**Chapter 3**

**Quantum Search Algorithm and its Applications**

Because in **IBM**’s quantum computers, they only provide quantum gates of single quantum bit and two quantum bits, quantum gates of three quantum bits and many quantum bits must manually be decomposed into quantum gates of single quantum bit and two quantum bits. A good quantum algorithm of solving any given problem with the size of the input of *n* bits must have a constant successful probability of measuring its answer(s) that is close to one as soon as possible. In this chapter, we first illustrate how to decompose quantum gates of three quantum bits and many quantum bits into quantum gates of single quantum bit and two quantum bits. Next, we introduce how to write quantum programs with version 2.0 of Open QASM to implement decomposition among various kinds of quantum gates. A *quantum search algorithm* that is sometimes known as *Grover*’*s algorithm* to find an item in unsorted databases with 2*n* items that satisfies any given condition can give a quadratic speed-up and is the best one known. Hence, we then describe how to write quantum programs with version 2.0 of Open QASM to implement the quantum search algorithm in order to solve various applications.

**3.1 Introduction to the Search Problem**

It is assumed that a set *X* is equal to {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*}. From the set *X* the minimum element is *x*10 *x*20 … *xn* − 10 *xn*0 with *n* bits and the maximum element is *x*11 *x*21 … *xn* − 11 *xn*1 with *n* bits. For convenience of presentation, in the set *X* the decimal value of the minimum element with *n* bits is 0 and the decimal value of the maximum element with *n* bits is 2*n* − 1. We regard the set *X* as an unsorted database containing 2*n* items (elements) with each item has *n* bits.

A search problem is to that from the set *X* that is {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*} and is also an unsorted database with 2*n* items (elements) *M* items (elements) satisfy any given condition and we would like to find one of *M* solutions, where 1 ≤ *M* ≤ 2*n*. A common formulation of the search problem is as follows. For any given oracular function *Of*: {*x*1 *x*2 … *xn* − 1 *xn* | ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*} → {0, 1}, there are *M* inputs of *n* bits from its domain, say *λM*, that satisfies the condition *Of*(*λM*) = 1, whereas for all other inputs of *n* bits from the same domain, *ω*, for 0 ≤ *ω* ≤ 2*n* − 1 and *ω* ≠ *λM*, *Of* (*ω*) = 0. The search problem is to find one of *M* solutions.

The most efficient classical algorithm for the search problem is to check whether the items (elements) in the domain one by one satisfy *Of*(*λM*) = 1 or not. If an item (element) satisfies the required condition that is *Of*(*λM*) = 1, then the most efficient classical algorithm is terminated. Otherwise, it continues to examine whether next item (element) satisfies *Of*(*λM*) = 1 or not until the answer is found. The number of solutions that is one is the worst case in the search problem. For the worst case in the search problem, the best time complexity of finding the desired answer (item) is O(1), the average time complexity of finding the desired answer (item) is O() and the worst time complexity of finding the desired answer (item) is O(2*n*).

**3.2 Introduction to the Satisfiability Problem**

Let us consider an example *F*(*x*1, *x*2) = (*x*2 ∨ *x*1) ∧ ( ∨ ) ∧ (*x*1). The two variables *x*2 and *x*1 are two Boolean variables and their values could be 0 or 1. We suppose that 0 is “false” and 1 is “true”. A symbol “∨” is the “logical or” operation and a symbol “∧” is the “logical and” operation. Therefore, a Boolean formula *x*2 ∨ *x*1 is 0 only if both *x*2 and *x*1 are 0; a Boolean formula *x*2 ∧ *x*1 is 1 only if both *x*2 and *x*1 are 1. We regard the Boolean formula *x*2 ∨ *x*1 as one clause and we regard the Boolean formula *x*2 ∧ *x*1 as another clause.

We give to represent the “negation” of *x*1 and we give to represent the “negation” of *x*2. A Boolean formula is 1 if *x*1 is 0 and is 0 if *x*1 is 1. A Boolean formula is 1 if *x*2 is 0 and is 0 if *x*2 is 1. Of course, we also regard the Boolean formula and the Boolean formula as two different clauses. The satisfiability problem that is a **NP-complete** problemis to find Boolean values of *x*2 and *x*1 to make the formula *F*(*x*1, *x*2) to be true that is equal to 1.

In this example, the answer is *x*2 = 0 and *x*1 = 1. In the formula *F*(*x*1, *x*2), it actually includes three *clauses*: the first clause is “(*x*2 ∨ *x*1)”, the second clause is “( ∨ )” and the third clause is “(*x*1)”. A clause is a formula of the form *x*1 ∨ *x*2 ∨ … *xn* − 1 ∨ *xn*, where each variable *xk* for 1 ≤ *k* ≤ *n* is a Boolean variable or its negation. Because the quantum program of implementing this example uses more quantum bits that exceed five quantum bits in the backend *ibmqx4* with five quantum bits in **IBM**’s quantum computers, we just use this example to explain what the satisfiability problem is. Next, we give **Definition 3-1** to introduce the satisfiability problem.

**Definition 3-1**: In general, a satisfiability problem contains a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*, where each clause *Cj* for 1 ≤ *j* ≤ *m* is a formula of the form *x*1 ∨ *x*2 ∨ … *xn* − 1 ∨ *xn* for each Boolean variable *xk* to 1 ≤ *k* ≤ *n*. Next, the question is to find values of each Boolean variable so that the whole formula has the value 1. This is the same as finding values of each Boolean variable that make each clause have the value 1.

From **Definition 3-1**, for a satisfiability problem with *n* Boolean variables and *m* clauses, we regard *m* clauses as any given oracular function *Of*(*x*1, *x*2, …, *xn* − 1,*xn*) and regard 2*n* inputs of *n* Boolean variables as its domain {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*}. The satisfiability problem with *n* Boolean variables and *m* clauses is to find inputs of *n* bits (*n* Boolean variables) from its domain so that the whole formula *Of*(*x*1, *x*2, …, *xn* − 1,*xn*) has the value 1. This is to say that a satisfiability problem with *n* Boolean variables and *m* clauses is actually a kind of search problems.

**3.2.1 Flowchart of Solving the Satisfiability Problem**

From **Definition 3-1**, a satisfiability problem contains a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*, where each clause *Cj* for 1 ≤ *j* ≤ *m* is a formula of the form *x*1 ∨ *x*2 ∨ … *xn* − 1 ∨ *xn* for each Boolean variable *xk* to 1 ≤ *k* ≤ *n*. For solving the satisfiability problem with *n* Boolean variables and *m* clause, we need to use auxiliary Boolean variables *rj, k* for 1 ≤ *j* ≤ *m* and 0 ≤ *k* ≤ *n* and auxiliary Boolean variables *sj* for 0 ≤ *j* ≤ *m*. Because we use auxiliary Boolean variables *rj,* 0 for 1 ≤ *j* ≤ *m* as the first operand of the first logical or operation (“∨”) in each clause, the initial value of each auxiliary Boolean variable *rj,* 0 for 1 ≤ *j* ≤ *m* is set to zero (0). This is to say that this setting does not change the correct result of the first logical or operation in each clause. We use ***CCNOT*** gates and ***NOT*** gates to implement the logical or operations in each clause and we apply auxiliary Boolean variables *rj, k* for 1 ≤ *j* ≤ *m* and 1 ≤ *k* ≤ *n* to store the result of implementing the logical or operations in each clause. This indicates that each auxiliary Boolean variable *rj, k* for 1 ≤ *j* ≤ *m* and 1 ≤ *k* ≤ *n* is actually the target bit of a ***CCNOT*** gate of implementing a logical or operation. Therefore, the initial value of each auxiliary Boolean variable *rj, k* for 1 ≤ *j* ≤ *m* and 1 ≤ *k* ≤ *n* is set to one (1).

We use an auxiliary Boolean variable *s*0 as the first operand of the first logical and operation (“∧”) in a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*. The initial value of the auxiliary Boolean variable *s*0 is set to one (1). This implies that this setting does not change the correct result of the first logical and operation in *C*1 ∧ *C*2 … ∧ *Cm*. We use ***CCNOT*** gates to implement the logical and operations in *C*1 ∧ *C*2 … ∧ *Cm* and we apply auxiliary Boolean variables *sj* for 1 ≤ *j* ≤ *m* to store the result of implementing the logical and operations in *C*1 ∧ *C*2 … ∧ *Cm*. This is to say that each auxiliary Boolean variable *sj* for 1 ≤ *j* ≤ *m* is actually the target bit of a ***CCNOT*** gate of implementing a logical and operation. Thus, the initial value of each auxiliary Boolean variable *sj* for 1 ≤ *j* ≤ *m* is set to zero (0). For the convenience of our presentation, we assume that |*Cj*| is the number of Boolean variable in the *j*th clause *Cj*.

Figure 3.1 is to flowchart of solving the satisfiability problem with *n* Boolean variables and *m* clauses. In Figure 3.1, in statement *S*1, it sets the index variable *j* of the first loop to one (1). Next, in statement *S*2, it executes the conditional judgement of the first loop. If the value of *j* is less than or equal to the value of *m*, then *next executed* instruction is statement *S*3. Otherwise, in statement *S*9, it executes an *End* instruction to terminate the task that is to find values of each Boolean variable so that the whole formula has the value 1 and this is the same as finding values of each Boolean variable that make each clause have the value 1.

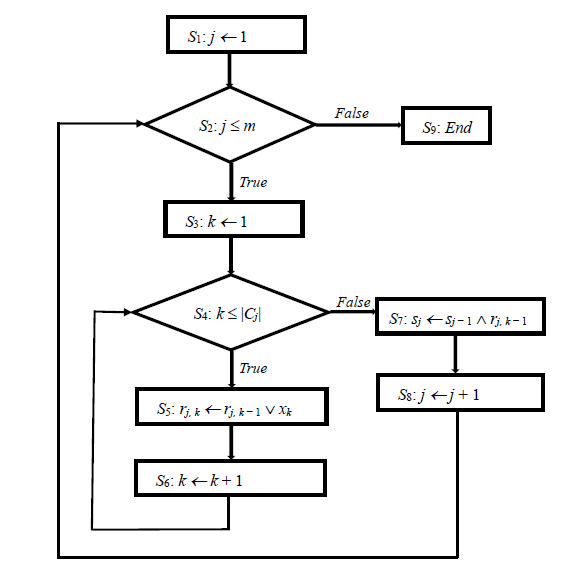


Figure 3.1: Flowchart of solving the satisfiability problem with *n* Boolean variables and *m* clauses.

In statement *S*3, it sets the index variable *k* of the second loop to one (1). Next, in statement *S*4, it executes the conditional judgement of the *second* loop. If the value of *k* is less than or equal to the number of Boolean variables in the *j*th clause *Cj*, then next executed instruction is statement *S*5. Otherwise, next executed instruction is statement *S*7. In statement *S*5, it implements a logical or operation “*rj, k* ← *rj, k* − 1 ∨ *xk*” that is the *k*th logical or operation in the *j*th clause *Cj*. Boolean variable *rj, k* − 1 is the first operand of the logical or operation and stores the result of the previous logical or operation. Boolean variable *xk* is the second operand of the logical or operation. Boolean variable *rj, k* stores the result of implementing the *k*th logical or operation in the *j*th clause *Cj*. Next, in statement *S*6, it increases the value of the index variable *k* to the second loop. Repeat to execute statement *S*4 through statement *S*6 until in statement *S*4 the conditional judgement becomes a *false* value.

When the value of the index variable *k* in the second loop is greater than the number of Boolean variables in the *j*th clause *Cj*, next executed instruction is statement *S*7. In statement *S*7, it executes a logical and operation “*sj* ← *sj* − 1 ∧ *rj, k* − 1” in *C*1 ∧ *C*2 … ∧ *Cm*. Boolean variable *sj* − 1 is the first operand of the logical and operation and stores the result of the previous logical and operation. Because the value of *k* is equal to |*Cj*| + 1, the value of (*k* − 1) is equal to |*Cj*|. Boolean variable *rj, k* − 1 is the second operand of the logical and operation and stores the result of implementing a formula of the form *x*1 ∨ *x*2 ∨ … *xn* − 1 ∨ *xn* in the *j*th clause *Cj*. Boolean variable *sj* stores the result of implementing the *j*th logical and operation in *C*1 ∧ *C*2 … ∧ *Cm*. Next, in statement *S*8, it increases the value of the index variable *j* to the first loop. Repeat to execute statement *S*2 through statement *S*8 until in statement *S*2 the conditional judgement becomes a *false* value. Because from **Definition 3-1** each clause *Cj* for 1 ≤ *j* ≤ *m* is a formula of the form *x*1 ∨ *x*2 ∨ … *xn* − 1 ∨ *xn*, from Figure 3.1 the total number of logical and operation and logical or operation is *m* logical and operations and () = (*m* × *n*) logical or operations. This is the cost of implementing *m* clauses of one time for one of 2*n* inputs to *n* Boolean variables. Therefore, the cost of implementing *m* clauses of 2*n* times for 2*n* inputs of *n* Boolean variables is (2*n* × *m*) logical and operations and (2*n* × *m* × *n*) logical or operations.

**3.2.2 Data Dependence Analysis for the Satisfiability Problem**

A data dependence arises from two statements that access or modify the same resource. *Data dependence analysis* is to judge whether it is safe to *reorder* or *parallelize* statements. In a satisfiability problem with *n* Boolean variables and *m* clauses, it consists of 2*n* inputs that are 2*n* combinational states of *n* Boolean variables. The first input is *x*10 *x*20 … *xn* − 10*xn*0, the second input is *x*10 *x*20 … *xn* − 10*xn*1 and so on with that the last input is *x*11 *x*21 … *xn* − 11*xn*1. Each input needs to complete those operations in Figure 3.1. Each input needs to use (*m* × (*n* + 1)) auxiliary Boolean variables *rj, k* for 1 ≤ *j* ≤ *m* and 0 ≤ *k* ≤ *n* and (*m* + 1) auxiliary Boolean variables *sj* for 0 ≤ *j* ≤ *m*. Since 2*n* inputs of *n* Boolean variables implement those operations in Figure 3.1 not to access or modify the same input and the same auxiliary Boolean variables, we can *parallelize* them without any error.

Let us consider another example that is *F*(*x*1, *x*2) = *x*1 ∧ *x*2, where two variables *x*1 and *x*2 are two Boolean variables and their values could be 0 or 1. In the formula *F*(*x*1, *x*2) = *x*1 ∧ *x*2, the first clause contains (*x*1) and the second clause includes (*x*2). The satisfiability problem for the Boolean formula *F*(*x*1, *x*2) = *x*1 ∧ *x*2 with two Boolean variable *x*1 and *x*2 is to find values of each Boolean variable so that the whole formula has the value 1. This is the same as finding values of each Boolean variable that make each clause have the value 1.

We regard the satisfiability problem for the Boolean formula *F*(*x*1, *x*2) = *x*1 ∧ *x*2 with two Boolean variable *x*1 and *x*2 as a search problem in which any given oracular function *Of* is the Boolean formula *F*(*x*1, *x*2) = *x*1 ∧ *x*2, its domain is {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2} and its range is {0, 1}. In the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 of the search problem, there are *M* inputs of *two* bits from its domain, say *λM* = *x*1 *x*2, that satisfies the condition *Of*(*λM*) = *Of*(*x*1 *x*2) = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 = 1. Whereas for all other inputs of two bits from the same domain, *ω* = *x*1 *x*2, for 0 ≤ *ω* ≤ 22 − 1 and *ω* ≠ *λM*, *Of* (*ω*) = *Of* (*x*1 *x*2) = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 = 0. The search problem is to find one of *M* solutions that is to find values of each Boolean variable so that the whole formula has the value 1. This is the same as finding values of each Boolean variable that make each clause have the value 1.

From the domain {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2} of the Boolean formula *F*(*x*1, *x*2) = *x*1 ∧ *x*2, there are four inputs *x*10 *x*20, *x*10 *x*21, *x*11 *x*20 and *x*11 *x*21. Because it contains two clauses in which each clause only consists of one Boolean variable, each input needs to complete “*r*1, 1 ← *r*1*,* 00 ∨ *x*1”, “*s*1 ← *s*01 ∧ *r*1, 1”, “*r*2, 1 ← *r*2*,* 00 ∨ *x*2” and “*s*2 ← *s*1 ∧ *r*2, 1”. The result of implementing a logical or operation “*r*1, 1 ← *r*1*,* 00 ∨ *x*1” is actually equal to the value of Boolean variable *x*1. This is to say that Boolean variable *r*1, 1 stores the value of Boolean variable *x*1. Next, a logical and operation “*s*1 ← *s*01 ∧ *r*1, 1” is equivalent to another logical and operation “*s*1 ← *s*01 ∧ *x*1”. Because the result of implementing “*s*1 ← *s*01 ∧ *x*1” is actually equal to the value of Boolean variable *x*1, Boolean variable *s*1 stores the value of Boolean variable *x*1.

Next, the result of implementing a logical or operation “*r*2, 1 ← *r*2*,* 00 ∨ *x*2” is actually equal to the value of Boolean variable *x*2. This indicates that Boolean variable *r*2, 1 stores the value of Boolean variable *x*2. Next, a logical and operation “*s*2 ← *s*1 ∧ *r*2, 1” is equivalent to another logical and operation “*s*2 ← *x*1 ∧ *x*2”. This is to say that the result of implementing “*r*1, 1 ← *r*1*,* 00 ∨ *x*1”, “*s*1 ← *s*01 ∧ *r*1, 1”, “*r*2, 1 ← *r*2*,* 00 ∨ *x*2” and “*s*2 ← *s*1 ∧ *r*2, 1” is the same as that of implementing “*s*2 ← *x*1 ∧ *x*2”. Therefore, four results of implementing *F*(*x*10, *x*20) = *x*10 ∧ *x*20, *F*(*x*10, *x*21) = *x*10 ∧ *x*21, *F*(*x*11, *x*20) = *x*11 ∧ *x*20 and *F*(*x*11, *x*21) = *x*11 ∧ *x*21 are respectively *s*20 (false), *s*20 (false), *s*20 (false) and *s*21 (true). Because 22 inputs of two Boolean variables implement those instructions above not to access or modify the same input and the same auxiliary Boolean variable, we can *parallelize* them without any error.

**3.2.3 Solution Space of Solving an Instance of the Satisfiability Problem**

For the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 of the search problem, its domain is {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2} and its range is {0, 1}. We regard its domain as its solution space in which there are four possible choices that satisfy *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 = 1. We use a basis {(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)} of the four-dimensional Hilbert space to construct solution space {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2}. We make use of (1, 0, 0, 0) to encode Boolean variable *x*10 and Boolean variable *x*20. Next, we use (0, 1, 0, 0) to encode Boolean variable *x*11 and Boolean variable *x*20. We apply (0, 0, 1, 0) to encode Boolean variable *x*10 and Boolean variable *x*21. Finally, we use (0, 0, 0, 1) to encode Boolean variable *x*11 and Boolean variable *x*21.

We use a linear combination of each element in the basis that is × (1, 0, 0, 0) + × (0, 1, 0, 0) + × (0, 0, 1, 0) + × (0, 0, 0, 1) = (, , , ) to construct solution space {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2}. The amplitude of each possible choice is all and the sum to the square of the absolute value of each amplitude is one. Because the length of the vector is one, it is a unit vector. This is to say that we use a unit vector to encode all of the possible choices that satisfy *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2. We call the square of the absolute value of each amplitude as the cost (the successful probability) of that choice that satisfies the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2. The cost (the successful probability) of the answer(s) is close to one as soon as possible.

**3.2.4 Implementing Solution Space to an Instance of the Satisfiability Problem**

In Listing 3.1, the program in the backend *ibmqx4* with five quantum bits in **IBM**’s quantum computer is to solve an instance of the satisfiability problem with *F*(*x*1, *x*2) = *x*1 ∧ *x*2 in which we illustrate how to write a quantum program to find values of each Boolean variable so that the whole formula has the value 1. Figure 3.2 is the quantum circuit of constructing solution space to an instance of the satisfiability problem with *F*(*x*1, *x*2) = *x*1 ∧ *x*2. The statement “OPENQASM 2.0;” on line one of Listing 3.1 is to indicate that the program is written with version 2.0 of Open QASM. Next, the statement “include "qelib1.inc";” on line two of Listing 3.1 is to continue parsing the file “qelib1.inc” as if the contents of the file were pasted at the location of the include statement, where the file “qelib1.inc” is **Quantum Experience (QE) Standard Header** and the path is specified relative to the current working directory.

|  |
| --- |
| 1. OPENQASM 2.0; 2. include "qelib1.inc"; 3. qreg q[5]; 4. creg c[5]; 5. x q[0]; 6. h q[3]; 7. h q[4]; 8. h q[0]; |

Listing 3.1: The program of solving an instance of the satisfiability problem with *F*(*x*1, *x*2) = *x*1 ∧ *x*2.

Next, the statement “qreg q[5];” on line three of Listing 3.1 is to declare that in the program there are five quantum bits. In the left top of Figure 3.2, five quantum bits are subsequently q[0], q[1], q[2], q[3] and q[4]. The initial value of each quantum bit is set to |0>. We use quantum bit q[3] to encode Boolean variable *x*1. We make use of quantum bit q[4] to encode Boolean variable *x*2. We apply quantum bit q[2] to encode auxiliary Boolean variable *s*2. We use quantum bit q[0] as an auxiliary working bit. We do not use quantum bit q[1].



Figure 3.2: The quantum circuit of constructing solution space to an instance of the satisfiability problem with *F*(*x*1, *x*2) = *x*1 ∧ *x*2.

For the convenience of our explanation, q[k]0 for 0 ≤ *k* ≤ 4 is to represent the value 0 of q[k] and q[k]1 for 0 ≤ *k* ≤ 4 is to represent the value 1 of q[k]. Similarly, for the convenience of our explanation, an initial state vector of constructing solution space to an instance of the satisfiability problem with *F*(*x*1, *x*2) = *x*1 ∧ *x*2 is as follows:

|Φ0> = |q[4]0> |q[3]0> |q[2]0> |q[1]0> |q[0]0> = |0> |0> |0> |0> |0> = |00000>.

Then, the statement “creg c[5];” on line four of Listing 3.1 is to declare that there are five classical bits in the program. In the left bottom of Figure 3.2, five classical bits are respectively c[0], c[1], c[2], c[3] and c[4]. The initial value of each classical bit is set to 0.

Next, the three statements “x q[0];”, “h q[3];” and “h q[4];” on line five through seven of Listing 3.1 is to implement one ***X*** gate (one ***NOT*** gate) and two Hadamard gates of the *first* time slot of the quantum circuit in Figure 3.2. The statement “x q[0];” actually completes × = = (|1>). This indicates that the statement “x q[0];” on line five of Listing 3.1 inverts |q[0]0> (|0>) into |q[0]1> (|1>). The two statements “h q[3];” and “h q[4];” both actually complete × = = = ( + ) = (|0> + |1>). This is to say that converting q[3] from one state |0> to another state (|0> + |1>) (its superposition) and converting q[4] from one state |0> to another state (|0> + |1>) (its superposition) are completed. Therefore, the superposition of the two quantum bits q[4] and q[3] is ( (|0> + |1>)) ( (|0> + |1>)) = (|0> |0> + |0> |1> + |1> |0> + |1> |1>) = (|00> + |01> + |10> + |11>). Because in the *first* time slot of the quantum circuit in Figure 3.2 there is no quantum gate to act on quantum bits q[2] and q[1], their current states |q[2]0> and |q[1]0> are not changed. This is to say that we obtain the following new state vector

|Φ1> = ( (|q[4]0> + |q[4]1>)) ( (|q[3]0> + |q[3]1)) (|q[2]0> |q[1]0> |q[0]1>)

= (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + |q[4]1> |q[3]1>) (|q[2]0>

|q[1]0> |q[0]1>)

= (|0> |0> + |0> |1> + |1> |0> + |1> |1>) (|0> |0>|1>).

Next, the statement “h q[0];” on line *eight* of Listing 3.1 is to implement one Hadamard gate of the *second* time slot of the quantum circuit in Figure 3.2. The statement “h q[0];” actually completes × = = = ( − ) = (|0> − |1>). This indicates that converting q[0] from one state |1> to another state (|0> − |1>) (its superposition) is completed. Because in the *second* time slot of the quantum circuit in Figure 3.2 there is no quantum gate to act on quantum bits q[4] through q[1], their current states are not changed. This indicates that we obtain the following new state vector

|Φ2> = ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + |q[4]1> |q[3]1>)) (|q[2]0>

|q[1]0>) (|q[0]0> − |q[0]1>))

= ( (|0> |0> + |0> |1> + |1> |0> + |1> |1>)) (|0> |0>) (|0> − |1>)).

In the new state vector |Φ2>, state |q[4]0> |q[3]0> encodes Boolean variable *x*10 and Boolean variable *x*20. State |q[4]0> |q[3]1> encodes Boolean variable *x*11 and Boolean variables *x*20. State |q[4]1> |q[3]0> encodes Boolean variable *x*10 and Boolean variable *x*21. State |q[4]1> |q[3]1> encodes Boolean variable *x*11 and Boolean variable *x*21. The amplitude of each choice is and the cost (the successful possibility) of becoming the answer(s) to each choice is the same and is equal to = 1/4.

**3.2.5 The Oracle to an Instance of the Satisfiability Problem**

The Oracle is to have the ability to *recognize* solutions to the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 of the satisfiability problem. The Oracle is to multiply the probability amplitude of the answer(s) by −1 and leaves any other amplitude unchanged. The Oracle of solving the satisfiability problem with the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 is a (22 × 22) matrix *B* that is equal to .

We assume that a (22 × 22) matrix *B*+ is the conjugate transpose of *B*. Because the transpose of *B* is equal to *B* and each element in the transpose of *B* is a real, the conjugate transpose of *B* is also equal to *B*. Hence, we obtain *B*+ = *B*. Because *B* and *B*+ are almost a (22 × 22) identity matrix, *B* × *B*+ = , and *B*+ × *B* = . Thus, we obtain *B* × *B*+ = *B*+ × *B*. This is to say that it is a unitary matrix (operator) to solve the satisfiability problem with the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2. Implementing the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 in the satisfiability problem is equivalent to implement the Oracle that is × = = × + × + × + × = |00> + |01> + |10> + |11>.

Four computational basis vectors , , and encode four states |00>, |01>, |10> and |11> and their current amplitudes are respectively (, , () and (). State |00> (|q[4]0> |q[3]0>) with the amplitude () encodes Boolean variable *x*10 and Boolean variable *x*20. State |01> (|q[4]0> |q[3]1>) with the amplitude () encodes Boolean variable *x*11 and Boolean variable *x*20. State |10> (|q[4]1> |q[3]0>) with the amplitude () encodes Boolean variable *x*10 and Boolean variable *x*21. State |11> (|q[4]1> |q[3]1>) with the amplitude () encodes Boolean variable *x*11 and Boolean variable *x*21. This is to say that the Oracle multiplies the probability amplitude of the answer with Boolean variable *x*11and Boolean variable *x*21 by −1 and leaves any other amplitude unchanged.

**3.2.6 Implementing the Oracle to an Instance of the Satisfiability Problem**

We use one ***CCNOT*** gate to implement the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 of the satisfiability problem. We use quantum bit q[3] to encode Boolean variable *x*1, we use quantum bit q[4] to encode Boolean variable *x*2 and we use quantum bit q[2] to encode Boolean variable *s*2. So, quantum bits q[3], q[4], q[2] are respectively the first control bit, the second control bit and the target bit of the ***CCNOT*** gate. Because we use the ***CCNOT*** gate to implement a logical and operation, the initial value to quantum bit q[2] is set to |0>.

From line *nine* through line *twenty-three* in Listing 3.1, there are the fifteen statements. They are subsequently “h q[2];”, “cx q[4],q[2];”, “tdg q[2];”, “cx q[3],q[2];”, “t q[2];”, “cx q[4],q[2];”, “tdg q[2];”, “cx q[3],q[2];”, “t q[4];”, “t q[2];”, “cx q[3],q[4];”, “h q[2];”, “t q[3];”, “tdg q[4];” and “cx q[3], q[4];”. They implement the ***CCNOT*** gate that completes the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2. Figure 3.3 is the qua-

|  |
| --- |
| **Listing 3.1 continued…**  // We use the following *fifteen* statements to implement a ***CCNOT*** gate.   1. h q[2]; 2. cx q[4],q[2]; 3. tdg q[2]; 4. cx q[3],q[2]; 5. t q[2]; 6. cx q[4],q[2]; 7. tdg q[2]; 8. cx q[3],q[2]; 9. t q[4]; 10. t q[2]; 11. cx q[3],q[4]; 12. h q[2]; 13. t q[3]; 14. tdg q[4]; 15. cx q[3], q[4]; |

ntum circuit of implementing the Oracle to an instance of the satisfiability problem with

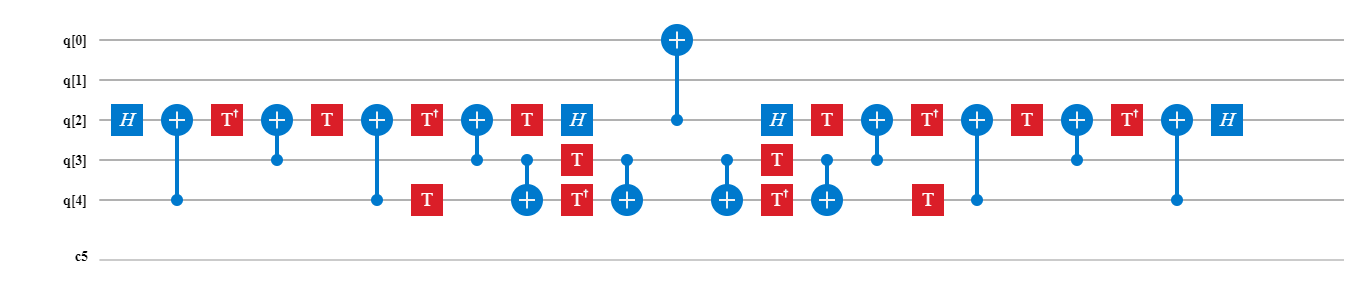


Figure 3.3: The quantum circuit of implementing the Oracle to an instance of the satisfiability problem with *F*(*x*1, *x*2) = *x*1 ∧ *x*2.

*F*(*x*1, *x*2) = *x*1 ∧ *x*2. They take the state vector |Φ2> = ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + |q[4]1> |q[3]1>)) (|q[2]0> |q[1]0>) (|q[0]0> − |q[0]1>)) as their input. After they actually implement six ***CNOT*** gates, two Hadamard gates, three ***T***+ gates and four ***T*** gates from the *first* time slot through the *eleventh* time slot in Figure 3.3, we obtain the following new state vector

|Φ3> = ( (|q[4]0> |q[3]0> |q[2]0> + |q[4]0> |q[3]1> |q[2]0> + |q[4]1> |q[3]0> |q[2]0> +

|q[4]1> |q[3]1> |q[2]1>)) (|q[1]0>) (|q[0]0> − |q[0]1>))

= ( (|0> |0> |0> + |0> |1> |0> + |1> |0> |0> + |1> |1> |1>)) (|0>) (|0> − |1>)).

Next, from line *twenty-four* in Listing 3.1, the statement “cx q[2],q[0];” takes the new state vector |Φ3> as its input. It multiplies the probability amplitude of the answer

|  |
| --- |
| **Listing 3.1 continued…**  // The Oracle multiplies the probability amplitude of the answer *x*11 *x*21 by −1 and  // leaves any other amplitude unchanged.   1. cx q[2],q[0]; |

|q[4]1> |q[3]1> encoding *x*11 *x*21 by −1 and leaves any other amplitude unchanged. This is to say that after the statement “cx q[2],q[0];” implements the ***CNOT*** gate in the *twelfth* time slot in Figure 3.3, we obtain the following new state vector

|Φ4> = ( (|q[4]0> |q[3]0> |q[2]0> + |q[4]0> |q[3]1> |q[2]0> + |q[4]1> |q[3]0> |q[2]0> +

(−1) |q[4]1> |q[3]1> |q[2]1>)) (|q[1]0>) (|q[0]0> − |q[0]1>))

= ( (|0> |0> |0> + |0> |1> |0> + |1> |0> |0> + (−1) |1> |1> |1>)) (|0>) (|0> −

|1>)).

Because quantum operations are reversible by nature, executing the reversed order of implementing the ***CCNOT*** gate can restore the auxiliary quantum bits to their initial states. From line *twenty-five* through line *thirty-nine* in Listing 3.1, there are the *fifteen* statements. They are “cx q[3],q[4];”, “tdg q[4];”, “t q[3];”, “h q[2];”, “cx q[3],q[4];”, “t q[2];”, “t q[4];”, “cx q[3],q[2];”, “tdg q[2];”, “cx q[4],q[2];”, “t q[2];”, “cx q[3],q[2];”, “tdg q[2];”, “cx q[4],q[2];” and “h q[2];”. They run the reversed order of implementing

|  |
| --- |
| **Listing 3.1 continued…**  // Because quantum operations are reversible by nature, executing the reversed  // order of implementing the ***CCNOT*** gate can restore the auxiliary quantum bits  // to their initial states.   1. cx q[3],q[4]; 2. tdg q[4]; 3. t q[3]; 4. h q[2]; 5. cx q[3],q[4]; 6. t q[2]; 7. t q[4]; 8. cx q[3],q[2]; 9. tdg q[2]; 10. cx q[4],q[2]; 11. t q[2]; 12. cx q[3],q[2]; 13. tdg q[2]; 14. cx q[4],q[2]; 15. h q[2]; |

the ***CCNOT*** gate that completes the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2. They take the new state vector |Φ4> as their input. After they actually complete six ***CNOT*** gates, two Hadamard gates, three ***T***+ gates and four ***T*** gates from the *thirteenth* time slot through the *last* time slot in Figure 3.3, we obtain the following new state vector

|Φ5> = ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1> |q[3]1>))

(|q[2]0> |q[1]0>) (|q[0]0> − |q[0]1>))

= ( (|0> |0> + |0> |1> + |1> |0> + (−1) |1> |1>)) (|0> |0>) (|0> − |1>)).

In the state vector |Φ2>, the amplitude of each element in solution space {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2} is (1/2). In the state vector |Φ5>, the amplitude to three elements *x*10 *x*20, *x*10 *x*21, *x*11 *x*20 in solution space is all (1/2) and the amplitude to the element *x*11 *x*21 in solution space is (−1/2). This indicates that *thirty-one* statements from line *nine* through *thirty-nine* in Listing 3.1 complete × that is to complete the Oracle of solving the satisfiability problem with the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2.

**3.2.7 The Grover Diffusion Operator to Amplify the Amplitude of the Answers in the Satisfiability Problem**

We assume that a (2*n* × 1) vector |*u*> is and another (2*n* × 1) vector |*v*> is . The transpose of |*v*> is a (1 × 2*n*) vector that is . **Definition 3-2** introduces the outer product |*u*> <*v*| of two vectors |*u*> and |*v*>.

**Definition 3-2**: The outer product |*u*> <*v*| of two vectors |*u*> and |*v*> is a (2*n* × 2*n*) matrix *W* that is × = .

We assume that represents *n* Hadamard gates and represents *n* quantum bits in which the value of each quantum bit is equal to |0>. After we use a unitary operator to operate *n* quantum bits, , the state |0> of each quantum bit is converted into its superposition (|0> + |1>). This is to say that the superposition to *n* quantum bits with is = ( (|0> + |1>))) = = ( + … + ) = . is the uniform superposition of states, is a (2*n* × 1) vector and its length is one. This indicates that it is a unit vector.

The matrix *D* that is a (2*n* × 2*n*) matrix defines the Grover diffusion operator *D* as follows:

*Da*, *b* = if *a ≠ b* and *Da*, *a* = − 1.

This diffusion transform, *D*, can be implemented as *D* = 2 − = (2 − ) . The rotation matrix *R* that is a (2*n* × 2*n*) matrix defines a phase shifter operator, (2 − ), as follows:

*Ra*, *b* = 0 if *a ≠ b*; *Ra*, *a* = 1 if *a* = 0; *Ra*, *a* = −1 if *a ≠* 0.

This implies that the phase shifter operator, (2 − ), negates all the states except for |0>. It turns out that a quantum circuit with a phase shift operator, 2 − , that negates all the states except for |0> sandwiched between gates can implement the Grover diffusion operator *D*. We use **Lemma 3-1** to show that the Grover diffusion operator, *D* = 2 − = (2 − ) , is a unitary operator.

**Lemma 3-1**: The Grover diffusion operator, *D* = 2 − = (2 − ) , is a unitary operator.

**Proof**:

The outer product of the (2*n* × 1) vector with itself leads to a (2*n* × 2*n*) matrix *V* that is × = . Subtracting the identity matrix from the double of the (2*n* × 2*n*) matrix *V*, we obtain a new (2*n* × 2*n*) matrix *D*that is − = .

We assume that a (2*n* × 2*n*) matrix *D*+ is the conjugate transpose of *D*. We assume that *N* is equal to (1/2*n*). Because the transpose of *D* is equal to *D* and each element in the transpose of *D* is a real, the conjugate transpose of *D* is equal to *D*. Hence, we obtain *D*+ = *D*. Since (*D* × *D*+)*a*, *a* = ()2 + ()2 × (*N* − 1) = 1 and (*D* × *D*+)*a*, *b* = () × () + () × () + ()2 × (*N* − 2) = 0, we obtain *D* × *D*+ = . Because *D*+ = *D* and *D* × *D*+ = , we obtain *D*+ × *D* = *D* × *D*+ = . Because = () and = , we obtain *D* = 2 − = 2 − = 2 − = (2 − ) . From the statements above, it is at once inferred that the Grover diffusion operator, *D* = 2 − = (2 − ) is a unitary operator. ◼

**3.2.8 Implementing the Grover Diffusion Operator to Amplify the Amplitude of the Answer in an Instance of the Satisfiability Problem**

The new state vector |Φ5> is ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1> |q[3]1>)) (|q[2]0> |q[1]0>) (|q[0]0> − |q[0]1>)). It consists of two subsystem. The first subsystem is ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1> |q[3]1>)) and the second subsystem is (|q[2]0> |q[1]0>) (|q[0]0> − |q[0]1>)). The two subsystems are independent each other. Amplifying the amplitude of each answer in the satisfiability problem with the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 just needs to consider the first subsystem in the new state vector |Φ5>. Because for the satisfiability problem with the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 the (22 × 1) vector encodes the first subsystem of the new state vector |Φ5> and is a (22 × 22) diffusion operator, amplifying the amplitude of the answer is to complete × = . This is to say that the amplitude of the answer *x*11 *x*21 is one and the amplitude of other three possible choices *x*10 *x*20, *x*10 *x*21 and *x*11 *x*20 is all zero.

The quantum circuit in Figure 3.4 implements the Grover diffusion operator,

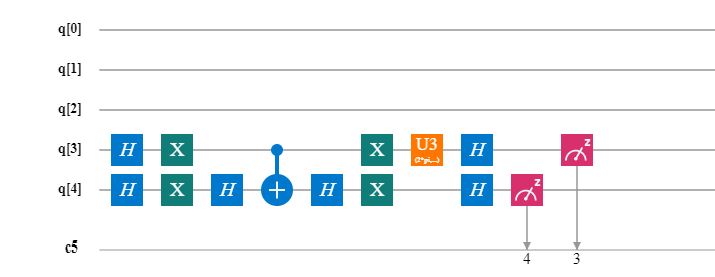


Figure 3.4: The quantum circuit of implementing the Grover diffusion operator, (2 − ) , to an instance of the satisfiability problem with *F*(*x*1, *x*2) = *x*1 ∧ *x*2.

(2 − ) . The statements “h q[3];” and “h q[4];” from line *for*-

|  |
| --- |
| **Listing 3.1 continued…**  //We complete the amplitude amplification of the answer.   1. h q[3]; 2. h q[4]; |

*ty* through line *forty-one* in Listing 3.1 complete the first *H*⊗2 gate in the diffusion operator, (2 − ) . They takes ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1> |q[3]1>)) as their input and complete two Hadamard gates in the first time slot of Figure 3.4. State ( |q[4]0> |q[3]0>) is converted into state (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + |q[4]1> |q[3]1>). State ( |q[4]0> |q[3]1>) is converted into state (|q[4]0> |q[3]0> − |q[4]0> |q[3]1> + |q[4]1> |q[3]0> − |q[4]1> |q[3]1>). State ( |q[4]1> |q[3]0>) is converted into state (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> − |q[4]1> |q[3]0> − |q[4]1> |q[3]1>). State ( |q[4]1> |q[3]1>) is converted into state (|q[4]0> |q[3]0> − |q[4]0> |q[3]1> − |q[4]1> |q[3]0> + |q[4]1> |q[3]1>). This is to say that we obtain the following new state vector

|Φ6> = ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1> |q[3]1>)).

Next, from line forty-two through forty-three in Listing 3.1 the two statements “x q[3]” and “x q[4]” implement two ***NOT*** gates in the second time slot of Figure 3.4. They take ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1> |q[3]1>)) in the new state vector |Φ6> as their input. State ( |q[4]0> |q[3]0>) is converted into st-

|  |
| --- |
| **Listing 3.1 continued…**  // We complete phase shifters.   1. x q[3]; 2. x q[4]; |

ate (|q[4]1> |q[3]1>). State ( |q[4]0> |q[3]1>) is converted into state (|q[4]1> |q[3]0>). State ( |q[4]1> |q[3]0>) is converted into state (|q[4]0> |q[3]1>). State ( |q[4]1> |q[3]1>) is converted into state (|q[4]0> |q[3]0>). This indicates that we obtain the following new state vector

|Φ7> = ( ((−1) |q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + |q[4]1> |q[3]1>)).

Next, from line forty-four in Listing 3.1 the statement “h q[4];” implements one Hadamard gate in the third time slot of Figure 3.4. They take ( ((−1) |q[4]0> |q[3]0> +

|  |
| --- |
| **Listing 3.1 continued…**   1. h q[4]; |

|q[4]0> |q[3]1> + |q[4]1> |q[3]0> + |q[4]1> |q[3]1>)) in the new state vector |Φ7> as their input. State ( |q[4]0> |q[3]0>) is converted into state (|q[4]0> |q[3]0> + |q[4]1> |q[3]0>). State ( |q[4]0> |q[3]1>) is converted into state (|q[4]0> |q[3]1> + |q[4]1> |q[3]1>). State ( |q[4]1> |q[3]0>) is converted into state (|q[4]0> |q[3]0> − |q[4]1> |q[3]0>). State ( |q[4]1> |q[3]1>) is converted into state (|q[4]0> |q[3]1> − |q[4]1> |q[3]1>). This is to say that we obtain the following new state vector

|Φ8> = ( (|q[4]0> |q[3]1> − |q[4]1> |q[3]0>)).

Next, from the line forty-five in Listing 3.1 the statement “cx q[3],q[4];” implements one ***CNOT*** gate in the fourth time slot of Figure 3.4. They take (|q[4]0>

|  |
| --- |
| **Listing 3.1 continued…**   1. cx q[3],q[4]; |

|q[3]1> − |q[4]1> |q[3]0>) in the new state vector |Φ8> as their input. State ( |q[4]0> |q[3]1>) is converted into state ( |q[4]1> |q[3]1>). State ( |q[4]1> |q[3]0>) is converted into state ( |q[4]1> |q[3]0>). This indicates that we obtain the following new state vector

|Φ9> = ( (|q[4]1> |q[3]1> − |q[4]1> |q[3]0>)).

Next, from the line forty-six in Listing 3.1 the statement “h q[4];” implements one Hadamard gate in the fifth time slot of Figure 3.4. They take ( (|q[4]1> |q[3]1> − |q[4]1> |q[3]0>)) in the new state vector |Φ9> as their input. State ( |q[4]1> |q[3]1>) is

|  |
| --- |
| **Listing 3.1 continued…**   1. h q[4]; |

converted into state (|q[4]0> |q[3]1> − |q[4]1> |q[3]1>). State ( |q[4]1> |q[3]0>) is converted into state (|q[4]0> |q[3]0> − |q[4]1> |q[3]0>). This indicates that we obtain the following new state vector

|Φ10> = ((−1) |q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1>|q[3]1>).

Next, from the line forty-seven through line forty-eight in Listing 3.1 the two statements “x q[4];” and “x q[3];” implements two ***X*** (***NOT***) gates in the *sixth* time slot

|  |
| --- |
| **Listing 3.1 continued…**   1. x q[4]; 2. x q[3]; |

of Figure 3.4. They take ((−1) |q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1>|q[3]1>) in the new state vector |Φ10> as their input. State ( |q[4]0> |q[3]0>) is converted into state ( |q[4]1> |q[3]1>). State ( |q[4]0> |q[3]1>) is converted into state ( |q[4]1> |q[3]0>). State ( |q[4]1> |q[3]0>) is converted into state ( |q[4]0> |q[3]1>). State ( |q[4]1> |q[3]1>) is converted into state ( |q[4]0> |q[3]0>). This is to say that we obtain the following new state vector

|Φ11> = ((−1) |q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1>|q[3]1>).

Next, from line forty-nine in Listing 3.1, the statement “u3(2\*pi,0\*pi,0\*pi) q[3];” completes one u3(2\*pi,0\*pi,0\*pi) gate that is a (2 × 2) matrix in the s-

|  |
| --- |
| **Listing 3.1 continued…**   1. u3(2\*pi,0\*pi,0\*pi) q[3]; |

eventh time slot of Figure 3.4. It takes ((−1) |q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1>|q[3]1>) in the new state vector |Φ11> as its input. State ( |q[4]0> |q[3]0>) receives one phase (−1). State ( |q[4]0> |q[3]1>) receives one phase (−1). State ( |q[4]1> |q[3]0>) receives one phase (−1). State ( |q[4]1> |q[3]1>) receives one phase (−1). This indicates that we obtain the following new state vector

|Φ12> = ((−1 × −1) |q[4]0> |q[3]0> + (−1) |q[4]0> |q[3]1> + (−1) |q[4]1> |q[3]0> + (−1

× −1) |q[4]1>|q[3]1>)

= (|q[4]0> |q[3]0> + (−1) |q[4]0> |q[3]1> + (−1) |q[4]1> |q[3]0> + |q[4]1>|q[3]1>).

Those quantum gates from the *second* time slot through the seventh time slot of Figure 3.4 completes the phase shifter, (2 − ), in the Grover diffusion operator to an instance of the satisfiability problem with *F*(*x*1, *x*2) = *x*1 ∧ *x*2. Next, from the line fifty through line fifty-one in Listing 3.1 the two statements “h q[4];”

|  |
| --- |
| **Listing 3.1 continued…**   1. h q[4]; 2. h q[3]; |

and “h q[3];” implement two Hadamard gates in the eighth time slot of Figure 3.4. They complete the second *H*⊗2 gate in the diffusion operator, (2 − ) . They take (|q[4]0> |q[3]0> + (−1) |q[4]0> |q[3]1> + (−1) |q[4]1> |q[3]0> + |q[4]1>|q[3]1>) in the new state vector |Φ12> as their input. State ( |q[4]0> |q[3]0>) is converted into state ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + |q[4]1> |q[3]1>)). State ( |q[4]0> |q[3]1>) is converted into state ( (|q[4]0> |q[3]0> − |q[4]0> |q[3]1> + |q[4]1> |q[3]0> − |q[4]1> |q[3]1>)). State ( |q[4]1> |q[3]0>) is converted into state ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> − |q[4]1> |q[3]0> − |q[4]1> |q[3]1>)). State ( |q[4]1> |q[3]1>) is converted into state ( (|q[4]0> |q[3]0> − |q[4]0> |q[3]1> − |q[4]1> |q[3]0> + |q[4]1> |q[3]1>)). This is to say that we obtain the following new state vector

|Φ13> = |q[4]1>|q[3]1>.

Next, from line fifty-two in Listing 3.1 the statement “measure q[4] -> c[4];” is to measure the fifth quantum bit q[4] and to record the measurement outcome by overwriting the fifth classical bit c[4]. From line fifty-three in Listing 3.1 the statement “measure q[3] -> c[3];” is to measure the fourth quantum bit q[3] and to record the measurement outcome by overwriting the fourth classical bit c[3]. They complete the measurement from the ninth time slot through the tenth time slot of Figure 3.4.

|  |
| --- |
| **Listing 3.1 continued…**  // We complete the measurement of the answer.   1. measure q[4] -> c[4]; 2. measure q[3] -> c[3]; |

In the backend *ibmqx4* with five quantum bits in **IBM**’s quantum computers, we use the command “simulate” to execute the program in Listing 3.1. The measured result appears in Figure 3.5. From Figure 3.5, we obtain the answer 11000 (c[4] = q[4] = |1>, c[3] = q[3] = |1>, c[2] = q[2] = |0>, c[1] = q[1] = |0> and c[0] = q[0] = |0>) with the probability 1 (100%). This is to say that with the possibility 1 (100%) we obtain that the value of quantum bit q[3] is |1> and the value of quantum bit q[4] is |1>. For solving an instance of the satisfiability problem with *F*(*x*1, *x*2) = *x*1 ∧ *x*2 we use quantum bit q[3] to encode Boolean variable *x*1 and use quantum bit q[4] to encode Boolean variable *x*2. Therefore, the answer is to that the value of Boolean variable *x*1 is 1 (one) and the value of Boolean variable *x*2 is 1 (one).

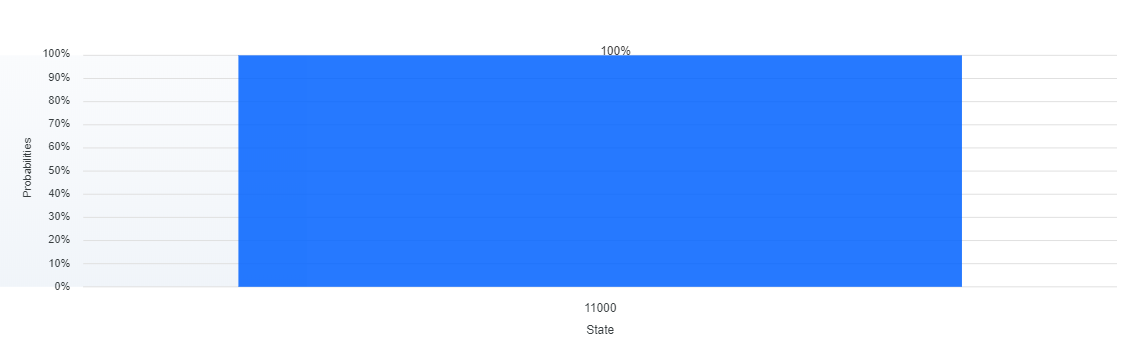


Figure 3.5: After the measurement to solve an instance of the satisfiability problem with *F*(*x*1, *x*2) = *x*1 ∧ *x*2 is completed, we obtain the answer 11000 with the probability 1 (100%).

**3.2.9 The Quantum Search Algorithm to the Satisfiability Problem**

A satisfiability problem has *n* Boolean variables and any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*). Any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*) is a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*. Each clause *Cj* for 1 ≤ *j* ≤ *m* is a formula of the form *x*1 ∨ *x*2 ∨ … *xn* − 1 ∨ *xn* for each Boolean variable *xk* to 1 ≤ *k* ≤ *n*. The question is to find values of each Boolean variable so that any given oracular function *Of*(*x*1 *x*2 …, *xn* − 1 *xn*) (the whole formula) has the value 1. We use the quantum search algorithm to find one of *M* solutions to the question, where 0 ≤ *M* ≤ 2*n*.

The quantum circuit in Figure 3.6 is to implement the quantum search algorithm to solve an instance of the satisfiability problem with *n* Boolean variables and *m* clauses. The first quantum register in the left top of Figure 3.6 is (). This is to say that the initial value of each quantum bit is |0>. The second quantum register in the left bottom of Figure 3.6 has (*m* × *n* + 2 × *m* + 1) quantum bits and is an auxiliary quantum register.

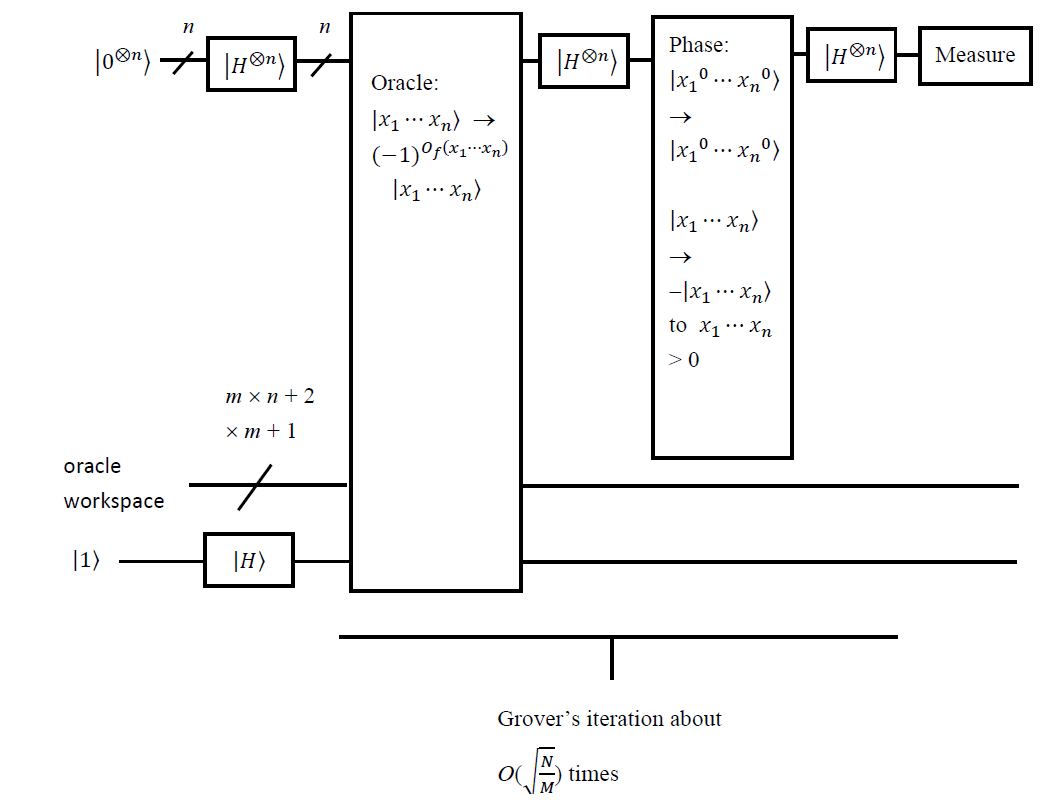


Figure 3.6: Circuit of implementing the quantum search algorithm to solve an instance of the satisfiability problem with *n* Boolean variables and *m* clauses.

The initial value of each quantum bit in the second quantum register is |0> or |1> that is dependent on implementing a logical or operation or a logical and operation. The third quantum register in the left bottom of Figure 3.6 is ().

**3.2.10 The First Stage of the Quantum Search Algorithm to the Satisfiability Problem**

In Figure 3.6, the first stage of the quantum search problem to solve an instance of the satisfiability problem with *n* Boolean variables and *m* clauses is to use *n* Hadamardgates () to operate the first quantum register (). This indicates that it generates the superposition of *n* quantum bits that is () = (). Solution space of solving an instance of the satisfiability problem with *n* Boolean variables and *m* clauses is {*x*1 *x*2 … *xn* − 1 *xn* | ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*}. In the first stage of the quantum search algorithm, state () with the amplitude () encodes the *first* element *x*10 *x*20 … *xn*0 in solution space and so on with that state () with the amplitude () encodes the *last* element *x*11 *x*21 … *xn*1 in solution space. In the first stage of the quantum search algorithm, it uses one Hadamard gate to operate the third quantum register (). This is to say that it generates the superposition of the third quantum register () that is ().

**3.2.11 The Second Stage of the Quantum Search Algorithm to the Satisfiability Problem**

In Figure 3.6, the *second* stage of the quantum search algorithm to solve an instance of the satisfiability problem with *n* Boolean variables and *m* clauses is to complete the Oracle. The Oracle is to have the ability to *recognize* solutions in the satisfiability problem with *n* Boolean variables and *m* clauses. The Oracle is to multiply the probability amplitude of the answer(s) by −1 and leaves any other amplitude unchanged. Any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*) is a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*. Each clause *Cj* for 1 ≤ *j* ≤ *m* is a formula of the form *x*1 ∨ *x*2 ∨ … *xn* − 1 ∨ *xn* for each Boolean variable *xk* to 1 ≤ *k* ≤ *n*. Therefore, for implementing any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*), we need to complete (*m* × *n*) logical or operations and (*m*) logical and operations.

In oracle workspace in the second stage of the quantum search algorithm, we use auxiliary quantum bits |*rj, k*> for 1 ≤ *j* ≤ *m* and 0 ≤ *k* ≤ *n* to encode auxiliary Boolean variables *rj, k* for 1 ≤ *j* ≤ *m* and 0 ≤ *k* ≤ *n*. We use auxiliary quantum bits |*sj*> for 0 ≤ *j* ≤ *m* to encode auxiliary Boolean variables *sj* for 0 ≤ *j* ≤ *m*. Since we use auxiliary quantum bits |*rj,* 0> for 1 ≤ *j* ≤ *m* as the first operand of the first logical or operation (“∨”) in each clause, the initial value of each auxiliary quantum bit |*rj,* 0> for 1 ≤ *j* ≤ *m* is set to |0>. This implies that this setting does not change the correct result of the first logical or operation in each clause. We use a ***CCNOT*** gate and four ***NOT*** gates to implement each logical or operation in each clause. We apply auxiliary quantum bits |*rj, k*> for 1 ≤ *j* ≤ *m* and 1 ≤ *k* ≤ *n* to store the result of implementing the logical or operations in each clause. This is to say that each auxiliary quantum bit |*rj, k*> for 1 ≤ *j* ≤ *m* and 1 ≤ *k* ≤ *n* is actually the target bit of a ***CCNOT*** gate of implementing a logical or operation. Thus, the initial value of each auxiliary quantum bit |*rj, k*> for 1 ≤ *j* ≤ *m* and 1 ≤ *k* ≤ *n* is set to |1>.

We use an auxiliary quantum bit |*s*0> as the first operand of the first logical and operation (“∧”) in any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*) with a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*. The initial value of the auxiliary quantum bit |*s*0> is set to |1>. This is to say that this setting does not change the correct result of the first logical and operation in any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*) with a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*. We use a ***CCNOT*** gate to implement each logical and operation in any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*) with a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*. We apply auxiliary quantum bits |*sj*> for 1 ≤ *j* ≤ *m* to store the result of implementing the logical and operations in any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*) with a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*. This indicates that each auxiliary quantum bit |*sj*> for 1 ≤ *j* ≤ *m* is actually the target bit of a ***CCNOT*** gate of implementing a logical and operation. Therefore, the initial value of each auxiliary quantum bit |*sj*> for 1 ≤ *j* ≤ *m* is set to |0>.

A ***CCNOT*** gate and four ***NOT*** gate can implement a logical or operation. A ***CCNOT*** gate can implement a logical and operation. From Figure 3.1, implementing any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*) is to complete (*m* × *n*) logical or operations and (*m*) logical and operations. This is to say that implementing any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*) is to complete (*m* × *n* + *m*) ***CCNOT*** gates and (4 × *m* × *n*) ***NOT*** gates. Quantum bit |*sm*> is to store the result of implementing any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*). If the value to quantum bit |*sm*> is equal to |1>, then any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*) has the value 1 (one). Otherwise, it has the value 0 (zero).

We use one ***CNOT*** gate to multiply the probability amplitude of the answer(s) by −1 and to leave any other amplitude unchanged, where quantum bit () is the target bit of the ***CNOT*** gate and quantum bit (|*sm*>) is the control bit of the ***CNOT*** gate. When the value of the control bit (|*sm*>) is equal to (|1>), the target bit becomes () = (−1) (). This is to multiply the probability amplitude of the answer(s) by −1. When the value of the control bit (|*sm*>) is equal to (|0>), the target bit still is (). This is to leave any other amplitude unchanged.

Because quantum operations are reversible by nature, executing the reversed order of implementing any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*) can restore the auxiliary quantum bits to their initial states. This is to say that the second stage of the quantum search algorithm to solve the satisfiability problem with *n* Boolean variables and *m* clauses converts () into (). The cost of completing the Oracle in the second stage of the quantum search algorithm in Figure 3.6 is to implement (2 × (*m* × *n* + *m*)) ***CCNOT*** gates, (8 × *m* × *n*) ***NOT*** gates and one ***CNOT*** gate.

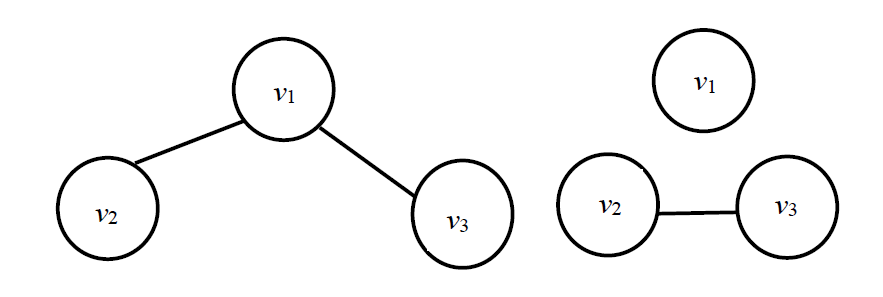
**3.2.12 The Third Stage of the Quantum Search Algorithm to the Satisfiability Problem**

In Figure 3.6, the *third* stage of the quantum search algorithm to solve an instance of the satisfiability problem with *n* Boolean variables and *m* clauses is to complete the Grover diffusion operator. Implementing the Grover diffusion operator is equivalent to implement (2 − ) . A phase shifter operator, (2 − ) negates all the states except for (). In Figure 3.6, the *third* stage of the quantum search problem is to use that the phase shift operator, 2 − , negates all the states except for () sandwiched between gates. This indicates that the *third* stage of the quantum search algorithm to solve an instance of the satisfiability problem with *n* Boolean variables and *m* clauses is to increase the amplitude of the answer(s) and to decrease the amplitude of the non-answer(s).

For solving an instance of the satisfiability problem with *n* Boolean variables and *m* clauses, we regard the Oracle and the Grover diffusion operator from the second stage through the third stage of the quantum search algorithm in Figure 3.6 as a subroutine. We call the subroutine as the *Grover iteration*. After repeat to execute the Grover iteration of O() times, the successful probability of measuring the answer(s) is at least (1/2). When the value of (*M* / *N*) is equal to (1 / 4), the successful probability of measuring the answer(s) is one (100%) with the Grover iteration of one time. This is the *best* case of the quantum search algorithm to solve an instance of the satisfiability problem with *n* Boolean variables and *m* clauses. When the value of *M* is equal to one, the successful probability of measuring the answer(s) is at least (1 / 2) with the Grover iteration of O() times. This is the *worst* case of the quantum search algorithm to solve an instance of the satisfiability problem with *n* Boolean variables and *m* clauses. This indicates that the quantum search algorithm to solve an instance of the satisfiability problem with *n* Boolean variables and *m* clauses only gives a quadratic speed-up.

**3.3 Introduction to the Maximal Clique Problem**

We assume that a graph *G* = (*V*, *E*) has *n* vertices and *θ* edges, where *V* is a set of vertices in *G* and *E* is a set of edges in *G*. For a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges, its *complementary* graph is to contain all edges missing in the *original* graph *G* = (*V*, *E*) with *n* vertices and *θ* edges. Therefore, we assume that its *complementary* graph  = (*V*, ) has *n* vertices and *m* edges in which each edge in is out of *E*, where *V* is a set of vertices in and is a set of edges in . This is to say that a graph *G* = (*V*, *E*) and its *complementary* graph = (*V*, ) has the same vertices and its *complementary* graph = (*V*, ) contains all edges missing in the *original* graph *G* = (*V*, *E*). In Figure 3.7a, the graph has three vertices {*v*1, *v*2, *v*3} and two edges {(*v*1, *v*2), (*v*1, *v*3)}. In Figure 3.7b, its *complementary* graph has the same three vertices {*v*1, *v*2, *v*3} and one edge {(*v*2, *v*3)} missing in the *original* graph in Figure 3.7a.



1. (b)

Figure 3.7: (a) The graph has three vertices and two edges. (b) Its *complementary* graph has the same vertices and one edge missing in the *original* graph.

Mathematically, a *clique* for a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges is defined as a set of vertices in which every vertex is connected to every other vertex by an edge. This is to say that mathematically, a *clique* for a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges is a *complete* subgraph to *G*. For the graph with three vertices {*v*1, *v*2, *v*3} and two edges {(*v*1, *v*2), (*v*1, *v*3)} in Figure 3.7a, there are eight subsets of vertices. Eight subsets of vertices are subsequently {*v*1, *v*2, *v*3}, {*v*1, *v*2}, {*v*1, *v*3}, {*v*2, *v*3}, {*v*1}, {*v*2}, {*v*3} and an empty subset of vertices that is {}.

Any two vertices connected in the *complementary* graph with three vertices {*v*1, *v*2, *v*3} and one edge {(*v*2, *v*3)} in Figure 3.7b are disconnected in the *original* graph with three vertices {*v*1, *v*2, *v*3} and two edges {(*v*1, *v*2), (*v*1, *v*3)} in Figure 3.7a. This is to say that any two vertices connected in the *complementary* graph cannot be members of the same clique. Because the edge (*v*2, *v*3) in the *complementary* graph in Figure 3.7b connects the two vertices *v*2 and *v*3, eight subsets of vertices containing the two vertices *v*2 and *v*3 are not a clique. A subset of vertices {*v*1, *v*2, *v*3} contains the two vertices *v*2 and *v*3, so it is not a clique. Another subset of vertices {*v*2, *v*3} consists of the two vertices *v*2 and *v*3 and therefore it is not a clique. This indicates that other six subsets of vertices {*v*1, *v*2}, {*v*1, *v*3}, {*v*1}, {*v*2}, {*v*3} and an empty subset of vertices that is {} are all a clique.

The number of vertex to clique {*v*1, *v*2} and clique {*v*1, *v*3} is both two. The number of vertex to clique {*v*1}, clique {*v*2} and clique {*v*3} is all one. The number of vertex to clique {} that is an empty subset of vertices is zero. The *maximal* clique problem that is a **NP-complete** problemis to find a *maximum*-*sized* clique in the graph with three vertices {*v*1, *v*2, *v*3} and two edges {(*v*1, *v*2), (*v*1, *v*3)} in Figure 3.7a. Therefore, clique {*v*1, *v*2} and clique {*v*1, *v*3} are two *maximum*-*sized* cliques to solve the *maximal* clique problem for the graph with three vertices {*v*1, *v*2, *v*3} and two edges {(*v*1, *v*2), (*v*1, *v*3)} in Figure 3.7a. Because the quantum program of implementing this example uses more quantum bits that exceed five quantum bits in the backend *ibmqx4* with five quantum bits in **IBM**’s quantum computers, we just use this example to explain what the maximal clique problem is. Next, we give **Definition 3-3** to describe the maximal clique problem.

**Definition 3-3**: Mathematically, a *clique* for a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges is defined as a set of vertices in which every vertex is connected to every other vertex by an edge. The maximal clique problem to a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges is to find a *maximum*-*sized* clique in the graph *G* = (*V*, *E*) with *n* vertices and *θ* edges.

From **Definition 3-3**, all of the possible solutions to the clique problem of graph *G* with *n* vertices and θ edges consist of 2*n* possible choices. Every possible choice corresponds to a subset of vertices (a possible clique in *G*). Hence, we assume that a set *X* contains 2*n* possible choices and the set *X* is equal to {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*}. This is to say that the length of each element in {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*} is *n* bits and every element stands for one of 2*n* possible choices.

For the sake of presentation, we suppose that *xd*0 denotes the fact that the value of *xd* is zero and *xd*1 denotes the fact that the value of *xd* is one. If an element *x*1 *x*2 … *xn* − 1 *xn* in {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*} is a clique and the value of *xd* for 1 ≤ *d* ≤ *n* is one, then *xd*1 represents that the *d*th vertex is within the clique. If an element *x*1 *x*2 … *xn* − 1 *xn* in {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*} is a clique and the value of *xd* for 1 ≤ *d* ≤ *n* is zero, then *xd*0 stands for that the *d*th vertex is not within the clique.

From {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*}, the first element *x*10 *x*20 … *xn* − 10 *xn*0 encodes an empty set of vertices that is {}. The second element *x*10 *x*20 … *xn* − 10 *xn*1 encodes a set of vertices {*vn*}. The third element *x*10 *x*20 … *xn* − 11 *xn*0 encodes a set of vertices {*vn* − 1} and so on with that the last element *x*11 *x*21 … *xn* − 11 *xn*1 encodes a set of vertices {*v*1 *v*2 … *vn* − 1 *vn*}. We regard {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*} as an unsorted database containing 2*n* possible choices (2*n* possible cliques) to the maximal clique problem of graph *G* with *n* vertices and θ edges.

**3.3.1 Flowchart of Recognize Cliques to the Maximal Clique Problem**

From Definition 3-3, solving the clique problem for a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges and its complementary graph = (*V*, ) with *n* vertices and *m* edges in which each edge is out of *E* is to find a subset *V1* of vertices with size *r* that satisfies *V1* to be a maximum-sized clique in *G*. This indicates that all of the possible solutions are 2*n*subset of vertices in which each subset of vertices corresponds to a possible clique. Any two vertices connected in the *complementary* graph = (*V*, ) with *n* vertices and *m* edges are disconnected in the *original* graph with *G* = (*V*, *E*) with *n* vertices and *θ* edges. This is to say that any two vertices connected in the *complementary* graph cannot be members of the same clique.

If any one of 2*n* possible choices (cliques) does not include any edge inthe *complementary* graph , then it is a clique in the original graph *G*. Otherwise, it is not a clique in the original graph *G*. Boolean variables *xi*and *xj*encode vertices *vi* and *vj*for 1 ≤ *i* ≤ *n* and 1 ≤ *j* ≤ *n*. Inthe *complementary* graph , the *k*th edge is = (*vi* , *vj* )to 1 ≤ *k* ≤ *m*. The requested condition of deciding whether any one of 2*n* subsets of vertices does not include the *k*th edge = (*vi* , *vj* ) or not is to satisfy a formula of the form that is the true value, where is one **NAND** gate. We regard a formula of the form as a clause. When the value of Boolean variable *xi* is 1 (one) and the value of Boolean variable *xj* is 1 (one), the output (result) of implementing is 0 (zero). This indicates that each possible choice containing the two vertices *vi* and *vj* is not a clique. When the value of Boolean variable *xi* is 1 (one) and the value of Boolean variable *xj* is 0 (zero), the output (result) of implementing is 1 (one). This is to say that each possible choice consisting of vertex *vi* and not containing vertex *vj* is perhaps a clique. When the value of Boolean variable *xi* is 0 (zero) and the value of Boolean variable *xj* is 1 (one), the output (result) of implementing is 1 (one). This implies that each possible choice consisting of vertex *vj* and not containing vertex *vi* is perhaps a clique. When the value of Boolean variable *xi* is 0 (zero) and the value of Boolean variable *xj* is 0 (zero), the output (result) of implementing is 1 (one). This indicates that each possible choice not containing vertex *vi* and vertex *vj* is perhaps a clique.

The requested condition of checking whether any one of 2*n* subsets of vertices does not include *m* edges in the complementary graph or not is to satisfy a formula of the form () that is the true value. We regard a formula of the form () as a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*, where each clause *Cj* for 1 ≤ *j* ≤ *m* is a formula of the form to Boolean variables *xi*and *xj*for 1 ≤ *i* ≤ *n* and 1 ≤ *j* ≤ *n*. Any one of 2*n* subsets of vertices is a clique if finding values of each Boolean variable satisfy the whole formula has the value 1 (one). This is the same as finding values of each Boolean variable that make each clause have the value 1 (one).

Recognizing clique(s) is equivalent to implement a formula of the form (). Therefore, we need to make use of auxiliary Boolean variables *rk* for 1 ≤ *k* ≤ *m* and auxiliary Boolean variables *sk* for 0 ≤ *k* ≤ *m*. We use ***CCNOT*** gates to implement the only ***NAND*** gate () in each clause and we apply auxiliary Boolean variables *rk* for 1 ≤ *k* ≤ *m* to store the result of implementing the only ***NAND*** gate () in each clause. This is to say that each auxiliary Boolean variable *rk* for 1 ≤ *k* ≤ *m* is actually the target bit of a ***CCNOT*** gate of implementing a ***NAND*** gate (). Hence, the initial value of each auxiliary Boolean variable *rk* for 1 ≤ *k* ≤ *m* is set to one (1).

We make use of an auxiliary Boolean variable *s*0 as the first operand of the first logical and operation (“∧”) in a Boolean formula of the form (). The initial value of the auxiliary Boolean variable *s*0 is set to one (1). This is to say that this setting does not change the correct result of the first logical and operation in (). We apply ***CCNOT*** gates to implement the logical and operations in () and we use auxiliary Boolean variables *sk* for 1 ≤ *k* ≤ *m* to store the result of implementing the logical and operations in (). This indicates that each auxiliary Boolean variable *sk* for 1 ≤ *k* ≤ *m* is actually the target bit of a ***CCNOT*** gate of implementing a logical and operation. Therefore, the initial value of each auxiliary Boolean variable *sk* for 1 ≤ *k* ≤ *m* is set to zero (0).

Figure 3.8 is to flowchart of recognizing cliques to the maximal clique problem for a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges and its complementary graph = (*V*, ) with *n* vertices and *m* edges. In Figure 3.8, in statement *S*1, it sets the index variable *k* of the first loop to one (1). Next, in statement *S*2, it executes the conditional judgement of the first loop. If the value of *k* is less than or equal to the value of *m*, then *next executed* instruction is statement *S*3. Otherwise, in statement *S*6, it executes an *End* instruction to terminate the task that is to find values of each Boolean variable so that the whole formula has the value 1 and this is the same as finding values of each Boolean variable that make each clause have the value 1.

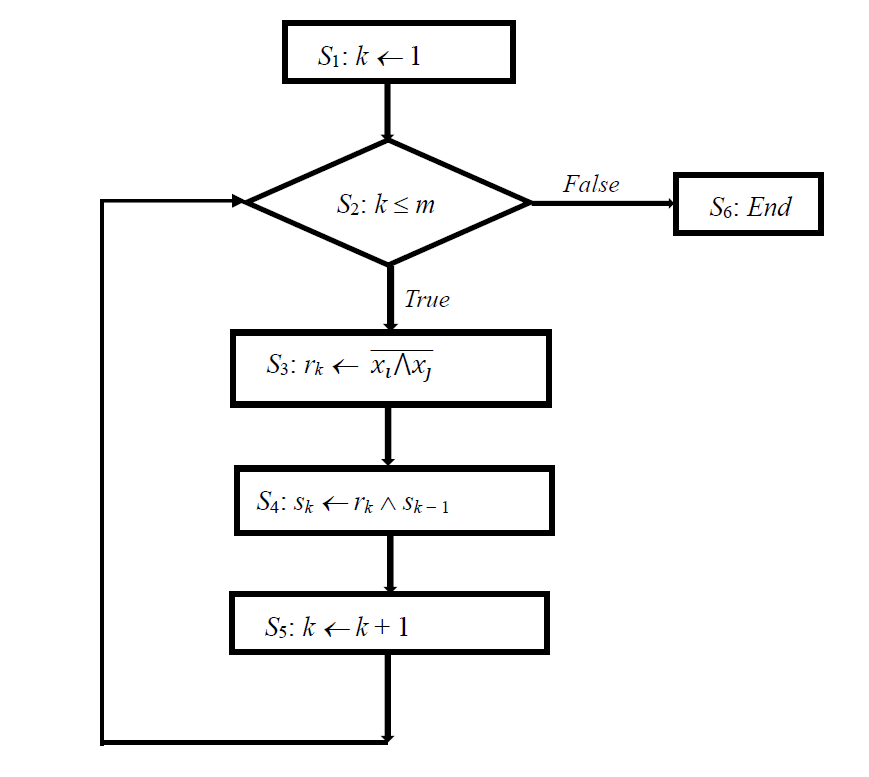


Figure 3.8: Flowchart of recognizing cliques to the maximal clique problem for a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges and its complementary graph = (*V*, ) with *n* vertices and *m* edges.

In statement *S*3, it implements a ***NAND*** gate “*rk ←* ”. Boolean variables *xi* and *xj* respectively encode vertex *vi* and vertex *vj* that are connected by the *k*th edge in the complementary graph = (*V*, ) with *n* vertices and *m* edges. Boolean variable *rk* stores the result of implementing () (the *k*th ***NAND*** gate). Next, in statement *S*4, it executes a logical and operation “*sk ← rk* ∧ *sk* − 1” that is the *kth* clause in (). Boolean variable *rk* stores the result of implementing the *k*th ***NAND*** gate and is the first operand of the logical and operation. Boolean variable *sk* − 1 is the second operand of the logical and operation and stores the result of the previous logical and operation. Next, in statement *S*5, it increases the value of the index variable *k* to the first loop. Repeat to execute statement *S*2 through statement *S*5 until in statement *S*2 the conditional judgement becomes a *false* value. From Figure 3.8, the total number of ***NAND*** gate is *m* ***NAND*** gates. The total number of logical and operation is *m* ***AND*** gates (logical and operations). Therefore, the cost of recognizing clique(s) is to implement *m* ***NAND*** gates and *m* ***AND*** gates.

**3.3.2 Flowchart of Computing the Number of Vertex in Each Clique to the Maximal Clique Problem**

For a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges and its complementary graph = (*V*, ) with *n* vertices and *m* edge, solution space of the maximal clique problem is 2*n* subsets of vertices. We use {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*} to encode 2*n* subsets of vertices. After each element encoding one subset of vertices in {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*} completes each operation in Figure 3.8, Boolean variable *sm* stores the result of deciding whether it is a clique or not. If the value of Boolean variable *sm* is equal to 1 (one), then it is a clique. Otherwise, it is not a clique.

For computing the number of vertex, we need auxiliary Boolean variables *zi*+1, *j* and *zi*+1, *j*+1 for 0 ≤ *i* ≤ *n* − 1 and 0 ≤ *j* ≤ *i*. Auxiliary Boolean variables *zi*+1, *j* and *zi*+1, *j*+1 for 0 ≤ *i* ≤ *n* − 1 and 0 ≤ *j* ≤ *i* are set to the initial value 0 (zero). Boolean variable *zi*+1, *j*+1 for 0 ≤ *i* ≤ *n* − 1 and 0 ≤ *j* ≤ *i* is to store the number of vertex in a clique after figuring out the influence of Boolean variable *xi* + 1 that encodes the (*i* + 1)th vertex to the number of ones (vertices). If the value of Boolean variable *zi*+1, *j*+1 for 0 ≤ *i* ≤ *n* − 1 and 0 ≤ *j* ≤ *i* is equal to 1 (one), then this indicates that there are (*j* + 1) ones (vertices) in the clique. Boolean variable *zi*+1, *j* for 0 ≤ *i* ≤ *n* − 1 and 0 ≤ *j* ≤ *i* is to store the number of vertex in a clique after figuring out the influence of Boolean variable *xi* + 1 that encodes the (*i* + 1)th vertex to the number of ones (vertices). If the value of Boolean variable *zi*+1, *j* for 0 ≤ *i* ≤ *n* − 1 and 0 ≤ *j* ≤ *i* is equal to 1 (one), then this is to say that there are *j* ones (vertices) in the clique.

In a clique, Boolean variable *x*1 encodes the *first* vertex *v*1. If the value of Boolean variable *x*1 is equal to 1 (one), then the first vertex *v*1 is within the clique and it increases the number of vertex to the clique. If the value of Boolean variable *x*1 is equal to 0 (zero), then the first vertex *v*1 is not within the clique and it reserves the number of vertex to the clique. Therefore, the influence of Boolean variable *x*1 to increase the number of vertex to a clique is to satisfy the formula (*sm* ∧ *x*1) that is the true value. Similarly, the influence of Boolean variable *x*1 to reserve the number of vertex to a clique is to satisfy the formula (*sm* ∧ ) that is the true value.

In a clique, Boolean variable *xi* + 1 encodes the (*i* + 1)th vertex *vi* + 1 for 1 ≤ *i* ≤ *n* – 1. If the value of Boolean variable *xi* + 1 is equal to 1 (one), then the (*i* + 1)th vertex *vi* + 1 is within the clique and it increases the number of vertex to the clique. If the value of Boolean variable *xi* + 1 is equal to 0 (zero), then the (*i* + 1)th vertex *vi* + 1 is not within the clique and it reserves the number of vertex to the clique. Therefore, the influence of Boolean variable *xi* + 1 to increase the number of vertex to a clique that has currently has *j* vertices is to satisfy the formula (*xi* + 1 ∧ *zi*, *j*) that is the true value. Similarly, the influence of Boolean variable *xi* + 1 to reserve the number of vertex to a clique that has currently has *j* vertices is to satisfy the formula ( ∧ *zi*, *j*) that is the true value.

Figure 3.9 is the logical flowchart of counting the number of vertex in each clique to the maximal clique problem for a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges and its complementary graph = (*V*, ) with *n* vertices and *m* edges. In Figure 3.9, in statement *S*1, it implements a logical and operation “*z*1,1 ← *sm* ∧ *x*1” that is one ***AND***  gate. Boolean variable *z*1, 1 stores the result of implementing one ***AND*** gate (*sm* ∧ *x*1). If the value of Boolean variable *z*1, 1 is equal to 1 (one), then it increases the number of vertex so that the number of vertex in each clique with the first vertex *v*1 is one. Next, in statement *S*2, it implements a logical and operation “*z*1,0 ← *sm* ∧ ” that is one ***AND*** gate (*sm* ∧ ). Boolean variable *z*1, 0 stores the result of implementing one ***AND*** gate (*sm* ∧ ). If the value of Boolean variable *z*1, 0 is equal to 1 (one), then it reserves the number of vertex so that the number of vertex in each clique *without* the first vertex *v*1 is zero.

Next, in statement *S*3, it sets the index variable *i* of the first loop to one. Next, in st-

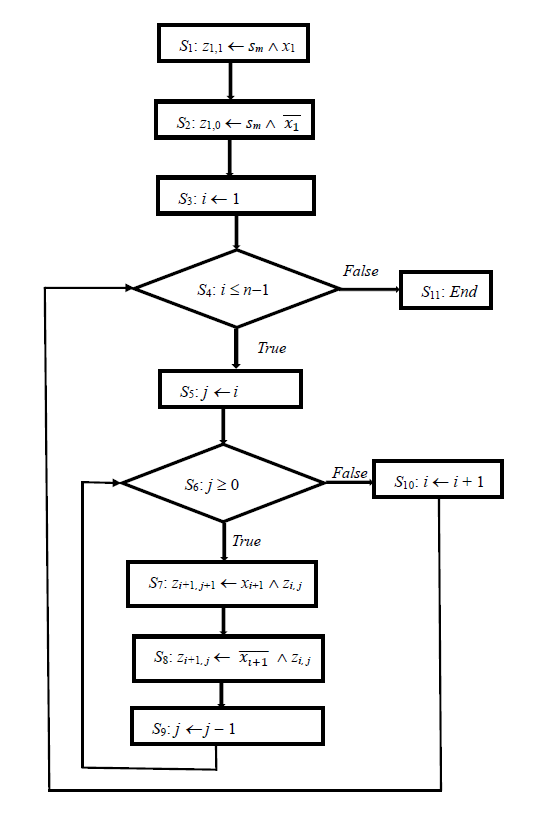


Figure 3.9: Flowchart of computing the number of vertex in each clique to the maximal clique problem for a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges and its complementary graph = (*V*, ) with *n* vertices and *m* edges.

atement *S*4, it executes the conditional judgement of the first loop. If the value of *i* is less than or equal to the value of (*n* −1), then *next executed* instruction is statement *S*5. Otherwise, in statement *S*11, it executes an *End* instruction to terminate the task that is to count the number of vertex in each clique. In statement *S*5, it sets the index variable *j* of the second loop to the value of the index variable *i* in the first loop. Next, in statement *S*6, it executes the conditional judgement of the *second* loop. If the value of *j* is greater than or equal to zero, then next executed instruction is statement *S*7. Otherwise, next executed instruction is statement *S*10.

In statement *S*7, it implements a logical and operation “*zi+*1*, j*+1← *xi*+1∧ *zi, j*” that is one ***AND*** gate. Boolean variable *xi*+1encodes the (*i* + 1)th vertex and is the first operand of the logical and operation. Boolean variable *zi, j* is the second operand of the logical and operation. Boolean variable *zi, j* stores the number of vertex in a clique after figuring out the influence of Boolean variable *xi* that encodes the *i*th vertex to the number of ones (vertices). If the value of Boolean variable *zi*, *j* is equal to 1 (one), then this indicates that there are *j* ones (vertices) in the clique. Boolean variable *zi+*1*, j*+1stores the result of implementing the logical and operation “*zi+*1*, j*+1← *xi*+1∧ *zi, j*”. This is to say that Boolean variable *zi*+1, *j*+1 stores the number of vertex in a clique after figuring out the influence of Boolean variable *xi* + 1 that encodes the (*i* + 1)th vertex to the number of ones (vertices). If the value of Boolean variable *zi*+1, *j*+1 is equal to 1 (one), then this implies that there are (*j* + 1) ones (vertices) in the clique.

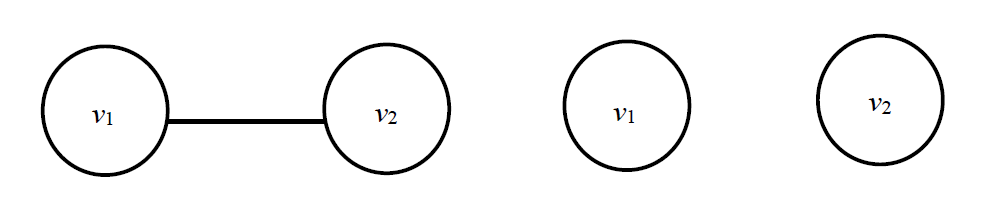
Next, in statement *S*8, it implements a logical and operation “*zi+*1*, j* ← ∧ *zi, j*” that is one ***AND*** gate. Boolean variable *xi*+1encodes the (*i* + 1)th vertex and its negation is the first operand of the logical and operation. Boolean variable *zi, j* is the second operand of the logical and operation. Boolean variable *zi, j* stores the number of vertex in a clique after figuring out the influence of Boolean variable *xi* that encodes the *i*th vertex to the number of ones (vertices). If the value of Boolean variable *zi*, *j* is equal to 1 (one), then this is to say that there are *j* ones (vertices) in the clique. Boolean variable *zi+*1*, j* stores the result of implementing the logical and operation “*zi+*1*, j* ← ∧ *zi, j*”. This indicates that Boolean variable *zi*+1, *j* stores the number of vertex in a clique after figuring out the influence of Boolean variable *xi* + 1 that encodes the (*i* + 1)th vertex to the number of ones (vertices). If the value of Boolean variable *zi*+1, *j* is equal to 1 (one), then this is to say that there are *j* ones (vertices) in the clique.

Next, in statement *S*9, it decreases the value of the index variable *j* in the second loop. Repeat to execute statement *S*6 through statement *S*9 until in statement *S*6 the conditional judgement becomes a *false* value. Next, in statement *S*10, it increases the value of the index variable *i* in the first loop. Repeat to execute statement *S*4 through statement *S*10 until in statement *S*4 the conditional judgement becomes a *false* value. When in statement *S*4 the conditional judgement becomes a *false* value, next executed statement is statement *S*11. In statement *S*11, it executes an *End* instruction to terminate the task that is to count the number of vertex in each clique. The cost of one time to complete each operation in Figure 3.9 is to implement (*n* × (*n* +1)) ***AND*** gates and () ***NOT*** gates. This is to say that the cost of counting the number of vertex to a clique is to implement (*n* × (*n* +1)) ***AND*** gates and () ***NOT*** gates. Therefore, for counting the number of vertex in all cliques, the cost is to implement (2*n* × *n* × (*n* +1)) ***AND*** gates and (2*n* × ) ***NOT*** gates.

**3.3.3 Data Dependence Analysis for the Maximal Clique Problem**

A data dependence arises from two statements that read or write the same memory. *Data dependence analysis* is to decide whether to *reorder* or *parallelize* statements is safe or not. In a maximal clique problem for a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges and its complementary graph = (*V*, ) with *n* vertices and *m* edges, it contains 2*n* subsets of vertices (2*n* possible choices). We use a set {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*} to encode 2*n* subsets of vertices. In the set {*x*1 *x*2 … *xn* − 1 *xn*| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*}, the first element *x*10 *x*20 … *xn* − 10*xn*0 encodes an empty subset without any vertex. The second element *x*10 *x*20 … *xn* − 10*xn*1 encodes {*vn*}. The third element *x*10 *x*20 … *xn* − 11*xn*0 encodes {*vn* − 1} and so on with that the last element *x*11 *x*21 … *xn* − 11*xn*1 encodes {*v*1 *v*2 … *vn* − 1 *vn*} that contains each vertex. Each element needs to implement those operations in Figure 3.8 and Figure 3.9. Each element needs to use *m* auxiliary Boolean variables *rk* for 1 ≤ *k* ≤ *m*, (*m* + 1) auxiliary Boolean variables *sk* for 0 ≤ *k* ≤ *m* and ((*n* × (*n* + 3)) / 2) auxiliary Boolean variables *zi*+1, *j* and *zi*+1, *j*+1 for 0 ≤ *i* ≤ *n* − 1 and 0 ≤ *j* ≤ *i*. Because 2*n* subsets of vertices (2*n* inputs) implement those instructions from Figure 3.8 through Figure 3.9 not to access or modify the same input and the same auxiliary Boolean variables, we can *parallelize* them without any error.

Let us consider another graph in Figure 3.10a and its complementary graph in Figure 3.10b. In Figure 3.10a, the graph has two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)}. In Figure 3.10b, its *complementary* graph has the same two vertices {*v*1, *v*2} and no edge missing in the *original* graph in Figure 3.10a. For the graph in Figure 3.10a with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} and its complementary graph in Figure 3.10b with the same two vertices {*v*1, *v*2} and no edge, solving the maximal clique problem is to find a subset *V1* of vertices with size *r* that satisfies *V1* to be a maximum-sized clique. This is to say that the value of *n* is equal to two, the value of *θ* is equal to one and the value of *m* is equal to zero.



1. (b)

Figure 3.10: (a) The graph has two vertices and one edge. (b) Its *complementary* graph has the same vertices and no edge missing in the *original* graph.

We regard the maximal clique problem for the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a as a search problem. Any given oracular function *Of* is to implement those instructions from Figure 3.8 to Figure 3.9 to recognize the maximal clique(s). Its domain is {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2} to encode 22 subsets of vertices. In the domain {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2}, the first element *x*10 *x*20 encodes an empty subset without vertex. The second element *x*10 *x*21 encodes {*v*2}. The third element *x*11 *x*20 encodes {*v*1}. The fourth element *x*11 *x*21 encodes {*v*1, *v*2}.

From the domain {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2}, there are four inputs *x*10 *x*20, *x*10 *x*21, *x*11 *x*20 and *x*11 *x*21. Because the value of *m* is equal to zero, there is no edge in its complementary graph in Figure 3.10b. Therefore, each input does not need to implement those instructions in Figure 3.8. Each input is a clique. This indicates that an empty subset {}, {*v*2}, {*v*1} and {*v*1, *v*2} are all cliques. Next, for computing the number of vertex in each clique, each input needs to implement “*z*1,1 ← *s*01 ∧ *x*1”, “*z*1,0 ← *s*01 ∧ ”, “*z*2*,* 2← *x*2∧ *z*1,1”, “*z*2*,*1← ∧ *z*1*,*1”, “*z*2*,* 1← *x*2∧ *z*1,0” and “*z*2*,*0← ∧ *z*1*,*0” in Figure 3.9. After each input completes six instructions above, the input *x*11 *x*21 has the result *z*2*,* 21 and other inputs *x*10 *x*20, *x*10 *x*21 and *x*11 *x*20 have the same result *z*2*,* 20. Because Boolean variable *z*2*,* 21 indicates that the input *x*11 *x*21 encodes {*v*1, *v*2} to be the maximal clique, the answer is to {*v*1, *v*2} and the number of vertex in the answer is two. Because 22 subsets of vertices implement six instructions above not to access or modify the same input and the same auxiliary Boolean variable, we can *parallelize* them without any error.

**3.3.4 Solution Space of Solving an Instance of the Maximal Clique Problem**

In the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a, an empty subset {}, {*v*2}, {*v*1} and {*v*1, *v*2} are all cliques. The maximal clique problem to the graph in Figure 3.10a is to find a maximum-sized clique in which the number of vertex is two. Implementing the instruction “*z*1,1 ← *s*01 ∧ *x*1” is equivalent to implement the instruction “*z*1,1 ← *x*1” in which Boolean variable *z*1,1 actually stores the value of *x*1. Therefore, implementing the instruction “*z*2*,* 2← *x*2∧ *z*1,1” is equivalent to implement the instruction “*z*2*,* 2← *x*2∧ *x*1”. So, any given oracular function *Of* to recognize a maximal-sized clique for the graph in Figure 3.10a is to implement the instruction “*z*2*,* 2← *x*2∧ *x*1”. Its domain is {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2} and its range is {0, 1}.

We regard its domain as its solution space in which there are four possible choices that satisfy *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 = 1. We make use of a basis {(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)} of the four-dimensional Hilbert space to construct solution space {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2}. We use (1, 0, 0, 0) to encode Boolean variable *x*10and Boolean variable *x*20 that represent an empty clique {} without any vertex. Next, we apply (0, 1, 0, 0) to encode Boolean variable *x*11and Boolean variable *x*20 that represent a clique {*v*1}. We make use of (0, 0, 1, 0) to encode Boolean variable *x*10and Boolean variable *x*21 that represent a clique {*v*2}. Finally, we apply (0, 0, 0, 1) to encode Boolean variable *x*11and Boolean variable *x*21 that represent the maximal clique {*v*1, *v*2}.

We use a linear combination of each element in the basis that is × (1, 0, 0, 0) + × (0, 1, 0, 0) + × (0, 0, 1, 0) + × (0, 0, 0, 1) = (, , , ) to construct solution space {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2}. The amplitude of each possible choice is all and the sum to the square of the absolute value of each amplitude is one. The length of the vector is one, so it is a unit vector. This indicates that we make use of a unit vector to encode all of the possible choices that satisfy *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2. We call the square of the absolute value of each amplitude as the cost (the successful probability) of that choice that satisfies the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2. The cost (the successful probability) of the answer(s) is close to one as soon as possible.

**3.3.5 Implementing Solution Space of Solving an Instance of the Maximal Clique Problem**

In Listing 3.2, the program in the backend *ibmqx4* with five quantum bits in **IBM**’s quantum computer is to solve an instance of the maximal clique problem to the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a. Because the given oracular function of recognizing the maximal clique(s) is *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2, in Listing 3.2, we introduce how to write a quantum program to find values of each Boolean variable so that the whole formula has the value 1. Figure 3.11 is the quantum circuit of constructing solution space to the question. The statement “OPENQASM 2.0;”

|  |
| --- |
| 1. OPENQASM 2.0; 2. include "qelib1.inc"; 3. qreg q[5]; 4. creg c[5]; 5. x q[0]; 6. h q[3]; 7. h q[4]; 8. h q[0]; |

Listing 3.2: The program of solving an instance of the maximal clique problem to the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a.

on line one of Listing 3.2 is to point out that the program is written with version 2.0 of Open QASM. Next, the statement “include "qelib1.inc";” on line two of Listing 3.2 is to continue parsing the file “qelib1.inc” as if the contents of the file were pasted at the location of the include statement, where the file “qelib1.inc” is **Quantum Experience (QE) Standard Header** and the path is specified relative to the current working directory.

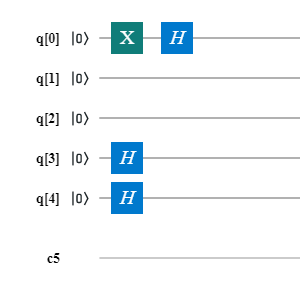


Figure 3.11: The quantum circuit of constructing solution space to an instance of the maximal clique problem to the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a.

Next, the statement “qreg q[5];” on line three of Listing 3.2 is to declare that in the program there are five quantum bits. In the left top of Figure 3.11, five quantum bits are respectively q[0], q[1], q[2], q[3] and q[4]. The initial value of each quantum bit is set to |0>. We apply quantum bit q[3] to encode Boolean variable *x*1. We use quantum bit q[4] to encode Boolean variable *x*2. We make use of quantum bit q[2] to encode auxiliary Boolean variable *z*2, 2. We apply quantum bit q[0] as an auxiliary working bit. We do not use quantum bit q[1].

For the convenience of our explanation, q[k]0 for 0 ≤ *k* ≤ 4 is to represent the value 0 of q[k] and q[k]1 for 0 ≤ *k* ≤ 4 is to represent the value 1 of q[k]. Similarly, for the convenience of our explanation, an initial state vector of constructing solution space to an instance of the maximal clique problem in the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a is as follows:

|α0> = |q[4]0> |q[3]0> |q[2]0> |q[1]0> |q[0]0> = |0> |0> |0> |0> |0> = |00000>.

Next, the statement “creg c[5];” on line four of Listing 3.2 is to declare that there are five classical bits in the program. In the left bottom of Figure 3.11, five classical bits are subsequently c[0], c[1], c[2], c[3] and c[4]. The initial value of each classical bit is set to 0.

Next, the three statements “x q[0];”, “h q[3];” and “h q[4];” on line five through seven of Listing 3.2 is to complete one ***X*** gate (one ***NOT*** gate) and two Hadamard gates of the *first* time slot of the quantum circuit in Figure 3.11. The statement “x q[0];” actually implements × = = (|1>). This is to say that the statement “x q[0];” on line five of Listing 3.2 inverts |q[0]0> (|0>) into |q[0]1> (|1>). The two statements “h q[3];” and “h q[4];” both actually run × = = = ( + ) = (|0> + |1>). This implies that converting q[3] from one state |0> to another state (|0> + |1>) (its superposition) and converting q[4] from one state |0> to another state (|0> + |1>) (its superposition) are implemented. Thus, the superposition of the two quantum bits q[4] and q[3] is ( (|0> + |1>)) ( (|0> + |1>)) = (|0> |0> + |0> |1> + |1> |0> + |1> |1>) = (|00> + |01> + |10> + |11>). Since in the *first* time slot of the quantum circuit in Figure 3.11 there is no quantum gate to act on quantum bits q[2] and q[1], their current states |q[2]0> and |q[1]0> are not changed. This indicates that we obtain the following new state vector

|α1> = ( (|q[4]0> + |q[4]1>)) ( (|q[3]0> + |q[3]1)) (|q[2]0> |q[1]0> |q[0]1>)

= (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + |q[4]1> |q[3]1>) (|q[2]0>

|q[1]0> |q[0]1>)

= (|0> |0> + |0> |1> + |1> |0> + |1> |1>) (|0> |0>|1>).

Then, the statement “h q[0];” on line *eight* of Listing 3.2 is to execute one Hadamard gate of the *second* time slot of the quantum circuit in Figure 3.11. The statement “h q[0];” actually implements × = = = ( − ) = (|0> − |1>). This implies that converting q[0] from one state |1> to another state (|0> − |1>) (its superposition) is performed. Because in the *second* time slot of the quantum circuit in Figure 3.11 there is no quantum gate to act on quantum bits q[4] through q[1], their current states are not changed. This is to say that we obtain the following new state vector

|α2> = ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + |q[4]1> |q[3]1>)) (|q[2]0>

|q[1]0>) (|q[0]0> − |q[0]1>))

= ( (|0> |0> + |0> |1> + |1> |0> + |1> |1>)) (|0> |0>) (|0> − |1>)).

In the new state vector |α2>, state |q[4]0> |q[3]0> encodes Boolean variable *x*10 and Boolean variable *x*20 that represent a possible choice without any vertex. State |q[4]0> |q[3]1> encodes Boolean variable *x*11 and Boolean variable *x*20 that represent a possible choice with the *first* vertex *v*1. State |q[4]1> |q[3]0> encodes Boolean variable *x*10 and Boolean variable *x*21 that represent a possible choice with the *second* vertex *v*2. State |q[4]1> |q[3]1> encodes Boolean variable *x*11 and Boolean variable *x*21 that represent a possible choice with the two vertices *v*1 and *v*2. The amplitude of each possible choice is and the cost (the successful possibility) of becoming the answer(s) to each possible choice is the same and is equal to = 1/4.

**3.3.6 The Oracle to an Instance of the Maximal Clique Problem**

For solving the maximal clique problem of the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a, the Oracle is to have the ability to *recognize* the maximal-sized clique(s). The Oracle is to multiply the probability amplitude of the maximal-sized clique(s) by −1 and leaves any other amplitude unchanged. Since the given oracular function of recognizing the maximal-sized clique(s) is *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2, the Oracle to solve an instance of the maximal clique problem is a (22 × 22) matrix *B* that is equal to .

We suppose that a (22 × 22) matrix *B*+ is the conjugate transpose of matrix *B*. The transpose of matrix *B* is equal to *itself* (matrix *B*) and each element in the transpose of matrix *B* is a real, so the conjugate transpose of matrix *B* is also equal to itself (matrix *B*). Therefore, we obtain *B*+ = *B*. Because matrix *B* and its conjugate transpose *B*+ are almost a (22 × 22) identity matrix, *B* × *B*+ = , and *B*+ × *B* = . Hence, we get *B* × *B*+ = *B*+ × *B*. This indicates that it is a unitary matrix (operator) to solve an instance of the maximal clique problem in the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} with the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2. Implementing the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 that recognizes solution(s) in an instance of the maximal clique problem is equivalent to implement the Oracle that is × = = × + × + × + × = |00> + |01> + |10> + |11>.

Four computational basis vectors , , and encode four states |00> (|q[4]0> |q[3]0>), |01> (|q[4]0> |q[3]1>), |10> (|q[4]1> |q[3]0>) and |11> (|q[4]1> |q[3]1>) and their current amplitudes are respectively , , and ). State |00> (|q[4]0> |q[3]0>) with the amplitude () encodes Boolean variable *x*10 and Boolean variable *x*20 that represent a possible choice without any vertex. State |01> (|q[4]0> |q[3]1>) with the amplitude () encodes Boolean variable *x*11 and Boolean variable *x*20 that represent a possible choice with the first vertex *v*1. State |10> (|q[4]1> |q[3]0>) with the amplitude () encodes Boolean variable *x*10 and Boolean variable *x*21 that represent a possible choice with the second vertex *v*2. State |11> (|q[4]1> |q[3]1>) with the amplitude () encodes Boolean variable *x*11 and Boolean variable *x*21 that represent the maximal-sized clique with the two vertices *v*1 and *v*2. This indicates that the Oracle multiplies the probability amplitude of the maximal-sized clique with the two vertices *v*1 and *v*2 by −1 and leaves any other amplitude unchanged.

**3.3.7 Implementing the Oracle to an Instance of the Maximal Clique Problem**

We apply one ***CCNOT*** gate to run the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 that recognizes the maximal-sized clique(s) to the maximal clique problem in the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a. We make use of quantum bit q[3] to encode Boolean variable *x*1, we use quantum bit q[4] to encode Boolean variable *x*2 and we apply quantum bit q[2] to encode Boolean variable *z*2, 2. Therefore, quantum bits q[3], q[4], q[2] are subsequently the first control bit, the second control bit and the target bit of the ***CCNOT*** gate. We make use of the ***CCNOT*** gate to implement a logical and operation, so the initial value to quantum bit q[2] is set to |0>.

From line *nine* through line *twenty-three* in Listing 3.2, there are the fifteen statements. They are subsequently “h q[2];”, “cx q[4],q[2];”, “tdg q[2];”, “cx q[3],q[2];”,

|  |
| --- |
| **Listing 3.2 continued…**  // We use the following *fifteen* statements to implement a ***CCNOT*** gate.   1. h q[2]; 2. cx q[4],q[2]; 3. tdg q[2]; 4. cx q[3],q[2]; 5. t q[2]; 6. cx q[4],q[2]; 7. tdg q[2]; 8. cx q[3],q[2]; 9. t q[4]; 10. t q[2]; 11. cx q[3],q[4]; 12. h q[2]; 13. t q[3]; 14. tdg q[4]; 15. cx q[3], q[4]; |

“t q[2];”, “cx q[4],q[2];”, “tdg q[2];”, “cx q[3],q[2];”, “t q[4];”, “t q[2];”, “cx q[3],q[4];”, “h q[2];”, “t q[3];”, “tdg q[4];” and “cx q[3], q[4];”. They complete the ***CCNOT*** gate that implements the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 that recognizes the maximal-sized clique(s) to the maximal clique problem in the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a. Figure 3.12 is the quantum circuit of implementing the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 that recognizes the maximal-sized clique(s) to the maximal clique problem in the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a.

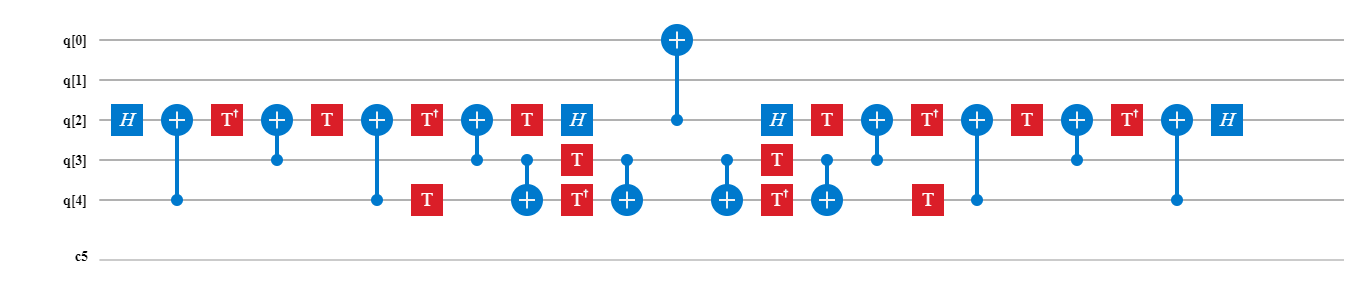


Figure 3.12: The quantum circuit of implementing the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 that recognizes the maximal-sized clique(s) to the maximal clique problem in the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a.

They take the state vector |α2> = ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + |q[4]1> |q[3]1>)) (|q[2]0> |q[1]0>) (|q[0]0> − |q[0]1>)) as their input. After they actually perform six ***CNOT*** gates, two Hadamard gates, three ***T***+ gates and four ***T*** gates from the *first* time slot through the *eleventh* time slot in Figure 3.12, we gain the following new state vector

|α3> = ( (|q[4]0> |q[3]0> |q[2]0> + |q[4]0> |q[3]1> |q[2]0> + |q[4]1> |q[3]0> |q[2]0> +

|q[4]1> |q[3]1> |q[2]1>)) (|q[1]0>) (|q[0]0> − |q[0]1>))

= ( (|0> |0> |0> + |0> |1> |0> + |1> |0> |0> + |1> |1> |1>)) (|0>) (|0> − |1>)).

Then, from line *twenty-four* in Listing 3.2, the statement “cx q[2],q[0];” takes the new state vector |α3> as its input. It multiplies the probability amplitude of the answer

|  |
| --- |
| **Listing 3.2 continued…**  // The Oracle multiplies the probability amplitude of the maximal-sized clique {*v*1, // *v*2} by −1 and leaves any other amplitude unchanged.   1. cx q[2],q[0]; |

|q[4]1> |q[3]1> encoding the maximal-sized clique {*v*1, *v*2} by −1 and leaves any other amplitude unchanged. This implies that after the statement “cx q[2],q[0];” completes the ***CNOT*** gate in the *twelfth* time slot in Figure 3.12, we get the following new state vector

|α4> = ( (|q[4]0> |q[3]0> |q[2]0> + |q[4]0> |q[3]1> |q[2]0> + |q[4]1> |q[3]0> |q[2]0> +

(−1) |q[4]1> |q[3]1> |q[2]1>)) (|q[1]0>) (|q[0]0> − |q[0]1>))

= ( (|0> |0> |0> + |0> |1> |0> + |1> |0> |0> + (−1) |1> |1> |1>)) (|0>) (|0> −

|1>)).

Since quantum operations are reversible by nature, executing the reversed order of implementing the ***CCNOT*** gate can restore the auxiliary quantum bits to their initial states. From line *twenty-five* through line *thirty-nine* in Listing 3.2, there are the *fifteen* statements. They are “cx q[3],q[4];”, “tdg q[4];”, “t q[3];”, “h q[2];”, “cx q[3],q[4];”, “t q[2];”, “t q[4];”, “cx q[3],q[2];”, “tdg q[2];”, “cx q[4],q[2];”, “t q[2];”, “cx q[3],q[2];”, “tdg q[2];”, “cx q[4],q[2];” and “h q[2];”. They run the reversed order of implementing

|  |
| --- |
| **Listing 3.2 continued…**  // Because quantum operations are reversible by nature, executing the reversed  // order of implementing the ***CCNOT*** gate can restore the auxiliary quantum bits  // to their initial states.   1. cx q[3],q[4]; 2. tdg q[4]; 3. t q[3]; 4. h q[2]; 5. cx q[3],q[4]; 6. t q[2]; 7. t q[4]; 8. cx q[3],q[2]; 9. tdg q[2]; 10. cx q[4],q[2]; 11. t q[2]; 12. cx q[3],q[2]; 13. tdg q[2]; 14. cx q[4],q[2]; 15. h q[2]; |

the ***CCNOT*** gate that performs the given oracular function *Of* = *F*(*x*1, *x*2) = *x*1 ∧ *x*2 that recognizes the maximal-sized clique(s). They take the new state vector |α4> as their input. After they actually complete six ***CNOT*** gates, two Hadamard gates, three ***T***+ gates and four ***T*** gates from the *thirteenth* time slot through the *last* time slot in Figure 3.12, we obtain the following new state vector

|α5> = ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1> |q[3]1>))

(|q[2]0> |q[1]0>) (|q[0]0> − |q[0]1>))

= ( (|0> |0> + |0> |1> + |1> |0> + (−1) |1> |1>)) (|0> |0>) (|0> − |1>)).

In the state vector |α2>, the amplitude of each element in solution space {*x*1 *x*2| ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ 2} is (1/2). In the state vector |α5>, the amplitude to three elements *x*10 *x*20 encoding an empty clique, *x*10 *x*21 encoding a clique {*v*2}, *x*11 *x*20 encoding a clique {*v*1} in solution space is all (1/2) and the amplitude to the element *x*11 *x*21 encoding the maximal-sized clique {*v*1, *v*2} in solution space is (−1/2). This is to say that *thirty-one* statements from line *nine* through *thirty-nine* in Listing 3.2 complete × that is to implement the Oracle that recognizes the maximal-sized clique(s) to solve the maximal clique problem in the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a.

**3.3.8 Implementing the Grover Diffusion Operator to Amplify the Amplitude of the Solution in an Instance of the Maximal Clique Problem**

The new state vector |α5> is ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1> |q[3]1>)) (|q[2]0> |q[1]0>) (|q[0]0> − |q[0]1>)). It includes two independentsubsystem. The first subsystem is ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1> |q[3]1>)) and the second subsystem is (|q[2]0> |q[1]0>) (|q[0]0> − |q[0]1>)). Amplifying the amplitude of each solution in the maximal clique problem in the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a only needs to consider the first subsystem in the new state vector |α5>. Since for an instance of the maximal clique problem the (22 × 1) vector encodes the first subsystem of the new state vector |α5> and is a (22 × 22) diffusion operator, amplifying the amplitude of the solution is to implement × = . This indicates that the amplitude of the solution that encodes the maximal-sized clique {*v*1, *v*2} is one and the amplitude of other three possible choices is all zero.

The quantum circuit in Figure 3.13 implements the Grover diffusion operator, (2 − ) . From line *forty* through line *fifty-one* in Listing 3.2, there are the *twelve* statements. They are subsequently “h q[3];”, “h q[4];”, “x q[3];”, “x q[4];”, “h q[4];”, “cx q[3],q[4];”, “h q[4];”, “x q[4];”, “x q[3];”, “u3(2\*pi,0\*pi,0\*pi) q[3];”, “h q[4];” and “h q[3];”.

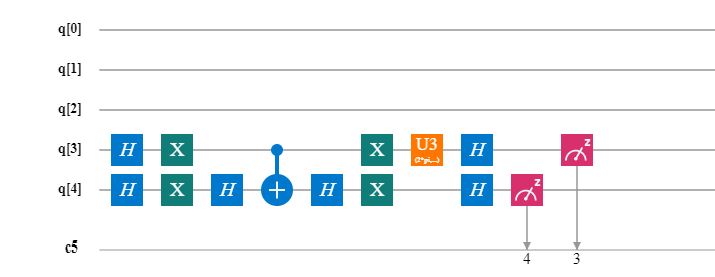


Figure 3.13: The quantum circuit of implementing the Grover diffusion operator, (2 − ) , to an instance of the maximal clique problem in the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a.

|  |
| --- |
| **Listing 3.2 continued…**  //We complete the amplitude amplification of the answer.  40 h q[3];  41 h q[4];  // We complete phase shifters  42 x q[3];  43 x q[4];  44 h q[4];  45 cx q[3],q[4];  46 h q[4];  47 x q[4];  48 x q[3];  49 u3(2\*pi,0\*pi,0\*pi) q[3];  50 h q[4];  51 h q[3]; |

They take the new state vector |α5> ( (|q[4]0> |q[3]0> + |q[4]0> |q[3]1> + |q[4]1> |q[3]0> + (−1) |q[4]1> |q[3]1>)) as their input. They complete the diffusion operator, (2 − ) from the first time slot through the eighth time slot in Figure 3.13. This is to say that we obtain the following new state vector

|α6> = |q[4]1>|q[3]1>.

Next, from line fifty-two in Listing 3.2 the statement “measure q[4] -> c[4];” is to measure the *fifth* quantum bit q[4] and to record the measurement outcome by overwriting the *fifth* classical bit c[4]. From line fifty-three in Listing 3.2 the statement “measure q[3] -> c[3];” is to measure the *fourth* quantum bit q[3] and to record the measurement outcome by overwriting the *fourth* classical bit c[3]. They implement the measurement from the ninth time slot through the tenth time slot of Figure 3.13.

|  |
| --- |
| **Listing 3.2 continued…**  // We complete the measurement of the answer.   1. measure q[4] -> c[4]; 2. measure q[3] -> c[3]; |

In the backend *ibmqx4* with five quantum bits in **IBM**’s quantum computers, we apply the command “simulate” to execute the program in Listing 3.2. The measured result appears in Figure 3.14. From Figure 3.14, we get the answer 11000 (c[4] = q[4] = |1>, c[3] = q[3] = |1>, c[2] = q[2] = |0>, c[1] = q[1] = |0> and c[0] = q[0] = |0>) with the probability 1 (100%). This implies that with the possibility 1 (100%) we gain that the value of quantum bit q[3] is equal to |1> and the value of quantum bit q[4] is equal to |1>. Therefore, the maximal-sized clique is to {*v*1, *v*2}.

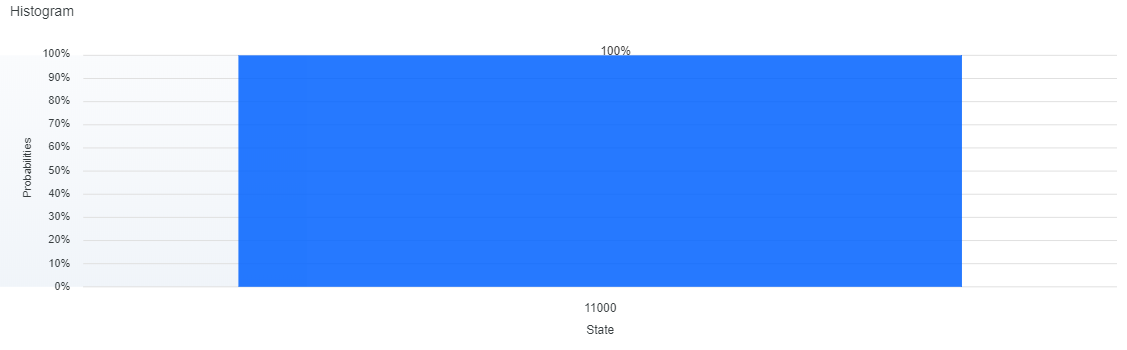


Figure 3.14: After the measurement to solve an instance of the maximal clique problem in the graph with two vertices {*v*1, *v*2} and one edge {(*v*1, *v*2)} in Figure 3.10a is completed, we obtain the answer 11000 with the probability 1 (100%).

**3.3.9 The Quantum Search Algorithm to the Maximal Clique Problem**

A maximal clique problem to a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges and its *complementary* graph = (*V*, ) with *n* vertices and *m* edges in which each edge in is out of *E* is to find the maximal-sized clique(s) among 2*n* subsets of vertices. Any given oracular function *Of*(*x*1 *x*2 … *xn* − 1 *xn*) is to recognize the maximal-sized clique(s) among 2*n* subsets of vertices. It implements flowchart of recognizing cliques in Figure 3.8. Next, it implements flowchart of computing the number of vertex in each clique in Figure 3.9. We make use of the quantum search algorithm to find one of *M* solutions among 2*n* subsets of vertices, where 0 ≤ *M* ≤ 2*n*.

The quantum circuit in Figure 3.15 is to complete the quantum search algorithm to

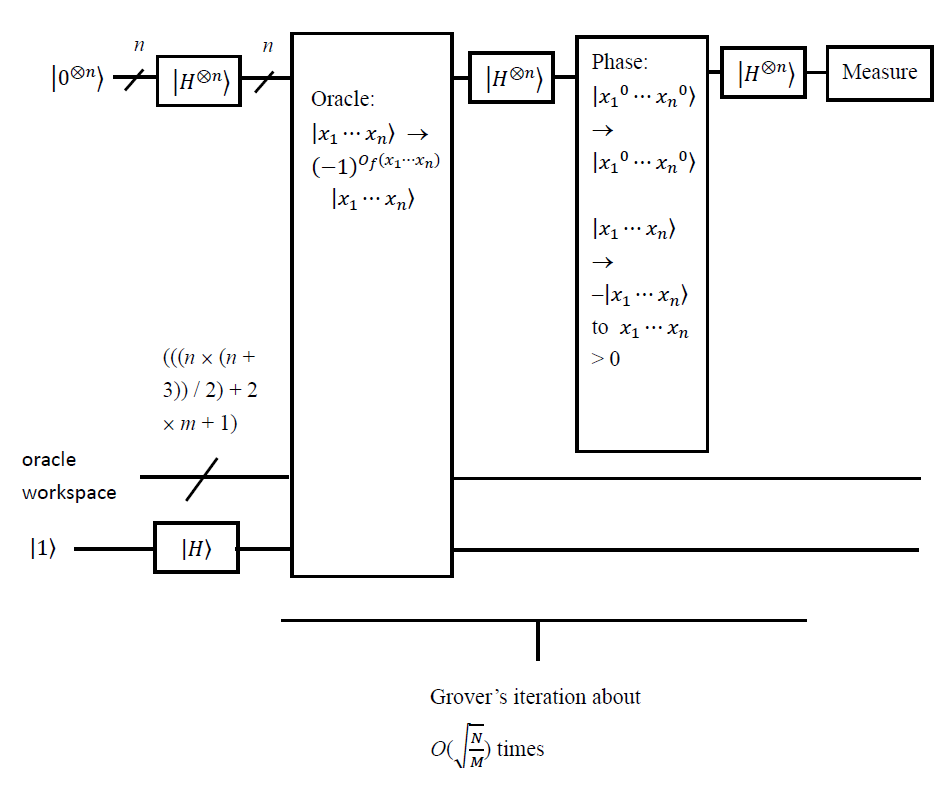


Figure 3.15: Circuit of implementing the quantum search algorithm to solve an instance of the maximal problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges.

solve an instance of the maximal clique problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges. The first quantum register in the left top of Figure 3.15 is (). This implies that the initial value of each quantum bit is |0>. The second quantum register in the left bottom of Figure 3.15 has (((*n* × (*n* + 3)) / 2) + 2 × *m* + 1) quantum bits and is an auxiliary quantum register. The initial value of each quantum bit in the second quantum register is |0> or |1> that is dependent on implementing ***NAND*** operations or ***AND*** operations. The third quantum register in the left bottom of Figure 3.15 is ().

**3.3.10 The First Stage of the Quantum Search Algorithm to the Maximal Clique Problem**

In Figure 3.15, the first stage of the quantum search problem to solve an instance of the maximal clique problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges is to apply *n* Hadamardgates () to operate the first quantum register (). This is to say that it produces the superposition of *n* quantum bits that is () = (). The superposition () encodes solution space {*x*1 *x*2 … *xn* − 1 *xn* | ∀ *xd* ∈ {0, 1} for 1 ≤ *d* ≤ *n*} to solve an instance of the maximal clique problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges. This indicates that state () with the amplitude () encodes an empty subset (choice) of vertex, state () with the amplitude () encodes {*vn*} and so on with that state () with the amplitude () encodes {*v*1 *v*2 … *vn*}. In the first stage of the quantum search algorithm, it uses another Hadamard gate to operate the third quantum register (). This implies that it yields the superposition of the third quantum register () that is ().

**3.3.11 The Second Stage of the Quantum Search Algorithm to the Maximal Clique Problem**

In Figure 3.15, the *second* stage of the quantum search algorithm to solve an instance of the maximal clique problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges and its *complementary* graph = (*V*, ) with *n* vertices and *m* edges in which each edge in is out of *E* is to implement the Oracle. The Oracle is to have the ability to *recognize* solutions that are the maximal-sized cliques among 2*n*subsets (possible choices) of vertices. The Oracle is to multiply the probability amplitude of the maximal-sized clique(s) by −1 and leaves any other amplitude unchanged.

The *first* main task of the Oracle is to that recognizing clique(s) among 2*n*possible choices (subsets) of vertices is equivalent to implement a formula of the form (). This is to say that for completing the *first* main task of the Oracle we need to use (2 × *m* + 1) auxiliary quantum bits and to complete *m* ***NAND*** operations and *m* ***AND*** operations that is equivalent to implement (2 × *m*) ***CCNOT*** gates.

Next, the *second* main task of the Oracle is to that computing the number of vertex in each clique is equivalent to decide that the influence of each vertex is to increase the number of vertex to each clique or to reserve the number of vertex to each clique. This indicates that for performing the *second* main task of the Oracle we need to use ((*n* × (*n* + 3)) / 2) auxiliary quantum bits and to implement (*n* × (*n* +1)) ***AND*** operations and () ***NOT*** operations that is equivalent to complete (*n* × (*n* +1)) ***CCNOT*** gates and () ***NOT*** gates.

In oracle workspace in the second stage of the quantum search algorithm, we make use of auxiliary quantum bits |*rk*> for 1 ≤ *k* ≤ *m* to encode auxiliary Boolean variables *rk* for 1 ≤ *k* ≤ *m*. We apply auxiliary quantum bits |*sk*> for 0 ≤ *k* ≤ *m* to encode auxiliary Boolean variables *sk* for 0 ≤ *k* ≤ *m*. We use auxiliary quantum bits |*zi*+1, *j*> and |*zi*+1, *j*+1> for 0 ≤ *i* ≤ *n* − 1 and 0 ≤ *j* ≤ *i* to encode auxiliary Boolean variables *zi*+1, *j* and *zi*+1, *j*+1 for 0 ≤ *i* ≤ *n* − 1 and 0 ≤ *j* ≤ *i*.

We use ***CCNOT*** gates to implement each ***NAND*** gate () in a formula of the form () and we apply auxiliary quantum bits |*rk*> for 1 ≤ *k* ≤ *m* to store the result of implementing each ***NAND*** gate (). This is to say that each auxiliary quantum bit |*rk*> for 1 ≤ *k* ≤ *m* is actually the target bit of a ***CCNOT*** gate of implementing a ***NAND*** gate (). Therefore, the initial value of each auxiliary quantum bit |*rk*> for 1 ≤ *k* ≤ *m* is set to one |1>.

We use an auxiliary quantum bit |*s*0> as the first operand of the first logical and operation (“∧”) in a formula of the form (). The initial value of the auxiliary quantum bit |*s*0> is set to |1>. This indicates that this setting does not change the correct result of the first logical and operation. We make use of a ***CCNOT*** gate to implement each logical and operation in a formula of the form (). We apply auxiliary quantum bits |*sk*> for 1 ≤ *k* ≤ *m* to store the result of implementing each logical and operation. This implies that each auxiliary quantum bit |*sk*> for 1 ≤ *k* ≤ *m* is actually the target bit of a ***CCNOT*** gate of implementing a logical and operation. Thus, the initial value of each auxiliary quantum bit |*sk*> for 1 ≤ *k* ≤ *m* is set to |0>.

We use a ***CCNOT*** gate to implement (*sm* ∧ *x*1) that is to compute the influence of the first vertex to increase the number of vertex in each clique. We apply a ***CCNOT*** gate and two ***NOT*** gates to implement (*sm* ∧ ) that is to compute the influence of the first vertex to reserve the number of vertex in each clique. We make use of auxiliary quantum bits |*z*1, 1> to store the result of implementing (*sm* ∧ *x*1) and apply auxiliary quantum bits |*z*1, 0> to store the result of implementing (*sm* ∧ ). This is to say that the two auxiliary quantum bits |*z*1, 1> and |*z*1, 0> are actually the target bits of two ***CCNOT*** gates of implementing two logical and operations. Therefore, the initial value of the two auxiliary quantum bits |*z*1, 1> and |*z*1, 0> is set to |0>.

We use a ***CCNOT*** gate to implement (*xi* + 1 ∧ *zi*, *j*) that is to figure the influence of the (*i* + 1)th vertex *vi* + 1 for 1 ≤ *i* ≤ *n* – 1 to increase the number of vertex in each clique. We apply a ***CCNOT*** gate and two ***NOT*** gates to implement ( ∧ *zi*, *j*) that is to deal with the influence of the (*i* + 1)th vertex *vi* + 1 for 1 ≤ *i* ≤ *n* – 1 to reserve the number of vertex in each clique. We make use of auxiliary quantum bits |*zi*+1, *j*+1> and |*zi*+1, *j*> for 0 ≤ *i* ≤ *n* − 1 and 0 ≤ *j* ≤ *i* to store the result of implementing them. This is to say that auxiliary quantum bit |*zi*+1, *j*+1> and |*zi*+1, *j*> for 0 ≤ *i* ≤ *n* − 1 and 0 ≤ *j* ≤ *i* are the target bits of the corresponding ***CCNOT*** gates of implementing logical and operations. Hence, the initial value of auxiliary quantum bit |*zi*+1, *j*+1> and |*zi*+1, *j*> for 0 ≤ *i* ≤ *n* − 1 and 0 ≤ *j* ≤ *i* is set to |0>. Quantum bit |*zn*, *j*> for *n* ≥ *j* ≥ 0 is to store the result that has *j* ones after computing the influence of *n* vertices to the number of vertex in each clique. If the value of quantum bit |*zn*, *j*> for *n* ≥ *j* ≥ 0 is equal to |1>, then it has *j* vertices.

We use one ***CNOT*** gate to multiply the probability amplitude of the maximal-sized clique(s) by −1 and to leave any other amplitude unchanged, where quantum bit () is the target bit of the ***CNOT*** gate and quantum bit (|>) is the control bit of the ***CNOT*** gate. When the value of the control bit (|>) is equal to (|1>), the target bit becomes () = (−1) (). This is to multiply the probability amplitude of the answer(s) by −1. When the value of the control bit (|>) is equal to (|0>), the target bit still is (). This is to leave any other amplitude unchanged.

Quantum operations are reversible by nature, so executing the reversed order of implementing the two main tasks of the Oracle can restore the auxiliary quantum bits to their initial states. This indicates that the second stage of the quantum search algorithm to solve an instance of the maximal clique problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges converts () into (). The cost of completing the Oracle in the second stage of the quantum search algorithm in Figure 3.15 is to implement (2 × ((2 × *m*) + (*n* × (*n* + 1)))) ***CCNOT*** gates, (*n* × (*n* + 1)) ***NOT*** gates and one ***CNOT*** gate.

Because the number of the vertices to the maximal clique(s) is from *n* through zero, we first check whether there is/are clique(s) with *n* vertices. The condition is to implement one ***CNOT*** gate to multiply the probability amplitude of the answer(s) with *n* vertices by −1 and to leave any other amplitude unchanged. If the answer(s) with *n* vertices are found, then we terminate the execution of the program. Otherwise, we need to continue to test whether there is/are clique(s) with (*n* − 1) vertices, (*n* − 2) vertices and so on with that has one vertex until the answer(s) are found.

**3.3.12 The Third Stage of the Quantum Search Algorithm to the Maximal Clique Problem**

In Figure 3.15, the *third* stage of the quantum search algorithm to solve an instance of the maximal clique problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges and its *complementary* graph = (*V*, ) with *n* vertices and *m* edges in which each edge in is out of *E* is to perform the Grover diffusion operator that is a (2*n* × 2*n*) matrix *D* in which *Da*, *b* = if *a ≠ b* and *Da*, *a* = − 1.Executing the Grover diffusion operator is equivalent to implement (2 − ) . A phase shifter operator, (2 − ) negates all the states except for (). In Figure 3.15, the *third* stage of the quantum search problem is to apply that the phase shift operator, 2 − , negates all the states except for () sandwiched between gates. This is to say that the *third* stage of the quantum search algorithm to solve an instance of the maximal clique problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges is to increase the amplitude of the maximal-sized clique(s) and to decrease the amplitude of the non-answer(s).

For solving an instance of the maximal clique problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges, we regard the Oracle and the Grover diffusion operator from the second stage through the third stage of the quantum search algorithm in Figure 3.15 as a subroutine. We call the subroutine as the *Grover iteration*. Though the number of the vertices to the maximal clique(s) is from *n* through zero, in advanced it is *unknown*. Therefore, we first test whether there is/are clique(s) with *n* vertices. The condition is to implement one ***CNOT*** gate to multiply the probability amplitude of the answer(s) with *n* vertices by −1 and to leave any other amplitude unchanged. If there is/are clique(s) with *n* vertices, then after repeat to execute the Grover iteration of O() times, the successful probability of measuring the answer(s) with *n* vertices is at least (1/2). Otherwise, the measurement has a failed result and we need to continue to check whether there is/are clique(s) with (*n* − 1) vertices, (*n* − 2) vertices and so on with that has one vertex until the answer(s) is/are found.

When the value of (*M* / *N*) is equal to (1 / 4), the successful probability of measuring the answer(s) is one (100%) with the Grover iteration of one time. This is the *best* case of the quantum search algorithm to solve an instance of the maximal clique problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges. When the value of *M* is equal to one, the successful probability of measuring the answer(s) is at least (1 / 2) with the Grover iteration of O() times. This is the *worst* case of the quantum search algorithm to solve an instance of the maximal clique problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges. This implies that the quantum search algorithm to solve an instance of the maximal clique problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges only gives a quadratic speed-up.

**3.4 Summary**

In this chapter, we gave a formal illustration of the search problem. We provided the satisfiability problem and the maximal clique problem as two examples of the search problem. First, we provided flowcharts of solving the satisfiability problem and the maximal clique problem. We also introduced data dependence analysis of solving the satisfiability problem and the maximal clique problem. We illustrated solution space of solving the satisfiability problem and the maximal clique problem. Next, we described the quantum circuits of implementing solution space for solving the satisfiability problem and the maximal clique problem. We also introduced the Oracle and provided the quantum circuits of implementing the Oracle to solve the satisfiability problem and the maximal clique problem. We then illustrated the Grover diffusion operator to amplify the amplitude of the answers to solve the satisfiability problem and the maximal clique problem. We also introduced the quantum circuits of implementing the Grover diffusion operator to amplify the amplitude of the answers in the satisfiability problem and in the maximal clique problem. Finally, we offered the two quantum search algorithms to solve the satisfiability problem and the maximal clique problem.

**3.5 Bibliographical Notes**

In this chapter, a more detailed introduction to the Search Problem can be found in the recommended books that are [Imre and Balazs 2005; Lipton and Regan 2014; Nielsen and Chuang 2000; Silva 2018]. A complete description for the satisfiability problem with *m* clauses and *m* Boolean variables can be found in the famous article [Cook 1972] and in the famous textbook [Garey and Johnson 1979]. A complete quantum algorithm for solving an instance of the satisfiability problem with *m* clauses and *m* Boolean variables can be found in the famous article [Chang et al 2008].

A complete illustration to the maximal clique problem to a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges and its complementary graph = (*V*, ) has *n* vertices and *m* edges in which each edge in is out of *E* can be found in the famous article [Karp 1975] and in the famous textbook [Garey and Johnson 1979]. A complete quantum algorithm for solving an instance of the maximal clique problem in a graph *G* = (*V*, *E*) with *n* vertices and *θ* edges and its complementary graph = (*V*, ) has *n* vertices and *m* edges in which each edge in is out of *E* can be found in the famous article [Chang et al 2018].

A complete description of quantum search algorithm can be found in the famous article [Grover 1996] and in the famous textbooks [Imre and Balazs 2005; Lipton and Regan 2014; Nielsen and Chuang 2000; Silva 2018]. A detailed introduction (quantum circuit) of implementing the Grover diffusion operator can be found in the famous articles [Coles et al 2018; Mandviwalla et al 2018] and in the famous textbooks [Imre and Balazs 2005; Lipton and Regan 2014; Nielsen and Chuang 2000; Silva 2018].

**3.6 Exercises**

3.1 We suppose that a graph *G* = (*V*, *E*) has that *V* is a set of vertices in *G* and *E* is a set of edges in *G*. Also we assume that *V* is {*v*1, …, *vn*} and *E* is {(*va*, *vb*)| *va* and *vb* are, respectively, elements in *V*}. We suppose that |*V*| is the number of vertices in *V* and |*E*| is the number of edges in *E*. Also we assume that |*V*| is equal to *n* and |*E*| is equal to *m*. Mathematically, a *vertex cover* of graph *G* = (*V*, *E*) with *n* vertices and *m* edges is a subset *V*1 ⊆ *V* of vertices such that for each edge (*va*, *vb*) in *E*, at least one of *va* and *vb* belongs to *V*1. The vertex cover problem of graph *G* = (*V*, *E*) with *n* vertices and *m* edges is to find a *minimum-sized* vertex cover in *G*. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem.

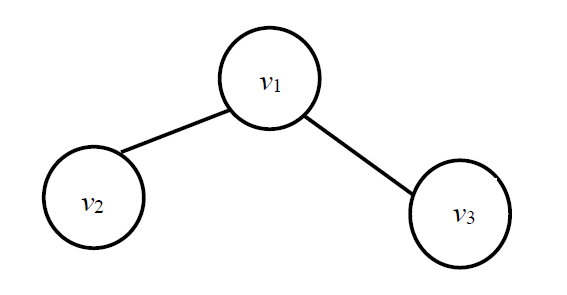


Figure 3.16: Graph *G*1 with three vertices and two edges of our problem.

Figure 3.16 is one graph of our problem. In Figure 3-16, graph *G*1 consists of three vertices and two edges. This graph denotes a vertex cover problem. All of the vertex covers in graph *G*1 are {*v*1}, {*v*2, *v*1}, {*v*3, *v*1}, {*v*3, *v*2}, and {*v*3, *v*2, *v*1}. The *minimum-sized* vertex cover for graph *G*1 is {*v*1}. Thus, the size of the vertex cover problem in Figure 3-16 is one. Please design a quantum algorithm to solve an instance of the vertex cover problem of graph *G* = (*V*, *E*) with *n* vertices and *m* edges.

3.2 We assume that a graph *G* = (*V*, *E*) has that *V* is a set of vertices in the graph *G* and *E* is a set of edges in the graph *G*. For graph *G* = (*V*, *E*), its *complementary* graph = (*V*,) in which each edge in is out of *E*. We also suppose that *V* is {*v*1, …, *vn*} and *E* is {(*va*, *vb*)| *va* and *vb* are, respectively, elements in *V*}. We assume that |*V*| is the number of vertices in *V* and |*E*| is the number of edges in *E*. We also suppose that |*V*| is equal to *n* and |*E*| is equal to *m*. An *independent-set* of graph *G* = (*V*, *E*) with *n* verticesand *m* edges is a subset *V*1 ⊆ *V* of vertices such that for all *va*, *vb* ∈ *V*1 the edge (*va*, *vb*) is *not* in *E*. The independent-set problem of graph *G* = (*V*, *E*) with *n* vertices and *m* edges is to find a *maximum-sized* independent set in graph *G*. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem.

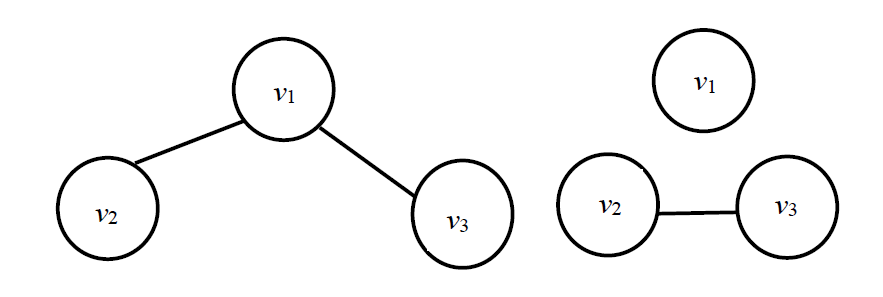


Figure 3.17: Graph *G*1 with three vertices and two edges and its *complementary* graph with the same vertices and one edge for our problem.

Figure 3.17 is one graph and its complementary graph of our problem. In Figure 3-17, graph *G*1 consists of three vertices {*v*3, *v*2, *v*1} and two edge {(*v*1, *v*2), (*v*1, *v*3)} and its complementary graph includes the same vertices and one edge {(*v*2, *v*3)}. All of the independent sets in graph *G*1 are ∅, {*v*1}, {*v*2}, {*v*3} and {*v*3, *v*2}. The maximum-sized independent set for graph *G*1 is {*v*3, *v*2}. Therefore, the size of the independent set problem in Figure 3-17 is two. Please design a quantum algorithm to solve an instance of the independent-set problem of graph *G* = (*V*, *E*) with *n* vertices and *m* edges.

3.3We suppose that a graph *G* = (*V*, *E*) has that *V* is a set of vertices in *G* and *E* is a set of edges in *G*. Also we assume that *V* is {*v*1, …, *vn*} and *E* is {(*va*, *vb*)| *va* and *vb* are, respectively, elements in *V*}. We suppose that |*V*| is the number of vertices in *V* and |*E*| is the number of edges in *E*. Also we assume that |*V*| is equal to *n* and |*E*| is equal to *m*. Mathematically, a *dominating* *set* to graph *G* = (*V*, *E*) with *n* vertices and *m* edges is a subset *V*1 ⊆ *V* of vertices such that for all *u* ∈ *V*−*V*1 there is a *v* ∈ *V*1 for which (*u*, *v*) ∈ *E*. The dominating-set problem in graph *G* = (*V*, *E*) with *n* vertices and *m* edges is to find a *minimum*-*size* dominating set in graph *G* = (*V*, *E*) with *n* vertices and *m* edges. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem.

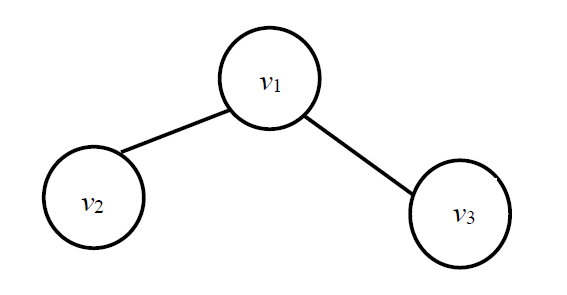


Figure 3.18: Graph *G*1 with three vertices and two edges to our problem.

Figure 3.18 is one graph of our problem. In Figure 3-18, Graph *G*1 has three vertices and two edges, where {*v*1, *v*2, *v*3} is the set of vertices and {(*v*1, *v*2), (*v*1, *v*3)} is the set of edges. The answer of the dominating-set problem for Graph *G*1 with three vertices and two edges in Figure 3-18 is {*v*1} that is the *minimum*-*size* dominating set. Hence, the size of the dominating-set problem in Figure 3-18 is one. Please design a quantum algorithm to solve an instance of the dominating-set problem for graph *G* = (*V*, *E*) with *n* vertices and *m* edges.

3.4 We assume that a finite set *S* is {*un*, …, *u*1}, where *ui* is the *i*th element in *S* with 1 ≤ *i* ≤ *n*. We also suppose that |*S*| is the number of elements in *S* and |*S*| is equal to *n*. We suppose that a collection *C* of subsets of the finite set *S* is {*C*1, …, *Cm*}, where *Cj* is the *j*th element in *C* with 1 ≤ *j* ≤ *m*. We also assume that |*C*| is the number of subsets in *C* and |*C*| is equal to *m*. Mathematically, a hitting-set is to find whether there is a subset *S*1 ⊆ *S* such that *S*1 contains at least one element from each subset in *C*. The hitting-set problem with an *n*-element finite set *S* and an *m*-element collection *C* of subsets for *S* is to find a minimum-sized hitting-set. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem.

|  |
| --- |
| S = {2, 1} and C = {{1}} |

Figure 3.19: A finite set *S* and a collection *C* of subsets of *S* to our problem.

Figure 3.19 is one example of our problem. In Figure 3-19, as an example, we consider a simple case in which the finite set *S* is {2, 1} and the collection *C* is {{1}}. *S* = {2, 1} and *C* = {{1}} define a hitting-set problem. For *S* = {2, 1} and *C* = {{1}}, there are two hitting sets that are, respectively, {1} and {1, 2}. The answer (a minimum-sized hitting-set) of the hitting-set problem for *S* = {2, 1} and *C* = {{1}} is {1}. Please design a quantum algorithm to solve an instance of the hitting-set problem with an *n*-element finite set *S* and an *m*-element collection *C* of subsets of *S*.

3.5 We assume that *W* = {*w*1, … *wq*}, *X* = {*x*1, …, *xq*} and *Y* = {*y*1, …, *yq*}are disjoint sets. We also suppose that a finite set *C* ⊆ *W* × *X* × *Y* and *C* is {(*wk*, *xl*, *ym*)| *wk* ∈ *W*, *xl* ∈ *X*, and *ym* ∈ *Y* for *q* ≥ *k*, *l*, and *m* ≥ 1}. We assume that |*C*| is the number of elements in *C* and |*C*| ≥ *t*, where *t* is a positive integer. A 3-*dimensional matching* for *C*, *W*, *X*, and *Y* is a subset *C*1 ⊆ *C* with |*C*1| ≤ *t* such that no two elements of *C*1 agree in any coordinate. The 3-dimensional matching problem to *C*, *W*, *X*, and *Y* is to find a *maximum*-*sized* 3-dimensional matching. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem.

|  |
| --- |
| *W* = {1, 2}, *X* = {3, 4}, *Y* = {5, 6} and *C* = {(1, 3, 5), (2, 4, 6), (1, 4, 6)} |

Figure 3.20: Finite sets *W*, *X,* and *Y* and a finite set *C* ⊆ *W* × *X* × *Y* to our problem.

In Figure 3.20, three finite sets *W*, *X,* and *Y* are {1, 2}, {3, 4} and {5, 6}, respectively. A finite set *C* ⊆ *W* × *X* × *Y* and *C* = {(1, 3, 5), (2, 4, 6), (1, 4, 6)}. The four sets, *W*, *X*, *Y*, and *C* denote a 3-dimensional matching problem in Figure 3.20. The *maximum*-*size* 3-dimensional matching for *C* is {(1, 3, 5), (2, 4, 6)}. Hence, the size of the 3-dimensional matching problem in Figure 3.20 is two. Please design a quantum algorithm to solve an instance of the 3-dimensional matching problem to *C*, *W*, *X*, and *Y*.

3.6 We assume that a finite set *S* is {*s*1, …, *sd*}, where *se* is the *e*th element for 1 ≤ *e* ≤ *d* in *S*. We also suppose that |*S*| is the number of elements in *S* and |*S*| is equal to *d*. We suppose that a collection *C* of subsets of the finite set *S* is {*C*1, …, *Cf*}, where *Cg* is the *g*th element for 1 ≤ *g* ≤ *f* in *C*. We also assume that |*C*| is the number of subsets in *C* and |*C*| is equal to *f*. We assume that a positive integer *k* is less than or equal to |*C*|. Mathematically, the set-basis problem is to find a collection *B* of subsets of *S* with |*B*| = *k* such that for each *Cg* ∈ *C*, there is a sub-collection of *B* whose union is exactly *Cg*. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem.

In Figure 3.21, a finite set *S* is {1, 2} and a collection *C* of subsets of *S* is {{1}, {2}}. The finite set *S* that is {1, 2} and the collection *C* that is {{1}, {2}} denote a set-basis problem in Figure 3.21. The set-basis for the finite set *S* that is {1, 2} and the collection *C* that is {{1}, {2}} in Figure 3.21 is {{1}, {2}}. Hence, the size of the set-basis problem for the finite set *S* that is {1, 2} and the collection *C* that is {{1}, {2}} in Figure 3.21 is two. Please design a quantum algorithm to solve an instance of the set-basis problem to a finite set *S* and a collection *C* of subsets of *S*.

|  |
| --- |
| *S* = {1, 2} and *C* = {{1}, {2}} |

Figure 3.21: A finite set *S* and a collection *C* of subsets of *S* in our problem.

3.7 We assume that a finite set *S* is {*s*1, …, *sq*}, where *sm* is the *m*th element for 1 ≤ *m* ≤ *q*. Also we suppose that |*S*| is the number of elements in *S* and |*S*| is equal to *q*. We assume that a collection *C* of subsets of *S* is {*C*1, …, *Cn*}, where each subset *Ck* is a subset of *S* for 1 ≤ *k* ≤ *n*. Also we suppose that |*C*| is the number of subset in *C* and |*C*| is equal to *n*. We suppose that the number of subsets in *C* is greater than or equal to *l*, where *l* is a positive number. The set-packing problem for a finite set *S* and a collection *C* of subsets of *S* is to find a sub-collection *C*1 ⊆ *C* such that *C*1 contains at least *l* mutually distinct sets. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem.

In Figure 3.22, a finite set *S* is {1, 2} and a collection *C* of subsets of the finite set *S* is {{1}, {2}}. The finite set *S* that is {1, 2} and the collection *C* that is {{1}, {2}} denote a set-packing problem in Figure 3.22. The answer of the set-packing problem for the finite set *S* that is {1, 2} and the collection *C* that is {{1}, {2}} in Figure 3.22 is {{1}, {2}}. Therefore, the size of the set-packing problem for the finite set *S* that is {1, 2} and the collection *C* that is {{1}, {2}} in Figure 3.22 is two. Please design a quantum algorithm to solve an instance of the set-packing problem to a finite set *S* and a collection *C* of subsets of *S*.

|  |
| --- |
| *S* = {1, 2} and *C* = {{1}, {2}} |

Figure 3.22: A finite set *S* and a collection *C* of subsets of *S* in our problem.

3.8 We assume that a finite set *S* is {*s*1, …, *sd*}, where *se* is the *e*th element for 1 ≤ *e* ≤ *d* in *S*. We also suppose that |*S*| is the number of elements in *S* and |*S*| is equal to *d*. We suppose that a collection *C* of subsets of the finite set *S* is {*C*1, …, *Cf*}, where *Cg* is the *g*th element for 1 ≤ *g* ≤ *f* in *C*. We also assume that |*C*| is the number of subsets in *C* and |*C*| is equal to *f*. Mathematically, the set-splitting problem is to find whether there is a partition of *S* into two subsets *S*1 and *S*2 such that no subset in *C* is entirely contained in either *S*1 or *S*2. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem.

In Figure 3.23, a finite set *S* is {1, 2} and a collection *C* of subsets of *S* is {{1, 2}}. The finite set *S* that is {1, 2} and the collection *C* that is {{1, 2}} in Figure 3.23 define a set-splitting problem. The set splitting for the finite set *S* that is {1, 2} and the collection *C* that is {{1, 2}} in Figure 3.23 is *S*1 = {1} and *S*2 = {2} or *S*1 = {2} and *S*2 = {1}. Please design a quantum algorithm to solve an instance of the set-splitting problem to a finite set *S* and a collection *C* of subsets of *S*.

|  |
| --- |
| *S* = {1, 2} and *C* = {{1, 2}} |

Figure 3.23: A finite set *S* and a collection *C* of subsets of *S* in our problem.

3.9 A 3-satisfiability problem with *n* Boolean variables that are {*x*1 *x*2 … *xn* − 1 *xn*} and *m* clauses contains a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*, where each clause *Cj* for 1 ≤ *j* ≤ *m* is a formula of the form *xp* ∨ *xq* ∨ *xr* for three Boolean variables *xp*, *xq* and *xr* to 1 ≤ *p*, *q* and *r* ≤ *n*. This is to say that each clause *Cj* for 1 ≤ *j* ≤ *m* only includes three Boolean variables. Next, the question is to find values of each Boolean variable so that the whole formula has the value 1. This is the same as finding values of each Boolean variable that make each clause have the value 1. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem.

As an example, we consider that a 3-satisfiability problem with three Boolean variables that are {*x*1 *x*2 *x*3} and oneclause that is (*x*1 ∨ *x*2 ∨ *x*3). The three variables *x*1, *x*2 and *x*3 are Boolean variables and their values are allowed to range only over two values 0 and 1. We usually think of 0 as “false” and 1 as “true”. The symbol “∨” is the “logical or” operation. The answers that satisfy that the clause (*x*1 ∨ *x*2 ∨ *x*3) has the value 1 are subsequently *x*10 *x*20 *x*31 (001), *x*10 *x*21 *x*30 (010), *x*10 *x*21 *x*31 (011), *x*11 *x*20 *x*30 (100), *x*11 *x*20 *x*31 (101), *x*11 *x*21 *x*30 (110) and *x*11 *x*21 *x*31 (111). Please design a quantum algorithm to solve an instance of the 3-satisfiability problem with *n* Boolean variables that are {*x*1 *x*2 … *xn* − 1 *xn*} and *m* clauses.

3.10 A not-all-equal 3-satisfiability problem with *n* Boolean variables that are {*x*1 *x*2 … *xn* − 1 *xn*} and *m* clauses contains a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*, where each clause *Cj* for 1 ≤ *j* ≤ *m* is a formula of the form *xp* ∨ *xq* ∨ *xr* for three Boolean variables *xp*, *xq* and *xr* to 1 ≤ *p*, *q* and *r* ≤ *n*. This indicates that each clause *Cj* for 1 ≤ *j* ≤ *m* only includes three Boolean variables. Next, the question is to find values of each Boolean variable so that each clause *Cj* for 1 ≤ *j* ≤ *m* has *at least* one *true* Boolean variable and one *false* Boolean variable so that the whole formula has the value 1. This is the same as finding values of each Boolean variable so that each clause *Cj* for 1 ≤ *j* ≤ *m* has *at least* one *true* Boolean variable and one *false* Boolean variable that make each clause have the value 1. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem.

As an example, we consider that a not-all-equal 3-satisfiability problem with three Boolean variables that are {*x*1 *x*2 *x*3} and oneclause that is (*x*1 ∨ *x*2 ∨ *x*3). The three variables *x*1, *x*2 and *x*3 are Boolean variables and their values are allowed to range only over two values 0 and 1. We usually think of 0 as “false” and 1 as “true”. If the value of a Boolean variable is equal to 1 (one), we call it as one *true* Boolean variable. If the value of a Boolean variable is equal to 0 (zero), we call it as one *false* Boolean variable. The symbol “∨” is the “logical or” operation. The answers that satisfy that the clause (*x*1 ∨ *x*2 ∨ *x*3) has *at least* one *true* Boolean variable and one *false* Boolean variable that make the clause have the value 1 are subsequently *x*10 *x*20 *x*31 (001), *x*10 *x*21 *x*30 (010), *x*10 *x*21 *x*31 (011), *x*11 *x*20 *x*30 (100), *x*11 *x*20 *x*31 (101) and *x*11 *x*21 *x*30 (110). Please design a quantum algorithm to solve an instance of the not-all-equal 3-satisfiability problem with *n* Boolean variables that are {*x*1 *x*2 … *xn* − 1 *xn*} and *m* clauses.

3.11 A one-in-three 3-satisfiability problem with *n* Boolean variables that are {*x*1 *x*2 … *xn* − 1 *xn*} and *m* clauses contains a Boolean formula of the form *C*1 ∧ *C*2 … ∧ *Cm*, where each clause *Cj* for 1 ≤ *j* ≤ *m* is a formula of the form *xp* ∨ *xq* ∨ *xr* for three Boolean variables *xp*, *xq* and *xr* to 1 ≤ *p*, *q* and *r* ≤ *n*. This is to say that each clause *Cj* for 1 ≤ *j* ≤ *m* only includes three Boolean variables. Next, the question is to find values of each Boolean variable so that each clause *Cj* for 1 ≤ *j* ≤ *m* has *exactly* one *true* Boolean variable so that the whole formula has the value 1. This is the same as finding values of each Boolean variable so that each clause *Cj* for 1 ≤ *j* ≤ *m* has *exactly* one *true* Boolean variable that make each clause have the value 1. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem.

As an example, we consider that a one-in-three 3-satisfiability problem with three Boolean variables that are {*x*1 *x*2 *x*3} and oneclause that is (*x*1 ∨ *x*2 ∨ *x*3). The three variables *x*1, *x*2 and *x*3 are Boolean variables and their values are allowed to range only over two values 0 and 1. We usually think of 0 as “false” and 1 as “true”. If the value of a Boolean variable is equal to 1 (one), we call it as one *true* Boolean variable. If the value of a Boolean variable is equal to 0 (zero), we call it as one *false* Boolean variable. The symbol “∨” is the “logical or” operation. The answers that satisfy that the clause (*x*1 ∨ *x*2 ∨ *x*3) has *exactly* one *true* Boolean variable that makes the clause have the value 1 are subsequently *x*10 *x*20 *x*31 (001), *x*10 *x*21 *x*30 (010) and *x*11 *x*20 *x*30 (100). Please design a quantum algorithm to solve an instance of the one-in-three 3-satisfiability problem with *n* Boolean variables that are {*x*1 *x*2 … *xn* − 1 *xn*} and *m* clauses.

3.12 We assume that a finite set *S* is {*s*1, …, *sA*}, where *sk* is one element in *S* for 1 ≤ *k* ≤ *A*. We also suppose that |*S*| is the number of elements in *S* and |*S*| is equal to *A*. We assume that a collection *C* of subsets of *S* is {*C*1, …, *CB*}, where *Ci* is a subset of *S* for 1 ≤ *i* ≤ *B*. We also suppose that |*C*| is the number of elements in *C* and |*C*| is equal to *B*. We assume that *m* is a positive integer. A *set* *cover* for *S* is a sub-collection *C*1 ⊆ *C* with |*C*1| ≤ *m* such that every element of *S* belongs to at least one member of *C*1, where |*C*1| is the number of elements in *C*1. The set cover problem is to find a *minimum*-*size* set cover for *S*. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem.

In Figure 3.24, a finite set *S* is {1, 2, 3} and a collection *C* of subsets of *S* is {{1}, {2}, {3}, {2, 3}}.The finite set *S* that is {1, 2, 3} and the collection *C* of subsets of *S* that is {{1}, {2}, {3}, {2, 3}} in Figure 3.24 denote a set cover problem. The *minimum*-*size* set cover for the finite set *S* that is {1, 2, 3} and the collection *C* of subsets of *S* that is {{1}, {2}, {3}, {2, 3}} in Figure 3.24 is {{1}, {2, 3}}. Please design a quantum algorithm to solve an instance of the set cover problem for a finite set *S* and a collection *C* of subsets of *S*.

|  |
| --- |
| *S* ={1, 2, 3} and *C* = {{1}, {2}, {3}, {2, 3}} |

Figure 3.24: A finite set *S* and a collection *C* of subsets of *S* in our problem.

3.13 We assume that a finite set *S* is {*s*1, …, *s*3 × *q*}, where *sk* is one element in *S* for 1 ≤ *k* ≤ 3 × *q*. We also suppose that |*S*| is the number of elements in *S* and is equal to 3 × *q*. We assume that a collection *C* of 3-element subsets to *S* is {*C*1, …, *CB*} and each subset *Ci* contains three elements in *S* for 1 ≤ *i* ≤ *B*. We assume that |*C*| is the number of elements in *C*. A*n* *exact* *cover* for *S* is a sub-collection *C*1 ⊆ *C* such that every element of *S* occurs in exactly one member of *C*1. The problem of exact cover by 3-sets is to find a *minimum*-*size* exact cover for *S*. It is a ***NP-Complete*** problem and can be formulated as a “computational search” problem. In Figure 3.25, a finite set *S* is {1, 2, 3, 4, 5, 6} and a collection *C* of 3-element subsets to *S* is {{1, 2, 3}, {3, 4, 5}, {4, 5, 6}}.Te finite set *S* that is {1, 2, 3, 4, 5, 6} and the collection *C* of 3-element subsets to *S* that is {{1, 2, 3}, {3, 4, 5}, {4, 5, 6}} in Figure 3.25 denote a problem of exact cover by 3-sets. The *minimum*-*size* exact cover for the finite set *S* is {1, 2, 3, 4, 5, 6} and the collection *C* of 3-element subsets to *S* that is {{1, 2, 3}, {3, 4, 5}, {4, 5, 6}} in Figure 3.25 is {{1, 2, 3}, {4, 5, 6}}. Please design a quantum algorithm to solve an instance of the problem of exact cover by 3-sets for a finite set *S* and a collection *C* of 3-element subsets to *S*.

|  |
| --- |
| *S* ={1, 2, 3, 4, 5, 6} and *C* = {{1, 2, 3}, {3, 4, 5}, {4, 5, 6}} |

Figure 3.25: A finite set *S* and a collection *C* of 3-element subsets to *S* in our problem.