DISCRETE OPTIMIZATION: A QUANTUM REVOLUTION?

Stefan Cremmers, Luis Fernando Pérez Armas

REPRINT | 3324



UCLouvain

CORE | LIDAM
Louvain Institute of Data Analysis and Modeling in economics and statistics

CORE

Voie du Roman Pays 34, L1.03.01

B-1348 Louvain-la-Neuve

Tel (32 10) 47 43 04

Email:lidam-library@uclouvain.be

https://uclouvain.be/en/research-institutes/lidam/core/core-reprints

European Journal of Operational Research 323 (2025) 378–408

Contents lists available at ScienceDirect

European Journal of Operational Research

journal homepage: www.elsevier.com/locate/eor

Discrete Optimization

Discrete optimization: A quantum revolution?

Stefan Creemers [a,*], Luis Fernando Pérez Armas [b]

[a] Université catholique de Louvain, Center for Operations Research and Econometrics (CORE), Voie du Roman Pays 34, B-1348 Louvain-la-Neuve, Belgium
[b] IESEG School of Management, Univ. Lille, CNRS, UMR 9221 - LEM - Lille Economie Management, F-59000 Lille, France

ARTICLE INFO

Keywords:
Quantum
Computing
Algorithm
Knapsack
Grover

ABSTRACT

We develop several quantum procedures and investigate their potential to solve discrete optimization problems. First, we introduce a binary search procedure and illustrate how it can be used to effectively solve the binary knapsack problem. Next, we introduce two other procedures: a hybrid branch-and-bound procedure that allows to exploit the structure of the problem and a random-ascent procedure that can be used to solve problems that have no clear structure and/or are difficult to solve using traditional methods. We explain how to assess the performance of these procedures and perform an elaborate computational experiment. Our results show that we can match the worst-case performance of the best classical algorithms when solving the binary knapsack problem. After improving and generalizing our procedures, we show that they can be used to solve any discrete optimization problem. To illustrate, we show how to solve the quadratic binary knapsack problem. For this problem, our procedures outperform the best classical algorithms. In addition, we demonstrate that our procedures can be used as heuristics to find (near-) optimal solutions in limited time Not only does our work provide the tools required to explore a myriad of future research directions, it also shows that quantum computing has the potential to revolutionize the field of discrete optimization.

1. Introduction

The idea of quantum computing was first launched in 1980, when Benioff defines a quantum mechanical model of a Turing machine. A few years later, Feynman (1982) states the idea of a universal quantum simulator. Building on the work of Benioff and Feynman, Deutsch (1985) describes the first universal quantum computer that is able to efficiently simulate any other quantum computer. In addition, Deutsch also formulates the Deutsch algorithm, the first quantum algorithm that has a proven speedup when compared to classical algorithms. Later, Deutsch and Jozsa (1992) extend this algorithm. Deutsch and Jozsa, however, consider a problem (the Deutsch–Jozsa problem) that has no practical use, and it takes until 1994 for quantum computing to really take off. In 1994, Shor presents a quantum algorithm to find the prime factors of large integers in polynomial time (whereas the best classical algorithm requires sub-exponential time). In theory, Shor's algorithm can be used to break many of the cryptography schemes in use today. Not surprisingly, the publication of Shor's algorithm sparked an enormous interest in quantum computing. A few years later, Grover (1996) presents a quantum algorithm that achieves a quadratic speedup when performing an unstructured search. Arguably, Grover's algorithm is the most important algorithm in quantum computing today. It can be used to solve a multitude of problems and is the cornerstone of many other quantum algorithms (see, for example, the Quantum Algorithm Zoo[1] that currently lists 64 quantum algorithms, many of which rely on Grover's algorithm). In what follows, we also focus on Grover-based quantum algorithms, and use them to solve discrete optimization problems.

In this paper, we consider discrete optimization problems that assign discrete values to a set of n decision variables in order to optimize some objective value V. Discrete optimization problems can be formulated as follows:

$$\text{Maximize (or minimize)} \quad V(\mathbf{x})$$

$$\text{s.t.} \quad \mathbf{x} \in \Omega$$

Where $\mathbf{x} = \{x_1, \ldots, x_n\}$ is a solution that assigns a discrete value x_i to each decision variable $i : 1 \leq i \leq n$, and Ω is the set of all feasible solutions that respect the logical constraints that define the optimization problem. For most optimization problems, it is easy to evaluate the value and feasibility of a solution $\mathbf{x}$, however, it can be hard to determine an optimal solution $\mathbf{x}^*$ and its solution value V^*. Discrete optimization problems often have an underlying combinatorial structure, hence the link with combinatorial optimization (Parker & Rardin, 1988). Next to combinatorial optimization problems, integer programming problems and constraint programming problems can also

* Corresponding author.
E-mail addresses: stefan.creemers@uclouvain.be (S. Creemers), l.perezarmas@ieseg.fr (L.F.P. Armas).

[1] https://quantumalgorithmzoo.org/

https://doi.org/10.1016/j.ejor.2024.12.016
Received 26 March 2024; Accepted 10 December 2024
Available online 18 December 2024
0377-2217/© 2024 Elsevier B.V. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

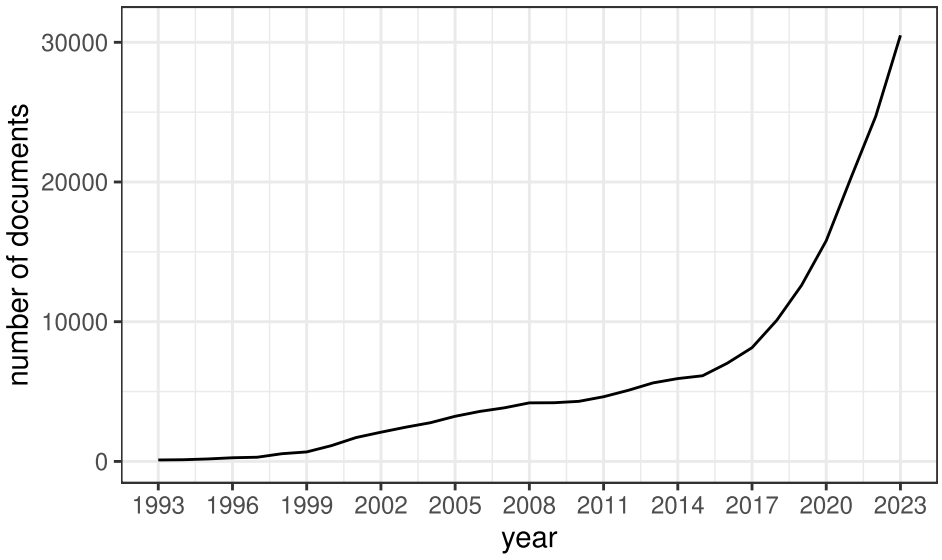


Fig. 1. Number of English documents that have been published according to Google Scholar when conducting a search for keyword "quantum computing".

be seen as discrete optimization problems (refer to, for example, Wolsey (2021) and Campbell et al. (2019)).

Discrete optimization problems are at the core of Operations Research (OR), however, they are also mentioned as one of the four use cases of quantum computing by McKinsey (2021). According to McKinsey, the quadratic speedup offered by Grover's algorithm has potential applications in almost every industry. In addition, McKinsey (2023) estimates that quantum computing has an economic potential of up to $1,270 billion (to be realized by 2035 across four industries: chemicals, life sciences, finance, and automotive). They also show that $5.4 billion has already been invested in quantum computing, with companies such as IBM, Google, Amazon, and Microsoft already offering commercial quantum-computing services. Next to the private sector, governments are also investing heavily in quantum technologies: McKinsey (2023) reports that governments have announced up to $34 billion of investments in quantum technologies. In addition, investments in quantum computing are expected to further increase in the upcoming years. This increasing trend can also be seen in the number of published papers that deal with quantum computing. For instance, Fig. 1 shows the number of English documents that have been published since 1993 according to Google Scholar when conducting a search using the keyword "quantum computing".

Notwithstanding the tremendous interest in quantum computing, the topic has only received limited attention from the OR community (with only few publications in OR journals that mention "quantum computing"). This is somewhat surprising given the potential that has been ascribed to quantum computers to solve complex optimization problems. It is our goal to (further) open up the field of quantum computing to the OR community, and to show its potential to solve discrete optimization problems. For this purpose, we introduce several new procedures, and demonstrate how these procedures can be used to effectively solve the binary knapsack problem. We also explain how to assess the performance of quantum algorithms and show that our procedures can match the worst-case performance of the best classical algorithms when solving the binary knapsack problem. Our procedures, however, can easily be adapted to solve any other discrete optimization problem. To illustrate, we show how our procedures can be used to solve the quadratic binary knapsack problem. For this problem, our procedures outperform the best classical algorithms.

It is beyond the scope of this paper to provide an exhaustive overview of the literature on quantum computing (refer to the Quantum Algorithm Zoo, Nielsen and Chuang (2010), Hidary (2021), and Abbas et al. (2023)) for an overview of the most important results. In addition, even though we briefly touch upon the basics of quantum computing in Appendix A, it is not our goal to provide a comprehensive introduction/tutorial (refer to, for example, Aaronson (2018) for an introductory course and to Nannicini (2020) for a tutorial that does not involve quantum physics). Moreover, in this paper, we do not consider adiabatic quantum computation (see, for example, Albash and Lidar (2018) and Glover et al. (2022) for an introduction). As a result, we do not cover variational quantum algorithms such as the Quantum Approximate Optimization Algorithm (QAOA; see Farhi et al. (2014) for their seminal paper on QAOA, Kurowski et al. (2023) for an example paper that uses QAOA for solving job shop scheduling problems, and Blekos et al. (2023) for a recent review on QAOA and its variants). Instead, we focus on Grover-based algorithms, and provide an OR researcher with the most essential tools that allow to apply quantum computing for optimization purposes. Using these tools, we demonstrate the potential of quantum computing to revolutionize the field of discrete optimization. This revolution, however, is not yet at our doorstep as our results assume the existence of a noiseless universal quantum computer. Even in the most ambitious roadmap, such a computer is not expected to hit the markets any time soon (for a discussion on the development of quantum computers, refer to Preskill (2018), MacQuarrie et al. (2020), Babbush et al. (2021), and Cheng et al. (2023)).

The remainder of this paper is structured as follows. In Section 2, we briefly discuss Grover's algorithm. Next, in Section 3, we explain how Grover's algorithm can be used as a subroutine in a binary search procedure (referred to as BSP). This binary search procedure is used in Section 4 to solve 108000 instances of the binary knapsack problem. As BSP is unable to match the worst-case performance of the best classical algorithms when solving the binary knapsack problem, we also introduce two new procedures: a hybrid branch-and-bound procedure (referred to as HBB) in Section 5 and a random-ascent procedure (referred to as RAP) in Section 6. We show that these procedures can match the worst-case performance of the best classical algorithms when solving the binary knapsack problem. In Section 7, we demonstrate how HBB and RAP can be further improved. In addition, both procedures are generalized in Section 8 such that they can be used to solve any discrete optimization problem. To illustrate, in Section 9, we show how to solve the quadratic binary knapsack problem. Section 10 discusses our results, provides ample directions for future research, and concludes.

2. Grover's algorithm

Grover's algorithm is an iterative procedure that allows to find a target entry in an unstructured database that has N entries using only $\pi/4\sqrt{N}$ iterations. In each iteration, a function $f_{\mathbf{x}}$ is called that verifies whether entry $\mathbf{x}$ is the target entry. In what follows, we assume $N = 2^n$. As a result, vector $\mathbf{x} = \{x_1, \ldots, x_n\}$ can be used to represent entry $\mathbf{x}$, where x_i is a binary variable that corresponds to bit i of entry $\mathbf{x}$. Compared to a classical algorithm, that uses $f_{\mathbf{x}}$ to verify at most $N-1$ entries, Grover's algorithm achieves a quadratic speedup. If there is more than one target entry, Grover's algorithm can be used to find one of the m target entries using $\pi/4\sqrt{m^{-1}2^n}$ iterations, again resulting in a quadratic speedup. As Grover's algorithm is exactly optimal, this quadratic speedup is the best we can hope for when performing an unstructured search on a quantum computer (see also Bennett et al. (1997), Zalka (1999), and Grover and Radhakrishnan (2005)).

A textbook implementation of Grover's algorithm performs $I(n,m)$ iterations to find one of the m target entries in a database that has n entries, where $I(n,m)$ rounds $\pi 4^{-1}\sqrt{m^{-1}2^n}$ to the nearest integer. After performing $I(n,m)$ iterations, we are (almost) certain to "measure" a target entry (i.e., upon observing, the quantum system collapses into a basis state that corresponds to a target entry). For instance, if $n = 3$ and $m = 1$, $\pi 4^{-1}\sqrt{m^{-1}2^n} = 2.22$ and $I(n,m) = 2$. After performing two iterations, we are 94.53 percent certain to measure the target entry (which is less than 100 percent because $\pi 4^{-1}\sqrt{m^{-1}2^n}$ is not an integer; refer to Appendix B for a procedure that can be used to determine the probability to measure a target entry after performing a given number of Grover iterations). If we perform less than $I(n,m)$ Grover iterations, the probability to measure a target entry decreases. For instance, if $n = 3$ and $m = 1$, we only have a 78.13 percent chance to measure the target entry after performing one Grover iteration. Conversely, if we perform more than $I(n,m)$ Grover iterations, the probability to measure a target entry also decreases. For instance, if $n = 3$ and $m = 1$, we only have a 33.01 percent chance to measure a target entry after performing three Grover iterations. In other words, to maximize the probability to measure a target entry, it is important to perform the right number of Grover iterations. In the literature, this problem is also referred to as the "soufflé" problem (see also Brassard (1997); refer to Appendix A for a more detailed discussion of Grover's algorithm in particular and of quantum computing in general).

3. Using Grover's algorithm to solve the binary knapsack problem

In this section, we show how to use Grover's algorithm to find the optimal solution of the binary knapsack problem. The binary knapsack problem is an $\mathcal{NP}$-complete problem that has been studied for more than a century (see, for example, Kellerer et al. (2004) and Cacchiani et al. (2022a) for a review of the literature). The goal of the binary knapsack problem is to maximize the value of a knapsack by adding items from a pool of n items. Each item $i : 1 \leq i \leq n$ has a weight w_i and a value v_i. The weight of the selected items should not exceed W, the maximum weight capacity of the knapsack. The binary knapsack problem can be formulated as:

$$\text{Maximize} \quad \sum_{i=1}^{n} v_i x_i \tag{1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq W \tag{2}$$

$$x_i \in \{0,1\}, \quad \forall\ 1 \leq i \leq n \tag{3}$$

In this formulation, x_i is a binary decision variable that determines whether or not item i has been added to the knapsack, objective function (1) maximizes the value of the knapsack, constraint (2) ensures that the items in the knapsack do not exceed the maximum weight W, and constraint (3) ensures that decision variables are binary. In what follows, we assume $w_i, v_i \in \mathbb{Z}_{\geq 0}$ for all $i : 1 \leq i \leq n$.

To solve the binary knapsack problem on a quantum computer, we have to effectively implement a function $f_{\mathbf{x}}$ that returns 1 if $\mathbf{x}$ is an optimal solution (and 0 otherwise). Given an implementation of $f_{\mathbf{x}}$, we can equip Grover's algorithm with $f_{\mathbf{x}}$, and use it to identify an optimal solution (refer to Appendix C for an implementation of $f_{\mathbf{x}}$ and its use in Grover's algorithm). A solution $\mathbf{x}$ is optimal if it respects the weight constraint and if it has a value of at least V^*:

$$\sum_{i=1}^{n} v_i x_i \geq V^* \tag{4}$$

Unfortunately, however, we do not know V^*. As a result, a procedure is needed to determine V^*. In what follows, we adopt a binary search procedure that evaluates different values V until V^* has been found. For a given value V, we perform $I(n,m)$ Grover iterations to maximize the probability to measure one of the m valid solutions (i.e., a solution $\mathbf{x}$ that respects the weight constraint and has a value of at least V). Again, however, we are faced with a problem: we do not know m; we do not know how many solutions have a value of at least V. If m is unknown, we cannot determine the correct number of Grover iterations, and we may run into the soufflé problem. To resolve this problem, we run Grover's algorithm for different values of $\hat{m}$ until a valid solution is found, where m and $\hat{m}$ denote the real (unknown) and the assumed number of valid solutions, respectively. To sample different values of $\hat{m}$, we first assume $\hat{m} = 2^n$ (i.e., we assume that all solutions are valid) and evaluate $\hat{m} = \hat{m}/2$ in subsequent runs of Grover's algorithm until a valid solution is found (if no valid solution can be found, we conclude that no solution exists that has a value of at least V and that respects the weight constraint). Algorithm 1 (hereafter referred to as GUM) outlines this approach. Even though other approaches have been suggested in the literature (see, for example, Boyer et al. (1996)), GUM is of particular interest because: (1) GUM requires at most $O(\sqrt{m^{-1}2^n})$ calls to function $f_{\mathbf{x}}$, (2) GUM often requires less function calls as a valid solution may be found in early iterations of the procedure, and (3) GUM is almost certain to return a valid solution (if it exists; if no valid solution exists, GUM returns $\emptyset$). For a more detailed discussion on the performance of GUM, refer to Appendix D.

Algorithm 1: Procedure that uses Grover's algorithm to find a valid solution when the number of valid solutions (m) is unknown in a system that has n binary decision variables.

```
procedure GUM(n)
    m̂ = 2^n;
    do
        perform I(n, m̂) iterations of Grover's algorithm equipped
          with f_x;
        measure solution x;
        if f_x = 1 then
            return x;
        else if m̂ > 1 then
            m̂ = m̂/2;
        else
            return ∅;
    while true;
```

Using GUM, we can now introduce the binary search procedure presented in Algorithm 2 (hereafter referred to as BSP). After initializing a minimum knapsack value $V_{\min}$ and a maximum knapsack value $V_{\max}$, BSP evaluates $L = \lceil \log_2(1 + (V_{\max} - V_{\min})) \rceil$ iterations of an outer loop. In each iteration of the outer loop, we initialize a knapsack value V as the largest integer that is smaller-than-or-equal-to the center of the interval that has endpoints $V_{\min}$ and $V_{\max}$. Next, we run GUM to try and measure a valid solution $\mathbf{x}$ for which $f_{\mathbf{x}} = 1$. Procedure GUM performs at most $(n+1)$ iterations of an inner loop to evaluate all values of $\hat{m}$. In each of these iterations, we perform $I(n,\hat{m})$ Grover iterations and we measure solution $\mathbf{x}$. Because the outcome of Grover's algorithm is probabilistic, it is possible that solution $\mathbf{x}$ is not valid (e.g., solution $\mathbf{x}$

may have a weight that exceeds W). In addition, if no valid solution exists, any basis state that we measure will correspond to an invalid solution. Therefore, we need to verify whether $\mathbf{x}$ is valid; we need to verify whether $f_{\mathbf{x}} = 1$. If the validity of solution $\mathbf{x}$ has been verified, GUM returns a valid solution $\mathbf{x}$ and $V_{\min}$ is updated (i.e., $V_{\min} = V+1$). If, on the other hand, GUM does not return a valid solution (either because we incorrectly measure an invalid solution and $f_{\mathbf{x}} = 0$, or because no valid solution was found after processing all values of $\hat{m}$), we update $V_{\max}$ (i.e., $V_{\max} = V - 1$). This process continues until $V_{\min} > V_{\max}$. The last found solution and its value are optimal. Note that, in order to reduce the number of iterations, lower- and upper-bound procedures may be used to initialize $V_{\min}$ and $V_{\max}$.

Algorithm 2: Binary search procedure (BSP) used to solve the binary knapsack problem using Grover's algorithm.

initialize $\mathbf{x}^* = \emptyset$, $V^* = 0$, $V_{\min}$, $V_{\max}$, and start procedure BSP$(n, V_{\min}, V_{\max})$;
procedure BSP$(n, V_{\min}, V_{\max})$
  do
    $V = \lfloor 0.5(V_{\min} + V_{\max}) \rfloor$;
    $\mathbf{x}$ = GUM(n);
    if $\mathbf{x} \neq \emptyset$ **then**
      update optimal solution $\mathbf{x}^* = \mathbf{x}$ and its value $V^* = V$;
      $V_{\min} = V + 1$;
      break;
    else
      $V_{\max} = V - 1$;
      break;
  while $V_{\min} \leq V_{\max}$;

In what follows, we use an example binary knapsack problem to demonstrate the dynamics of BSP. In the example, we have a pool of three items and a knapsack that has maximum weight capacity $W = 4$. The weights and values of the items are listed in Table 1 (the optimal solution of the example knapsack problem is $\mathbf{x}^* = \{1, 0, 1\}$ and has value $V^* = 5$). The steps of BSP are illustrated in Fig. 2 . First, we let $V_{\min} = 0$ and $V_{\max} = \sum_{i=1}^{n} v_i = 6$. Next, we initiate a first iteration of the outer loop, and initialize $V = \lfloor 0.5(V_{\min} + V_{\max}) \rfloor = 3$. In a first iteration of the GUM inner loop, we let $\hat{m} = 2^n = 8$ and perform $I(n, \hat{m}) = 1$ Grover iteration. Given $V = 3$, there are two valid solutions (i.e., solution $\{1, 0, 0\}$ and $\{1, 0, 1\}$). One can verify that, after a single Grover iteration, we have a 50 percent probability to measure $\{1, 0, 0\}$ and a 50 percent probability to measure $\{1, 0, 1\}$. In other words, if we observe the quantum system after one Grover iteration, we are certain to measure a valid solution that has at least value $V = 3$. After verifying that $\{1, 0, 0\}$ (or $\{1, 0, 1\}$) is a valid solution, we let $V_{\min} = V + 1 = 4$, update the best-found solution, and start a second iteration of the outer loop. In this second iteration, we let $V = 5$. Next, we initiate a first iteration of the inner loop, let $\hat{m} = 8$, and again perform $I(n, \hat{m}) = 1$ Grover iteration. Given $V = 5$, however, one can verify that there is only one valid solution (i.e., solution $\{1, 0, 1\}$). If there is only a single valid solution, a single Grover iteration only results in a 78.13 percent chance to measure the target solution. In other words, there is a 21.88 percent chance that we measure a solution that is not valid. This illustrates why every solution measured after using Grover's algorithm needs to be verified. If the validity of the solution has been verified, we update the best-found solution, let $V_{\min} = 6$, and start a third iteration of the outer loop in which $V = 6$ (in this third iteration, no valid solution can be found for any value of $\hat{m}$, and we let $V_{\max} = V - 1 = 5$; as a result, $V_{\min} > V_{\max}$, and the procedure stops). If, on the other hand, no valid solution was identified (with probability 0.2188), we let $\hat{m} = \hat{m}/2 = 4$, and we perform $I(n, \hat{m}) = 1$ Grover iteration. Again, we have a 78.13 percent chance to measure solution $\{1, 0, 1\}$. As such, we have a 17.09 percent chance to end up a situation where we did not measure a valid solution in the first iteration of the inner loop but did measure a valid solution in the second iteration of the inner loop. If a valid solution was measured, we update the best-found solution, let $V_{\min} = 6$, and start a third iteration of the outer loop in which $V = 6$. If no valid solution was measured, we let $\hat{m} = 2$, and perform $I(n, \hat{m}) = 2$ Grover iterations. With 2 iterations, we have a 94.53 percent chance to identify a valid solution. If, however, we still fail to identify a valid solution, we let $\hat{m} = 1$, and again perform $I(n, \hat{m}) = 2$ Grover iterations. In total, we are 99.99 percent certain to measure $\{1, 0, 1\}$ as a valid solution during the second iteration of the outer loop. There is, however, a 0.0143 percent probability to measure the wrong basis state. If this happens, verification of the solution fails, we cannot reduce $\hat{m}$ any further, and we let $V_{\max} = V - 1 = 4$ (i.e., we have wrongfully concluded that $V = 5$ has no valid solution; this illustrates why it is possible that we are unable to identify an optimal solution if Grover's algorithm is used in an optimization procedure). At this point, we enter a third iteration of the outer loop with $V = 4$ (here again, we have a 99.99 percent probability to identify $\{1, 0, 1\}$ as the only valid solution that has value $V \geq 4$, after which we update $V_{\min}$ such that $V_{\min} > V_{\max}$, and the procedure stops; conversely, we have a 0.0143 percent probability to identify an invalid solution, after which we update $V_{\max}$ such that $V_{\min} > V_{\max}$, and the procedure also stops). Even though we are almost certain to find the optimal solution, we also require at least 8 Grover iterations (and at most 13). In addition, we need to verify the validity of at least 6 knapsack solutions (and at most 9). For each Grover iteration, we actually require two calls to function $f_{\mathbf{x}}$ (one call to function $f_{\mathbf{x}}$ and one call to reverse function $f_{\mathbf{x}}^{\dagger}$ to "uncompute"; refer to Appendix C for additional details). Each validation also requires a call to function $f_{\mathbf{x}}$. Therefore, to solve the example knapsack problem using BSP, we need at least 22 calls to function $f_{\mathbf{x}}$, and at most 35 (see also Appendix E). Note that a brute-force classical approach would require us to evaluate only 8 knapsack solutions.

Table 1
Data for example knapsack problem with $n = 3$ items.

i	w_i	v_i
1	2	3
2	3	1
3	2	2
$W = 4$		

4. Performance of BSP

In the previous section, we have seen that the quadratic speedup of Grover's algorithm does not necessarily result in a speedup when solving discrete optimization problems. To further investigate whether BSP can offer a speedup when compared to classical algorithms, we first have a look at the complexity of BSP. To solve a discrete optimization problem that has n binary decision variables and optimal objective value V^*, BSP performs L iterations of an outer loop, where L is a logarithmic function of the bounds on V^*. For each iteration of the outer loop, BSP performs up to $(n + 1)$ runs of Grover's algorithm. For each run $i : 0 \leq i \leq n$, we perform $I(n, 2^{(n-i)})$ Grover iterations. Each Grover iteration requires two calls to function $f_{\mathbf{x}}$. In addition, after each run of Grover's algorithm, we need to verify whether the measured solution is valid, resulting in another call to function $f_{\mathbf{x}}$. In total, BSP requires at most $L \sum_{i=0}^{n} \left(1 + 2I(n, 2^{(n-i)})\right) \approx O(L\sqrt{2^n})$ calls to $f_{\mathbf{x}}$. If we consider the binary knapsack problem, function $f_{\mathbf{x}}$ requires $O(n)$ operations. As a result, BSP requires at most $O(Ln\sqrt{2^n})$ operations to solve a binary knapsack problem that has n items. In practice, however, BSP may run much faster because not all $(n + 1)$ runs of Grover's algorithm may be required (i.e., a valid solution may be measured in run $i < (n + 1)$; see also the discussion on the performance of GUM in Appendix D).

To verify the practical runtime of BSP, we solve a set of 108000 instances of the binary knapsack problem. We use the instance generator of Pisinger (2005) to generate six sets of 1000 problems for each value of $n : n \in \{3, 4, \ldots, 19, 20\}$:

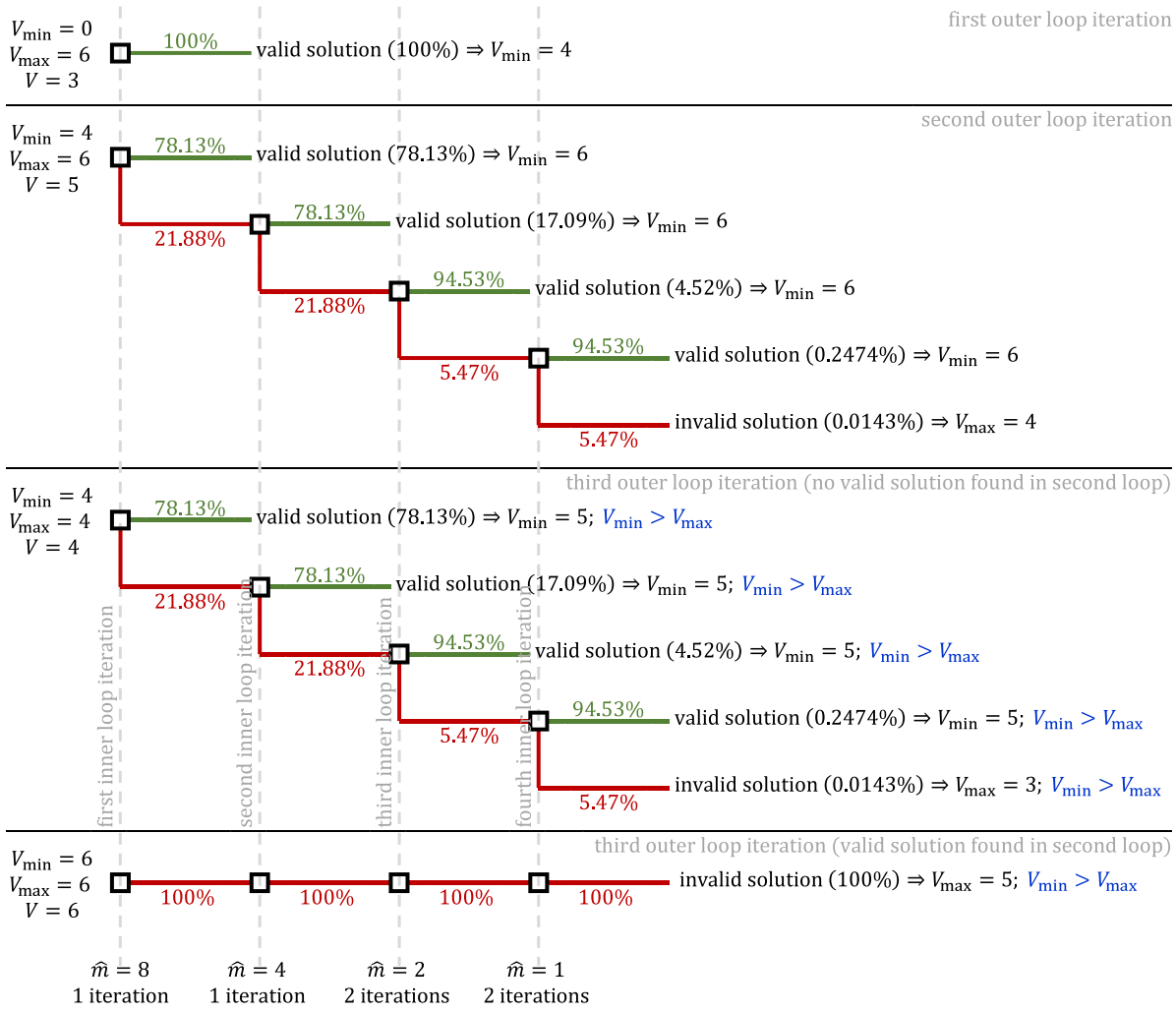


Fig. 2. Illustration of the dynamics of procedure BSP for the example knapsack problem. In each node (represented by a square), we perform 1 to 2 Grover iterations (depending on the value of $\hat{m}$), we measure the basis state, and we verify whether the corresponding solution is valid. If the solution is valid (indicated in green), $V_{\min}$ is updated. If, on the other hand, the solution is not valid (indicated in red), we update $\hat{m}$ and/or $V_{\max}$ (if $\hat{m} = 1$). The procedure stops if $V_{\min} > V_{\max}$ (indicated in blue). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

- Set 1 has uncorrelated values and weights that range from 1 to 1000.
- Set 2 has strongly correlated values and weights that range from 1 to 1000.
- Set 3 has weights that range from 1 to 1000 and values $v_i = 3\lceil w_i/3 \rceil$ for each item i (so-called "profit-ceiling instances").
- Set 4 has uncorrelated values and weights that range from 1 to 1 000 000.
- Set 5 has strongly correlated values and weights that range from 1 to 1 000 000.
- Set 6 has weights that range from 1 to 1 000 000 and values $v_i = 3\lceil w_i/3 \rceil$ for each item i.

The "hardness" of each set is determined by the type of instances, and by the range of values and weights. Instances that have uncorrelated values and weights are considered to be very easy to solve. Instances that have strongly correlated values and weights are considered to be hard to solve. The so-called "profit-ceiling" instances are considered very hard to solve. A large range of weights and values also makes instances more difficult to solve.

We use a Discrete-Time Markov Chain (DTMC) to assess the performance of BSP when solving the 108000 instances (see Appendix E for a detailed discussion on how to evaluate the performance of a quantum algorithm). The results of the experiment are summarized in Table 2 and Fig. 3. Table 2 reports the minimum probability to find an optimal solution for each problem set and for each value of $n : 3 \leq n \leq 20$. As expected, BSP can be used with confidence to solve all but the smallest binary knapsack problems. Fig. 3, on the other hand, shows the expected worst-case number of function calls as well as the expected average-case number of function calls when solving 1000 binary knapsack problems for each problem set and for each value of n. From Fig. 3, we can conclude that: (1) the type of problem instance only has limited impact on the performance of the procedure (a discussion on the slight differences among the different problem sets is available in Section 7), (2) the range of the values has a logarithmic effect on the expected number of function calls, and (3) the expected average-case number of function calls is much smaller than the expected worst-case number of function calls. To further explore the practical performance of BSP, we return to Table 2 that reports the expected worst-case number of function calls as well as the expected average-case number of function calls divided by $L\sqrt{2^n}$. It seems that, in the worst case, BSP requires $\approx 5L\sqrt{2^n}$ calls to function $f_{\mathbf{x}}$. On average, however, only $\approx 3L\sqrt{2^n}$ calls are required. Even though this is a significant reduction in the number of function calls, this still implies that BSP requires (even on average) more operations than the best classical algorithms that require at most $O(nW)$ or $O(n\sqrt{2^n})$ operations (see also (Kellerer et al., 2004)). As such, BSP is unable to match the worst-case performance of the best classical algorithms. In what follows, we consider that the

Table 2
Minimum probability to find an optimal solution, expected worst-case, and expected average-case number of function calls divided by $L\sqrt{2^n}$ when using procedure BSP for solving 1000 binary knapsack problems for each problem set and for each value of $n : 3 \leq n \leq 20$.

n	minimum probability						$\mathrm{E}\left[\mathrm{WCcalls}/L\sqrt{2^n}\right]$						$\mathrm{E}\left[\mathrm{calls}/L\sqrt{2^n}\right]$					
	set 1	set 2	set 3	set 4	set 5	set 6	set 1	set 2	set 3	set 4	set 5	set 6	set 1	set 2	set 3	set 4	set 5	set 6
3	0.91	0.93	0.93	0.93	0.93	0.95	5.41	5.40	5.41	5.53	5.53	5.53	3.13	3.35	3.24	3.33	3.43	3.40
4	0.97	0.99	0.97	0.99	0.99	0.99	5.50	5.50	5.50	5.61	5.61	5.61	3.18	3.34	3.29	3.38	3.46	3.45
5	0.98	0.98	0.98	0.98	0.98	0.98	5.40	5.44	5.41	5.52	5.54	5.52	3.13	3.32	3.27	3.39	3.50	3.47
6	0.99	0.99	0.99	0.99	0.99	0.99	5.40	5.42	5.40	5.51	5.52	5.51	3.21	3.35	3.30	3.48	3.56	3.53
7	0.99	0.99	0.99	0.99	0.99	0.99	5.45	5.42	5.45	5.54	5.53	5.54	3.29	3.38	3.32	3.54	3.61	3.58
8	0.99	1.00	1.00	0.99	0.99	0.99	5.47	5.43	5.47	5.56	5.54	5.56	3.30	3.40	3.30	3.61	3.65	3.62
9	1.00	1.00	1.00	1.00	1.00	1.00	5.41	5.42	5.41	5.52	5.52	5.52	3.28	3.39	3.23	3.60	3.65	3.58
10	1.00	1.00	1.00	1.00	1.00	1.00	5.35	5.40	5.35	5.45	5.48	5.45	3.22	3.37	3.16	3.58	3.65	3.53
11	1.00	1.00	1.00	1.00	1.00	1.00	5.34	5.42	5.34	5.44	5.48	5.44	3.20	3.36	3.12	3.57	3.62	3.52
12	1.00	1.00	1.00	1.00	1.00	1.00	5.32	5.37	5.32	5.40	5.43	5.40	3.22	3.34	3.08	3.54	3.60	3.46
13	1.00	1.00	1.00	1.00	1.00	1.00	5.31	5.30	5.32	5.38	5.39	5.38	3.19	3.29	3.04	3.51	3.59	3.43
14	1.00	1.00	1.00	1.00	1.00	1.00	5.30	5.24	5.30	5.37	5.34	5.37	3.19	3.22	2.97	3.51	3.54	3.39
15	1.00	1.00	1.00	1.00	1.00	1.00	5.27	5.19	5.27	5.34	5.30	5.34	3.16	3.18	2.94	3.49	3.50	3.35
16	1.00	1.00	1.00	1.00	1.00	1.00	5.23	5.16	5.23	5.31	5.27	5.31	3.12	3.14	2.88	3.46	3.46	3.26
17	1.00	1.00	1.00	1.00	1.00	1.00	5.18	5.15	5.18	5.29	5.25	5.29	3.09	3.09	2.82	3.46	3.45	3.22
18	1.00	1.00	1.00	1.00	1.00	1.00	5.15	5.16	5.15	5.26	5.25	5.26	3.03	3.08	2.78	3.41	3.43	3.16
19	1.00	1.00	1.00	1.00	1.00	1.00	5.14	5.18	5.14	5.24	5.26	5.24	3.03	3.07	2.74	3.40	3.41	3.12
20	1.00	1.00	1.00	1.00	1.00	1.00	5.14	5.20	5.14	5.24	5.27	5.24	3.00	3.08	2.71	3.40	3.43	3.07

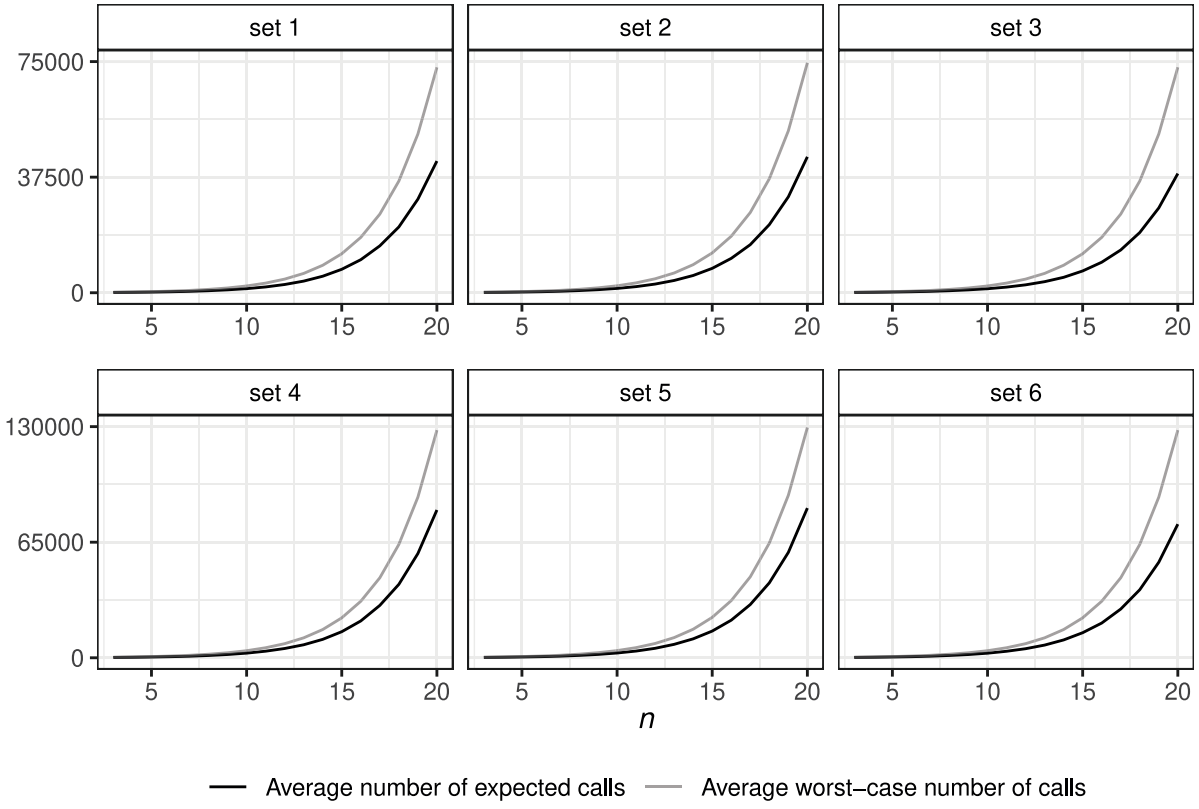


Fig. 3. Comparison of expected worst-case number of function calls (gray) and expected average-case number of function calls (black) when using procedure BSP for solving 1000 binary knapsack problems for each problem set and for each value of $n : 3 \leq n \leq 20$. Note that instances of sets 4, 5, and 6 (that have values ranging from 1 to 1 000 000) require twice as many operations to solve as instances of sets 1, 2, and 3 (that have values ranging from 1 to 1000); indicating a logarithmic effect.

best classical algorithms require $O(n\sqrt{2^n})$ operations (i.e., we assume that W is sufficiently large such that $O(nW) > O(n\sqrt{2^n})$).

5. Hybrid branch-and-bound procedure

In this section, we introduce a new hybrid branch-and-bound procedure (hereafter referred to as HBB) that, in contrast to BSP, is able to match the worst-case performance of the best classical algorithms when solving the binary knapsack problem. An outline of HBB is given by Algorithm 3. Procedure HBB uses a recursive depth-first strategy to search a tree that has n levels. At each level $i : 1 \leq i \leq n$, we verify whether item i can be added to the current (partial) solution $\mathbf{x}_{1(i-1)} = \{x_1, \ldots, x_{(i-1)}\}$ that has value $V = \sum_{j=1}^{(i-1)} x_j v_j$ and remaining weight capacity $R = W - \sum_{j=1}^{(i-1)} x_j w_j$. If the weight of item i does not exceed the remaining weight capacity, we add item i (i.e., we let $x_i = 1$) and let $\mathbf{x}_{1i} = \{x_1, \ldots, x_{(i-1)}\} \cup \{x_i\}$, $V' = V + v_i$, and $R' = R - w_i$. In addition, if $V' > V^*$, we also update the best-found solution and its value. Next, we use GUM to assess whether a better solution can still be found at a lower level of the search tree. If such a solution $\mathbf{x}_{(i+1)n}$ can be found, we update

the best-found solution and enter a new recursion in which item i is added. After (potentially) exploring the branch in which we add item i, we also explore the branch in which we do not add item i (i.e., in which $x_i = 0$). At a given level of the tree, ω items remain to be added. For small ω, however, it is possible that GUM (that relies on Grover's algorithm) is unable to measure a valid solution (even if that solution exists). In addition, for small ω, the quadratic speedup (and hence the computational gain) obtained by Grover's algorithm is limited. Therefore, we propose a "hybrid" procedure that uses: (1) GUM if more than ω items remain, and (2) a classical approach if ω items remain. If ω items remain, the classical approach determines the set of remaining items in Ω_{in} (i.e., the set of all partial solutions that consider items $j \geq i$) that best improves the current (partial) solution without violating the maximum weight constraint. If such a solution ($\mathbf{x}^\star_{in}$; with value $V^\star_{in}$ and weight $W^\star_{in}$) can be found, we update the best-found solution. A hybrid approach has already been proposed by, for example, Ambainis et al. (2018) who describe an algorithm that is dedicated to solving the traveling salesman problem. Procedure HBB is also similar to the backtracking algorithm of Montanaro (2016), however, the procedure of Montanaro is not hybrid, and uses a quantum counting algorithm to determine whether a valid solution exists at a deeper level of the tree. As has been shown by Creemers and Pérez (2023), however, quantum counting algorithms are dominated by GUM when verifying whether a valid solution exists. Later, Montanaro (2020) has used his backtracking algorithm in a branch-and-bound procedure to solve the Sherrington-Kirkpatrick model. Other (non-hybrid) branch-and-bound procedures have been proposed by, Markevich and Trushechkin (2019), who find solutions for the traveling salesman problem that do not exceed a given length, Ambainis and Kokainis (2022), who improve the backtracking algorithm of Montanaro, and Chakrabarti et al. (2022), who obtain a universal speedup for branch-and-bound, and who perform numerical experiments for solving the Sherrington-Kirkpatrick model, the maximum independent set problem, and the mean–variance portfolio-optimization problem..

Algorithm 3: Hybrid branch-and-bound procedure to solve the binary knapsack problem using GUM.

initialize $\mathbf{x}^* = \emptyset$, $V^* = 0$, and start procedure HBB$(\emptyset, 0, W, 1)$;
Procedure HBB$(\mathbf{x}_{1(i-1)}, V, R, i)$
  if $n - i < \omega$ **then**
    use classical algorithm to find $\mathbf{x}^\star_{in} \in \Omega_{in}$ that has highest value $V^\star_{in} > V^* - V$ and $W^\star_{in} \leq R$;
    if $\mathbf{x}^\star_{in} \neq \emptyset$ **then**
      update best-found solution $\mathbf{x}^* = \mathbf{x}_{1(i-1)} \cup \mathbf{x}^\star_{in}$ and its value $V^* = V + V^\star_{in}$;
  else
    if $w_i \leq R$ **then**
      Let $x_i = 1$, $\mathbf{x}_{1i} = \mathbf{x}_{1(i-1)} \cup \{x_i\}$, $V' = V + v_i$, and $R' = R - w_i$;
      if $V' > V^*$ **then**
        Update best-found solution $\mathbf{x}^* = \mathbf{x}_{1i} \cup \{\mathbf{0}\}$ and its value $V^* = V'$;
      Use GUM$(n-i)$ to measure $\mathbf{x}_{(i+1)n} \in \Omega_{(i+1)n}$ that has value $V_{(i+1)n} > V^* - V'$ and $W_{(i+1)n} \leq R'$;
      if $\mathbf{x}_{(i+1)n} \neq \emptyset$ **then**
        update best-found solution $\mathbf{x}^* = \mathbf{x}_{1i} \cup \mathbf{x}_{(i+1)n}$ and its value $V^* = V' + V_{(i+1)n}$;
        HBB$(\mathbf{x}_{1i}, V', R', i+1)$;
    Let $x_i = 0$, $\mathbf{x}_{1i} = \mathbf{x}_{1(i-1)} \cup \{x_i\}$, and run HBB$(\mathbf{x}_{1i}, V, R, i+1)$;

To further illustrate the dynamics of HBB, we use HBB to solve the example binary knapsack problem. The steps of the procedure are illustrated in Fig. 4. In this example, we assume $\omega = 1$. To initialize the procedure, we let $\mathbf{x}^* = \emptyset$, $V^* = 0$, and enter a first recursion with current solution $\mathbf{x} = \emptyset$, value $V = 0$, remaining weight capacity $R = W = 4$, and $i = 1$. In this first recursion, a classical procedure should not be used (i.e., $n - i \geq \omega$), and sufficient weight remains for item $i = 1$ to be added (i.e., $w_1 \leq R$). As a result, we update the current partial solution, and let $x_1 = 1$, $\mathbf{x}_{11} = \{1\}$, $V' = 3$, and $R' = 2$. We also update the best-found solution and let $\mathbf{x}^* = \{1, 0, 0\}$ and $V^* = 3$. Next, we use GUM to measure a solution $\mathbf{x}_{(i+1)n} \in \Omega_{(i+1)n}$ that has a value $V_{(i+1)n}$ larger than $V^* - V'$ (i.e., a value larger than 0) and a weight $W_{(i+1)n}$ that does not exceed R'. After adding item 1, two items are left, and $\mathbf{x}_{(i+1)n} = \{0, 1\}$ is the only valid solution (i.e., $\mathbf{x}^*$ can only be improved if we add item 3). In a system that has two items and only one valid solution, GUM has a 100 percent probability to measure the valid solution after performing $I(2, 4) = 1$ Grover iteration. As a result, after calling GUM, we update the best-found solution, and let $\mathbf{x}^* = \{1, 0, 1\}$ and $V^* = 5$. After updating the best-found solution, we enter a second recursion. In this recursion, $(n-i)$ is still smaller than ω, and as such, we do not use a classical procedure to try and improve the current solution. We are, however, also unable to add item 2 to the existing solution (that already contains item 1; $w_2 > R$). As a result, we enter a new recursion without adding item 2. In this recursion, $(n - i) < \omega$, and we use a classical procedure to determine whether a partial solution can still be found that allows to improve the best-found solution. As the optimal solution has already been found, no improvement is possible, and we exit this recursion. After ascending to recursion depth 2, we again ascend one level, and end up at recursion level 1, in which we enter a new recursion where item 1 is not added. In this recursion, the current value equals $V = 0$ and the remaining weight capacity is $R = 4$. As a result, item 2 can be added, we update the current solution, and use GUM to measure a solution $\mathbf{x}_{(i+1)n} \in \Omega_{(i+1)n}$ that has a value bigger than $V^* - V' = 4$. No such solution can be found, and therefore, after performing two Grover iterations (one for $\hat{m} = 2$ and one for $\hat{m} = 1$), we complete the procedure with a 100 percent probability to find the optimal solution. In total, to find the optimal solution, HBB requires 1 Grover iteration in which a solution of two items is evaluated and 2 Grover iterations in which a solution of a single item is evaluated (in contrast to BSP which requires at most 13 Grover iterations in which a solution of three items is evaluated; note that it requires more operations to evaluate a solution that has more items).

In the worst-case, HBB expects to perform $O(n\sqrt{2^n})$ operations when solving the binary knapsack problem (thereby matching the worst-case performance of the best classical algorithms; refer to Appendix F for a detailed discussion on the complexity of HBB). To assess the average performance of HBB, we use Monte Carlo simulation to simulate the execution of HBB 1000 times for each of the problem instances defined in the previous section for $n : n \in \{10, 11, \ldots, 19, 20\}$ (resulting in 66000 instances of the binary knapsack problem). In the experiment, we let $\omega = 5$ (i.e., if 5 items remain, a classical approach is used to determine which partial solution best improves the current solution; a value $\omega = 5$ is chosen because Grover's algorithm is sufficiently accurate when at least 6 items are considered). Note that, in the discussion of the results that follows, we omit the (insignificant) number of operations required by the classical approach. The results of the experiment are presented in Fig. 5. Fig. 5 shows the expected number operations divided by $n\sqrt{2^n}$ (denoted $E[\eta/n\sqrt{2^n}]$; note that we divide by $n\sqrt{2^n}$ to show that HBB requires $\approx \sqrt{2^n}$ operations) when solving the binary knapsack problems using HBB (center panel). For comparison, we also include the performance of BSP (left panel). It is obvious that HBB significantly outperforms BSP. In addition, it seems that HBB can solve instances that have a larger range of values (i.e., instances of sets 4, 5, and 6) without extra effort (this in contrast to BSP that uses a binary search procedure to identify the optimal value; a larger range of values results in more operations). For both procedures, however, we observe (slight) differences in the required number of operations to solve instances of different sets. To further explore these differences, we refer to Table 3 that reports the expected number of times that a feasible solution value occurs over all instances in a set. For example, for set 3, Table 3 shows that a feasible solution value is expected to occur 252.58 times for an

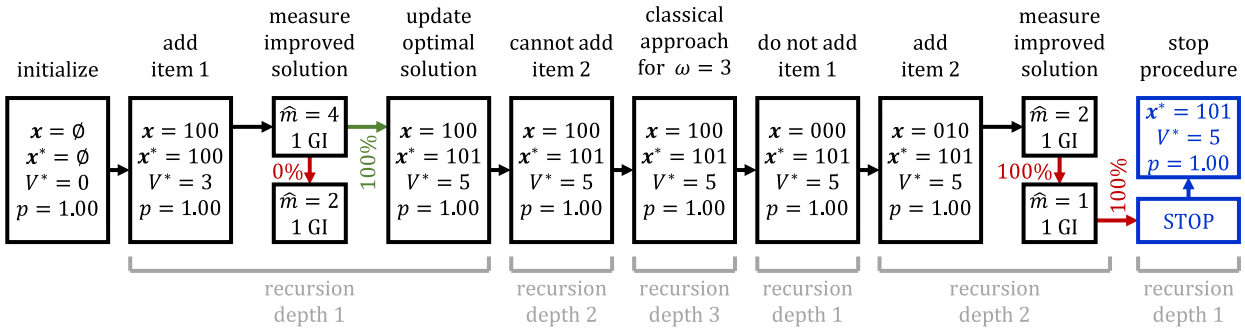


Fig. 4. Illustration of the dynamics of procedure HBB when solving the example binary knapsack problem.

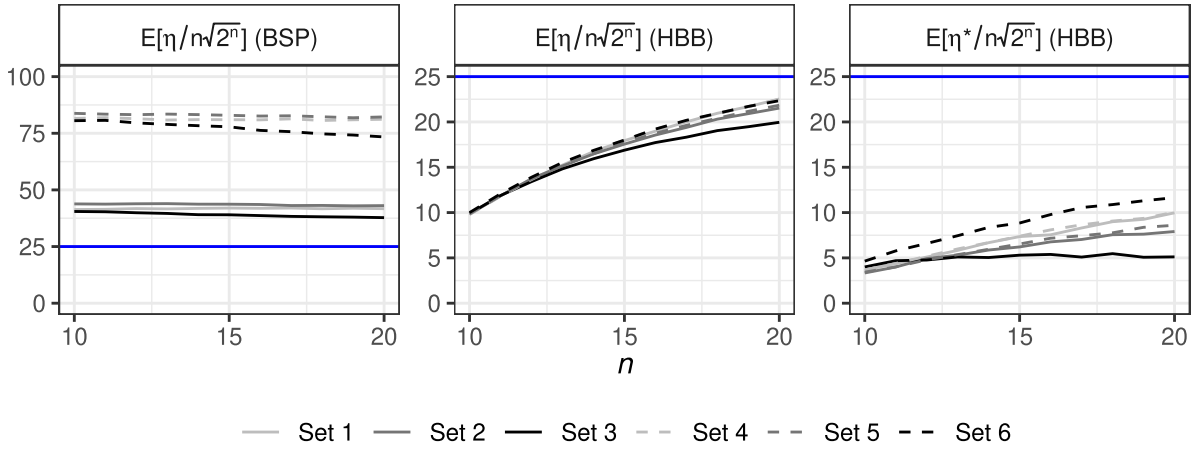


Fig. 5. Expected number of operations divided by $n\sqrt{2^n}$ required to solve a binary knapsack problem using procedures BSP and HBB, as well as the expected number of operations divided by $n\sqrt{2^n}$ required to find an optimal solution for the first time using HBB, when solving 1000 binary knapsack problems for each problem set and for each value of $n : 10 \leq n \leq 20$ (lines of sets 1 and 4 are light gray, sets 2 and 5 are dark gray, sets 3 and 6 are black, sets 1, 2, and 3 are full, and sets 4, 5, 6 are dashed). The blue line indicates a value of 25 on the vertical axis. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

instance that has $n = 20$ items (i.e., given a feasible solution value V, we expect to find 252.58 feasible solutions that have the same value V). In contrast, for set 4, a feasible solution value only is expected to occur 1.06 times for an instance that has $n = 20$ items. If more solutions have a particular value V, more solutions can be removed from the search space if GUM can show that a solution exists that has a value higher than V (i.e., it is easier to find the best solution in a set of solutions that have 2 values that occur 500 times than in a set of solutions that have 1000 values that only occur once). As a result, for BSP we see small differences between sets 1 and 2, and between sets 4, 5, and 6 (with slightly increasing differences for instances of set 6 if n is large). For HBB, that is not impacted by an increase in the range of values, we see almost no difference between sets 1 and 4 (instances that have uncorrelated weights/values), and between sets 2 and 5 (instances that have correlated weights/values). There is, however, a slight difference between instances that have uncorrelated weights/values (sets 1 and 4) and instances that have correlated weights/values (sets 2 and 5). This illustrates that, rather than the range of values, the structure of the instance can have an impact on the performance of HBB. We also observe that uncorrelated/correlated instances (sets 1, 2, 4, and 5) differ from profit-ceiling instances that are both the most easy to solve (set 3) and the most difficult to solve (set 6). Here again, the instances of set 3 are easiest to solve because they have a significantly higher frequency of feasible solution values. Set 6, on the other hand, may be the hardest to solve because the branch-and-bound procedure is less capable of exploiting the structure of profit-ceiling instances (when compared to uncorrelated/correlated instances, profit-ceiling instances have items that have a ratio of value over weight that is always close to 1). Notwithstanding the differences among sets, it appears that, for all sets, the expected number of operations (divided by $n\sqrt{2^n}$) converges to some constant (note that this also has been verified for some sets for values of n larger than 20). Therefore, we conclude that HBB expects to perform $\approx n\sqrt{2^n}$ operations. Last but not least, Fig. 5 also reports the expected number of operations required to find an optimal solution for the first time (denoted $E[\eta^*/n\sqrt{2^n}]$; right panel). We observe that HBB can find an optimal solution for the first time using $\approx n\sqrt{2^n}$ operations. The difference between the right and center panel, however, also shows that, once an optimal solution has been found, a significant amount of time is spent to prove that this solution is optimal (i.e., in the remaining nodes of the search tree, GUM needs to evaluate all values of $\hat{m}$ in order to be sure that no valid solution can be found that improves the optimal solution). This seems to be the main bottleneck of HBB. In the upcoming section, we try to resolve this issue, and present a procedure that minimizes the time that is spent to show that a solution is optimal.

6. Random-ascent procedure

In the previous section, we presented a hybrid branch-and-bound procedure (HBB) that is able to solve a binary knapsack problem that has n items using at most $O(n\sqrt{2^n})$ operations. Most of these operations, however, are spent on proving that the optimal solution cannot be improved. To minimize the time spent on proving that a solution is optimal, we propose another quantum algorithm. This simple, yet elegant, procedure requires only one call to GUM once an optimal solution has been found. The procedure can be seen as a random-ascent procedure that achieves a random improvement of the best-found solution value in each iteration. In what follows, we use this procedure to solve the binary knapsack problem. Algorithm 4 (hereafter referred to as RAP)

Table 3
Average expected number of times that a feasible solution value occurs for each problem set and for each value of $n : 10 \leq n \leq 20$.

n	set 1	set 2	set 3	set 4	set 5	set 6
10	1.10	1.10	1.41	1.00	1.00	1.00
11	1.19	1.20	1.81	1.00	1.00	1.00
12	1.38	1.39	2.61	1.00	1.00	1.00
13	1.75	1.78	4.13	1.00	1.00	1.00
14	2.49	2.57	6.96	1.00	1.00	1.00
15	3.92	4.10	12.17	1.00	1.00	1.01
16	6.60	6.95	21.78	1.00	1.01	1.02
17	11.54	12.20	39.62	1.01	1.01	1.03
18	20.69	21.92	72.88	1.02	1.02	1.06
19	37.72	40.03	135.20	1.03	1.03	1.12
20	69.57	73.95	252.58	1.06	1.07	1.25

provides an outline of the procedure. After initializing $x^* = \emptyset$ and $V^* = 0$, RAP enters a first iteration in which GUM is used to measure a solution that has a value $V' > V^*$ and weight $W' \leq W$. If such a solution is found, we update the best-found solution, and start a new iteration. This process continues until an optimal solution has been found. If we are lucky, we find an optimal solution after one iteration. If we are unlucky, it may take $S \leq 2^n$ iterations (where S is the number of feasible solution values). In general, however, we expect to find an optimal solution after $(\lceil \log_2(S) \rceil + 1)$ iterations. In addition, in the first iterations, we may expect GUM to find a solution using few Grover iterations (as there are still many solutions that can improve the best-found solution at that point). In later iterations (if few valid solutions remain), GUM has to evaluate more (increasingly smaller) values of $\hat{m}$ that require (increasingly larger) number of Grover iterations. If, eventually, an optimal solution has been found, GUM is called one last time to verify that no solution can be found that still improves the optimal solution.

Algorithm 4: Random-ascent procedure to solve the binary knapsack problem using GUM.

initialize $\mathbf{x}^* = \emptyset$, $V^* = 0$, and start procedure RAP(n);
Procedure RAP(n)
  do
    Use GUM(n) to measure $\mathbf{x}'$ that has value $V' > V^*$ and $W' \leq W$;
    if $\mathbf{x}' \neq \emptyset$ **then**
      update best-found solution $\mathbf{x}^* = \mathbf{x}'$ and its value $V^* = V'$;
  while $\mathbf{x}' \neq \emptyset$;

To further illustrate the dynamics of RAP, we use it to solve the example binary knapsack problem. The steps of the procedure are illustrated in Fig. 6. After initializing, we use GUM to measure any of the four feasible solutions that have a value $V > 0$. There is a 6.25 percent probability that no valid solution was measured (leading us to conclude that there is no feasible solution). There is, however, also a 93.75 percent probability that a valid solution was measured. In this case, there are 4 options. First, we may measure the optimal solution itself (with a probability of 25 percent). After measuring the optimal solution, we perform one more run of GUM to show that no solution can still be found that improves the optimal solution. Second, we may measure solution $\mathbf{x} = \{1, 0, 0\}$ that has $V = 3$. In this case, we perform another iteration in which we run GUM again and have a 99.99 percent probability to measure the only solution that can still improve a value of $V = 3$ (i.e., the optimal solution that has $V = 5$). After measuring the optimal solution, we perform one more run of GUM to verify that the optimal solution has been found. Third, we may measure solution $\mathbf{x} = \{0, 0, 1\}$ that has $V = 2$. In this case, we have a 100 percent probability to measure one of two solutions that have a value $V > 2$. If the optimal solution is measured, we run GUM to show that no more improvement can be made. If, on the other hand, we measure solution $\mathbf{x} = \{1, 0, 0\}$, we need to run GUM again to measure the optimal solution and once more to prove that there is no solution that can still improve the optimal solution. Fourth, we may measure solution $\mathbf{x} = \{0, 0, 1\}$ that has $V = 1$. In this case, all the aforementioned scenarios are possible after running GUM again. In the end, RAP has a 93.20 percent probability to measure the optimal solution (see also the blue boxes in Fig. 6).

In the worst case, there are $S = 2^n$ feasible solution values, and we expect to perform $\lceil \log_2(S) \rceil + 1 = (n + 1)$ iterations of GUM. In each iteration $i : 0 \leq i \leq n$, we expect that $m = 2^n/2^i = 2^{(n-i)}$ feasible solution values remain, and that GUM performs $\sum_{j=0}^{n}(1 - \phi_{i-1})P(n, 2^{(n-i)}, 2^{(n-j)})I(n, 2^{(n-j)})$ Grover iterations, where ϕ_i is the probability of having measured a valid solution after i GUM iterations and $P(n, m, I)$ is the probability to measure one of the m valid solutions in a set of 2^n solutions after performing I Grover iterations (see Appendix B for a simple procedure to calculate $P(n, m, I)$). ϕ_i may be obtained through the following recursive relationship:

$$\phi_i = \phi_{i-1} + (1 - \phi_{i-1})P(n, m, I(n, 2^{(n-i)})), \tag{5}$$

where $\phi_{-1} = 0$. We expect to find an optimal solution after $(n+1)$ iterations of GUM, after which we call GUM one last time to prove that the optimal solution cannot be improved. In this last run of GUM, we require $\sum_{j=0}^{n} I(n, 2^{(n-j)})$ Grover iterations. As explained earlier, each Grover iteration requires two calls to a function $f_{\mathbf{x}}$. In addition, we expect to validate at most $(n+2)$ measured solutions, requiring another $(n+2)$ calls to function $f_{\mathbf{x}}$. Therefore, in the worst case, we expect that RAP requires $\sum_{j=0}^{n} 2I(n, 2^{(n-j)}) + (n + 2) + \sum_{i=0}^{n}\sum_{j=0}^{n}\left(1 - \phi_{i-1}\right) P(n, 2^{(n-i)}, I(n, 2^{(n-j)}))$ $2I(n, 2^{(n-j)})$ calls to function $f_{\mathbf{x}}$. Fig. 7 illustrates the worst-case number of expected calls required by RAP divided by $n\sqrt{2^n}$ for $n : 1 \leq n \leq 60$. We observe that the worst-case number of expected calls converges to $13.72n\sqrt{2^n}$, implying that: (1) RAP has worst-case expected runtime $O(\mu\sqrt{2^n})$ for solving any discrete optimization problem that has n binary decision variables (where μ is the number of operations required by $f_{\mathbf{x}}$), and (2) RAP can solve the binary knapsack problem using $O(n\sqrt{2^n})$ operations (in the worst case; thereby matching the performance of the best classical algorithms).

To verify the effective runtime of RAP, we use Monte Carlo simulation to simulate its execution 1000 times for each of the aforementioned 66000 problem instances. The results are summarized in Fig. 8 that shows the expected number of operations (divided by $n\sqrt{2^n}$) when solving the binary knapsack problems using HBB and RAP (top left and right panels, respectively). We observe that RAP easily outperforms HBB, and that its performance remains well below its worst-case performance (indicated by the red line in the right top panel). Fig. 8 also shows the expected number of operations (divided by $n\sqrt{2^n}$) that are required to find an optimal solution for the first time. Here also, we see that RAP (right bottom panel) outperforms HBB (left bottom panel). Before we conclude, however, that RAP dominates HBB, we first investigate whether the performance of both procedures can be improved.

7. Improving procedures HBB and RAP

Both HBB and RAP can be improved in several ways. First, a heuristic solution procedure can be used to provide a better initial solution (resulting in less recursions/iterations to be performed by both procedures, respectively). Second, GUM lies at the core of the performance of HBB and RAP. Therefore, if we can improve GUM, we also improve HBB and RAP. Note that, as the true number of valid solutions (i.e., m) is unknown, GUM takes a sample of assumed numbers of valid solutions (i.e., $\hat{m}$) by initializing $\hat{m}$ as the total number of solutions, and by letting $\hat{m} = \hat{m}/2$ in subsequent iterations. If no solution is found after evaluating $\hat{m} = 1$, the procedure stops. Better sampling methods, however, may exist (see, for example, Boyer et al. (1996) for an alternative sampling method). In this section, we investigate alternative sampling methods, and generalize GUM by: (1) dividing $\hat{m}$ by κ rather than by 2, and (2) stopping the procedure once $\hat{m} \leq \tau$. Fig. 9

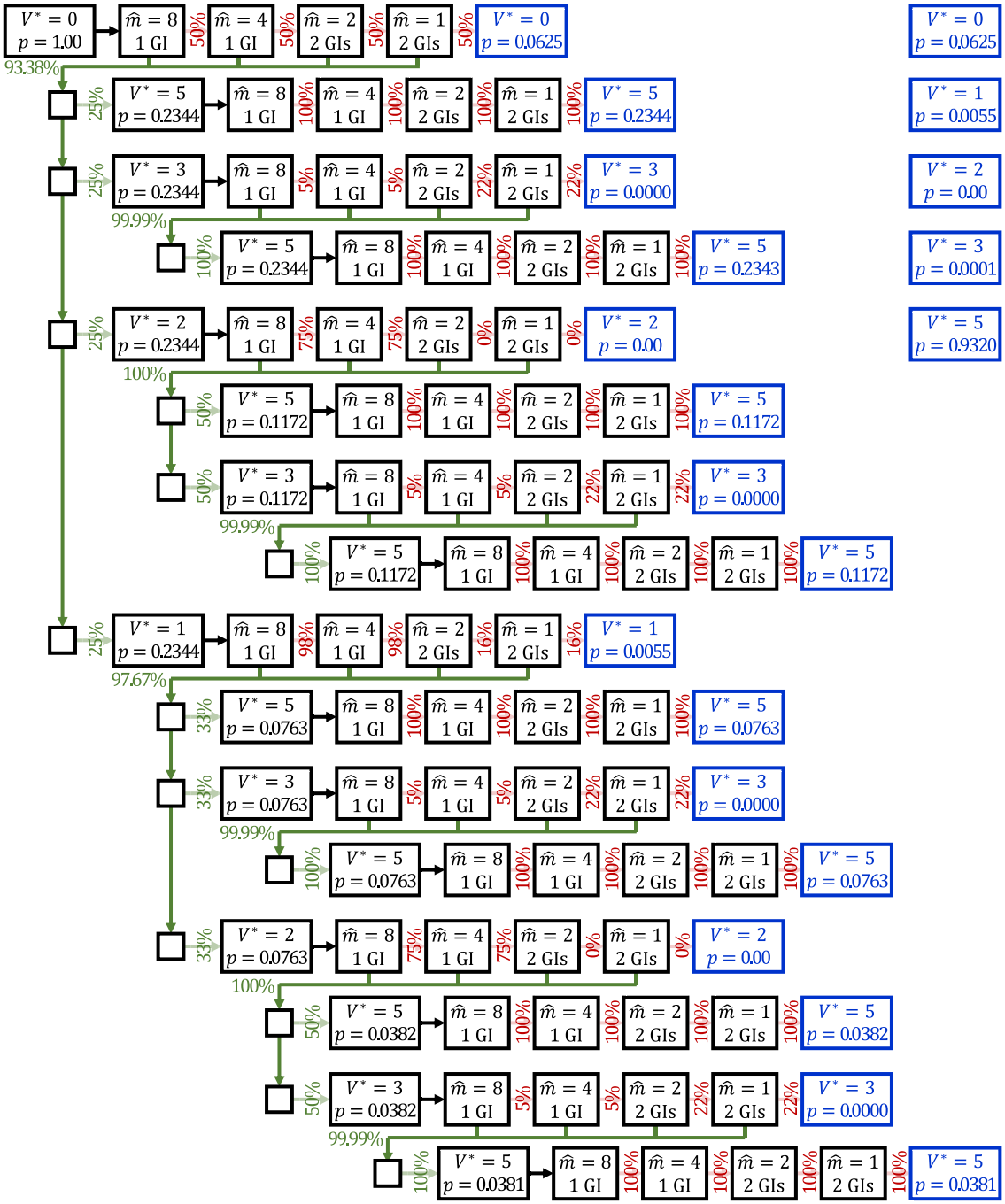


Fig. 6. Illustration of the dynamics of RAP when solving the example binary knapsack problem.

illustrates the sample of values of $\hat{m}$ (and the corresponding number of Grover iterations) that is used in a system that has $n = 6$ items and $m = 6$ valid solutions: (1) in case of regular GUM (left column), (2) if we let $\tau = 4$ (middle column), and (3) if we let $\kappa = 4$ (right column). For instance, if $\tau = 4$, we do not evaluate values $\hat{m} = 2$ and $\hat{m} = 1$. If, on the other hand, $\kappa = 4$, we skip $\hat{m} = 32$, $\hat{m} = 8$, and $\hat{m} = 2$. If less values of $\hat{m}$ are evaluated, less Grover iterations are required, however, it also becomes less likely that we measure a valid solution (in the three cases, after evaluating all values of $\hat{m}$, we have a probability of 1.00, 1.00, and 0.99 to measure one of the $m = 6$ valid solutions, respectively). In addition, as another improvement, note that, currently, HBB and RAP do not use any information of past runs of GUM. If a solution was found assuming a particular value of $\hat{m}$, however, it makes sense to assume that in a subsequent (more restricted) search there are at most $\hat{m}$ valid solutions (i.e., it does not make sense to once more reinitialize $\hat{m}$ as the total number of solutions; we can use the value of $\hat{m}$ for which a solution was found in a previous search to start searching in a subsequent, more restricted search). These ideas are reflected in Algorithm 5 (referred to as GGUM in what follows) that generalizes GUM.

Next to these (potential) improvements, HBB may also benefit from classical techniques to improve the performance of a branch-and-bound procedure. For instance, when solving the binary knapsack problem, items can be sorted by their ratio of value over weight. In addition, Algorithm 6 can be used to calculate an upper bound on the maximum value that can still be obtained in any node at level i (hereafter referred to as UBi; see also Dantzig (1957)). If this upper bound does not exceed

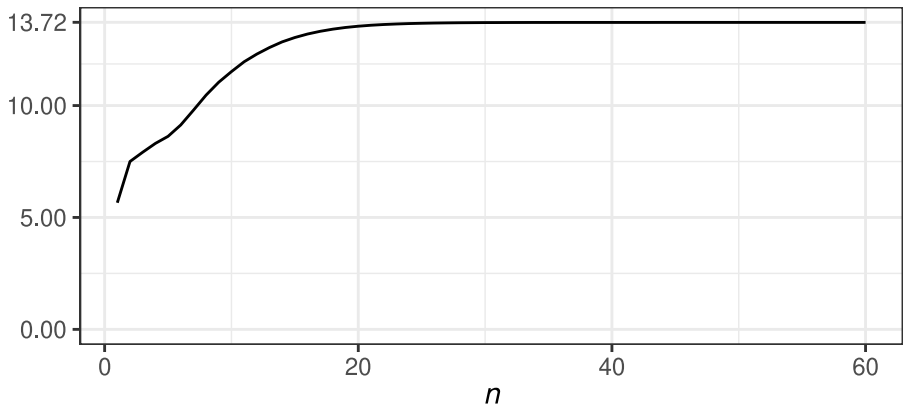


Fig. 7. Worst-case expected number of operations divided by $n\sqrt{2^n}$ required by procedure RAP for each value of $n : 1 \leq n \leq 60$.

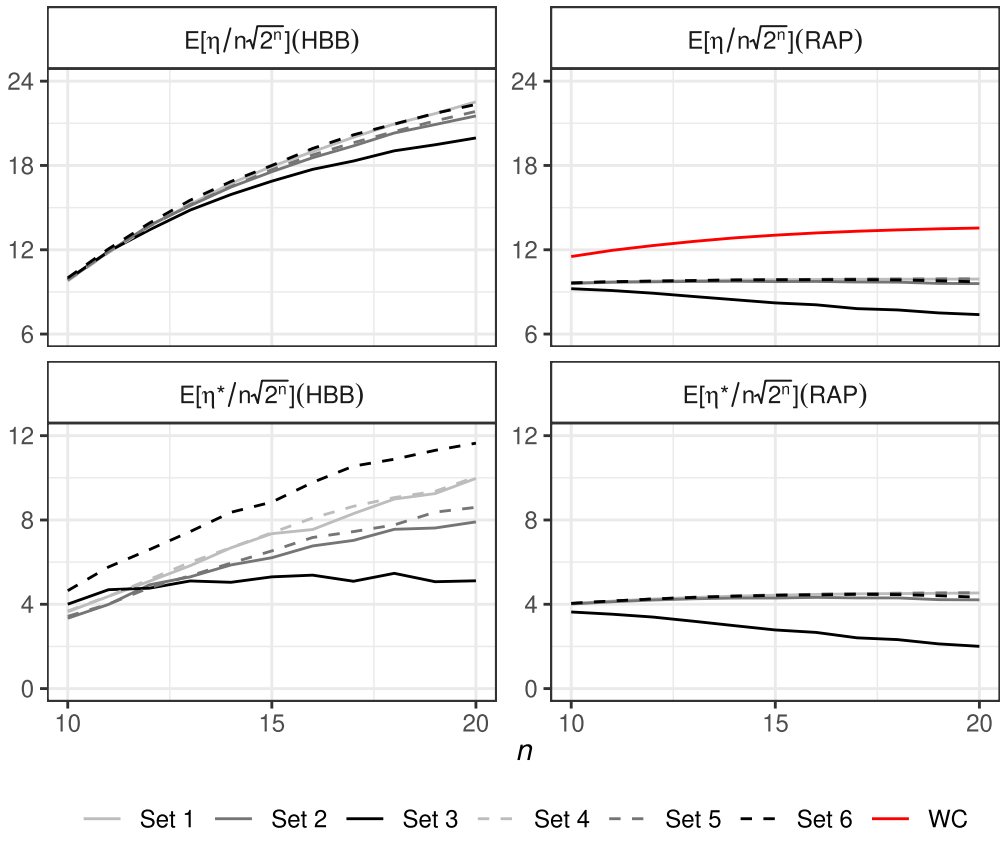


Fig. 8. Expected number of operations divided by $n\sqrt{2^n}$ required to solve a binary knapsack problem using HBB and RAP, as well as expected number of operations divided by $n\sqrt{2^n}$ required to find an optimal solution for the first time using both procedures, when solving 1000 binary knapsack problems for each problem set and for each value of $n : 10 \leq n \leq 20$ (lines of sets 1 and 4 are light gray, sets 2 and 5 are dark gray, sets 3 and 6 are black, sets 1, 2, and 3 are full, and sets 4, 5, 6 are dashed). The worst-case expected performance of procedure RAP is indicated by the red line (see also Fig. 7). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the best-found solution, we can fathom the node. Note that, in contrast to HBB, RAP cannot benefit from this upper-bound procedure.

In what follows, we perform a computational experiment in which we solve 66000 instances of the binary knapsack problem, and assess the impact of:

- using information of the past run of GGUM to determine the initial value of $\hat{m}$ in the subsequent run of GGUM (referred to as the "base case"; all improvements mentioned below depart from this base case).
- using a heuristic procedure to calculate a lower bound that serves as an initial solution. The heuristic procedure first sorts items based on their ratio of value over weight (higher ratios are ranked first). Next, we initialize an empty knapsack, and add items in order of the list. If an item cannot be added (because its weight exceeds the remaining weight capacity), we move to the next item in the list. The procedure stops if all items in the list have been processed.
- assuming a value of $\tau = 4$ (i.e., $\hat{m} = 1$ and $\hat{m} = 2$ are not evaluated by GGUM).

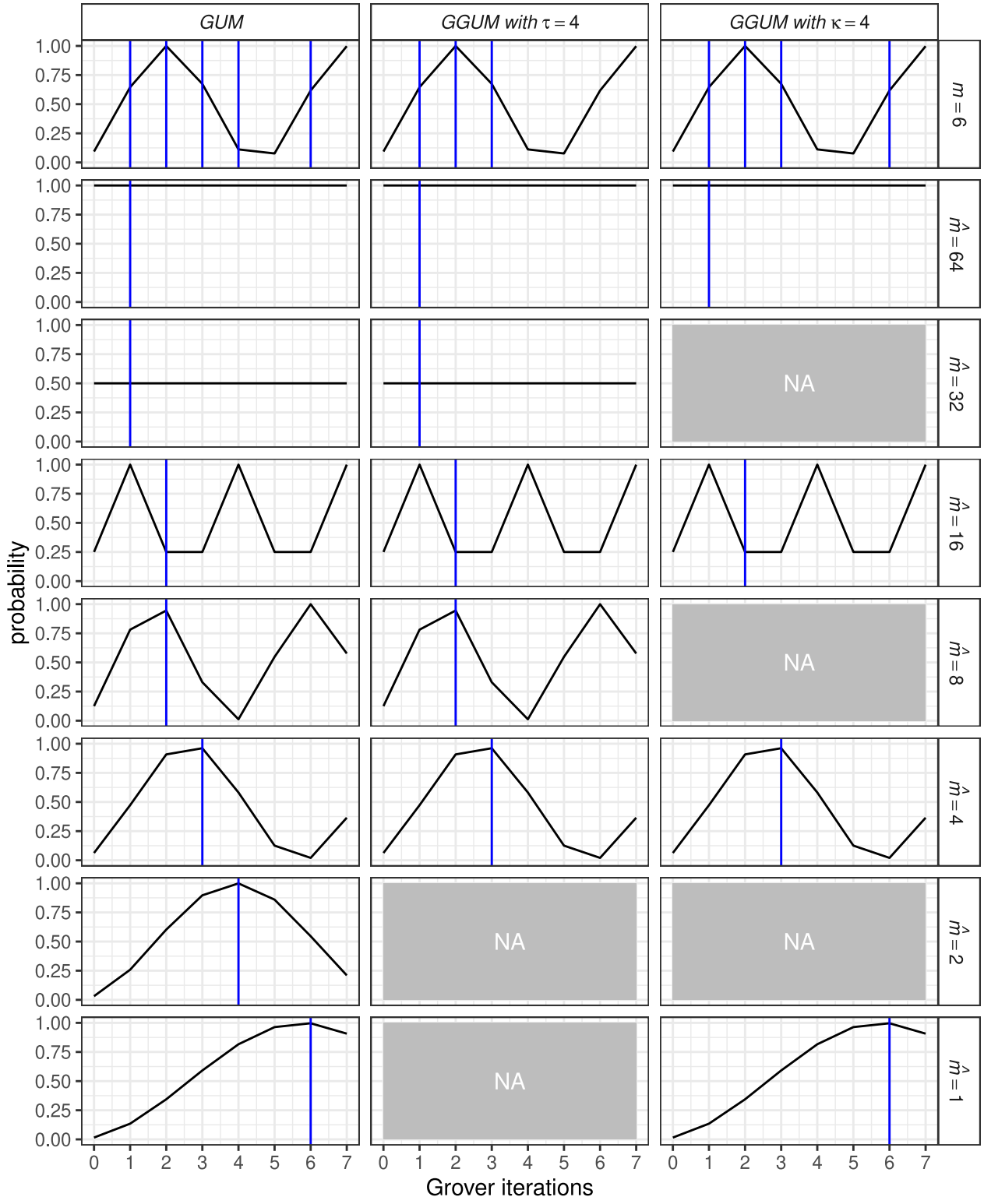


Fig. 9. Sample of values of $\hat{m}$ (and corresponding number of Grover iterations) that are used in case of regular GUM, GGUM with $\tau = 4$, and GGUM with $\kappa = 4$. Each graph presents the probability to measure a valid solution in a system that has $n = 6$ items if $m = 6$ (top row), $\hat{m} = 64$ (second row), $\hat{m} = 32$ (third row), $\hat{m} = 16$ (fourth row), $\hat{m} = 8$ (fifth row), $\hat{m} = 4$ (sixth row), $\hat{m} = 2$ (seventh row), and $\hat{m} = 1$ (eight row). In rows 2–8, the blue line represents the number of Grover iterations given by $I(n, \hat{m})$. In the top row, the blue lines correspond to the sample of values of $\hat{m}$ used by each approach to measure any of the $m = 6$ valid solutions. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Algorithm 5: Generalized procedure that uses Grover's algorithm to find a valid solution when the number of valid solutions is unknown in a system that has n binary decision variables, and where $f_{\mathbf{x}}$ returns 1 if solution $\mathbf{x}$ is valid (and 0 otherwise).

```
procedure GGUM(n, m̂)
    do
        perform I(n, m̂) iterations of Grover's algorithm equipped
          with f_x;
        measure solution x;
        if f_x = 1 then
            return x and m̂;
        else if m̂ > τ then
            m̂ = m̂/κ;
        else
            return ∅;
    while true;
```

Algorithm 6: Procedure that calculates an upper bound on the value of the current knapsack solution (that has value V and remaining weight capacity R) in a node at level i of the search tree. Note that we assume that items are sorted by their ratio of value over weight.

```
procedure UBi(V, R, i)
    for j = i to n do
        if w_j ≤ R then
            let V = V + v_j and R = R − w_j;
        else
            let V = V + (R/w_j) v_j;
            break;
    return V;
```

- assuming a value of $\kappa = 3$ (i.e., we evaluate $\hat{m} = 2^n$, $\hat{m} = 2^n/3$, $\hat{m} = 2^n/9$, etc.).
- using upper bound UBi to fathom nodes when using HBB.

The results of the experiment are summarized in Fig. 10. For different scenarios, Fig. 10 reports the expected number of operations (divided by $n\sqrt{2^n}$) required by HBB and RAP (left two columns) as well as the expected number of operations (divided by $n\sqrt{2^n}$) required to find an optimal solution for the first time (right two columns). Fig. 10 also shows the average probability to find an optimal solution for each of the scenarios/procedures (red line in the left two columns; note that the expected number of operations to find an optimal solution for the first time only is calculated for those instances for which an optimal solution was found). For comparison, we also included the results of HBB and RAP without improvements (first row; these results are identical to those presented in Fig. 8).

A first conclusion that can be drawn by observing Fig. 10 is that initializing $\hat{m}$ as the last value of $\hat{m}$ in the previous run (base case; reported in row 2) has a significant impact on the number of required operations without significantly reducing the probability to measure an optimal solution. Next, even though the use of a lower bound (row 3) has a positive impact, the effect is limited except for sets 1 and 4 (where items have uncorrelated values and weights). This can be explained by the fact that items in sets 2 and 5 have similar ratios (values and weights are correlated) and items in sets 3 and 6 have ratios close to 1 (item i has value $v_i = 3\lceil w_i/3 \rceil$). As a result, for these sets, our lower bound (based on the ratio of value over weight) may not be very effective. In row 4, on the other hand, we see that letting $\tau = 4$ once more has a significant effect on the number of required iterations without impacting the probability to measure an optimal solution too much. In contrast, in row 5, we evaluate the impact of letting $\kappa = 3$ for RAP (second and fourth column). Even though we are able to reduce the number of required operations, we also observe that the probability to measure an optimal solution reduces as n increases. In other words, with $\kappa = 3$, the sample of values of $\hat{m}$ becomes too sparse to still guarantee that GGUM can still measure a valid (and hence optimal) solution. In row 5, we also illustrate the impact of upper bound UBi on the performance of HBB. We observe a significant reduction of the number of operations required to find an optimal solution (first column) and the number of operations required to find an optimal solution for the first time (third column). The impact is most outspoken for instances where the structure of the problem can be exploited (i.e., instances with uncorrelated and correlated values and weights). For sets 3 and 6, the impact is less clear. For instances in set 6, the effectiveness of UBi is limited by the fact that items have a ratio of value over weight that is close to 1. For set 3, this effect is offset by the fact that instances in set 3 have feasible solutions values that have a high frequency (see also Table 3). Last but not least, in row 6 we evaluate the impact of all improvements (except $\kappa = 3$ and upper bound UBi in the case of RAP; referred to as "full case" in Fig. 10) on the performance of HBB and RAP. Here we observe that both procedures have roughly the same probability to find an optimal solution, however, RAP no longer dominates HBB. Only when solving instances of set 6, RAP still has the upper hand (i.e., RAP may only be able to outperform HBB if HBB cannot exploit the structure of the problem). Perhaps more importantly, row 6 also illustrates that HBB and RAP may require (far) less than $n\sqrt{2^n}$ operations to find an optimal solution for the first time. This indicates that they may be used as heuristics to obtain (near-) optimal solutions by observing the best-found solution after a given time.

8. Generalizing procedures HBB and RAP

Until now, we have considered discrete optimization problems that have binary decision variables. Procedures HBB and RAP, however, can easily be adapted to solve discrete optimization problems that consider discrete decision variables. If we assume that a decision variable can take on 2^b values, we require b bits to represent the decision variable (i.e., in the case of binary decision variables, $b = 1$, and a single bit suffices to represent a decision variable). If each of the n decision variables requires b bits, function $f_{\mathbf{x}}$ evaluates a solution that has nb bits rather than to n bits. In other words, there are 2^{nb} possible solutions, and GUM will evaluate up to $\log_2(2^{nb}) = nb$ values of $\hat{m}$. For each value of $\hat{m}$, GUM performs $I(2^{nb}, \hat{m})$ Grover iterations (that each require two calls to function $f_{\mathbf{x}}$) and a single call to function $f_{\mathbf{x}}$ to verify the validity of the measured solution. As a result, if we assume it takes μ operations to evaluate a solution, HBB and RAP require at most $O(\mu\sqrt{2^{nb}})$ operations.

9. Solving other discrete optimization problems

Procedures BSP, HBB, and RAP can be used to solve any discrete optimization problem. To illustrate, we show how to solve the quadratic binary knapsack problem. The quadratic binary knapsack problem is a straightforward generalization of the binary knapsack problem that can be formulated as:

$$\text{Maximize} \quad \sum_{i=1}^{n}\sum_{i=j}^{n} v_{ij} x_i x_j \tag{6}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq W \tag{7}$$

$$x_i \in \{0,1\}, \quad \forall\ 1 \leq i \leq n \tag{8}$$

Where v_{ii} is the value obtained if item i is added to the knapsack and v_{ij} is the value obtained if both items i and j are added to the knapsack. In this formulation, objective function (6) maximizes the value of the

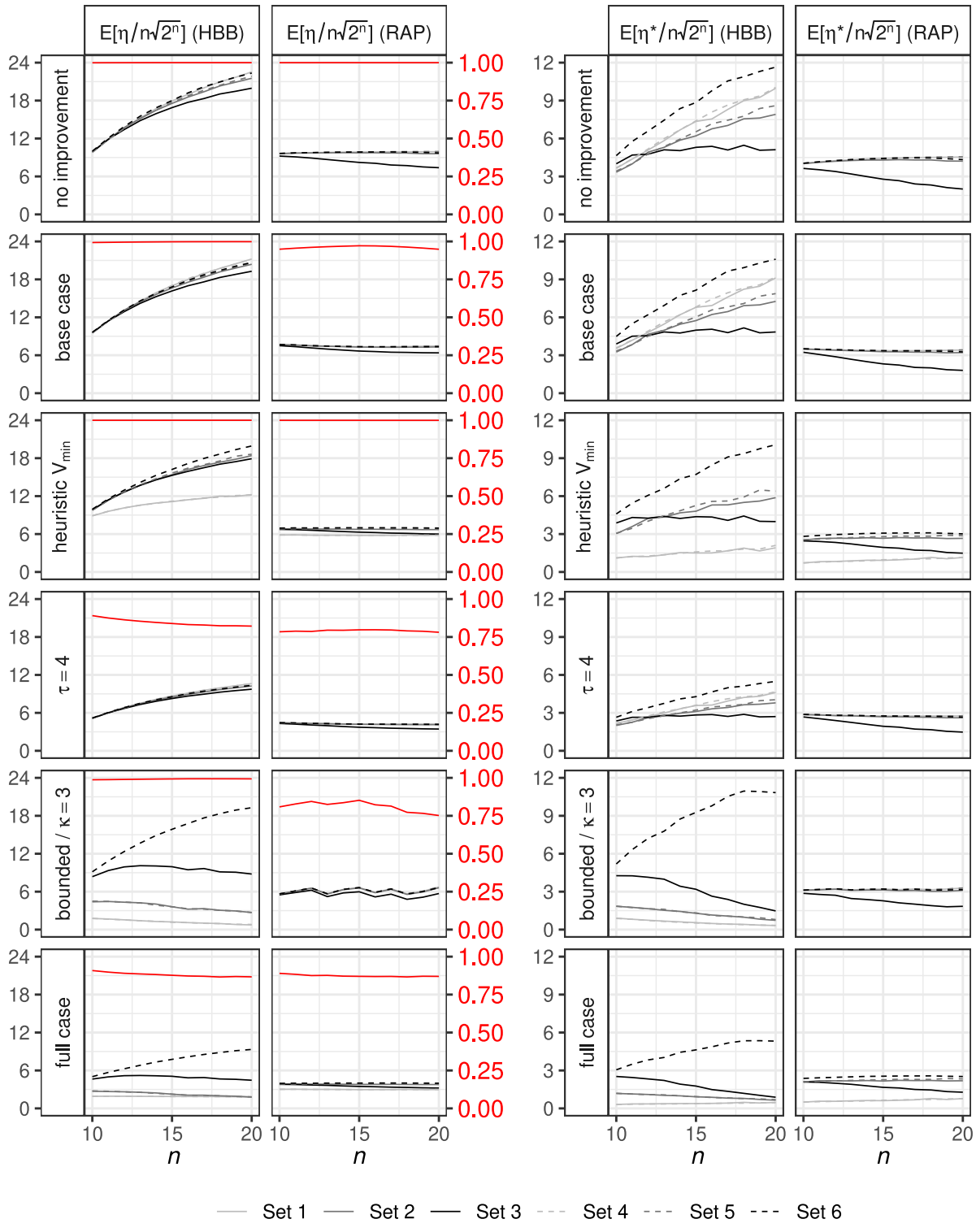


Fig. 10. Expected operations (divided by $n\sqrt{2^n}$) to solve a binary knapsack problem using HBB and RAP as well as the expected operations (divided by $n\sqrt{2^n}$) to find an optimal solution for the first time, when solving 1000 knapsack problems for each problem set, for various values of n, and for various scenarios (lines of sets 1 and 4 are light gray, sets 2 and 5 are dark gray, sets 3 and 6 are black, sets 1, 2, and 3 are full, and sets 4, 5, 6 are dashed). A red line shows the average probability to identify an optimal solution. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

knapsack and constraints (7)–(8) ensure that: (1) the weight capacity of the knapsack is not exceeded and (2) decision variables are binary. The quadratic binary knapsack problem is strongly $\mathcal{NP}$-complete and is considered to be much harder to solve than the regular binary knapsack problem (that is only weakly $\mathcal{NP}$-complete). The problem has been widely studied and has applications in, for instance, the optimization of portfolios and the location of facilities (see also Pisinger (2007), Cacchiani et al. (2022b), and Fennich et al. (2024) for a review of the relevant literature). The best exact methods adopt branch-and-bound algorithms and linearize the objective function using, for instance,

Lagrangian relaxation. Even though the complexity of these algorithms is difficult to assess, we can assume that they require far more than $O(n\sqrt{2^n})$ operations (i.e., the worst-case number of operations required to solve the regular binary knapsack problem).

To solve the (quadratic) binary knapsack problem, procedures HBB and RAP require $O(\mu\sqrt{2^n})$ operations, where $O(\mu)$ corresponds to the number of operations required by a function $f_{\mathbf{x}}$ that evaluates whether solution $\mathbf{x}$ is valid. If we solve a regular binary knapsack problem, function $f_{\mathbf{x}}$ loops over each item (i.e., $O(\mu) = O(n)$). If, on the other hand, we solve a quadratic binary knapsack problem, $f_{\mathbf{x}}$ loops over each pair of items (i.e., $O(\mu) = O(n^2)$). In other words, procedures HBB and RAP require $O^*(\sqrt{2^n})$ operations when solving both regular as well as quadratic binary knapsack problems (i.e., a quadratic objective function does not result in a significant increase in the number of operations required by HBB and RAP). As a result, if we assume that the best classical algorithms require far more operations when solving a quadratic binary knapsack problem, we can conclude that they are be outperformed by procedures HBB and RAP.

Note that, in contrast to classical algorithms that solve the quadratic programming problem, procedures HBB and RAP do not have to linearize the objective function. In fact, regardless of whether objective function and/or constraints are linear, HBB and RAP require $O(\mu\sqrt{2^n})$ operations (in case of binary decision variables) or $O(\mu\sqrt{2^{nb}})$ operations (in case 2^b discrete values can be assigned to each of the n decision variables). In other words, as long as $O(\mu) < O(\sqrt{2^{nb}})$, the performance of HBB and RAP is determined by the number of decision variables and the number of discrete values that can be assigned to each decision variable (i.e., by nb). Therefore, HBB and RAP benefit from compact formulations that, if possible, minimize nb by increasing the number of operations that are performed by function $f_{\mathbf{x}}$ (i.e., the goal is to minimize nb at the cost of μ such that $O(\mu) < O(\sqrt{2^{nb}})$.

10. Conclusions

In this paper, we investigate the potential of quantum computing to solve discrete optimization problems using Grover-based algorithms. Arguably, Grover's algorithm is the most important quantum algorithm. It can be used to perform an unstructured search and achieves a quadratic speedup when compared to classical algorithms. As a search algorithm, it allows to solve discrete optimization problems by identifying valid solutions (i.e., feasible solutions that have an objective value of at least V; assuming we consider a maximization problem). To identify one of the m valid solutions in a set of 2^n solutions, Grover's algorithm needs $O(\sqrt{m^{-1}2^n})$ calls to a function $f_{\mathbf{x}}$ that returns 1 if solution $\mathbf{x}$ is valid (and 0 otherwise).

When using Grover's algorithm to solve discrete optimization problems, we essentially face two problems: (1) we do not know the number of valid solutions that have a value of at least V, and (2) we do not know the optimal solution value V^*. To resolve the first problem, we use a procedure (GUM) that samples different values of $\hat{m}$ (i.e., the assumed number of valid solutions) and that is almost certain to return a valid solution (if it exists). We show that GUM requires at most $O(\sqrt{2^n})$ calls to function $f_{\mathbf{x}}$ if we are solving a discrete optimization problem that has n binary decision variables. To resolve the second problem, we first adopt a binary search procedure (BSP) that evaluates L values objective values V (where L is a logarithmic function of the bounds on the optimal solution value V^*). Next, with both problems resolved, we use BSP and illustrate how to effectively solve the binary knapsack problem. We show how to assess the performance of BSP, and perform a computational experiment in which 108000 binary knapsack problems are solved. Our results indicate that BSP requires $O(nL\sqrt{2^n})$ operations. The best classical algorithms, however, require $O(n\sqrt{2^n})$ operations. As such, BSP is unable to match the worst-case performance of the best classical algorithms when solving the binary knapsack problem.

To match the worst-case performance of the best classical algorithms, we propose two new procedures that also rely on GUM to find a valid solution: a hybrid branch-and-bound procedure (HBB) and a random-ascent procedure (RAP). To solve any discrete optimization problem that has n decision variables, HBB uses a depth-first search strategy to explore a tree that has n levels. At level $i : 1 \leq i \leq n$, we visit nodes that represent partial solutions where values have been assigned to decision variables $j < i$. If we are currently in a node at level i, and x_i is the (valid) value that has been assigned to decision variable i, HBB uses GUM to determine whether we can find a valid assignment for decision variables $j > i$ such that the best solution that was found so far can still be improved. Only if such a solution can be found, HBB explores the branch where decision variable i is assigned value x_i. Because Grover's algorithm (and hence GUM) is less accurate when finding solutions for smaller problems, we propose a hybrid approach, and use a classical algorithm to solve partial problems that consider ω decision variables or less. In addition, note that HBB can exploit classical techniques to improve the performance of a branch-and-bound procedure (e.g., dominance rules and/or bounding procedures). In contrast, RAP, cannot exploit such techniques, and is a far more simple procedure that iteratively uses GUM to draw a "random" solution from the set of solutions that can still improve the best-found solution. Even though simple, we show that RAP expects to solve any discrete optimization problem that has n binary decision variables using at most $O(\mu\sqrt{2^n})$ operations, where μ represents the number of operations required to verify whether a solution is valid. To investigate the practical runtime of HBB and RAP, we use both procedures to solve 66000 instances of the binary knapsack problem (i.e., a subset of the aforementioned 108000 instances). The results show that both procedures are able to solve a binary knapsack problem that has n items using $\approx n\sqrt{2^n}$ operations.

To further improve the performance of HBB and RAP, we assess the impact of several improvements. Because both procedures rely on GUM to find valid solutions, we first generalize GUM in the following way: (1) if a valid solution was found assuming $\hat{m}$ solutions in the last search, we use $\hat{m}$ as the initial number of valid solutions in the next search (that considers a more restricted problem that has at most $\hat{m}$ valid solutions), (2) in each iteration, rather than letting $\hat{m} = \hat{m}/2$, we let $\hat{m} = \hat{m}/\kappa$, and (3) we stop the procedure after processing $\hat{m} = \tau$ rather than after $\hat{m} = 1$. The generalized procedure (GGUM) allows us to evaluate different strategies in which different samples of $\hat{m}$ are used to find a valid solution. Next, we also investigate the impact of using a heuristic to initialize the starting solution (rather than assuming an initial solution that has value $V = 0$). To investigate the potential of techniques that improve the performance of the branch-and-bound of HBB, we also adopt an upper-bound procedure to fathom nodes (note that this upper-bound procedure cannot be used by RAP). The results of the experiment show that, even though we significantly improve performance, both procedures still require $\approx n\sqrt{2^n}$ operations to solve a binary knapsack problem that has n items. The results, however, also show that HBB can outperform RAP if the structure of the problem can be exploited efficiently. In addition, the results indicate that HBB and RAP can be used as heuristics to find (near-) optimal solutions using (far) less than $n\sqrt{2^n}$ operations.

Procedures HBB and RAP can also be generalized such that they can be used to solve any discrete optimization problem using $O(\mu\sqrt{2^{nb}})$ operations, where 2^b is the number of discrete values that can be assigned to any of the n decision variables (e.g., if $b = 1$, decision variables are binary and HBB and RAP require $O(\mu\sqrt{2^n})$ operations). As such, procedures HBB and RAP may be seen as general-purpose solvers that can solve well-known combinatorial problems as well as complex non-linear integer programming problems for which no dedicated algorithms exist. This is where the power of both procedures lies: their general applicability to solve any discrete optimization problem. Note that, even though HBB and RAP are general-purpose algorithms, they already match the worst-case performance of the best classical

algorithm for solving the binary knapsack problem; a problem that has been studied for more than a century. To illustrate that our procedures can also be used to solve other discrete optimization problems, we show how to solve the quadratic binary knapsack problem; a generalization of the regular binary knapsack problem that is strongly $\mathcal{NP}$-complete. Our results indicate that HBB and RAP are able to outperform the best classical algorithms.

Not only do HBB and RAP offer state-of-the-art performance when solving discrete optimization problems, they can also be used as heuristic procedures. In addition, as a branch-and-bound procedure, HBB allows to exploit the structure of a problem to further improve performance. In contrast, RAP can be used to solve problems that have no clear structure or that have a structure that is difficult to exploit. To combine the strengths of HBB and RAP, new procedures may be developed whose performance can be tailored depending on how effectively we can exploit the structure of the problem. The development of such new procedures, however, is left as a direction for future research (see also Appendix G for an example procedure).

Note that procedures HBB and RAP can easily run in parallel. In the case of RAP, for instance, given i quantum processors, i instances of RAP can be running at the same time, and we expect an optimality gap of $1/(i+1)$ after completing a first iteration of RAP on all processors (e.g., if we have 20 quantum processors, we expect an optimality gap of less than 5 percent). In addition, we do not have to wait until completing an iteration of RAP to already share best-found solutions. In fact, for every value of $\hat{m}$ that is evaluated using GGUM, if a solution is found that improves the best-found solution, subsequent searches on all quantum processors can use this best solution as a lower bound. It would be interesting to investigate the computational gains that can be achieved this way. Similarly, the parallel execution of other Grover-based algorithms may be an interesting topic for future research.

Our work also has several limitations that remain to be addressed. First, future research should evaluate the potential of procedures such as HBB and RAP to solve discrete optimization problems other than the (quadratic) binary knapsack problem. By comparing the performance of HBB, RAP, and state-of-the-art classical algorithms, we can identify problems that are more suited to be solved by quantum algorithms. Second, HBB and RAP are general-purpose algorithms that can be used to solve any discrete optimization problem. Therefore, procedures that are dedicated to solving a particular discrete optimization problem may outperform HBB, RAP, and perhaps also the best classical algorithms. The development of such dedicated procedures is a promising direction for future research. Third, procedures such as HBB and RAP can be extended such that they can also solve more general optimization problems (e.g., mixed-integer programming problems). Fourth, we focus on Grover-based algorithms. Next to Grover-based algorithms, however, other approaches (such as adiabatic quantum computation) may also be of interest. Fifth, HBB and RAP are exact procedures that, at best, can offer a quadratic speedup when compared to the classical algorithms. For large and/or complex problems, heuristic procedures are still required. Even though our results indicate that HBB and RAP can also be used as heuristic procedures, further research is required to investigate the potential of procedures such as HBB and RAP to find good solutions in limited time. Last but not least, we assume the existence of a fault-tolerant universal quantum computer that can effectively run HBB and RAP. If we look at IBM's (ambitious) quantum roadmap, it becomes clear that it may take many more years before such a computer will hit the market. Once a fault-tolerant quantum computer does become available, however, state-of-the-art procedures such as HBB and RAP have the potential to revolutionize the field of discrete optimization.

The contributions of this paper may be summarized as follows: (1) we provide OR researchers with the essential tools to solve discrete optimization problems using Grover-based quantum algorithms, (2) we propose several quantum procedures and show how they can be used to solve the binary knapsack problem, (3) we demonstrate how to assess the performance of a quantum algorithm, (4) we show that our procedures match (outperform) the best classical procedures when solving the (quadratic) binary knapsack problem, (5) we discuss several improvements that can further increase the performance of our procedures, (6) we illustrate that our procedures can also be used as heuristics to find (near-) optimal solutions, and (7) we show that our procedures can be generalized such that they can be used as general-purpose solvers that have the potential to revolutionize the field of discrete optimization.

CRediT authorship contribution statement

Stefan Creemers: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Luis Fernando Pérez Armas:** Validation, Methodology, Conceptualization.

Appendix A. Introduction into quantum computing

In this appendix, we first introduce the fundamentals of quantum computing (Appendix A.1) and define the operations that are required to establish universal quantum computing (Appendix A.2). Next, in Appendix A.3, we present the Deutsch algorithm and its extension to the Deutsch–Jozsa algorithm. The Deutsch–Jozsa algorithm serves as an important stepping stone towards Grover's algorithm. Grover's algorithm itself is described in detail in Appendix A.4.

A.1. Fundamentals of quantum computing

In 1939, Dirac introduced the "bra-ket" notation to facilitate the study of quantum systems. In this notation, a "ket" $|c\rangle$ denotes a column vector c defined in a complex vector space $\mathcal{V}$, a "bra" $\langle r|$ denotes a row vector r that acts as a linear map on $\mathcal{V}$, and a "bra-ket" $\langle r\rangle c$ represents the inner product of $\langle r|$ and $|c\rangle$. In what follows, we adopt bra-ket notation.

In quantum computing, information is stored in so-called "qubits". A qubit can be seen as a two-state quantum system that has basis states $|0\rangle = (1,0)^\top$ and $|1\rangle = (0,1)^\top$. These basis states are the quantum equivalent of states 0 and 1 of a classical bit. In contrast to a classical bit, however, a qubit may also be in a superposition of both basis states. The state of a quantum system is described by a so-called "wave function". In case of a single qubit, the wave function is a linear combination of basis states $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \tag{9}$$

where α and β are complex numbers that are also known as the "probability amplitudes" of wave function $|\psi\rangle$. By observing (i.e., measuring) a qubit in superposition, the state of the quantum system collapses into one of the basis states. The probability that a qubit collapses into basis state $|0\rangle$ or $|1\rangle$ is proportional to the square of probability amplitudes α and β, respectively. This result is also known as the Born rule (1926). Given that α and β are complex numbers, we have:

$$|\alpha|^2 + |\beta|^2 = 1. \tag{10}$$

In systems with two (or more) qubits, the basis states can be obtained as the Kronecker product of the basis states of each individual qubit. For instance, if we consider two qubits, a first basis state is given by:

$$|00\rangle = |0\rangle \otimes |0\rangle = (1,0)^\top \otimes (1,0)^\top = (1,0,0,0)^\top.$$

The other basis states are $|01\rangle = (0,1,0,0)^\mathsf{T}$, $|10\rangle = (0,0,1,0)^\mathsf{T}$, and $|11\rangle = (0,0,0,1)^\mathsf{T}$. If both qubits are in superposition, we get:

$$\begin{aligned}|\Psi\rangle &= |\psi_1\rangle \otimes |\psi_2\rangle \\ &= (\alpha_1|0\rangle + \beta_1|1\rangle) \otimes (\alpha_2|0\rangle + \beta_2|1\rangle) \\ &= \alpha_1\alpha_2|00\rangle + \alpha_1\beta_2|01\rangle + \beta_1\alpha_2|10\rangle + \beta_1\beta_2|11\rangle,\end{aligned}$$

where $|\Psi\rangle$ is used to denote the state of a set of qubits.

Similar to classical computing, quantum computing uses logic gates to perform operations. In contrast to the gates of classical computing, however, quantum gates have to be reversible, and are represented by unitary linear operators (i.e., a linear operator U is unitary if $UU^{-1} = UU^\dagger = I$, where I is the identity matrix). Even though there are many quantum gates, we need to introduce one gate in particular: the Hadamard gate (refer to, for example, Aaronson (2018) for a comprehensive overview of other quantum gates such as the Pauli and Toffoli gates). The Hadamard gate is used to put qubits in equal superposition. In case of a single qubit, the Hadamard gate (represented by the letter "H") operates as follows:

$$\begin{aligned}\mathrm{H}|0\rangle &= \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} \\ &= \frac{|0\rangle+|1\rangle}{\sqrt{2}} = |+\rangle, \quad \mathrm{H}|1\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} = \frac{|0\rangle-|1\rangle}{\sqrt{2}} = |-\rangle.\end{aligned}$$

Because H is an involutory matrix (i.e., a square matrix that is its own inverse), we have $\mathrm{H}|+\rangle = \mathrm{HH}|0\rangle = |0\rangle$ and $\mathrm{H}|-\rangle = \mathrm{HH}|1\rangle = |1\rangle$. Note that a qubit in state $|+\rangle$ or $|-\rangle$ has an equal probability to collapse into basis state $|0\rangle$ or $|1\rangle$ when measured. In general, when applied to a set of n qubits, the Hadamard gate is defined as the n^{th} tensor power of H (denoted $\mathrm{H}^{\otimes n}$), and the probability to measure any of the 2^n basis states equals 2^{-n}.

A.2. Universal quantum computing

In what follows, we first introduce the X, CX, and Toffoli quantum gates, and show how these gates can be used to create the quantum equivalent of the AND, OR, XOR, and NOT gates of classical computing.

The X gate (also referred to as the Pauli-X gate) is the quantum equivalent of the classical NOT gate and transforms input state $|0\rangle$ into output state $|1\rangle$ (and vice versa). Formally, X is a two-by-two matrix that operates as follows:

$$\mathrm{X}|0\rangle = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}0\\1\end{pmatrix} = |1\rangle, \quad \mathrm{X}|1\rangle = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}1\\0\end{pmatrix} = |0\rangle.$$

The CX (or controlled-X) gate has two input qubits: a target qubit on which an X operation may be performed, and a control qubit that determines whether or not the X operation is performed. If the control qubit is in basis state $|1\rangle$, the X operation is performed on the target qubit. Otherwise, if the control qubit is in basis state $|0\rangle$, the target qubit remains unchanged. Formally, CX is a four-by-four matrix that operates as follows:

$$\mathrm{CX}|00\rangle = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}\begin{pmatrix}1\\0\\0\\0\end{pmatrix} = \begin{pmatrix}1\\0\\0\\0\end{pmatrix} = |00\rangle.$$

One can verify that $\mathrm{CX}|10\rangle = |11\rangle$, $\mathrm{CX}|01\rangle = |01\rangle$, and $\mathrm{CX}|11\rangle = |10\rangle$.

The Toffoli gate is a controlled-CX (CCX) gate that has two control qubits. Only if both control qubits are in basis state $|1\rangle$, the X operation is performed on the target qubit. Formally, CCX is an eight-by-eight matrix that operates as follows:

$$\mathrm{CCX}|000\rangle = \begin{pmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0\end{pmatrix}\begin{pmatrix}1\\0\\0\\0\\0\\0\\0\\0\end{pmatrix} = \begin{pmatrix}1\\0\\0\\0\\0\\0\\0\\0\end{pmatrix} = |000\rangle.$$

One can verify that $\mathrm{CCX}|100\rangle = |100\rangle$, $\mathrm{CCX}|010\rangle = |010\rangle$, $\mathrm{CCX}|110\rangle = |111\rangle$, $\mathrm{CCX}|001\rangle = |001\rangle$, $\mathrm{CCX}|101\rangle = |101\rangle$, $\mathrm{CCX}|011\rangle = |011\rangle$, and $\mathrm{CCX}|111\rangle = |110\rangle$.

We can use the X, CX, and CCX gates to create circuits that mimic the classical AND, OR, XOR, and NOT gates: the quantum equivalent of the AND gate corresponds to a CCX gate with target qubit initialized as $|0\rangle$, the quantum equivalent of the OR gate is a series of a CX, X, CCX, and X gate with target qubit initialized as $|0\rangle$, the quantum equivalent of the XOR gate is a series of two CX gates with target qubit initialized as $|0\rangle$, and the quantum equivalent of the NOT gate is simply the X gate. The circuits of all aforementioned gates are shown in Fig. 11.

The NOT, AND, and OR operation suffice to establish universal classical computing. As a result, the X, CX, and CCX gate (i.e., the quantum equivalent of the NOT, AND, and OR operation on a classical computer) allow to perform any classical operation on a quantum computer (see also Deutsch (1989) who has shown that any classical operation can be implemented on a quantum computer using only a polynomial overhead). Quantum computers, however, are not limited to classical operations. To establish universal quantum computing, we require a set of quantum gates that allow us to approximate any other quantum gate (i.e., any unitary operator) efficiently within a given error bound (following the Solovay–Kitaev theorem). Several sets of universal quantum gates exist. For instance, in 2003, Aharonov has shown that the CCX and the Hadamard gate form a set of universal quantum gates.

A.3. The deutsch algorithm

The Deutsch algorithm was first published in (1985), has been generalized by Deutsch and Jozsa in (1992), and has further been improved by Cleve et al. (1998). Even though the Deutsch algorithm solves a problem that has no practical use, it is of particular interest to us because (1) it is the first quantum algorithms for which it has been shown that it can solve a problem more efficiently than a classical algorithm, and (2) it is a stepping stone towards Grover's algorithm that is introduced in the next section.

The Deutsch algorithm solves the following problem: we are given a function f_x that maps a binary input x to a binary output in a way that is unknown to us. Even though function f_x acts as a black box, we are told that f_x is either "constant" or "balanced"; f_x is said to be constant if it always produces the same output, and f_x is balanced if it produces a zero for half of the inputs and a 1 for the other half of the inputs. As such, f_x has to be one of the following four functions:

$$\begin{aligned}&f_0 = 0 \text{ and } f_1 = 0 \;\Rightarrow\; f \text{ is constant (case a)},\\ &f_0 = 1 \text{ and } f_1 = 1 \;\Rightarrow\; f \text{ is constant (case b)},\\ &f_0 = 0 \text{ and } f_1 = 1 \;\Rightarrow\; f \text{ is balanced (case c)},\\ &f_0 = 1 \text{ and } f_1 = 0 \;\Rightarrow\; f \text{ is balanced (case d)}.\end{aligned}$$

To solve the Deutsch problem, it suffices to determine whether f_x is constant or balanced. The traditional way to solve this problem would be to evaluate both f_0 and f_1, and to observe whether the outputs are the same (i.e., f_x is constant if $f_0 = f_1$, and f_x is balanced otherwise). This solution, however, requires two calls to function f_x. Using the Deutsch algorithm, we only require a single call to function f_x.

The circuit of the Deutsch algorithm is presented in Fig. 12 and can be broken down into four steps (indicated by the red dashed lines

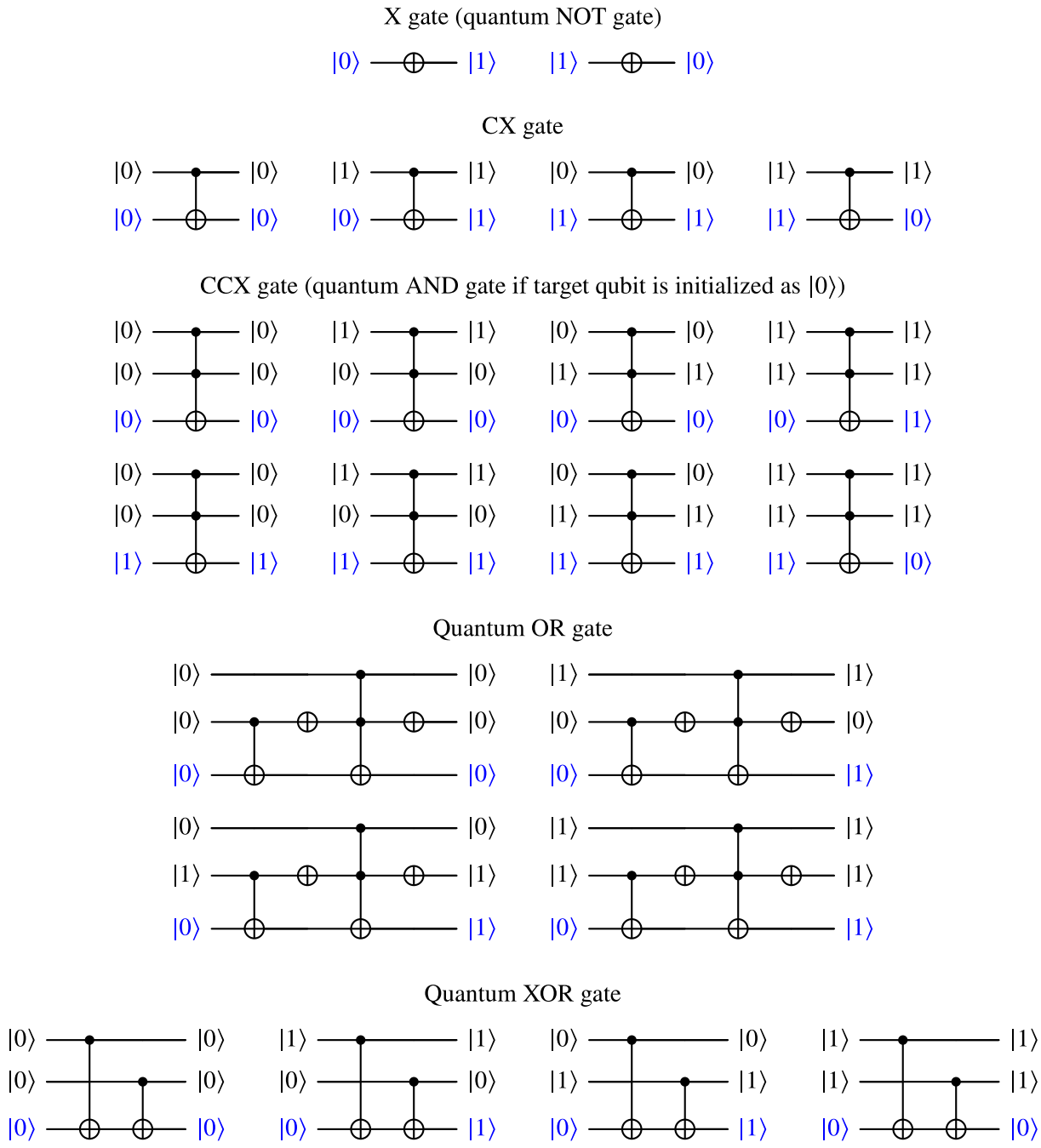


Fig. 11. Circuits of the X, CX, and CCX quantum gates, as well as the quantum equivalents of the classical AND, OR, XOR, and NOT gates. The target qubit is indicated in blue. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

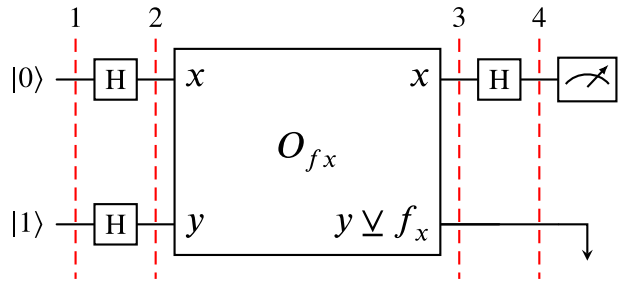


Fig. 12. Circuit of the Deutsch algorithm.

Table A.4
Mapping of basis states by oracle O_{fx} for each of the four cases.

case	$O_{fx}(\|00\rangle)$	$O_{fx}(\|01\rangle)$	$O_{fx}(\|10\rangle)$	$O_{fx}(\|11\rangle)$
(a)	$\|00\rangle$	$\|01\rangle$	$\|10\rangle$	$\|11\rangle$
(b)	$\|01\rangle$	$\|00\rangle$	$\|11\rangle$	$\|10\rangle$
(c)	$\|00\rangle$	$\|01\rangle$	$\|11\rangle$	$\|10\rangle$
(d)	$\|01\rangle$	$\|00\rangle$	$\|10\rangle$	$\|11\rangle$

in the circuit). In a first step, we initialize two qubits, and obtain quantum state $|\Psi_1\rangle = |01\rangle$. Next, in a second step, both qubits are put in superposition using a Hadamard gate:

$$|\Psi_2\rangle = \mathrm{H}^{\otimes 2}|01\rangle = \frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & 1 & -1 & -1\\ 1 & -1 & -1 & 1\end{bmatrix}\begin{pmatrix}0\\1\\0\\0\end{pmatrix} = \frac{1}{2}\begin{pmatrix}1\\-1\\1\\-1\end{pmatrix} = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) = |+-\rangle.$$

In step 3, we pass both qubits through a (purely-quantum) oracle gate O_{fx} that, unlike us, has complete knowledge of function f_x. Oracle O_{fx} uses its knowledge of f_x to map state $|xy\rangle$ to $|x(y \veebar f_x)\rangle$, where $\veebar$ represents the XOR operation, and where x and y represent the first and second qubit, respectively. After applying oracle O_{fx}, we get:

$$|\Psi_3\rangle = \frac{1}{2}\left(|0\rangle \otimes \left(|(0 \veebar f_0)\rangle - |(1 \veebar f_0)\rangle\right) + |1\rangle \otimes \left(|(0 \veebar f_1)\rangle - |(1 \veebar f_1)\rangle\right)\right).$$

For $x \in \{0,1\}$, one can verify that $|x\rangle \otimes (|0 \veebar f_x\rangle - |1 \veebar f_x\rangle) = (-1)^{f_x}|x\rangle \otimes (|0\rangle - |1\rangle)$, and hence, the sign (or phase) of the second qubit is "kicked back" to the first qubit if $f_x = 1$. As a result, the sign of the first qubit "flips", and $|\Psi_3\rangle$ may also be written as:

$$|\Psi_3\rangle = \frac{1}{2}\left((-1)^{f_0}|0\rangle \otimes (|0\rangle - |1\rangle) + (-1)^{f_1}|1\rangle \otimes (|0\rangle - |1\rangle)\right).$$

Quantum state $|\Psi_3\rangle$ can further be simplified depending on whether f_x is constant or balanced. For each of the four aforementioned cases, Table A.4 lists the mappings of oracle O_{fx}.

Using these mappings, we get:

$$|\Psi_{3.a}\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |+-\rangle,$$

$$|\Psi_{3.b}\rangle = \frac{1}{2}(-|00\rangle + |01\rangle - |10\rangle + |11\rangle) = -\frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = -|+-\rangle,$$

$$|\Psi_{3.c}\rangle = \frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle) = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |--\rangle,$$

$$|\Psi_{3.d}\rangle = \frac{1}{2}(-|00\rangle + |01\rangle + |10\rangle - |11\rangle) = -\frac{|0\rangle - |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = -|--\rangle.$$

In case (a), the state of the system remains unchanged (i.e., $|\Psi_2\rangle = |\Psi_{3.a}\rangle$). In cases (b) and (d), the global phase of the system changes (indicated in blue). The global phase of a system, however, has no impact on the state of the system; $|\Psi\rangle$ and $-|\Psi\rangle$ are said to be equivalent. In cases (c) and (d), the local phase of the first qubit changes (indicated in red). The impact of this change becomes clear in the last step of the Deutsch algorithm. In the last step, we dispose of the second qubit (it is no longer needed) and observe that the first qubit is in state $|+\rangle$ if f_x is constant (cases (a) and (b)), and in state $|-\rangle$ if f_x is balanced (cases (c) and (d)). Therefore, if the first qubit is passed through a second Hadamard in the last step, we measure $|\Psi_4\rangle = \mathrm{H}|+\rangle = |0\rangle$ if f_x is constant, and $|\Psi_4\rangle = \mathrm{H}|-\rangle = |1\rangle$ if f_x is balanced. As such, the Deutsch algorithm is able to determine whether f_x is constant or balanced with only a single call to function f_x. This result was further generalized by Deutsch and Jozsa, who use a function $f_{\mathbf{x}}$ that has n binary inputs $\mathbf{x} = \{x_1, \ldots, x_n\}$ (rather than only a single binary input, as was the case in the Deutsch algorithm).

A.4. Grover's algorithm

The goal of Grover's algorithm is to find a single target entry in an unstructured database that has 2^n entries. Similar to Deutsch and Jozsa, Grover uses a function $f_{\mathbf{x}}$ that has n binary inputs that are modeled using n qubits. The resulting quantum system has 2^n basis states that each correspond to an entry of the database. In contrast to Deutsch and Jozsa, however, $f_{\mathbf{x}}$ is not constant or balanced, but returns 1 if the target entry is given as an input, and 0 otherwise. Table A.5 provides an example with three binary inputs, resulting in a quantum system that has eight basis states. In this example, function $f_{\mathbf{x}}$ outputs 1 only for target entry 101 (represented by basis state $|101\rangle$). Throughout the remainder of this section, this example will be used to illustrate the different steps of Grover's algorithm. The first steps of Grover's algorithm are almost identical to those of the Deutsch–Jozsa algorithm. The only difference is that oracle $O_{f_{\mathbf{x}}}$ only flips the phase of the basis state that corresponds to the target entry (i.e., oracle $O_{f_{\mathbf{x}}}$ flips the phase of those basis states for which $f_{\mathbf{x}}$ returns 1; note that $O_{f_{\mathbf{x}}}$ is a purely-quantum oracle). In our example, $|\Psi_1\rangle = |0001\rangle$, $|\Psi_2\rangle = |+++-\rangle$, and $|\Psi_3\rangle$ is given by (the basis state whose phase has been flipped is indicated in blue):

$$|\Psi_3\rangle = \frac{1}{\sqrt{8}}(|000\rangle + |100\rangle + |010\rangle + |110\rangle + |001\rangle - |101\rangle + |011\rangle + |111\rangle) \otimes |-\rangle.$$

At this stage, we have only called oracle $O_{f_{\mathbf{x}}}$ once, however, if we measure the first three qubits, there is an equal probability that the system collapses into either one of the eight basis states. In other words, we have one chance out of eight to find out that $|101\rangle$ is the target basis state; measuring the state of the system at this stage is equivalent to randomly guessing the target entry. To increase the probability of measuring the correct basis state, Grover uses a so-called "diffusion" operator. In short, the diffusion operator is a quantum gate that reflects the probability amplitudes of the basis states about the average probability amplitude. The goal of the diffusion operator is to amplify the probability amplitude of the target basis state. For three qubits, the diffusion operator is given by:

$$\mathrm{H}^{\otimes 3}(2|0^{\otimes 3}\rangle\langle 0^{\otimes 3}| - I_8)\mathrm{H}^{\otimes 3} = \begin{pmatrix}-0.75 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25\\ 0.25 & -0.75 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25\\ 0.25 & 0.25 & -0.75 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25\\ 0.25 & 0.25 & 0.25 & -0.75 & 0.25 & 0.25 & 0.25 & 0.25\\ 0.25 & 0.25 & 0.25 & 0.25 & -0.75 & 0.25 & 0.25 & 0.25\\ 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & -0.75 & 0.25 & 0.25\\ 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & -0.75 & 0.25\\ 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & -0.75\end{pmatrix},$$

where $|0^{\otimes n}\rangle$ is the n^{th} tensor power of $|0\rangle$, $|c\rangle\langle r|$ is the outer product of $|c\rangle$ and $\langle r|$, and I_{2^n} is an identity matrix of dimension $2^n \times 2^n$. When

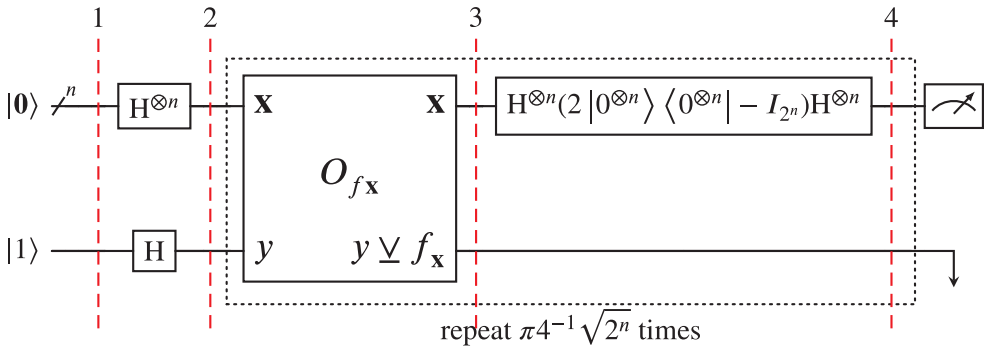


Fig. 13. Circuit of Grover's algorithm.

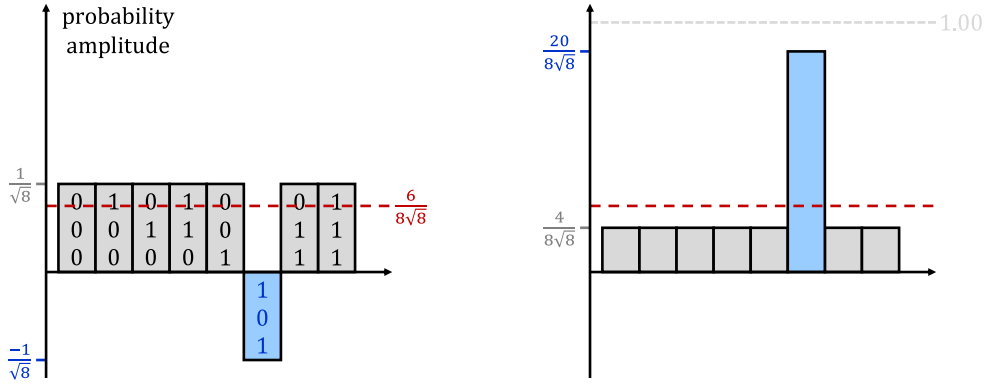


Fig. 14. Probability amplitudes of the different basis states before and after applying the diffusion operator in a first iteration (target basis state is colored blue, other basis states are colored gray, and the average amplitude used by the diffusion operator is colored red). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table A.5
Example database with eight entries that are represented by the basis states of a quantum system that has three qubits. The target entry is printed in blue. Function $f_{\mathbf{x}}$ returns 1 only for the target entry.

x	000	100	010	110	001	101	011	111
$f_{\mathbf{x}}$	0	0	0	0	0	1	0	0

applied to our example, one can verify that the average probability amplitude is $6(8\sqrt{8})^{-1}$, and we get:

$$|\Psi_4\rangle = \left(\frac{20}{8\sqrt{8}}|101\rangle + \frac{4}{8\sqrt{8}}(|000\rangle + |100\rangle + |010\rangle + |110\rangle + |001\rangle + |011\rangle + |111\rangle)\right) \otimes |-\rangle.$$

In other words, the probability amplitude of the target basis state (i.e., $-(8\sqrt{8})^{-1}$) is reflected about $6(8\sqrt{8})^{-1}$, resulting in probability amplitude $20(8\sqrt{8})^{-1}$. The probability amplitudes of all other basis states (i.e., $(8\sqrt{8})^{-1}$) are also reflected, resulting in probability amplitudes $4(8\sqrt{8})^{-1}$. These reflections are illustrated in Fig. 14.

The circuit of Grover's algorithm is presented in Fig. 13. Once again, the algorithm can be broken down into four steps (indicated by the red dashed lines in the circuit).

After a first iteration, the probability of identifying the target basis state has increased from 0.125 to 0.7813 (i.e., $(20(8\sqrt{8})^{-1})^2$). We can further increase this probability by repeating steps 3 and 4. We get:

$$|\Psi_{3.2}\rangle = \left(-\frac{20}{8\sqrt{8}}|101\rangle + \frac{4}{8\sqrt{8}}(|000\rangle + |100\rangle + |010\rangle + |110\rangle + |001\rangle + |011\rangle + |111\rangle)\right) \otimes |-\rangle,$$

resulting in an average probability amplitude of $(8\sqrt{8})^{-1}$, and:

$$|\Psi_{4.2}\rangle = \left(\frac{22}{8\sqrt{8}}|101\rangle - \frac{2}{8\sqrt{8}}(|000\rangle + |100\rangle + |010\rangle + |110\rangle + |001\rangle + |011\rangle + |111\rangle)\right) \otimes |-\rangle.$$

After a second iteration, the probability of success increases to 0.9453 (i.e., $(22(8\sqrt{8})^{-1})^2$); we are 94.53 percent certain to identify the target basis state after only two calls to oracle $O_{f_{\mathbf{x}}}$. Another iteration, however, will not increase the success probability:

$$|\Psi_{3.3}\rangle = \left(-\frac{22}{8\sqrt{8}}|101\rangle - \frac{2}{8\sqrt{8}}(|000\rangle + |100\rangle + |010\rangle + |110\rangle + |001\rangle + |011\rangle + |111\rangle)\right) \otimes |-\rangle,$$

resulting in an average probability amplitude of $-4.5(8\sqrt{8})^{-1}$, and:

$$|\Psi_{4.3}\rangle = \left(\frac{13}{8\sqrt{8}}|101\rangle - \frac{7}{8\sqrt{8}}(|000\rangle + |100\rangle + |010\rangle + |110\rangle + |001\rangle + |011\rangle + |111\rangle)\right) \otimes |-\rangle.$$

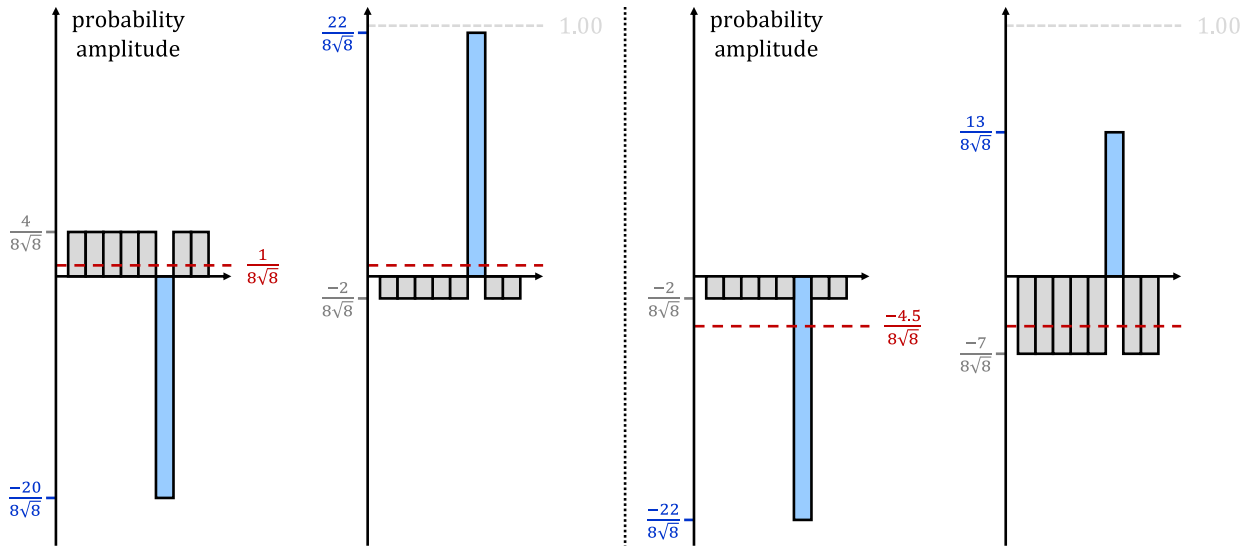


Fig. 15. Probability amplitudes of the different basis states before and after applying the diffusion operator in a second (left panel) and third (right panel) iteration (target basis state is colored blue, other basis states are colored gray, and the average amplitude used by the diffusion operator is colored red). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table A.6
Average probability amplitude, probability amplitude of the target basis state, probability amplitude of the other basis states, and probability to successfully identify the target basis state before step 4 for different numbers of iterations.

iteration	average amplitude	target amplitude	other amplitude	success probability
1	0.2652	−0.3536	0.3536	0.1250
2	0.0442	−0.8839	0.1768	0.7813
3	−0.1989	−0.9723	−0.0884	0.9453
4	−0.3425	−0.5745	−0.3094	0.3301
5	−0.3149	0.1105	−0.3757	0.0122
6	−0.1298	0.7403	−0.2541	0.5480
7	0.1202	0.9999	−0.0055	0.9998
8	0.3100	0.7596	0.2458	0.5770
9	0.3449	0.1395	0.3743	0.0195
10	0.2073	−0.5504	0.3156	0.3029
11	−0.0339	−0.9650	0.0991	0.9313
12	−0.2582	−0.8972	−0.1669	0.8049
13	−0.3534	−0.3807	−0.3495	0.1450
14	−0.2719	0.3261	−0.3573	0.1063
15	−0.0544	0.8698	−0.1865	0.7566
16	0.1902	0.9787	0.0776	0.9578

After a third iteration, the probability of success drops to 0.3301. Fig. 15 illustrates the second and third iteration.

For different numbers of iterations, Table A.6 lists the average probability amplitude, the probability amplitudes of the target and other basis states, and the probability to successfully identify the target basis state before the diffusion operator is applied. In addition, Fig. 16 shows the probability amplitude of the target basis state after each iteration (i.e., after applying the diffusion operator). From Fig. 16, one can observe that the probability amplitude of the target basis state can be approximated by a sine function that has its first extremum after $I(n,m) = 2$ iterations.

Appendix B. Probability to measure target basis state after I Grover iterations

Let $P(n,m,I)$ denote the probability to measure one of the m target basis state after performing I Grover iterations in a system that has 2^n basis states. To obtain $P(n,m,I)$, we use Algorithm 7. In a first step of Algorithm 7, we initialize the current probability amplitude of the other basis states (α_o), the current probability amplitude of the target basis states (α_t), and the current iteration (i). Next, in each iteration i, we calculate the average probability amplitude ($\bar{\alpha}$) as the weighted sum of the probability amplitudes of the target and the other basis states. We use the average probability amplitude to update the probability amplitude of the target and other states (i.e., we reflect α_t and α_o about $\bar{\alpha}$) and return $P(n,m,I)$ as α_t^2 if $i \geq I$.

Algorithm 7: Procedure to determine probability to measure a target basis state in a system that has n qubits and m target basis states.

initialize $\alpha_o = (\sqrt{2^n})^{-1}$, $\alpha_t = -\alpha_o$, and $i = 1$;
while $i < I$ **do**
  $\bar{\alpha} = 2^{-n}((2^n - m)\alpha_o + m\alpha_t)$;
  $\alpha_t = \alpha_t - 2\bar{\alpha}$;
  $\alpha_o = 2\bar{\alpha} - \alpha_o$;
  $i = i + 1$;
return α_t^2;

Appendix C. Solving the binary knapsack problem using Grover's algorithm

In order to use Grover's algorithm to effectively solve the binary knapsack problem, we cannot rely on an oracle function (that has complete knowledge of all valid solutions). Instead, we have to implement function $f_{\mathbf{x}}$ on a quantum computer. The implementation of $f_{\mathbf{x}}$ as a quantum circuit is presented in Fig. 17. The circuit may be divided into three parts (indicated by the red dashed lines in the circuit). In the first two parts, we determine whether knapsack $\mathbf{x}$ satisfies the weight- and value constraints (constraints (2) and (4) of the binary knapsack problem formulation available in Section 3). In a third part, we obtain the output of function $f_{\mathbf{x}}$. In what follows, we use an example to demonstrate the dynamics of the circuit. In the example, we have a pool of three items and a knapsack that has maximum weight capacity $W = 4$. The weights and values of the items are listed in Table C.7. Since we are working with qubits, all weights and values need to be converted to binary values. Let b denote the number of qubits that are required to represent the largest weight or value used in the example. In our example, $b = 3$, and the qubits that represent the maximum weight capacity are initialized as follows: $|c_1^W\rangle = |1\rangle$, $|c_2^W\rangle = |0\rangle$, and $|c_3^W\rangle = |0\rangle$ (with $\mathbf{c}^W = \{c_1^W, \ldots, c_b^W\}$). The binary conversions of the weights and values of the items are listed in Table C.7, where

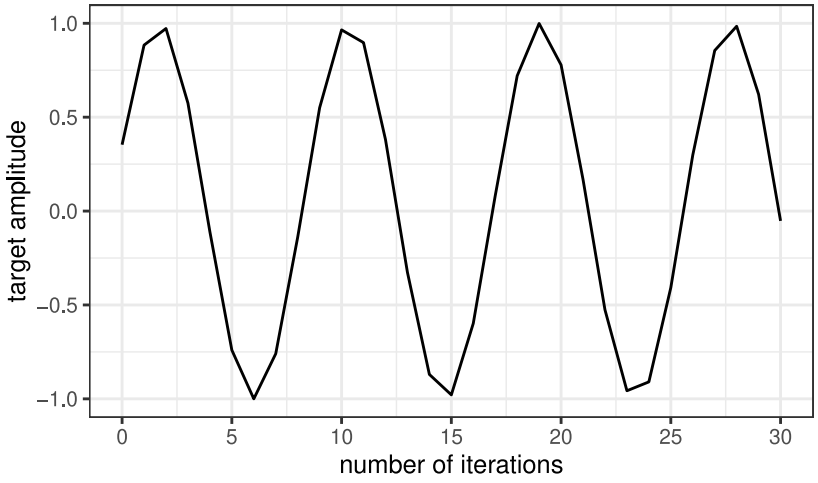


Fig. 16. Probability amplitude of the target basis state after each Grover iteration.

Table C.7
Data for example knapsack problem with $n = 3$ items.

i	w_i	c^w_{i1}	c^w_{i2}	c^w_{i3}	v_i	c^v_{i1}	c^v_{i2}	c^v_{i3}
1	2	0	1	0	3	0	1	1
2	3	0	1	1	1	0	0	1
3	2	0	1	0	2	0	1	0

$\mathbf{c}^w = \{\mathbf{c}^w_1, \dots, \mathbf{c}^w_n\}$, $\mathbf{c}^v = \{\mathbf{c}^v_1, \dots, \mathbf{c}^v_n\}$, $\mathbf{c}^w_i = \{c^w_{i1}, \dots, c^w_{ib}\}$, and $\mathbf{c}^v_i = \{c^v_{i1}, \dots, c^v_{ib}\}$. In our example, $|\mathbf{c}^W\rangle$, $|\mathbf{c}^w\rangle$, and $|\mathbf{c}^v\rangle$ are initialized as $|100\rangle$, $|010011010\rangle$, and $|011001010\rangle$, respectively. Next, imagine that we use the circuit of function $f_{\mathbf{x}}$ to evaluate knapsack $\mathbf{x} = \{1, 0, 1\}$. In this case, $|\mathbf{x}\rangle$ is initialized as $|101\rangle$. In a first part of the circuit, we distinguish three steps (indicated by the blue dotted lines in the circuit). In a first step (1a), we determine how much weight each item contributes to knapsack $\mathbf{x}$. The contributed weight of the items is represented by $\mathbf{q}^w = \{\mathbf{q}^w_1, \dots, \mathbf{q}^w_n\}$, with $\mathbf{q}^w_i = \{q^w_{i1}, \dots, q^w_{ib}\}$. $|\mathbf{q}^w\rangle$ is initialized as $|\mathbf{0}\rangle$ and is used to store the result of an AND operation on $|\mathbf{x}\rangle$ and $|\mathbf{c}^w\rangle$. In our example, one can verify that $|\mathbf{q}^w\rangle = |010000010\rangle$. In a second step (1b), we obtain the weight of knapsack $\mathbf{x}$ itself (represented by $\mathbf{q}^W = \{q^W_1, \dots, q^W_b\}$). $|\mathbf{q}^W\rangle$ is initialized as $|\mathbf{0}\rangle$ and is used to store the result of $\sum_{i=1}^n |\mathbf{q}^w_i\rangle$. To perform the aforementioned addition, we can create our own adder circuits or use a more efficient circuit that was proposed by Draper (2000). In our example, $|\mathbf{q}^W\rangle = |100\rangle$. As a last step (1c), we verify whether knapsack $\mathbf{x}$ satisfies the weight constraint. This can be done using the quantum equivalent of a digital magnitude comparator (see, for example, Xia et al. (2018)). The digital magnitude comparator compares the weight of knapsack $\mathbf{x}$ (represented by $\mathbf{q}^W$) and the maximum knapsack weight (represented by $\mathbf{c}^W$), and outputs 1 if a smaller-than-or-equal-to condition is met. The result is stored in a qubit that is initialized as $|q^W\rangle = |0\rangle$. In our example, the weight constraint is satisfied, and $|q^W\rangle$ evaluates to $|1\rangle$. This concludes the first part of the circuit. The second part of the circuit is almost identical to the first part, and verifies whether the value of the knapsack (stored in $|\mathbf{q}^V\rangle = \sum_{i=1}^n |\mathbf{q}^v_i\rangle$, where $\mathbf{q}^V = \{q^V_1, \dots, q^V_b\}$, $\mathbf{q}^v = \{\mathbf{q}^v_1, \dots, \mathbf{q}^v_n\}$, and $\mathbf{q}^v_i = \{q^v_{i1}, \dots, q^v_{ib}\}$) is at least equal to value V (stored in $|\mathbf{c}^V\rangle$, where $\mathbf{c}^V = \{c^V_1, \dots, c^V_b\}$). If this is the case, the value constraint is satisfied, and $|q^V\rangle = |1\rangle$. If, however, knapsack $\mathbf{x}$ does not satisfy the value constraint, then $|q^V\rangle = |0\rangle$. In a last part of the circuit, we obtain the outcome of function $f_{\mathbf{x}}$ itself (stored in a qubit that is initialized as $|\delta\rangle = |0\rangle$). If both constraints are satisfied, $|\delta\rangle = |1\rangle$ (and $|\delta\rangle = |0\rangle$ otherwise).

Fig. 18 presents the circuit of Grover's algorithm equipped with function $f_{\mathbf{x}}$ (the steps of the algorithm are indicated by red dashed lines). In this circuit, $\mathbf{x}$, $\mathbf{c}$, $\mathbf{q}$, and δ are the qubits that are used to store the decision variables, the constant values that are used by function $f_{\mathbf{x}}$, the outcomes of operations performed by function $f_{\mathbf{x}}$, and the result of function $f_{\mathbf{x}}$, respectively. In addition, y is a qubit that is used to flip the phase of valid solutions. $|y\rangle$ is initialized as $|1\rangle$. The initialization of $|\mathbf{c}\rangle$ depends on the problem instance. All other qubits are initialized as $|0\rangle$. The gates grouped in blue replace oracle gate $O_{f_{\mathbf{x}}}$ in the original circuit of Grover's algorithm (see Fig. 13 in Appendix A.4). These gates perform three operations. First, function $f_{\mathbf{x}}$ evaluates whether $\mathbf{x}$ is valid, and stores the result in $|\delta\rangle$. Next, a XOR operation is used to map $|\delta y\rangle$ to $|\delta(y \veebar \delta)\rangle$. After that, function $f^\dagger_{\mathbf{x}}$ applies all gates of function $f_{\mathbf{x}}$ in reverse order. By doing so, function $f^\dagger_{\mathbf{x}}$ reinitializes $|\mathbf{q}\rangle$ and $|\delta\rangle$ (as $|\mathbf{0}\rangle$ and $|0\rangle$, respectively) such that they are ready for a next Grover iteration (note that $\mathbf{c}$ and y are unaffected, and remain in state $|\mathbf{c}\rangle$ and $|-\rangle$ until they are discarded at the end of the procedure). This process ensures that the circuit is reversible and is also known as "uncomputation" (refer to Nielsen and Chuang (2010) for more details).

Appendix D. Performance of GUM

The goal of GUM is to identify one of the m valid solutions in a system that has 2^n solutions. An outline of the procedure is provided by Algorithm 1 in the main text. The expected number of calls to function $f_{\mathbf{x}}$ required by GUM is given by:

$$\sum_{i=0}^{n} \left((i+1) \sum_{j=0}^{i} 2I(n, 2^{(n-j)}) \right) (1 - \phi_{i-1}) P(n, m, I(n, 2^{(n-i)})), \tag{11}$$

where ϕ_i is the probability of having measured a valid solution after i GUM iterations and can be obtained through the following recursive relationship (with $\phi_{-1} = 0$):

$$\phi_i = \phi_{i-1} + (1 - \phi_{i-1}) P(n, m, I(n, 2^{(n-i)})). \tag{12}$$

If we are lucky, GUM identifies a valid solution in the first run of Grover's algorithm (i.e., when assuming $\hat{m} = 2^n$). In this case, we require two times $I(n, 2^n)$ calls to $f_{\mathbf{x}}$ (once to flip the phase of all valid solutions and once to uncompute) and one call to verify whether the measured solution is valid. If, on the other hand, no valid solution was found in the first run (with probability $1 - P(n, m, I(n, 2^n))$), we perform a second run of Grover's algorithm in which $\hat{m} = 2^{(n-1)}$. If we find a solution in this run (with probability $P(n, m, I(n, 2^{(n-1)}))$), we have performed $1 + 2I(n, 2^n)$ calls to $f_{\mathbf{x}}$ in a previous run, and will perform another $1 + 2I(n, 2^{(n-1)})$ calls in this run. In total, we have performed $2 + \sum_{j=0}^{1} 2I(n, 2^{(n-j)})$ calls to function $f_{\mathbf{x}}$. In general, after

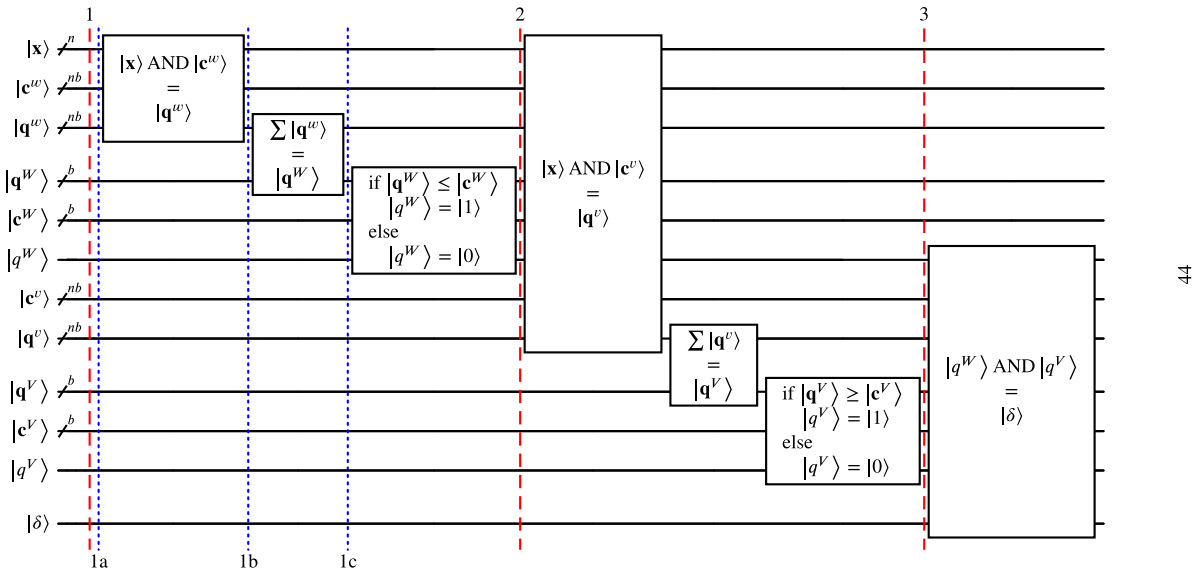


Fig. 17. Circuit of function $f_{\mathbf{x}}$ that can be used to verify whether a knapsack solution $\mathbf{x}$ has a weight smaller than W and a value of at least V.

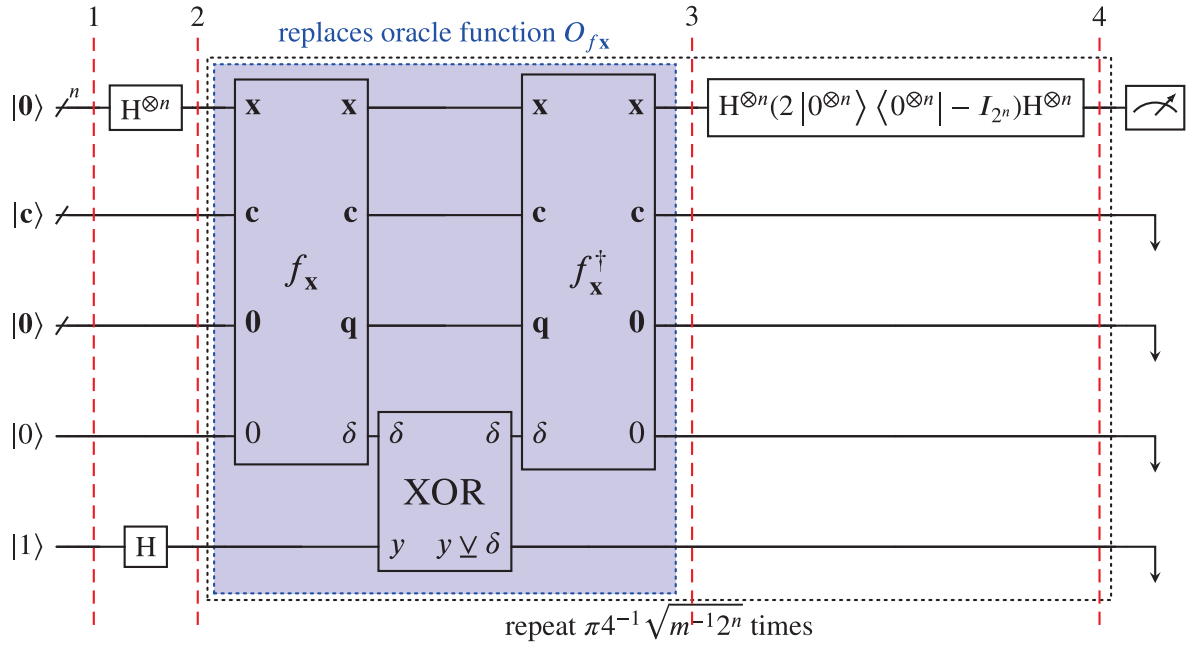


Fig. 18. Circuit of Grover's algorithm equipped with function $f_{\mathbf{x}}$.

i runs, we have performed $(i+1)\sum_{j=0}^{i} 2I(n, 2^{(n-j)})$ calls to function $f_{\mathbf{x}}$. In the worst case, no valid solution can be found, and GUM requires $(n+1)+\sum_{i=0}^{n} 2I(n, 2^{(n-i)}) \approx O(\sqrt{2^n})$ function calls.

To further investigate the performance of GUM, we perform a computational experiment. The results of the experiment are summarized in Fig. 19. The top panel of Fig. 19 shows the number of operations (divided by $\sqrt{m^{-1}2^n}$) required by GUM for various values of m and n (note that we divide by $\sqrt{m^{-1}2^n}$ to show that GUM requires $\approx \sqrt{m^{-1}2^n}$ operations). We observe that the expected number of operations is rather constant if $m < 2^{(n-2)}$. Only for values of $m > 2^{(n-2)}$, the expected number of operations starts to increase. This is not surprising as approximation $I(n, m)$ (that is used by GUM to determine the number of Grover iterations) is inaccurate if $m > 2^{(n-2)}$ (see also the middle panel of Fig. 19 that shows the probability that Grover's algorithm is able to measure any of the m solutions if $I(n, m)$ iterations are performed). To resolve this problem, we can use $f_{\mathbf{x}}$ to evaluate, for example, 20 random solutions before running GUM. If we have a probability of at least $2^{(n-2)} = 0.25$ to find a valid solution, with 20 trials we are at least 99.68 percent certain to find a valid solution. If no valid solution was found after 20 trials, we can safely assume there are (far) less than $2^{(n-2)}$ valid solutions, and we run GUM. Last but not least, the bottom panel of Fig. 19 shows the average and the maximum number of expected operations (divided by $\sqrt{m^{-1}2^n}$) required by GUM over all values of m, for various values of n. We observe that the average and the maximum number of operations converge as n increases. In addition, the expected number of operations (divided by $\sqrt{m^{-1}2^n}$) shown in the top panel also converges as n increases. As a result, we conjecture that GUM requires $O(\sqrt{m^{-1}2^n})$ operations to find any of the m valid solutions in a set of 2^n solutions (rather than $O(\sqrt{2^n})$ operations).

Approximation $I(n, m)$ cannot always be used with confidence to identify a valid solution for all values of m. Procedure GUM, however, relies on approximation $I(n, m)$ to determine the number of Grover

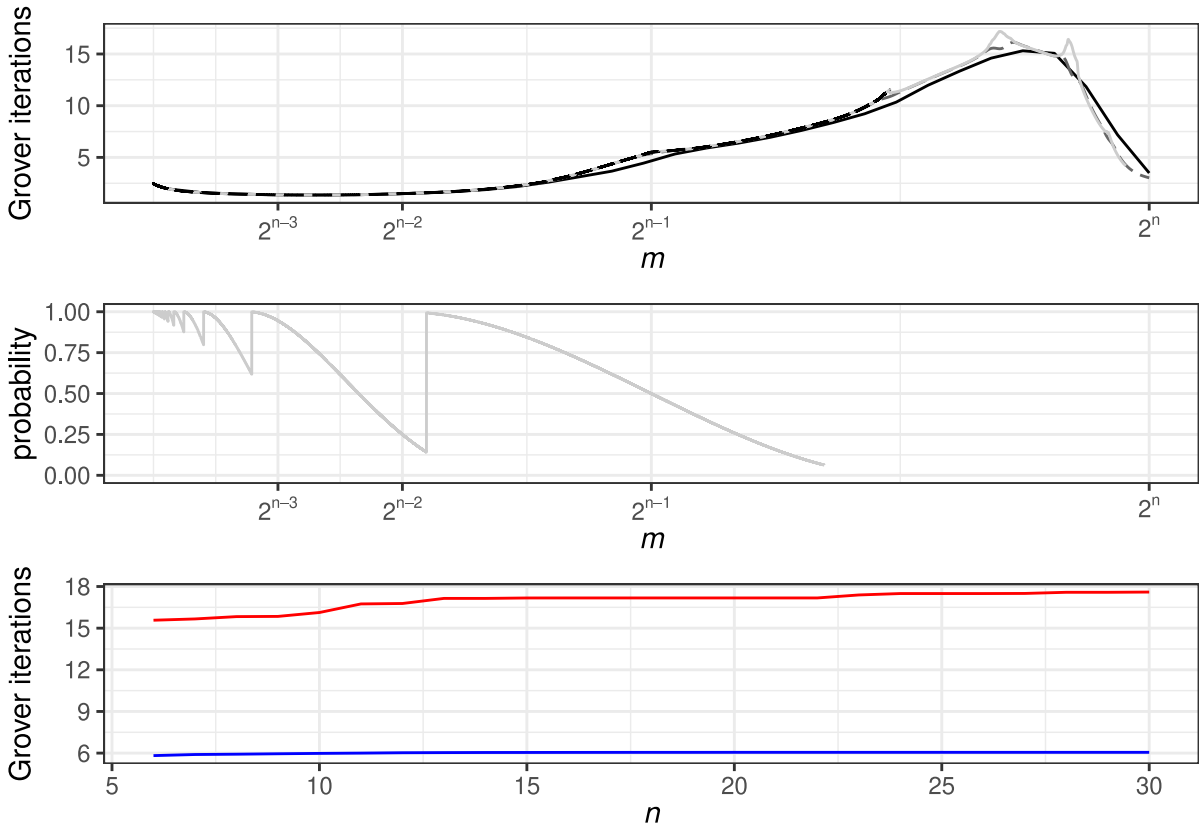


Fig. 19. The top panel shows the expected number of operations divided by $\sqrt{m^{-1}2^n}$ required by GUM for various values of m, and for $n = 6$ (black full line), $n = 10$ (dark gray dashed line), $n = 16$ (black dashed line), and $n = 22$ (gray full line). The middle panel shows the probability to measure one of the m valid solutions using Grover's algorithm if $I(n, m)$ Grover iterations are used. The bottom panel shows the average (blue) and maximum (red) number of expected operations divided by $\sqrt{m^{-1}2^n}$ required by GUM over all values of m, for various values of n. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

iterations to be used in each run of Grover's algorithm. Therefore, we might wonder whether GUM is able to always identify a valid solution if such a solution exists? Before answering this question, note that it is certainly possible to construct an example where GUM only has a small probability to find a valid solution. For non-trivial values of n, however, one can be quite confident that GUM will return a valid solution. To explain why, we observe two facts. First, if there are m valid solutions, the probability of measuring a valid solution after i runs can be obtained using Eq. (12). Second, the probability amplitude of the target basis states can be approximated by a sine function (see also Fig. 16 in Appendix A.4). As a result, if we assume that $\hat{m} = 2^n$, $\hat{m} = 2^{(n-1)}$, …, $\hat{m} = 2$, $\hat{m} = 1$ is a random sample of that sine function, $P(n, m, I(n, \hat{m}))$ equals 0.5 on average, and hence, ϕ_n approaches 1, even for small n. This is also illustrated in Table D.8. Table D.8 presents ϕ_i after each of the $(n + 1)$ runs of Grover's algorithm for a system that has $n = 6$ qubits, and for various values of m (the values of m that are used in the table were selected because they have a low probability to yield a valid solution, or because they require a large number of Grover iterations in order to ensure a probability of at least 0.95 to measure a valid solution; see also Fig. 20).

Table D.8 also reports the expected total number of Grover iterations required to measure a valid solution (last column). It is clear that, regardless the value of m, we are almost certain to measure a valid solution. In the worst case, $m = 59$, and we have to perform $(n + 1) = 7$ runs of Grover's algorithm to be 97.9 percent certain to measure a valid solution (this is the smallest value of ϕ_n over all values of m). In other words, even for small n, we are almost certain to measure a valid solution. If, on the other hand, no valid solution exists (i.e., if $m = 0$), we are 100 percent certain that no valid solution will be found after performing 7 runs (requiring a total of $1 + 1 + 2 + 2 + 3 + 4 + 6 = 19$ Grover iterations). Note that, if $m > 0$, we expect to perform far less Grover iterations. This illustrates why GUM may require far less than $O(\sqrt{m^{-1}2^n})$ calls to function $f_{\mathbf{x}}$ (i.e., its expected performance is far better than its worst-case performance).

Appendix E. Assessing performance of a quantum algorithm

The most straightforward way to assess the performance of a quantum algorithm is to implement the algorithm on a quantum computer (or a simulation thereof). Current quantum computers and simulators, however, are limited to solving trivial optimization problems. In addition, access to current quantum computers is limited and/or expensive. Therefore, in this paper, we adopt a different approach (for a tutorial on how to implement quantum algorithms on existing quantum computers/simulators, refer to, for example, the documentation of IBM's Qiskit or Google's Cirq). Rather than implementing the full circuit of a quantum algorithm, we calculate the probability of each possible outcome of the subcircuits (i.e., subroutines) used by the algorithm. Using these probabilities, we can model the transitions between different system states that result from applying the algorithm. In our case, rather than implementing Grover's algorithm, we use Algorithm 7 to obtain $P(n, m, I)$; the probability of measuring one of the m valid solutions in a system that has 2^n valid solutions after performing I Grover iterations. Using $P(n, m, I)$, we can determine the probability to transition from one system state to another. We propose two methods to assess the performance of a quantum algorithm: the exact evaluation of the performance using a Discrete-Time Markov Chain (DTMC) and

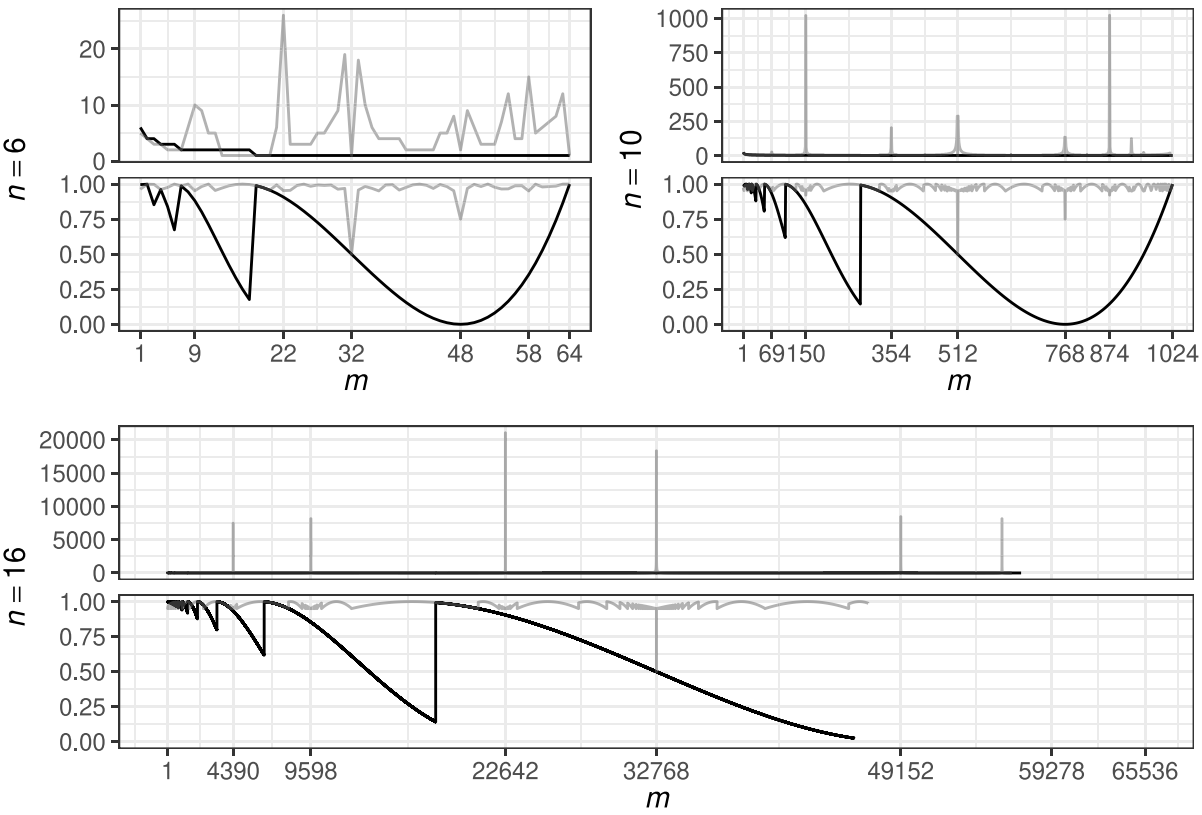


Fig. 20. Comparison of the performance of approximation $\pi 4^{-1}\sqrt{m^{-1}2^n}$ (black line; implemented as $I(n,m)$) and procedure GIρ (gray line; with $\rho = 0.95$). For each value $n : n \in \{6, 10, 16\}$, the top panel shows the number of Grover iterations whereas the bottom panel shows the probability to measure a target basis state. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table D.8
Probability to measure one of the m valid solutions after i runs of Grover's algorithm in a system that has 6 qubits and expected number of Grover iterations (E[GI]) required to successfully measure a valid solution, for various values of m. The worst performance (over all values of m) corresponds to the case where $m = 59$, with a probability of 0.979 to measure a valid solution after $(n+1)$ runs; after a total of 19 Grover iterations (indicated in red).

i	0	1	2	3	4	5	6	
$\hat{m}$	64	32	16	8	4	2	1	
$I(n,\hat{m})$	1	1	2	2	3	4	6	
m				ϕ_i				E[GI]
0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	19.00
1	0.135	0.251	0.509	0.678	0.868	0.976	1.000	5.981
3	0.371	0.604	0.916	0.982	1.000	1.000	1.000	2.644
6	0.646	0.875	1.000	1.000	1.000	1.000	1.000	1.605
9	0.836	0.973	0.997	1.000	1.000	1.000	1.000	1.229
17	0.997	1.000	1.000	1.000	1.000	1.000	1.000	1.003
22	0.908	0.991	0.991	0.991	0.999	0.999	1.000	1.158
31	0.547	0.795	0.881	0.931	0.973	0.983	0.988	2.290
32	0.500	0.750	0.875	0.938	0.969	0.984	0.992	2.508
33	0.453	0.701	0.874	0.947	0.968	0.988	0.996	2.690
48	0.000	0.000	0.750	0.938	0.984	0.984	0.996	4.770
58	0.354	0.583	0.583	0.583	0.719	0.969	0.981	5.511
59	0.436	0.682	0.689	0.696	0.745	0.921	0.979	4.829
64	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

the simulation of this DTMC using Monte Carlo simulation. In what follows, we use both methods to assess the performance of BSP.

A DTMC can be used to model the evolution of a quantum algorithm starting from an initial state (i.e., the start of the algorithm) until we end up in an absorbing state (i.e., the completion of the algorithm). In case of BSP, a state may be defined by tuple $(V^*, V_{\min}, V_{\max}, \hat{m}, \text{GI})$, where GI represents the number of Grover iterations that have been performed to reach state $(V^*, V_{\min}, V_{\max}, \hat{m}, \text{GI})$. When departing from state $(V^*, V_{\min}, V_{\max}, \hat{m}, \text{GI})$, the possible transitions and their probabilities are listed in Table E.9. If $V_{\min} > V_{\max}$, the algorithm stops and we end up in an absorbing state. If, on the other hand, $V_{\min} \leq V_{\max}$, there are three possibilities. If a valid solution is found, we transition to a state where V^* and $V_{\min}$ are updated. If no valid solution is found, we either transition to a state where $\hat{m}$ is updated or where $V_{\max}$ and $\hat{m}$ are updated (if $\hat{m} = 1$). The probability to find a valid solution in state $(V^*, V_{\min}, V_{\max}, \hat{m}, \text{GI})$ is given by $P(n, m_V, I(n,\hat{m}))$, where m_V

Table E.9
Possible DTMC transitions when departing from state $(V^*, V_{\min}, V_{\max}, \hat{m}, \text{GI})$.

resulting state	transition probability	conditions
$(V^*, V_{\min}, V_{\max}, \hat{m}, \text{GI})$	1	$V_{\min} > V_{max}$
$(V, (V+1), V_{\max}, 2^n, (\text{GI} + I(n, \hat{m})))$	$P(n, m_V, I(n, \hat{m}))$	if $V_{\min} \leq V_{\max}$
$(V^*, V_{\min}, V_{\max}, \hat{m}/2, (\text{GI} + I(n, \hat{m})))$	$1 - P(n, m_V, I(n, \hat{m}))$	$V_{\min} \leq V_{\max} \wedge \hat{m} > 1$
$(V^*, V_{\min}, (V-1), 2^n, (\text{GI} + I(n, \hat{m})))$	$1 - P(n, m_V, I(n, \hat{m}))$	if $V_{\min} \leq V_{\max} \wedge \hat{m} = 1$

Table E.10
DTMC that assesses the performance of BSP when solving the example binary knapsack problem. For each state of the DTMC we list the visit probability, the tuple, the transition if a solution is found, the transition if no solution is found, and the number of function calls performed to reach the state. The initial state is indicated in gray and absorbing states are indicated in blue. Probabilities are expressed as percentages.

state	visit	departure state tuple					target	transition	target	transition	function
	probability	V^*	$V_{\min}$	$V_{\max}$	$\hat{m}$	GI	state	probability	state	probability	calls
1	100	0	0	6	8	0	2	100	NA	0	0
2	100	3	4	6	8	1	3	78.13	8	21.88	3
3	78.125	5	6	6	8	2	NA	0	4	100	6
4	78.125	5	6	6	4	3	NA	0	5	100	9
5	78.125	5	6	6	2	4	NA	0	6	100	12
6	78.125	5	6	6	1	6	NA	0	7	100	17
7	78.125	5	6	5	8	8	7	100	NA	0	22
8	21.875	3	4	6	4	2	9	78.13	14	21.88	6
9	17.08984	5	6	6	8	3	NA	0	10	100	9
10	17.08984	5	6	6	4	4	NA	0	11	100	12
11	17.08984	5	6	6	2	5	NA	0	12	100	15
12	17.08984	5	6	6	1	7	NA	0	13	100	20
13	17.08984	5	6	5	8	9	13	100	NA	0	25
14	4.785156	3	4	6	2	3	15	94.53	20	5.469	9
15	4.523468	5	6	6	8	5	NA	0	16	100	14
16	4.523468	5	6	6	4	6	NA	0	17	100	17
17	4.523468	5	6	6	2	7	NA	0	18	100	20
18	4.523468	5	6	6	1	9	NA	0	19	100	25
19	4.523468	5	6	5	8	11	19	100	NA	0	30
20	0.261688	3	4	6	1	5	21	94.53	26	5.469	14
21	0.247377	5	6	6	8	7	NA	0	22	100	19
22	0.247377	5	6	6	4	8	NA	0	23	100	22
23	0.247377	5	6	6	2	9	NA	0	24	100	25
24	0.247377	3	4	4	1	11	NA	0	25	100	30
25	0.247377	5	6	5	8	13	25	100	NA	0	35
26	0.014311	3	4	4	8	7	27	78.13	28	21.88	19
27	0.011181	4	5	4	8	8	27	100	NA	0	22
28	0.003131	3	4	4	4	8	29	78.13	30	21.88	22
29	0.002446	4	5	4	8	9	29	100	NA	0	25
30	0.000685	3	4	4	2	9	31	94.53	32	5.469	25
31	0.000647	4	5	4	8	11	31	100	NA	0	30
32	0.000037	3	4	4	1	11	33	94.53	34	5.469	30
33	0.000035	4	5	4	8	13	33	100	NA	0	35
34	0.000002	3	4	3	8	13	34	100	NA	0	35

denotes the number of valid solutions that have a value of at least $V = \lfloor 0.5(V_{\min} + V_{\max}) \rfloor$.

We now illustrate how to use a DTMC to assess the performance of BSP when solving the example binary knapsack problem. Table E.10 lists the states, the probabilities to visit each of the states, and the transitions that are possible from each state. For instance, we have a 4.785 percent probability to visit state 14 (with $V^* = 3$, $V_{\min} = 4$, $V_{\max} = 6$, $\hat{m} = 2$, and GI = 3). In state 14, $V = 5$ and there is only one valid solution (i.e., $m_V = 1$). As a result, if we perform $I(n, \hat{m}) = I(3, 2) = 2$ Grover iterations, we have a 94.53 percent probability to measure a valid solution and to transition to state 15 (with updated $V^* = 5$ and $V_{\min} = 6$). There is, however, also a 5.469 percent probability that we transition to state 20 (with updated $\hat{m} = 1$). By observing the probabilities to end up in each of the absorbing states, we can determine: (1) the best-case number of required Grover iterations, (2) the worst-case number of required Grover iterations, and (3) the expected number of required Grover iterations. For the example binary knapsack project, we require 8, 13, and 8.319 Grover iterations, respectively. The probabilities to end up in each of the absorbing states can also be used to verify the probability to identify an optimal solution (which amounts to 99.9857 percent for the example binary knapsack problem).

Next to using an exact DTMC approach, we can also use Monte Carlo simulation to approximate the performance of a quantum algorithm. In this case, random numbers are drawn to determine the transitions between system states of the DTMC. A simulation iteration starts in an initial state (that corresponds to the start of the algorithm) and ends in an absorbing state (that corresponds to the completion of the algorithm). After performing sufficient simulation iterations, we can approximate the probability to end up in each of the absorbing states. In turn, these probabilities allow us to approximate the expected performance of the quantum algorithm as well as the probability that an optimal solution is identified. In case of BSP, the state of the system may once again be defined by tuple $(V^*, V_{\min}, V_{\max}, \hat{m}, \text{GI})$ and all possible transitions are listed in Table E.9. In any simulation iteration, if $V_{\min} > V_{\max}$, we reached an absorbing state and the simulation iteration is finished. If, on the other hand, $V_{\min} \leq V_{\max}$, we draw a random number and verify whether this random number is larger than $P(n, m_V, I(n, \hat{m}))$. If the random number is larger, no valid solution was found, and a transition is made towards a state where we update $\hat{m}$ and/or $V_{\max}$. Otherwise, if the random number is smaller-than-or-equal-to $P(n, m_V, I(n, \hat{m}))$, a valid solution was found, and we transition towards a state where we update V^* and $V_{\min}$. Note that, if a valid solution is found in state $(V^*, V_{\min}, V_{\max}, \hat{m}, \text{GI})$ (with probability $P(n, m_V, I(n, \hat{m}))$), we can obtain this solution by randomly drawing a solution from the set of m_V valid solutions (each valid solution has probability m_V^{-1} of being measured).

Next, we illustrate how Monte Carlo simulation can be used to approximate the performance of BSP when solving the example binary knapsack problem. Fig. 21 provides an example of a single simulation iteration. In each simulation iteration, we start from state $(V^*, V_{\min}, V_{\max}, \hat{m}, \text{GI}) = (0, 0, 6, 8, 0)$. In this state, $V = 3$, $m_V = 2$,

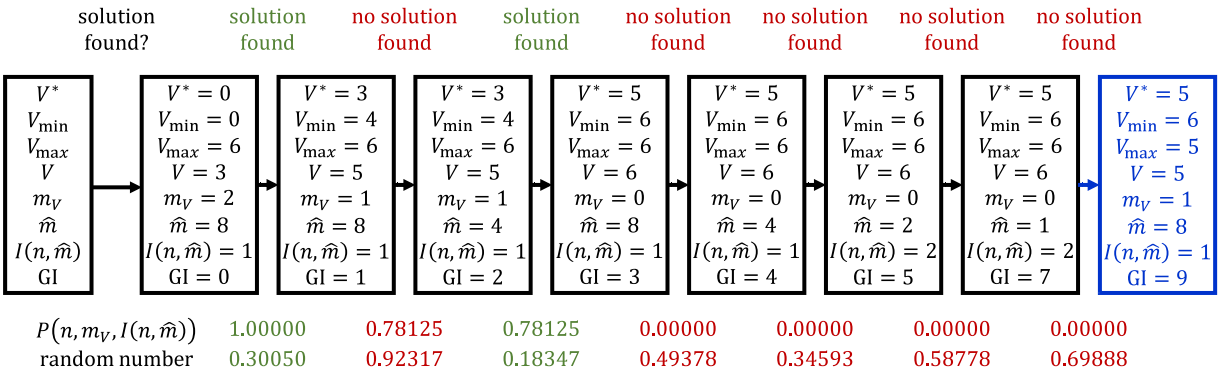


Fig. 21. Example simulation iteration when using Monte Carlo simulation to assess the performance of BSP when solving the example binary knapsack problem. For each run of Grover's algorithm, we compare the probability of measuring a valid solution (i.e., $P(n,m,I(n,\widehat{m}))$) and a random number. If the random number if smaller-than-or-equal-to $P(n,m,I(n,\widehat{m}))$, a valid solution is found (indicated in green), otherwise no solution is found (indicated in red). The simulation completes if $V_{\min} > V_{\max}$ (indicated in blue). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

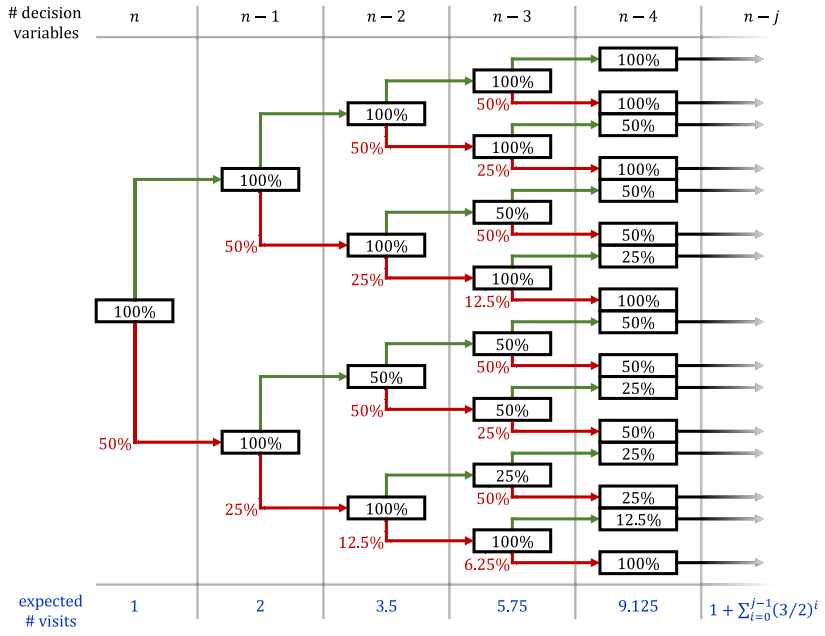


Fig. 22. Illustration of upper bound on the number of operations required by HBB when solving a discrete optimization problem that has n binary decision variables. Each box represents a node in the search tree that may be visited by HBB. The probability to visit a node is displayed inside the box. If a decision variable is assigned a value of 1, we proceed to the next node if GUM can find a solution that still improves the best-found solution (green arrow). If a decision variable is assigned a value of 0, we automatically proceed to the next node, however, the probability that we can still find a solution that improves the best-found solution decreases by 50% (red arrows; success probabilities are indicated in red). The expected number of visited nodes (for each number of remaining decision variables) is also reported (indicated in blue). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

and we are certain to identify a valid solution after a single iteration of Grover's algorithm (i.e., $P(n,m_V,I(n,\widehat{m})) = P(3,2,1) = 1$). As a result, regardless of the random number that is drawn, we transition to state $(3,4,6,8,1)$. In this state, $V = 5$, $m_V = 1$, and there is only a 78.125 percent chance that we identify a valid solution. There are two options. First, if a random number is drawn that is larger than 0.78125, we say that no solution was found and transition to state $(3,4,6,4,2)$. If, on the other hand, we draw a random number smaller-than-or-equal-to 0.78125, we say that a solution was found and we transition to state $(5,6,6,8,2)$. This process continues until we end up in an absorbing state (state $(5,6,6,8,1,9)$ in Fig. 21). After performing a sufficient number of iterations, we can once again approximate: (1) the best-case number of Grover iterations, (2) the worst-case number of Grover iterations, and (3) the expected number of Grover iterations. In addition, we can also approximate the probability to identify an optimal solution by keeping track of how often an optimal solution value was observed at the end of a simulation iteration.

Both the DTMC and the Monte Carlo simulation method allow to analyze the performance of a quantum algorithm. Whereas Monte Carlo simulation provides an approximation of the performance, a DTMC can be used to obtain exact results. Both methods, however, have one drawback: we have to be able to determine the probabilities to transition from one system state to another. In the case of Grover-based algorithms (i.e., in our case), this implies that we need to determine $P(n,m_V,I(n,\widehat{m}))$. In order to determine $P(n,m_V,I(n,\widehat{m}))$, however, we need to know m_V; we need to know the number of valid solutions that have a value of at least V. In general, we do not know m_V. In this paper, we use a brute-force approach to calculate m_V for each possible value of V. With 2^n possible solutions, this is still feasible for n up to 30. For larger n, however, a brute-force approach would

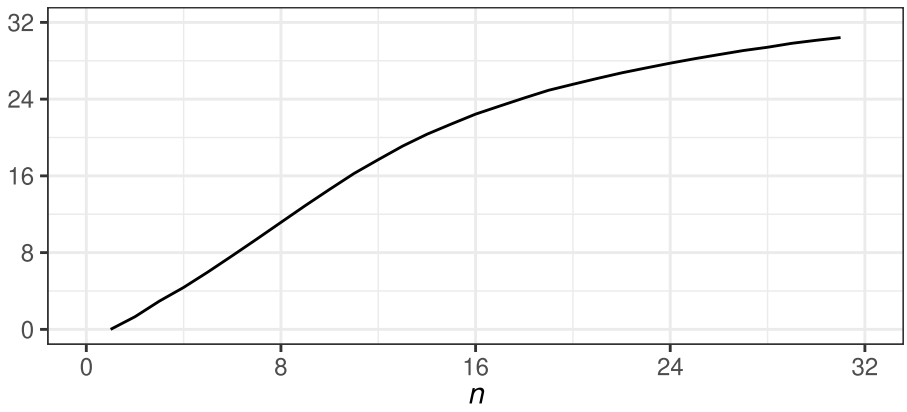


Fig. 23. Worst-case expected number of operations divided by $n\sqrt{2^n}$ required by procedure HBB for each value of $n : 1 \leq n \leq 31$.

no longer be feasible (the use of memory and/or computation times becomes prohibitive). Therefore, for larger n, approximation algorithms may be used to approximate m_V. For instance, Monte Carlo simulation can be used to draw a sample of random solutions that can be used to approximate the distribution of m_V for different values of V. The larger the sample, the more accurate our estimate of m_V will be. The investigation of such approximations, as well as the study of other methods to assess the performance of quantum algorithms, is left as a direction for future research.

Appendix F. Complexity of HBB

In Section 6, we have seen that RAP expects to require at most $O^*(\sqrt{2^n})$ operations to solve a discrete optimization problem that has n binary decision variables. Unfortunately, in contrast to RAP, it is more difficult to determine the exact number of operations required by HBB. We can, however, define the following upper bound:

$$\sum_{j=1}^{n-1} \mu_{(n-j)}^{\texttt{GUM}} \left(1 + \sum_{i=0}^{j-1} (3/3)^i\right). \tag{13}$$

where $\mu_{(n-j)}^{\texttt{GUM}}$ represents the worst-case number of operations required by GUM when searching a system that has $2^{(n-j)}$ solutions. To understand how to obtain this upper bound, consider Fig. 22. Given n binary decision variables, Fig. 22 displays the probability to visit each node (represented by a box) in the search tree that is used by HBB. In addition, the figure also reports the probability to visit a node (i.e., the percentage inside the box) as well as the probability that GUM can find a solution that can still improve the best solution that was found so far (represented by the probabilities in red). Starting from the root node, we let $x_1 = 1$ (i.e., we assign a value of 1 to the first decision variable) and end up in a node that considers $(n-1)$ remaining decision variables (following the green arrow). In this node, we perform GUM on a system that has $(n-1)$ remaining decision variables. At this point we are almost certain that GUM will find a solution that improves the best-found solution (i.e., the solution where $x_1 = 1$ and $x_i = 0$ for all $1 < i < n$). As such, we assume we are 100% certain to visit the next node where we consider $(n-2)$ remaining decision variables (and where $x_1 = 1$ and $x_2 = 1$). If once again, we assume that we are 100% certain that GUM will find a better solution, we end up in a node where we consider $(n-3)$ decision variables. This process continues until we are at the bottom of the tree. At the bottom of the tree, we climb back up. Eventually, we return to the root node, where we now have explored all solutions for which $x_1 = 1$. Next, we let $x_1 = 0$ and end up in a node that has $(n-1)$ remaining decision variables (following the red arrow). In this node, we perform GUM on a system that has $(n-1)$ remaining decision variables. At this point, we are only 50% certain (probability in red) that GUM can still find a solution that improves the best-found solution (as we have a 50% probability that an optimal solution was found in the first half of the tree; note that we assume that there is no structure in the solution space, that each solution has a unique solution value, and that an optimal solution can be located anywhere with equal probability). As such, we only have a 50% probability to end up in the node where $x_1 = 0$, $x_2 = 1$, and we are left with $(n-2)$ decision variables. Conversely, when letting $x_2 = 0$, the probability to find a solution that still improves the best-found solution decreases from 50% to 25%. As a result, it becomes less and less likely that nodes in the right/bottom of the tree are visited. In fact, for each remaining number of decision variables $(n-j)$, we can express the expected number of nodes that are visited by:

$$1 + \sum_{i=0}^{j-1} (3/2)^i. \tag{14}$$

For instance, in the worst case, we expect to visit 5.75 nodes that perform GUM on a system that has $(n-3)$ remaining decision variables (i.e., the sum of probabilities to visit nodes that have $(n-3)$ remaining decision variables).

From Eq. (13), it follows that, in the worst case, HBB expects to perform at most $O^*((3/2)^2)$ operations. Note that this is an upper bound that does not take into account: (1) the fact that we are not always 100% certain that GUM will find a better solution and (2) GUM does not perform the worst-case number of operations in each node. Therefore, to better define the expected worst-case complexity of HBB, we perform a computational experiment. In the experiment, we use HBB to find the optimal solution of discrete optimization problems that have $n : n \in \{1, \ldots, 31\}$ binary decision variables. For each value of n, we analyze 10000 problem instances. For each problem instance, we assign a random unique value to each of the 2^n solutions (i.e., the generated instances have no structure that can be exploited by HBB). We use HBB to solve all problem instances and keep track of the number of operations that are required to find the optimal solution (and to prove that it is optimal). The results of the experiment are summarized in Fig. 23 that displays the average number of required operations divided by $n\sqrt{2^n}$. We can see that, as n increases, the number of operations divided by $n\sqrt{2^n}$ converges (similar to what can be seen in Fig. 7 for RAP). As a result, we conjecture that HBB has expected worst-case performance $O(n\sqrt{2^n})$ when solving discrete optimization problems that have n binary decision variables.

Note that, a classical branch-and-bound procedure that uses a depth-first search strategy requires $O^*(2^n)$ operations to solve a discrete optimization problem that has n binary decision variables. Procedure HBB, however, only requires $O^*(\sqrt{2^n})$. This quadratic speedup is obtained because, rather than exploring a branch, HBB can use GUM to "peek ahead", and to see whether a solution can still be found in the

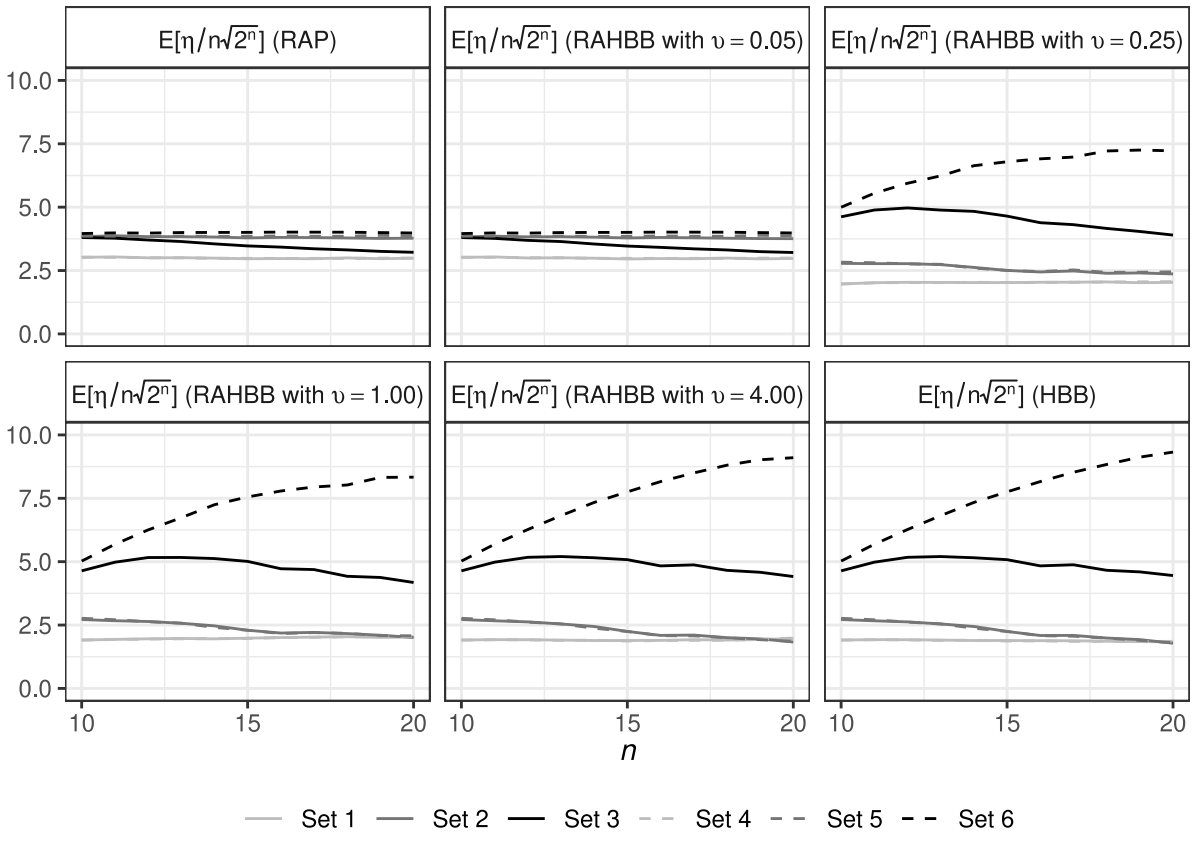


Fig. 24. Expected number of operations divided by $n\sqrt{2^n}$ required to solve a binary knapsack problem using HBB, RAP, and RAHBB for various values of υ, when solving 1000 binary knapsack problems for each problem set, for each value of $n : 10 \leq n \leq 20$ (lines of sets 1 and 4 are light gray, sets 2 and 5 are dark gray, sets 3 and 6 are black, sets 1, 2, and 3 are full, and sets 4, 5, 6 are dashed). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

branch that improves the best-found solution. The cost of "peeking ahead" is limited thanks to the quadratic speedup that is obtained by Grover's algorithm (used in GUM) when searching an unstructured set of solutions. It would be interesting to develop procedures that determine when to best use GUM to peek ahead when searching a tree. Such procedures could be used to further reduce the number of operations required by HBB.

Appendix G. Combining procedures HBB and RAP

In this section, we combine both HBB and RAP. Whereas HBB has the disadvantage that a significant amount of time is spent on proving that a (partial) solution is optimal, it has the advantage that it can exploit the structure of a problem (resulting in the removal of subsets of solutions from the search space). RAP, on the other hand, minimizes the time spent on proving that a solution is optimal, however, it always searches the entire solution space. To combine the strengths of both procedures, HBB can be equipped with RAP at deeper levels of the search tree. This way: (1) we benefit from the speedup created by the branch-and-bound procedure at higher levels of the search tree, (2) we spend less time to prove that a partial solution is optimal in deeper levels of the tree, and (3) we also counter the drawback of RAP because it is used to find a partial solution in only a subset of the solution space. In addition, note that at higher levels of the search tree, there is not so much difference between the searches of HBB and RAP (i.e., at higher levels of the tree, both HBB and RAP search the same solution space). As a result, it makes sense to use RAP mainly at deeper levels of the search tree.

Algorithm 8 (hereafter referred to as RAHBB) presents one possible implementation that uses RAP if the number of Grover iterations required by HBB exceeds a given threshold $n\upsilon$. Procedure RAHBB also uses upper bound UBi to fathom nodes and is equipped with GGUM rather than with GUM. We use RAHBB to solve the 66000 instances of the binary knapsack problem assuming the same settings as those used in the "full case" discussed in Section 7 (i.e., we use a lower bound heuristic to set an initial solution, we let $\tau = 4$, we use upper-bound procedure UBi to fathom nodes, and we use the number of valid solutions of the last search as the initial number of valid solutions in the subsequent search). The results are summarized in Fig. 24. Fig. 24 presents the number of operations (divided by $n\sqrt{2^n}$) that are required by RAHBB for various values of υ. Fig. 24 also reports the performance of RAP (top left panel) and HBB (bottom right panel). We can observe that, for sufficiently small υ, the performance of RAHBB is almost identical to that of RAP (i.e., RAHBB skips HBB, and immediately performs RAP). On the other hand, if υ is sufficiently large, RAHBB is identical to HBB (i.e., RAHBB never performs RAP). Fig. 24 shows that, depending on our ability to exploit the structure of the problem, a combination of HBB and RAP may be used to reduce the overall expected number of operations.

Algorithm 8: Hybrid branch-and-bound procedure that uses a random-ascent procedure if the number of required Grover iterations exceeds $n\upsilon$. Both procedures use GGUM to measure a valid solution.

initialize $\mathbf{x}^*$ and V^* using lower bound heuristic and start procedure RAHBB$(\emptyset, 0, W, 2^n, 1)$;
Procedure RAHBB$(\mathbf{x}, V, R, \hat{m}, i)$
  if UBi$(V, R, i) > V^*$ **then**
    if $n - i < \omega$ **then**
      use classical algorithm to find $\mathbf{x}^\star_{in} \in \Omega_{in}$ that has highest value $V^\star_{in} > V^* - V$ and $W^\star_{in} \leq R$;
      if $\mathbf{x}^\star_{in} \neq \emptyset$ **then**
        update best-found solution $\mathbf{x}^* = \mathbf{x}_{1(i-1)} \cup \mathbf{x}^\star_{in}$ and its value $V^* = V + V^\star_{in}$;
    else if $I(n - i, \hat{m}) \leq n\upsilon$ **then**
      if $w_i \leq R$ **then**
        Let $x_i = 1$, $\mathbf{x}_{1i} = \mathbf{x}_{1(i-1)} \cup \{x_i\}$, $V' = V + v_i$, and $R' = R - w_i$;
        Use GGUM$(n - i, \hat{m})$ to obtain $\hat{m}'$ and to measure $\mathbf{x}_{(i+1)n} \in \Omega_{(i+1)n}$ that has value $V_{(i+1)n} > V^* - V'$ and $W_{(i+1)n} \leq R'$;
        if $\mathbf{x}_{(i+1)n} \neq \emptyset$ **then**
          update best-found solution $\mathbf{x}^* = \mathbf{x}_{1i} \cup \mathbf{x}_{(i+1)n}$ and its value $V^* = V' + V_{(i+1)n}$;
          RAHBB$(\mathbf{x}_{1i}, V', R', \hat{m}', i + 1)$;
      Let $x_i = 0$, $\mathbf{x}_{1i} = \mathbf{x}_{1(i-1)} \cup \{x_i\}$, and run RAHBB$(\mathbf{x}, V, R, \hat{m}, i + 1)$;
    else
      do
        Use GGUM$(n - i, \hat{m})$ to obtain $\hat{m}'$ and to measure $\mathbf{x}_{(i+1)n} \in \Omega_{(i+1)n}$ that has value $V_{(i+1)n} > V^* - V'$ and $W_{(i+1)n} \leq R'$;
        if $\mathbf{x}_{(i+1)n} \neq \emptyset$ **then**
          Let $\hat{m} = \hat{m}'$ and update best-found solution $\mathbf{x}^* = \mathbf{x}_{1i} \cup \mathbf{x}_{(i+1)n}$ and its value $V^* = V' + V_{(i+1)n}$;
      while $\mathbf{x} \neq \emptyset$;

Appendix H. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.ejor.2024.12.016.

References

Aaronson, S. (2018). *Lecture notes, Introduction to quantum information science*. UT Austin.

Abbas, A., Ambainis, A., Augustino, B., Bärtschi, A., Buhrman, H., Coffrin, C., Cortiana, G., Dunjko, V., Egger, D. J., Elmegreen, B. G., Franco, N., Fratini, F., Fuller, B., Gacon, J., Gonciulea, C., Gribling, S., Gupta, S., Hadfield, S., Heese, R., ... Zoufal, C. (2023). Quantum optimization: Potential, challenges, and the path forward. preprint at http://dx.doi.org/10.48550/arXiv.2312.02279.

Aharonov, D. (2003). A simple proof that toffoli and Hadamard are quantum universal. preprint at http://dx.doi.org/10.48550/arXiv.quant-ph/0301040.

Albash, T., & Lidar, D. A. (2018). Adiabatic quantum computation. *Reviews of Modern Physics*, *90*(1), Article 015002.

Ambainis, A., Balodis, K., Iraids, J., Kokainis, M., Prūsis, K., & J., Vihrovs (2018). Quantum speedups for exponential-time dynamic programming algorithms. preprint at http://dx.doi.org/10.48550/arXiv.1807.05209.

Ambainis, A., & Kokainis, M. (2022). Quantum algorithm for tree size estimation, with applications to backtracking and 2-player games. preprint at http://dx.doi.org/10.48550/arXiv.1704.06774.

Babbush, R., McClean, J. R., Newman, M., Gidney, C., Boixo, S., & Neven, H. (2021). Focus beyond quadratic speedups for error-corrected quantum advantage. *PRX Quantum*, *2*(1), Article 010103.

Benioff, P. (1980). The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by turing machines. *Journal of Statistical Physics*, *22*(5), 563–591.

Bennett, C. H., Bernstein, E., Brassard, G., & Vazirani, U. (1997). Strengths and weaknesses of quantum computing. *SIAM Journal on Computing*, *26*(5), 1510–1523.

Blekos, K., Brand, D., Ceschini, A., Chou, C.-H., Li, R.-H., Pandya, K., & Summer, A. (2023). A review on quantum approximate optimization algorithm and its variants. preprint at http://dx.doi.org/10.48550/arXiv.2306.09198.

Born, M. (1926). Zur Quantenmechanik der Stoßvorgänge. *Zeitschrift für Physik*, *37*(12), 863–867.

Boyer, M., Brassard, G., Høyer, P., & Tapp, A. (1996). Tight bounds on quantum searching. In *Proceedings of the fourth workshop on physics and computation*. New England Complex Systems Institute.

Brassard, G. (1997). Searching a quantum phone book. *Science*, *275*(5300), 627–628.

Cacchiani, V., Iori, M., Locatelli, A., & Martello, S. (2022a). Knapsack problems — An overview of recent advances. Part I: Single knapsack problems. *Computers & Operations Research*, *143*, Article 105692.

Cacchiani, V., Iori, M., Locatelli, A., & Martello, S. (2022b). Knapsack problems — An overview of recent advances. Part II: Multiple, multidimensional, and quadratic knapsack problems. *Computers & Operations Research*, *143*, Article 105693.

Campbell, E., Khurana, A., & Montanaro, A. (2019). Applying quantum algorithms to constraint satisfaction problems. *Quantum*, *3*(167).

Chakrabarti, S., Minssen, P., Yalovetzky, R., & Pistoia, M. (2022). Universal quantum speedup for branch-and-bound, branch-and-cut, and tree-search algorithms. preprint at http://dx.doi.org/10.48550/arXiv.2210.03210.

Cheng, B., Deng, X. H., Gu, X., He, Y., Hu, G., Huang, P., Li, J., Lin, B. C., Lu, D., Lu, Y., Qiu, C., Wang, H., Xin, T., Yu, S., Yung, M. H., Zeng, J., Zhang, S., Zhong, Y., Peng, X., ... Yu, D. (2023). Noisy intermediate-scale quantum computers. *Frontiers in Physics*, *18*.

Cleve, R., Ekert, A., Macchiavello, C., & Mosca, M. (1998). Quantum algorithms revisited. *Proceedings of The Royal Society of London. Series A. Mathematical, Physical and Engineering Sciences*, *454*(1969), 339–354.

Creemers, S., & Pérez, L. F. (2023). Discrete optimization: Limitations of existing quantum algorithms. preprint at http://dx.doi.org/10.2139/ssrn.4527268.

Dantzig, G. B. (1957). Discrete-variable extremum problems. *Operations Research*, *5*(2), 266–288.

Deutsch, D. (1985). Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of The Royal Society of London. Series A. Mathematical, Physical and Engineering Sciences*, *400*(1818), 97–117.

Deutsch, D. (1989). Quantum computational networks. *Proceedings of The Royal Society of London. Series A. Mathematical, Physical and Engineering Sciences*, *425*(1868), 73–90.

Deutsch, D., & Jozsa, R. (1992). Rapid solution of problems by quantum computation. *Proceedings of The Royal Society of London. Series A. Mathematical, Physical and Engineering Sciences*, *439*(1907), 553–558.

Dirac, P. A. M. (1939). A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, *35*(3), 416–418.

Draper, T. G. (2000). Addition on a quantum computer. preprint at http://dx.doi.org/10.48550/arXiv.quant-ph/0008033.

Farhi, E., Goldstone, J., & Gutmann, S. (2014). A quantum approximate optimization algorithm, preprint at http://dx.doi.org/10.48550/arXiv.1411.4028.

Fennich, M. E., Fomeni, F. Djeumou., & Coelho, L. C. (2024). A novel dynamic programming heuristic for the quadratic knapsack problem. *European Journal of Operational Research*, *319*(1), 102–120.

Feynman, R. P. (1982). Simulating physics with computers. *International Journal of Theoretical Physics*, *21*(6–7), 467–488.

Glover, F., Kochenberger, G., & Hennig, R. (2022). Quantum bridge analytics I: a tutorial on formulating and using QUBO models. *Annals of Operations Research*, *314*(1), 141–183.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *Proc. annu. ACM symp. theory comput.* (pp. 212–219).

Grover, L. K., & Radhakrishnan, J. (2005). Is partial quantum search of a database any easier? In *Proceedings of the 17th annual ACM symposium on parallelism in algorithms and architectures* (pp. 186–194).

Hidary, J. D. (2021). *Quantum computing: an applied approach*. Switzerland AG, Cham, Switzerland: Springer Nature.

Kellerer, H., Pferschy, U., & Pisinger, D. (2004). *Knapsack problems*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Kurowski, K., Pecyna, T., Slysz, M., Rózycki, R., Waligóra, G., & Węglarz, J. (2023). Application of quantum approximate optimization algorithm to job shop scheduling problem. *European Journal of Operational Research*, *310*(2), 518–528.

MacQuarrie, E. R., Simon, C., Simmons, S., & Maine, E. (2020). The emerging commercial landscape of quantum computing. *Nature Reviews Physics*, *2*, 596–598.

Markevich, E. A., & Trushechkin, A. S. (2019). Quantum branch-and-bound algorithm and its application to the travelling salesman problem. *Mathematical Sciences*, *241*, 168–184.

McKinsey (2021). *Quantum computing: an emerging ecosystem and industry use cases*. McKinsey.

McKinsey (2023). *Quantum technology monitor*. McKinsey.

Montanaro, A. (2016). Quantum walk speedup of backtracking algorithms. preprint at http://dx.doi.org/10.48550/arXiv.1509.02374.

Montanaro, A. (2020). Quantum speedup of branch-and-bound algorithms. *Physical Review*, *2*(1), Article 013056.

Nannicini, G. (2020). An introduction to quantum computing, without the physics. *SIAM Review*, *62*(4), 936–981.
Nielsen, M., & Chuang, I. (2010). *Quantum computation and quantum information: 10th anniversary edition*. Cambridge: Cambridge University Press.
Parker, R. G., & Rardin, R. L. (1988). *Discrete optimization*. San Diego: Academic Press, Inc.
Pisinger, D. (2005). Where are the hard knapsack problems? *Computers & Operations Research*, *32*(9), 2271–2284.
Pisinger, D. (2007). The quadratic knapsack problem — a survey. *Discrete Applied Mathematics*, *155*(5), 623–648.
Preskill, J. (2018). Quantum Computing in the NISQ era and beyond. *Quantum 2*, *2*, 79–99.
Shor, P. W. (1994). Algorithms for quantum computation: Discrete logarithms and factoring. *Proc. Annu. Symp. FOCS*, *12*, 4–134.
Wolsey, L. A. (2021). *Integer programming*. Hoboken: Wiley.
Xia, H., Li, H., Zhang, H., Liang, Y., & Xin, J. (2018). An efficient design of reversible multi-bit quantum comparator via only a single ancillary bit. *International Journal of Theoretical Physics*, *57*(12), 3727–3744.
Zalka, C. (1999). Grover's quantum searching algorithm is optimal. *Physical Review A*, *60*(4), 2746–2751.