From Bit-Parallelism to Quantum String Matching for Labelled Graphs

Massimo Equi ✉ 📷
Department of Computer Science, University of Helsinki, Finland

Arianne Meijer-van de Griend ✉ 📷
Department of Computer Science, University of Helsinki, Finland

Veli Mäkinen ✉ 📷
Department of Computer Science, University of Helsinki, Finland

---- **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. A classic example is computing the edit distance of two strings of length n, which can be solved in $O(n^2/w)$ time. In a reasonable classical model of computation, one can assume $w = \Theta(\log n)$, and obtaining significantly better speed-ups is unlikely in the light of conditional lower bounds obtained for such problems.

In this paper, we study the connection of bit-parallelism to quantum computation, aiming to see if a bit-parallel algorithm could be converted to a quantum algorithm with better than logarithmic speed-up. We focus on *string matching in labeled graphs*, the problem of finding an exact occurrence of a string as the label of a path in a graph. This problem admits a quadratic conditional lower bound under a very restricted class of graphs (Equi et al. ICALP 2019), stating that no algorithm in the classical model of computation can solve the problem in time $O(|P||E|^{1-\epsilon})$ or $O(|P|^{1-\epsilon}|E|)$. We show that a simple bit-parallel algorithm on such restricted family of graphs (level DAGs) can indeed be converted into a realistic quantum algorithm that attains subquadratic time complexity $O(|E|\sqrt{|P|})$.

1 Introduction

Exact string matching problem is to decide if a pattern string P appears as a substring of a text string T. In the classical models of computation, this problem can be solved in $O(|P| + |T|)$ time [10]. Different quantum algorithms for this basic problem have been developed [13, 14, 16], resulting into different solutions, the best of which finds a match in $O(\sqrt{|T|}(\log^2 |T| + \log |P|))$ time [13] with high probability. These assume the pattern and text are stored in quantum registers, requiring thus $O(|P|+|T|)$ qubits to function. Moreover, these approaches may rely on applying a linear number of quantum gates in parallel on different qubits. For example, Niroula and Nam [13] perform $O(\log(|T|))$ rounds of parallel swaps, executing $O(|T|)$ swaps in parallel per round.

In the classical models of computation, an analogy for these assumptions is to assume that the text has been preprocessed for subsequent queries. For example, one can build a Burrows-Wheeler transform -based index structure for the text in time $O(|T|)$ [4], assuming $T \in \{1, 2, \ldots, \sigma\}^*$, where $\sigma \leq |T|$. Then, one can query the pattern from the index in $O(|P| \log \log \sigma)$ time [4, Theorem 6.2]. In this light, quantum models can offer only limited benefit over the classical models for exact string matching.

Motivated by this difficulty in improving linear-time solvable problems using quantum approaches, let us consider problems known to be solved in quadratic time. For example, approximate string matching problem is such a problem: decide if a pattern string P is within edit distance k from a substring of a text string T, where edit distance is the number of single symbol insertions, deletions, and substitution needed to convert a string to another. This problem can be solved using *bit-parallelism* in $O(\lceil |P|/w \rceil |T|)$ time [11], under the Random Access Memory (RAM) model with computer word size w. A reasonable assumption is that $w = \Theta(\log |T|)$, so that this model reflects the capacity of classical computers. Thus, when $|P| = |T| = n$, this bit-parallel algorithm for approximate string matching takes time at least $\Omega(n^{2-\epsilon})$ for all $\epsilon > 0$, as $\log n = o(n^\epsilon)$ for all $\epsilon > 0$. It is believed that this quadratic bound cannot be significantly improved, as there is a matching conditional lower bound saying that if approximate pattern matching could be solved in time $O(n^{2-\epsilon})$ with some $\epsilon > 0$, then the Orthogonal Vector Hypothesis (OVH) and thus the Strong Exponential Time Hypothesis (SETH) would not hold [2]. As these hypotheses are about classical models of computation, it is natural to ask if the quadratic barrier could be broken with quantum computation.

In this quest for breaking the quadratic barrier, we study another problem with a bit-parallel solution and a conditional lower bound. 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 $v_1 \to v_2 \to \cdots v_t$, $(v_i, v_{i+1}) \in E$ for $1 \leq i < t$, spells string $\ell(v_1)\ell(v_2)\cdots\ell(v_t)$. There is an OVH lower bound conditionally refuting an $O(|P||E|^{1-\epsilon})$ or $O(|P|^{1-\epsilon}|E|)$ time solution [7]. This conditional lower bound holds even if graph G is a *level* DAG: for every two nodes u and v, holds the property that every path from u to v has the same length. On DAGs, this string matching on labeled graphs (SMLG) problem can be solved in $O(\lceil |P|/w \rceil |E|)$ time [15] in the bit-parallel model, 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 connection to quantum computation. We consider a specific model of quantum computation, the Quantum Random Access Memory (QRAM) model [8], in which we have access to "quantum arrays", and we assume that integer values like $|P|$, $|V|$ or $|E|$ fit into a (quantum) memory word. Under this model, we turn the bit-parallel solution into a quantum algorithm that solves SMLG on level DAGs with high probability in $O(|E|\sqrt{|P|})$ time, breaking through the classical quadratic conditional lower bound.

Classical conditional lower bounds are not new to be broken by quantum computing. For example, the quadratic Orthogonal Vectors problem itself can be solved in subquadratic time (linear using QRAM) using quantum computing. This is not the only problem to have a better-than-quadratic solution in the quantum realm [17]. Nevertheless, to the best of our knowledge, we are the first to propose a subquadratic time algorithm for SMLG, even if restricted to a specific class of graphs. Moreover, the translation of a bit-parallel strategy to a quantum-parallel one is an original technique, and we are not aware of any other work utilising it.

An earlier work [6] provided a quantum algorithm solving SMLG in time $O(\sqrt{|V||E|}|P|)$. When the graph is non-sparse, that is $|V| = O(\sqrt{|E|})$, the time complexity becomes $O(|E|^{\frac{3}{4}}|P|)$, which is an improvement over classical algorithms. We offer a different kind of trade-off, limiting ourselves to a special class of graphs, but obtaining a better time complexity. We also note that, even if no subquadratic classical algorithm exists for non-sparse graphs, the existing classical reduction from OV [7] produces a sparse level DAG, for which our quantum algorithm runs in subquadratic time.

As mentioned above, in some previous works [13, 14, 16] (and references in [16]) algorithms have been proposed to solve string matching in plain text in the QRAM model, under the assumption that a large number of quantum gates, possibly linear, can be applied in parallel when acting on different qubits. We find this assumption to be too restrictive, as even the classical RAM model does not adopt it, since in such a model of computation many operations would become trivial. Instead, our algorithm works without the need for such an assumption.

The paper is structured as follows. We revisit exact pattern matching and derive a simple quantum algorithm for it, in order to introduce the quantum machinery. Then we give a brute-force quantum algorithm for SMLG, which we later improve on level DAGs. This improvement is based on extending the Shift-And algorithm [3], whose quantum version we extend for level DAGs.

In what follows, we assume the reader is familiar with the basic notions in quantum computing as covered in textbooks [12].

2 Preliminaries

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

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

In *directed labelled graph* (DAG) $G = (V, E, \ell)$, V is the set of nodes, E is the sets of vertices, and $\ell : V \to \Sigma$ is a labelling function that assigns a character of the alphabet to each node. We assume the nodes to be indexed as $v_0, v_1, \ldots v_{n-1}$ in topological order, where $n = |V|$. For $v_i \in V$, $\ell(v_i)$ is its label. Set of nodes $in(v_i) = \{j \mid (v_j, v_i) \in E\}$ contains the indexes of the in-neighbours of v_i, and $D_i = |in(v_i)|$ is the in-degree of v_i. If, for $0 \le d \le D_i - 1$, v_k is the d-th in-neighbour of v_i according to the topological indexing that we defined above, we express this fact using notation $k = in_i(d)$, where $in_i : [0, D_i - 1] \to [0, n-1]$.

In this work, we study the problem of *string matching in labelled graphs*, that consists in finding a match for a pattern string $P[0..m-1]$ in a labelled graphs G over alphabet Σ, where P has a *match* in G if there is a path $v_1, \ldots, v_k$ such that $P = \ell(v_1) \cdots \ell(v_k)$ (we also say that P *occurs* in G, and that $v_1, \ldots, v_k$ is an *occurrence* of P). Notice that if $|P| = 1$, a classic visit of the graph solves the problem in linear time, thus we always assume $|P| \geq 2$.

▶ **Problem 1** (String Matching in Labeled Graphs (SMLG)).
INPUT: *A labeled graph $G = (V, E, L)$ and a pattern string P, both over an alphabet Σ.*
OUTPUT: True *if and only if there is at least one occurrence of P in G.*

3   Quantum Notation and 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 $|0\rangle = \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ and $|1\rangle = \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$ but, unlike a classical bit, it can also be a linear combination of the two states, a *superposition*: $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$. The complex values α and β are called the *amplitudes* of $|\psi\rangle$. Measuring a qubit in superposition will result in either $|0\rangle$ or $|1\rangle$ with probabilities $|\alpha|^2$ and $|\beta|^2$, respectively. Note that this notation can easily be generalised to integer states $|n\rangle$ using the tensor product between the quantum states of the binary representation on n: $|n\rangle = \bigotimes_{i \in binary(n)} |i\rangle$, and in this case we use the term *quantum register*. Throughout the paper, we will use notation $|q\rangle_Q$ to denote that qubit Q is in state $|q\rangle$. We use lower case letters for quantum states and capital letters for qubits.

In this work, we mainly use the NOT gate X, the controlled NOT CX, and the Toffoli gate CCX. We also apply an OR gate, that computes a logical *or* between two qubits and stores the results in a third qubit. This can easily be obtained with a simple combination of X gates with a Toffoli gate.

Furthermore, to define some quantum states, we use Kronecker's delta function $\delta_{x,y}$, which is $\delta_{x,y} = 1$ if $x = y$ and $\delta_{x,y} = 0$ otherwise. Given superposition $|\psi\rangle = \sum_{i=0}^{n-1} \alpha_i |i\rangle_I |\delta_{c,i}\rangle_Q$, the delta function specifies that qubit Q is in state $|1\rangle$ iff $i = c$, as in the following example

$$\sum_{i=0}^{3} \alpha_i |i\rangle_I |\delta_{0,i}\rangle_Q = \alpha_0 |0\rangle_I |1\rangle_Q + \alpha_1 |1\rangle_I |0\rangle_Q + \alpha_2 |2\rangle_I |0\rangle_Q + \alpha_3 |3\rangle_I |0\rangle_Q$$

where I is a quantum register of at least two qubits.

We assume to have a *quantum random access memory* (QRAM) able to use a quantum register as an index to access classical data. Let $m_0, m_1, \ldots, m_{n-1}$ be the data stored in QRAM M. Given quantum register I, the operation that reads data from M into quantum register Q initialized to $|0\rangle$ using I as index is defined as follows [8]:

$$\sum_{i=0}^{n-1} \alpha_i |i\rangle_I |0\rangle_Q \xrightarrow{\text{QRAM read}} \sum_{i=1}^{n} \alpha_i |i\rangle_I |0 \oplus m_i\rangle_Q = \sum_{i=1}^{n} \alpha_i |i\rangle_I |m_i\rangle_Q .$$

Notice that this is a unitary operation, and thus reading the same data into the same register twice will reset such a register to the value it had before performing the reading operation. In terms of time complexity, the execution of the read operation is proportional to the number of qubits in quantum register I. Under the Word-QRAM model with memory-word size $O(\log n)$ for inputs of size n, we can assume to be able to perform a QRAM read operation in $O(1)$, because $O(\log n)$ qubits are enough for register I to index an input of size n. Indeed, this reflect the same assumption of the classical Word-RAM model, where operations on memory words are assumed to be constant.

4 String Matching in Plain Text

A quantum computer, with access to QRAM, can solve the problem of finding an exact match for a pattern string P into a text string T in time $O(|P|\sqrt{|T|})$, with high probability. We explain a simple solution to this problem. Let $|T| = n$ and $|P| = m$, then $T = t_0 t_1 \cdots t_{n-1}$ and $P = p_0 p_1 \cdots p_{m-1}$ are two strings defined over a binary alphabet, that is $t_i, p_j \in \{0, 1\}$ for $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$. We use qubits C_T and C_P initialized to $|0\rangle$ to track the current characters of T and P, and we assume to have the text and the pattern stored in qubits in the following way:

$$|0\rangle_{C_T} |0\rangle_{C_P} |t_0\rangle_{T_0} |t_1\rangle_{T_1} \cdots |t_{n-1}\rangle_{T_{n-1}} |p_0\rangle_{P_0} |p_1\rangle_{P_1} \cdots |p_{m-1}\rangle_{P_{m-1}}.$$

We also use auxiliary qubits $A_{-1}, A_0, A_1 \cdots A_{m-1}$, and quantum registers I, J, and Q, all three of $\log n$ qubits. We initialize A_{-1} and Q to $|1\rangle$, while $A_0, A_1 \cdots A_{m-1}$, I and J are all initialized to $|0\rangle$. We prepare quantum register I in an equally balanced superposition spanning all the text positions, that is $|0\rangle_I \to 1/\sqrt{n} \sum_{i=0}^{n-1} |i\rangle_I$, assuming n to be a power of 2, without loss of generality. If this is not the case, we generate a superposition as large as the first power of two greater than n, then standard techniques can be applied to handle the additional substates, as explained in Appendix A.

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 $T[i..i+m-1]$ and try to match each character with $P[0..m-1]$, storing the intermediate results of such comparisons in registers $A_0, A_1, \cdots, A_{m-1}$. More precisely, at iteration j, $0 \leq j \leq m-1$, we compute a logical *xor* between t_{i+j} and p_j storing the result in C_P via a CX gate with control C_T and target C_P. Then, we apply a X gate to C_P, which now stores $|\neg(t_{i+j} \oplus p_j)\rangle_{C_P} = |t_{i+j} = p_j\rangle_{C_P}$. At this point, we apply a Toffoli gate with controls C_P and A_{j-1}, storing the value in target qubit A_j. We now reset C_T and C_P to $|0\rangle$ by applying to them the same gates again, but in reverse order. As last step in iteration j, we increase both I and J by 1 by performing transformation $1/\sqrt{n} \sum_{i=0}^{n-1} |1\rangle_Q |i\rangle_I |j\rangle_J \to 1/\sqrt{n} \sum_{i=0}^{n-1} |1\rangle_Q |i+1\rangle_I |j+1\rangle_J$ (this of course requires two separate addition operations), where the addition is intended to be modulo 2^n. This allows us to read the next character of the pattern at the next iteration.

After the last iteration, we can run Grover's operator [9] where the marked items are represented by $|a_{m-1,i}\rangle_{A_{m-1}} = |1\rangle$, and then measure register $|I\rangle$ to locate the ending position of a match. Of course, we do not know the exact number of marked items, and we address this issue by guessing the number of items and rerunning the whole algorithm a constant number of times. We illustrate the entire procedure in Algorithm 1.

The algorithm is correct because, after each iteration of the *for* loop, we correctly keep track of the positions of the text that are active matches for the current prefix of the pattern.

▶ **Lemma 1.** *After iteration j of the for loop of Algorithm 1, let qubits I and A_j be in superposition $1/\sqrt{n} \sum_{i=0}^{n-1} |i\rangle_I |a_{j,i}\rangle_{A_j}$. Then, $|a_{j,i}\rangle_{A_j} = |1\rangle$ if and only if $T[i..i+j] = P[0..j]$, where $0 \leq j \leq m-1$ and $0 \leq i \leq n-1$.*

Proof. At iteration 0, after applying gates CX and X, C_P stores $|\neg(T[i] \oplus P[0])\rangle_{C_P}$ and A_{-1} stores $|1\rangle_{A_{-1}}$, thus the Toffoli gate simply copies value $\neg(T[i] \oplus P[0])$ to A_0. Because we are working with a binary alphabet, $\neg(T[i] \oplus P[0])$ equals $T[i] = P[0]$, and thus we obtain superposition $1/\sqrt{n} \sum_{i=0}^{n-1} |i\rangle_I |T[i] = P[0]\rangle_{A_0}$.

$$
\begin{array}{ccccc}
I & C_T & C_P & A_0 & A_1 \\
\frac{1}{2\sqrt{2}}\,|1\rangle\,|B\rangle\,|A\rangle\,|0\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|2\rangle\,|A\rangle\,|A\rangle\,|0\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|3\rangle\,|A\rangle\,|A\rangle\,|1\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|4\rangle\,|A\rangle\,|A\rangle\,|1\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|5\rangle\,|B\rangle\,|A\rangle\,|1\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|6\rangle\,|B\rangle\,|A\rangle\,|0\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|7\rangle\,|A\rangle\,|A\rangle\,|0\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|0\rangle\,|B\rangle\,|A\rangle\,|1\rangle\,|0\rangle
\end{array}
\;\longrightarrow\;
\begin{array}{ccccc}
I & C_T & C_P & A_0 & A_1 \\
\frac{1}{2\sqrt{2}}\,|1\rangle\,|B\rangle\,|0\rangle\,|0\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|2\rangle\,|A\rangle\,|1\rangle\,|0\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|3\rangle\,|A\rangle\,|1\rangle\,|1\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|4\rangle\,|A\rangle\,|1\rangle\,|1\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|5\rangle\,|B\rangle\,|0\rangle\,|1\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|6\rangle\,|B\rangle\,|0\rangle\,|0\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|7\rangle\,|A\rangle\,|1\rangle\,|0\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|0\rangle\,|B\rangle\,|0\rangle\,|1\rangle\,|0\rangle
\end{array}
\;\longrightarrow\;
\begin{array}{ccccc}
I & C_T & C_P & A_0 & A_1 \\
\frac{1}{2\sqrt{2}}\,|1\rangle\,|B\rangle\,|0\rangle\,|0\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|2\rangle\,|A\rangle\,|1\rangle\,|0\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|3\rangle\,|A\rangle\,|1\rangle\,|1\rangle\,|1\rangle+ \\
\frac{1}{2\sqrt{2}}\,|4\rangle\,|A\rangle\,|1\rangle\,|1\rangle\,|1\rangle+ \\
\frac{1}{2\sqrt{2}}\,|5\rangle\,|B\rangle\,|0\rangle\,|1\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|6\rangle\,|B\rangle\,|0\rangle\,|0\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|7\rangle\,|A\rangle\,|1\rangle\,|0\rangle\,|0\rangle+ \\
\frac{1}{2\sqrt{2}}\,|0\rangle\,|B\rangle\,|0\rangle\,|1\rangle\,|0\rangle
\end{array}
$$

Figure 1 An example of the evolution of the superposition after one iteration of Algorithm 1. The first arrow represents the application of a CX gate with control T_1 and target P_1, and the application of a X gate on P_1. The second arrow represents the application of a Toffoli gate with controls P_1 and A_0, and target A_1. Intuitively, in the first step we are checking that $T[i+j] = P[j]$; in the second step we combine the result of this check with the contribution of the previous iteration(s). Characters A and B are to be considered binary values.

At iteration j, we assume by induction that register A_{j-1} stores $|a_{j-1,i}\rangle_{A_{j-1}} = |1\rangle$ if and only if $T[i \ldots i+j-1] = P[0 \ldots j-1]$. Gates CX and X compute $\neg(T[i+j] \oplus P[j])$ storing it in C_P. We then apply the Toffoli gate with controls C_P and A_{j-1}, and target A_j, obtaining superposition $1/\sqrt{n} \sum_{i=0}^{n-1} |i\rangle_I |a_{j-1,i} \wedge T[i+j] = P[j]\rangle_{A_j}$. Thus, $|a_{j,i}\rangle_{A_j} = |a_{j-1,i} \wedge T[i+j] = P[j]\rangle_{A_j}$ is $|1\rangle$ if and only if $T[i..i+j] = P[0..j]$. ◄

As mentioned above, we have to be careful in running Grover's search algorithm at the end of Algorithm 1. We defer these details to the full proof of Theorem 5 given in Appendix C. For now, we assume that we are able to retrieve with arbitrarily high probability $1 - (7/8)^c$ a marked substate representing a match. Combining this with Lemma 1, we obtain the claimed result.

▶ **Theorem 2.** *Given a text string T, pattern string P and integer $c > 0$, Algorithm 1 finds a match for P in T in time $O(c(|P|\sqrt{|T|}))$. If there is no match, the algorithm returns a negative answer with probability $p = 1$. If there is at least one match, the algorithm returns the index of the last position of a match with probability $p > 1 - (7/8)^c$.*

Proof. For the correctness, consider Lemma 1 where $j = m = |P|$, which is the number of times we run the *for* loop. In this case, $|a_{|P|,i}\rangle_{A_{|P|}} = |1\rangle$ if and only if $T[i..i+|P|-1] = P[0..|P|-1]$. Thus, measuring these substates yields a correct solutions. The details of how to perform such a measurement respecting the time complexity and probability of success are deferred to the full proof of Theorem 5 in Appendix C. ◄

5 String Matching in Labeled Graphs

5.1 Quantum Brute-force Algorithm for SMLG

In SMLG we are given pattern string P with characters in alphabet Σ and a node-labeled graph $G = (V, E)$, with labelling function $\ell : V \to \Sigma$. We are asked to find a path (or, actually, a walk) $\pi = v_1, v_2, \ldots, v_{|P|}$ in G such that $\ell(v_1) \circ \ell(v_2) \circ \ldots \circ \ell(v_{|P|}) = P$, where $\circ$ denotes string concatenation.

One could try to obtain a quantum algorithm for SMLG by generalizing the idea we presented for plain text. The idea would be to list all possible paths of length $|P|$ in the graph, and then mark those ones that are actual matches for P. Unfortunately, the superposition

■ **Algorithm 1** An algorithm for solving exact string matching in plain text that, using QRAM, achieves $O(|P|\sqrt{|T|})$ time complexity. The details of how to handle Grover's search at the end are given in Theorem 5, whose full proof is deferred to Appendix C.

Input: Text T stored as $|t_0\rangle_{T_0} |t_1\rangle_{T_1} \cdots |t_{n-1}\rangle_{T_{n-1}}$, pattern string P stored as $|p_0\rangle_{P_0} |p_1\rangle_{P_1} \cdots |p_{m-1}\rangle_{P_{m-1}}$, integer c

Output: A position of T where a match for P ends, if any

1 for *c times* **do**

2 Initialize quantum registers $I, J, A_0, A_1 \cdots A_{m-1}$ as $|0\rangle_I |0\rangle_J |0\rangle_{A_0} |0\rangle_{A_1} |0\rangle_{A_{m-1}}$;

3 Initialize quantum register A_{-1} and Q as $|1\rangle_{A_{-1}}$ and $|1\rangle_Q$;

   // Apply $H^{\otimes \log n}$ to register I

4 $|0\rangle_I \to \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle_I$;

5 **for** *m times* **do**

      // Read $T[i]$ in C_T and $P[j]$ in C_P using registers I and J as
         indexes

6 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle_I |j\rangle_J |0\rangle_{C_T} |0\rangle_{C_P} \to \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle_I |j\rangle_J |t_i\rangle_{C_T} |p_j\rangle_{C_P}$;

      // Apply CX with control C_T and target C_P

7 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i\rangle_{C_T} |p_j\rangle_{C_P} \to \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i\rangle_{C_T} |t_i \oplus p_j\rangle_{C_P}$;

      // Apply X to C_P

8 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i \oplus p_j\rangle_{C_P} \to \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |\neg(t_i \oplus p_j)\rangle_{C_P} = \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i = p_j\rangle_{C_P}$;

      // Apply Toffoli with controls C_P and A_{j-1}, and target A_j

9 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i = p_j\rangle_{C_P} |a_{j-1}\rangle_{A_{j-1}} |0\rangle_{A_j} \to$
      $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i = p_j\rangle_{C_P} |a_{j-1}\rangle_{A_{j-1}} |(t_i = p_j) \wedge a_{j-1}\rangle_{A_j}$;

      // Reset C_T and C_P to $|0\rangle$ via uncomputation

10 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |\neg(t_i \oplus p_j)\rangle_{C_P} \to \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i \oplus p_j\rangle_{C_P}$;

11 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i\rangle_{C_T} |t_i \oplus p_j\rangle_{C_P} \to \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |t_i\rangle_{C_T} |p_j\rangle_{C_P}$;

12 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle_I |j\rangle_J |t_i\rangle_{C_T} |p_j\rangle_{C_P} \to$
      $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle_I |j\rangle_J |t_i \oplus t_i\rangle_{C_T} |p_j \oplus p_j\rangle_{C_P} = \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |i\rangle_I |j\rangle_J |0\rangle_{C_T} |0\rangle_{C_P}$;

      // Increment indexes I and J

13 $\frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |1\rangle_Q |i\rangle_I |j\rangle_J \to \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} |1\rangle_Q |i \oplus 1\rangle_I |j + 1\rangle_J$;

14 Apply gate Z to qubit R_{n-1}, so that the sign of the amplitude is flipped if $|r_{n-1,j}\rangle_{R_{n-1}} = |1\rangle$;

15 Choose $K \in [0, |P|]$ uniformly at random;

16 Run Grover's iterate operator the optimal number of times assuming to have K solutions, with the oracle function being lines 5–14 of this algorithm;

17 Measure R_{n-1} into classical register R_{cl};

18 **if** $R_{cl} = 1$ **then**

19   Measure I into classical register I_{cl} and **return** I_{cl}

20 return *no*

---

would be as large as there are paths of length $|P|$, and thus the overall time complexity would be $O(|P|\sqrt{|V|^{|P|}})$. Moreover, an adjacency matrix would be needed to check the existence of edges between nodes in constant time, yielding a space complexity of $O(|V|^2)$ qubits. We conclude that more involved techniques are needed.

5.2   The Classical Shift-And Algorithm

We first introduce the classical *shift-and* algorithm [3] 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 logaritmic number of qubits. This approach allows us to achieve better performances than the brute force algorithm.

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

1. $B \leftarrow B + 1$;
2. $B \leftarrow B \wedge M[*][T[i]]$;
3. if $B[m-1] = 1$, return *yes*;
4. $B \leftarrow B << 1$.

Operation 1 sets the least significant bit of B to 1, which is needed to test $P[0]$ against $T[i]$. Operation 2 computes a bit-wise *and* between B and the column of M corresponding to character $T[i]$. Remember that $M[j][T[i]] = 1$ means $P[j] = T[i]$, thus this operation leaves each bit $B[j]$ set to 1 if and only if it was already set to 1 before this step and the the j-th character of the pattern matches the current character of the text. At this point, if bit $B[m-1]$ is set to 1 we have found a match for P, and Operation 3 will return *yes*. For the other positions, if bit $B[j]$ is set to 1, then we know that prefix $P[0..j]$ matches $T[i-j+1..i]$, and Operation 4 shifts the bits in B by one position, so that in the next iteration we will check whether $P[j+1]$ matches $T[i+1]$.

In labeled DAG $G = (V, E)$, each node $v_i \in V$ has a single-character label $\ell(v_i)$. We generalize the shift-and algorithm to labeled DAGs by computing a bit-vector B_i for each node $v \in V$, initializing them to zero. Consider a BFS visit of DAG G. When visiting node v_i, each bit-vector B_k of its in-neighbour $v_k \in in(v_i)$ represents a set of prefixes of P matching a path in the graph ending at v_k. Thus, we merge all of this information together by taking the bit-wise *or* of all of the in-neighbours of v_i, that is we replace Operation 1 with $B_i \leftarrow 1 + \bigvee_{v_k \in in(v_i)} B_k$. Operations 2, 3 and 4 are performed as before. An example of the state of the data structures after the execution of the algorithm is shown in Figure 2, and the body of the iteration now is:

1. $B_i \leftarrow 1 + \bigvee_{v_k \in in(v)} B_k$;
2. $B_i \leftarrow B_i \wedge M[*][T[i]]$;
3. if $B_i[m-1] = 1$, return *yes*;
4. $B_i \leftarrow B_i << 1$.
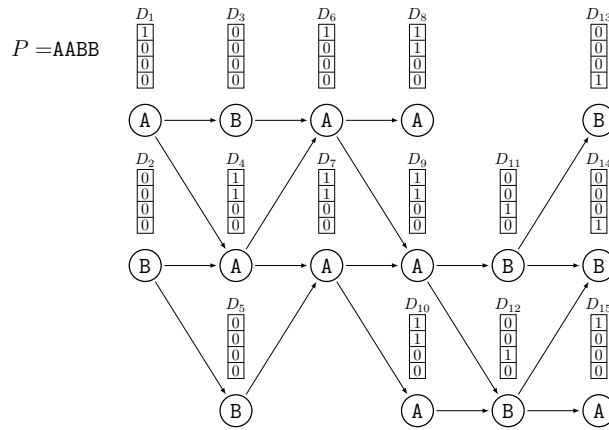
Figure 2 The adaptation of the classical algorithm for matching pattern P in *level* DAG G. Each bit-vector D_v represent the result after the merging of the bit-vectors of the in-neighbours of v and before the shifting.

5.3 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 DAG such that, for every two nodes v and w, every path from v to w has the same length, as for the DAG in Figure 2. We also note that *degenerate strings* [1] can be represented as level DAGs. We use a function representing in-neighbours:

$$in_i(d) = \text{index of the } d\text{-th in-neighbour of } v_i$$

Our approach aims to represent each bit vector B_i with a single qubit V_i set up in a proper superposition, and translate the bit-wise operations to parallel operations across such superposition. In the algorithm, we use the following qubits and quantum registers. Quantum registers I and J store the index of a node and the position in the pattern, respectively. Qubit V_i represents, in superposition, the bit-vector of the node v_i, and qubit $E_{i,d}$ stores the contribution of edge $(v_{in_i(d)}, v_i) \in E$ in the update of qubit V_i, for, $0 \le i \le n-1$, $0 \le d \le D_i - 1$ and $D_i = indeg(v_i)$. Quantum register C stores label $\ell(v_i)$ of the node in the current iteration, and is used to fetch the content of the corresponding matrix column, which we will store in qubit M. Occurrences of the pattern encountered during the execution of the algorithm are stored in qubit R_i. Qubits V_i' and R_i' are auxiliary qubits used to store intermidiate results, and we also use auxiliary qubits A and B and auxiliary quantum register Q to implement necessary operations. Moreover, we assume to have access to QRAM.

5.3.1 The algorithm

Assume all the quantum registers and qubits to be initialized to $|0\rangle$, except Q initialized to $|1\rangle$. The algorithm starts by setting quantum register J in a balanced superposition, by applying the Hadamard gate on each one of its qubits. Then, we initialize qubits A so that $|a_j\rangle_A = |1\rangle$ for $j = 0$, and $|a_j\rangle_A = |0\rangle$ otherwise. We do the same with qubit B, with the difference that $|b_j\rangle_A = |1\rangle$ for $j = m-1$, and $|b_j\rangle_A = |0\rangle$ otherwise. We can do these operations with two applications of a generalized Toffoli gate, using register J as control and qubits A and then B as targets. In the case of qubit A, we first apply an X gate to every qubit of register J, we then apply the Toffoli gate, and finally we undo the applications of

the X gate. The generalised Toffoli has a cost proportional to the number of qubits in J, that is logarithmic in the size of the input, and because this is an operation between a single quantum register and a qubit, we can assume it to be constant in the Word-QRAM model. We then initialize the qubits representing the bit-vectors of the nodes at level 0. This is done with the same operations described below for the main loop, the only difference being that these nodes do not have in-neighbours and thus we can simplify some operations. Specifically, we load each entry of the character matrix in superposition and we use it and qubit A as controls of a Toffoli gate which thus flips to $|1\rangle$ sub-state $|v_{i,j}\rangle_{V_i}$ if $P[0]$ matches $\ell(v_i)$.

The rest of the algorithm maintains almost the same overall structure, with the exception of one necessary adaptation. In a DAG of L levels where L_l is the set of nodes at level l, for $0 \leq l \leq L - 1$, we iterate over them one at the time, and for each level we process its nodes one after the other. As we will better explain later, we wait before applying the quantum equivalent of the shift operation once we scanned the whole level, not after processing every node. The overall idea is to translate the classical bit-parallel operations into analogous quantum operations that work across the superposition. This translation of bit-parallelism to superposition parallelism is the core of our technique, and we now describe how to apply it to each operation. The pseudocode of the entire procedure is given in Algorithm 1, where all the arithmetic operations are to be considere modulo $2^{|P|}$. We only omit the pseudocode for procedures `SourceNodesInit()`, `IncreaseI()` and `IncreaseJ()`, which is to be found in Appendix B. We also assume $|P|$ to be a power of two. If this is not the case, we generate a superposition as large as the first power of two greater than $|P|$, then standard techniques can be used to handle the additional substates, as explained in Appendix A.

Operation 1 (line 10) can be broken down into two simpler operations: computing the bit-wise *or* and adding 1. In our translation to quantum computing, each sub-state of superposition $\sum_{j=0}^{m-1} |j\rangle_J |v_{i,j}\rangle_{V_i}$ represents an entry of the classical bit-vector used in the Shift-And algorithm. Thus, what was a bit-wise *or* is now easily translated into the application of few quantum gates. Notice that, to compute the logical *or* between two generic qubits P and Q and store the result in qubit R, we can follow De Morgan's rules and apply an X gate to both P and Q, apply a Toffoli gate with controls P and Q and target R, apply an X gate to R, and finally apply an X gate to P and Q again to restore their initial values. In our case, at iteration i, we use qubit $E_{i,d}$ to store the *or* computed among the first $d+1$ in-neighbours $v_{in_i(0)}, \ldots, v_{in_i(d)}$ of node v_i, and we compute it in the following way. Let

$$\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |v_{in_i(0),j}\rangle_{V_{in_i(0)}} |v_{in_i(1),j}\rangle_{V_{in_i(1)}} \cdots |v_{in_i(d-1),j}\rangle_{V_{in_i(d-1)}} |e_{i,d-1,j}\rangle_{E_{i,d-1}}$$

be such that

$$e_{i,d-1,j} = v_{in_i(0),j} \vee v_{in_i(1),j} \cdots \vee v_{in_i(d-1),j}.$$

We compute the value of $E_{i,d}$ from $E_{i,d-1}$ and $V_{in_i(d)}$ as

$$\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |v_{in_i(d),j}\rangle_{V_{in_i(d)}} |e_{i,d-1,j}\rangle_{E_{i,d-1}} |0\rangle_{E_{i,d}} \rightarrow$$

$$\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |v_{in_i(d),j}\rangle_{V_{in_i(d)}} |e_{i,d-1,j}\rangle_{E_{i,d-1}} |v_{in_i(d),j} \vee e_{i,d-1,j}\rangle_{E_{i,d}}.$$

Once we processed the last in-neighbour, E_{i,D_i-1} stores the *or* computed among all in-neighbours, where D_i is the number of in-neighbours of node v_i.

We implement the classic operation of adding 1 by computing an *or* with qubit A and storing the result in V_i'. Since $|a_j\rangle_A = |\delta_{0,j}\rangle$, we obtain $|0\rangle_{V_i'} \to |v_{i,j}'\rangle_{V_i'}$ where $|v_{i,j}'\rangle_{V_i'} = |1\rangle$ for $j = 0$, while $|v_{i,j}'\rangle_{V_i'} = |e_{i,D_i-1,j}\rangle$ for $1 \le j \le m-1$.

Operation 2 (line 11) is implemented as a Toffoli-gate application with qubits M and V_i' as control and V_i as target.

$$\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |m_{j,\ell(v_i)}\rangle_M |v_{i,j}'\rangle_{V_i'} |0\rangle_{V_i} \to \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |m_{j,\ell(v_i)}\rangle_M |v_{i,j}'\rangle_{V_i'} |m_{j,\ell(v_i)} \wedge v_{i,j}'\rangle_{V_i}$$

Operation 3 (line 12) is replaced by storing in register R_i the presence of a match ending at node v_i. This requires an intermediate step in which we use qubit B to filter the content of V_i. In fact, qubit V_i now is in state $|v_{i,j}\rangle_{V_i} = |1\rangle$ for those values of j such that $P[0..j]$ has a match ending at v_i in the graph, and $|v_{i,j}\rangle_{V_i} = |0\rangle$ otherwise. Since we only care about potential full matches represented by $|v_{i,m-1}\rangle_{V_i}$, we use B, which is in state $|\delta_{m-1,j}\rangle_B$, as control qubit of a Toffoli gate, the other control qubit being V_i and the target qubit being R_i'.

$$\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j}\rangle_{V_i} |\delta_{m-1,j}\rangle_B |0\rangle_{R_i'} \to \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j}\rangle_{V_i} |\delta_{m-1,j}\rangle_B |v_{i,j} \wedge \delta_{m-1,j}\rangle_{R_i'}$$

Then, using the same technique as in Operation 1, we compute an *or* between R_i' and R_{i-1}, storing the result in R_i.

$$\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j} \wedge \delta_{m-1,j}\rangle_{R_i'} |r_{i-1,j}\rangle_{R_{i-1}} |0\rangle_{R_i} \to$$

$$\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j} \wedge \delta_{m-1,j}\rangle_{R_i'} |r_{i-1,j}\rangle_{R_{i-1}} |(v_{i,j} \wedge \delta_{m-1,j}) \vee r_{i-1,j}\rangle_{R_i}$$

After this operation, $|r_{i,m-1}\rangle_{R_i}$ is turned to $|1\rangle$ if there is a full match of P ending at v_i, otherwise $|r_{i,m-1}\rangle_{R_i}$ is left unaltered.

Operation 4 (line 14) consists in shifting all bits of the classical bit-vector by one position. In the quantum setting, we can perform this operation by adding 1 to index register J and then reorganising the sum: $|1\rangle_{C_1} |j\rangle_J \to |1\rangle_{C_1} |j+1\rangle_J$. Notice that this changes value $|j\rangle_J$ in every term of the superposition to $|j+1\rangle_j$. This can be interpreted as "shifting" value k_j of generic register K from $|j\rangle_J |k_j\rangle_K$ to $|j+1\rangle_J |k_j\rangle_K$. Because this operation acts on every quantum register and qubit in this way, we have to reset qubits A and B to $|0\rangle$ before performing this operation and reinitialize their values afterwards, so that we prevent their values to be shifted. For the same reason, we also have to wait until having processed the whole level, otherwise we would shift the values of all the nodes at the previous level and compromise the computation.

As last step of the algorithm, we run Grover's search that uses as oracle function the whole procedure described up to this point, and then applies a Z gate on qubit R. Thus, the marked sub-states are those such that $|r_{i,j}\rangle_{R_i} = |1\rangle$, which get mapped to $-|r_{i,j}\rangle_{R_i}$. Sub-states such that $|r_{i,j}\rangle_{R_i} = |0\rangle$ remain unaltered. As for the case of string matching in plain text, we rerun the whole algorithm a constant number of times to boost the probability of success, as explained in Theorem 5 and Appendix C. Algorithm 1 shows the entire procedure.

To prove the correctness of Algorithm 1, we formalise the key properties in the following lemmas. We start by ensuring that the shift operation provides the desired result. Let l and y be the total number of times that we started the execution of the middle *for*-loop

■ **Algorithm 2** Algorithm for testing whether pattern string P has a match in level DAG G, running in $O(|E|\sqrt{|P|})$.

---

1 time.

Input: Graph G, pattern P, and constant c.

Output: Returns *yes* if P occurs in G, otherwise *no*.

2 **for** *c times* **do**

3   Initialize quantum register Q to $|1\rangle$;

4   Initialize quantum registers $I, J, A, B, C, M, V_i, V_i', R_i, R_i', E_{i,d}$ to $|0\rangle$ where $i \in [0, n-1]$ and $d \in [0, D_i - 1]$;

     `// Apply Hadamard to J`

5 $|0\rangle_J \rightarrow \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J$;

6   `SourceNodesInit`$(I, J, A, C, V_0, \ldots, V_{n-1}, V_0', \ldots, V_{n-1}')$;

7   `IncreaseJ`(J, A, B);

     `// L is the number of levels`

8   **for** $l \in [1, L-1]$ **do**   `// scan every level`

9   **for** $|L_l|$ *times* **do**   `// scan every node in the level`

10   `OperationOne`$(l, I, C, M, E_{i,0}, \ldots, E_{i,D_i-1}, V_i')$;

11   `OperationTwo`(M, V_i', V_i);

             `// Invariant 1 holds here`

12   `OperationThree`(B, V_i, R_i', R_i);

13   `IncreaseI`(I, M, C);

14   `OperationFour`(J, A, B);

         `// Invariant 2 holds here`

15   Apply gate Z to qubit R_{n-1}, so that the sign of the amplitude is flipped if $|r_{n-1,j}\rangle_{R_{n-1}} = |1\rangle$;

16   Choose $K \in [0, |P|]$ uniformly at random;

17   Run Grover's iterate operator the optimal number of times assuming to have K solutions, with the oracle function being lines 6–15 of this algorithm;

18   Measure R_{n-1} into classical register R_{cl};

19   **if** $R_{cl} = 1$ **then**

20   **return** *yes*

21 **return** *no*

---

1 Function OperationOne$(l, I, C, M, E_{i,0}, \ldots, E_{i,D_i-1}, V_i')$:

2   **for** $|L_l|$ *times* **do**   // scan every node in the level
      // scan every node in $in(v_i)$

3 $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |i\rangle_I |0\rangle_C \to \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |i\rangle_I |\ell(v_i)\rangle_C$;

4 $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |0\rangle_M \to \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |m_{\ell(v_i),j}\rangle_M$;

5 $k \leftarrow in_i(0)$;   // Classical operation

6 $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{k,j}\rangle_{V_k} |0\rangle_{E_{i,0}} \to \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{k,j}\rangle_{V_k} |v_{k,j}\rangle_{E_{i,0}}$;

7   **for** $d \in [1, D_i - 1]$ **do**   // scan every node in $in(v_i)$

8 $k \leftarrow in_i(d)$;   // Classical operation
        // Add the contribution of the current in-neighbour

9 $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{k,j}\rangle_{V_k} |e_{i,d-1,j}\rangle_{E_{i,d-1}} |0\rangle_{E_{i,d}} \to$

          $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{k,j}\rangle_{V_k} |e_{i,d-1,j}\rangle_{E_{i,d-1}} |e_{i,d-1,j} \vee v_{k,j}\rangle_{E_{i,d}}$;

       // Turn to $|1\rangle$ the substate corresponding to $j = 0$

10 $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |\delta_{0,j}\rangle_A |e_{i,D_i-1,j} \vee v_{k,j}\rangle_{E_{i,D_i-1}} |0\rangle_{V_i'} \to$

        $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |\delta_{0,j}\rangle_A |e_{i,D_i-1,j} \vee v_{k,j}\rangle_{E_{i,D_i-1}} |e_{i,D_i-1,j} \vee v_{k,j} \vee \delta_{0,j}\rangle_{V_i'}$;

11 Function OperationTwo(M, V_i', V_i):
   // Compute the *and* with the column of the matrix

12 $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |m_{\ell(v_i),j}\rangle_M |v_{i,j}'\rangle_{V_i'} |0\rangle_{V_i} \to \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |m_{\ell(v_i),j}\rangle_M |v_{i,j}'\rangle_{V_i'} |m_{\ell(v_i),j} \wedge v_{i,j}'\rangle_{V_i}$;

13 Function OperationThree(B, V_i, R_i', R_i):
   // Set $|r_{i,m-1}\rangle_{R_i} = |1\rangle$ if there is a match ending at v_i
   // Apply Toffoli on V_i, B and R_i'

14 $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j}\rangle_{V_i} |\delta_{m-1,j}\rangle_B |0\rangle_{R_i'} \to \frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j}\rangle_{V_i} |\delta_{m-1,j}\rangle_B |v_{i,j} \wedge \delta_{m-1,j}\rangle_{R_i'}$;
   // Apply logic *or* on R_i', R_{i-1} and R_i

15 $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j} \wedge \delta_{m-1,j}\rangle_{R_i'} |r_{i-1,j}\rangle_{R_{i-1}} |0\rangle_{R_i} \to$

      $\frac{1}{\sqrt{m}} \sum_{j=0}^{m-1} |v_{i,j} \wedge \delta_{m-1,j}\rangle_{R_i'} |r_{i-1,j}\rangle_{R_{i-1}} |(v_{i,j} \wedge \delta_{m-1,j}) \vee r_{i-1,j}\rangle_{R_i}$;

16 Function OperationFour(J, A, B):

17   IncreaseJ(J, A, B);

(lines 9–13) and of the outer *for*-loop (lines 8–14), respectively. That is, $y = x + \sum_{\lambda=1}^{l-1} |L_\lambda|$ for $l \geq 2$, where $x \in [0, |L_l|]$ is the number of times that we started the execution of the middle *for*-loop during the l-th iteration of the outer *for*-loop. Notice that $y = 0$ when $l = 0$, and $y = x$ when $l = 1$.

▶ **Lemma 3** (Invariant 1). *During the l-th execution of the outer* for*-loop (lines 8–14) and the y-th execution of the middle* for*-loop (lines 9–13) of Algorithm 1, but before the y-th execution of* `OperationThree()` *(line 14),* Invariant 1 *holds: for every qubit V_i such that $i \in L_l$ and $i \leq t$, we have substate $|v_{i,j}\rangle_{V_i} = |1\rangle$ if and only if there exists a path in G ending at v_i and matching $P[0, j]$, where $t = |L_0| + x - 1$ is the index of the last node v_t Algorithm 1 visited so far.*

Proof. We proceed by strong induction on y, defined as above.

Base case, $y = 0$.   In this case, we executed the initialization but we have not run yet neither the outer nor the middle *for*-loop. Thus, $l = 0$, $t = |L_0| - 1$, and qubits V_i such that $i \in L_0$ and $i \leq t$ are those with in-degree zero, which are initialized by function `SourceNodesInit()`. For each such i, given that J is in state $\sum_{j=0}^{m-1} |j\rangle_J$, function `SourceNodesInit()` first loads character $\ell(v_i)$ in register C and matrix entry $m_{\ell(v_i),j}$ in register M, in superposition. Then, with regard to t, it performs transformation

$$\sum_{j=0}^{m-1} |m_{\ell(v_i),j}\rangle_M |\delta_{0,j}\rangle_A |0\rangle_{V_i} \to \sum_{j=0}^{m-1} |m_{\ell(v_i),j}\rangle_M |\delta_{0,j}\rangle_A |m_{\ell(v_i),j} \wedge \delta_{0,j}\rangle_{V_i},$$

where, by definition, $v_{t,j} = m_{\ell(v_i),j} \wedge \delta_{0,j}$. Thus, $|v_{t,j}\rangle = |0\rangle$ for every $j \neq 0$ because of $\delta_{0,j}$, and $|v_{t,j}\rangle_{V_i} = |m_{\ell(v_i),j}\rangle_{V_i}$ for $j = 0$, which in turn means that $|v_{t,0}\rangle_{V_i} = |1\rangle$ if and only if $P[0, 0] = \ell(v_i)$.

Inductive case, $y \geq 1$.   We further divide our analysis in two sub-cases.

First sub-case, $x = |L_l|$.   In this case, y is the last iteration of the inner *for*-loop during the l-th iteration of the outer *for*-loop. We assume the inductive hypothesis to hold after the execution of `OperationTwo()`. We execute `OperationThree()` and `IncreaseI()`, which do not change the state of any V_z, for any $z \in [0, |V| - 1]$. Now, we have to perform `OperationFour()` (line 14) before starting iteration $y + 1$ of the middle *for*-loop, which will start iteration $l + 1$ of the outer *for*-loop. Assuming the inductive hypothesis, the application of `OperationFour()` makes every V_z with $z \in L_l$ such that $|j'\rangle_J |v_{z,j}\rangle_{V_z} = |j'\rangle |1\rangle$, where $j' = j + 1$, if and only if there is a match for $P[0, j]$ in G ending at v_z, otherwise $|j'\rangle_J |v_{z,j}\rangle_{V_z} = |j'\rangle |0\rangle$. Then, we start iteration $y + 1$ ($l + 1$). Notice that we update V_i if and only if $i \in L_{l+1}$ and, in any previous iteration of the middle *for*-loop, this could have never been the case, thus every $|v_{i,j}\rangle_{V_i}$, $i \in L_{l+1}$, is currently set to $|0\rangle$. The same holds for every V_i'. The *for*-loop inside `OperationOne()` computes a logic *or* between all the qubits representing all the in-neighbours of v_i. Indeed, before running this *for*-loop, we have $|j'\rangle_J |v_{in_i(0),j}\rangle_{E_{i,0}}$. After one iteration, we have $|j'\rangle |v_{in_i(0),j} \vee v_{in_i(1),j}\rangle_{E_{i,1}}$. After two iteration, we have $|j'\rangle |v_{in_i(0),j} \vee v_{in_i(1),j} \vee v_{in_i(2),j}\rangle_{E_{i,2}}$. After $D_i - 1$ iterations, we have $|j'\rangle |e_{i,D_i-1,j'}\rangle_{E_{i,D_i-1}}$, where

$$e_{i,D_i-1,j'} = \bigvee_{d=0}^{D_i-1} v_{in_i(d),j}.$$

We store an intermediate result in V_i', $|v_{i,j'}'\rangle_{V_i'}$, where $v_{i,j'}' = e_{i,D_i-1,j'}$ except for $j' = 0$, because we make sure that $|v_{i,0}'\rangle_{V_i'} = |1\rangle$ thanks to the *or* operation with qubit A, which stores $|\delta_{0,j'}\rangle_A$. Now we compute the logical *and* with the entry of the matrix, as in the base case, obtaining $|v_{i,j'}\rangle_{V_i}$, where

$$v_{i,j'} = m_{\ell(v_i),j'} \wedge (\delta_{0,j'} \vee \bigvee_{d=0}^{D_i-1} v_{in_i(d),j}).$$

Applying the inductive hypothesis, this translates to

$$v_{i,j+1} = (P[j+1] = \ell(v_i)) \wedge \left((j+1=0) \vee \bigvee_{d=0}^{D_i-1} P[0..j] \text{ has a match ending at } v_{in_i(d)} \right)$$
$$= P[0..j+1] \text{ has a match ending at } v_i$$

Thus, the statement of the lemma holds for $y + 1$.

Second sub-case, $x < |L_l|$. The reasoning is analogous to the previous case, the only difference being that j does not increase and thus we have to look back by $x + 1$ iterations, when j was increased the last time. This requires to assume that the inductive hypothesis was holding for iteration $y - x$, that is correct because, by strong induction, we assume the inductive hypothesis to hold for every $y' \leq y$ while proving the statement for $y + 1$. ◄

▶ **Lemma 4** (Invariant 2). *After line 14 of Algorithm 1, Invariant 2 holds: if there exists at least one match for P in G ending at some v_i such that $i \leq t$, then there exists at least one j, $0 \leq j \leq m - 1$, such that $|r_{t,j}\rangle_{R_i} = |1\rangle$, where v_t is the last node we visited in Algorithm 1 before line 14.*

Proof. We proceed by induction on the number l of times that we run the *for*-loop at lines 8–14.

Base case, $l = 0$. In this case, nodes v_t such that $t \in L_{l'}$, $l' \leq 0$ are those with in-degree zero, while the *for*-loop at lines 8–14 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, $|r_{i,j}\rangle_{R_i} = |0\rangle$ for every $0 \leq j \leq m - 1$.

Inductive case, $1 \leq l \leq L - 1$. By inductive hypothesis, we assume the statement of the lemma to be true right after running iteration l of the *for*-loop at lines 8–14, and thus right before executing `IncreaseJ()` at line 14. After the execution of `IncreaseJ()`, the new state is $\sum_{j=0}^{m-1} |j'\rangle_J |r_{i,j}\rangle_{R_i}$, where $j' = j + 1$ and v_i is the last node visited so far. Then, we start iteration $l + 1$, processing $i' \in L_{l+1}$, $i' = t + 1$. We execute `OperationOne()` `OperationTwo()`, which do not affect register $R_{i'}$. Then we run the operations at lines 12–12, obtaining $|r_{i',j'}\rangle_{R_i}$ where $r_{i',j'} = (v_{i',j'} \wedge \delta_{m-1,j'}) \vee r_{t,j}$. Let us consider the first time we run the middle *for*-loop during iteration $l + 1$ of the outer *for*-loop. If P has a match ending at some v_z, $z < i'$, the inductive hypothesis guarantees $r_{t,j} = 1$ for some j. Otherwise, if P does not have any such match, then $r_{t,j} = 0$ for all j. In this second case, if P has a match ending at $v_{i'}$, we know by Lemma 3 that $v_{i',m-1} = 1$. This, combined with the fact that $\delta_{m-1,m-1} = 1$, correctly implies that $r_{i',m-1} = 1$, proving the statement for this specific i' and $j' = m - 1$. If P has no match ending at $v_{i'}$, then $v_{i',m-1} = 0$, and $r_{i',j'} = 0$ for all j', which must be the case when no match has been found yet. To conclude the proof, notice that the same reasoning applies for the subsequent iterations of the middle *for*-loop by using every time the

previous instance of this reasoning in place of the inductive hypothesis. That is, we use $r_{i',j'}$ when proving the statement for $r_{i'+1,j'}$ and so on, until we prove the statement for $r_{t',j'}$, where $v_{t'}$ is the last node with index in L_{l+1}. At this point, we exit the middle *for*-loop and the statement of the lemma is proven for $l+1$.   ◄

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

▶ **Theorem 5.** *Given pattern string P of length at least 2 and level DAG G, Algorithm 1 returns the right answer for the **SMLG** problem on P and G with probability $p > 1 - (7/8)^c$, for any given integer c.*

Proof. After running the outer *for*-loop of Algorithm 1 $L-1$ times, we exit such a loop, and we know we have visited all the nodes (nodes in L_0 where visited during the initialization). If we consider Lemma 4 applied in the case of $t = n-1$, we are considering all the nodes, which means that if P has no match ending in G, then no substate of register R_{n-1} is such that $|r_{n-1,j}\rangle_{R_{n-1}} = |1\rangle$, for any j. Instead, if P has a match in G, then at least one substate of R_{n-1} is such that $|r_{n-1,j}\rangle_{R_{n-1}} = |1\rangle$, for some j. We use standard techniques that consist in rerunning the algorithm a constant number of times to boost the probability of measuring such a state, and achieve the desired one. Appendix C provides a more detailed analysis.   ◄

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

▶ **Theorem 6.** *The time complexity of Algorithm 1 is $O(|E|\sqrt{|P|})$ in the QRAM model, and the space complexity is $O(|E| + |V|)$.*

Proof. The algorithm uses $|V|$ qubits V_i, and the same amount of qubits V'_i, R_i, R'_i; qubits $E_{i,d}$ are a total of $|E|$ qubits, and the rest are a constant number of qubits and registers. Thus, the space complexity is $O(|E| + |V|)$.

With the *for*-loop in function `SourceNodesInit()`, the algorithm visits the nodes in L_0, which are at most $O(|V|)$. The iteration conditions at lines 8 and 9 make the algorithm visit every node. For each such iteration, we perform a constant number of operations except for the *for*-loop in `OperationOne()`. This *for*-loop visits all the in-neighbours of a node, each time performing a constant number of operations, and $\sum_{i=0}^{|V|-1} |in(i)| = |E|$. All of the aforementioned operations can be implemented with a constant number of quantum-gate applications, each affecting a constant number of qubits (3 at most), or by performing a load operation from the QRAM, assumed to require constant time. At the end of the algorithm, we run Grover's search procedure on a superposition of $2|P|$ states, using the entire algorithm as the oracle function.

Summing everything up, we spend $O(|V|)$ time for the initialization, $O(|V| + |E|)$ time in the *for*-loops, and $O(|E|\sqrt{|P|})$ time for Grover's search procedure. The total time complexity is thus dominated by $O(|E|\sqrt{|P|})$.   ◄

─── **References** ───

1   Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Comparing degenerate strings. *Fundam. Informaticae*, 175(1-4):41–58, 2020.

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. `doi:10.1137/15M1053128`.

3   Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992. `doi:10.1145/135239.135243`.

4   Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Trans. Algorithms*, 16(2), March 2020. `doi:10.1145/3381417`.

5   Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. Tight bounds on quantum searching. *Fortschritte der Physik: Progress of Physics*, 46(4-5):493–505, 1998.

6   Parisa Darbari, Daniel Gibney, and Sharma V. Thankachan. Quantum time complexity and algorithms for pattern matching on labeled graphs. In *String Processing and Information Retrieval - 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8-10, 2022, Proceedings*, volume 13617 of *Lecture Notes in Computer Science*, pages 303–314. Springer, 2022. `doi:10.1007/978-3-031-20643-6_22`.

7   Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. In *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.

8   Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Physical review letters*, 100(16):160501, 2008.

9   Lov K. Grover. A fast quantum mechanical algorithm for database search. In *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. `doi:10.1145/237814.237866`.

10   Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. `doi:10.1137/0206024`.

11   Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, 1999. `doi:10.1145/316542.316550`.

12   Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. `doi:10.1017/CBO9780511976667`.

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

14   Hariharan Ramesh and V Vinay. String matching in $O(\sqrt{n} + \sqrt{m})$ quantum time. *Journal of Discrete Algorithms*, 1(1):103–110, 2003. Combinatorial Algorithms. `doi:10.1016/S1570-8667(03)00010-8`.

15   Mikko Rautiainen, Veli Mäkinen, and Tobias Marschall. Bit-parallel sequence-to-graph alignment. *Bioinform.*, 35(19):3599–3607, 2019. `doi:10.1093/bioinformatics/btz162`.

16   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.

17   Jorg Van Renterghem. The implications of breaking the strong exponential time hypothesis on a quantum computer. Master's thesis, Ghent University, 2019.

A   Reductions to the power-of-two case

For string matching in plain text, if $|T|$ is not a power of two, in addition to quantum register I for indexing, we use also quantum register I', of the same size. Let x be the only integer x such that $|T| < 2^x < 2|T|$. Generate superposition $\sum_{i'=0}^{2^x-1} |i'\rangle_{I'} |0\rangle_I$ and compute $\sum_{i'=0}^{2^x-1} |i'\rangle_{I'} |i' \bmod |T|\rangle_I$. Now run the algorithm using I as normal. This creates some redundant substates, but does not affect the correctness of the algorithm.

For SMLG in level DAGs, if $|P|$ is not a power of two, we generate a superposition of size 2^x, where x is the only integer such that $|P| < 2^x < 2|P|$. Then, it suffices to assume that every entry that we read from the QRAM to the additional substates between $|P|$ and 2^x

is always initialized to $|1\rangle$, because this is the neutral value in a logical *and*, an thus in the application of the Toffoli gate. Therefore, in these substates, a qubit R_i can and will be set to value $|1\rangle_{R_i}$ if and only if a previous "shift" carried $|1\rangle_{R_{i-1}}$.

Alternatively, if $|P|$ is not a power of two, we can classically reduce the problem to this case. We add new symbol \$ to the alphabet. Then, we pad P with as many \$ at the end as needed to reach the next power of two. For each level in the DAG, we add a new node with label \$, and we place an edge for every node in that level to the new node. We connect all this new nodes in a chain, and we also add a chain of $|P|$ such nodes after the last level (they create new levels consisting only of one node). The pattern now can overflow in these nodes after finding a proper match in the DAG. Finally, we apply the same binary encoding as in the plain text case, now replacing every node with a chain of two nodes, sending all the incoming edges to the first node and making all the outgoing edges leave from the second node. Overall, we add one new node per level, and one new edge per node, plus $|P|$ additional nodes and edges after the last level. This takes time $O(|E| + |P|)$.

B   Additional pseudo-code

```
1  Function SourceNodesInit(I, J, A, C, V_0, ..., V_{n-1}, V'_0, ..., V'_{n-1}):
```

// Initialize $|a_j\rangle_A$ so that $a_j = 1$ if $j = 0$, $a_j = 0$ otherwise

2    $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |0\rangle_A \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |\delta_{0,j}\rangle_A$;

// Initialize $|b_j\rangle_B$ so that $b_j = 1$ if $j = m - 1$, $b_j = 0$ otherwise

3    $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |0\rangle_B \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |\delta_{m-1,j}\rangle_B$;

// L_0 is the set of nodes in level 0.

4   **for** $|L_0|$ *times* **do**

// Read node label $\ell(v_i)$ in C

5      $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |i\rangle_I |0\rangle_C \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |i\rangle_I |\ell(v_i)\rangle_C$;

// Read the matrix entries for character $\ell(v_i)$ in M

6      $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |0\rangle_M \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |m_{\ell(v_i),j}\rangle_M$;

// Apply Toffoli to qubits M, A and V_i

7      $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |m_{\ell(v_i),j}\rangle_M |\delta_{0,j}\rangle_A |0\rangle_{V_i} \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |m_{\ell(v_i),j}\rangle_M |\delta_{0,j}\rangle_A |m_{\ell(v_i),j} \wedge \delta_{0,j}\rangle_{V_i}$;

// Reset M and C

8      $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |m_{\ell(v_i),j}\rangle_M \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_C |m_{\ell(v_i),j} \oplus m_{\ell(v_i),j}\rangle_M =$

     $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |\ell(v_i)\rangle_{C_i} |0\rangle_M$;

9      $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |i\rangle_I |\ell(v_i)\rangle_C \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |i\rangle_I |\ell(v_i) \oplus \ell(v_i)\rangle_C = \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J |i\rangle_I |0\rangle_C$;

// Increase I by one to visit the next node

10      $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |1\rangle_Q |i\rangle_I \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |1\rangle_Q |i+1\rangle_I$;

---

1 **Function** `IncreaseI(I, M, C)`:

    `// Reset M and C`

2 $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |\ell(v_i)\rangle_C \, |m_{\ell(v_i),j}\rangle_M \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |\ell(v_i)\rangle_C \, |m_{\ell(v_i),j} \oplus m_{\ell(v_i),j}\rangle_M \to$

     $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |\ell(v_i)\rangle_{C_i} \, |0\rangle_M;$

3 $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |i\rangle_I \, |\ell(v_i)\rangle_C \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |i\rangle_I \, |\ell(v_i) \oplus \ell(v_i)\rangle_C = \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |i\rangle_I \, |0\rangle_C;$

    `// Increase I`

4 $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |1\rangle_Q \, |i\rangle_I \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |1\rangle_Q \, |i+1\rangle_I;$

---

1 **Function** `IncreaseJ(J, A, B)`:

    `// Reset A and B`

2 $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |\delta_{0,j}\rangle_A \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |0\rangle_A;$

3 $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |\delta_{m-1,j}\rangle_B \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |0\rangle_B;$

    `// Increase J`

4 $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |1\rangle_Q \, |j\rangle_J \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |1\rangle_Q \, |j+1\rangle_J;$

    `// Reinitialize A and B`

5 $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |0\rangle_A \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |\delta_{0,j}\rangle_A;$

6 $\frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |0\rangle_B \to \frac{1}{\sqrt{m}} \sum\limits_{j=0}^{m-1} |j\rangle_J \, |\delta_{m-1,j}\rangle_B;$

C   Full proof of Theorem 5

Proof. After running the outer *for*-loop of Algorithm 1 $L-1$ times, we exit such a loop, and we know we have visited all the nodes (nodes in L_0 where visited during the initialization). If we consider Lemma 4 applied in the case of $t = n-1$, we are considering all the nodes, which means that if P has no match ending in G, then no substate of register R_{n-1} is such that $|r_{n-1,j}\rangle_{R_{n-1}} = |1\rangle$, for any j. Instead, if P has a match in G, then at least one substate of R_{n-1} is such that $|r_{n-1,j}\rangle_{R_{n-1}} = |1\rangle$, for some j.

The *for* loop that we run at the end of the algorithm ensures to achieve high probability of success. The probability of success p in Grover's search algorithm is the sinusoidal function $p(K) = \sin^2((2K+1)\theta)$ [5], where $\theta = \sin^{-1}\left(\sqrt{\frac{M}{N}}\right)$, N is the search space, M is the number of good solutions and K is the number of iterations of the Grover's operator. This function has period $\lambda_M \approx \frac{\pi}{2}\sqrt{\frac{N}{M}} - 1$. Consider the case $M = 1$. If we choose a random number of iterations K between 1 and λ_1, we have $p(K) \geq 1/2$ with probability $1/2$. This is because half of the material of the function is above the horizontal line of $1/2$. When $p(K) \geq 1/2$, the probability of measuring a wrong result is $p_{\text{top}} \leq 1/2$. When $p(K) \geq 1/2$, the probability of measuring a wrong result is greater than $1/2$, but anyway $p_{\text{bottom}} \leq 1$. If we run the process c times, the overall probability of failure (measuring a wrong result) p_f is then

$$p_f = \left(p_{\text{top}} \frac{1}{2} + p_{\text{bottom}} \frac{1}{2} \right)^c \leq \left(\frac{1}{2}\frac{1}{2} + 1 \cdot \frac{1}{2} \right)^c = \left(\frac{3}{4} \right)^c$$

Thus, the probability of success (measuring a correct result) is $p_s = 1 - (3/4)^c$.

In the general case $1 < M \leq N$, the period λ_M of function $p(K)$ is smaller than period λ_1 of the case $M = 1$. We can still use the same random number of iterations K between 1 and λ_1, as nearly half of the material of the function $p(K)$ is above the horizontal line of $1/2$: the worst case is when λ_M is little over half of λ_1. In this case we know that $p(K)$ will be sampled uniformly over half of the range of period λ_1, but the other half may have biased sampling. Namely, the other half of the function might have more material below $1/2$ than above. To have a safe estimate, we assume that the probability of returning the wrong result in the biased case is $p_{\text{biased}} = 1$. That is, if we run the process c times, the overall probability of failure (measuring a wrong result) p_f is then

$$p_f = \left(p_{\text{biased}} \frac{1}{2} + (p_{\text{top}} \frac{1}{2} + p_{\text{bottom}} \frac{1}{2}) \frac{1}{2} \right)^c \leq \left(1 \cdot \frac{1}{2} + (\frac{1}{2}\frac{1}{2} + 1 \cdot \frac{1}{2})\frac{1}{2} \right)^c = \left(\frac{7}{8} \right)^c$$

Thus, the probability of success (measuring a correct result) is $p_s = 1 - (7/8)^c$. ◀