Quantum Speedups for Dynamic Programming on n-Dimensional Lattice Graphs

Adam Glos
Institute of Theoretical and Applied Informatics, Polish Academy of Sciences, Warsaw, Poland

Martins Kokainis
Centre for Quantum Computer Science, Faculty of Computing, University of Latvia, Riga, Latvia

Ryuhei Mori
School of Computing, Tokyo Institute of Technology, Japan

Jevgēnijs Vihrovs
Center for Quantum Computer Science, Faculty of Computing, University of Latvia, Riga, Latvia

Abstract

Motivated by the quantum speedup for dynamic programming on the Boolean hypercube by Ambainis et al. (2019), we investigate which graphs admit a similar quantum advantage. In this paper, we examine a generalization of the Boolean hypercube graph, the n-dimensional lattice graph $Q(D,n)$ with vertices in $\{0,1,\dots,D\}^n$. We study the complexity of the following problem: given a subgraph G of $Q(D,n)$ via query access to the edges, determine whether there is a path from 0^n to D^n. While the classical query complexity is $\widetilde{\Theta}((D+1)^n)$, we show a quantum algorithm with complexity $\widetilde{O}(T_D^n)$, where $T_D < D+1$. The first few values of T_D are $T_1 \approx 1.817$, $T_2 \approx 2.660$, $T_3 \approx 3.529$, $T_4 \approx 4.421$, $T_5 \approx 5.332$. We also prove that $T_D \geq \frac{D+1}{\mathrm{e}}$ (here, $\mathrm{e} \approx 2.718$ is the Euler's number), thus for general D, this algorithm does not provide, for example, a speedup, polynomial in the size of the lattice.

While the presented quantum algorithm is a natural generalization of the known quantum algorithm for $D=1$ by Ambainis et al., the analysis of complexity is rather complicated. For the precise analysis, we use the saddle-point method, which is a common tool in analytic combinatorics, but has not been widely used in this field.

We then show an implementation of this algorithm with time and space complexity $\mathrm{poly}(n)^{\log n} T_D^n$ in the QRAM model, and apply it to the Set Multicover problem. In this problem, m subsets of $[n]$ are given, and the task is to find the smallest number of these subsets that cover each element of $[n]$ at least D times. While the time complexity of the best known classical algorithm is $O(m(D+1)^n)$, the time complexity of our quantum algorithm is $\mathrm{poly}(m,n)^{\log n} T_D^n$.

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

Keywords and phrases Quantum query complexity, Dynamic programming, Lattice graphs

Digital Object Identifier 10.4230/LIPIcs.MFCS.2021.50

Related Version *Full Version*: `https://arxiv.org/abs/2104.14384`

Funding *Adam Glos*: Supported in part by National Science Center under grant agreement 2019/32/T/ ST6/00158 and 2019/33/B/ST6/02011.
Martins Kokainis: Supported by "QuantERA ERA-NET Cofund in Quantum Technologies implemented within the European Union's Horizon 2020 Programme" (QuantAlgo project).
Ryuhei Mori: Supported in part by JST PRESTO Grant Number JPMJPR1867 and JSPS KAKENHI Grant Numbers JP17K17711, JP18H04090, JP20H04138, and JP20H05966.
Jevgēnijs Vihrovs: Supported in part by the project "Quantum algorithms: from complexity theory to experiment" funded under ERDF programme 1.1.1.5.

Acknowledgements We would like to thank Krišjānis Prūsis for helpful discussions and comments. We also thank anonymous reviewers for helpful comments and suggestions on the presentation.

1 Introduction

Dynamic programming (DP) algorithms have been widely used to solve various NP-hard problems in exponential time. Bellman, Held and Karp showed how DP can be used to solve the Travelling Salesman Problem in $\widetilde{O}(2^n)$[1] time using DP [5, 22], which still remains the most efficient classical algorithm for this problem. Their technique can be used to solve a plethora of different problems [16, 7].

The DP approach of Bellman, Held and Karp solves the subproblems corresponding to subsets of an n-element set, sequentially in increasing order of the subset size. This typically results in an $\widetilde{\Theta}(2^n)$ time algorithm, as there are 2^n distinct subsets. What kind of speedups can we obtain for such algorithms using quantum computers?

It is natural to consider applying Grover's search, which is known to speed up some algorithms for NP-complete problems. For example, we can use it to search through the 2^n possible assignments to the SAT problem instance on n variables in $\widetilde{O}(\sqrt{2^n})$ time. However, it is not immediately clear how to apply it to the DP algorithm described above. Recently, Ambainis et al. showed a quantum algorithm that combines classical precalculation with recursive applications of Grover's search that solves such DP problems in $\widetilde{O}(1.817^n)$ time, assuming the QRAM model of computation [4].

In their work, the authors applied this result to obtain quantum speedups for the algorithms solving graph vertex ordering problems like Pathwidth and Sum Cut [7], and using a more involved analysis, for the Graph Bandwidth problem [12]. They also used similar ideas to provide speedups for the Travelling Salesman, Feedback Arc Set and Minimum Set Cover problems by combining the Divide & Conquer and DP techniques. Subsequently, these ideas have been used to construct quantum speedups for the Graph Coloring [29], Minimum Steiner Tree [26] and finding the optimal variable ordering for the binary decision diagrams (OBDDs) [30]. More surprisingly, [1] used the quantum speedup for the Minimum Set Cover to prove non-trivial conditional lower bounds for the k-SUM problem [1] (assuming the Set Cover Conjecture, which states that Minimum Set Cover cannot be solved classically in time $O((2-\delta)^n)$ for any $\delta > 0$).

The $\widetilde{O}(1.817^n)$ quantum speedup of Ambainis et al. for the aforementioned DP algorithm on the subsets of the n-element set examines the underlying transition graph, which can be seen as a directed n-dimensional Boolean hypercube, with edges connecting smaller weight vertices to larger weight vertices. A natural question arises, for what other graphs there exist quantum algorithms that achieve a speedup over the classical DP? In this work, we examine a generalization of the hypercube graph, the n-dimensional lattice graph with vertices in $\{0, 1, \ldots, D\}^n$.

While the classical DP for this graph has running time $\widetilde{\Theta}((D+1)^n)$, as it examines all vertices, we prove that there exists a quantum algorithm (in the QRAM model) that solves this problem in time and space $\mathrm{poly}(n)^{\log n} T_D^n$ for $T_D < D + 1$ (Theorems 4, 7). Our algorithm essentially is a generalization of the algorithm of Ambainis et al. We show the following running time for small values of D:

Table 1 The complexity of the quantum algorithm.

D	1	2	3	4	5	6
T_D	1.81692	2.65908	3.52836	4.42064	5.33149	6.25720

[1] $f(n) = \widetilde{O}(g(n))$ if $f(n) = O(\log^c(g(n))g(n))$ for some constant c.

A detailed summary of our numerical results is given in Section 5.3. Note that the case $D = 1$ corresponds to the hypercube, where we have the same algorithm and complexity as Ambainis et al. In our proofs, we extensively use the saddle point method from analytic combinatorics to estimate the asymptotic value of the combinatorial expressions arising from the complexity analysis.

It is interesting to compare our analysis and that of Ambainis et al. Their original algorithm is recursive, and solves instances of the problem of smaller size (on the subcubes of smaller dimensions). These instances are symmetrical, so the recursive calls can be analyzed identically, and the calculation of the complexity is relatively simple. However, this is not the case in our generalization for the n-dimensional lattice. To see this, consider, for example, the lattice $\{0,1,2\}^n$. In the recursive calls, our algorithm will examine sublattices $\{0,1\}^{n_1} \times \{0,1,2\}^{n_2}$ with fixed maximum vertex weight $w = n_1 + 2n_2$. The first obstacle is that now there are many different cases (n_1, n_2) to analyze. The second obstacle is that recursively we have to solve the problem for a lattice $\{0,1\}^{n_1} \times \{0,1,2\}^{n_2}$: now it becomes difficult to describe and analyze the sublattices examined in the recursion of depth at least 2. To solve these issues, we first make an ansatz that the exponential complexity of the algorithm on the lattice $\times_{d=1}^{D}\{0,1,\dots,d\}^{n_d}$ can be expressed as $T_1^{n_1}T_2^{n_2}\cdots T_D^{n_D}$, for some positive constants T_d. Then we make use of the saddle point method to find such optimal constants (that minimize T_D), and also prove that the ansatz is correct. Our analysis provides exactly the same $\widetilde{O}(1.816...^n)$ complexity for the hypercube algorithm as by Ambainis et al.

Next, we also prove a lower bound on the query complexity of the algorithm for general D. Our motivation is to check whether our algorithm, for example, could achieve complexity $\widetilde{O}((D+1)^{cn})$ for large D for some $c < 1$. We prove that this is not the case: more specifically, for any D, the algorithm performs at least $\widetilde{\Omega}\Big(\big(\frac{D+1}{\mathrm{e}}\big)^n\Big)$ queries (Theorem 5), where $\mathrm{e} = 2.71828\dots$ is the Euler's number.

As an example application, we apply our algorithm to the Set Multicover problem (SMC), which is a generalization of the Set Cover problem. In this problem, the input consists of m subsets of the n-element set, and the task is to calculate the smallest number of these subsets that together cover each element at least D times, possibly with overlap and repetition. While the best known classical algorithm has running time $O(m(D+1)^n)$ [27, 24], our quantum algorithm has running time $\mathrm{poly}(m,n)^{\log n}T_D^n$, improving the exponential complexity (Theorem 8).

The paper is organized as follows. In Section 2, we formally introduce the n-dimensional lattice graph and some of the notation used in the paper. In Section 3, we define the generic query problem that models the examined DP. In Section 4, we describe our quantum algorithm. In Section 5, we establish the query complexity of this algorithm and prove the aforementioned lower bound. In Section 6, we discuss the implementation of this algorithm and establish its time complexity. Finally, in Section 7, we show how to apply our algorithm to SMC, and discuss other related problems.

2 Preliminaries

The n-dimensional lattice graph is defined as follows. The vertex set is given by $\{0,1,\dots,D\}^n$, and the edge set consists of directed pairs of two vertices u and v such that $v_i = u_i + 1$ for exactly one i, and $u_j = v_j$ for $j \neq i$. We denote this graph by $Q(D,n)$. Alternatively, this graph can be seen as the Cartesian product of n paths on $D+1$ vertices. The case $D = 1$ is known as the Boolean hypercube and is usually denoted by Q_n.

We define the *weight* of a vertex $x \in V$ as the sum of its coordinates $|x| \coloneqq \sum_{i=1}^{n} x_i$. Denote $x \leq y$ iff for all $i \in [n]$, $x_i \leq y_i$ holds. If additionally $x \neq y$, denote such relation by $x < y$.

Throughout the paper we use the standard notation $[n] \coloneqq \{1, \ldots, n\}$. In Section 7.1, we use notation for the superset $2^{[n]} \coloneqq \{S \mid S \subseteq [n]\}$ and for the characteristic vector $\chi(S) \in \{0,1\}^n$ of a set $S \in [n]$ defined as $\chi(S)_i = 1$ iff $i \in S$, and 0 otherwise.

We write $f(n) = \mathrm{poly}(n)$ to denote that $f(n) = O(n^c)$ for some constant c. We also write $f(n,m) = \mathrm{poly}(n,m)$ to denote that $f(n,m) = O(n^c m^d)$ for some constants c and d.

For a multivariable polynomial $p(x_1, \ldots, x_m)$, we denote by $[x_1^{c_1} \cdots x_m^{c_m}]p(x_1, \ldots, x_m)$ its coefficient at the multinomial $x_1^{c_1} \cdots x_m^{c_m}$.

3 Path in the hyperlattice

We formulate our generic problem as follows. The input to the problem is a subgraph G of $Q(D,n)$. The problem is to determine whether there is a path from 0^n to D^n in G. We examine this as a query problem: a single query determines whether an edge (u,v) is present in G or not.

Classically, we can solve this problem using a dynamic programming algorithm that computes the value $\mathrm{dp}(v)$ recursively for all v, which is defined as 1 if there is a path from 0^n to v, and 0 otherwise. It is calculated by the Bellman, Held and Karp style recurrence [5, 22]:

$$\mathrm{dp}(v) = \bigvee_{(u,v)\in E} \{\mathrm{dp}(u) \wedge ((u,v) \in G)\}, \qquad \mathrm{dp}(0^n) = 1.$$

The query complexity of this algorithm is $O(n(D+1)^n)$. From this moment we refer to this as the *classical dynamic programming algorithm*.

The query complexity is also lower bounded by $\widetilde{\Omega}((D+1)^n)$. Consider the sets of edges E_W connecting the vertices with weights W and $W+1$,

$$E_W \coloneqq \{(u,v) \mid (u,v) \in Q(D,n), |u| = W, |v| = W+1\}.$$

Since the total number of edges is equal to $(D+1)^{n-1}Dn$, there is such a W that $|E_W| \geq (D+1)^{n-1}Dn/Dn = (D+1)^{n-1}$ (in fact, one can prove that the largest size is achieved for $W = \lfloor nD/2 \rfloor$ [13], but it is not necessary for this argument). Any such E_W is a cut of H_D, hence any path from 0^n to D^n passes through E_W. Examine all G that contain exactly one edge from E_W, and all other edges. Also examine the graph that contains no edges from E_W, and all other edges. In the first case, any such graph contains a desired path, and in the second case there is no such path. To distinguish these cases, one must solve the OR problem on $|E_W|$ variables. Classically, $\Omega(|E_W|)$ queries are needed (see, for example, [8]). Hence, the classical (deterministic and randomized) query complexity of this problem is $\widetilde{\Theta}((D+1)^n)$. This also implies $\widetilde{\Omega}(\sqrt{(D+1)^n})$ quantum lower bound for this problem [6].

4 The quantum algorithm

Our algorithm closely follows the ideas of [4]. We will use the well-known generalization of Grover's search:

▶ **Theorem 1** (Variable time quantum search (VTS), Theorem 3 in [3]). *Let $\mathcal{A}_1$, …, $\mathcal{A}_N$ be quantum algorithms that compute a function $f : [N] \to \{0,1\}$ and have query complexities t_1, …, t_N, respectively, which are known beforehand. Suppose that for each $\mathcal{A}_i$, if $f(i) = 0$, then $A_i = 0$ with certainty, and if $f(i) = 1$, then $A_i = 1$ with constant success probability. Then there exists a quantum algorithm with constant success probability that checks whether $f(i) = 1$ for at least one i and has query complexity $O\left(\sqrt{t_1^2 + \ldots + t_N^2}\right)$. Moreover, if $f(i) = 0$ for all $i \in [N]$, then the algorithm outputs 0 with certainty.*

Even though Ambainis formulates the main theorem for zero-error inputs, the statement above follows from the construction of the algorithm.

Now we describe our algorithm. We solve a more general problem: suppose $s, t \in \{0, 1, \ldots, D\}^n$ are such that $s < t$ and we are given a subgraph of the n-dimensional lattice

$$\bigtimes_{i=1}^{n} \{s_i, \ldots, t_i\},$$

and the task is to determine whether there is path from s to t. We need this generalized problem because our algorithm is recursive and is called for sublattices.

Define $d_i \coloneqq t_i - s_i$. Let n_d be the number of indices $i \in [n]$ such that $d_i = d$. Note that the minimum and maximum weights of the vertices of this lattice are $|s|$ and $|t|$, respectively.

We call a set of vertices with fixed total weight a *layer*. The algorithm will operate with K layers (numbered 1 to K), with the k-th having weight $|s|+W_k$, where $W_k \coloneqq \left\lfloor \sum_{d=1}^{D} \alpha_{k,d} d n_d \right\rfloor$. Denote the set of vertices in this layer by

$$\mathcal{L}_k \coloneqq \{v \mid |v| = |s| + W_k\}.$$

Here, $\alpha_{k,d} \in (0, 1/2)$ are constant parameters that have to be determined before we run the algorithm. The choice of $\alpha_{k,d}$ does not depend on the input to the algorithm, similarly as it was in [4]. For each $k \in [K]$ and $d \in [D]$, we require that $\alpha_{k,d} < \alpha_{k+1,d}$. In addition to the K layers defined in this way, we also consider the $(K+1)$-th layer $\mathcal{L}_{K+1}$, which is the set of vertices with weight $|s| + W_{K+1}$, where $W_{K+1} \coloneqq \left\lfloor \frac{|t|-|s|}{2} \right\rfloor$. We can see that the weights $W_1, \ldots, W_{K+1}$ defined in this way are non-decreasing.

The informal description of the algorithm (Path) is as follows. First, we use the classical dynamic programming to calculate which vertices v with weight $|v| \leq |s| + W_1$ are reachable from s. Then, we store all of these answers in memory. Symmetrically, we also calculate from which vertices v with weight $|v| \geq |t| - W_1$ we can reach t, and also store this in memory. We refer to these steps as the classical precalculation part.

Next, we use VTS to search for a vertex $v^{(K+1)}$ in the layer $\mathcal{L}_{K+1}$ such that there is path from s to $v^{(K+1)}$ and from v to t. The LayerPath function is then used to detect whether there is a path from s to $v^{(K+1)}$. First, we use VTS to search for a vertex $v^{(K)} \in \mathcal{L}_K$ such that: (1) there exists a path from $v^{(K)}$ to $v^{(K+1)}$; (2) there exists a path from s to $v^{(K)}$. The first condition we can check using Path recursively for the lattice bounded by the vertices $v^{(K)}$ and $v^{(K+1)}$. The second condition is checked recursively using LayerPath in a similar fashion. Finally, for the vertex $v^{(1)} \in \mathcal{L}_1$, the LayerPath will need to check whether there is a path from s to $v^{(1)}$: this can be then simply read out from the memory, using the results of the precalculation part. We then similarly find whether t is reachable from $v^{(K)}$.

5 Query complexity

For simplicity, let us examine the lattice

$$\bigtimes_{i=1}^{n} \{0, \ldots, t_i - s_i\},$$

as the analysis is identical.

Let the number of positions with maximum coordinate value d be n_d. We make an ansatz that the exponential complexity can be expressed as

$$T(n_1, \ldots, n_D) \coloneqq T_1^{n_1} T_2^{n_2} \cdot \ldots \cdot T_D^{n_D}$$

Algorithm 1 The quantum algorithm for detecting a path in the hyperlattice.

PATH$(s,\ t)$:

1. Calculate $n_1, \ldots, n_D$, and $W_1, \ldots, W_{K+1}$. If $W_k = W_{k+1}$ for some k, determine whether there exists a path from s to t using classical dynamic programming and return.
2. Otherwise, first perform the precalculation step. Let $\mathrm{dp}(v)$ be 1 iff there is a path from s to v. Calculate $\mathrm{dp}(v)$ for all vertices v such that $|v| \le |s| + W_1$ using classical dynamic programming. Store the values of $\mathrm{dp}(v)$ for all vertices with $|v| = |s| + W_1$.
   Let $\mathrm{dp}'(v)$ be 1 iff there is a path from v to t. Symmetrically, we also calculate $\mathrm{dp}'(v)$ for all vertices with $|v| = |t| - W_1$.
3. Define the function LAYERPATH(k, v) to be 1 iff there is a path from s to v such that $v \in \mathcal{L}_k$. Implement this function recursively as follows.
   - LAYERPATH$(1, v)$ is read out from the stored values.
   - For $k > 1$, run VTS over the vertices $u \in \mathcal{L}_{k-1}$ such that $u < v$. The required value is equal to
   $$\textsc{LayerPath}(k, v) = \bigvee_u \{\textsc{LayerPath}(k-1, u) \wedge \textsc{Path}(u, v)\}.$$
4. Similarly define and implement the function LAYERPATH$'(k, v)$, which denotes the existence of a path from v to t such that $v \in \mathcal{L}'_k$ (where $\mathcal{L}'_k$ is the layer with weight $|t| - W_k$). To find the final answer, run VTS over the vertices in the middle layer $v \in \mathcal{L}_{K+1}$ and calculate
   $$\bigvee_v \big\{\textsc{LayerPath}(K+1, v) \wedge \textsc{LayerPath}'(K+1, v)\big\}.$$

for some values $T_1, T_2, \ldots, T_D > 1$ (we also can include n_0 and T_0, however, $T_0 = 1$ always and doesn't affect the complexity). We prove it by constructing generating polynomials for the precalculation and quantum search steps, and then approximating the required coefficients asymptotically. We use the saddle point method that is frequently used for such estimation, specifically the theorems developed in [9].

5.1 Generating polynomials

First we estimate the number of edges of the hyperlattice queried in the precalculation step. The algorithm queries edges incoming to the vertices of weight at most W_1, and each vertex can have at most n incoming edges. The size of any layer with weight less than W_1 is at most the size of the layer with weight exactly W_1, as the size of the layers is non-decreasing until weight W_{K+1} [13]. Therefore, the number of queries during the precalculation is at most $n \cdot W_1 \cdot |\mathcal{L}_1| \le n^2 D |\mathcal{L}_1|$, as $W_1 \le nD$. Since we are interested in the exponential complexity, we can omit n and D, thus the exponential query complexity of the precalculation is given by $|\mathcal{L}_1|$.

Now let $P_d(x) \coloneqq \sum_{i=0}^{d} x^i$. The number of vertices of weight W_1 can be written as the coefficient at x^{W_1} of the generating polynomial

$$P(x) \coloneqq \prod_{d=0}^{D} P_d(x)^{n_d}.$$

Indeed, each $P_d(x)$ in the product corresponds to a single position $i \in [n]$ with maximum value d and the power of x in that factor represents the coordinate of the vertex in this position. Therefore, the total power that x is raised to is equal to the total weight of the vertex, and coefficient at x^{W_1} is equal to the number of vertices with weight W_1. Since the total query complexity of the algorithm is lower bounded by this coefficient, we have

$$T(n_1, \dots, n_D) \geq \left[x^{W_1}\right] P(x). \tag{1}$$

Similarly, we construct polynomials for the LayerPath calls. Consider the total complexity of calling LayerPath recursively until some level $1 \leq k \leq K$ and then calling Path for a sublattice between levels $\mathcal{L}_k$ and $\mathcal{L}_{k+1}$. Define the variables for the vertices chosen by the algorithm at level i (where $k \leq i \leq K+1$) by $v^{(i)}$. The Path call is performed on a sublattice between vertices $v^{(k)}$ and $v^{(k+1)}$, see Fig. 1.

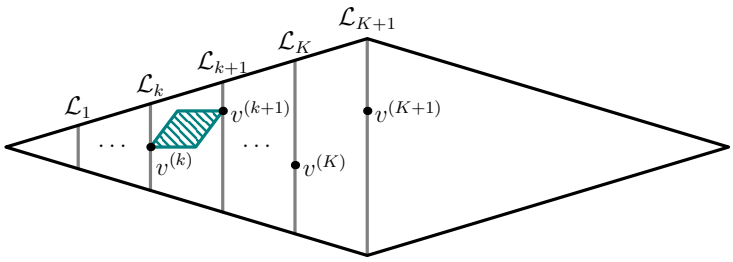

Figure 1 The choice of the vertices $v^{(i)}$ and the application of Path on the sublattice.

Define

$$S_{k,d}(x_{k,k}, \dots, x_{k,K+1}) \coloneqq \sum_{i=0}^{d} T_i^2 \cdot \sum_{\substack{p_k, \dots, p_{K+1} \in [0,d] \\ p_{k+1} \leq \dots \leq p_{K+1} \\ p_{k+1} - p_k = i}} \prod_{j=k}^{K+1} x_{k,j}^{p_j}.$$

Again, this corresponds to a single coordinate. The variable $x_{k,j}$ corresponds to the vertex $v^{(j)}$ and the power p_j corresponds to the value of $v^{(j)}$ in that coordinate.

Examine the following multivariate polynomial:

$$S_k(x_{k,k}, \dots, x_{k,K+1}) \coloneqq \prod_{d=0}^{D} S_{k,d}^{n_d}(x_{k,k}, \dots, x_{k,K+1}).$$

We claim that the coefficient

$$\left[x_{k,k}^{W_k} \cdots x_{k,K+1}^{W_{K+1}}\right] S_k(x_{k,k}, \dots, x_{k,K+1})$$

is the required total complexity squared.

First of all, note that the value of this coefficient is the sum of t^2, where t is the variable for the running time of Path between $v^{(k)}$ and $v^{(k+1)}$, for all choices of vertices $v^{(k)}$, $v^{(k+1)}$, …, $v^{(K+1)}$. Indeed, the powers p_j encode the values of coordinates of $v^{(j)}$, and a factor of T_i^2 is present for each multinomial that has $p_{k+1} - p_k = i$ (that is, $v_l^{(k+1)} - v_l^{(k)} = i$ for the corresponding position l).

Then, we need to show that the sum of t^2 equals the examined running time squared. Note that the choice of each vertex $v^{(j)}$ is performed using VTS. In general, if we perform VTS on the algorithms with running times s_1, …, s_N, then the total squared running time

is equal to $s_1^2 + \ldots + s_N^2$ by Theorem 1. By repeating this argument in our case inductively at the choice of each vertex $v^{(j)}$, we obtain that the final squared running time indeed is the sum of all t^2.

Therefore, the square of the total running time of the algorithm is lower bounded by

$$T(n_1, \ldots, n_D)^2 \geq \left[x_{k,k}^{W_k} \cdots x_{k,K+1}^{W_{K+1}}\right] S_k(x_{k,k}, \ldots, x_{k,K+1}). \tag{2}$$

Together the inequalities (1) and (2) allow us to estimate T. The total time complexity of the quantum algorithm is twice the sum of the coefficients given in Eq. (1) and (2) for all $k \in [K]$ (twice because of the calls to LayerPath and its symmetric counterpart LayerPath′). This is upper bounded by $2K$ times the maximum of these coefficients. Since $2K$ is a constant, and there are $O(\log n)$ levels of recursion (see Appendix A), in total this contributes only $(2K)^{O(\log n)} = \mathrm{poly}(n)$ factor to the total complexity of the quantum algorithm.

5.2 Saddle point approximation

In this section, we show how to describe the tight asymptotic complexity of $T(n_1, \ldots, n_D)$ using the saddle point method (a detailed review can be found in [15], Chapter VIII). Our main technical tool will be the following theorem.

▶ **Theorem 2.** *Let $p_1(x_1, \ldots, x_m)$, …, $p_D(x_1, \ldots, x_m)$ be polynomials with non-negative coefficients. Let n be a positive integer and $b_1, \ldots, b_D$ be non-negative rational numbers such that $b_1 + \ldots + b_D = 1$ and $b_d n$ is an integer for all $d \in [D]$. Let $a_{i,d}$ be rational numbers (for $i \in [m]$, $d \in [D]$) and $\alpha_i \coloneqq a_{i,1}b_1 + \ldots + a_{i,D}b_D$. Suppose that $\alpha_i n$ are integer for all $i \in [m]$. Then*

(1) $[x_1^{\alpha_1 n} \cdots x_m^{\alpha_m n}] \prod_{d=1}^D p_d(x_1, \ldots, x_m)^{b_d n} \leq \left(\inf_{x_1,\ldots,x_m>0} \prod_{d=1}^D \left(\frac{p_d(x_1,\ldots,x_m)}{x_1^{a_{1,d}} \cdots x_m^{a_{m,d}}}\right)^{b_d}\right)^n$

(2) $[x_1^{\alpha_1 n} \cdots x_m^{\alpha_m n}] \prod_{d=1}^D p_d(x_1, \ldots, x_m)^{b_d n} = \Omega\left(\left(\inf_{x_1,\ldots,x_m>0} \prod_{d=1}^D \left(\frac{p_d(x_1,\ldots,x_m)}{x_1^{a_{1,d}} \cdots x_m^{a_{m,d}}}\right)^{b_d}\right)^n\right)$,

where Ω depends on the variable n.

Proof. To prove this, we use the following saddle point approximation.[2]

▶ **Theorem 3** (Saddle point method, Theorem 2 in [9])**.** *Let $p(x_1, \ldots, x_m)$ be a polynomial with non-negative coefficients. Let $\alpha_1, \ldots, \alpha_m$ be some rational numbers and let n_i be the series of all integers j such that $\alpha_k j$ are integers and $\left[x_1^{\alpha_1 j} \cdots x_m^{\alpha_m j}\right] p(x_1, \ldots, x_m)^j \neq 0$. Then*

$$\lim_{i\to\infty} \frac{1}{n_i} \log([x_1^{\alpha_1 n_i} \cdots x_m^{\alpha_m n_i}] p(x_1, \ldots, x_m)^{n_i}) = \inf_{x_1,\ldots,x_m>0} \log\left(\frac{p(x_1, \ldots, x_m)}{x_1^{\alpha_1} \cdots x_m^{\alpha_m}}\right).$$

Let $p(x_1, \ldots, x_m) \coloneqq \prod_{d=1}^D p_d(x_1, \ldots, x_m)^{b_d}$, then

$$\frac{p(x_1,\ldots,x_m)}{x_1^{\alpha_1}\cdots x_m^{\alpha_m}} = \frac{\prod_{d=1}^D p_d(x_1,\ldots,x_m)^{b_d}}{x_1^{\alpha_1}\cdots x_m^{\alpha_m}} = \prod_{d=1}^D \frac{p_d(x_1,\ldots,x_m)^{b_d}}{x_1^{a_{1,d}b_d}\cdots x_m^{a_{m,d}b_d}} = \prod_{d=1}^D \left(\frac{p_d(x_1,\ldots,x_m)}{x_1^{a_{1,d}}\cdots x_m^{a_{m,d}}}\right)^{b_d}.$$

[2] Setting $\gamma = 1$ in the statement of the original theorem.

For the first part, as $p(x_1,\dots,x_m)^n$ has non-negative coefficients, the coefficient at the multinomial $x_1^{\alpha_1 n}\cdots x_m^{\alpha_m n}$ is upper bounded by

$$\inf_{x_1,\dots,x_m>0}\frac{p(x_1,\dots,x_m)^n}{x_1^{\alpha_1 n}\cdots x_m^{\alpha_m n}}=\left(\inf_{x_1,\dots,x_m>0}\frac{p(x_1,\dots,x_m)}{x_1^{\alpha_1}\cdots x_m^{\alpha_m}}\right)^n$$

The second part follows directly by Theorem 3. ◀

5.2.1 Optimization program

To determine the complexity of the algorithm, we construct the following optimization problem. Recall that the Algorithm 1 is given by the number of layers K and the constants $\alpha_{k,d}$ that determine the weight of the layers, so assume they are fixed known numbers. Assume that $\alpha_{k,d}$ are all rational numbers between 0 and 1/2 for $k\in[K]$; indeed, we can approximate any real number with arbitrary precision by a rational number. Also let $T_0=1$ and $\alpha_{K+1,d}=1/2$ for all $d\in[D]$ for convenience.

Examine the following program $\mathrm{OPT}(D,K,\{\alpha_{k,d}\})$:

$$\begin{aligned}
\text{minimize } T_D \qquad \text{s.t.} \qquad & T_d \geq \frac{P_d(x)}{x^{\alpha_{1,d}d}} && \forall d\in[D]\\
& T_d^2 \geq \frac{S_{k,d}(x_{k,k},\dots,x_{k,K+1})}{x_{k,k}^{\alpha_{k,d}d}\cdots x_{k,K+1}^{\alpha_{K+1,d}d}} && \forall d\in[D],\forall k\in[K]\\
& T_d\geq 1 && \forall d\in[D]\\
& x>0 &&\\
& x_{k,j}>0 && \forall k\in[K],\forall j\in\{k,\dots,K+1\}
\end{aligned}$$

Let $n\coloneqq n_1+\ldots+n_D$ and $\alpha_k\coloneqq\frac{\sum_{d=1}^D\alpha_{k,d}dn_d}{n}$. Suppose that $T_1,\dots,T_D$ is a feasible point of the program. Then by Theorem 2 (1) (setting $b_i\coloneqq n_i/n$ and $a_{i,d}\coloneqq\alpha_{i,d}d$) we have

$$[x^{\alpha_1 n}]P(x)\leq\inf_{x>0}\prod_{d=1}^D\left(\frac{P_d(x)}{x^{\alpha_{1,d}d}}\right)^{n_d}\leq T_1^{n_1}\cdots T_D^{n_D}.$$

Similarly,

$$\begin{aligned}
[x_{k,k}^{\alpha_k n}\cdots x_{k,K+1}^{\alpha_{K+1}n}]S_k(x_{k,k},\dots,x_{k,K+1})&\leq\inf_{x_{k,k},\dots,x_{k,K+1}>0}\prod_{d=1}^D\left(\frac{S_{k,d}(x_{k,k},\dots,x_{k,K+1})}{x_{k,k}^{\alpha_{k,d}d}\cdots x_{k,K+1}^{\alpha_{K+1,d}d}}\right)^{n_d}\\
&\leq(T_1^{n_1}\cdots T_D^{n_D})^2.
\end{aligned}$$

Therefore, the program provides an upper bound on the complexity. There are two subtleties that we need to address for correctness: firstly, what happens when $\alpha_k n$ is not an integer; secondly, the case when $W_k=W_{k+1}$ for some k. We show that both do not raise an issue in Appendix B.

5.2.2 Optimality of the program

In the start of the analysis, we made an assumption that the exponential complexity $T(n_1,\dots,n_D)$ can be expressed as $T_1^{n_1}\cdots T_D^{n_D}$. In Appendix C, using the lower bound of Theorem 3 (2), we show that $\mathrm{OPT}(D,K,\{\alpha_{k,d}\})$ (which gives an upper bound on the complexity) can indeed achieve such value and also gives the best possible solution.

5.2.3 Total complexity

Finally, in Appendix D we argue that there exists a choice for the parameters $\{\alpha_{k,d}\}$ such that $\mathrm{OPT}(D, K, \{\alpha_{k,d}\}) < D + 1$. Therefore, putting all together, we have the main result:

▶ **Theorem 4.** *There exists a bounded-error quantum algorithm that solves the path in the n-dimensional lattice problem using $\widetilde{O}(T_D^n)$ queries, where $T_D < D + 1$. The optimal value of T_D can be found by optimizing $\mathrm{OPT}(D, K, \{\alpha_{k,d}\})$ over K and $\{\alpha_{k,d}\}$.*

5.3 Complexity for small D

To find the estimate on the complexity for small values of D and K, we have optimized the value of $\mathrm{OPT}(D, K, \{\alpha_{k,d}\})$ using Mathematica (minimizing over the values of $\alpha_{k,d}$). Table 2 compiles the results obtained by the optimization. In case of $D = 1$, we recovered the complexity of the quantum algorithm from [4] for the path in the hypercube problem, which is a special case of our algorithm.

Table 2 The complexity of the quantum algorithm for small values of D and K.

	$D=1$	$D=2$	$D=3$	$D=4$	$D=5$	$D=6$
$K=1$	1.86793	2.76625	3.68995	4.63206	5.58735	6.55223
$K=2$	1.82562	2.67843	3.55933	4.46334	5.38554	6.32193
$K=3$	1.81819	2.66198	3.53322	4.42759	5.34059	6.26840
$K=4$	1.81707	2.65939	3.52893	4.42148	5.33263	6.25862
$K=5$	1.81692	2.65908	3.52836	4.42064	5.33149	6.25720

For $K = 1$, we were able to estimate the complexity for up to $D = 18$. Figure 2 shows the values of the difference between $D + 1$ and T_D for this range.

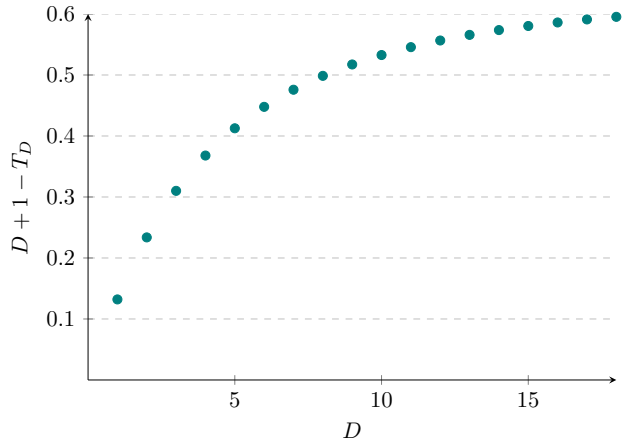

Figure 2 The advantage of the quantum algorithm over the classical for $K = 1$.

Our Mathematica code used for determining the values of T_D can be accessed at `https://doi.org/10.5281/zenodo.4603689`. In Appendix E, we list the parameters for the case $K = 1$.

5.4 Lower bound for general D

Even though Theorem 4 establishes the quantum advantage of the algorithm, it is interesting how large the speedup can get for large D. In this section, we prove that the speedup cannot be substantial, more specifically:

▶ **Theorem 5.** *For any fixed integers $D \geq 1$ and $K \geq 1$, Algorithm 1 performs $\widetilde{\Omega}\left(\left(\frac{D+1}{\mathrm{e}}\right)^n\right)$ queries on the lattice $Q(D, n)$.*

Proof. The structure of the proof is as follows. First, we prove that if $\alpha_{1,D} > \frac{1}{4}$, then the number of queries used in the algorithm during the precalculation step 2 is at least $\widetilde{\Omega}((0.664554(D+1))^n)$ queries. Then, we prove that if $\alpha_{1,D} \leq \frac{1}{4}$, then the quantum search part in steps 3 and 4 performs at least $\widetilde{\Omega}\left(\left(\frac{D+1}{\mathrm{e}}\right)^n\right)$ queries. Therefore, depending on whether $\alpha_{1,D} > \frac{1}{4}$, one of the precalculation or the quantum search performs $\widetilde{\Omega}((c(D+1))^n)$ queries for constant c, and the claim follows, since $\frac{1}{\mathrm{e}} < 0.664554$. Due to space limitations, the proof can be accessed online in the full version of the paper [18] (Appendix B). ◀

6 Time complexity

In this section we examine a possible high-level implementation of the described algorithm and argue that there exists a quantum algorithm with the same exponential time complexity as the query complexity.

Firstly, we assume the commonly used QRAM model of computation that allows to access N memory cells in superposition in time $O(\log N)$ [17]. This is needed when the algorithm accesses the precalculated values of dp. Since in our case N is always at most $(D+1)^n$, this introduces only a $O(\log((D+1)^n)) = O(n)$ additional factor to the time complexity.

The main problem that arises is the efficient implementation of VTS. During the VTS execution, multiple quantum algorithms should be performed in superposition. More formally, to apply VTS to algorithms $\mathcal{A}_1, \ldots, \mathcal{A}_N$, we should specify the *algorithm oracle* that, given the index of the algorithm i and the time step t, applies the t-th step of $\mathcal{A}_i$ (see Section 2.2 of [11] for formal definition of such an oracle and related discussion). If the algorithms $\mathcal{A}_i$ are unstructured, the implementation of such an oracle may take even $O(N)$ time (if, for example, all of the algorithms perform a different gate on different qubits at the t-th step).

We circumvent this issue by showing that it is possible to use only Grover's search to implement the algorithm, retaining the same exponential complexity (however, the sub-exponential factor in the complexity will increase). Nonetheless, the use of VTS in the query algorithm not only achieves a smaller query complexity, but also allowed to prove the estimate on the exponential complexity, which would not be so amiable for the algorithm that uses Grover's search.

6.1 Implementation

The main idea of the implementation is to fix a "class" of vertices for each of the $2K+1$ layers examined by the algorithm, and do this for all $r = O(\log n)$ levels of recursion. We will essentially define these classes by the number of coordinates of a vertex in such layer that are equal to $0, 1, \ldots, D$. Then, we can first fix a class for each layer for all levels of recursion classically. We will show that there are at most n^{D^2} different classes we have to consider at each layer. Since there are $2K+1$ layers at one level of recursion, and $O(\log n)$ levels of recursion, this classical precalculation will take time $n^{O(D^2 K \log n)}$. For each such choice of classes, we will run a quantum algorithm that checks for the path in the hyperlattice constrained on these classes of the vertices the path can go through. The advantage of

the quantum algorithm will come from checking the permutations of the coordinates using Grover's search. The time complexity of the quantum part will be $n^{O(K \log n)} T_D^n$ (T_d^n as in the query algorithm, and $n^{O(K \log n)}$ from the logarithmic factors in Grover's search), therefore the total time complexity will be $n^{O(D^2 K \log n)} \cdot n^{O(K \log n)} T_D^n = n^{O(D^2 K \log n)} T_D^n$, thus the exponential complexity stays the same.

6.2 Layer classes

In all of the applications of VTS in the algorithm, we use it in the following scenario: given a vertex x, examine all vertices y with fixed weight $|y| = W$ such that $y < x$ (note that VTS over the middle layer $\mathcal{L}_{K+1}$ can be viewed in this way by taking x to be the final vertex in the lattice, and VTS over the vertices in the layers symmetrical to $\mathcal{L}_{K+1}$ can be analyzed similarly).

We define a *class* of y's (in respect to x) in the following way. Let $n_{a,b}$ be the number of $i \in [n]$ such that $y_i = a$ and $x_i = b$, where $a \leq b$. All y in the same class have the same values of $n_{a,b}$ for all a, b. Also define a *representative* of a class as a single particular y from that class; we will define it as the lexicographically smallest such y.

As mentioned in the informal description above, we can fix the classes for all layers examined by the quantum algorithm and generate the corresponding representatives classically. Note that in our quantum algorithm, recursive calls work with the sublattice constrained on the vertices $s \leq y \leq t$ for some $s < t$, so for each position of y_i we should have also $y_i \geq s_i$; however, we can reduce it to lattice $0^n \leq y' \leq x$, where $x_i := t_i - s_i$ for all i. To get the real value of y, we generate a representative y', and set $y_i := y'_i + s_i$.

Consider an example for $D = 2$. The following figure illustrates the representative y (note that the order of positions of x here is lexicographical for simplicity, but it may be arbitrary).

$$x = 00\ldots011\ldots\ldots\ 122\ldots\ldots\ldots\ldots\ 2$$
$$y = \underbrace{00\ldots0}_{n_{0,0}}\underbrace{00\ldots0}_{n_{0,1}}\underbrace{11\ldots1}_{n_{1,1}}\underbrace{00\ldots0}_{n_{0,2}}\underbrace{11\ldots1}_{n_{1,2}}\underbrace{22\ldots2}_{n_{2,2}}$$

Figure 3 The (lexicographically smallest) representative for y for $D = 2$.

Note that $n_{a,b}$ can be at most n. Therefore, there are at most n^{D^2} choices for classes at each layer. Thus the total number of different sets of choices for all layers is $n^{O(D^2 K \log n)}$. For each such set of choices, we then run a quantum algorithm that checks for a path in the sublattice constrained on these classes.

6.3 Quantum algorithm

The algorithm basically implements Algorithm 1, with VTS replaced by Grover's search. Thus we only describe how we run the Grover's search. We will also use the analysis of Grover's search with multiple marked elements.

▶ **Theorem 6** (Grover's search). *Let $f : S \to \{0, 1\}$, where $|S| = N$. Suppose we can generate a uniform superposition $\frac{1}{\sqrt{N}} \sum_{x \in S} |x\rangle$ in $O(\text{poly}(\log N))$ time, and there is a bounded-error quantum algorithm $\mathcal{A}$ that computes $f(x)$ with time complexity T. Suppose also that there is a promise that either there are at least k solutions to $f(x) = 1$, or there are none. Then there exists a bounded-error quantum algorithm that runs in time $O(T \log N \sqrt{N/k})$, and detects whether there exists x such that $f(x) = 1$.*

Proof. First, it is well-known that in the case of k marked elements, Grover's algorithm [21] needs $O(\sqrt{N/k})$ iterations. Second, the gate complexity of one iteration of Grover's search is known to be $O(\log N)$. Finally, even though $\mathcal{A}$ has constant probability of error, there is a result that implements Grover's search with a bounded-error oracle without introducing another logarithmic factor [23]. ◀

Now, for a class $\mathcal{C}$ of y's (for a fixed x) we need to generate a superposition $\frac{1}{\sqrt{|\mathcal{C}|}}\sum_{y\in\mathcal{C}}|y\rangle$ efficiently to apply Grover's algorithm. We will generate a slightly different superposition for the same purposes. Let $I_1,\dots,I_D$ be sets $I_d \coloneqq \{i\in[n] \mid x_i = d\}$ and let $n_d \coloneqq |I_d|$. Let $y_{\mathcal{C}}$ be the representative of $\mathcal{C}$. We will generate the superposition

$$\bigotimes_{d=0}^{D}\frac{1}{\sqrt{n_d!}}\sum_{\pi\in S_{n_d}}|\pi(y_{\mathcal{C}\,I_d})\rangle\,|\pi\rangle\,, \tag{3}$$

where $y_{\mathcal{C}\,I_d}$ are the positions of $y_{\mathcal{C}}$ in I_d.

We need a couple of procedures to generate such state. First, there exists a procedure to generate the uniform superposition of permutations $\frac{1}{\sqrt{n!}}\sum_{\pi\in S_n}|\pi_1,\dots,\pi_n\rangle$ that requires $O(n^2\log n)$ elementary gates [2, 10]. Then, we can build a circuit with $O(\mathrm{poly}(n))$ gates that takes as an input $\pi\in S_n$, $s\in\{0,1,\dots,D\}^n$ and returns $\pi(s)$. Such an circuit essentially could work as follows: let $t\coloneqq 0^n$; then for each pair $i,j\in[n]$, check whether $\pi(i)=j$; if yes, let $t_j \leftarrow t_j + s_{\pi(i)}$; in the end return t. Using these two subroutines, we can generate the required superposition using $O(\mathrm{poly}(n))$ gates (we assume D is a constant).

However, we do not necessarily know the sets I_d, because the positions of x have been permuted by previous applications of permutations. To mitigate this, note that we can access this permutation in its own register from the previous computation. That is, suppose that x belongs to a class $\mathcal{C}'$ and $x=\sigma(x_{\mathcal{C}'})$, where $x_{\mathcal{C}'}$ is the representative of $\mathcal{C}'$ generated by the classical algorithm from the previous subsection. Then we have the state $|\sigma(x_{\mathcal{C}'})\rangle\,|\sigma\rangle$.

We can then apply σ to both $\pi(y_{\mathcal{C}})$ and π. That is, we implement the transformation

$$|\pi(y_{\mathcal{C}})\rangle\,|\pi\rangle \to |\sigma(\pi(y_{\mathcal{C}}))\rangle\,|\sigma\pi\rangle\,.$$

Such transformation can also be implemented in $O(\mathrm{poly}(n))$ gates. Note that now we store the permutation $\sigma\pi$ in a separate register, which we use in a similar way recursively.

Finally, examine the number of positive solutions among $\pi(y_{\mathcal{C}})$. That is, for how many π there exists a path from $\pi(y)$ to x? Suppose that there is a path from y to x for some $y\in\mathcal{C}$. Examine the indices I_d; for $n_{a,d}$ of these indices i we have $y_i=a$. There are exactly $n_{a,d}!$ permutations that permute these indices and don't change y. Hence, there are $\prod_{a=0}^{d} n_{a,d}!$ distinct permutations $\pi\in S_{n_d}$ such that $\pi(y)=y$.

Therefore, there are $k\coloneqq\prod_{d=0}^{D}\prod_{a=0}^{d} n_{a,d}!$ distinct permutations π among the considered such that $\pi(y)=y$. The total number of considered permutations is $N\coloneqq\prod_{d=0}^{D} n_d!$. Among these permutations, either there are no positive solutions, or at least k of the solutions are positive. Grover's search then works in time $O(T\log N\sqrt{N/k})$. In this case, N/k is exactly the size of the class $\mathcal{C}$, because $\frac{n_d!}{n_{0,d}!\cdots n_{d,d}!}$ is the number of unique permutations of $y_{\mathcal{C}\,P_d}$, the multinomial coefficient $\binom{n_d}{n_{0,d},\dots,n_{d,d}}$. Hence the state Eq. (3) effectively replaces the need for the state $\frac{1}{\sqrt{|\mathcal{C}|}}\sum_{y\in\mathcal{C}}|y\rangle$.

6.4 Total complexity

Finally, we discuss the total time complexity of this algorithm. The exponential time complexity of the described quantum algorithm is at most the exponential query complexity because Grover's search examines a single class $\mathcal{C}$, while VTS in the query algorithm examines

all possible classes. Since Grover's search has a logarithmic factor overhead, the total time complexity of the quantum part of the algorithm is what is described in Section 5 multiplied by $n^{O(K \log n)}$, resulting in $n^{O(K \log n)} T_1^{n_1} \cdots T_D^{n_D}$.

Since there are $n^{O(D^2 K \log n)}$ sets of choices for the classes of the layers, the final total time complexity of the algorithm is $n^{O(D^2 K \log n)} T_1^{n_1} \cdots T_D^{n_D}$.

For the space complexity, note that the precalculation step requires asymptotically the same exponential amount of space as time, thus $T_1^{n_1} \cdots T_D^{n_D}$ is also the exponential space complexity of the algorithm.

Therefore, we have the following result.

▶ **Theorem 7.** *Assuming QRAM model of computation, there exists a quantum algorithm that solves the path in the n-dimensional lattice problem with time and space complexity* $\mathrm{poly}(n)^{D^2 \log n} \cdot T_D^n$.

7 Applications

7.1 Set multicover

As an example application of our algorithm, we apply it to the Set Multicover problem (SMC). This is a generalization of the Minimum Set Cover problem. The SMC problem is formulated as follows:

Input: A set of subsets $\mathcal{S} \subseteq 2^{[n]}$, and a positive integer D.

Output: The size k of the smallest tuple $(S_1, \ldots, S_k) \in \mathcal{S}^k$, such that for all $i \in [n]$, we have $|\{j \mid i \in S_j\}| \geq D$, that is, each element is covered at least D times (note that each set $S \in \mathcal{S}$ can be used more than once).

Denote this problem by SMC_D, and $m \coloneqq |\mathcal{S}|$. This problem has been studied classically, and there exists an exact deterministic algorithm based on the inclusion-exclusion principle that solves this problem in time $\widetilde{O}(m(D+1)^n)$ and polynomial space [27, 24]. While there are various approximation algorithms for this problem, we are not aware of a more efficient classical exact algorithm.

There is a different simple classical dynamic programming algorithm for this problem with the same time complexity (although it uses exponential space), which we can speed up using our quantum algorithm. For a vector $x \in \{0, 1, \ldots, D\}^n$, define $\mathrm{dp}(x)$ to be the size k of the smallest tuple $(\mathcal{C}_1, \ldots, \mathcal{C}_k) \in \mathcal{S}^k$ such that for each i, we have $|\{j \in [k] \mid i \in \mathcal{C}_j\}| \geq x_i$. It can be calculated using the recurrence

$$\mathrm{dp}(0^n) = 0, \qquad \mathrm{dp}(x) = 1 + \min_{S \in \mathcal{S}} \{\mathrm{dp}(x')\},$$

where x' is given by $x'_i = \max\{0, x_i - \chi(S)_i\}$ for all i. Consequently, the answer to the problem is equal to $\mathrm{dp}(D^n)$. The number of distinct x is $(D+1)^n$, and $\mathrm{dp}(x)$ for a single x can be calculated in time $O(nm)$, if $\mathrm{dp}(y)$ has been calculated for all $y < x$. Thus the time complexity is $O(nm(D+1)^n)$ and space complexity is $O((D+1)^n)$.

Note that even though the state space of the dynamic programming here is $\{0, 1, \ldots, D\}^n$, the underlying transition graph is not the same as the hyperlattice examined in the quantum algorithm. A set $S \in \mathcal{S}$ can connect vertices that are $|S|$ distance apart from each other, unlike distance 1 in the hyperlattice. We can essentially reduce this to the hyperlattice-like transition graph by breaking such transition into $|S|$ distinct transitions.

More formally, examine pairs (x, S), where $x \in \{0, 1, \ldots, D\}^n$, $S \in \mathcal{S}$. Let $e(x, S) := \min\{i \in S \mid x_i > 0\}$; if there is no such i, let $e(x, S)$ be 0. Define a new function

$$\mathrm{dp}(x, S) = \begin{cases} 0, & \text{if } x = 0^n, \\ \mathrm{dp}(x - \chi(\{e(x, S)\}), S), & \text{if } e(x, S) > 0, \\ 1 + \min_{T \in \mathcal{S}, e(x,T)>0}\{\mathrm{dp}(x - \chi(\{e(x, T)\}), T\}, & \text{if } e(x, S) = 0. \end{cases}$$

The new recursion also solves SMC_D, and the answer is equal to $\min_{S \in \mathcal{S}}\{\mathrm{dp}(D^n, S)\}$.

Examine the underlying transition graph between pairs (x, S). We can see that there is a transition between two pairs (x, S) and (y, T) only if $y_i = x_i + 1$ for exactly one i, and $y_i = x_i$ for other i. This is the n-dimensional lattice graph $Q(D, n)$. Thus we can apply our quantum algorithm with a few modifications:

- We now run Grover's search over (x, S) with fixed $|x|$ for all $S \in \mathcal{S}$. This adds a $\mathrm{poly}(m, n)$ factor to each run of Grover's search.
- Since we are searching for the minimum value of dp, we actually need a quantum algorithm for finding the minimum instead of Grover's search. We can use the well-known quantum minimum finding algorithm that retains the same query complexity as Grover's search [14][3]. It introduces only an additional $O(\log n)$ factor for the queries of minimum finding to encode the values of dp, since $\mathrm{dp}(x, S)$ can be as large as Dn.
- A single query for a transition between pairs (x, S) and (y, T) in this case returns the value of the value added to the dp at transition, which is either 0 or 1. If these pairs are not connected in the transition graph, the query can return ∞. Note that such query can be implemented in $\mathrm{poly}(m, n)$ time.

Since the total number of runs of Grover's search is $O(K \log n)$, the additional factor incurred is $\mathrm{poly}(m, n)^{O(K \log n)}$. This provides a quantum algorithm for this problem with total time complexity

$$\mathrm{poly}(m, n)^{O(K \log n)} \cdot n^{O(D^2 K \log n)} T_D^n = m^{O(K \log n)} n^{O(D^2 K \log n)} T_D^n.$$

Therefore, we have the following result.

▶ **Theorem 8.** *Assuming the QRAM model of computation, there exists a quantum algorithm that solves* SMC_D *in time and space* $\mathrm{poly}(m, n)^{\log n} T_D^n$*, where* $T_D < D + 1$*.*

7.2 Related problems

We are also aware of a couple of other works that implement the dynamic programming on the $\{0, 1, \ldots, D\}^n$ n-dimensional lattice.

Psaraftis examined the job scheduling problem [28], with application to aircraft landing scheduling. The problem requires ordering n groups of jobs with D identical jobs in each group. A cost transition function is given: the cost of processing a job belonging to group j after processing a job belonging to group i is given by $f(i, j, d_1, \ldots, d_n)$, where d_i is the number of jobs left to process. The task is to find an ordering of the nD jobs that minimizes the total cost. This is almost exactly the setting for our quantum algorithm, hence we get $\mathrm{poly}(n)^{\log n} T_D^n$ time quantum algorithm. Psaraftis proposed a classical $O(n^2(D + 1)^n)$ time

[3] Note that this algorithm assumes queries with zero error, but we apply it to bounded-error queries. However, it consists of multiple runs of Grover's search, so we can still use the result of [23] to avoid the additional logarithmic factor.

dynamic programming algorithm. Note that if $f(i, j, d_1, \ldots, d_n)$ are unstructured (can be arbitrary values), then there does not exist a faster classical algorithm by the lower bound of Section 3.

However, if $f(i, j, d_1, \ldots, d_n)$ are structured or can be computed efficiently by an oracle, there exist more efficient classical algorithms for these kinds of problems. For instance, the many-visits travelling salesman problem (MV-TSP) asks for the shortest route in a weighted n-vertex graph that visits vertex i exactly D_i times. In this case, $f(i, j, d_1, \ldots, d_n) = w(i, j)$, where $w(i, j)$ is the weight of the edge between i and j. The state-of-the-art classical algorithm by Kowalik et al. solves this problem in $\widetilde{O}(4^n)$ time and space [32]. Thus, our quantum algorithm does not provide an advantage. It would be quite interesting to see if there exists a quantum speedup for this MV-TSP algorithm.

Lastly, Gromicho et al. proposed an exact algorithm for the job-shop scheduling problem [20, 31]. In this problem, there are n jobs to be processed on D machines. Each job consists of D tasks, with each task to be performed on a separate machine. The tasks for each job need to be processed in a specific order. The time to process job i on machine j is given by p_{ij}. Each machine can perform at most one task at any moment, but machines can perform the tasks in parallel. The problem is to schedule the starting times for all tasks so as to minimize the last ending time of the tasks. Gromicho et al. give a dynamic programming algorithm that solves the problem in time $O((p_{\max})^{2n}(D+1))^n$, where $p_{\max} = \max_{i,j}\{p_{ij}\}$.

The states of their dynamic programming are also vectors in $\{0, 1, \ldots, D\}^n$: a state x represents a partial completion of tasks, where x_i tasks of job i have already been completed. Their dynamic programming calculates the set of task schedulings for x that can be potentially extended to an optimal scheduling for all tasks. However, it is not clear how to apply Grover's search to calculate a whole set of schedulings. Therefore, even though the state space is the same as in our algorithm, we do not know whether it is possible to apply it in this case.

References

1. Amir Abboud. Fine-Grained Reductions and Quantum Speedups for Dynamic Programming. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:13, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICALP.2019.8.
2. Daniel S. Abrams and Seth Lloyd. Simulation of many-body fermi systems on a universal quantum computer. *Phys. Rev. Lett.*, 79:2586–2589, September 1997. doi:10.1103/PhysRevLett.79.2586.
3. Andris Ambainis. Quantum search with variable times. *Theory of Computing Systems*, 47(3):786–807, 2010. doi:10.1007/s00224-009-9219-1.
4. Andris Ambainis, Kaspars Balodis, Jānis Iraids, Martins Kokainis, Krišjānis Prūsis, and Jevgēnijs Vihrovs. Quantum speedups for exponential-time dynamic programming algorithms. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '19, page 1783–1793, USA, 2019. Society for Industrial and Applied Mathematics. doi:10.1137/1.9781611975482.107.
5. Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *J. ACM*, 9(1):61–63, 1962. doi:10.1145/321105.321111.
6. Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and weaknesses of quantum computing. *SIAM Journal on Computing*, 26(5):1510–1523, 1997. doi:10.1137/S0097539796300933.

7 Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. A note on exact algorithms for vertex ordering problems on graphs. *Theory of Computing Systems*, 50(3):420–432, 2012. doi:10.1007/s00224-011-9312-0.

8 Harry Buhrman and Ronald de Wolf. Complexity measures and decision tree complexity: a survey. *Theoretical Computer Science*, 288(1):21–43, 2002. doi:10.1016/S0304-3975(01)00144-X.

9 David Burshtein and Gadi Miller. Asymptotic enumeration methods for analyzing LDPC codes. *IEEE Transactions on Information Theory*, 50:1115–1131, 2004. doi:10.1109/TIT.2004.828064.

10 Mitchell Chiew, Kooper de Lacy, Chao-Hua Yu, Sam Marsh, and Jingbo B. Wang. Graph comparison via nonlinear quantum search. *Quantum Information Processing*, 18:302, August 2019. doi:10.1007/s11128-019-2407-2.

11 Arjan Cornelissen, Stacey Jeffery, Maris Ozols, and Alvaro Piedrafita. Span Programs and Quantum Time Complexity. In Javier Esparza and Daniel Kráľ, editors, *45th International Symposium on Mathematical Foundations of Computer Science (MFCS 2020)*, volume 170 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.MFCS.2020.26.

12 Marek Cygan and Marcin Pilipczuk. Faster exact bandwidth. In *Graph-Theoretic Concepts in Computer Science*, pages 101–109, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-92248-3_10.

13 N. G. de Bruijn, C^{A}. van Ebbenhorst Tengbergen, and D. Kruyswijk. On the set of divisors of a number. *Nieuw Archief voor Wiskunde*, 23(2):191–193, 1951. URL: https://research.tue.nl/en/publications/on-the-set-of-divisors-of-a-number.

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

15 Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, USA, 1 edition, 2009. doi:10.1017/CBO9780511801655.

16 Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Springer Science & Business Media, 2010.

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

18 Adam Glos, Martins Kokainis, Ryuhei Mori, and Jevgēnijs Vihrovs. Quantum speedups for dynamic programming on n-dimensional lattice graphs, 2021. arXiv:2104.14384.

19 I. J. Good. Saddle-point Methods for the Multinomial Distribution. *The Annals of Mathematical Statistics*, 28(4):861–881, 1957. doi:10.1214/aoms/1177706790.

20 Joaquim A. S. Gromicho, Jelke J. van Hoorn, Francisco Saldanha da Gama, and Gerrit T. Timmer. Solving the job-shop scheduling problem optimally by dynamic programming. *Computers & Operations Research*, 39(12):2968–2977, 2012. doi:10.1016/j.cor.2012.02.024.

21 Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237814.237866.

22 Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of SIAM*, 10(1):196–210, 1962. doi:10.1145/800029.808532.

23 Peter Høyer, Michele Mosca, and Ronald de Wolf. Quantum search on bounded-error inputs. In *Automata, Languages and Programming*, ICALP'03, page 291–299, Berlin, Heidelberg, 2003. Springer-Verlag. doi:10.1007/3-540-45061-0_25.

24 Qiang-Sheng Hua, Yuexuan Wang, Dongxiao Yu, and Francis C.M. Lau. Dynamic programming based algorithms for set multicover and multiset multicover problems. *Theoretical Computer Science*, 411(26):2467–2474, 2010. doi:10.1016/j.tcs.2010.02.016.

25 Bronisław Knaster, Casimir Kuratowski, and Stefan Mazurkiewicz. Ein beweis des fixpunktsatzes für n-dimensionale simplexe. *Fundamenta Mathematicae*, 14(1):132–137, 1929. doi:10.4064/fm-14-1-132-137.

26 Masayuki Miyamoto, Masakazu Iwamura, Koichi Kise, and François Le Gall. Quantum speedup for the minimum steiner tree problem. In Donghyun Kim, R. N. Uma, Zhipeng Cai, and Dong Hoon Lee, editors, *Computing and Combinatorics*, pages 234–245, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-58150-3_19.

27 Jesper Nederlof. Inclusion exclusion for hard problems. Master's thesis, Utrecht University, 2008. URL: https://webspace.science.uu.nl/~neder003/MScThesis.pdf.

28 Harilaos N. Psaraftis. A dynamic programming approach for sequencing groups of identical jobs. *Operations Research*, 28(6):1347–1359, 1980. doi:10.1287/opre.28.6.1347.

29 Kazuya Shimizu and Ryuhei Mori. Exponential-time quantum algorithms for graph coloring problems. In Yoshiharu Kohayakawa and Flávio Keidi Miyazawa, editors, *LATIN 2020: Theoretical Informatics*, pages 387–398, Cham, 2020. Springer International Publishing. arXiv:1907.00529.

30 Seiichiro Tani. Quantum Algorithm for Finding the Optimal Variable Ordering for Binary Decision Diagrams. In Susanne Albers, editor, *17th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2020)*, volume 162 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 36:1–36:19, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SWAT.2020.36.

31 Jelke J. van Hoorn, Agustín Nogueira, Ignacio Ojea, and Joaquim A. S. Gromicho. An corrigendum on the paper: Solving the job-shop scheduling problem optimally by dynamic programming. *Computers & Operations Research*, 78:381, 2017. doi:10.1016/j.cor.2016.09.001.

32 Łukasz Kowalik, Shaohua Li, Wojciech Nadara, Marcin Smulewicz, and Magnus Wahlström. Many Visits TSP Revisited. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 66:1–66:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2020.66.

A Depth of recursion

Note that the algorithm stops the recursive calls if for at least one k, we have $W_k = W_{k+1}$, in which case it runs the classical dynamic programming on the whole sublattice at step 1. That happens when

$$\left\lfloor \sum_{d=1}^{D} \alpha_{k,d} d n_d \right\rfloor = \left\lfloor \sum_{d=1}^{D} \alpha_{k+1,d} d n_d \right\rfloor.$$

If this is true, then we also have $\sum_{d=1}^{D} \alpha_{k+1,d} d n_d - \sum_{d=1}^{D} \alpha_{k,d} d n_d = c$ for some constant $c < 1$. By regrouping the terms, we get

$$\sum_{d=1}^{D} (\alpha_{k+1,d} - \alpha_{k,d}) d n_d = c.$$

Denote $h \coloneqq \min_{d \in [D]} \{\alpha_{k+1,d} - \alpha_{k,d}\}$. Then $\sum_{d=1}^{D} d n_d \leq \frac{c}{h}$. Note that the left hand side is the maximum total weight of a vertex. However, at each recursive call the difference between the vertices with the minimum and maximum total weights decreases twice, since the VTS call at step 4 runs over the vertices with weight half the current difference. Since c and h is constant, after $O(\log(nD)) = O(\log n)$ recursive calls the recursion stops. Moreover, the classical dynamic programming then runs on a sublattice of constant size, hence adds only a factor of $O(1)$ to the overall complexity.

Lastly, we can address the contribution of the constant factor of VTS from Theorem 1 to the complexity of our algorithm. At one level of recursion there are $K+1$ nested applications of VTS, and there are $O(\log n)$ levels of recursion. Therefore, the total overhead incurred is $O(1)^{O(K \log n)} = \mathrm{poly}(n)$, since K is a constant.

B Complexity technicalities of the optimization program

For correctness, we need to address the following two subtleties.

- The numbers $\alpha_k n$ might not be integer; in Algorithm 1, the weights of the layers are defined by $W_k = \lfloor \alpha_k n \rfloor$. This is a problem, since the inequalities in the program use precisely the numbers $\alpha_{k,d}$. Examine the coefficient $[x_1^{\lfloor \alpha_1 n \rfloor} \cdots x_m^{\lfloor \alpha_m n \rfloor}] p(x_1, \ldots, x_m)$ in such general case (when we need to round the powers). Let $\delta_k := \alpha_k n - \lfloor \alpha_k n \rfloor$, here $0 \leq \delta_k < 1$. Then, by Theorem 2 (1),

$$\left[x_1^{\lfloor \alpha_1 n \rfloor} \cdots x_m^{\lfloor \alpha_m n \rfloor}\right] p(x_1, \ldots, x_m)^n \leq \inf_{x_1, \ldots, x_m \geq 0} \frac{p(x_1, \ldots, x_m)^n}{x_1^{\alpha_1 n - \delta_1} \cdots x_m^{\alpha_m n - \delta_m}}. \tag{4}$$

  Now let $\hat{x}_1, \ldots, \hat{x}_m$ be the arguments that achieve $\inf_{x_1, \ldots, x_m \geq 0} \frac{p(x_1, \ldots, x_m)}{x_1^{\alpha_1} \cdots x_m^{\alpha_m}}$. Since $0 \leq \delta_k < 1$, we have $\hat{x}_k^{\delta_k} \leq \max\{\hat{x}_k, 1\}$. Hence, (4) is at most

$$(\hat{x}_1^{\delta_1} \cdots \hat{x}_m^{\delta_m}) \cdot \frac{p(\hat{x}_1, \ldots, \hat{x}_m)^n}{\hat{x}_1^{\alpha_1 n} \cdots \hat{x}_m^{\alpha_m n}} \leq \left(\prod_{k=1}^{m} \max\{\hat{x}_k, 1\} \right) \cdot \left(\inf_{x_1, \ldots, x_m \geq 0} \frac{p(x_1, \ldots, x_m)}{x_1^{\alpha_1} \cdots x_m^{\alpha_m}} \right)^n.$$

  As the additional factor is a constant, we can ignore it in the complexity.
- The second issue is when $W_k = W_{k+1}$ for some k. Then according to Algorithm 1, we run the classical algorithm with complexity $\widetilde{\Theta}((D+1)^n)$. However, in that case n is constant (see Appendix A), which gives only a constant factor to the complexity.

C Proof of the optimality of the optimization program

- First, we prove that $\mathrm{OPT}(D, K, \{\alpha_{k,d}\})$ has a feasible solution. For that, we need to show that all polynomials in the program can be upper bounded by a constant for some fixed values of the variables.
  First of all, $\frac{P_d(x)}{x^{\alpha_{1,d} d}}$ is upper bounded by $d+1$ (setting $x = 1$). Now fix k and examine the values $\frac{S_{k,d}(x_{k,k}, \ldots, x_{k,K+1})}{x_{k,k}^{\alpha_{k,d} d} \cdots x_{k,K+1}^{\alpha_{K+1,d} d}}$. Examine only such assignments of the variables $x_{k,j}$ that $x_{k,k} x_{k,k+1} = 1$ and $x_{k,j} = 1$ for all other $j > k+1$. Now we write the polynomial as a univariate polynomial $S_{k,d}(y) := S_{k,d}(1/y, y, 1, 1, \ldots, 1)$. Note that for any summand of $S_{k,d}(y)$, if it contains some T_i^2 as a factor, then it is of the form $x_{k,k}^{p_k} x_{k,k+1}^{p_k + i} \cdot T_i^2 = y^i T_i^2$. Hence the polynomial can be written as $S_{k,d}(y) = \sum_{i=0}^{d} c_i y^i T_i^2$ for some constants $c_1, \ldots, c_d$. From this we can rewrite the corresponding program inequality and express T_d^2:

$$T_d^2 \geq \frac{\sum_{i=0}^{d} c_i y^i T_i^2}{y^{(\alpha_{k+1,d} - \alpha_{k,d}) d}} \tag{5}$$

$$T_d^2 \geq \frac{\sum_{i=0}^{d-1} c_i y^i T_i^2}{y^{(\alpha_{k+1,d} - \alpha_{k,d}) d}} + y^{(1 - \alpha_{k+1,d} + \alpha_{k,d}) d} c_d T_d^2$$

$$T_d^2 \geq \frac{1}{1 - y^{(1 - \alpha_{k+1,d} + \alpha_{k,d}) d} c_d} \cdot \frac{\sum_{i=0}^{d-1} c_i y^i T_i^2}{y^{(\alpha_{k+1,d} - \alpha_{k,d}) d}}.$$

Note that c_d are constants that do not depend on T_i. If the right hand side is negative, then it follows that the original inequality Eq. (5) does not hold. Thus we need to pick such y that the right hand side is positive for all d. Hence we require that

$$y < \left(\frac{1}{c_d}\right)^{\frac{1}{(1-\alpha_{k+1,d}+\alpha_{k,d})d}}.$$

Since the right hand side is a constant that does not depend on T_i, we can pick such y that satisfies this inequality for all d. Then it follows that all T_i is also upper bounded by some constants (by induction on i).

Now the question remains whether the optimal solution to $\mathrm{OPT}(D, K, \{\alpha_{k,d}\})$ gives the optimal complexity. That is, is the complexity $T_1^n \cdots T_D^{n_D}$ given by the optimal solution of the optimization program such that T_D is the smallest possible?

Suppose that indeed the complexity of the algorithm is upper bounded by $T_1^n \cdots T_D^{n_D}$ for some T_1, ..., T_D. We will derive a corresponding feasible point for the optimization program.

Examine the complexity of the algorithm for $n_1 = b_1 n, \ldots, n_D = b_D n$ for some fixed rational b_i such that $b_1 + \ldots + b_D = 1$. The coefficients of the polynomials P and S_k give the complexity of the corresponding part of the algorithm (precalculation, and quantum search until the k-th level, respectively). Such coefficients are of the form $[x_1^{\alpha_1 n} \cdots x_m^{\alpha_m n}] \prod_{d=1}^{D} p_d(x_1, \ldots, x_m)^{n_d}$. Let $A_d \coloneqq T_d$, if $p = P$, and $A_d \coloneqq T_d^2$, if $p = S_k$. Then we have

$$A_1^{n_1} \cdots A_D^{n_D} \geq [x_1^{\alpha_1 n} \cdots x_m^{\alpha_m n}] \prod_{d=1}^{D} p_d(x_1, \ldots, x_m)^{n_d}. \tag{6}$$

On the other hand, (6) is at least

$$\Omega\left(\left(\inf_{x_1,\ldots,x_m>0} \prod_{d=1}^{D} \left(\frac{p_d(x_1, \ldots, x_m)}{x_1^{a_{1,d}} \cdots x_m^{a_{m,d}}}\right)^{b_d}\right)^n\right)$$

when n grows large by Theorem 2 (2) (setting $a_{i,d} \coloneqq \alpha_{i,d} d$). Then, in the limit $n \to \infty$, we have

$$A_1^{b_1} \cdots A_D^{b_D} \geq \inf_{x_1,\ldots,x_m>0} \prod_{d=1}^{D} \left(\frac{p_d(x_1, \ldots, x_m)}{x_1^{a_{1,d}} \cdots x_m^{a_{m,d}}}\right)^{b_d}. \tag{7}$$

Now let Δ_{D-1} be the standard D-simplex defined by $\{b \in \mathbb{R}^D \mid b_1 + \ldots + b_D = 1, b_d \geq 0\}$. Define $F_d(x) \coloneqq \frac{p_d(x_1,\ldots,x_m)}{x_1^{a_{1,d}} \cdots x_m^{a_{m,d}}}$, and $F(b, x) \coloneqq \prod_{d=1}^{D} F_d(x)^{b_d}$ for $b \in \Delta_{D-1}$ and $x \in \mathbb{R}^m_{>0}$.

First, we prove that for a fixed b, the function $F(b, x)$ is strictly convex. Examine the polynomial $p_d(x_1, \ldots, x_m)$, which is either $P_d(x)$ or $S_{k,d}(x_{k,k}, \ldots, x_{k,K+1})$. It was shown in [19], Theorem 6.3 that if the coefficients of $p_d(x_1, \ldots, x_m)$ are non-negative, and the points $(c_1, \ldots, c_m)$, at which

$$[x_1^{c_1} \cdots x_m^{c_m}] p_d(x_1, \ldots, x_m) > 0,$$

linearly span an m-dimensional space, then $\log(F_d(x))$ is a strictly convex function. If $p_d = P_d$, then this property immediately follows, because there is just one variable x and the polynomial is non-constant. For $p_d = S_{k,d}$, the polynomial consists of summands of the form $T_{c_{k+1}-c_k}^2 x_{k,k}^{c_k} x_{k,k+1}^{c_{k+1}} \cdots x_{k,K+1}^{c_{K+1}}$, for $c_k \leq c_{k+1} \leq \ldots \leq c_{K+1}$. Note that the

coefficient $T^2_{c_{k+1}-c_k}$ is positive. Thus the points $(c_k, \dots, c_{K+1}) = (0, \dots, 0, 1, \dots, 1)$ indeed linearly span a $(K-k+2)$-dimensional space. Therefore, $\log(F_d(x))$ is strictly convex. Then also the function $\sum_{d=1}^{D} b_d \log(F_d(x)) = \log(F(b,x))$ is strictly convex (for fixed b), as the sum of strictly convex functions is convex. Therefore, $F(b,x)$ is strictly convex as well.

Therefore, the argument $\hat{x}(b)$ achieving $\inf_{x \in \mathbb{R}^m_{>0}} F(b,x)$ is unique. Let $\hat{F}_d(b) \coloneqq F_d(\hat{x}(b))$ and define D subsets of the simplex $C_d := \{b \in \Delta_{D-1} \mid \hat{F}_d(b) \leq A_d\}$. We will apply the following result for these sets:

▶ **Theorem 9** (Knaster-Kuratowski-Mazurkiewicz lemma [25]). *Let the vertices of Δ_{D-1} be labeled by integers from 1 to D. Let $C_1, \dots, C_D$ be a family of closed sets such that for any $I \subseteq [D]$, the convex hull of the vertices labeled by I is covered by $\cup_{d \in I} C_d$. Then $\cap_{d \in [D]} C_d \neq \varnothing$.*

We check that the conditions of the lemma apply to our sets. First, note that $F(b,x)$ is continuous and strictly convex for a fixed b, hence $\hat{x}(b)$ is continuous and thus $\hat{F}_d(b)$ is continuous as well. Therefore, the "threshold" sets C_d are closed.

Secondly, let $I \subseteq [D]$ and examine a point b in the convex hull of the simplex vertices labeled by I. For such a point, we have $b_d = 0$ for all $d \notin I$. For the indices $d \in I$, for at least one we should have $\hat{F}_d(b) \leq A_d$, otherwise the inequality in Eq. (7) would be contradicted. Note that it was stated only for rational b, but since $\hat{F}_d(b)$ are continuous and any real number can be approximated with a rational number to arbitrary precision, the inequality also holds for real b. Thus indeed any such b is covered by $\cup_{d \in I} C_d$.

Therefore, we can apply the lemma and it follows that there exists a point $b \in \Delta_{D-1}$ such that $A_d \geq \hat{F}_d(b)$ for all $d \in [D]$. The corresponding point $\hat{x}(b)$ is a feasible point for the examined set of inequalities in the optimization program.

D Proof of the quantum speedup

Examine the algorithm with only $K = 1$; the optimal complexity for any $K > 1$ cannot be larger, as we can simulate K levels with $K+1$ levels by setting $\alpha_{2,d} = \alpha_{1,d} + \epsilon$ for $\epsilon \to 0$ for all $d \in [D]$. For simplicity, denote $\alpha_d := \alpha_{1,d}$.

- Now examine the precalculation inequalities in $\mathrm{OPT}(D, 1, \{\alpha_{1,d}\})$. For any values of $\alpha_{1,d}$, if we set $x = 1$, we have $\frac{P_d(x)}{x^{\alpha_d d}} = \frac{\sum_{i=0}^{d} x^i}{x^{\alpha_d d}} = d + 1$. The derivative is equal to

$$\left(\frac{\sum_{i=0}^{d} x^i}{x^{\alpha_d d}}\right)' = \frac{x^{\alpha_d d} \cdot \sum_{i=1}^{d} i x^{i-1} - \alpha_d d x^{\alpha_d d - 1} \cdot \sum_{i=0}^{d} x^i}{x^{2\alpha_d d}} = \frac{d(d+1)}{2} - \alpha_d d(d+1)$$

  at point $x = 1$. Thus when $\alpha_d < \frac{1}{2}$, the derivative is positive. It means that for arbitrary $\alpha_d < \frac{1}{2}$, there exists some $x(d)$ such that $\frac{P_d(x)}{x^{\alpha_d d}} < d + 1$, and $\frac{P_d(x)}{x^{\alpha_d d}}$ monotonically grows on $x \in [x(d), 1]$. Thus, for arbitrary setting of $\{\alpha_d\}$ such that $\alpha_d < \frac{1}{2}$ for all $d \in [D]$, we can take $\hat{x} \coloneqq \max_{d \in [D]}\{x(d)\}$ as the common parameter, in which case all $\frac{P_d(\hat{x})}{\hat{x}^{\alpha_d d}} < d + 1$.
- Now examine the set of the quantum search inequalities. Let $y \coloneqq x_{1,1}$ and $z \coloneqq x_{1,2}$ for simplicity. Then such inequalities are given by

$$T_d^2 \geq S_{1,d}(y,z) = \frac{\sum_{i=0}^{d} T_i^2 \sum_{p=0}^{d-i} y^p z^{p+i}}{y^{\alpha_d d} z^{d/2}}.$$

Now restrict the variables to condition $yz = 1$. In that case, the polynomial above simplifies to

$$S_{1,d}(z) := \frac{\sum_{i=0}^{d} T_i^2 \sum_{p=0}^{d-i} z^i}{y^{\frac{d}{2}+d\left(\alpha_d-\frac{1}{2}\right)} z^{d/2}} = \left(\sum_{i=0}^{d} T_i^2 (d-i+1) z^i\right) \cdot z^{d\left(\alpha_d-\frac{1}{2}\right)}.$$

We now find such values of z and $\alpha_1, \ldots, \alpha_D$ so that $S_{1,d}(z) < (d+1)^2$ for all $d \in [D]$, where $T_1, \ldots, T_D$ are any values such that $T_d \leq d+1$ for all $d \in [D]$. Denote $\hat{S}_{1,d}(z)$ to be $S_{1,d}(z)$ with $T_d = d+1$ for all $d \in [D]$, then $\hat{S}_{1,d}(z) < (d+1)^2$ as well. Now let T_d be the maximum of $\frac{P_d(\hat{x})}{\hat{x}^{\alpha_d d}}$ from the previous bullet and $\hat{S}_{1,d}(z)$. Then, $T_d < d+1$, and we have both $T_d \geq \frac{P_d(\hat{x})}{\hat{x}^{\alpha_d d}}$ and $T_d^2 \geq \hat{S}_{1,d}(z) \geq S_{1,d}(z)$, since $S_{1,d}(z)$ cannot become larger when T_d decrease.
Now we show how to find such z and $\alpha_1, \ldots, \alpha_D$. Examine the sum in the polynomial $\hat{S}_{1,d}(z)$

$$\sum_{i=0}^{d} (i+1)^2 (d-i+1) z^i = (d+1) + \sum_{i=1}^{d} (i+1)^2 (d-i+1) z^i.$$

Examine the second part of the sum. We can find a sufficiently small value of $z \in (0,1)$ such that this part is smaller than any value $\epsilon > 0$ for all $d \in [D]$. Now, let $\alpha_d = \frac{1}{2} - \frac{c}{d}$ for some constant $c > 0$. Then

$$z^{d\left(\alpha_d-\frac{1}{2}\right)} = z^{-c}$$

for all $d \in [D]$. Thus, the total value of the sum now is at most $(d+1+\epsilon)z^{-c}$. As $z^{-1} > 1$, take a sufficiently small value of c so that this value is at most $(d+1)^2$.

E Numerical results for $K = 1$

$D = 1$

$T_1 = 1.86793$
$x = 0.464808$
$x_{1,1} = 6.0606$
$x_{1,2} = 0.104715$
$\alpha_{1,1} = 0.317317$

$D = 2$

$T_1 = 1.87788$
$T_2 = 2.76626$
$x = 0.595073$
$x_{1,1} = 5.74769$
$x_{1,2} = 0.12725$
$\alpha_{1,1} = 0.314447$
$\alpha_{1,2} = 0.337219$

$D = 3$

$T_1 = 1.89454$
$T_2 = 2.77944$
$T_3 = 3.68995$
$x = 0.684299$
$x_{1,1} = 5.41613$
$x_{1,2} = 0.146775$
$\alpha_{1,1} = 0.310059$
$\alpha_{1,2} = 0.336865$
$\alpha_{1,3} = 0.351627$

$D = 4$

$$
\begin{aligned}
T_1 &= 1.91039\\
T_2 &= 2.80346\\
T_3 &= 3.7035\\
T_4 &= 4.63207\\
x &= 0.747046\\
x_{1,1} &= 5.11625\\
x_{1,2} &= 0.163892\\
\alpha_{1,1} &= 0.306472\\
\alpha_{1,2} &= 0.335557\\
\alpha_{1,3} &= 0.351929\\
\alpha_{1,4} &= 0.362866
\end{aligned}
$$

$D = 5$

$$
\begin{aligned}
T_1 &= 1.92386\\
T_2 &= 2.828\\
T_3 &= 3.72975\\
T_4 &= 4.64486\\
T_5 &= 5.58737\\
x &= 0.792588\\
x_{1,1} &= 4.8582\\
x_{1,2} &= 0.178964\\
\alpha_{1,1} &= 0.304026\\
\alpha_{1,2} &= 0.334429\\
\alpha_{1,3} &= 0.351624\\
\alpha_{1,4} &= 0.36331\\
\alpha_{1,5} &= 0.371992
\end{aligned}
$$

$D = 6$

$$
\begin{aligned}
T_1 &= 1.93495\\
T_2 &= 2.85009\\
T_3 &= 3.75806\\
T_4 &= 4.6709\\
T_5 &= 5.600\\
T_6 &= 6.55224\\
x &= 0.826544\\
x_{1,1} &= 4.63595\\
x_{1,2} &= 0.192435\\
\alpha_{1,1} &= 0.302631\\
\alpha_{1,2} &= 0.333786\\
\alpha_{1,3} &= 0.351339\\
\alpha_{1,4} &= 0.363364\\
\alpha_{1,5} &= 0.372425\\
\alpha_{1,6} &= 0.379599
\end{aligned}
$$