Quantum Speedups for Polynomial-Time Dynamic Programming Algorithms

Susanna Caroppo
Roma Tre University, Rome, Italy

Giordano Da Lozzo
Roma Tre University, Rome, Italy

Giuseppe Di Battista
Roma Tre University, Rome, Italy

Michael T. Goodrich
University of California, Irvine, CA, USA

Martin Nöllenburg
TU Wien, Austria

Abstract

We introduce a quantum dynamic programming framework that allows us to directly extend to the quantum realm a large body of classical dynamic programming algorithms. The corresponding ***quantum dynamic programming algorithms*** retain the same space complexity as their classical counterpart, while achieving a computational speedup. For a combinatorial (*search* or *optimization*) problem $\mathcal{P}$ and an instance I of $\mathcal{P}$, such a speedup can be expressed in terms of the average degree δ of the dependency digraph $G_{\mathcal{P}}(I)$ of I, determined by a recursive formulation of $\mathcal{P}$. The nodes of this graph are the subproblems of $\mathcal{P}$ induced by I and its arcs are directed from each subproblem to those on whose solution it relies. In particular, our framework allows us to solve the considered problems in $\tilde{O}(|V(G_{\mathcal{P}}(I))|\sqrt{\delta})$ time. As an example, we obtain a quantum version of the Bellman-Ford algorithm for computing shortest paths from a single source vertex to all the other vertices in a weighted n-vertex digraph with m edges that runs in $\tilde{O}(n\sqrt{nm})$ time, which improves the best known classical upper bound when $m \in \Omega(n^{1.4})$.

2012 ACM Subject Classification Theory of computation → Dynamic programming; Theory of computation → Quantum complexity theory

Keywords and phrases Dynamic Programming, Quantum Algorithms, Quantum Random Access Memory

Digital Object Identifier 10.4230/LIPIcs.WADS.2025.14

Related Version *Full Version*: `https://arxiv.org/abs/2507.00823` [16]

Funding This research was supported, in part, by MUR of Italy (PRIN Project no. 2022ME9Z78 – NextGRAAL and PRIN Project no. 2022TS4Y3N – EXPAND), and the U.S. NSF under grant 2212129.

1 Introduction

Quantum computing represents a paradigm shift in computation, leveraging the unique principles of quantum mechanics – superposition, entanglement, and interference – to solve problems that are intractable for classical computers. These principles allow quantum algorithms to achieve significant speedups for tasks such as factoring large numbers [54], searching unsorted databases [39], simulating complex physical systems [34], computational geometry [2, 5, 7, 29], and graph drawing [14, 15, 27, 28]. Classical computing has introduced fundamental algorithmic design paradigms that enable the efficient solution of combinatorial problems. Among these, for problems that exhibit a recursive structure, *dynamic programming*

and *divide and conquer*, stand out predominantly for their efficiency and wide applicability. In this research, we introduce a framework that extends classical dynamic programming algorithms [9, 10, 11, 20, 21, 24, 33, 40, 41, 42, 43, 47, 49] to faster quantum counterparts. This framework applies to combinatorial search and optimization problems tackled by computing dynamic programming tables.

Dynamic Programming. Let $\mathcal{P}$ be a combinatorial problem and let I be an instance of $\mathcal{P}$ of size n. The **dynamic programming paradigm** for algorithm design can often be applied to compute a solution $\mathrm{sol}(I)$ of I for $\mathcal{P}$, if $\mathcal{P}$ satisfies the *optimal substructure property*, i.e., an optimal solution for I can be decomposed into (interchangeable) optimal solutions for subinstances of I, and the recursive computations of solutions for larger instances require solutions for *overlapping subproblems* – unlike most divide-and-conquer algorithms, in which the subproblems do not overlap. The underlying idea of (bottom-up) dynamic programming is to create a table D, which stores the (values of) optimal solutions for all relevant subinstances of I, starting from the smallest ones and then computing optimal solutions for larger and larger subinstances by suitably combining the existing solutions of smaller subinstances. Often, the construction of $\mathrm{sol}(I)$, given D, is a simple task[1]. Therefore, the time and space complexity for solving $\mathcal{P}$ are asymptotically bounded by those of constructing and storing D, respectively. In particular, the construction time of D can be easily upper bounded by multiplying its size by the time required to recursively compute each table entry. For instance, a table D of size n^2 and a linear-time computation for each entry yield a running time of $O(n^3)$.

Quantum Dynamic Programming. Quantum dynamic programming algorithms have recently attracted considerable interest in the exponential-time regime (which we review later). Surprisingly, instead, polynomial-time quantum algorithms have not yet received as much attention, despite the potential for significant speedups in practical applications. Khadiev and Safina [45] proposed polynomial-time quantum dynamic programming algorithms for solving problems on directed acyclic graphs (DAGs). Furrow [30] gives a polynomial-time quantum dynamic programming algorithm for the Coin Change problem and for the Maximum Subarray Sum[2] problem.

The pioneering work by Ambainis et al. [4] introduced two quantum dynamic programming frameworks that provide a novel approach for speeding up some **exponential-time** classical dynamic programming algorithms, developed to address NP-complete problems. Both frameworks use classical computation to construct partial dynamic programming tables, stored in QRAM, to be accessed via quantum search subroutines. The first, and simpler, framework addresses a subclass of set problems where the solution for a set S of size n can be determined by considering all partitions of S into two sets of sizes k and $n-k$ (for any positive integer k), and by selecting the optimal option. In this setting, a quantum advantage is obtained by suitably selecting the subset of the entries to be classically precomputed so as to balance the time needed to compute the solution (obtained by identifying optimal combinations of subsets) to the problem by performing (bounded-depth) recursive quantum

[1] The solution $\mathrm{sol}(I)$ of the actual instance I is frequently found in the "last" cell of D. Such entry is later denoted as $D[i_1^*][i_2^*]\dots[i_k^*]$.

[2] It is worth remarking that, as discussed in [30], the proposed algorithm, albeit based on computing an auxiliary table, is not solved via dynamic programming, but rather by following a greedy approach.

search primitives. For instance, this framework allows to obtain fast exponential-time quantum algorithms for Traveling Salesman [4], for One-Sided Crossing Minimization [13, 14][3], for Graph Coloring [53], and for Dynamic programming across the subset [56]. The second, more complex, framework is based on the ability of efficiently solving a *Path in the Hypercube* (PHC) problem. Given a subgraph G of the Boolean hypercube, where edges are directed from vertices of lower Hamming weight to vertices of higher Hamming weight, the problem asks to determine if there exists a directed path in G from the vertex 0^n to the vertex 1^n. In this setting, a quantum advantage is reached by combining the access to precomputed partial dynamic programming tables, with recursive applications of the quantum algorithm solving PHC. For instance, this framework allows to obtain fast exponential-time quantum algorithms for Domatic Number [6] and for Treewidth Computation [48]. Furthermore, Glos et al. [36] generalized this framework to a quantum algorithm for finding a path in n-dimensional lattice graphs.

Our contributions. While exponential-time quantum algorithms aim to tackle problems beyond the reach of classical computation, polynomial-time quantum algorithms can provide substantial efficiency gains for problems already considered tractable. In this work, we focus on polynomial-time quantum algorithms and introduce a framework that systematically extends classical dynamic programming algorithms into accelerated quantum counterparts, by harnessing quantum parallelism and amplitude amplification. Specifically, we developed quantum subroutines that integrate quantum search primitives, such as *min*, *max*, *find*, and *findAll*, within a dynamic programming context. These subroutines construct a superposition enabling access to the specific subset of previously computed entries, stored in a Quantum Random Access Memory (QRAM), that is needed for computing the currently-considered entry. This, in turn, enables us to harness the full potential of quantum search primitives by restricting the search space to the relevant candidates.

To demonstrate the broad usability of our framework, we apply it to several well-known problems for which the best worst-case classical algorithms rely on dynamic programming; see Table 1. For space reasons, in this paper, we only discuss in detail the application of the framework to the Single-Source Shortest Path and Membership in Context-Free Language problems. The applications to the remaining problems in Table 1 are presented in the full version of the paper [16].

2 Preliminaries

In this section, we introduce preliminary notation and definitions.

Notation. Given two k-tuples of integers, $\langle a_1, a_2, \ldots, a_k\rangle$ and $\langle b_1, b_2, \ldots, b_k\rangle$, we say that $\langle a_1, a_2, \ldots, a_k\rangle$ *lexicographically precedes* $\langle b_1, b_2, \ldots, b_k\rangle$, denoted by $\langle a_1, a_2, \ldots, a_k\rangle \prec \langle b_1, b_2, \ldots, b_k\rangle$, if there exists an index $j \in \{1, 2, \ldots, k\}$ such that $a_i = b_i$ for all $i < j$, and $a_j < b_j$. The relation $\prec$ defines a total order among the k-tuples of integers. Given a directed graph $G = (V, E)$, we denote by $\deg(v)$ the *degree* of a vertex v of G, that is, the number of arcs in E having v as their tail or head. Moreover, we denote by $\deg_{out}(v)$ the *outdegree* of a vertex v of G, that is, the number of arcs in E having v as their tail. The average degree of

[3] A quantum algorithm for Two-Sided Crossing Minimization, solely based on Grover's search, has been presented in [17, 15].

Table 1 Comparison of classical and quantum time complexities, classified based on the operator $\mathrm{op} \in \{\min, \max, \mathrm{find}, \mathrm{findAll}\}$ in their recursive fomulation.

Problem	op	Classical Complexity	Quantum Complexity
Minimum-Weight Triangulation of Convex Polygon	min	$O(n^3)$ [49]	$\tilde{O}(n^2\sqrt{n})$
All-Pairs Shortest Paths	min	$O(n^3 \log\log n/\log^2 n)$ [40]	$\tilde{O}(n^2\sqrt{n}\log n)$
Single-Source Shortest Paths Section 5.1	min	$\tilde{O}(mn^{4/5})$ [42]	$\tilde{O}(n\sqrt{nm})$
Multi-Criteria Boundary Labeling	min	*po*-leaders $O(n^3)$ [10, 11]	$\tilde{O}(n^2\sqrt{n})$
	min	*do*-leaders $O(n^5)$ [10, 11]	$\tilde{O}(n^4\sqrt{n})$
Segmented Least Squares	min	$O(n^2)$ [9, 47]	$\tilde{O}(n\sqrt{n})$
RNA Secondary Structure	max	$O(n^3)$ [47]	$\tilde{O}(n^2\sqrt{n})$
Rod Cutting	max	$O(n^2)$ [20]	$\tilde{O}(n\sqrt{n})$
Largest Divisible Subset in Array	max	$O(n^2)$ [25]	$\tilde{O}(n\sqrt{n})$
Unbounded Knapsack	max	$O(Wn)$ [21]	$\tilde{O}(W\sqrt{n})$
Viterbi Path Problem	max	$O(T \times \lvert S\rvert^2)$ [43]	$\tilde{O}(T \times \lvert S\rvert \times \sqrt{\lvert S\rvert})$
Text Segmentation	find	$O(n^2)$ [24]	$\tilde{O}(n\sqrt{n})$
Membership in Context-Free Language (CYK) Section 5.2	findAll	$O(n^{2.37})$ [58]	$\tilde{O}(n^2\sqrt{n})$

G is $\delta = \frac{1}{n}\sum_{v\in V(G)} \deg(v) = \frac{2m}{n}$. In order to simplify the notation, we use $[h]$ to denote the set $\{0, \ldots, h-1\}$, where h is a positive integer. Also, given positive integers a and b, we denote $\lceil \frac{a}{b} \rceil$ as $\frac{a}{b}$ and $\lceil \log a \rceil$ as $\log a$. If $f(n) = O(\log^c n)$ for some constant c, we write $f(n) = polylog(n)$. In case $f(n) = O(n^d polylog(n))$ for some constant d, we use the notation $f(n) = \tilde{O}(n^d)$ (see, e.g., [59]).

Combinatorial problems. A *combinatorial search problem* $\mathcal{P}$ is a triple $\langle \Lambda, S, R\rangle$, where:

- Λ is the set of *instances* of $\mathcal{P}$;
- S is the set of *solutions* of $\mathcal{P}$; and
- $R \subseteq \Lambda \times S$ is a binary relation that associates each instance $I \in \Lambda$ with a set $SOL(I) = \{s \in S : (I, s) \in R\}$. The elements in $SOL(I)$ are the *feasible solutions* of $\mathcal{P}$ for I.

A *combinatorial optimization problem* $\mathcal{P}$ is a quintuple $\langle \Lambda, S, R, f_{\mathcal{P}}, g\rangle$, where:

- $\langle \Lambda, S, R\rangle$ is a combinatorial search problem,
- $f_{opt} : S \to Y$ is the *optimization function*, where Y is a totally ordered set (usually $Y \in \{\mathbb{N}, \mathbb{R}\}$);
- $g : 2^S \to S$ is the *comparator function*, with $g \in \{\min, \max\}$.

An *optimal solution* of $\mathcal{P}$ for I is any feasible solution s^* in $SOL(I)$ such that $s^* = \arg\min_{s\in SOL(I)} f_{opt}(s)$, if $g = \min$, and $s^* = \arg\max_{s\in SOL(I)} f_{opt}(s)$, if $g = \max$. For ease of notation, in the following, we denote by $\mathrm{sol}(I)$, both a feasible solution of a combinatorial search problem and an optimal solution of a combinatorial optimization problem. Also, we refer to combinatorial search/optimization problems simply as *combinatorial problems*.

Dynamic programming. Let $\mathcal{P}$ be a combinatorial problem and let I be an instance of $\mathcal{P}$. A (bottom-up) dynamic programming algorithm for $\mathcal{P}$ can be implemented by executing the following steps:

Dynamic Programming Steps:

Table Setup: Determine integers $d_1, d_2, \ldots, d_k$ that depend on I and create a table D of dimension $d_1 \times d_2 \times \ldots \times d_k$. Initialize some "easy" entries of D (Base Case) directly using information from I and initialize the remaining "difficult" entries with a default value representing the fact that such entries have not yet acquired their final value.

Table Update: Compute the "difficult" entries of D (Recursive Case) by evaluating a **recursive formula** that expresses each entry $D[i_1][i_2]\ldots[i_k]$ in terms of a subset of the (already computed) entries $D[j_1][j_2]\ldots[j_k]$ corresponding to optimal solutions for structurally related subproblems such that $(j_1, j_2 \ldots, j_k) \prec (i_1, i_2 \ldots, i_k)$.

Solution Retrieval: Return the value contained in a specific entry of $D[i_1^*][i_2^*]\ldots[i_k^*]$ (whose position $\langle i_1^*, i_2^*, \ldots, i_k^*\rangle$ in D depends on I and $\mathcal{P}$) or retrieve a solution of I by inspecting D (usually exploiting the information contained in $D[i_1^*][i_2^*]\ldots[i_k^*]$).

Whereas the **efficiency** of this algorithm design paradigm lies in the fact that solutions of *overlapping problems* are stored in the table and hence need to be computed only once, its **correctness** lies in the fact that "difficult" entries can be computed by exploiting the values of previously computed ("easy" and "difficult") entries (by the *optimal substructure property*). Clearly, the **Table Update** is the most interesting and challenging step in the overall approach. Fortunately, many optimization problems $\mathcal{P}$, including those considered in this paper, naturally exhibit a simple recursive formulation for the entries of their dynamic programming table D. Consider an entry $D[i_1][i_2]\ldots[i_k]$ of D. Let $S_{i_1,i_2,\ldots,i_k}$ be the *dependency set* of $D[i_1][i_2]\ldots[i_k]$, composed of the indices of the entries of D on which the computation of the entry $D[i_1][i_2]\ldots[i_k]$ depends. Observe that $S_{i_1,i_2,\ldots,i_k}$ is some subset of $\mathbb{N}^k$ such that for each entry $\langle j_1, j_2, \ldots, j_k\rangle$ in $S_{i_1,i_2,\ldots,i_k}$ we have that $\langle j_1, j_2, \ldots, j_k\rangle \prec \langle i_1, i_2, \ldots, i_k\rangle$. Then, the recursive formula for $D[i_1][i_2]\ldots[i_k]$ is of the form:

$$D[i_1][i_2]\ldots[i_k] = \mathrm{op}_{\mathcal{X}\in\mathcal{C}_{i_1,i_2,\ldots,i_k}} f_{\mathcal{P}}(i_1, i_2, \ldots, i_k, \mathcal{X}), \tag{1}$$

where: **(1)** $\mathrm{op} \in \{\min, \max, \mathrm{find}, \mathrm{findAll}\}$; **(2)** $C_{i_1,i_2,\ldots,i_k}$ is a set, called *generating set*, composed of h-element subsets of $S_{i_1,i_2,\ldots,i_k}$, where each subset provides the input to construct a particular candidate value for $D[i_1][i_2]\ldots[i_k]$ (where h is an integer constant, called *dependency index*, determined by $\mathcal{P}$, which specifies the number of subinstances into which each instance is divided in the recursive definition); **(3)** $f_{\mathcal{P}}$ is a function specific for problem $\mathcal{P}$ that computes a candidate value for $D[i_1][i_2]\ldots[i_k]$, assuming that all entries of D with indices in $S_{i_1,i_2,\ldots,i_k}$ have already been computed.

▶ Remark 1. The *optimal substructure property* of a problem $\mathcal{P}$ is formally captured by the dependency set $S_{i_1,i_2,\ldots,i_k}$, whose entries $\langle j_1, j_2, \ldots, j_k\rangle$ must satisfy $\langle j_1, j_2, \ldots, j_k\rangle \prec \langle i_1, i_2, \ldots, i_k\rangle$.

In most problems, the dependency index is a small integer, usually equal to 1 or 2. A textbook example of a problem fitting Equation (1) with $h = 1$ is the Coin Change problem [47]. Given a set of positive integer coin denominations $c_1 < c_2 < \cdots < c_r$ and a target sum T, the goal is to achieve T using the fewest possible number of coins, assuming an unlimited supply of each denomination, or determine if it is not possible to obtain T. Let D be a dynamic programming table of size $T + 1$, whose entries $D[i]$ represent the minimum number of coins needed to make up the sum i, with $D[i] = \infty$ if it is not possible. The base case of the dynamic programming approach is $D[0] = 0$. For the recursive case, to compute

$D[i]$ for $i > 0$, we have to consider all possible choices for the first coin. Once the first coin is selected, the remaining amount must be reached optimally. This can be expresses by the recursive formula

$$D[i] = \begin{cases} 0, & \text{if } i = 0 \\ \min\limits_{j:c_j \leq i}(1 + D[i - c_j]), & \text{if } i > 0 \end{cases} \tag{2}$$

Clearly, the recursive case of Equation (2) matches the pattern of Equation (1). In fact, we can set $S_i = \{j : c_j \leq i\}$, $C_i = \{\{j\} : c_j \leq i\}$, $h = 1$, and $f_{\mathcal{P}}(i, \{j\}) = 1 + D[i - c_j]$.

A notable example of a problem fitting Equation (1) with $h = 2$ is the Matrix Chain Multiplication problem. Given a sequence of n matrices, $A_1, A_2, \ldots, A_n$, and their dimensions $p_0, p_1, p_2, \ldots, p_n$, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, the problem asks to determine the order of matrix multiplications that minimizes the total number of scalar multiplications needed to obtain $A_1 \times A_2 \times \ldots \times A_n$. Note that, in the Matrix Chain Multiplication problem, we are not actually multiplying matrices; instead, the goal is only to determine an order for multiplying matrices that has the lowest cost. Recall that matrix multiplication is associative, therefore we aim at grouping the above multiplications to minimize the total number of scalar multiplications.

Let D be a dynamic programming table of size $n \times n$, whose entries $D[i][j]$, with $1 \leq i \leq j \leq n$, store the minimum number of scalar multiplications needed to compute the product $A_i \times A_{i+1} \times \ldots \times A_j$. The base case of the dynamic programming approach is $D[i][i] = 0$. For the recursive case, to compute $D[i][j]$ for $j > i$, we have to consider all possible choices to split the product at a matrix A_k, with $i \leq k < j$. This can be expressed by the recursive formula:

$$D[i][j] = \begin{cases} 0, & \text{if } i = j \\ \min\limits_{i \leq k < j}(D[i][k] + D[k+1][j] + p_{i-1}p_k p_j), & \text{if } i < j \end{cases} \tag{3}$$

Clearly, the recursive case of Equation (3) matches the pattern of Equation (1). In fact, we can set $S_{i,j} = \{(i, k), (k + 1, j) : i \leq k < j\}$, $C_{i,j} = \{\{(i, k), (k + 1, j)\} : i \leq k < j\}$, $h = 2$, and $f_{\mathcal{P}}(i, j, \{(i, k), (k + 1, j)\}) = D[i][k] + D[k + 1][j] + p_{i-1}p_k p_j$.

3 Quantum Tools

In this section, we provide the reader with the quantum primitives needed in this research. For a comprehensive introduction to the quantum computing field see, e.g., Nielsen and Chuang [50], and Aaronson [1].

Qubits are the fundamental units of quantum computing. They differ from classical bits in that they can exist in a superposition of the two classical states **0** and **1**. This unique characteristic enables quantum computers to perform, in parallel, multiple computations, allowing in some cases to achieve a substantial (and sometimes exponential) speedup over classical systems. Mathematically, a qubit is represented in a Hilbert space as a two-dimensional vector[4] $|\psi\rangle = \binom{\alpha}{\beta} \in \mathbb{C}^2$, that is, $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$ are the two orthonormal basis states and the complex coefficients α and β are the *amplitudes* of such states. The likelihood of measuring the qubit in state $|0\rangle$ or $|1\rangle$ is given by $|\alpha|^2$

[4] In Dirac's notation, a *ket* such as $|v\rangle$, where v is an arbitrary label, represents a vector corresponding to a quantum state.

and $|\beta|^2$, respectively. Therefore, α and β must satisfy the **normalization condition** $|\alpha|^2+|\beta|^2= 1$, ensuring that the total probability remains 1. A *quantum state* over n qubits is a unit vector in the Hilbert space $\mathbb{C}^{2^n}$. The *computational basis* $\{|j\rangle\}_{j\in[2^n]}$ consists of quantum states, where $|j\rangle$ is the unit vector with a 1 in the j-th index and 0 elsewhere. A computational basis state $|j\rangle$ can be interpreted as a classical bit string j, allowing quantum systems to simulate classical algorithms. Any quantum state can be expressed as a weighted sum (or *superposition*) of these basis states:

$$|\Psi\rangle = \sum_{j=0}^{2^n-1} \alpha_j |j\rangle ,$$

where the amplitudes satisfy the normalization condition $\sum_{j=0}^{2^n-1}|\alpha_j|^2= 1$. If at least two coefficients α_j in the above expression are nonzero, the state is said to be in *superposition*. When measuring $|\Psi\rangle$, the state collapses to $|j\rangle$ with probability $|\alpha_j|^2$.

In this paper, we focus on quantum computations performed in the **circuit model of computation**. In this model, quantum algorithms are specified by *quantum circuits*, obtained by composing quantum gates, that perform specific quantum computations. The process begins with the initialization of qubits, followed by the application of gate operations that modify their states. Finally, the circuit's output is obtained by measuring the qubits. This operation collapses their quantum states into classical binary outcomes. *Quantum gates* are the fundamental building blocks of quantum circuits, analogous to classical logic gates in traditional computing. They are used to manipulate quantum states while preserving key quantum properties like superposition and entanglement. Specifically, a quantum gate performs a linear transformation on its input quantum state, meaning that a superposition of states is mapped to the corresponding superposition of their images. In particular, any such a transformation U must be *unitary*, satisfying the condition $\mathbb{I} = U^\dagger U = UU^\dagger$, where $\mathbb{I}$ denotes the identity matrix and $U^\dagger$ denotes the transpose conjugate of U.

As a consequence, quantum computation is inherently *reversible* as long as no measurement is performed. That is, given an output quantum state $|\phi\rangle$, obtained by applying U to an initial quantum state $|\psi\rangle$, the original state can be fully recovered by applying $U^{-1} = U^\dagger$ to $|\phi\rangle$. An important quantum gate (often used in the initialization of qubits) is the Hadamard gate $H = \frac{1}{\sqrt{2}}\left(\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right)$, which can be used a preliminary step to transform the $|0\rangle$ state into to a uniform state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ of the two basis states.

Quantum Random Access Memory (QRAM) is the "quantum analog" of conventional RAM, designed to store and access data in a quantum format. Like RAM, QRAM consists of three main components: an input (or address) register a, an output (or data) register d, and memory arrays. However, unlike traditional RAM, QRAM's input and output registers are composed of qubits rather than classical bits, while the memory arrays can be either classical or quantum, depending on the application [35]. A key feature of QRAM lies in its method of memory access. Instead of retrieving data from a single memory location at a time, QRAM leverages quantum superposition to access multiple memory locations simultaneously. Specifically, while a classical RAM uses n bits to randomly access one of $N = 2^n$ distinct memory cells, a QRAM uses n qubits to address a quantum superposition of all N memory cells simultaneously. When a quantum computer requires access to a superposition of memory cells, the address register[5] a must hold a superposition of addresses,

[5] A set comprising multiple qubits is a *register*. A quantum register a with n qubits $|q_i\rangle$, with $i \in [n]$, is denoted as a tensor product $|\psi\rangle_a = |q_0\rangle \otimes |q_1\rangle \otimes \cdots \otimes |q_{n-1}\rangle$.

Algorithm 1 Procedure StatePrep prepares a uniform superposition over the set $C_{i_1,i_2,\ldots,i_k}$.

procedure StatePrep($i_1 i_2 \ldots i_k$)
  input: Registers $|i_1 i_2 \ldots i_k\rangle$ and an ℓ-bit zero register $|0^\ell\rangle$
  output: Uniform superposition state $|\Psi_{i_1,i_2,\ldots,i_k}\rangle$
  Apply the quantum circuit U_{f_C} to determine $\lambda_{i_1,i_2,\ldots,i_k} = |C_{i_1,i_2,\ldots,i_k}|$
  Prepare the state $|i_1 i_2 \ldots i_k\rangle\,|\lambda_{i_1,i_2,\ldots,i_k}\rangle$
  Introduce an additional $\log(\lambda_{i_1,i_2,\ldots,i_k})$-bit zero register $|0^{\log \lambda_{i_1,i_2,\ldots,i_k}}\rangle$
  Apply Hadamard gates to the qubits of the last register to prepare the uniform superposition:

$$|\Psi_{i_1,i_2,\ldots,i_k}\rangle = \frac{1}{\sqrt{\lambda_{i_1,i_2,\ldots,i_k}}}\,|i_1, i_2, \ldots, i_k\rangle\,|\lambda_{i_1,i_2,\ldots,i_k}\rangle \sum_{u=0}^{\lambda_{i_1,i_2,\ldots,i_k}-1} |u\rangle$$

  return $|\Psi_{i_1,i_2,\ldots,i_k}\rangle$
end procedure

represented as $\sum_j \alpha_j\,|j\rangle_a$. In response, the QRAM returns a corresponding superposition of data in the data register d, ensuring that the retrieved data remains correlated with the address register:

$$\sum_j \alpha_j\,|j\rangle_a\,|0^k\rangle_d \overset{QRAM}{\longrightarrow} \sum_j \alpha_j\,|j\rangle_a\,|\delta_j\rangle_d\,,$$

where δ_j represents the content of the j-th memory cell, k bits suffice to classically encode the content of any memory cell, and $|0^k\rangle$ denotes the quantum basis state composed of k qubits set to $|0\rangle$. QRAM plays a crucial role in several quantum algorithms, such as quantum searching [39], minimum/maximum finding [23], counting [12], period finding and discrete logarithm [54], to cite a few. In particular, the use of QRAM enables us to exploit quantum search primitives that involve condition checking on data stored in random access memory. Specifically, the QRAM may be used by an oracle to check conditions based on the data stored in memory, marking the superposition states that correspond to feasible or optimal solutions [35, 51, 60, 62].

We now describe the **quantum subroutines** used in our algorithms. Observe that every function f implemented as a classical circuit can be approximated with arbitrary accuracy, using a discrete set of quantum gates, by a quantum circuit with only a polylogarithmic overhead [22, 46, 50]. Therefore, in the following, we assume that every classical function f, provided as classical circuit C_f, can be passed to and accessed by our quantum procedures as a quantum version of C_f. In the remainder, for any computable function f, we use the notation U_f to denote the corresponding quantum circuit.

Let W be a ground set and let ω be an element of W. Let $bin(\omega)$ be the integer corresponding to a binary encoding of ω. For ease of notation, in the remainder, we will denote the state $|bin(\omega)\rangle$ simply as $|\omega\rangle$. In particular, given an element $\mathcal{X} \in C_{i_1,i_2,\ldots,i_k}$, we will use the state $|\mathcal{X}\rangle$.

Consider a computational problem $\mathcal{P}$ that can be solved via dynamic programming building a k-dimensional table D of size $d_1 \times \ldots \times d_k$. Recall that by $C_{i_1,i_2,\ldots,i_k}$ we denote the generating set of an entry $D[i_1][i_2]\ldots[i_k]$ of D. We let $\lambda_{i_1,i_2,\ldots,i_k} = |C_{i_1,i_2,\ldots,i_k}|$ and we let $\sigma = \sum_{i=1}^{k} \log d_i$.

Algorithm 2 Procedure TableIndexPrep for retrieving the u-th element of $C_{i_1,i_2,\ldots,i_k}$.

procedure TableIndexPrep$(i_1, i_2, \ldots, i_k, u)$
  input: Registers $|i_1, i_2, \ldots, i_k\rangle$, the register $|u\rangle$, and an $h \cdot \sigma$-bit zero register $|0^{h\cdot\sigma}\rangle$
  output: State $|E_{i_1,i_2,\ldots,i_k,u}\rangle$ encoding the u-th element $\gamma(\langle i_1, i_2, \ldots, i_k\rangle, u)$ of $C_{i_1,i_2,\ldots,i_k}$
  Apply the quantum circuit U_γ to retrieve the element $X_u = \gamma(\langle i_1, i_2, \ldots, i_k\rangle, u)$
  Store X_u in the output register, thus obtaining the state:

$$|E_{i_1 i_2 \ldots i_k,u}\rangle = |i_1 i_2 \ldots i_k\rangle\,|u\rangle\,|X_u\rangle$$

  return $|E_{i_1 i_2 \ldots i_k,u}\rangle$
end procedure

Subroutine StatePrep. The first subroutine prepares a uniform superposition $\Psi_{i_1,i_2,\ldots,i_k}$ of the integers in $[\lambda_{i_1,i_2,\ldots,i_k}]$; see the pseudocode of Algorithm 1. It assumes the existence of a classical procedure f_C that determines $\lambda_{i_1,i_2,\ldots,i_k}$ given the values $i_1, i_2, \ldots, i_k$. The subroutine starts with the register $|i_1 i_2 \ldots i_k\rangle$ and with a register storing an ℓ-bit zero string $|0^\ell\rangle$ (where the value of ℓ will be discussed later). It exploits the quantum gate U_λ to determine $\lambda_{i_1,i_2,\ldots,i_k}$ and prepares the state $|i_1 i_2 \ldots i_k\rangle\,|\lambda_{i_1,i_2,\ldots,i_k}\rangle$. Then, the subroutine takes in input an additional register storing a $\log(\lambda_{i_1,i_2,\ldots,i_k})$-bit zero string and applies Hadamard gates to each of the qubits of such a register to prepare the uniform superposition state:

$$|\Psi_{i_1,i_2,\ldots,i_k}\rangle = \frac{1}{\sqrt{\lambda_{i_1,i_2,\ldots,i_k}}}\,|i_1 i_2 \ldots i_k\rangle\,|\lambda_{i_1,i_2,\ldots,i_k}\rangle \sum_{u=0}^{\lambda_{i_1,i_2,\ldots,i_k}-1} |u\rangle$$

We use the signature `StatePrep`$(i_1, i_2, \ldots, i_k)$ to denote calls to the subroutine StatePrep.

Subroutine TableIndexPrep. The second quantum subroutine prepares the state $|E_{i_1 i_2 \ldots i_k,u}\rangle$ that correlates the integers in $[\lambda_{i_1,i_2,\ldots,i_k}]$ with the elements of $C_{i_1 i_2 \ldots i_k}$; see the pseudocode of Algorithm 2. Our quantum subroutine assumes the existence of a classical **injective function** $\gamma : \mathbb{N}^k \times \mathbb{N} \to C_{i_1 i_2 \ldots i_k}$ that maps the pair $\langle\langle i_1 i_2 \ldots i_k\rangle, u\rangle$ to an element of $C_{i_1 i_2 \ldots i_k}$. Observe that the elements of $C_{i_1,i_2,\ldots,i_k}$ admit a binary representation $bin(\mathcal{X})$ of length $h \cdot \sigma$. The subroutine starts with the register $|i_1 i_2 \ldots i_k\rangle$, the register $|u\rangle$, and with a register storing an $(h \cdot \sigma)$-bit zero string $|0^{h\cdot\sigma}\rangle$ and uses U_γ to prepare the state:

$$|E_{i_1 i_2 \ldots i_k,u}\rangle = |i_1 i_2 \ldots i_k\rangle\,|u\rangle\,|\gamma(\langle i_1, i_2, \ldots, i_k\rangle, u)\rangle\,.$$

We use the signature `TableIndexPrep`$(i_1, i_2, \ldots, i_k, u)$ to denote calls to the subroutine TableIndexPrep.

Subroutine DP. The third quantum subroutine uses either **(i)** the quantum min/max finding algorithms (QMin and QMax) due to Dürr and Høyer [23], or **(ii)** the quantum finding algorithm (QFind) due to Grover to search for an item satisfying a condition in an unsorted list [1] or **(iii)** the quantum finding algorithm (QFindAll) due to Ambainis to search for all items satisfying a condition in an unsorted list [3]. In particular, QMin and QMax allow to determine the element in a ground set of size N that minimizes/maximizes a given function in $\tilde{O}(\sqrt{N})$ time, QFind (resp. QFindAll) allows to determine an element (resp. all the elements) in a ground set of size N that satisfy a certain condition in $\tilde{O}(\sqrt{N})$ time (resp. $\sqrt{NM}$ time, where M is the number of elements satisfying the condition).

Algorithm 3 Procedure DP for computing the value $\mathrm{op}_{\mathcal{X}\in\mathcal{C}_{i_1,i_2,\ldots,i_k}} f_{\mathcal{P}}(i_1,i_2,\ldots,i_k,\mathcal{X})$. We use the signature $\texttt{QMaxMin}(i_1,i_2,\ldots,i_k,f_{\mathcal{P}},\preccurlyeq)$, $\texttt{QMaxMax}(i_1,i_2,\ldots,i_k,f_{\mathcal{P}},\max,\preccurlyeq)$, $\texttt{QFind}(i_1,i_2,\ldots,i_k,f_{\mathcal{P}})$, and $\texttt{QFindAll}(i_1,i_2,\ldots,i_k,f_{\mathcal{P}})$ to denote calls to the algorithms QMaxMin, QMaxMin, QFind, and QFindAll, respectively.

procedure $\texttt{DP}(i_1,i_2,\ldots,i_k,f_{\mathcal{P}},op,\preccurlyeq=\texttt{null})$
  input: Registers $|i_1,i_2,\ldots,i_k\rangle$, the function $f_{\mathcal{P}}$, and an operator op $\in$ {min, max, find, findAll}. Additionally, a comparator $\preccurlyeq$, if op $\in$ {min, max}.
  output: The value $\mathrm{op}_{\mathcal{X}\in\mathcal{C}_{i_1,i_2,\ldots,i_k}} f_{\mathcal{P}}(i_1,i_2,\ldots,i_k,\mathcal{X})$ to be assigned to $D[i_1][i_2]\ldots[i_k]$
  Invoke the subroutine $\texttt{StatePrep}(i_1,i_2,\ldots,i_k)$ to prepare the superposition state $|\Psi_{i_1,i_2,\ldots,i_k}\rangle$
  Apply the subroutine $\texttt{TableIndexPrep}(i_1,i_2,\ldots,i_k,u)$, in parallel to each basis state of $|\Psi_{i_1,i_2,\ldots,i_k}\rangle$, to prepare the superposition $|\Phi_{i_1,i_2,\ldots,i_k}\rangle$
  if op = min **then**
    Apply $\texttt{QMaxMin}(i_1,i_2,\ldots,i_k,f_{\mathcal{P}},\preccurlyeq)$ to $|\Phi_{i_1,i_2,\ldots,i_k}\rangle$
  else if op = max **then**
    Apply $\texttt{QMaxMax}(i_1,i_2,\ldots,i_k,f_{\mathcal{P}},\preccurlyeq)$ to $|\Phi_{i_1,i_2,\ldots,i_k}\rangle$
  else if op is find **then**
    Apply $\texttt{QFind}(i_1,i_2,\ldots,i_k,f_{\mathcal{P}})$ to $|\Phi_{i_1,i_2,\ldots,i_k}\rangle$
  else if op is findAll **then**
    Apply $\texttt{QFindAll}(i_1,i_2,\ldots,i_k,f_{\mathcal{P}})$ to $|\Phi_{i_1,i_2,\ldots,i_k}\rangle$
  end if
  return $\mathrm{op}_{\mathcal{X}\in\mathcal{C}_{i_1,i_2,\ldots,i_k}} f_{\mathcal{P}}(i_1,i_2,\ldots,i_k,\mathcal{X})$
end procedure

The subroutine (see the pseudocode of Algorithm 3) takes as input at least: **(1)** the integers $i_1,i_2,\ldots,i_k$; **(2)** the function $f_{\mathcal{P}}$; and **(3)** an operator op $\in$ {min, max, find, findAll}. Furthermore, if op $\in$ {min, max}, it additionally takes as an input a comparator, $\preccurlyeq$, to maximize (or minimize) over, that defines a total ordering of the values in the codomain of $f_{\mathcal{P}}$ (i.e., the values stored in D). First, the subroutine invokes $\texttt{StatePrep}$ $(i_1,i_2,\ldots,i_k)$ to prepare the superposition state $|\Psi_{i_1,i_2,\ldots,i_k}\rangle$ and then applies $\texttt{TableIndexPrep}$, in parallel to each basis state of $|\Psi_{i_1,i_2,\ldots,i_k}\rangle$, to prepare the superposition state:

$$|\Phi_{i_1,i_2,\ldots,i_k}\rangle = \frac{1}{\sqrt{\lambda_{i_1,i_2,\ldots,i_k}}}\,|i_1,i_2,\ldots,i_k\rangle\,|\lambda_{i_1,i_2,\ldots,i_k}\rangle\sum_{u=0}^{\lambda_{i_1,i_2,\ldots,i_k}-1}|u\rangle\,|\gamma(\langle i_1,i_2,\ldots,i_k\rangle,u)\rangle\,.$$

▶ Remark 2. The time $T'_{i_1,i_2,\ldots,i_k}$ required to prepare the state $|\Phi_{i_1,i_2,\ldots,i_k}\rangle$ is bounded by the time needed to execute gates $U_{f_{\mathcal{C}}}$ and U_{γ}.

Observe that each of the states $|i_1,i_2,\ldots,i_k\rangle\,|\lambda_{i_1,i_2,\ldots,i_k}\rangle\,|u\rangle\,|\gamma(\langle i_1,i_2,\ldots,i_k\rangle,u)\rangle$ composing $|\Phi_{i_1,i_2,\ldots,i_k}\rangle$ provides the input for computing, using $f_{\mathcal{P}}$ and the quantum search subroutines described above, a candidate value for $D[i_1][i_2]\ldots[i_k]$, that is, the registers $|i_1,i_2,\ldots,i_k\rangle$ and $|\gamma(\langle i_1,i_2,\ldots,i_k\rangle,u)\rangle$. Therefore, the subroutine proceeds by applying the algorithm QMin, QMax, QFind, or QFindAll, depending on whether the chosen operator op is equal to min (or max), find, or findAll, respectively, **only to** the candidate values for $D[i_1][i_2]\ldots[i_k]$ determined by the entries in $C_{i_1,i_2,\ldots,i_k}$.

We use the signatures $\texttt{DP}(i_1,i_2,\ldots,i_k,f_{\mathcal{P}},op)$ and $\texttt{DP}(i_1,i_2,\ldots,i_k,f_{\mathcal{P}},op,\preccurlyeq)$ to denote calls to the subroutine DP, when op $\in$ {find, findAll} or op $\in$ {min, max}, respectively.

Altogether, we obtain the following main algorithmic theorem.

▶ **Theorem 3.** *Let $\mathcal{P}$ be a combinatorial problem, let I be an instance of $\mathcal{P}$, and let n be the size of I. Also, let $T_{\mathcal{P}}$ be the time needed to compute quantumly the function $f_{\mathcal{P}}$ and let $T'_{i_1,i_2,\ldots,i_k}$ be the time needed to prepare $|\Phi_{i_1,i_2,\ldots,i_k}\rangle$. Suppose that each of the values of the entries of D can be represented using w bits with $w \in O(polylog(n))$. Then, the following holds for subroutine* `DP`*:*

- *If* op $=$ find, `DP` *determines a value of $f_{\mathcal{P}}$ for an entry $D[i_1][i_2]\ldots[i_k]$ in $\tilde{O}(T'_{i_1,i_2,\ldots,i_k} + T_{\mathcal{P}}\sqrt{\lambda_{i_1,i_2,\ldots,i_k}})$ time.*
- *If* op $=$ findAll, `DP` *determines all the M possible distinct values of $f_{\mathcal{P}}$ for an entry $D[i_1][i_2]\ldots[i_k]$ in $\tilde{O}(T'_{i_1,i_2,\ldots,i_k} + T_{\mathcal{P}}\sqrt{\lambda_{i_1,i_2,\ldots,i_k} M})$ time.*
- *If* op $\in \{\min, \max\}$*, consider some total ordering defined by $\lessdot$ of the data values in D such that comparison according to such an ordering can be performed in $O(w)$ time. Then,* `DP` *determines the minimum (or maximum) value of $f_{\mathcal{P}}$ for an entry $D[i_1][i_2]\ldots[i_k]$, under the specified ordering, in $\tilde{O}(T'_{i_1,i_2,\ldots,i_k} + T_{\mathcal{P}}\sqrt{\lambda_{i_1,i_2,\ldots,i_k}})$ time.*

4 Quantum Dynamic Programming

In this section, we describe a framework that exploits the quantum subroutines defined in Section 3 to obtain quantum speedups for many computational problems solved classically using dynamic programming algorithms.

The recursive formula that specifies how to compute the entries of the dynamic programming table D used for solving a computational problem $\mathcal{P}$ on a instance I determines a **dependency digraph** $G_{\mathcal{P}}(I)$. The nodes of $G_{\mathcal{P}}(I)$ are in one-to-one correspondence with the entries of D, i.e., the subproblems of $\mathcal{P}$ defined by I. For each node $n_{i_1,i_2,\ldots,i_k}$ of $G_{\mathcal{P}}(I)$ associated with the entry $D[i_1][i_2]\ldots[i_k]$ of D, graph $G_{\mathcal{P}}(I)$ contains an arc directed from $n_{i_1,i_2,\ldots,i_k}$ to each of the nodes $n_{j_1,j_2,\ldots,j_k}$ associated with the entries $D[j_1][j_2]\ldots[j_k]$ that occur in the recursive formula for $D[i_1][i_2]\ldots[i_k]$. These are the entries indexed by the tuples $\langle j_1, j_2, \ldots, j_k\rangle \in S_{i_1,i_2,\ldots,i_k}$. Clearly, by the optimal substructure property, $G_{\mathcal{P}}(I)$ is a directed acyclic graph.

Let $\mathcal{P}$ be a combinatorial problem and suppose that it admits a dynamic programming algorithm that relates the dependency set $S_{i_1,i_2,\ldots,i_k}$ and the generating set $C_{i_1,i_2,\ldots,i_k}$ of an entry $D[i_1][i_2]\ldots[i_k]$ as follows. For each element $\langle j_1, j_2, \ldots, j_k\rangle \in S_{i_1,i_2,\ldots,i_k}$, there exists a unique set $\mathcal{X} \in C_{i_1,i_2,\ldots,i_k}$ such that $\langle j_1, j_2, \ldots, j_k\rangle \in \mathcal{X}$. We say that a dynamic programming algorithm exhibiting the above characteristic is *simple* and that $\mathcal{P}$ is a *simple problem*. Observe that, for a simple dynamic programming algorithm, it holds that $\lambda_{i_1,i_2,\ldots,i_k} = |C_{i_1,i_2,\ldots,i_k}| = \frac{|S_{i_1,i_2,\ldots,i_k}|}{h}$. In particular, this implies that, for each node $n_{i_1,i_2,\ldots,i_k}$ of $G_{\mathcal{P}}(I)$, it holds that $\deg_{out}(n_{i_1,i_2,\ldots,i_k}) = |S_{i_1,i_2,\ldots,i_k}| = h \cdot \lambda_{i_1,i_2,\ldots,i_k}$. We have that several combinatorial problems, including those listed in Table 1, as we prove later, are simple problems.

The next lemma shows how classical simple dynamic programming algorithms can be quantumly enhanced to reduce their time complexity, while maintaining the same storage requirements as in the classical setting. Such an improvement applies to problems whose recursive formulation satisfies Equation (1). Table 1 provides an overview of the speedups obtainable via Theorem 4 for many well-known problems.

▶ **Theorem 4.** *Let $\mathcal{P}$ be a simple combinatorial problem, let I be an instance of $\mathcal{P}$, and let n be the size of I. Suppose that each of the values of the entries of D can be represented using w bits with $w \in O(polylog(n))$. Consider a simple classical dynamic programming algorithm $\mathcal{A}$ that solves $\mathcal{P}$ for I by computing each entry $D[i_i][i_2]\ldots[i_k]$ of a table D of size $d_1 \times d_2 \times \ldots \times d_k$, where the values d_i depend on I, using a recurrence formula of the same form as Equation* (1)*:*

$$D[i_1][i_2]\dots[i_k] = \mathrm{op}_{\mathcal{X}\in\mathcal{C}_{i_1,i_2,\dots,i_k}} f_{\mathcal{P}}(i_1,i_2,\dots,i_k,\mathcal{X}),$$

where $\mathrm{op}\in\{\min,\max,\mathrm{find},\mathrm{findAll}\}$. *Also, let* $T_{\mathcal{P}}$, T_{f_C} *and* T_γ *be the time needed to compute quantumly the functions* $f_{\mathcal{P}}$, f_C, *and* γ, *respectively. Finally, let* $T' = T_{f_C} + T_\gamma$, *let* M *be the maximum number of solutions of any subproblem of* $\mathcal{P}$ *for* I, *and let* δ *be the average degree of* $G_{\mathcal{P}}(I)$.

Then, there exists a quantum dynamic programming algorithm $Q_{\mathcal{A}}$ *that solves* $\mathcal{P}$ *for* I, *using QRAM, with the following time and space bounds:*

- *If* $\mathrm{op}\in\{\mathrm{find},\min,\max\}$, *then* $Q_{\mathcal{A}}$ *solves* $\mathcal{P}$ *for* I *using* $\tilde{O}(|V(G_{\mathcal{P}}(I))|(T'+T_{\mathcal{P}}\sqrt{\delta}))$ *time and* $\tilde{O}(|V(G_{\mathcal{P}}(I))|)$ *space.*
- *If* $\mathrm{op}=\mathrm{findAll}$, *then* $Q_{\mathcal{A}}$ *solves* $\mathcal{P}$ *for* I *using* $\tilde{O}(|V(G_{\mathcal{P}}(I))|(T'+T_{\mathcal{P}}\sqrt{\delta\cdot M}))$ *time and* $\tilde{O}(M|V(G_{\mathcal{P}}(I))|)$ *space.*

Proof. In the following, we describe the algorithm $Q_{\mathcal{A}}$ and prove its space and time complexity. We describe $Q_{\mathcal{A}}$ in terms of the three steps that implement the dynamic programming approach provided by $\mathcal{A}$.

Table Setup. As in the classical scenario, the algorithm starts by initializing a dynamic programming table QD of size $d_1\times d_2\times\ldots\times d_k$. Differently from the classical dynamic programming table D used by $\mathcal{A}$, however, the table QD is stored in QRAM. Suppose that each of the values of the entries of D (and thus of QD) can be represented using w bits. Let f_{init} be the classical function that, provided with the tuple $\langle i_1,i_2,\dots,i_k\rangle$ that addresses the entry $D[i_1][i_2]\dots[i_k]$, computes the initialization value of $D[i_1][i_2]\dots[i_k]$. To this aim, for each $\langle i_1,i_2,\dots,i_k\rangle\in[d_1]\times[d_2]\times\ldots\times[d_k]$, we provide the gate U_{init} with the address register $|i_1i_2\dots i_k\rangle_a$ and with a data register storing a w-bit zero string $|0^w\rangle$. The application of U_{init} to these registers results in the state $|i_1i_2\dots i_k\rangle_a|f_{init}(\langle i_1,i_2,\dots,i_k\rangle)\rangle_d$, which forms the input for a `QRAM write` operation that initializes $QD[i_1][i_2]\dots[i_k]$.

Table Update. We process the tuples $\langle i_1,i_2,\dots,i_k\rangle\in[d_1]\times[d_2]\times\ldots\times[d_k]$ in lexicographic order. For each tuple $\langle i_1,i_2,\dots,i_k\rangle$, the algorithm invokes either the quantum subroutine `DP`$(i_1,i_2,\dots,i_k,f_{\mathcal{P}},op)$ or `DP`$(i_1,i_2,\dots,i_k,f_{\mathcal{P}},op,\lessdot)$, based on whether $\mathrm{op}\in\{\mathrm{find},\mathrm{findAll}\}$ or $\mathrm{op}\in\{\min,\max\}$, respectively. This allows us to compute $\mathrm{op}_{\mathcal{X}\in\mathcal{C}_{i_1,i_2,\dots,i_k}} f_{\mathcal{P}}(i_1,i_2,\dots,i_k,\mathcal{X})$, which is stored in an output register $|d_{i_1,i_2,\dots,i_k}\rangle_o$ correlated with the address register $|i_1i_2\dots i_k\rangle_a$. Finally, the state $|i_1i_2\dots i_k\rangle_a|d_{i_1,i_2,\dots,i_k}\rangle_o$ forms the input for a `QRAM write` operation that updates $QD[i_1][i_2]\dots[i_k]$.

Solution Retrieval. Let $\langle i_1^*,i_2^*,\dots,i_k^*\rangle$ be the entry of D whose value determines the solution to $\mathcal{P}$. Then, a solution for $\mathcal{P}$ can be obtained by reading the entry $QD[i_1^*][i_2^*]\dots[i_k^*]$.

Next, we analyze the space complexity of $Q_{\mathcal{A}}$. As in its classic counterpart, the space complexity of $Q_{\mathcal{A}}$ is asymptotically bounded by the storage requirement of the dynamic programming table, which can be upper bounded by multiplying the number of entries of QD, the number w of bits to represent each of the values of the entries of QD (i.e., the solutions to the subproblems of $\mathcal{P}$), and the maximum number M of solutions of any subproblem of $\mathcal{P}$ for I. Also, recall that the entries of QD are in one-to-one correspondence with the nodes of $G_{\mathcal{P}}(I)$. Therefore, the space complexity of $Q_{\mathcal{A}}$ is in $O(w(|V(G_{\mathcal{P}}(I))|)$, if $\mathrm{op}=\{\mathrm{find},\min,\max\}$, and in $O((w\cdot M)|V(G_{\mathcal{P}}(I))|)$, if $\mathrm{op}=\mathrm{findAll}$.

Finally, we analyze the time complexity of $Q_{\mathcal{A}}$. Obviously, the time needed to compute function f_{init} is in $O(T_{\mathcal{P}})$. Therefore, the time complexity $\mathrm{Time}(I)$ of $Q_{\mathcal{A}}$ on input I is asymptotically bounded by the time required to execute the **Table Update** step. We now

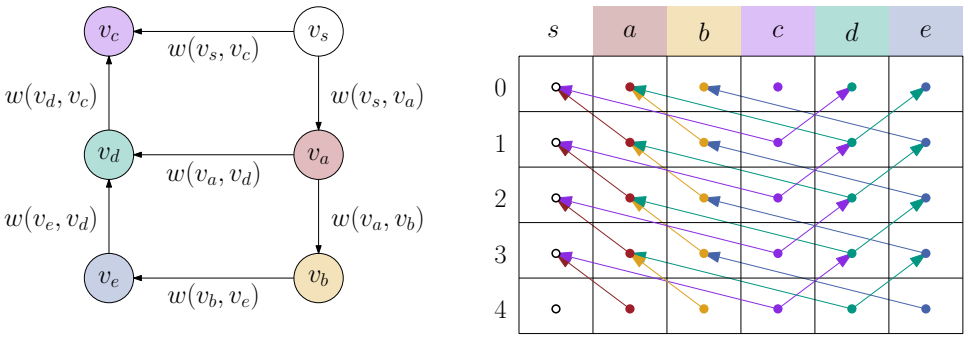
give the corresponding bound. For simplicity, we assume first that op $\in \{\text{find}, \min, \max\}$. Let $n = |V(G_{\mathcal{P}}(I))|$ and recall that δ denotes the average degree of $G_{\mathcal{P}}(I)$. In order to upper bound $\text{Time}(I)$, we proceed as follows. For $i = 1, \ldots, \log\log n$, let V_i be the subset of $V(G_{\mathcal{P}}(I))$ whose degree is between $2^{2^{i-1}}\delta$ and $2^{2^i}\delta$, and let V_0 be the subset of $V(G_{\mathcal{P}}(I))$ whose degree is less than 2δ. Recall that, by Theorem 3, the quantum subroutine DP determines a value of $f_{\mathcal{P}}$ for an entry $D[i_1][i_2]\ldots[i_k]$ in $\tilde{O}(T'_{i_1,i_2,\ldots,i_k} + T_{\mathcal{P}}\sqrt{\lambda_{i_1,i_2,\ldots,i_k}})$ time, that $T'_{i_1,i_2,\ldots,i_k} \leq T'$, and that, for each node $n_{i_1,i_2,\ldots,i_k}$ of $G_{\mathcal{P}}(I)$, it holds that $\lambda_{i_1,i_2,\ldots,i_k} \leq \deg_{out}(n_{i_1,i_2,\ldots,i_k})$ since $\mathcal{P}$ is simple. Therefore, we have:

$$\begin{aligned}
\text{Time}(I) &= \sum_{n_{i_1,i_2,\ldots,i_k} \in V(G_{\mathcal{P}}(I))} \tilde{O}\Big(T'_{i_1,i_2,\ldots,i_k} + T_{\mathcal{P}}\sqrt{\lambda_{i_1,i_2,\ldots,i_k}}\Big) \leq \\
&\leq \sum_{n_{i_1,i_2,\ldots,i_k} \in V(G_{\mathcal{P}}(I))} \tilde{O}\Big(T' + T_{\mathcal{P}}\sqrt{\deg_{out}(n_{i_1,i_2,\ldots,i_k})}\Big) \leq \\
&\leq \tilde{O}\Big(|V_0|(T' + T_{\mathcal{P}}\sqrt{2\delta})\Big) + \sum_{i=1}^{\log\log n} \tilde{O}\Big(|V_i|(T' + T_{\mathcal{P}}\sqrt{2^{2^i}\delta})\Big) = \\
&= \tilde{O}(|V_0|T') + \tilde{O}(|V_0|T_{\mathcal{P}}\sqrt{\delta}) + \sum_{i=1}^{\log\log n} \tilde{O}(|V_i|T') + (T_{\mathcal{P}} \cdot \sqrt{\delta}) \sum_{i=1}^{\log\log n} \tilde{O}(|V_i|2^{2^{i-1}}) \quad (4)
\end{aligned}$$

Next, we provide an upper bound for $|V_i|$. Clearly, there exist at most $2n$ nodes $n_{i_1,i_2,\ldots,i_k}$ of $G_{\mathcal{P}}(I)$ such that $\deg_{out}(n_{i_1,i_2,\ldots,i_k}) < 2\delta$ (that is, $|V_0| \leq 2n$). Also, there exist at most n nodes $n_{i_1,i_2,\ldots,i_k}$ of $G_{\mathcal{P}}(I)$ such that $2\delta \leq \deg_{out}(n_{i_1,i_2,\ldots,i_k}) < 4\delta$ (that is, $|V_1| \leq n$). Similarly, there exist at most $\frac{n}{2}$ nodes $n_{i_1,i_2,\ldots,i_k}$ of $G_{\mathcal{P}}(I)$ such that $4\delta \leq \deg_{out}(n_{i_1,i_2,\ldots,i_k}) < 16\delta$ (that is, $|V_2| \leq \frac{n}{2}$). More in general, for $i = 1, \ldots, \log\log n$, there exist at most $\frac{n}{2^{2^{i-1}-1}} = \frac{2n}{2^{2^{i-1}}}$ nodes of $G_{\mathcal{P}}(I)$ such that $2^{2^{i-1}}\delta \leq \deg_{out}(n_{i_1,i_2,\ldots,i_k}) < 2^{2^i}\delta$ (that is, $|V_i| \leq \frac{n}{2^{2^{i-1}-1}} = \frac{2n}{2^{2^{i-1}}}$). In Equation (4), we can then upper bound V_0 with $2n$, and $|V_i|$ with $\frac{2n}{2^{2^{i-1}}}$, if $i > 1$. Therefore, we have:

$$\begin{aligned}
\text{Time}(I) &= \tilde{O}(|V_0|T') + \tilde{O}(|V_0|T_{\mathcal{P}}\sqrt{\delta}) + \sum_{i=1}^{\log\log n} \tilde{O}(|V_i|T') + (T_{\mathcal{P}} \cdot \sqrt{\delta}) \sum_{i=1}^{\log\log n} \tilde{O}(|V_i|2^{2^{i-1}}) \leq \\
&\leq \tilde{O}(n \cdot T') + \tilde{O}(nT_{\mathcal{P}}\sqrt{\delta}) + \tilde{O}(n \cdot T') + (T_{\mathcal{P}} \cdot \sqrt{\delta}) \sum_{i=1}^{\log\log n} \tilde{O}(|V_i|2^{2^{i-1}}) \leq \\
&\leq \tilde{O}(n \cdot T') + \tilde{O}(nT_{\mathcal{P}}\sqrt{\delta}) + (T_{\mathcal{P}} \cdot \sqrt{\delta}) \sum_{i=1}^{\log\log n} \tilde{O}(\frac{2n}{2^{2^{i-1}}}2^{2^{i-1}}) = \\
&= \tilde{O}(n \cdot T') + \tilde{O}(nT_{\mathcal{P}}\sqrt{\delta}) + (T_{\mathcal{P}} \cdot \sqrt{\delta})\tilde{O}(n) \sum_{i=1}^{\log\log n} 2 \\
&= \tilde{O}\Big(n(T' + (T_{\mathcal{P}} \cdot \sqrt{\delta})(1 + 2\log\log n))\Big) = \tilde{O}\Big(n(T' + T_{\mathcal{P}} \cdot \sqrt{\delta})\Big)
\end{aligned} \quad (5)$$

Altogether, Equation (5) shows the desired bound for the running time of $Q_{\mathcal{A}}$ when op $\in \{\text{find}, \min, \max\}$. For the case when op = findAll the same analysis applies with $\sqrt{\delta}$ replaced by $\sqrt{\delta \cdot M}$ according to Theorem 3. This concludes the proof. ◀



(a) An instance $I = (G, w, s = v_s)$ of SSSP.

(b) The dependency graph $G_{\mathcal{P}}(I)$ of I and the dynamic programming table D for instance I. Each node of $G_{\mathcal{P}}(I)$ is placed inside the corresponding entry of D.

Figure 1 Illustrations for problem SSSP.

5 Applications of the Quantum Dynamic Programming Framework

In this section, we consider the Single-Source Shortest Paths (SSSP) problem and the Membership in Context-Free Language (MCFL) problem. Since both the celebrated Bellman-Ford algorithm for SSSP and Cocke-Younger-Kasami algorithm for MCFL are based on dynamic programming recurrences that respect Equation (1), they provide primary examples for computational problems amenable to a quantum speedup via Theorem 4.

In the following for the SSSP and the MCFL problems, and in the full version [16] for the remaining problems listed in Table 1, we demonstrate the applicability of Theorem 4 as follows. First, we present the corresponding classical dynamic programming algorithm. In particular, we describe (i) the **subproblems** whose solution is stored in the dynamic programming table, (ii) the **optimal substructure property** of the problem, and (iii) the **overlap** among subproblems. Then, if needed, we transform the recurrence relation of the dynamic programming algorithm in such a way that it matches the pattern of Equation (1), and we define the dependency set $S_{i_1,i_2,\ldots,i_k}$, the generating set $C_{i_1,i_2,\ldots,i_k}$, the dependency index h, and the function $f_{\mathcal{P}}$. Also, we bound the average degree of the dependency digraph $G_{\mathcal{P}}(I)$ in terms of the size of the instances. Further, we show that the dynamic programming algorithm is simple. Finally, we bound the time complexity of the functions $f_{\mathcal{P}}$, f_C, and γ. Altogether, this allows us to the establish our quantum speedups via Theorem 4.

5.1 Quantum Bellman-Ford for Single-Source Shortest Paths

Let $G = (V, E)$ be a weighted n-vertex m-edge digraph, where each arc $e = uv \in E$, directed from u to v, has an associated weight $w(e)$. Let $p = (u_0, u_1, \ldots, u_k)$ be a directed path from u_0 to u_k. The *weight* of p is $\sum_{(u_iu_{i+1})\in E(p)} w(u_iu_{i+1})$, and the *length* of p is $|E(p)| = k$. A *shortest path* from u to v is a minimum-weight directed path from u to v. The *shortest path distance* from u to v is the weight of a shortest path from u to v.

Next, we define the Single-Source Shortest Paths problem (see Figure 1a for the illustration of an instance of this problem).

Single-Source Shortest Paths (SSSP)

Input: A weighted digraph $G = (V, E)$, a function $w : E \to \mathbb{R}$, and a source vertex $s \in V$.
Output: The shortest-path distances between s and v, for all $v \in V$.

Subproblems. For all $i \in [n]$ and for all $v_j \in V$, a subproblem is defined as finding the smallest weight of a directed path, of length at most i, from the source vertex s to the vertex v_j. Since the maximum length of a shortest path is at most $n - 1$, we have n^2 subproblems.

Optimal substructure. Let D be a dynamic programming table of size $n \times n$, whose entries $D[i][j]$ store the smallest weight of a directed path, of length at most i, from s to v_j, for all $i \in [n]$ and for all $v_j \in V$. If the smallest-weight path p_{s,v_j}, of length at most i, from s to v_j uses uv_j as its last arc, then the subpath of p_{s,v_j} from s to u must be a smallest-weight path, of length at most $i - 1$, from s to u. Consider a smallest-weight path P, of length at most i, from s to v_j, whose weight is stored in $D[i][j]$:

- If P is of length at most $i - 1$, then $D[i][j] = D[i - 1][j]$;
- If P is of length (at most) i and its last edge is $v_k v_j$, then $D[i][j] = D[i - 1][k] + w(v_k v_j)$.

Overlapping subproblems. The value $D[i - 1][k]$ must be accessed to compute all entries $D[i][y]$, where $v_k v_y \in E$.

The above yields the following dynamic programming algorithm for SSSP, due to Bellman and Ford [8, 26].

DP 5.1 (SSSP)

The entries of D can be computes as follows:

Base case: if $i = 0$:

$$D[i][j] = \begin{cases} \infty, \text{ for all } v_j \neq s \\ 0, \text{ if } v_j = s \end{cases}$$

Recursive case: if $i < j$:

$$D[i][j] = \min(D[i - 1][j], \min_{v_k v_j \in E}(D[i - 1][k] + w(v_k v_j))) \tag{6}$$

Note that Equation (6) can be rewritten as:

$$D[i][j] = \min_{v_k v_j \in E}(\min(D[i - 1][j], D[i - 1][k] + w(v_k v_j))),$$

which matches the pattern of Equation (1). In particular, we have that: $S_{i,j} = \{(i - 1, k) : v_k v_j \in E\}$, $C_{i,j} = \{\{(i - 1, k)\} : v_k v_j \in E\}$, $h = 1$, and $f_{\mathcal{P}}(i, j, \{(i - 1, k)\}) = \min(D[i - 1][j], D[i - 1][k] + w(v_k v_j))$. Note that, SSSP is a simple problem, since each entry of $C_{i,j}$ stems from a distinct entry of $S_{i,j}$. Also observe that, the values in D are bounded by $W = (n - 1) \max_{v_i v_j \in E} w(v_i v_j)$. As in [31, 32, 37], we assume that the weights defined by w are in $O(n^c)$, where c is a constant. This implies that each of the entries of D can be represented using $\log W \in O(\log n)$ bits. The time $T_{\mathcal{P}}$, T_{f_C}, and T_γ needed for computing quantumly $f_{\mathcal{P}}$, f_C, and γ is then $O(\log W) = O(\log n)$, $O(\log n)$, and $O(\log n)$, respectively.

Thus, we have that the time $T' = T_{f_C} + T_\gamma$ (see the statement of Theorem 3) is in $O(\log n)$. Finally, we bound the average degree of $G_{\mathcal{P}}(I)$. Recall that an instance I of SSSP is a tuple $I = (G, w, s)$. Note that, $|V(G_{\mathcal{P}}(I))|$ is the number of subproblems, that is, n^2. The n nodes of $V(G_{\mathcal{P}}(I))$ corresponding to the subproblems $D[0][\cdot]$ have outdegree 0. Instead, by the definition of $C_{i,j}$, each node $n_{i,j}$ of $G_{\mathcal{P}}(I)$ corresponding to a subproblem $D[i][j]$ has outdegree equal to $\deg_{out}(v_j)$; refer to Figure 1b. Therefore, for any $i \in \{1, \ldots, n-1\}$, we have that $\sum_{v_j \in V} \deg_{out}(n_{i,j}) = m$. It follows that the average degree δ of $G_{\mathcal{P}}(I)$ is equal to $\frac{2m(n-1)}{n^2} = \frac{2m}{n} - \frac{2m}{n^2} < \frac{2m}{n}$. Therefore, by Theorem 4, there exists a quantum dynamic programming algorithm that solves SSSP for an n-vertex m-edge digraph, using QRAM, in $\tilde{O}(n\sqrt{nm})$ time using $O(n^2)$ space.

Finally, we compare the current-best classical time bound for the SSSP problem with our quantum bound. We remark that the running time of the Bellman-Ford algorithm is $O(n^3)$ time. The current-fastest classical algorithm for SSSP, due to Huang, Jin, and Quanrud [42], runs in $\tilde{O}(mn^{4/5})$ time. Thus, our quantum dynamic programming version of the Bellman-Ford algorithm improves upon [42] when $m \in \Omega(n^{\frac{7}{5}}) = \Omega(n^{1.4})$.

We remark that, as in the classical setting, by explicitly computing the dynamic programming table, our algorithm allows to detect a negative cycle in $O(m)$ time.

5.2 Quantum Cocke-Younger-Kasami for Membership in Context-Free Language

A *formal* grammar G is defined[38] a 4-tuple (V, Σ, R, S), where: **(i)** V is the finite set of non-terminal symbols; **(ii)** T is the finite set of terminal symbols; **(iii)** P is is the set of production rules; and **(iv)** $S \in V$ is the start symbol.

Let G be a *Context-Free Grammar* (CFG) in **Chomsky Normal Form (CNF)**.

A CFG is in CNF if all its production rules are of one of the following forms: **(1)** $A \to BC$ where A, B, C are non-terminal symbols (variables), i.e., a non-terminal symbol A can be replaced by two non-terminal symbols, **(2)** $A \to a$ where A is a non-terminal symbol and a is a terminal symbol, i.e., a non-terminal symbol A can be replaced by a single terminal symbol a.

Let $L(G)$ be the language composed of the strings that can be generated by the grammar G.

Next, we define the Membership in Context-Free Language problem.

Membership in Context-Free Language (MCFL)

Input: A constant-size context-free grammar G in CNF and a nonempty string $w = w_1 w_2 \ldots w_n$, where each w_i is a terminal symbol.

Output: A boolean value `True`, if the string w can be derived from the start non-terminal symbol S of the grammar G, i.e., $w \in L(G)$; `False`, otherwise.

Subproblems. A subproblem is defined as determing whether a consecutive substring of w can be derived from a non-terminal symbol A. Since a string of n symbols has $\binom{n}{2}$ consecutive substrings, we have $\frac{n(n+1)}{2}$ subproblems.

Optimal substructure. Let D be a dynamic programming table of size $n \times n$ (where the entries above the main diagonal are not used), whose entries $D[i][j]$ store the set of non-terminal symbols that can generate the substring $w_j w_{j+1} \ldots w_{j+i-1}$ of w, i.e., the substring of w of length i that starts at the j-th symbol of w. The set of non-terminal symbols in $D[i][j]$ is defines as follows:

- $D[i][j] = \{A : (A \to BC) \in G \wedge \exists k \in [1, i) : (B \in D[k][j]) \wedge (C \in D[i-k][j+k])\}$

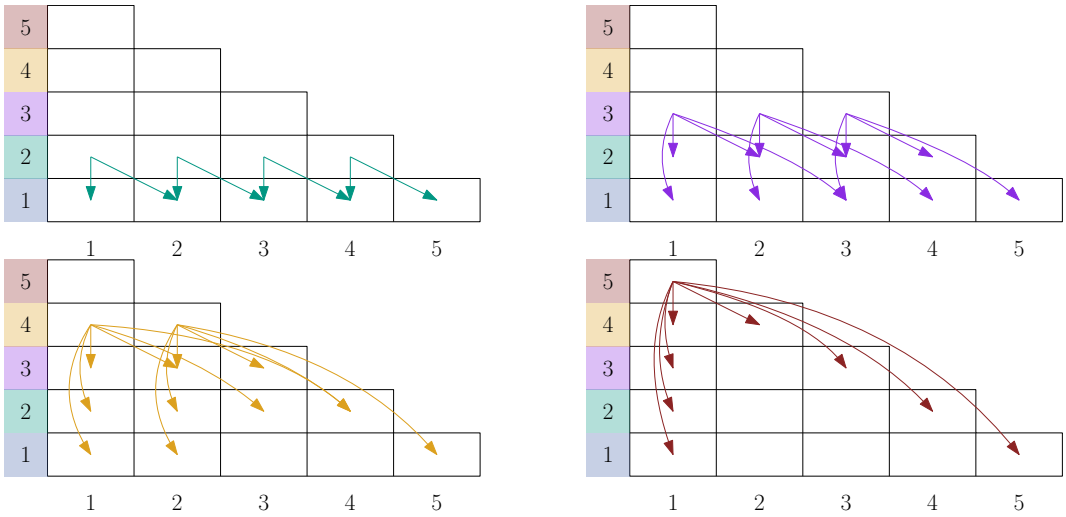


Figure 2 Graph $G_{\mathcal{P}}(I)$ for the dynamic programming algorithm that solves MCFL.

Overlapping subproblems. The value $D[k][j]$ must be accessed to compute all the entries of the form $D[u][j]$, with $u > k$; whereas the value $D[i-k][j+k]$ must be accessed to compute all the entries of the form $D[u][j+k]$, with $u > i-k$.

The above yields the following dynamic programming algorithm for MCFL, due to Cocke-Younger-Kasami [61, 44, 18].

DP 5.2 (MCFL)

The entries of D can be computes as follows:

BASE CASE: if $i = 1$:

$$D[i][j] = \{A \mid (A \to w_j) \in G\}$$

RECURSIVE CASE: if $i < j$:

$$D[i][j] = \text{findAll}_{1 \le k < i}\{A : (A \to BC \in G) \wedge (B \in D[k][j]) \wedge (C \in D[i-k][j+k])\} \tag{7}$$

Note that Equation (7) matches the pattern of Equation (1) of Theorem 4. Observe that, w can be generated from G if and only if $S \in D[n][1]$. In particular, we have that: $S_{i,j} = \{(k,j),(i-k,j+k) : 1 \le k < i\}$, $C_{i,j} = \{\{(k,j),(i-k,j+k)\} : 1 \le k < i\}$, $h = 2$, and $f_{\mathcal{P}}(i,j,\{(k,j),(i-k,j+k)\}) = \{A : (A \to BC \in G) \wedge (B \in D[k][j]) \wedge (C \in D[i-k][j+k])\}$. Note that, MCFL is a simple problem, since each entry of $C_{i,j}$ stems from a distinct entry of $S_{i,j}$. The time needed for computing quantumly $f_{\mathcal{P}}$, f_C, and γ is then $O(1)$, $O(\log n)$, and $O(\log n)$, respectively. Therefore, we have that $T' \in O(\log n)$.

Finally, we bound the average degree of $G_{\mathcal{P}}(I)$. Recall that an instance I of MCFL is a pair (G,w). Note that, $|V(G_{\mathcal{P}}(I))|$ is the number of subproblems defined by I, that is, $\frac{n(n+1)}{2}$. Note that, the n nodes of $V(G_{\mathcal{P}}(I))$ corresponding to the subproblems $D[1][j]$ have outdegree 0. Instead, by the definition of $C_{i,j}$, each node $n_{i,j}$ of $G_{\mathcal{P}}(I)$ corresponding to

a subproblem $D[i][j]$ has outdegree equal to $2(i-1)$; refer to Figure 2. Therefore, for any $i \in \{1,\ldots,n\}$, we have that $\sum_{j=1}^{n+1-i} \deg_{out}(n_{i,j}) = 2(n+1-i)(i-1)$. The total number of edges of $G_{\mathcal{P}}(I)$ is equal to

$$\sum_{i=1}^{n} 2(n+1-i)(i-1) \leq 2n \sum_{i=1}^{n} i = 2n\Big(\frac{n(n+1)}{2}\Big) = n^2(n+1)$$

It follows that the average degree δ of $G_{\mathcal{P}}(I)$ is upperbounded by $(n^2(n+1))/(\frac{n(n+1)}{2}) = 2n$. Therefore, by Theorem 4, there exists a quantum dynamic programming algorithm that solves MCFL for a constant-size context-free grammar G and a string w on length n, using QRAM, in $\tilde{O}(n^2\sqrt{n})$ time using $O(n^2)$ space.

Finally, we compare the current-best classical time bound for the MCFL problem with our quantum bound. We remark that the running time of the CYK algorithm is $O(n^3)$. Valiant demonstrated in 1975 that membership in context-free language recognition can be performed at least as efficiently as Boolean matrix multiplication [57]. Since Boolean matrix multiplication can be done in $O(n^{2.81})$ time using Strassen's algorithm [55], this implies an indirect $O(n^{2.81})$-time algorithm for testing membership in a context-free language. However, such an algorithm does not allow to directly retrieve the sequence of productions used to derive w. The current-fastest classical algorithm for matrix multiplication has a time complexity of $O(n^{2.371552})$ [58], building upon the Coppersmith-Winograd algorithm [19]. Observe that, the constant factors hidden by the Big O notation are alone so substantial that these fast matrix multiplication algorithms are impractical for matrix sizes manageable by current computers. Finally, we remark that our approach is methodologically simpler than relying upon matrix multiplication, as it directly accelerates an "easy" algorithm for context-free parsing. In fact, using matrix multiplication for parsing requires a more complex, three-step process: **(1)** transforming the parsing instance into a format suitable for matrix multiplication, **(2)** applying the sophisticated (and practically cumbersome) matrix multiplication algorithm described in [58], and **(3)** converting the output of the recognition process (provided by the matrix multiplication) into a parsing solution[6].

6 Conclusions and Open Problems

In this paper, we presented a framework that systematically extends classical dynamic programming algorithms that exhibit specific characteristics in the computation of their dynamic programming table, into accelerated quantum counterparts. These characteristics pertain to the structure of the *dependency set*, specifying the global set of entries involved in computing a table entry, and to the structure of the *generating set*, specifying the groups of entries simultaneously involved in providing a candidate value for a table entry.

By leveraging quantum search primitives, such as *find*, *findAll*, *min*, and *max*, and the QRAM, we transform dynamic programming recurrence relations into efficient quantum subroutines. Our approach lowers the computational cost of constructing each entry in the dynamic programming table, often achieving a significant speedup over the best-known classical algorithms. We demonstrate the versatility of this framework by applying it to several well-known problems, including Single-Source Shortest Paths.

This work establishes a foundation for developing more efficient quantum algorithms for a class of dynamic programming problems, paving the way for several research directions. A natural extension is to investigate the applicability of our framework to dynamic programming

[6] Note that Ruzzo showed parsing is slightly harder than recognition by a logarithmic factor [52].

algorithms that do not have the above characteristics. It is also worth investigating whether similar quantum speedups can be achieved for algorithms that iteratively refine estimates of the optimal solution, such as the Floyd-Warshall algorithm. Finally, a key challenge is optimizing the space complexity of quantum dynamic programming algorithms. Due to the limitations of quantum memory, it is crucial to develop strategies that avoid storing the entire dynamic programming table. This may involve computing entries on-the-fly, leveraging quantum speedups for efficient on-demand computation, and minimizing the number of entries stored in superposition.

References

1 Scott Aaronson. Introduction to quantum information science lecture notes, April 2019. URL: https://www.scottaaronson.com/qclec.pdf.

2 Scott Aaronson, Nai-Hui Chia, Han-Hsuan Lin, Chunhao Wang, and Ruizhe Zhang. On the quantum complexity of closest pair and related problems. In Shubhangi Saraf, editor, *35th Computational Complexity Conference, CCC 2020*, volume 169 of *LIPIcs*, pages 16:1–16:43. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.CCC.2020.16.

3 Andris Ambainis. Quantum search algorithms. *SIGACT News*, 35(2):22–35, June 2004. doi:10.1145/992287.992296.

4 Andris Ambainis, Kaspars Balodis, Janis Iraids, Martins Kokainis, Krisjanis Prusis, and Jevgenijs Vihrovs. Quantum speedups for exponential-time dynamic programming algorithms. In Timothy M. Chan, editor, *SODA 2019*, pages 1783–1793. SIAM, 2019. doi:10.1137/1.9781611975482.107.

5 Andris Ambainis and Nikita Larka. Quantum algorithms for computational geometry problems. In Steven T. Flammia, editor, *15th Conference on the Theory of Quantum Computation, Communication and Cryptography, TQC 2020*, volume 158 of *LIPIcs*, pages 9:1–9:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.TQC.2020.9.

6 Andris Ambainis and Ilja Repko. Quantum algorithm for the domatic number problem. *Balt. J. Mod. Comput.*, 12(3), 2024. doi:10.22364/BJMC.2024.12.3.03.

7 Vladimirs Andrejevs, Aleksandrs Belovs, and Jevgenijs Vihrovs. Quantum algorithms for hopcroft's problem. In Rastislav Královic and Antonín Kucera, editors, *49th International Symposium on Mathematical Foundations of Computer Science, MFCS 2024*, volume 306 of *LIPIcs*, pages 9:1–9:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.MFCS.2024.9.

8 Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958. URL: http://www.jstor.org/stable/43634538.

9 Richard Bellman. On the approximation of curves by line segments using dynamic programming. *Commun. ACM*, 4(6):284, 1961. doi:10.1145/366573.366611.

10 Marc Benkert, Herman J. Haverkort, Moritz Kroll, and Martin Nöllenburg. Algorithms for multi-criteria one-sided boundary labeling. In Seok-Hee Hong, Takao Nishizeki, and Wu Quan, editors, *Graph Drawing, 15th International Symposium, GD 2007*, volume 4875 of *Lecture Notes in Computer Science*, pages 243–254. Springer, 2007. doi:10.1007/978-3-540-77537-9_25.

11 Marc Benkert, Herman J. Haverkort, Moritz Kroll, and Martin Nöllenburg. Algorithms for multi-criteria boundary labeling. *J. Graph Algorithms Appl.*, 13(3):289–317, 2009. doi:10.7155/JGAA.00189.

12 Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum counting. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP'98*, volume 1443 of *LNCS*, pages 820–831. Springer, 1998. doi:10.1007/BFB0055105.

13 Susanna Caroppo, Giordano Da Lozzo, and Giuseppe Di Battista. Quantum algorithms for one-sided crossing minimization. In Stefan Felsner and Karsten Klein, editors, *32nd International Symposium on Graph Drawing and Network Visualization, GD 2024*, volume 320 of *LIPIcs*, pages 20:1–20:9. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.GD.2024.20.

14 Susanna Caroppo, Giordano Da Lozzo, and Giuseppe Di Battista. Quantum algorithms for one-sided crossing minimization. *Theoretical Computer Science*, 1052:115424, 2025. doi:10.1016/j.tcs.2025.115424.

15 Susanna Caroppo, Giordano Da Lozzo, and Giuseppe Di Battista. Quantum graph drawing. *J. Graph Algorithms Appl.*, 29(2):3–47, 2025. doi:10.7155/JGAA.V29I2.3039.

16 Susanna Caroppo, Giordano Da Lozzo, Giuseppe Di Battista, Michael T. Goodrich, and Martin Nöllenburg. Quantum speedups for polynomial-time dynamic programming algorithms, 2025. arXiv:2507.00823.

17 Susanna Caroppo, Giordano Da Lozzo, and Giuseppe Di Battista. Quantum graph drawing. In Ryuhei Uehara, Katsuhisa Yamanaka, and Hsu-Chun Yen, editors, *Proc. 18th International Conference and Workshops on Algorithms and Computation, WALCOM 2024*, volume 14549 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2024. doi:10.1007/978-981-97-0566-5_4.

18 John Cocke. *Programming languages and their compilers: Preliminary notes*. New York University, 1969.

19 Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990. doi:10.1016/S0747-7171(08)80013-2.

20 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: http://mitpress.mit.edu/books/introduction-algorithms.

21 Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

22 Christopher M. Dawson and Michael A. Nielsen. The solovay-kitaev algorithm. *Quantum Inf. Comput.*, 6(1):81–95, 2006. doi:10.26421/QIC6.1-6.

23 Christoph Dürr and Peter Høyer. A quantum algorithm for finding the minimum. *CoRR*, quant-ph/9607014, 1996. arXiv:quant-ph/9607014.

24 Jeff Erickson. *Algorithms*. Independently published, 2019. URL: http://jeffe.cs.illinois.edu/teaching/algorithms/.

25 Geeks for Geeks. Largest divisible subset in array. https://www.geeksforgeeks.org/largest-divisible-subset-array/, 2024. Accessed: (21/02/2025).

26 L. R. Ford. *Network Flow Theory*, pages 253–313. Springer New York, New York, NY, 1997. doi:10.1007/0-387-22633-8_9.

27 Shion Fukuzawa, Michael T. Goodrich, and Sandy Irani. Quantum Tutte embeddings. *CoRR*, abs/2307.08851, 2023. doi:10.48550/arXiv.2307.08851.

28 Shion Fukuzawa, Michael T. Goodrich, and Sandy Irani. Quantum Tutte embeddings. In Michael A. Bekos and Markus Chimani, editors, *31st International Symposium on Graph Drawing and Network Visualization, GD*, volume 14466 of *LNCS*, pages 241–243. Springer, 2023.

29 Shion Fukuzawa, Michael T. Goodrich, and Sandy Irani. Quantum combine and conquer and its applications to sublinear quantum convex hull and maxima set construction. In Oswin Aichholzer and Haitao Wang, editors, *41st International Symposium on Computational Geometry, SoCG 2025*, volume 332 of *LIPIcs*, pages 51:1–51:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.SOCG.2025.51.

30 Bartholomew Furrow. A panoply of quantum algorithms. *Quantum Inf. Comput.*, 8(8):834–859, 2008. doi:10.26421/QIC8.8-9-11.

31 Harold N. Gabow. Scaling algorithms for network problems. *J. Comput. Syst. Sci.*, 31(2):148–168, 1985. doi:10.1016/0022-0000(85)90039-X.

32 Harold N. Gabow and Robert Endre Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989. doi:10.1137/0218069.

33 Andreas Gemsa, Benjamin Niedermann, and Martin Nöllenburg. Trajectory-based dynamic map labeling. In Leizhen Cai, Siu-Wing Cheng, and Tak Wah Lam, editors, *ISAAC 2013*, volume 8283 of *LNCS*, pages 413–423. Springer, 2013. doi:10.1007/978-3-642-45030-3_39.

34 I. M. Georgescu, S. Ashhab, and Franco Nori. Quantum simulation. *Rev. Mod. Phys.*, 86:153–185, March 2014. doi:10.1103/RevModPhys.86.153.

35 Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Phys. Rev. Lett.*, 100:160501, April 2008. doi:10.1103/PhysRevLett.100.160501.

36 Adam Glos, Martins Kokainis, Ryuhei Mori, and Jevgenijs Vihrovs. Quantum speedups for dynamic programming on n-dimensional lattice graphs. In Filippo Bonchi and Simon J. Puglisi, editors, *MFCS 2021*, volume 202 of *LIPIcs*, pages 50:1–50:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.MFCS.2021.50.

37 Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM J. Comput.*, 24(3):494–504, 1995. doi:10.1137/S0097539792231179.

38 Raymond Greenlaw and James H. Hoover. *Fundamentals of the Theory of Computation: Principles and Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

39 Lov K. Grover. A fast quantum mechanical algorithm for database search. In Gary L. Miller, editor, *STOC 1996*, pages 212–219. ACM, 1996. doi:10.1145/237814.237866.

40 Yijie Han and Tadao Takaoka. An $O(n^3 \log\log n/\log^2 n)$ time algorithm for all pairs shortest paths. *J. Discrete Algorithms*, 38-41:9–19, 2016. doi:10.1016/J.JDA.2016.09.001.

41 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

42 Yufan Huang, Peter Jin, and Kent Quanrud. Faster single-source shortest paths with negative real weights via proper hop distance. In Yossi Azar and Debmalya Panigrahi, editors, *SODA 2025*, pages 5239–5244. SIAM, 2025. doi:10.1137/1.9781611978322.178.

43 Daniel Jurafsky and James H. Martin. *Speech and language processing - an introduction to natural language processing, computational linguistics, and speech recognition*. Prentice Hall series in artificial intelligence. Prentice Hall, 2000.

44 Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*, 1966.

45 Kamil Khadiev and Liliya Safina. Quantum algorithm for dynamic programming approach for dags. applications for zhegalkin polynomial evaluation and some problems on dags. In Ian McQuillan and Shinnosuke Seki, editors, *UCNC 2019*, volume 11493 of *LNCS*, pages 150–163. Springer, 2019. doi:10.1007/978-3-030-19311-9_13.

46 Alexei Y. Kitaev, A. H. Shen, and Mikhail N. Vyalyi. *Classical and Quantum Computation*, volume 47 of *Graduate studies in mathematics*. American Mathematical Society, 2002. URL: https://bookstore.ams.org/gsm-47/.

47 Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006.

48 Vladislavs Klevickis, Krisjanis Prusis, and Jevgenijs Vihrovs. Quantum speedups for treewidth. In François Le Gall and Tomoyuki Morimae, editors, *TQC 2022*, volume 232 of *LIPIcs*, pages 11:1–11:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.TQC.2022.11.

49 G.T. Klincsek. Minimal triangulations of polygonal domains. In Peter L. Hammer, editor, *Combinatorics 79*, volume 9 of *Annals of Discrete Mathematics*, pages 121–123. Elsevier, 1980. doi:10.1016/S0167-5060(08)70044-X.

50 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information (10th Anniversary edition)*. Cambridge University Press, 2016. URL: https://www.cambridge.org/de/academic/subjects/physics/quantum-physics-quantum-information-and-quantum-computation/quantum-computation-and-quantum-information-10th-anniversary-edition.

51 Koustubh Phalak, Avimita Chatterjee, and Swaroop Ghosh. Quantum random access memory for dummies. *Sensors*, 23(17):7462, 2023. doi:10.3390/S23177462.

52 Walter L. Ruzzo. On the complexity of general context-free language parsing and recognition (extended abstract). In Hermann A. Maurer, editor, *Automata, Languages and Programming, 6th Colloquium*, volume 71 of *LNCS*, pages 489–497. Springer, 1979. doi:10.1007/3-540-09510-1_39.

53 Kazuya Shimizu and Ryuhei Mori. Exponential-time quantum algorithms for graph coloring problems. *Algorithmica*, 84(12):3603–3621, 2022. doi:10.1007/S00453-022-00976-2.

54 Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Rev.*, 41(2):303–332, 1999. doi:10.1137/S0036144598347011.

55 Volker Strassen. Gaussian elimination is not optimal. *Matematika*, 14(3):127–128, 1970.

56 Vincent T'kindt, Federico Della Croce, and Mathieu Liedloff. Moderate exponential-time algorithms for scheduling problems. *Ann. Oper. Res.*, 343(2):753–783, 2024. doi:10.1007/S10479-024-06289-7.

57 Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, 1975. doi:10.1016/S0022-0000(75)80046-8.

58 Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In David P. Woodruff, editor, *SODA 2024*, pages 3792–3835. SIAM, 2024. doi:10.1137/1.9781611977912.134.

59 Gerhard J. Woeginger. Open problems around exact algorithms. *Discret. Appl. Math.*, 156(3):397–405, 2008. doi:10.1016/J.DAM.2007.03.023.

60 Shifan Xu, Connor T. Hann, Ben Foxman, Steven M. Girvin, and Yongshan Ding. Systems architecture for quantum random access memory. In *MICRO 2023*, pages 526–538. ACM, 2023. doi:10.1145/3613424.3614270.

61 Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Inf. Control.*, 10(2):189–208, 1967. doi:10.1016/S0019-9958(67)80007-X.

62 Mohammed Zidan, Abdel-Haleem Abdel-Aty, Ashraf Khalil, Mahmoud A. Abdel-Aty, and Hichem Eleuch. A novel efficient quantum random access memory. *IEEE Access*, 9:151775–151780, 2021. doi:10.1109/ACCESS.2021.3119588.