Quantum Information and Computation, Vol. 20, No.9&10 (2020) 0766-0786

QUANTUM-BASED ALGORITHM AND CIRCUIT DESIGN FOR BOUNDED KNAPSACK OPTIMIZATION PROBLEM

WENJUN HOU MAREK PERKOWSKI
Department of Electrical and Computer Engineering, Portland State University
1900 SW Fourth Ave., Portland, OR 97201, USA

Received December 19, 2019
Revised July 3, 2020

The Knapsack Problem is a prominent problem that is used in resource allocation and cryptography. This paper presents an oracle and a circuit design that verifies solutions to the decision problem form of the Bounded Knapsack Problem. This oracle can be used by Grover Search to solve the optimization problem form of the Bounded Knapsack Problem. This algorithm leverages the quadratic speed-up offered by Grover Search to achieve a quantum algorithm for the Knapsack Problem that shows improvement with regard to classical algorithms. The quantum circuits were designed using the Microsoft Q# Programming Language and verified on its local quantum simulator. The paper also provides analyses of the complexity and gate cost of the proposed oracle. The work in this paper is the first such proposed method for the Knapsack Optimization Problem.

Keywords: Quantum Computing, Grover's Algorithm, Knapsack Problem, Quantum Arithmetic Circuits, Reversible Circuit, Oracle, Q# Language, Complexity Analysis, Gate Count**.**
Communicated by: S Braunstein & K Mølmer

1 Introduction

Quantum superposition and especially entanglement can be manipulated to open pathways to obtaining algorithmic speedup relative to classical algorithms. Many difficult problems have already had quantum implementations created, such as graph coloring [1] and prime factorization [2].

For instance, Grover's Algorithm is a quantum implementation of the unsorted search. Normally, when given a function with domain size N, with one input vector outputting 1 and the rest outputting 0, a classical algorithm would need at most $N-1$ inquiries to determine which of the input values yields the 1. However, Grover's Algorithm performs a search that yields the wanted input value with complexity on the order of $O(\sqrt{N})$ oracle calls, which is a significant improvement if the oracle is made efficiently [3,4,5,6,7].

The Knapsack Problem and its variations are very prominent problems in computing. A 1999 study ranks the Knapsack Problem as the 19th most popular of 75 problems [8]. In addition, the Knapsack Problem has applications in resource allocation [9], the design of complex cryptosystems [10], and

numerous other areas. Because the decision problem form of the Bounded Knapsack Problem is NP-complete, its solution has potential to be optimized using Grover Search, as will be explained later in this paper. Thus, it is interesting to apply Grover Search to solve the optimization problem form of the Bounded Knapsack Problem.

Section 2 introduces and describes the Knapsack Problem and classical methods for its solution. Section 3 explains the Grover Search algorithm and its use with exponential search to solve optimization problems. Section 4 presents a quantum circuit design of a quantum oracle to apply the mentioned method to solve the Knapsack Problem. Sections 5 – 8 describe components of the oracle with detailed quantum circuits. Section 9 compares the complexity of the oracle and its usage with the complexity of existing classical algorithms. Section 10 explains the methods used to test the correctness of the oracle design. Section 11 evaluates the quantum gate cost of the oracle's implementation. Section 12 concludes this paper.

2 Description of the Knapsack Problem

The Knapsack Problem, including its decision and optimization problem forms, will now be briefly presented.

In the problem, there are *n* types of objects, indexed 0 to $n-1$. An object of type i has a weight of w_i and a profit of p_i. One can choose up to b_i instances of type i to include in a knapsack that has a capacity of W. x_i represents the number of instances of type i that are included. The optimization problem form of the Bounded Knapsack Problem seeks a combination vector $[x_0, x_1, \dots, x_{n-1}]$ that maximizes the total profit of the objects in the knapsack, without exceeding the capacity limits.

In the rest of this paper, the term "Knapsack Optimization Problem" will refer to the optimization form of the problem. In terms of equations, the Knapsack Optimization Problem can be described through Eq. (2.1), (2.2), and (2.3).

$$0 \le x_i \le b_i \quad \forall\, i = 0,1,\dots,n-1 \tag{2.1}$$

$$W' = \sum_{i=0}^{n-1} w_i\, x_i \le W \tag{2.2}$$

$$\text{maximize } P' = \sum_{i=0}^{n-1} p_i\, x_i \tag{2.3}$$

The Unbounded Knapsack Problem removes the upper-bound restriction in Eq. (2.1), so that x_i can be any non-negative integer.

The k-Bounded Knapsack Problem is a special case of the Bounded Knapsack Problem. In this variant, each type of object has the same upper bound: $b_i = k$ for each i. For instance, in the 4-Bounded Knapsack Problem, each type of object has a maximum of four instances.

A special case of the k-Bounded Knapsack Problem is the 0-1 knapsack problem [11]. As suggested by the name, the 0-1 Knapsack Problem allows at most one of each type of object, which is equivalent to setting $b_i = 1$ for all i.

Classical computing has several solutions to the Bounded Knapsack Problem.

The simple, straightforward approach is brute force, which iterates through all combination vectors $[x_0, x_1, \dots, x_{n-1}]$ and finds the satisfactory combination that has the greatest profit. The brute force strategy requires a number of operations that is proportional to N, the total number of combination vectors, which is expressed in Eq. (2.4).

$$N = \prod_{i=0}^{n-1}(b_i + 1) \tag{2.4}$$

Dynamic programming has also been used to solve the Knapsack Problem [12]. By using recursion, this algorithm conserves time as compared to the brute force approach. The dynamic programming method for solving the Knapsack Problem requires a time complexity that is expressed in Eq. (2.5).

$$O(nW). \tag{2.5}$$

Although the first factor in the expression is linear with respect to n, the second factor, which is W, is not polynomial with n. In fact, the problem size is typically proportional to $\log W$; therefore, W can easily grow even when n is small, significantly increasing the time complexity. Hence, the Knapsack (Optimization) Problem is classified as NP-hard [11].

In contrast to the Knapsack Optimization Problem, the decision problem form of the Bounded Knapsack Problem seeks whether a combination vector $[x_0, x_1, \dots, x_{n-1}]$ can be found that achieves a profit of at least P_I, without exceeding the capacity limits. The Knapsack Problem is NP-complete, as its solutions can be verified in polynomial time by calculating the expressions in Eq. (2.1) – (2.3). In the rest of this paper, the term "Knapsack Problem", if not specified to be of the decision problem form, will refer to the optimization form of the problem. In addition, the term "Knapsack Decision Problem" will refer to the decision problem form.

Although it is not expected to discover dramatic improvement to the time usage of the solution algorithm for Knapsack Optimization Problem, evaluating the complexity of its quantum variant is of interest.

3 Grover Search and Use in Optimization

Classical computing can search through an unsorted array and locate a specific key in time complexity that grows linearly with the array size. In quantum computing, however, there exists Grover's Algorithm that can perform searching in square root time [7], as long as the time required to perform key verification is minimal. Grover's Search Algorithm can be applied to numerous problems.

An oracle, as used in this paper, is a term that describes a quantum circuit operation with the function of evaluating the output of a Boolean function on a set of qubits, and flipping a separate "target" qubit if that Boolean function is satisfied. An example oracle is shown in Figure 3.1. Performing operations on qubits at the quantum level, Grover Search is given an oracle O_K and executes several Grover Loop Iterations (denoted by G) to be able to find one of the valid input qubit combinations that return 1 after measurement. The gates used for this described process are shown in Figure 3.2. The quantum circuits inside a Grover Loop Iteration are shown in Figure 3.3. This paper will not explore the

specific details of the Grover Search; such descriptions can be found in various papers and textbooks regarding quantum computing [3,4,5,6,7]. But it is important to note that if none of the input qubit combinations are valid, Grover Search will randomly return one of the invalid combinations. Thus, all responses from Grover Search should be verified by calling the oracle on the result.

The Grover Search uses $\frac{\pi}{4}\sqrt{\frac{N}{M}}$ calls to the Grover Loop Iteration, in which N is the search space size, otherwise known as the number of elements to search over, and M is the number of these elements that return 1 after measurement. A proof of this can be found in the textbook of Rieffel and Polak [4]. The quantum counting algorithm [13] can be used to calculate the value M and thus discover how many iterations are needed. One Grover Loop Iteration contains one call to oracle O_K. Thus, the Grover Search calls the oracle $\frac{\pi}{4}\sqrt{\frac{N}{M}}$ times, a time complexity that is proportional to the square root of N.

In an optimization problem, there exist N items that each correspond to a value. We wish to determine which item has the most optimal value. Without loss of generality, let us define the most optimal value as the greatest of the values. This definition allows us to easily relate it to the Knapsack Problem, which involves maximization. It is known that optimization problems can be solved through repeatedly answering the corresponding decision problem.

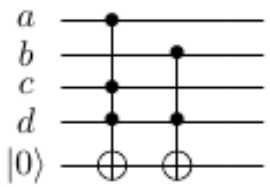

Figure 3.1: Example oracle. This oracle evaluates the expression $acd \oplus bd$ and flips the bottom "target" qubit if the expression is true.

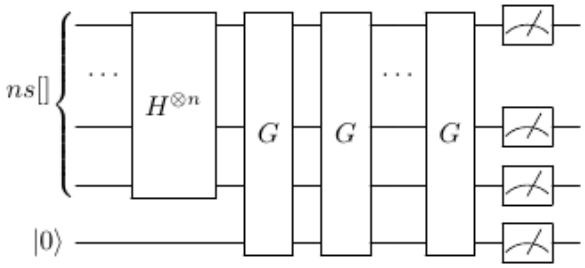

Figure 3.2: The Grover Search, consisting of a Walsh-Hadamard transform followed by $\frac{\pi}{4}\sqrt{\frac{N}{M}}$ calls to G, concluded by measurement.

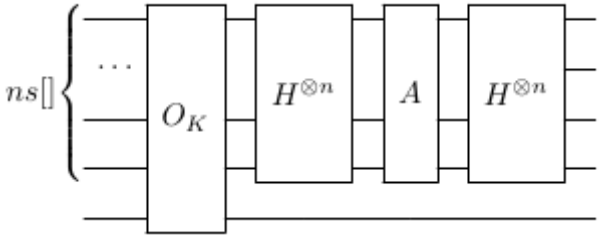

Figure 3.3: Grover Loop Iteration G, including a call to the problem's oracle O_K, followed by amplitude amplification, represented by A, which is called the diffusion operator.

There are a number of well-known pre-existing methods to perform this conversion. One such method is classical exponential search [14], which essentially performs a modified version of binary search on unbounded lists. We will use exponential search as an example to calculate and compare between algorithms' time complexities later in the paper. A detailed description of exponential search can be found in the original paper [14]. It is important to note that exponential search requires a number of comparisons proportional to the logarithm of the position of the sought item. In the context of optimization problems, the "position" of an item is its value. The exponential search identifies the item with the greatest value that solves the decision problem, and this item is the solution to the corresponding optimization problem.

Because the act of answering the decision problem through classical unary search requires iteration over all N items each time, this process of solving optimization problems often has a very high time complexity. However, using Grover Search to solve decision problems in $O(\sqrt{N})$ time significantly reduces the time complexity of solving the corresponding optimization problems, as long as an oracle can be designed to verify a solution to the decision problem in polynomial time.

Therefore, if an oracle can be designed to verify a solution to the Knapsack Decision Problem in polynomial time, then it can be used to solve the Knapsack Optimization Problem with Grover Search.

4 Design of Oracle to Verify Solutions to Knapsack Decision Problem

The method described in the previous section is able to solve the Knapsack Optimization Problem, if an oracle is designed to verify whether a given combination vector $[x_0, x_1, \dots, x_{n-1}]$ is a valid solution to the Knapsack Decision Problem. Qubits are thus needed to represent each x_i value for use in the oracle. For each i, x_i can range from 0 to b_i, which comprise $b_i + 1$ different possibilities. This requires $\lceil log_2(b_i + 1) \rceil$ qubits to represent the possibilities. Thus, the total number of input qubits for the oracle is $\sum_{i=0}^{n-1} \lceil log_2(b_i + 1) \rceil$. This expression does not consider any ancilla qubits used in the oracle.

The search space size of these $\sum_{i=0}^{n-1} \lceil log_2(b_i + 1) \rceil$ qubits is expressed in Eq. (4.1). In the limit in which b_i values are large, the ceiling function may be neglected. Note that the search space size is approximately equal to the number of combination vectors in the Knapsack Problem.

$$N = 2^{\sum_{i=0}^{n-1} \lceil log_2(b_i+1) \rceil} \approx 2^{\sum_{i=0}^{n-1} log_2(b_i+1)} = \prod_{i=0}^{n-1} (b_i + 1) \tag{4.1}$$

The oracle must check three conditions to ensure that a combination vector $[x_0, x_1, \ldots, x_{n-1}]$ is a valid solution:

(i) Ensure that the amount x_i of each object i is contained within the limit determined by b_i, in Eq. (2.1). This process is named "Bounds Checking."

(ii) Ensure that the total weight, W', described by Eq. (2.2) is less than or equal to W. This process is named "Weight Checking."

(iii) Ensure that the total profit, P', described by Eq. (2.3) is greater than the previous profit P_I. This process is named "Profit Maximization."

Both the Weight Checking and Profit Maximization modules in the quantum oracle utilize a custom quantum circuit constant adder function for the summation calculations. A constant adder adds a constant value onto a qubit-array integer. More on this component will be explicated in Section 6.

These three steps are realized through three quantum circuits in the Knapsack Oracle shown in Figure 4.1. Because the requirement is that all three conditions must be satisfied for the maximization problem, the result of each part is recorded on one of three separate ancilla qubits. These ancilla qubits are then used to control a multi-control Toffoli gate. In this way, the state of the bottom qubit will represent the validity of the input combination vector $[x_0, x_1, \ldots, x_{n-1}]$.

In order to return input qubits and ancilla qubits with their original values, the operations B, W, and P that are used to calculate the validity must be mirrored. To do this, the adjoints of B, W, and P, which are B^{-1}, W^{-1}, and P^{-1}, in which all gates are applied in reverse order, are also applied in reverse order.

Please note that we constructed the oracle bottom-up from realistic quantum gates rather than decomposing a unitary matrix to gates that may be difficult to realize. This method requires adding $\max(n, 2I) + 3$ ancilla bits (I represents the number of bits used to store the profit), which is a small price to pay for the ability to achieve a realistic cost of quantum circuits in the oracle.

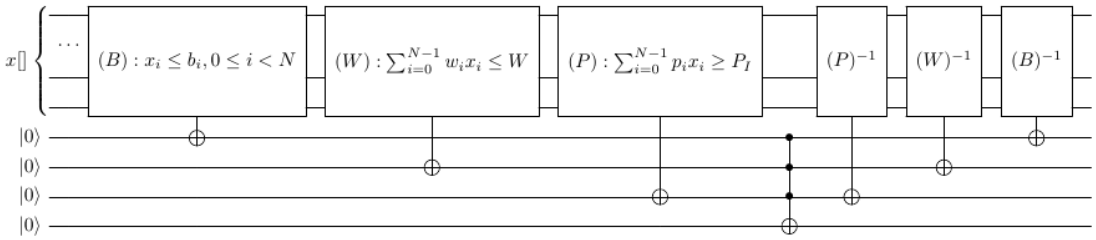

Figure 4.1: Architecture of the Knapsack Problem Oracle, detailing its three modules B, W, P. The bottom qubit represents the answer: the validity of a knapsack input combination.

5 Quantum Oracle Synthesis for Bounds Checking

The goal of the Bounds Checking Module is to ensure each x_i, representing the amount of object type i, is within the allowed range $[0, b_i]$. This is in accordance with the hardware specification shown in Figure 4.1. To accomplish this, comparators constructed with quantum gates are used on each set of qubits that represent values of x_i. For each i, x_i is compared with the corresponding constant b_i; the comparator flips a target ancilla qubit if and only if $x_i \leq b_i$ (representing whether x_i is valid). Thus, a total of n target ancilla qubits are used and must be ultimately reset. After each ancilla qubit is acted

upon and stores whether the amount contained is valid, a multi-control Toffoli gate is performed with all of the ancilla qubits as controls, shown in Figure 5.3. The target of this Toffoli is thus the result of applying the AND gate on all the ancilla qubits. This qubit represents the final answer sought by the Bounds Checking quantum circuit: return 1 (after measurement) if ALL of the x_i are less than or equal to b_i, otherwise return 0. Thus, the target of this Toffoli gate is returned as the output for the entire Bounds Checking.

As described, comparators are necessary to implement Bounds Checking. There are many existing methods to create comparators in quantum circuits, including the quantum bit string comparator (QBSC) of Oliveira and Ramos [15]. Although the QBSC can compare two quantum bit strings and inscribe the result on output qubits, it is not particularly well-suited for use in the Bounds Checking operation. This is because the QBSC presented in [15] performs an operation on two quantum bit strings, while a comparator necessary for Bounds Checking would perform an operation on one quantum bit string, using a constant, non-quantum b_i parameter. We will give the name "constant comparator" to such a comparator that compares a quantum bit string to a constant, non-quantum parameter.

Although it is theoretically possible to inscribe the value b_i into ancilla qubits, use these as the second quantum bit string, and allow direct usage of the QBSC, this would double the total number of ancilla qubits used, which is not ideal and could be easily avoided with other methods. Furthermore, although the Microsoft Q# libraries offer their own comparators for use, these comparators also are not constant comparators and thus were not deemed a good fit for Bounds Checking.

However, the general strategy used in the QBSC can still be applied to design a constant comparator. Only minor modifications are necessary to convert the QBSC to a quantum constant comparator. Specifically, we call the gates of the QBSC according to the bits of b_i, rather than using the qubits of a bit string as controls. Figure 5.1 shows an example circuit design for a less-than-or-equal-to constant comparator that compares qubit string qs with $b = 13$ (or 1101). Let us denote the number of bits in the binary integer comparand by D_q, which has value 4 in the case shown.

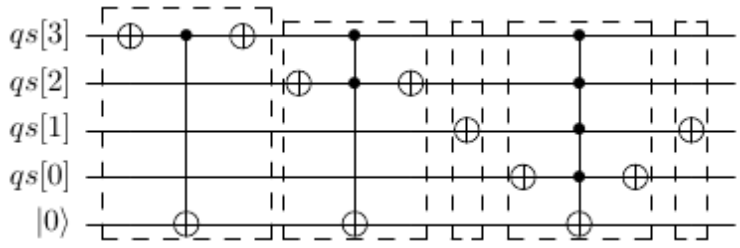

Figure 5.1: Circuit design for an example less-than-or-equal-to constant comparator, in which the input qubits are checked whether the integer that they represent is less than or equal to 13.

In this section, $qs[\,]$ will denote the qubit representation of the number that we want to compare to $b = 13$. For instance, if we want to compare $7\ (0111)$ to 13, then $qs = [|1\rangle, |1\rangle, |1\rangle, |0\rangle]$. The algorithm begins on the qubit for the MSB, $qs[D_q - 1]$, which in the case shown is $qs[3]$, and iterates

down to the LSB. Since the MSB in $13\ (1101)$ is 1, the comparator begins with an inverter on $qs[3]$, a Toffoli gate controlled by the $qs[3]$ and all higher qubits onto the target, and another inverter on $qs[3]$. Likewise, for each successively lower-valued qubit $qs[j]$, if the j^{th} digit in b is 1, the comparator applies an inverter on $qs[j]$, a Toffoli controlled by all qubits from $qs[j]$ to $qs[D_q - 1]$ onto the target, and another inverter on $qs[j]$, in that order. Otherwise, if the j^{th} digit in b is 0, only an inverter on $qs[j]$ is applied. Finally, after all digits have been iterated upon, the previous operations are mirrored to reset the values of the input qubits.

The oracle, as described above, will flip the sign of the bottom qubit if the integer represented by qs is less than or equal to b_i. Since the bottom qubit is initialized as $|0\rangle$, the aforementioned condition results in it becoming $|1\rangle$.

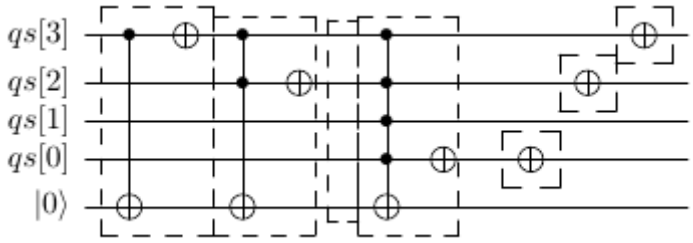

Figure 5.2: Circuit design for a greater-than comparator, in which the input qubits are checked whether the integer that they represent is greater than $\mathbf{2}$.

A similar circuit is designed for the greater-than comparator, which flips the target qubit if the quantum integer has a value greater than the classical integer. In the example comparator in Figure 5.2, this classical comparand is $2\ (0010)$. Beginning on the MSB ($qs[3]$) and iterating down to the LSB ($qs[0]$), if the corresponding binary digit in 2 is 0, then we apply a Toffoli gate onto the target qubit, controlled by that qubit and all higher qubits, then an inverter onto that qubit. Otherwise, if the binary digit is 1, nothing is done. In mirroring, all qubits of the former case have an inverter applied onto them.

The less-than-or-equal-to comparator in Figure 5.1 must be applied onto every set of input qubits that comprise an x_i integer. The result of each comparator is targeted onto an individual ancilla qubit, so the i^{th} qubit stores the state of whether x_i satisfies its bound. As it is desired that all of the bounds are satisfied, a multi-controlled Toffoli gate is used to apply the AND operation to these states and thus flips a final ancilla qubit according to the result of this AND operation. In this way, this final qubit represents whether all the bounds have been satisfied. Finally, all the gates used must be mirrored in the reverse order to reset all input and ancilla qubits. Because the effect of each comparator is only to flip the sign of a target ancilla qubit, the comparator needs to be internally mirrored, as seen in the right half of Figure 5.3.

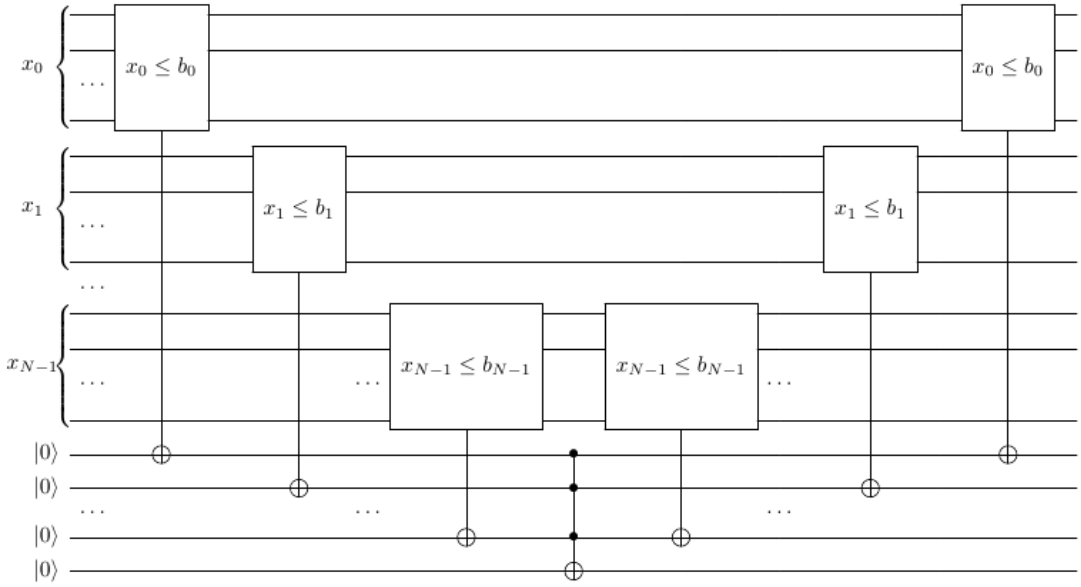

Figure 5.3: Complete Design of Bounds Checking Module using implementation of less-than-or-equal-to comparator that is shown in Figure 5.1.

6 Quantum Circuit Design for In-Place Binary Constant Integer Addition

Eq. (2.2) and (2.3) shown in Section 2 both utilize a summation to calculate the total weights and profits of specific sets of x-values. Thus, it is imperative to create a quantum circuit operation that performs addition on integers represented by qubits.

There exist many papers and implementations of the adder for quantum and reversible circuits [16,17,18,19,20,21,22,23]. In addition, Microsoft Q# Language [24], the programming language used for the implementation of the algorithm presented in this paper, also supports a package that implements the quantum full adder. However, these existing quantum adders are all designed to compute the sum of two arbitrary qubit strings qX and qY, whereas the necessary adder to implement the Knapsack Oracle would have to compute the sum of one constant non-quantum integer X, and one qubit string qY. Although it may be possible to use an existing two-string quantum adder by inscribing integer X into a qubit string, this would require twice as many qubits as necessary. A more practical way to implement a quantum adder for the Knapsack Oracle would be to design a "constant adder" that directly adds integer X onto qY.

The overall strategy and gate-calling used in the existing adders (depicted in Section 2 of [25]) can still be used to create the constant adder. The only modification necessary would be to call the gates based on the bits of X, rather than calling the gates using the qubits of qX as controls. We decided to design such a constant adder and implement it in Q#. The circuit diagram for an example of such a constant adder is depicted in Figure 6.1.

Please note that the primary goal was to design a constant adder that can be written with simplicity in Q#. Optimization of the circuit cost, although important, was a secondary goal. It is known that the optimal design of the adder circuit is obtained using CNOT, CV, and CV$^+$ gates [26], or using CNOT, H, T, T$^+$, and S gates. However, to keep the design simple for implementation and following the convention in quantum algorithm-level designs, we use inverters, CNOT, and Toffoli gates to carry out the logic of addition. This is a standard approach in high-level oracle design. Further optimizing these circuits using either of the two above libraries would require a separate paper. The proposed constant adder design can also be possibly improved by incorporating ideas from the Draper adder [25] that uses QFT, by developing a QFT-based adder that works on one qubit string instead of two.

Let D_Y denote the number of qubits in qY. The constant adder allocates an array qC of ancilla qubits, which will be used to store the carry-out information, similarly to the ancilla qubits in the depiction in [25]. In Figure 6.1, which depicts a constant adder circuit designed for constant $X = 5$, an example with $D_Y = 3$ bits is used.

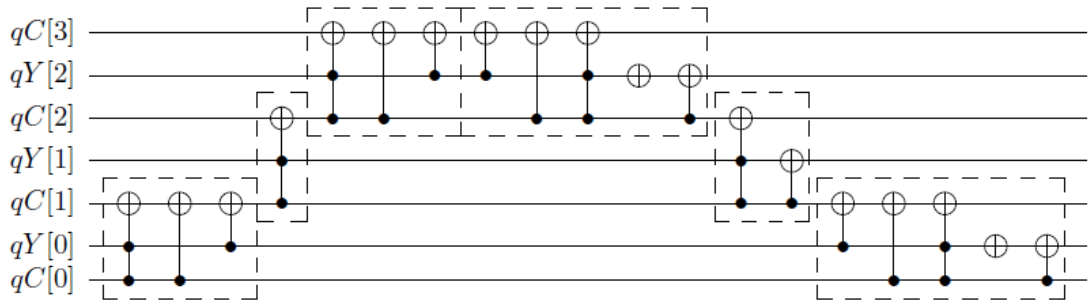

Figure 6.1: Circuit diagram for $\boldsymbol{D_Y}$-bit constant adder, with $\boldsymbol{D_Y = 3}$ and constant $\boldsymbol{X = 5}$ for this example.

The first three enclosed groupings of gates show the "ascending" steps, in which each of the carry-out values are initially calculated. In the "descending" groupings of gates, the goal is to not only reset the calculated carry-out values in the ancilla qubits, but also use these carry-out values to compute the final elements in the qY result value. The qY elements could not have been calculated during the ascension because the new values would have replaced the old, and the old values are necessary for calculating the successive carry-out values. Detailed derivations of the specific arithmetic blocks used can be found in the papers referenced.

Similar to classical computing, the data range of the qubit strings depends on the number of qubits. It may seem inconvenient that one must ensure, before calling the operation, that the array will be sufficiently large to hold the sum. However, this same problem is present in classical computing: for instance, a 32-bit unsigned integer in C++ can only hold integers in the range 0 to $2^{32} - 1$. Likewise, a qubit string of 32 qubits can hold unsigned integers from 0 to $2^{32} - 1$.

In Figure 6.1, where the circuit transitions from the "ascending" steps into the "descending" steps, one may notice that the last three ascending gates and the first three descending gates seem to cancel each other out, thus appearing redundant. Although they may seem gratuitous, their existence is a result of maintaining uniformity of the operations: each iteration of the carry-out operation calculates the next carry-out digit in addition to the current output digit. The last three ascending gates comprise the

calculation of the final carry-out digit, which is information that is not ultimately utilized. Thus, when the first three descending gates are performed, they immediately reverse the actions of the ascending gates without any net effect.

Although this operation may seem superfluous and a squander of time cost, its effect on the time complexity of the operation is negligible. One execution of the constant adder requires a number of operations that is linearly proportional to D_Y. The six gates comprise a constant cost, which, for sufficiently large D_Y, pales in comparison to the linear cost of the adder.

The minimal cost improvement gained from removing the six gates is insufficiently significant to warrant disrupting the loop-uniformity and thus is not implemented.

7 Quantum Circuit Design for Constant Integer Multiplication

Performing the summation for the Weight Checking and Bounds Checking requires a quantum multiplier operation to produce each product that is a summand. Such a multiplier must multiply an arbitrary qubit string Y with a constant non-quantum integer X, and add the product onto an arbitrary qubit string Z. We will refer to such a multiplier as "constant multiplier", to differentiate it from the quantum multipliers that calculate the result of two qubit strings, which are known from literature [16].

There are existing ways to perform quantum multiplication. The method in [16] does so by performing the summation in Eq. (7.1) in modulo 2^D (D denotes the number of qubits used to store Y and Z).

$$Z = XY = \sum_{i=0}^{D-1} 2^i Y[i]X \tag{7.1}$$

However, the method in [16] also features a controlling qubit *c* that acts as a condition of whether the multiplication is performed. If $c = |1\rangle$, the operation adds the product XY onto Z. But if $c = |0\rangle$, then this operation does not add the product XY onto Z, but simply adds Y. This extra feature is not necessary for implementing the desired Weight Checking and Bounds Checking summations, because we only need the $c = |1\rangle$ case, in which the product is added. Thus, the quantum circuit for the multiplier in the Knapsack Oracle will be slightly modified to only include the case in which $c = |1\rangle$.

Essentially, we repeatedly use the addition circuit described in the previous section to construct the quantum circuit constant multiplication operation that multiplies an integer X in classical form with an integer Y in qubit string form and then adds the product into a target qubit string Z.

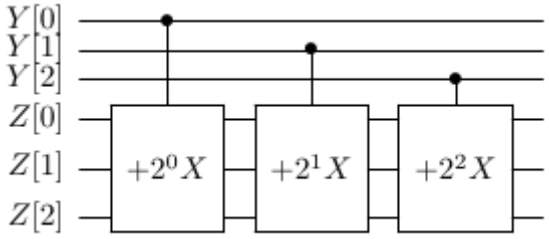

Figure 7.1: Circuit diagram for constant multiplication operation, as a series of multiple additions.

In the circuit shown in Figure 7.1, for the i^{th} digit of Y, $2^i X$ is added to the sum using the previous section's constant adder method, controlled by $Y[i]$. This is in accordance with Eq. (7.1), in which for the i^{th} digit of Y, the amount $2^i X$ is added to the sum if the value of $Y[i]$ is 1, and nothing is done if the value of $Y[i]$ is 0. If *Y* has *D* digits, this multiplication circuit requires at most D calls to the adder circuit.

In this way, the constant multiplier operation multiplies a qubit integer with a non-quantum constant integer and adds the result onto a third qubit integer. If each qubit in this third qubit integer is initially set at $|0\rangle$, the qubit integer will result in the value of that aforementioned product.

8 Application of the Binary Constant Integer Multiplication Quantum Circuit

In Eq. (2.2) and (2.3), multiplication is used between the weights/profits (w_i/p_i) and the object amounts (x_i) to sum the total profit P' and total weight W'. The summation of products will first be done with profits, then with weights.

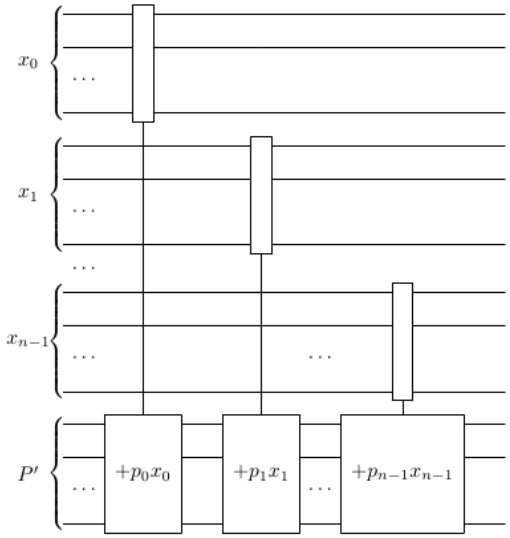

Figure 8.1: The circuit diagram to perform the summation in Eq. (2.3) to calculate *P'*.

Figure 8.1 displays a rather simple process. The addition-of-a-constant-product operation that Section 7 describes is used n times, each time to add the i^{th} product $p_i x_i$ onto P', which denotes the total profit that accrues from this particular combination vector $[x_0, x_1, \ldots, x_{n-1}]$. The blank, narrow rectangle in each gate that covers its corresponding x_i qubits is a new non-standard notation and represents the qubits that are used for the input. These qubits act somewhat as controlling qubits, as they determine the value that will be added to P', but they do not act like usual controlled qubits. Normally, a gate is applied if and only if all controls are $|1\rangle$. In this circuit, the gate will be applied and the magnitude of addition is determined by the integer that the controlling qubits represent. The vertical line

extending from each narrow rectangle represents several lines corresponding to the qubits that the rectangle covers.

If each p_i in the circuit is replaced by w_i and the bottom qubits represent W', then the function can also be used to calculate the total weight of this combination vector. In this way, if the qubits that describe all of $[x_0, x_1, \ldots, x_{n-1}]$ are given as input, the above circuit is able to perform the arithmetic in the summations of Eq. (2.2) and (2.3).

Generalizing the product summer to weights allows calculation of both the total profit P' and the total weight W', as shown in the half-oracle diagram in Figure 8.2, as well as the finalization of the optimization oracle. The half-oracle consists entirely of operations and gates that have been discussed in this paper heretofore.

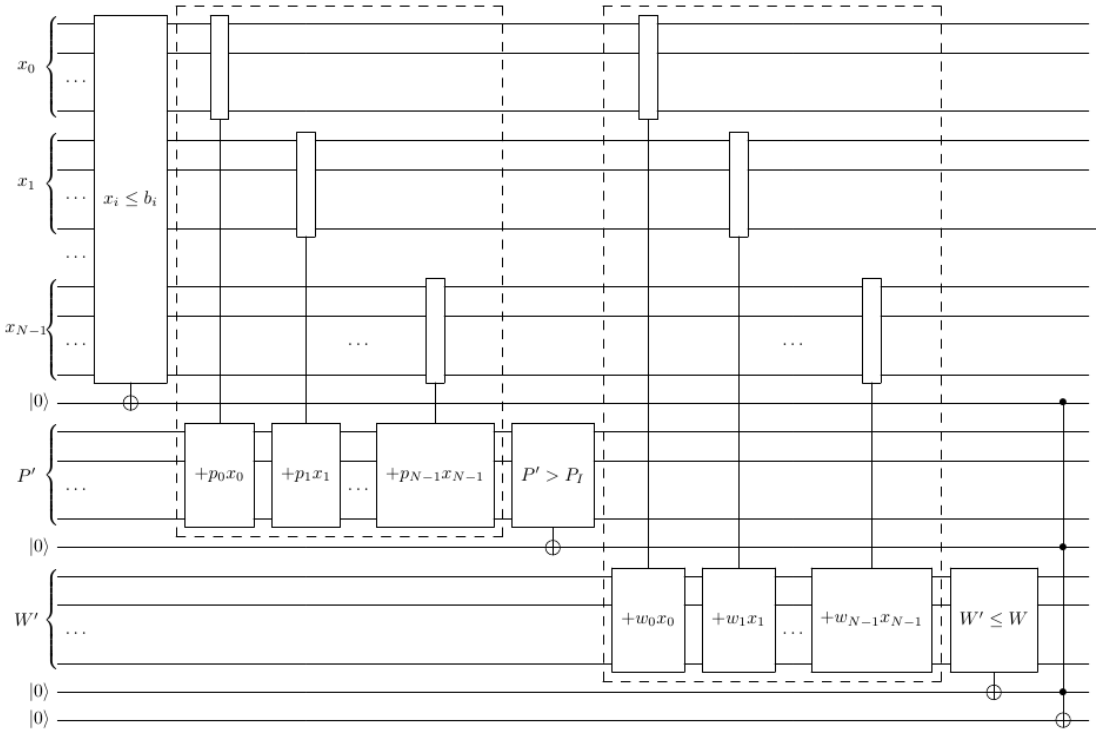

Figure 8.2: Half of the Knapsack Oracle, including its three modules. The other unshown half mirrors the shown half, and simply applies the adjoints in reverse order.

First, the Bounds-Checking oracle of Section 5, as denoted by the simplified label "$x_i \leq b_i$", is called and its output, which specifies whether all of the x_i are within bounds, is recorded in an ancilla qubit. Then the summation is called for P', the same procedure as that of the Fig. 8.1. The calculated value of P' is compared with a given base P_I value: if the new P' exceeds P_I (meaning this combination of x_i is more optimal than the previous), then the corresponding ancilla qubit is flipped and set to $|1\rangle$.

The same summation is applied to W', and W' is compared to the maximum weight W. However, this comparison is different than that of P': W' must **not** exceed W in order for the corresponding ancilla qubit to be flipped to $|1\rangle$.

Each of the first three ancilla qubits represents one of the three Equations from Section 2: Bounds-Checking, Weight Checking, and Profit Maximization. Each ancilla qubit will result in value $|1\rangle$ if the particular set of x_i satisfies its respective equation. Thus, the oracle culminates with a triple-controlled Toffoli gate with controls on these three ancilla qubits. If all three have value $|1\rangle$, meaning all three conditions are satisfied, then the final output qubit is flipped to $|1\rangle$, meaning the set of x_i will have been deemed a more optimal set.

Not shown in the diagram are the mirror gates that reset all input qubits to their original values and non-output ancilla qubits to $|0\rangle$. This is achieved by performing the adjoint of all the gates shown in the diagram (excluding the final Toffoli gate) in reverse order. The adjoint is a feature of Q#, the quantum language that was used to implement this oracle.

9 Complexity Comparison with Classical Dynamic Programming

Thus far, we have in Sections 5 – 8 described the design of the Knapsack Oracle. Because the Knapsack Oracle evaluates a solution of the Knapsack Decision Problem, we can apply the oracle to the maximization method described in Section 3 to solve the corresponding Knapsack Optimization Problem.

We again emphasize that the main focus of the paper is on the design of the quantum oracle, and not on the strategy by which the Grover Search is called in the overall algorithm. However, we also wish to verify that the quantum oracle demonstrates improvement in solving the Knapsack Problem, relative to existing classical algorithms. For other problems that can only be solved with brute force in classical computing, this section would not be necessary because it is obvious that Grover Search has quadratic speed-up relative to brute force. Knapsack Problem, unlike some other optimization problems, can be solved with pre-existing classical algorithms, such as dynamic programming, that are faster than brute force. The application of the Knapsack Oracle in the exponential search is straightforward, but it is necessary to evaluate the oracle's complexity to ensure the proposed oracle truly demonstrates speedup relative to dynamic programming.

The exponential search described in [14] is able to find a particular item in an unbounded list. In the context of the Knapsack Problem, the exponential search can find the solution to the Knapsack Optimization Problem in $O(\log_2 P)$ Grover Searches, in which P is the profit value of the solution. We may establish an upper bound to P that is the maximum achievable profit when combination vectors are not limited by weight constraints. This profit value occurs when, for each i, $x_i = b_i$.

$$P \leq \sum_{i=0}^{n-1} p_i b_i \tag{9.1}$$

We may also establish an upper bound to the complexity of a single Grover Search. Recall from Section 3 that Grover Search runs in $\frac{\pi}{4}\sqrt{\frac{N}{M}}$ oracle calls and that $N = \prod_{i=0}^{n-1}(b_i + 1)$ in the Bounded

Knapsack Problem. Because $M \geq 1$, the number of oracle calls that Grover Search uses is at most $\frac{\pi}{4}\sqrt{N}$, which is equivalent to the expression in Eq. (9.2).

$$\frac{\pi}{4}\sqrt{\prod_{i=0}^{n-1}(b_i+1)} \tag{9.2}$$

The complexity of the oracle itself may also be determined. We will analyze the complexity of each module in the oracle.

(i) (B) Bounds Checking contains $2n$ calls to the less-than-or-equal-to comparator, the i^{th} of which runs in $\lceil \log_2(b_i+1) \rceil \approx \log_2(b_i+1)$ complexity.
   (a) Thus, the complexity of (B) is $\sum_{i=0}^{n-1} \log_2(b_i+1)$.

(ii) The (W) Weight Checking and (P) Profit Checking modules consist of a summation and a comparator each.
   (a) A summation consists of n multiplications, the i^{th} of which runs in $\log_2(b_i+1)$ complexity. Thus, the complexity of the summation of products is the sum of each multiplication's complexity: $\sum_{i=0}^{n-1} \log_2(b_i+1)$.
   (b) The comparator has a negligible complexity relative to the summation.

The non-negligible complexities of (B) and the summations are on the same order, so the complexity of the oracle is concluded to be:

$$\sum_{i=0}^{n-1} \log_2(b_i+1) \tag{9.3}$$

Therefore, the total time complexity of the algorithm, using exponential search, is the product of the (upper bound of the) number of Grover Searches, the (upper bound of the) number of oracle calls per Grover Search, and the complexity of the oracle in Eq. (9.3).

$$\log_2\left(\sum_{i=0}^{n-1} p_i b_i\right) \times \sqrt{\prod_{i=0}^{n-1}(b_i+1)} \times \sum_{i=0}^{n-1} \log_2(b_i+1) \tag{9.4}$$

We desire to analyze how Eq. (9.4) grows with n in order to compare the algorithm's complexity with the complexity of classical methods, particularly dynamic programming. To simplify this comparison, we approximate Eq. (9.4) by considering the case in which bounds b_i and profits p_i are equal among the n objects. Let us set the common bound to b and the common profit to p. A simplified complexity is now expressed in terms of n in Eq. (9.5).

$$\log_2(npb) \times \sqrt{(b+1)^n} \times n \log_2(b+1) \tag{9.5}$$

The classical dynamic programming algorithm for the Knapsack Problem has a computational complexity that is shown in Eq. (9.6) [12].

$$O(nW) \tag{9.6}$$

in which n is the number of object types, and W is the maximum weight.

Several things are noticeable. First, the solution to the Knapsack Problem using the proposed quantum algorithm has quadratic speedup relative to classical brute-force search (linear search), as expected.

Second, the quantum algorithm's complexity is dependent on b_i, the bound values, while the classical dynamic programming algorithm is not. Thus, the quantum algorithm does not perform as well as the dynamic programming algorithm when the bounds are very high.

On the other hand, the quantum algorithm's complexity is constant with W, while the dynamic programming algorithm's complexity grows linearly with W. This difference implies that if W is extremely large or requires many digits for precision, the dynamic programming algorithm's complexity will grow just as quickly, while the quantum algorithm will not grow in complexity at all. This comprises a major improvement in the quantum algorithm compared to the dynamic programming algorithm, as it is very frequent in the real-world that data used for quantities such as W would be extremely large or would require high precision.

The quantum algorithm, therefore, is an improvement from the classical dynamic programming algorithm in cases when W is relatively large and the b_i values are relatively smaller.

10 Verification of Oracle Correctness

The Knapsack Oracle, as described, was written and implemented in the quantum language Q#, using Microsoft Quantum Development Kit (version 0.8.1907.1701). The QDK allows to create code in Visual Studio 2017 for quantum circuits and verify results using a local simulator [27].

It is necessary to ensure that the oracle implementation functions properly and returns correct outputs for various parameter sets. A parameter set is a set of information that details a possible combination of parameter values for a particular Knapsack Problem. Specifically, for the oracle, a parameter set consists of integers n, W, P, as well as the coefficient sets b_i, w_i, p_i for $0 \leq i < n$. Therefore, numerous parameter sets were created for use as test cases, using these variables. These are listed in the Table 10.1.

Table 10.1. Test data used to verify correctness of Knapsack Oracle.

n	W	P	$[b_i]$	$[w_i]$	$[p_i]$	Max Qubits
2	23	10	[7,5]	[2,5]	[1,3]	20
3	30	40	[6,5,2]	[2,3,10]	[2,3,15]	22
3	34	34	[4,7,2]	[6,3,1]	[5,2,1]	22
4	14	24	[4,3,2,3]	[1,2,3,1]	[2,4,9,2]	23
5	84	60	[8,3,3,2,6]	[7,7,2,2,3]	[3,2,9,6,5]	29

As corroborated by Table 10.1, the range of parameter sets that can be simulated is rather limited. This is due to the fact that qubits take exponential space: using an additional qubit will require double the memory. The current version of QDK, specifically, is able to simulate quantum circuits with up to 30 local qubits [28]. Each parameter set depicted in Table 10.1 had values chosen large enough to be able to accurately verify oracle correctness, but also small enough so that at no point during the verification program would the instantaneous number of qubits used exceed 30.

For each of the data sets shown in Figure 10.1, the verification program iterated through all x_i values that were permitted by the bounds set by b_i. The program, on each combination of x_i values, called the K oracle and checked that the W' and P' values calculated by the oracle were correct and that the oracle returned the correct final output. Since the oracle returned no such errors when run using Q#, it was deemed to have been designed correctly.

11 Analysis of Oracle Gate Usage

It is important to discuss the number of gates that are used in the Knapsack Oracle, because to implement the oracle in practice requires knowledge of its cost to estimate the resources necessary for construction. Previously in Section 9, we found the number of gates in the oracle to be on the order of $\sum_{i=0}^{n-1} \log_2 b_i$, and used this to verify that Grover Search using the Knapsack Oracle retains a quadratic speed-up in relation to classical brute force.

Although the order-of-magnitude estimation is sufficient to prove quadratic speed-up, a more precise count of the oracle gates is necessary to evaluate its cost. There are several ways to assign costs to various types of gates and enumerate the cost of an oracle. One method to evaluate quantum costs is described in [29], which contains a table detailing the costs of inverters, CNOT, and multi-controlled Toffoli gates. We will use this standard to evaluate the total gate cost of the Knapsack Oracle.

The comparators and constant adder circuits contain a majority of the gates used in the oracle. The less-than-or-equal-to comparator, in our implementation, requires $2D$ inverters, in which D denotes the number of qubits used to represent the quantum argument qY. The comparator also uses several Toffoli gates, the type and number of which depend on the binary digits of the integer comparand b. Specific counts of each type of gate for the less-than-or-equal-to comparator are shown in Table 11.1. The greater-than comparator also uses various types and numbers of gates, which are also dependent on the digits of the integer comparand b. Gate counts for the greater-than comparator are listed in Table 11.2.

The constant adder circuit used in the Knapsack Oracle uses only inverters, CNOT gates, and CCNOT gates. The number of each type of gate depends on the specific constant integer addend X as well as the number D of qubits used to represent the target quantum integer. Table 11.3 lists the counts of the three types of gates for sample constant adder circuits with varying values of X and D.

Using the gate cost analysis from [29], we evaluated the costs of the example test cases from Table 10.1. The calculated costs are listed in Table 11.4. However, it is also helpful to have a general formula for the cost. It is difficult to create a formula for the cost of a generalized Knapsack Oracle, but it is possible to determine a cost formula for cases in which the b_i values are all equal. This allows us to estimate the cost formula of the Knapsack Oracle. In terms of $n, b,$ and D, the minimum and maximum total costs of the Knapsack Oracle are given by Eq. (11.1) and Eq. (11.2).

$$\text{Minimum Cost: } (16\lceil \log_2 b \rceil^2 + (64D - 44)\lceil \log_2 b \rceil + 32)n \quad (11.1)$$

$$\text{Maximum Cost: } \left(2^{\lceil \log_2 b \rceil + 3} + (64D - 2)\lceil \log_2 b \rceil - 8\right)n \quad (11.2)$$

Table 11.1. Number of each type of gate used in sample less-than-or-equal-to comparator circuits. Columns for Toffoli gates with more than two controls are labeled using the number of controls m.

D	b	Counts of m-controlled Toffoli gate for Less-than-or-equal-to Circuit.							
		0 (Inverter)	1 (CNOT)	2 (CCNOT)	3	4	5	6	7
3	7	6	1	1	2	0	0	0	0
4	13	8	1	1	0	2	0	0	0
5	26	10	1	1	0	1	1	0	0
7	0	14	0	0	0	0	0	0	1
8	100	16	0	1	1	0	0	1	1

Table 11.2. Number of each type of gate used in sample greater-than comparator circuits. Columns for Toffoli gates with more than two controls are labeled using the number of controls.

D	b	Counts of m-controlled Toffoli gate for Greater-than Circuit.							
		0 (Inverter)	1 (CNOT)	2 (CCNOT)	3	4	5	6	7
3	7	0	0	0	0	0	0	0	0
4	13	2	0	0	1	0	0	0	0
5	26	4	0	0	1	0	1	0	0
7	0	14	1	1	1	1	1	1	1
8	100	8	1	0	0	1	1	0	1

Table 11.3. Number of each type of gate used in sample constant adder circuits.

D	X	0 (Inverter)	1 (CNOT)	2 (CCNOT)
5	4	1	9	0
5	10	2	13	10
8	38	3	20	16
7	50	3	19	14
9	17	2	17	18

Table 11.4. Extension of Table 10.1, with the calculated total minimum and maximum costs of various sample Knapsack Oracles. Minimum and maximum gate costs are referenced from [29].

n	W	P	$[b_i]$	$[w_i]$	$[p_i]$	Min. Cost	Max. Cost
2	23	10	[7,5]	[2,5]	[1,3]	2471	3011
3	30	40	[6,5,2]	[2,3,10]	[2,3,15]	3467	3763
3	34	34	[4,7,2]	[6,3,1]	[5,2,1]	3283	3579
4	14	24	[4,3,2,3]	[1,2,3,1]	[2,4,9,2]	4423	4719
5	84	60	[8,3,3,2,6]	[7,7,2,2,3]	[3,2,9,6,5]	7361	7631

12 Conclusions

The quantum circuit design and analysis presented in this paper have several important highlights and implications.

- The Knapsack Oracle is the first such proposed oracle for the Knapsack Optimization Problem.
- Knapsack Oracle was successfully designed to verify a particular solution to a Knapsack Decision Problem. Using any of the well-known methods (such as exponential search) to convert an optimization problem to a series of decision problems, the oracle can be used with Grover Search to solve the Knapsack Optimization Problem.
- Because the oracle runs in polynomial time, solving the Knapsack Problem using Grover Search retains a quadratic speed-up over the classical brute force algorithm.
- Using the oracle to solve the Knapsack Problem also has a lower time complexity than the dynamic programming algorithm for relatively small b_i and relatively large W.
- We designed detailed quantum circuits to implement the Knapsack Oracle in the Q# language and verified its correctness on its simulator.
- This paper analyzes the complexity and quantum gate cost of the oracle, which has not been done in literature.

Future work can include extension of our work to ternary quantum circuits or other types of multi-valued quantum circuits such as using general qudits. The Knapsack-related ideas from [30] can also be realized with circuits similar to those presented in this paper.

References

1. K. Shimizu, R. Mori (2019), *Exponential-time quantum algorithms for graph coloring problems*, School of Computing, Tokyo Institute of Technology, Japan, quant-ph/ 1907.00529.
2. P. Shor (1996), *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*, SIAM J. Comput., 26 (5): 1484–1509.
3. M.A. Nielsen, I.L. Chuang (2000), *Quantum Computation and Quantum Information*, Cambridge University Press.
4. E. Rieffel, W. Polak (2011), *Quantum Computing: A Gentle Introduction*, The MIT Press.
5. Y. Li, E. Tsai, M. Perkowski, X. Song (2019), *Grover-Based Ashenhurst-Curtis Decomposition Using Quantum Language Quipper*, Quantum Information and Computation, 19(1&2), 0035-0066.
6. A. Childs (2017), *Lecture Notes on Quantum Algorithms*. Dept. Computer Science, University of Maryland, College Park, Maryland.

7. A. Ambainis, K. Balodis, J. Iraids, M. Kokainis, K. Prusis, J. Vihrovs (2018), *Quantum Speedups for Exponential-Time Dynamic Programming Algorithms*, Centre for Quantum Computer Science, Faculty of Computing, University of Latvia, Riga, Latvia.
8. S.S. Skiena (1999), "Who is Interested in Algorithms and Why? Lessons from the Stony Brook Algorithm Repository", ACM SIGACT News. 30(3), 65–74.
9. N. Ferdosian, M. Othman, B.M. Ali, K.Y. Lun (2015), *Greedy-Knapsack Algorithm for Optimal Downlink Resource Allocation in LTE Networks*, Wireless Networks.
10. T. Nasako, Y. Murakami (2006), *A New Trapdoor in Modular Knapsack Public-Key Cryptosystem*, Proceedings 2006 29th Symposium on Information Theory and Its Applications, Hakodate, Hokkaido, Japan.
11. K. Lai (2006), *The Knapsack Problem and Fully Polynomial Time Approximation Schemes (FPTAS)*, Seminar in Theoretical Computer Science, MIT (Cambridge, MA).
12. R. Andonov, V. Poirriez, S. Rajopadhye (2000), *Unbounded Knapsack Problem: dynamic programming revisited*, European Journal of Operational Research.
13. G. Brassard, P. Høyer, A. Tapp (1998), *Quantum Counting*, 25th Intl. Colloquium on Automata, Languages, and Programming (ICALP), LNCS 1443, pp. 820-831; arXiv:quant-ph/9805082.
14. J.L. Bentley, A.C. Yao (1975), *An almost optimal algorithm for unbounded searching*, Information Processing Letters, 5(3), 82-87.
15. D. Oliveira, R. Ramos (2007), *Quantum bit string comparator: Circuits and applications*, Departamento de Engenharia de Teleinformatica, Universidad Federal do Ceara, Fortaleza, Brazil.
16. V. Vedral, A. Barenco, A. Ekert (1995), *Quantum Networks for Elementary Arithmetic Operations*, Phys. Rev. A 54(1); arXiv:quant-ph/9511018v1.
17. J. W. Bruce, M. A. Thornton, L. Shivakumaraiah, P. S. Kokate and X. Li (2002), *Efficient adder circuits based on a conservative reversible logic gate*, Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002, Pittsburgh, PA, USA, pp. 83-88.
18. T. Häner, M. Roetteler, K.M. Svore (2018), *Optimizing Quantum Circuits for Arithmeti*c; quant-ph: 1805.12445v1.
19. M. Haghparast, K. Navi (2007), *A Novel Reversible Full Adder Circuit for Nanotechnology Based Systems*, J. Applied Sci., 7(24), 3995-4000.
20. H.G. Rangaruju, U. Venugopal, K.N. Muralidhara, K.B. Raja (2010), *Low Power Reversible Parallel Binary Adder/Subtractor*, International Journal of VLSI Design & Communication Systems, 1.3(2010), 23-34; arXiv:1009.6218v1.
21. H. Thapliyal, N. Ranganathan (2011), *A new reversible design of BCD adder*, 2011 Design, Automation & Test in Europe, Grenoble, France.
22. S. Islam, R. Islam (2005), *Minimization of reversible adder circuits*, Asian J. Inform. Tech, 4(12), 1146-1151.
23. H. Thapliyal, M.B. Srinivas (2005), *A Novel Reversible TSG Gate and Its Application for Designing Reversible Carry Look-Ahead and Other Adder Architectures*, In: Srikanthan T., Xue J., Chang CH. (eds) Advances in Computer Systems Architecture. ACSAC 2005. Lecture Notes in Computer Science, vol 3740. Springer, Berlin, Heidelberg.
24. Microsoft Quantum Documentation and Q# API Reference (2019), URL: https://docs.microsoft.com/en-us/quantum/?view=qsharp-preview
25. T. Draper (2000), *Addition on a Quantum Computer*; arXiv:quant-ph/0008033.
26. B. Ali, T. Hirayama, K. Yamanaka, Y. Nishitani (2017), *Optimal MCT Circuits of Reversible Adder-Subtractors and Quantum Gate Realization*, Graduate School of Electrical Engineering and Computer Science, Iwate University, Morioka, Japan.
27. Quantum Development Kit (2019), URL: https://www.microsoft.com/en-us/quantum/development-kit.

28. R. LaRose (2019), *Overview and Comparison of Gate Level Quantum Software Platforms*, Michigan State University.
29. D. Maslov, G. Dueck (2004), *Improved Quantum Cost for n-bit Toffoli Gates. Faculty of Computer Science*, University of New Brunswick, Fredericton, Canada.
30. A. Montanaro (2019), *Quantum speedup of branch-and-bound algorithms*, Phys. Rev. Research 2, 013056(2020); arXiv:1906.10375.