit distance matrix $d_{0..m,0..n}$, where $d_{ij}$ is the edit distance of $A[1..i]$ and $B[1..j]$, and $A[1..m]$ and $B[1..n]$ are the input strings, where $m \leq n$. Then $d_{mn}$ is the edit distance of $A$ and $B$, and it can be computed using the recurrence $d_{ij} = \min(d_{i-1,j-1} + \delta(A[i], B[j]), d_{i-1,j} + 1, d_{i,j-1} + 1)$, where $\delta(a, b) = 0$ if $a = b$ and $\delta(a, b) = 1$ if $a \neq b$. The matrix can be filled in constant time per cell after initializing the zeroth column and row to $d_{i,0} = i$ and $d_{0,j} = j$, respectively, for integers $0 \leq i \leq m$ and $0 \leq j \leq n$. Such computation is possible in any evaluation order such that the neighboring three cells are already computed. To speed-up this $O(mn)$ time evaluation into $O(n[m/w])$, Myers [11] exploited the properties that $d_{i,j} - d_{i-1,j-1} \in \{0, 1\}$ and $d_{i,j} - d_{i-1,j}, d_{i,j} - d_{i,j-1} \in \{-1, 0, 1\}$. Thus, one can encode the differences between adjacent cells with few bit-vectors and it remains to see how one can update output bit-vectors (current frontier of computation) from input bit-vectors (previous frontier of computation).

Myers [11] defined the following sets of Boolean variables to derive the rules for this computation: $Pv_j(i) = 1$ iff $d_{i,j} - d_{i-1,j} = 1$, $Mv_j(i) = 1$ iff $d_{i,j} - d_{i-1,j} = -1$, $Ph_j(i) = 1$ iff $d_{i,j} - d_{i,j-1} = 1$, $Mh_j(i) = 1$ iff $d_{i,j} - d_{i,j-1} = -1$, $Xv_j(i)$, and $Xh_j(i)$. The latter two sets are auxiliary variables.
used for computing the rest. Myers derived the following formulas:

\[
\begin{align*}
Xv_j(i) &= (A[i] = B[j]) \text{ or } Mv_{j-1}(i) \\
Xh_j(i) &= (A[i] = B[j]) \text{ or } Mh_j(i-1) \\
Ph_j(i) &= Mv_{j-1}(i) \text{ or not } (Xh_j(i) \text{ or } Pv_{j-1}(i)) \\
Mh_j(i) &= Pv_{j-1}(i) \text{ and } Xh_j(i) \\
Pv_j(i) &= Mh_j(i-1) \text{ or not } (Xv_j(i) \text{ or } Ph_j(i-1)) \\
Mv_j(i) &= Ph_j(i-1) \text{ and } Xv_j(i)
\end{align*}
\]

Now one could compute values for \(w\) rows in parallel on a fixed column, except that \(Mh_j(i)\) depends on \(Mh_j(i-1)\) through the second and fifth formulas. To solve this dependency, Myers derived an alternative formula for \(Xh_j(i)\) with dependency on a range of other variables through a bitwise + operation. We note that this dependency can be solved without creating such long-range dependency simply by a coordinate change: Consider replacing each set of variables of the form \(Y_k\) with \(Y_{i+j}(i)\). We now get the following formulas:

\[
\begin{align*}
Xv_{i+j}(i) &= (A[i] = B[j]) \text{ or } Mv_{i+j-1}(i) \\
Xh_{i+j}(i) &= (A[i] = B[j]) \text{ or } Mh_{i+j-1}(i-1) \\
Ph_{i+j}(i) &= Mv_{i+j-1}(i) \text{ or not } (Xh_{i+j}(i) \text{ or } Pv_{i+j-1}(i)) \\
Mh_{i+j}(i) &= Pv_{i+j-1}(i) \text{ and } Xh_{i+j}(i) \\
Pv_{i+j}(i) &= Mh_{i+j-1}(i-1) \text{ or not } (Xv_{i+j}(i) \text{ or } Ph_{i+j-1}(i-1)) \\
Mv_{i+j}(i) &= Ph_{i+j-1}(i-1) \text{ and } Xv_{i+j}(i)
\end{align*}
\]

Now each set of formulas of the form \(Y_k(*)\) only depends on auxiliary already computed variables and those of the form \(Y_{k-1}(*)\). Thus, for each counter-diagonal \(i+j\) one can evaluate \(w\) of these values in parallel using Boolean logics and shifts by one. Algorithm 1[] shows the steps assuming \(m < w\). We use 0-base indexing for the bit-vectors and store values starting from the first row of \(d_{0..m,0..n}\); Figure 1 illustrates the mapping. We assume all bit-vectors being initially zero. Initialization of the values corresponding to the zeroth column and the zeroth row of the original matrix are integrated to the computation: at each left shift, the zeroth row needs to be initialized, and after each computation, the zeroth column needs to be reset to avoid side-effects from values not corresponding to the original matrix range. In addition, before computing counter-diagonals \(k \in [1..m+n]\) we need to set \(Pv_0 = 1\). Computation of the final edit distance is conducted by adding the differences in the last row (now shifted accordingly); the return value on the call with \(k = m + n\) gives the distance.

To solve approximate string matching, it is sufficient to interpret pattern \(P[1..m]\) as \(A\) and text \(T[1..n]\) as \(B\), and change the initialization and interpretation of the score computation: For initialization, it is sufficient to remove \(V1\) at lines 5 and 6 of Algorithm 1[] as this is the same as if the zeroth column of the original matrix is initialized to 0s, and thus a match can start at any position in \(T\). If score \(d \leq \kappa\) in Algorithm 1 at counter-diagonal \(k = j + m - 1\), then there is a substring of \(T\) ending at position \(j\) within edit distance at most \(\kappa\) from \(P\), where \(\kappa\) is a given threshold for approximate pattern matching. It is straightforward to extend these edit distance and approximate string matching algorithms to larger strings, similarly as in the original Myers' algorithm yielding \(O(n[m/w])\) time complexity. However, this assumes we have preprocessed the
Figure 1: The interpretation of the bit-vector values with respect to the original dynamic programming matrix (left) and with respect to the project matrix after coordinate change (right).

bit-vector Eq used in Algorithm 1, and as we will see next, its computation is the bottleneck of the approach.

Algorithm 1: Computing one counter-diagonal \( k = i + j \) in the arithmetic-free variant of Myers bit-parallel algorithm.

**Input:** Bit-vectors and score \( d \) from previous step, Eq\(_k\), and \( m \).

**Output:** Bit-vectors and score \( d \) of this step.

1. \( Xv_k = \text{Eq}_k \lor \text{Mv}_{k-1} \);
2. \( Xh_k = \text{Eq}_k \lor (\text{Mh}_{k-1} \ll 1) \);
3. \( Ph_k = \text{Mv}_{k-1} \lor \neg(Xh_k \lor \text{Pv}_{k-1}) \);
4. \( Mh_k = \text{Pv}_{k-1} \land Xh_k \);
5. \( Pv_k = (\text{Mh}_{k-1} \ll 1) \lor \neg(Xv_k \lor (\text{Ph}_{k-1} \ll 1)) \lor 1 \);
6. \( Mv_k = ((\text{Ph}_{k-1} \ll 1) \lor 1) \land Xv_k \);
7. \( P_k = (\text{Mh}_{k-1} \ll 1) \lor 1 \);
8. \( Mv_k = \text{Mv}_{k-1} \lor \neg(1 \ll k) \);
9. if \( k < m - 1 \) then
10. \[ d = m; \]
11. else
12. \[ d = d + ((\text{Ph}_k \gg (m - 1)) \land 1) - ((\text{Mh}_k \gg (m - 1)) \land 1); \]
13. return \( Ph_k, \text{Mh}_k, \text{Pv}_k, \text{Mv}_k, d \);

3.2 Preprocessing for counter-diagonalwise edit distance computation

Algorithm 1 assumes we have computed for each counter-diagonal \( k = i + j \) a bit-vector Eq\(_k[0..m-1]\) such that Eq\(_k[i - 1] = A[i] = B[k - i] \) for \( i \in [1..m] \). Let us compute indicator bit-vectors \( I^A_c[0..m-1] \) for \( c \in \Sigma \) such that \( I^A_c[i - 1] = 1 \) iff \( A[i] = c \). Similarly, let us compute indicator bit-vectors \( I^B_c[0..n-1] \) for \( c \in \Sigma \) such that \( I^B_c[j - 1] = 1 \) iff \( B[n - j] = c \). When \( 0 < k < n \), Eq\(_k = \lor c \in \Sigma I^A_c \land (I^B_c \gg (n - k)) \). When \( n \leq k < m + n \), Eq\(_k = \lor c \in \Sigma I^A_c \land (I^B_c \ll (k - n)) \). Notice that on strings longer than the computer word size, it is sufficient to consider the \( \lceil m/w \rceil \) integers representing the corresponding shifted portion of the larger string \( B[1..n] \) against \( A[1..m] \) at each step, so that this algorithm can be implemented to run in \( O(\Sigma |n/m/w|) \) time.

To reduce the dependency on the alphabet size, we can replace the indicator bit-vectors with a hierarchical representation. For this, as we assume the alphabet is ordered, we can first map it into
Lemma 1. Algorithm [4] is correct regardless of what values are stored in Eq[k][i], Ph[k][i], Mh[k][i], Pvk[k][i + 1], Mvk[k][i + 1], for i > k, and in Eq[k][i], Ph[k][i], Mh[k][i], Pvk[i], Mvk[i], for i ≤ k – n.

Proof. We prove the lemma for Pvk[i], the reasoning for Eq[k][i], Mvk[i], Ph[k][i] and Mh[k][i] is analogous.

Value Ph[k][i] maps to Ph[j][i], j = k – i, in the original matrix. For i > k, and thus k < m, we have j < 0, that is Ph[j][i] falls out of the bounds of the matrix and is ignored by the algorithm. For i ≤ k – n, and thus for k ≥ n, value Ph[k][i] still maps out of the bounds of the original matrix, since j = k – i ≥ n. Hence, it is ignored by the algorithm.

For all the other cases, notice that, in Algorithm [4] Pvk[i] depends on Mvk[k][i − 1], Ph[k−1][i − 1] (line 5) and Mvk−1[i] (line 7), and they have to be computed correctly. Mvk−1[i − 1] and Ph[k−1][i − 1] always falls into the bounds of the original matrix, thus from the correctness of Myer’s algorithm it follows that they store the right values. Instead, Mvk−1[i] falls outside the bounds of the original matrix if i = k. However, Mvk−1[k] is correctly initialized at the end of iteration k − 1 (line 5).

Theorem 3. Let A[1..m] and B[1..n] be two strings from an ordered alphabet of size σ, where σ ≤ m ≤ n. The computation of the unit cost edit distance of A and B can be solved in O(n[m/w] log σ) time under word RAM model with word size w using only Boolean operations and bit-shifts.

4 Quantum Preliminaries

In quantum computing, data is represented in quantum bits (qubits), the quantum analogue to classical bits. A qubit can be in two states, denoted as |0⟩ = (1 0) and |1⟩ = (0 1) but, unlike a classical bit, it can also be a combination of the two states, a superposition: |ψ⟩ = α |0⟩ + β |1⟩. The complex values α and β are called the amplitudes of |ψ⟩. Measuring a qubit in superposition will result in either |0⟩ or |1⟩ with probabilities |α|² and |β|², respectively.

Note that this notation can easily be generalised to integer states |n⟩ using the tensor product between the quantum states of the binary representation on n: |n⟩ = ⊗i∈binary(n) |i⟩. We use this for indexing the QRAM, see subsection 4.2 for a detailed description.

4.1 Quantum gates

The quantum analogue to the logical NOT operation is the matrix:

\[
X := \begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix},
\]

such that \( X |ψ⟩ = α |1⟩ + β |0⟩ \). We write \( X(Q) \) with the meaning of applying quantum NOT gate \( X \) on qubit \( Q \).
The application of the X gate can be controlled by another qubit, resulting in the quantum controlled-not:

$$CX := \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 
\end{bmatrix}.$$ 

We write $CX(Q, R)$, where $Q$ is the control qubit and $R$ is the target qubit. In superposition $|\psi\rangle = \sum_{i,j \in \{0,1\}^2} \alpha_{ij} |Q_i\rangle |R_j\rangle$, this means that $R_j$ is flipped if and only if $Q_i = 1$.

Similarly, the application of the CX gate can also be controlled by a third qubit. Resulting in the Toffoli gate:

$$CCX := \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 
\end{bmatrix}.$$ 

Given the superposition $|\psi\rangle = \sum_{i=0}^{n-1} \alpha_i |P_i\rangle |Q_i\rangle |R_i\rangle$, we write $CCX(P, Q, R)$ to denote the Toffoli gate that flips qubit $R_i$ if and only if both $P_i = Q_i = 1$.

If $Q$ is a quantum register of $|Q|$ qubits, we write $CQ = n(R)$, where $C$ is a generalized CX, that is a CX that flips $R_i$ if and only if $Q_i = n$.

Additionally, the operation $Set_0 I = k(Q)$ sets the substate $Q_i$ to 0: $Q_i \leftarrow 0$. Note that $Set_0 I = k(Q)$ and $C I = k(Q)$ can be done in time $O(\log n)$ where $n$ is the largest index $|n\rangle$ of $I$, given that $I$ is constructed from qubits, e.g. in a bucket-brigade QRAM [1,7,15].

### 4.2 Quantum RAM

We assume to have QRAM $M$ able to use a quantum register as an index to access classical data, which is stored as $N$ memory cells $M_1, M_2, \ldots, M_N$. Given quantum register $I$, the read operation from $M$ using $I$ as index is defined as follows [7]:

$$|\psi\rangle = \sum_{i=1}^{n} \alpha_i |I_i\rangle \xrightarrow{\text{RAM read}} |\psi'\rangle = \sum_{i=1}^{n} \alpha_i |I_i\rangle |M_{I_i}\rangle.$$ 

While memory $M$ remains unchanged, state $|\psi\rangle$ transits to $|\psi'\rangle$.

To achieve our goals, we also need to specify the meaning of storing data into the QRAM. Given superposition state $|\psi\rangle = \sum_{i=0}^{n-1} \alpha_i |I_i\rangle |\phi_i\rangle$, we define the write operation using quantum registers $I$ as index, and $\phi$ as the data to store. In this case, state $|\psi\rangle$ remains unchanged, while memory $M$ transits to $M'$:

$$M = \{M_1, M_2, \ldots, M_N\} \xrightarrow{\text{RAM write}} M' = \{M'_1, M'_2, \ldots, M'_N\}$$

where, for $1 \leq j \leq N$,

$$M'_j = M_j \quad \text{if} \quad j \notin \{I_1, I_2, \ldots, I_{n-1}\}$$

$$M'_j = \phi_i \quad \text{if} \quad j = I_i, \; 0 \leq i \leq n - 1.$$ 

A read operation from QRAM $M = \{M_1, M_2, \ldots, M_N\}$ using quantum register $Q$ as an index is denoted as $R \leftarrow M[Q]$. This means to load the content of $M$ into qubit $R$, that is $R_i \leftarrow M_{Q_i}$,
where arrow ← indicates assignment. Symmetrically, \( M |Q| ← R \) denotes a write operation of the content of \( R \) into QRAM \( M \) using \( Q \) as index, that is \( M_{Q_i} ← R_i \).

With these operations on QRAM, we define the two following computational models.

**Definition 1** (read-write word QRAM). *In the read-write word QRAM model, it is possible to perform read and write operations on a QRAM. Moreover, on inputs of size \( n \), we assume to have a memory word of size \( O(\log n) \). That is, every arithmetic or Boolean operation and every quantum gate application on quantum registers of size \( O(\log n) \) takes constant time \( O(1) \).*

**Definition 2** (read-only QRAM). *The read-write word QRAM model is the read-write word QRAM model where write operations in the QRAM are not possible.*

### 4.3 Superposition notation

We now introduce an alternative way to represent a superposition, that helps understanding the intuition behind bit-wise operations. Consider a quantum system where \( I \) is a quantum register and \( P, Q \) and \( R \) are single qubits. Given quantum superposition state \( |\psi\rangle = \sum_{i=0}^{n-1} \alpha_i |I_i\rangle |P_i\rangle |Q_i\rangle |R_i\rangle \) and QRAM \( M = \{ M_1, M_2, \ldots, M_N \} \), we represent them as

\[
\begin{array}{cccc}
I & P & Q & R \\
|I_1\rangle |P_1\rangle |Q_1\rangle |R_1\rangle & |0\rangle : M_1 \\
|I_2\rangle |P_2\rangle |Q_2\rangle |R_2\rangle & |1\rangle : M_2 \\
\vdots & \vdots & \vdots & \vdots \\
|I_n\rangle |P_n\rangle |Q_n\rangle |R_n\rangle & [N - 1] : M_n
\end{array}
\]

We dropped the amplitudes and the + sign since they do not add relevant information for our algorithms. In this representation, each row encodes a set of variables that can be updated in parallel with the other rows. Quantum register \( I \) serves as an index, and stores all the different numbers from 1 to \( n \). Thus, substates \( P_i, Q_i \) and \( R_i \) represents what would be single bits of bit-vectors in classical computing.

### 5 Quantum Operations

We now define some quantum subroutines able to encode in a superposition the effect that bit-wise operations have in bit-parallel computations. We also define the concept of “correct simulation”, which is necessary to prove the correctness of this translation.

**Definition 3.** Consider superposition

\[
|\psi\rangle = \sum_{i=0}^{n-1} \alpha_i |I_i\rangle |Q_i^{(1)}\rangle |Q_i^{(2)}\rangle \cdots |Q_i^{(k)}\rangle |R_i\rangle
\]

of quantum registers \( I, Q^{(1)}, \ldots, Q^{(k)}, k \in \mathbb{N} \), where \( I_i \) is for every \( 0 \leq i \leq n - 1 \). Let \( V^{(1)}, \ldots, V^{(k)} \) be bit-vectors of \( n \) positions, and let \( F(V^{(1)}, \ldots, V^{(k)}) \) be a Boolean formula where only bit-wise operators \( \neg, \wedge \) and \( \vee \) are used. We say that a quantum subroutine \( qF(Q^{(1)}, \ldots, Q^{(k)}, R) \) correctly simulates bit-wise operation \( W ← F(V^{(1)}, \ldots, V^{(k)}) \) if the following holds: if invariant \( Q_i^{(j)} = V^{(j)}[i], 0 \leq i \leq n - 1, 1 \leq j \leq k, \) holds true, then it holds true also after applying both operations \( qF(Q^{(1)}, \ldots, Q^{(k)}, R) \) and \( R ← F(V^{(1)}, \ldots, V^{(k)}) \).
Not, and, or. Let $U$, $V$ and $W$ be three classical bit-vectors, $P$ and $R$ three qubits, and $I$ a quantum register. We define the following quantum subroutines as sequences of quantum operations, so that input qubits $P$ and $Q$ are left unaltered and the result is stored in $R$.

\[
\begin{align*}
q\text{NOT}(Q,R) &= R \leftarrow 0; \; X(Q); \; CX(Q,R); \; X(Q) \\
q\text{AND}(P,Q,R) &= R \leftarrow 0; \; CCX(P,Q,R) \\
q\text{OR}(P,Q,R) &= R \leftarrow 0; \; X(P); \; CX(Q,R); \; X(R); \; X(P); \; X(Q)
\end{align*}
\]

The above subroutines correctly simulate classical bit-wise operations $V \leftarrow \neg V$, $W \leftarrow U \land V$ and $W \leftarrow U \lor V$, as we now formalize.

**Lemma 2.** Subroutines $q\text{NOT}(Q)$, $q\text{AND}(P,Q,R)$ and $q\text{OR}(P,Q,R)$ correctly simulate classical bit-wise operations $V \leftarrow \neg V$, $W \leftarrow U \land V$ and $W \leftarrow U \lor V$, respectively.

**Proof.** Consider superposition $\sum_{i=0}^{n-1} \alpha_i |I_i\rangle |P_i\rangle |Q_i\rangle |R_i\rangle$ where we assume $P_i = U[i]$, $Q_i = V[i]$ and $R_i = W[i]$, $0 \leq i \leq n - 1$, to hold true. All the subroutines start with operation $R \leftarrow 0$, which initializes qubit $R$ to zero, leaving us with $\sum_{i=0}^{n-1} \alpha_i |I_i\rangle |P_i\rangle |Q_i\rangle |R_i\rangle$. We continue by analyzing each subroutine individually. All the following reasoning applies to each $0 \leq i \leq n - 1$.

**Subroutine** $q\text{NOT}(Q)$. Operation $X(Q)$ flips every $Q_i$ to $\neg Q_i$, and consequently operation $CX(Q,R)$ flips $R_i$ to 1 if and only if $\neg Q_i = 1$, thus $R_i = \neg Q_i$. The second application of $X(Q)$ flips back each $\neg Q_i$ to its original value $Q_i$. Thus, the final superposition that we obtain is $\sum_{i=0}^{n-1} \alpha_i |I_i\rangle |P_i\rangle |Q_i\rangle |R_i\rangle$. Operation $W \leftarrow \neg V$ leaves bit-vectors $U$ and $V$ unaltered, while now $W[i] = \neg V[i]$. The lemma is proved since $Q_i = V[i]$.

**Subroutine** $q\text{AND}(P,Q,R)$. The Toffoli gate flips $R_i$ when $P_i = 1$ and $Q_i = 1$, that is $R_i = P_i \land Q_i$. Operation $W \leftarrow U \lor V$ leaves bit-vectors $U$ and $V$ unaltered, while now $W[i] = P[i] \lor V[i]$, proving the lemma since $P_i = U[i]$ and $Q_i = V[i]$.

**Subroutine** $q\text{OR}(P,Q,R)$. The first application of $X(P)$ and $X(Q)$ generates superposition $\sum_{i=0}^{n-1} \alpha_i |I_i\rangle |\neg P_i\rangle |\neg Q_i\rangle |R_i\rangle$.

The Toffoli gate generates superposition $\sum_{i=0}^{n-1} \alpha_i |I_i\rangle |\neg P_i\rangle |\neg Q_i\rangle |\neg P_i \land \neg Q_i\rangle$. Finally, $X(R)$ and the second application of $X(P)$ and $X(Q)$ set $R_i = (\neg P_i \land \neg Q_i) = P_i \lor Q_i$ and restore the initial values of $P$ and $Q$: $\sum_{i=0}^{n-1} \alpha_i |I_i\rangle |P_i\rangle |Q_i\rangle |P_i \lor Q_i\rangle$. Operation $W \leftarrow U \lor V$ leaves bit-vectors $U$ and $V$ unaltered, while now $W[i] = P[i] \lor V[i]$, proving the lemma since $P_i = U[i]$ and $Q_i = V[i]$.

**Left shift.** Following the same logic, we also simulate bit-wise shifts $V \ll 1$ and $V \gg 1$, but in this case it is necessary to read and write from the QRAM using register $I$ and a new register $J$.

\[
\begin{align*}
qLShift(I,Q) &= J \leftarrow I - 1 \mod n; \; M[I] \leftarrow Q; \; Q \leftarrow M[J]; \; Set0I_{i=0}(Q) \\
qRShift(I,Q) &= J \leftarrow I + 1 \mod n; \; M[I] \leftarrow Q; \; Q \leftarrow M[J]; \; Set0I_{i=n-1}(Q)
\end{align*}
\]

We assume $J$ to be an auxiliary quantum register allocated as we need it. The correctness of this subroutine is shown in Lemma 3 and an intuition is given in the following example for the left shift (the right shift is symmetric). We assume to have already performed operation $J \leftarrow I - k$, and vector $U$ and qubit $P$ are shown for reference.

| $UV$ | $UV$ | $I$ | $J$ | $P$ | $Q$ | $M$ | $I$ | $J$ | $P$ | $Q$ | $M$ | $I$ | $J$ | $P$ | $Q$ | $M$ |
|------|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 1  | $\ll 1$ | 0 0 | 0 | $\{0\}$ | $\{3\}$ | $\{0\}$ | $\{1\}$ | 0 | $\{0\}$ | $\{3\}$ | $\{0\}$ | $\{1\}$ | 0 | 1 | $\ll 1$ | $\{0\}$ | $\{1\}$ | $\{0\}$ |
| 1 0  | $\ll 1$ | 1 1 | 1 | $\{1\}$ | $\{0\}$ | $\{1\}$ | $\{0\}$ | 1 | $\{0\}$ | $\{1\}$ | $\{0\}$ | $\{1\}$ | 1 | 0 | $\ll 1$ | $\{0\}$ | $\{1\}$ | $\{0\}$ |
| 1 1  | $\ll 1$ | 1 0 | $\{2\}$ | $\{1\}$ | $\{1\}$ | $\{1\}$ | 2 | 0 | $\{2\}$ | $\{1\}$ | $\{1\}$ | 2 | 1 | 1 | $\ll 1$ | $\{0\}$ | $\{1\}$ | $\{0\}$ |
| 1 1  | $\ll 1$ | 1 1 | $\{3\}$ | $\{2\}$ | $\{1\}$ | $\{1\}$ | 3 | 0 | $\{3\}$ | $\{2\}$ | $\{1\}$ | $\{1\}$ | 3 | 1 | $\ll 1$ | $\{0\}$ | $\{1\}$ | $\{0\}$ |
Lemma 3. Subroutines $qLShift(I, Q)$ and $qRShift(I, Q)$ correctly simulate classical bit-wise operations $V \leftarrow V \ll 1$ and $V \leftarrow V \gg 1$, respectively.

Proof. We prove the lemma for $qLShift(I, Q)$; $qRShift(I, Q)$ is symmetrical. Starting from superposition $\sum_{i=0}^{n-1} \alpha_i |I_i\rangle |Q_i\rangle |Q'_i\rangle$ where $I_i = i$, $J_i = 0$ and assuming $Q_i = V[i]$ for every $0 \leq i \leq n - 1$, we apply $J \leftarrow I - 1$ and $M[J] \leftarrow Q$, obtaining superposition $\sum_{i=0}^{n-1} \alpha_i |I_i\rangle |J_i\rangle |Q_i\rangle |Q'_i\rangle$ and memory $M$ where $M_i = Q_i$. Applying $Q \leftarrow M[J]$ equals to $Q \leftarrow M[I - 1]$, that is $Q_i \leftarrow M[I - 1]$. Since $M_i = Q_i$, we have that $Q_i \leftarrow Q_{i-1}$, with the exception of $Q_0$ which is set to zero. This is consistent with $V \leftarrow V \ll 1$, that sets $V[i] = V[i - 1]$. That is, if $Q[i - 1] = V[i - 1] = x$ before performing the shift, then $Q[i] = x$ and $V[i] = x$ after performing the shift.

We also introduce an additional left-shifting subroutine to perform a cyclic shift by $k$ positions.

\begin{align*}
qLShift(I, Q, k) &= J \leftarrow I - k \mod n; \quad M[I] \leftarrow Q; \quad Q \leftarrow M[J] \\
qRShift(I, Q, k) &= J \leftarrow I + k \mod n; \quad M[I] \leftarrow Q; \quad Q \leftarrow M[J]
\end{align*}

In order perform this operation in constant time, we cannot afford to reset the first $k$ positions to zero. This prevents us from proving that $qKShift$ correctly simulates a similar shift in the classical algorithm but, as we will see later, this operation can anyway be used to correctly compute values Eq(k).

Result propagation. The last operation that we need is specific to quantum computing and it is what allows us to retrieve the correct result at the end of the computation. Consider superposition $\sum_{i=0}^{n-1} \alpha_i |I_i\rangle |Q_i\rangle |Q'_i\rangle$ of quantum registers $I$, $Q$ and $Q'$, where $Q'_i = 0$ for every $0 \leq i \leq n - 1$. With $n - 1$ special CNOT operations (that set to zero instead of flipping), we can obtain $Q_i = 0$ for $0 \leq i \leq n - 2$, leaving $Q_{n-1}$ unaltered. Then, we copy $Q$ to $Q'$ and we perform $qRShift(I, Q')$ and $Q \leftarrow Q + Q'$. We keep performing $qRShift(I, Q')$ and $Q \leftarrow Q + Q'$ $n - 1$ times, and the end result is that every substrate of $Q$ now stores $Q_{n-1}$, which we can retrieve by simply measuring $Q$. We call this subroutine $ResProp(I, Q)$, and we define it formally in Algorithm 2 assuming $Q'$ to be an ancilla qubit allocated as we need it. The following is an example of what happens in the superposition, assuming $n = 4$ and that we already set $Q_0, \ldots, Q_{n-2}$ to zero.

| $I$ | $Q$ | $Q'$ | $I$ | $Q$ | $Q'$ | $I$ | $Q$ | $Q'$ | $I$ | $Q$ | $Q'$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | $Q_3$ | $Q_3$ | 3 | $Q_3$ | 4 | 4 | $Q_3$ | 0 | 0 | 0 | 0 |

Algorithm 2: Subroutine $ResProp(I, Q, Q')$ to retrieve the value stored in $Q_{n-1}$ in superposition $\sum_{i=0}^{n-1} \alpha_i |I_i\rangle |Q_i\rangle |Q'_i\rangle$. Quantum gate $C_{I=k}^{\text{zero}}(Q)$ sets $Q_i = 0$ if $I_i = k$.

\begin{enumerate}
\item[] \textbf{Input:} Quantum registers $I$, $Q$ and $Q'$ in superposition $\sum_{i=0}^{n-1} \alpha_i |I_i\rangle |Q_i\rangle |Q'_i\rangle$, where $Q'_i = 0$ for every $0 \leq i \leq n - 1$.
\item[] \textbf{Output:} superposition $\sum_{i=0}^{n-1} \alpha_i |I_i\rangle |Q_{n-1}\rangle |Q'_i\rangle$, where $Q'_i = 0$ for every $0 \leq i \leq n - 1$
\item[] for $k = 0$ \textbf{upto} $n - 2$
\item[] \textbf{do}
\item[] $C_{I=k}^{\text{zero}}(Q)$;
\item[] \textbf{for} $k = n - 2$ \textbf{downto} 0 \textbf{do}
\item[] $qRShift(I, Q')$;
\item[] $Q \leftarrow Q + Q'$;
\end{enumerate}
The end result is that every substate of \( Q \) now stores \( Q_{n-1} \), which we can retrieve by simply measuring \( Q \).

6 Quantum Linear Algorithm for Edit Distance

In Section 5 we showed how a classical bit-parallel algorithm is able to compute the edit distance using only bit-wise logical operations and shifts. In Section 5 we provided the tools to translate each such operation into an analogous quantum operation, that uses quantum parallelism instead of bit-wise parallelism. Thus, we obtain a quantum linear time algorithm for edit distance as a translation of the bit-parallel algorithm.

**Theorem 1.** Let \( A[1..m] \) and \( B[1..n] \) be two strings from an ordered alphabet of size \( \sigma \), where \( \sigma \leq m \leq n \). The computation of the unit cost edit distance of \( A \) and \( B \) can be solved in \( O(n \log \sigma) \) time using \( O(n + m) \) QRAM qubits and \( O(\log n) \) working qubits, under read-write word QRAM model with word size \( O(\log n) \).

**Proof.** We initialize our quantum system by applying Hadamard gate \( H(\log n) \) to quantum register \( I \) consisting of \( \log n \) qubits, thus generating superposition \( \sum_{i=0}^{n-1} |I_i\rangle \), where \( I_i = i \). In the following, we assume to store number values in quantum registers of \( O(\log n) \) qubits and, if not specified differently, they are stored as the same value in every substate of the superposition. For example, to track iteration index \( k \), we assume to have quantum register \( K \) such that \( \sum_{i=0}^{n-1} |I_i\rangle |K_i\rangle \) and \( K_i = k \) for every \( 0 \leq i \leq n-1 \).

Then, thanks to Lemma 2 and 3, we can translate every operation in Algorithm 1 between lines 1 and 8 into a sequence of \( \text{qNOT}, \text{qAND}, \text{qOR}, \text{qLShift} \) and \( \text{qRShift} \) operations.

To handle the if and else branches at lines 10 and 12, we assume to have values \( d, m \) and \( k \) stored in quantum registers \( D, M \) and \( K \), respectively. The not trivial operation is the update of \( d \) in the else branch. We replace this operation with \( D \leftarrow D + \text{Ph}^{(k)} - \text{Mv}^{(k)} \), where \( \text{Ph}^{(k)} \) and \( \text{Mv}^{(k)} \) are qubits representing the homonymous bit-vectors, with the correctly computed values of iteration \( k \). This new operation potentially updates every \( D_i \) in superposition

\[
\sum_{i=0}^{n-1} |I_i\rangle |D_i\rangle |\text{Ph}_{i}^{(k)}\rangle |\text{Mh}_{i}^{(k)}\rangle
\]

with a different value. However, notice that \( D_{n-1} \) will always be updates with the right values of the distance computed so far. As explained at the end of this proof, at the end of the computation we use this fact together with operation \( \text{ResProp} \) to retrieve the correct value of the edit distance.

The only thing that is left is to handle the computation of values \( \text{Eq}_k \). To this end, we use one qubit to represent \( \text{Eq}_k \), one for each \( I_x^A \) and one for each \( I_x^B \), and then we translate operation \( \text{Eq}_k = \land_{x \in [0..b-1]} I_x^A \land (I_x^B \gg (n - k)) \), for \( 0 < k < n \), and operation \( \text{Eq}_k = \land_{x \in [0..b-1]} I_x^A \land (I_x^B \ll (k - n)) \), for \( n \leq k < m+n \), using quantum subroutines \( \text{qRShift}(I, I_x^B, n - k) \) and \( \text{qLShift}(I, I_x^B, k - n) \), respectively. As we noted when we introduced them, these operations do not correctly simulate a right (left) shift of \( k \) positions in a bit-vector, because \( \text{qRShift} \) (\( \text{qLShift} \)) cycles the values instead of setting the first \( n - k \) (last \( k - n \)) positions to zero. However, Lemma 4 guarantees that this translation into quantum operations maintains the correctness of the algorithm.

At the end of the computation, the correct value of the edit distance can be retrieved by applying subroutine \( \text{ResProp}(I, D) \) and then measuring register \( D \).

In the classical algorithm, these operations act on a bit-vectors of size \( w \) representing the columns of the dynamic programming matrix after the coordinate change and the indicator bit-vectors. These operations cost constant time when \( m \leq n \leq w \). In the word QRAM model with
word size $O(\log n)$, we can always make sure that this is the case, since we can always generate a superposition of $n$ substates using quantum registers of $O(\log n)$ qubits.

7 String Matching in Labeled Graphs

Consider exact pattern matching on a graph, that is, consider deciding if a pattern string $P \in \Sigma^*$ equals a labeled path in a graph $G = (V, E)$, where $V$ is the set of nodes and $E$ is the set of edges. Here we assume the nodes $v$ of the graph are labeled by $\ell(v) \in \Sigma$ and a path $v_1 \rightarrow v_2 \rightarrow \ldots v_t$, $(v_i, v_{i+1}) \in E$ for $1 \leq i < t$, spells string $\ell(v_1)\ell(v_2)\ldots\ell(v_t)$. There is an OVH lower bound conditionally refuting an $O(|P||E|^{1-\epsilon})$ or $O(|P|^{1-\epsilon}|E|)$ time solution \cite{4}. This conditional lower bound holds even if graph $G$ is a level DAG \cite{4}: for every two nodes $v$ and $w$, holds the property that every path from $v$ to $w$ has the same length. On DAGs, this string matching on labeled graphs (SMLG) problem can be solved in $O(|P|/w||E|)$ time \cite{17}, so the status of this problem is identical to that of approximate pattern matching on strings. However, the simplicity of the bit-parallel solution for SMLG on level DAGs enables a direct connection to quantum computation. Under the read-only word QRAM model, we turn the bit-parallel solution into a quantum algorithm that solves SMLG on level DAGs with high probability in $O(|E| + |V|)$ time, breaking through the classical quadratic conditional lower bound. The bit-parallel strategy allow us to use a limited number of qubits, that is, our qubits space complexity on level DAGs is $O(|P|)$ QRAM qubits and $O(maxL)$ working qubits, where $maxL$ is the number of nodes of the level with the highest number nodes. We remark that $maxL$ is always less or equal to the width of the graph, that is the minimum number of non-disjoint paths needed to cover all the nodes.

7.1 The Classical Shift-And Algorithm

We first introduce the classic shift-and algorithm for matching a pattern against a text and generalize it to work on graphs. Then, we show how the bit-vector data structure of that algorithm can be represented as a superposition of a logarithmic number of qubits. This approach allows to achieve better performances than the brute force algorithm for special types of graphs.

In the shift-and algorithm, we use bit vector $D$ of the same length of pattern $P$ to represent which of its prefixes are matching the text during the computation. Assuming integer-alphabet $\Sigma$, we also initialize bidimensional array $B$ of size $|P| \times |\Sigma|$ so that $B[j][c] = 1$ if and only if $P[j] = c$, and $B[j][c] = 0$ otherwise. The algorithm starts by initializing vector $D$ to zero and array $B$ as specified above. Then, we scan whole text $T$ performing the next four operations for each $T[i]$, $i \in [1, n]$:  

1. $D \leftarrow D \lor 1$;

2. $D \leftarrow D \land B[T[i]]$;

3. if $D[|P|] = 1$, return yes;

4. $D \leftarrow D \ll 1$.

In a labeled DAG $G = (V, E)$, each node $v$ has a single-character label $\ell(v)$. We generalize the shift-and algorithm to labeled DAGs by computing a different bit-vector $D_v$ for each node $v \in V$, initializing them to zero. Consider a topological-order visit of DAG $G$. When visiting node $v$, each bit-vector $D_w$ of its in-neighbor $w \in in(v)$ represents a set of prefixes of $P$ matching a path in the graph ending in $w$. Thus, we merge all of this information together by taking the bit-wise or of all
**Algorithm 3:** Computing one counter-diagonal $k = i+j$ as in Algorithm I using quantum operations instead of bit-parallel ones. Qubit $One$ is initialized with operation $C_{I=0}(One)$. Qubits $A$ and $A'$ are ancillae.

**Input:** Bit-vectors and score $d$ from previous step, Eq$_k$, and $m$.

**Output:** Bit-vectors and score $d$ of this step.

```plaintext
/* Xv$_k$ = Eq$_k$ ∨ Mv$_{k-1}$; */
1 qOR(Eq$_k$, Mv$_{k-1}$, Xv$_k$);

/* Xh$_k$ = Eq$_k$ ∨ (Mh$_{k-1}$ ≪ 1); */
2 A ← Mh$_{k-1}$; qLShift($I$, A);

/* Ph$_k$ = Mv$_{k-1}$ ∨ ¬(Xh$_k$ ∨ Pv$_{k-1}$); */
4 qOR(Xh$_k$, Pv$_{k-1}$, A);

/* A ← A'; qOR(Mv$_{k-1}$, APh$_k$); */
6 qOR(A, One, $A'$);

/* Mn$_{k'}$ = (Ph$_k$ ≪ ($m-1$)) ∧ Xh$_k$; */
7 qAND(Pv$_{k-1}$, Xv$_k$, Mh$_k$);

/* P$_v$ = (Mh$_{k-1}$ ≪ 1) ∨ ¬(Xv$_k$ ∨ (Ph$_{k-1}$ ≪ 1) ∨ 1); */
8 A ← Ph$_{k-1}$; qLShift(A);

/* A ← A'; qOR(Xv$_k$, A, A'); */
10 qOR(A', One, A');

/* Mv$_k$ = ((Ph$_{k}$ ≪ 1) ∨ 1) ∧ Xv$_k$; */
14 A ← Ph$_{k}$; qLShift(A);

/* Mv$_{k'}$ = Mv$_{k}$ ∨ ¬(1 ≪ k); */
19 qNOT(A, $A'$);

if $K < m-1$ then
  22 $D$ ← $m$;
else
  24 $A$ ← Ph$_k$; qLShift($A$);
  25 qAND($A$, One, $A'$);
  26 $A$ ← $A'$; $A'$ ← Mh$_k$;
  27 qLShift($A'$);
  28 qAND($A'$, One, $A''$);
  29 $A$ ← $A'$;
  30 $D$ ← $A$ + $A''$;
end
31 return Ph$_k$, Mh$_k$, Pv$_k$, Mv$_k$, $D$;
```
Figure 2: The adaptation of the classical algorithm for matching pattern \( P \) in *level* DAG \( G \). Each bit-vector \( D_v \) represent the result after the merging of the bit-vectors of the in-neighbours of \( v \) and before the shifting.

of the in-neighbors of \( v \), that is we replace Operation 1 with \( D_v \leftarrow 1 \lor \bigvee_{w \in \text{in}(v)} D_w \). Operations 2, 3 and 4 are performed as before.

### 7.2 Quantum Bit-Parallel Algorithm for Level DAGs

We make the classic techniques work in a quantum setting for a special class of DAGs, which we call *level* DAGs. A level DAG is a DAGs in which, for every two nodes \( v \) and \( w \), every path from \( v \) to \( w \) has the same length. Given any node \( s \) with no in-neighbours, we call *level* (or *depth*) the length \( d(v) \) of any path from \( s \) to \( v \); the level \( d(G) \) (or depth) of DAG \( G \) is then \( d(G) = \max\{d(v) : v \in V\} \).

The DAG from Figure 2 is a level DAG.

Our approach aims to represent each bit-vector \( D_v \) with a single qubit set up in a proper superposition, and realize the bit-wise operations as parallel operations across such superposition. To this end, we use quantum register \( I \) of size \( O(\log |P|) \) to generate the superposition, one quantum register \( D^{(v)} \) for each node \( v \in V \) consisting of one qubit, ancilla qubits \( A \), \( R \) and \( R' \) and additional register \( C \) of size \( O(\log |P|) \). Moreover, we assume to have access to a read-only QRAM.

Operation 1 is now \( D_v \leftarrow 1 \lor \bigvee_{w \in \text{in}(v)} D_w \), which can be easily translated into quantum operations with a series of qOR subroutines.

Operation 2 is now \( D_v \leftarrow D_v \land B[I][\ell(v)] \), and can be translated as an access to the QRAM and a qAND.

Operation 3 is now converted into updating the last sub-state \( R_{|P|−1} \) of ancilla qubit \( R \) as \( R_{|P|−1} \leftarrow R_{|P|−1} \lor D^{(v)}_{|P|−1} \), leaving \( R_0, \ldots, R_{|P|−2} \) unchanged. We implement this operation by
performing \( \text{CX}_{I \leftarrow |P\rangle}(A) \) and then computing \( R' \leftarrow (R' \lor D_v) \land A \) (translated as a \( q\text{OR} \) and a \( q\text{AND} \)) and finally \( R \leftarrow R \lor R' \) (as a \( q\text{OR} \)). We then reset ancilla qubits \( A \) and \( R' \) to \( |0\rangle \). The need for this procedure will be clear later, and the fact that an occurrence was found or not will be reported by Grover’s Search phase at the end of the algorithm. Operation 4 is not translated as a \( q\text{LSHIFT} \), because we have to avoid write operations in the QRAM. Thus, we simply perform \( I_j \leftarrow I_j + 1 \), thus cyclically shifting all the values across the superposition, and effectively performing the update \( D^{(v)}_{j+1} \leftarrow D^{(v)}_j \) on the values of \( D^{(v)} \), where \( j \) and \( j + 1 \) are intended modulo \( |P\rangle \).

We have to change the overall structure of the algorithm to handle the fact that each time that we perform the new shift operation, we are actually shifting all the vectors, not just the current \( D^{(v)} \). This poses a problem when we process a node at a certain level and another node of the same level has already been processed. To overcome this situation, we visit the graph one level at a time, and we update all the \( D^{(v)} \) vectors of that level before performing the shift. Algorithm 4 shows the entire procedure.

As last step of the algorithm, we run Grover’s search that uses as oracle function an identity function returning the value of register \( R \). Thus, if there is at least one \( R_j = |1\rangle \), the measurement at the end of the search will return \( R = |1\rangle \) with high probability. Otherwise, if for every \( j \) holds that \( R_j = |0\rangle \), than the measurement of Grover’s search will yield \( R_j = |0\rangle \).

To prove the correctness of Algorithm 4, we formalise the key properties in the following lemmas.

We need to guarantee two invariant conditions that have to hold after executing the inner \textit{for}-loop (line 12).

**Lemma 4.** After running the inner \textit{for}-loop in Algorithm 4 \( i \) times (and before performing \( I \leftarrow I + 1 \)), for every \( D^{(v)} \) such that \( d(v) = i \) holds that \( D^{(v)}_j = |1\rangle \) if and only if there exists a path in \( G \) ending at \( v \) and matching \( P[0, j] \).

**Proof.** We proceed by induction on the number \( i \) of times that we run the inner \textit{for}-loop.

**Base case**, \( i = 0 \). In this case, vectors \( D^{(v)} \) such that \( d(v) = 0 \) are those with in-degree zero, initialized by the first \textit{for}-loop, while the inner \textit{for}-loop has never run. For each such \( v \), the initialization \textit{for}-loop initially sets \( D^{(v)}_0 \) to \( |1\rangle \) (\( \text{CX} \) operation), and then resets it to \( |0\rangle \) if and only if it does not match \( B[|0\rangle, \ell(v)] \) (\( \text{and} \) operation). Thus, the only \( D^{(v)}_0 \) set to \( |1\rangle \) are the ones matching \( P[0, 0] \), while \( D^{(v)}_1, \ldots, D^{(v)}_P \) are correctly left to \( |0\rangle \).

**Inductive case**, \( 1 \leq i \leq d(G) \). After running the inner \textit{for}-loop \( i \) times, we have to perform \( I \leftarrow I + 1 \) before running the \textit{for}-loop for the \( i + 1 \)-th time. Assuming the inductive hypothesis, operation \( I \leftarrow I + 1 \) makes every \( D^{(w)} \) with \( d(w) = i \) such that \( D^{(w)}_i = |1\rangle \) if and only if there is a match for \( P[1, j - 1] \) in \( G \) ending at \( w \). Then we run the \textit{for}-loop for the \( i + 1 \)-th time. Notice that in any previous iteration of the \textit{for}-loop it could never be the case that \( d(v) = i + 1 \), and we update \( D^{(w)} \) if and only if \( d(v) = i + 1 \), thus every \( D^{(w)}_j \) is currently set to \( |0\rangle \). At this point, for each \( v \) such that \( d(v) = i + 1 \), we set \( D^{(v)}_0 \) to \( |1\rangle \) with operation \( \text{CX}(0, v, D_v) \). Then, we perform the bitwise \textit{or} and the bitwise \textit{and}, after which \( D^{(v)}_j = |1\rangle \) if and only if there is a match for \( P[1, j - 1] \) in \( G \) ending at \( w \) and \( \ell(v) \) matches \( P[j] \), where \( (w, v) \in E \). This means that \( D^{(v)}_j = |1\rangle \) if and only if a match for \( P[1, j - 1] \) ending at \( w \) can be extended to a match for \( P[1, j] \) ending at \( v \) using edge \((w, v)\). \( \square \)

**Lemma 5.** After running the inner \textit{for}-loop in Algorithm 4 \( i \) times (and before performing \( I \leftarrow I + 1 \)), there exists at least one \( j \) such that \( R_j = |1\rangle \) if and only if there exists at least one \( v \) such that \( P \) has a match in \( G \) ending at \( v \), where \( d(v) \leq i \).

16
Algorithm 4: The algorithm for testing whether pattern string $P$ has a match in level DAG $G$, running in $O(|E| + \sqrt{|P|})$ time.

**Input:** Level DAG $G$, pattern string $P$.

**Output:** yes, if $P$ has a match in $G$, no otherwise.

1. $I \leftarrow |0\rangle$;
2. for $v \in V$ do
3.   $D^{(v)} \leftarrow |0\rangle$;
4.   $A \leftarrow |0\rangle$;
5.   $R \leftarrow |0\rangle$;
6.   $R' \leftarrow |0\rangle$;
7.   $H^\otimes \log |P| I$;
8. for $s \in V$ such that $d(s) = 1$ do
9.   $CX_{I=0}(D^{(s)})$;
10. $D^{(s)} \leftarrow D^{(s)} \land B[I][\ell(v)]$;
11. for $l \leftarrow 1$ to $d(G)$ do
12.   for $v \in V$ such that $d(v) = l$ do
13.     $CX_{I=0}(D^{(v)})$;
14.     for $w \in \text{in}(v)$ do
15.       $qOR(D^{(v)}, D^{(w)}, Q')$;
16.       $D^{(v)} \leftarrow Q'$;
17.     $qAnd(D^{(v)}, B[I][\ell(v)], Q')$;
18.     $D^{(v)} \leftarrow Q'$;
19.     $CX_{I=|P|-1}(A)$;
20.     $qAND(D^{(v)}, A, R')$;
21.     $qOR(R, R', Q')$;
22.     $R \leftarrow Q'$;
23.     $CX_{I=|P|-1}(A)$;
24.     $R' \leftarrow A$;
25.     $A \leftarrow |0\rangle$;
26.     $I \leftarrow I + 1$;
27. Run Grover’s search with oracle function $f(I, D^{(1)}, \ldots, D^{(|V|)}, A, R, R') = R$;
28. Measure $R$ and return yes if $R = |1\rangle$, no otherwise.
**Proof.** Base case, \( i = 0 \). In this case, nodes \( v \) such that \( d(v) = 0 \) are those with in-degree zero, while the inner for-loop has never run. Since we are visiting only single-node paths and we are assuming that pattern \( P \) has length at least two, there can be no match for \( P \) ending at these nodes. Correctly, every \( R_j = |0\rangle \).

Inductive case, \( 1 \leq i \leq d(G) \). Right before running the inner for-loop for the \( i + 1 \)-th time, every \( D^w \) with \( d(w) = i \) is such that \( D^w_j = |1\rangle \) if and only if there is a match for \( P[i, j-1] \) in \( G \) ending at \( v \). This follows from the same reasoning as in the proof of Lemma 4 and from the inductive hypothesis. While running the inner for-loop for the \( i + 1 \)-th time, we observe that during an iteration for a single node \( v \) the statement of Lemma 4 restricted to only \( v \) holds after line 18. This means that \( D^v_{|P|-1} = |1\rangle \) if and only if \( P[0, |P| - 1] \) has a match ending at \( v \). At this point we set up register \( A \) with operation \( CX(0, I, D_v) \), and then we run \( R' \leftarrow D^v \wedge A \). This way, \( R'_0 \ldots R'_{|P|-2} \) are set to \( |0\rangle \) since \( A_0 = |0\rangle \ldots A_{|P| - 2} = |0\rangle \), while \( R'_{|P|-1} = D^v_{|P|-1} \) since \( A_{|P|-1} = |1\rangle \). Operation \( R \leftarrow R' \) leaves \( R_0 \ldots R_{|P|-2} \) unchanged, while \( R_{|P|-1} = 1 \) is set to \( |1\rangle \) either if it was already \( |1\rangle \) or if \( D^v_{|P|-1} = |1\rangle \). Then, \( A \) and \( R' \) are reset to \( |0\rangle \). If \( R_{|P|-1} \) was already \( |1\rangle \), then the inductive hypothesis guarantees that at some earlier iteration a path matching \( P \) was already found. If we turned \( R_{|P|-1} \) to \( |1\rangle \) in this iteration, we know that this happened if and only if \( D^v_{|P|-1} = |1\rangle \), which means that there is a path ending at \( v \) that matches \( P[0, |P| - 1] \) (that is, the full pattern \( P \)). Either way, we are guaranteed that \( R_{|P|-1} = |1\rangle \) if and only if there is a path matching \( P \) that ends at node \( v \), where \( d(v) \leq i \).

The correctness of the algorithm follows from the previous lemmas combined with few additional observations.

**Theorem 4.** Given a pattern string \( P \) of length at least 2 and level DAG \( G \), if \( P \) does not have a match in \( G \) Algorithm 4 returns no; if \( P \) has a match in \( G \) Algorithm 4 returns yes with high probability.

**Proof.** After running the inner for-loop of Algorithm 4 \( d(G) \) times we exit also the outer loop, having visited all the nodes. Thanks to Lemma 5, we know that there is at least one sub-state \( R_j \) of register \( R \) set to \( |1\rangle \) if and only if there is a node \( v \) such that \( P \) has a match in \( G \) ending at \( v \), where \( d(v) \leq d(G) \). moreover, condition \( d(v) \leq d(G) \) means any node in the graph. Thus, if \( P \) has no match in \( G \), every sub-state \( R_j \) is set to \( |0\rangle \), the measurement will output a \( |0\rangle \), thus the algorithm will return no. if \( P \) has at least one match in \( G \), there is at least one \( R_j = |1\rangle \), which will be amplified by Grover’s search. In turn, the algorithm will measure a \( |1\rangle \) and return yes with high probability.

Finally, the time complexity of our algorithm is linear in the size of the graph.

**Theorem 5.** The time complexity of Algorithm 4 is \( O(|E| + |V|) \) in the read-only word QRAM model, and the space complexity is \( O(|P|) \) QRAM qubits and \( O(|V|) \) working qubits.

**Proof.** The \( or \) operation that we perform to update \( D^v \) in the inner for-loop processes every edge once, and for each edge we perform a qubit \( or \). The nested for-loops scan every node once and, for each node, we perform a constant number of quantum operations. We assume that \( \log |P| \) fits in a quantum memory word (as it would be the case in a classical setting). Since we are scanning each node and edge once and each time we do a constant number of operations, the time complexity is \( O(|E| + |V|) \). Matrix \( B \) occupy \( O(|P| \Sigma) \) qubits, where \( \Sigma \) is the alphabet, and we use \( |V| \) qubits \( D_v \). Since we assume a constant alphabet, the space complexity is \( O(|P| + |V|) \).
We can improve Algorithm 4 to run in the same time complexity but using only \( O(maxL) \) qubits, where \( maxL \) is the number of nodes of the level with the highest number nodes. Notice that \( maxL \leq W \) always, where \( W \) is the minimum number of non-disjoint paths needed to cover all the nodes, that is the width of the DAG.

**Theorem 2.** The problem of SMLG on level DAG \( G = (V, E) \) and pattern \( P \) can be solved in \( O(|E| + \sqrt{|P|}) \) time using \( O(|P|) \) QRAM qubits and \( O(maxL) \) working qubits, in the read-only word QRAM model.

*Proof.* We can modify Algorithm 4 to satisfy the statement of the theorem. Instead of using a qubit \( D_v \) for each node \( v \in V \), we use \( 2maxL \) qubits, \( X_1, \ldots, X_{maxL} \) and \( Y_1, \ldots, Y_{maxL} \). This is because, by definition of width, for each level \( l \) there are at most \( maxL \) nodes \( v \) such that \( d(v) = l \). We also use classical arrays \( A_X \) and \( A_Y \), both \( f \) size \( |V| \), which map every node \( v \) to the qubit representing its bit-vector (we use two arrays for symmetry with the qubits but, since they will store the same values, one would be enough). We were already implicitly using a similar data structure in the unmodified Algorithm 4 but in that case we could assume to initialize it once at the beginning and never change it later. Here, we need to handle these arrays explicitly.

At each iteration \( i \) of the first *for*-loop (line 3), we visit each node \( s \in V \) such that \( d(s) = 1 \) and we set \( A_X[s] \leftarrow i \), which means that qubit \( X_i \) represents the bit-vector for node \( s \). We then perform the same operations of this *for*-loop as before, replacing \( D_s \) with \( X_i \).

At each iteration \( j \) of the inner *for*-loop (line 4), we set \( A_X[s] \leftarrow j \) and \( A_Y[s] \leftarrow j \), then we perform the same operations as before replacing \( D' \) with \( Y_{A_Y[v]} \) and \( D'' \) with \( X_{A_X[w]} \). After the execution of this entire *for*-loop, right before or right after \( I \leftarrow I + 1 \), we switch the role of \( X \) and \( Y \) by setting \( X_k \leftarrow Y_k, Y_k \leftarrow |0 \rangle \), for each \( 1 \leq k \leq maxL \).

Notice that the positions of \( A_X \) and \( A_Y \) that we access at iteration \( j \) are those of the current node, initialized at the beginning of the same iteration, and of the in-neighbors of said node, which have been initialized at some previous iteration. Thus, the positions of \( A_X \) and \( A_Y \) are always correctly initialized when we access them. Moreover, the update \( X_k \leftarrow Y_k, Y_k \leftarrow |0 \rangle \), for each \( 1 \leq k \leq maxL \), preserves the following invariant: after performing this update \( i \) times, thus having run the inner *for*-loop \( i \) times, qubits \( X_1, \ldots, X_{maxL} \) store the bit-vectors of the nodes of level \( j \). This holds true after the initialization *for*-loop, and can be seen to hold true by an induction argument on \( i \) similar to the ones in the previous lemmas. We conclude that, thanks to this invariant, the algorithm is correct.

The new update operation \( X_k \leftarrow Y_k, Y_k \leftarrow |0 \rangle \), for each \( 1 \leq k \leq maxL \), takes at most \( O(maxL) \) time for each level \( l \), thus \( O(|V|) \) time in total, not affecting the overall time complexity. Instead, the number of qubits that we are using is \( 2maxL \) plus a constant, that is the ancilla qubits and register \( I \); thus, we use \( O(|P| + maxL) \) qubits in total. This is because data structure \( B \) requires \( O(|P|\Sigma) \) space to be stored, where \( \Sigma \) is the alphabet, that we assume to have constant size. \( \square \)

### 8 String Matching in Plain Text under Word QRAM

In this section, we further motivate our choice of model of computation by proposing an analysis of exact string matching under our read-only word QRAM model.

The exact string matching problem is to decide if a pattern string \( P \) appears as a substring of a text string \( T \). As discussed in the introduction, this problem can be solved in \( O(|P| + |T|) \) time \([5]\) in the classical models of computation and in sublinear time \([14, 16, 18]\), for example in \( O(\sqrt{|T||\log^2 |T| + \log |P||}) \) time \([14]\), with quantum computing. These approaches assume that the text and the pattern could entirely fit into a quantum register each, they implicitly assume
to be able to generate a superposition containing all the substrings of text $T$ of length $|P|$ in sublinear time, or they explicitly assume that all such substrings are part of the input [18]. We find that these assumptions do not relate properly with classic computing, where we assume that a memory word can store at most one character $T[i]$ of the text. Moreover, in exact string matching we typically do not have access to all the substrings of $T$ of length $|P|$ as part of the input, and it is not clear how to generate a superposition storing them. These are indeed the main reasons behind the proposal of our read-write word QRAM model, and thus we would like to reanalyze exact string matching in light of our model.

In read-write word QRAM model, the assumption is to have $T$ and $P$ stored in quantum registers (actually, QRAM registers) as

$$|T[0]||T[1]| \ldots |T[n-1]| |P[0]| |P[1]| \ldots |P[m-1]|.$$  

Using additional register $|S|$ initialized to $|1\rangle$, and register $|I\rangle$ initialized to $|0\rangle$, we can solve exact string matching in time $O(|P| + \sqrt{|T|})$ as follows. We prepare quantum register $|I\rangle$ in an equally balanced superposition spanning all the text positions, that is $1/\sqrt{2^{|T|}} \sum_{i=0}^{|T|-1} |I_i\rangle$, where $I_i = i$ (assuming $|T|$ to be a power of 2 for simplicity). Each individual state $|i\rangle$ in the superposition represents a computation starting at position $i$ in the text. In each of these computations, we scan $T[i..i + |P| - 1]$ and try to match each character with $P[0..|P| - 1]$, storing the intermediate results of such comparisons in register $|S\rangle$. More precisely, at iteration $j$, $0 \leq j \leq |P| - 1$, we compute a logic and between $T[i + j] = P[j]$ and the value in $|S\rangle$. This procedure is correct because at the first step $|S\rangle = |1\rangle$, thus we simply compute and store $T[i + j] = P[j]$ in $|S\rangle$. At iteration $j$, we assume by induction that register $|S\rangle$ stores a 1 if $T[i\ldots i + j - 1] = P[0\ldots j-1]$, thus computing $T[i + j] = P[j] \land S$ tells us if we are extending a match or not. At the end, we can run Grover’s search algorithm to retrieve the superposition states where $|S\rangle = |1\rangle$, and then measure register $|I\rangle$ to locate the ending position of a match. We illustrate this procedure in Algorithm 5.

**Algorithm 5**: An algorithm for solving exact string matching in plain text, which achieves $O(|P| + \sqrt{|T|})$ time complexity thanks to the use of QRAM.

| Input: Text $T$, pattern string $P$ |
| Output: A position $idx$ of $T$ where a match for $P$ ends, if any |
| 1  $|\psi\rangle \leftarrow |T\rangle |P\rangle |I\rangle |S\rangle$; |
| 2  $|S\rangle \leftarrow |1\rangle$; |
| 3  $H^{|T|} |I\rangle$; |
| 4  for $j \in [0, |P| - 1]$ do |
| 5  $|S\rangle \leftarrow |T[I] = P[j] \land S\rangle$; |
| 6  $|I\rangle \leftarrow |I \oplus 1\rangle$; |
| 7  Run Grover’s search algorithm with oracle $f(T, PI, S) = S$; |
| 8  $idx \leftarrow$ measurement of register $|I\rangle$; |
| 9  $idx \leftarrow idx - 1$; |
| 10 return $idx$; |

In the following example we have $T = \text{AABABABB}$ and $P = \text{ABA}$. We demonstrate how Algorithm 5 works by simulating it on text $T$ and pattern string $P$. In the following example, we depict one term of the superposition per row, omitting the amplitudes and the plus sign, as the amplitudes

In the following example we have $T = \text{AABABABB}$ and $P = \text{ABA}$. We demonstrate how Algorithm 5 works by simulating it on text $T$ and pattern string $P$. In the following example, we depict one term of the superposition per row, omitting the amplitudes and the plus sign, as the amplitudes
are all the same and the plus sign does not add important information.

\[
\begin{align*}
|T\rangle |P\rangle |0\rangle |1\rangle & \rightarrow \\
|T\rangle |P\rangle |0\rangle |1\rangle & \rightarrow |T\rangle |P\rangle |0\oplus 1\rangle |1\wedge T|0\rangle = P[0] \rightarrow |T\rangle |P\rangle |1\oplus 1\rangle |1\wedge T|1\rangle = P[1] \\
|T\rangle |P\rangle |1\rangle |1\rangle & \rightarrow |T\rangle |P\rangle |1\oplus 1\rangle |1\wedge T|1\rangle = P[0] \rightarrow |T\rangle |P\rangle |2\oplus 1\rangle |1\wedge T|2\rangle = P[1] \\
|T\rangle |P\rangle |2\rangle |1\rangle & \rightarrow |T\rangle |P\rangle |2\oplus 1\rangle |1\wedge T|2\rangle = P[0] \rightarrow |T\rangle |P\rangle |3\oplus 1\rangle |0\wedge T|3\rangle = P[1] \\
|T\rangle |P\rangle |3\rangle |1\rangle & \rightarrow |T\rangle |P\rangle |3\oplus 1\rangle |1\wedge T|3\rangle = P[0] \rightarrow |T\rangle |P\rangle |4\oplus 1\rangle |1\wedge T|4\rangle = P[1] \\
|T\rangle |P\rangle |4\rangle |1\rangle & \rightarrow |T\rangle |P\rangle |4\oplus 1\rangle |1\wedge T|4\rangle = P[0] \rightarrow |T\rangle |P\rangle |5\oplus 1\rangle |0\wedge T|5\rangle = P[1] \\
|T\rangle |P\rangle |5\rangle |1\rangle & \rightarrow |T\rangle |P\rangle |5\oplus 1\rangle |1\wedge T|5\rangle = P[0] \rightarrow |T\rangle |P\rangle |6\oplus 1\rangle |1\wedge T|6\rangle = P[1] \\
|T\rangle |P\rangle |6\rangle |1\rangle & \rightarrow |T\rangle |P\rangle |6\oplus 1\rangle |1\wedge T|6\rangle = P[0] \rightarrow |T\rangle |P\rangle |7\oplus 1\rangle |0\wedge T|7\rangle = P[1] \\
|T\rangle |P\rangle |7\rangle |1\rangle & \rightarrow |T\rangle |P\rangle |7\oplus 1\rangle |1\wedge T|7\rangle = P[0] \rightarrow |T\rangle |P\rangle |0\oplus 1\rangle |0\wedge T|0\rangle = P[1]
\end{align*}
\]

**Acknowledgements**

We would like to thank Sabrina Maniscalco for giving useful feedback on technical details of the quantum computing framework.

**References**

[1] Srinivasan Arunachalam, Vlad Gheorghiu, Tomas Jochym-O’Connor, Michele Mosca, and Priyaa Varshinee Srinivasan. On the robustness of bucket brigade quantum ram. *New Journal of Physics*, 17(12):123010, 2015.

[2] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018.

[3] Mahdi Boroujeni, Soheil Ehsani, Mohammad Ghodsi, MohammadTaghi Hajiaghayi, and Saeed Seddighin. Approximating edit distance in truly subquadratic time: Quantum and mapreduce. *J. ACM*, 68(3):19:1–19:41, 2021.

[4] Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 55:1–55:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[5] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. Limit on the speed of quantum computation in determining parity. *Phys. Rev. Lett.*, 81:5442–5444, Dec 1998.

[6] François Le Gall and Saeed Seddighin. Quantum meets fine-grained complexity: Sublinear time quantum algorithms for string problems. In Mark Braverman, editor, *13th Innovations in Theoretical Computer Science Conference, ITCS 2022, January 31 - February 3, 2022*,
[7] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Phys. Rev. Lett.*, 100:160501, Apr 2008.

[8] Lov K. Grover. A fast quantum mechanical algorithm for database search. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219. ACM, 1996.

[9] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[10] Muhammad Murtaza Yousaf, Muhammad Umair Sadiq, Laeeq Aslam, Waqar Ul Qounain, and Shahzad Sarwar. A Novel Parallel Algorithm for Edit Distance Computation. *Mehran University Research Journal of Engineering and Technology*, 37(1):223–232, January 2018.

[11] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, 1999.

[12] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[13] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[14] Pradeep Niroula and Yunseong Nam. A quantum algorithm for string matching. *npj Quantum Information*, 7(1):1–5, 2021.

[15] Alexandru Paler, Oumarou Oumarou, and Robert Basmadjian. Parallelizing the queries in a bucket-brigade quantum random access memory. *Physical Review A*, 102(3):032608, 2020.

[16] H Ramesh and V Vinay. String matching in $\tilde{O}(\sqrt{n} + \sqrt{m})$ quantum time. *Journal of Discrete Algorithms*, 1(1):103–110, 2003. Combinatorial Algorithms.

[17] Mikko Rautiainen, Veli Mäkinen, and Tobias Marschall. Bit-parallel sequence-to-graph alignment. *Bioinform.*, 35(19):3599–3607, 2019.

[18] Kapil Kumar Soni and Ashwini Kumar Malviya. Design and analysis of pattern matching algorithms based on quram processing. *Arabian Journal for Science and Engineering*, 46(4):3829–3851, 2021.

[19] Alden H. Wright. Approximate string matching using within-word parallelism. *Softw. Pract. Exp.*, 24(4):337–362, 1994.