6

GAS: Grover Adaptive Search

If you do not expect the unexpected, you will not find it, for it is not to be reached by search or trail.

— Heraclitus

In this chapter, we are going to introduce another quantum method for solving combinatorial optimization problems. In this case, we are going to take Grover's algorithm — one of the most famous and celebrated quantum methods out there — as a starting point. Grover's algorithm is used to find elements that satisfy specific conditions in unsorted data structures. But, as we will soon see, it can be easily adapted to function minimization tasks — exactly what we need for our optimization problems! The resulting method is sometimes called **Grover Adaptive Search** or **GAS**.

It is important to note that GAS is essentially different from the kind of quantum algorithms that we have been studying so far in this part of the book. This method is not designed specifically for NISQ devices and would need fault-tolerant quantum computers to fully realize its potential. However, we have still decided to cover it because it is readily

implemented in some quantum programming libraries — such as Qiskit — and it can be helpful in comparing and benchmarking other quantum optimization algorithms.

We will start the chapter by refreshing some details about Grover's algorithm, including the circuits that we need in order to implement it and the role that **oracles** play in it. Then, we will talk about the **Dürr-Høyer** method, which uses Grover's techniques to find the minimum of certain types of functions. After that, we will particularize the algorithm to QUBO problems and we will study how to implement the kind of oracle that they require.

With all those tools, we will have everything that we need in order to formulate and solve optimization problems with GAS, so we will turn to explain how to use Qiskit's implementation of the algorithm. We will study the different options that are available to run the method and we will test it on several different examples.

After reading this chapter, you will understand the theoretical foundations of Grover Adaptive Search, you will know how to implement efficient oracles for optimization problems and how to use them with GAS, and you will be able to run Qiskit's implementation of the algorithm to solve your own optimization problems.

The topics covered in this chapter are as follows:

- Grover's algorithm
- Quantum oracles for combinatorial optimization
- Using GAS with Qiskit

6.1 Grover's algorithm

In this section, we will cover the most important properties of Grover's algorithm. We will not cover all the theoretical details behind the procedure — for that, we recommend the book by Nielsen and Chuang [16] and, especially, the lecture notes by John Watrous [46] — but we need to at least get familiar with how the method operates, what oracles are and how they are used in the algorithm, and what kind of circuits are needed to implement it.

Let's start with the basics. Grover's algorithm is used for searching elements that satisfy certain conditions. More formally, the algorithm assumes that we have a collection of

elements indexed by strings of n bits, and a Boolean function f that takes those binary strings and returns "true" (or 1) if the element indexed by the string satisfies the condition and "false" (or 0) otherwise. For instance, imagine that we are searching among 8 different elements and that the ones that satisfy the condition are indexed by the strings 010 and 100. Then, f will be the Boolean function such that $f(x) = 1$ if $x = 010$ or $x = 100$, and $f(x) = 0$ otherwise. To simplify the notation, from now on we will identify an element with the string x that is used to index it.

It is important to notice that, in this setting, we have no access to the inner workings of f. It acts like a black box. The only thing that we can do with the f function is call it on inputs and observe the outputs, thus checking whether the given input satisfies the condition that we are considering or not. Since we do not have any information about the indices of the elements that satisfy the condition, we cannot favour any position over any other. Thus, with a classical algorithm, if we are searching among N elements and only one of them satisfies the condition we are interested in, we will need to call f about $N/2$ times on average in order to find it. The element could be just anywhere! In fact, if we are extremely unlucky, we might need to use $N - 1$ calls (notice that we wouldn't need N calls: if we don't find the element after $N - 1$ different calls, we already know the remaining position to be the one where the element is located).

It may come as a big surprise, then, that with Grover's algorithm it is possible to find the hidden element with high probability (much more on this later in this section) by calling f around $\sqrt{N}$ times! This means that if we are searching among 1 000 000 elements, with a classical computer you would need to check f about 500 000 times on average, but calling f less than 1000 times would suffice in order to solve the problem with a quantum computer, at least with a high likelihood. What is more, the difference in the number of calls between the classical and the quantum methods grows bigger if N is higher.

How is this possible? It seems to defy all logic, but it rests on properties that we are already familiar with, such as superposition and entanglement. In fact, Grover's algorithm will query f with elements that are in superposition. But in order to understand this, we need to explore what quantum oracles are and how they can be used, so let's get to it!

6.1.1 Quantum oracles

We have mentioned that, in the setting of the search problem solved by Grover's algorithm, we are given a Boolean function f that we can use to determine whether an element is the one we are looking for or not. But what do we mean when we say that we are "given" this function?

In the classical case, this is more or less straightforward. If we were writing our code in Python, we could be given a function object that receives an n-bit string and returns `True` or `False`. Then, we could use that function in our own code to check the elements that we want to consider, without necessarily knowing how it is implemented.

But... what is the equivalent to that function definition when we are working with quantum circuits? The most natural assumption is that we are provided with a new quantum gate O_f that implements f and that we can use in our circuits whenever we need it. However, a quantum gate needs to be a unitary operation and, in particular, reversible, so we need to be a little bit careful in how we design it.

In the classical case, we had n inputs — the n bits of the string — and just one output. In the quantum case, we need at least n inputs — n qubits — but just one output would not work, because then it would be impossible to make the operation reversible, let alone unitary. In fact, as you surely remember, every quantum gate has the same number of inputs and outputs.

The usual approach, then, is to consider a quantum gate on $n+1$ qubits. The first n of these qubits will serve as the input and the additional one will be used to store the output. More formally, on any input $|x\rangle|y\rangle$, where x is an n-bit string and y is a single bit, the output of the O_f gate will be $|x\rangle|y \oplus f(x)\rangle$, where $\oplus$ denotes addition modulo 2 (see *Appendix B, Basic Linear Algebra*, for a refresher on modular arithmetic). This defines the action of the gate on the computational basis states and then we can extend it to the rest of the quantum states by linearity, as usual.

This may look like an odd choice. The "natural" thing to do might seem to be requiring the output to be $|x\rangle|f(x)\rangle$, right? But that would not be reversible in general, because we would

obtain the same output over the inputs $|x\rangle|0\rangle$ and $|x\rangle|1\rangle$. With our choice, though, the operation is reversible. If we applied O_f twice, we would obtain $|x\rangle|y \oplus f(x) \oplus f(x)\rangle$, which is equal to $|x\rangle|y\rangle$ because, when we are performing addition modulo 2, $f(x) \oplus f(x) = 0$ no matter the value of $f(x)$.

Exercise 6.1

Prove that O_f is not only reversible but also unitary and hence it deserves the name "quantum gate."

Usually, O_f is said to be a quantum oracle for f, because we can consult it to get the value of f on any input x without having to worry about its internal workings. In fact, if the input to O_f is $|x\rangle|0\rangle$, then the output is $|x\rangle|0 \oplus f(x)\rangle = |x\rangle|f(x)\rangle$ and we could hence recover $f(x)$ just by measuring the last qubit.

For any f, it is always possible to construct O_f by using just NOT and multi-controlled NOT gates — even if the resulting circuit is not the most efficient one in most cases. For instance, if f is a Boolean function on 3-bit strings such that f takes value 1 just on 101 and 011, then we can use the circuit depicted in *Figure 6.1*. Notice how we have used NOT gates before and after the multi-controlled gates to select those qubits that should be 0 in the input and to restore them to their original values.

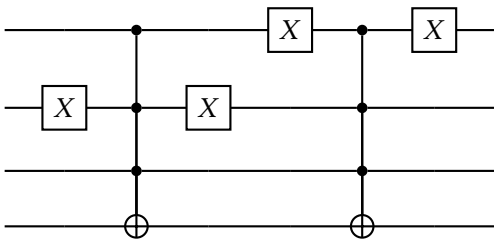

Figure 6.1: Oracle for the Boolean function f that takes value 1 on 101 and 011, and value 0 on the rest of the 3-bit strings

Exercise 6.2

Construct a circuit for O_f where f is a 4-bit Boolean function that takes value 1 on 0111, 1110, and 0101, and value 0 on any other input.

This settles how we are going to receive the Boolean function f that we can use to check whether a given element satisfies the conditions that we are interested in: the function will be given to us as a quantum oracle. Now it's time for us to show how we can use these quantum oracles in Grover's algorithm.

6.1.2 Grover's circuits

Let's say that we want to apply Grover's algorithm to a Boolean function f which receives binary strings of length n. In addition to the O_f oracle described in the previous section, the circuit used in Grover's algorithm involves two other blocks, as you can see in *Figure 6.2*.

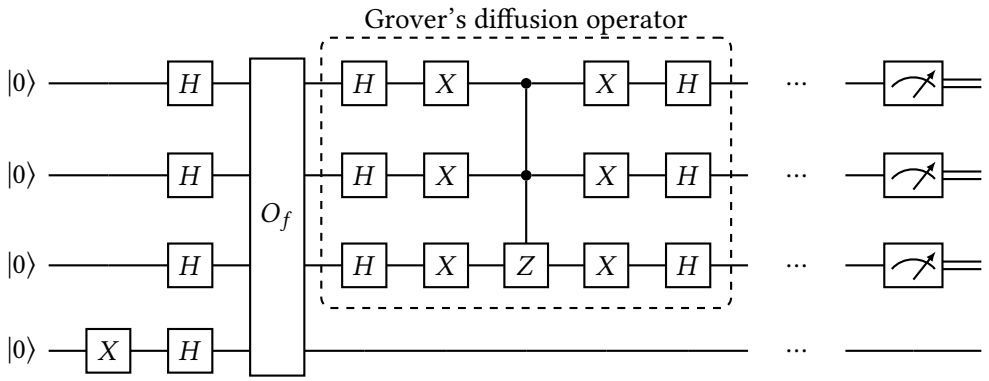

Figure 6.2: Circuit for Grover's algorithm in the case in which f receives strings of length 3 as input. The oracle O_f and Grover's diffusion operator are repeated, in that order, a number of times before the final measurements

The first block is composed of one-qubit gates that are applied to the initial state $|0 \cdots 0\rangle |0\rangle$, where the first register is of length n and the second one is of length 1. Thus, the state just

before applying the oracle is

$$H^{\otimes n+1}\left|0\right\rangle^{\otimes n}\left|1\right\rangle = \left|+\right\rangle^{\otimes n}\left|-\right\rangle = \frac{1}{\sqrt{2^n}}\left(\left(\left|0\right\rangle + \left|1\right\rangle\right)\cdots\left(\left|0\right\rangle + \left|1\right\rangle\right)\right)\left|-\right\rangle = \frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}\left|x\right\rangle\left|-\right\rangle,$$

because we apply the first X gate to $\left|0\right\rangle$ to obtain $\left|1\right\rangle$.

Notice that the first register of this state is a superposition of all basis states $\left|x\right\rangle$. This is exactly what we will use in order to evaluate f "in superposition" with our application of the O_f oracle. Indeed, by the definition of O_f, the state that we will have after the application of the oracle is

$$O_f\left(\frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}\left|x\right\rangle\left|-\right\rangle\right) = O_f\left(\frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}\left|x\right\rangle\left(\left|0\right\rangle - \left|1\right\rangle\right)\right) =$$
$$\frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}O_f\left|x\right\rangle\left(\left|0\right\rangle - \left|1\right\rangle\right) = \frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}\left|x\right\rangle\left(\left|0\oplus f(x)\right\rangle - \left|1\oplus f(x)\right\rangle\right),$$

where in the last two equalities he have used linearity together with the definition of O_f.

Let's focus on the $\left|0\oplus f(x)\right\rangle - \left|1\oplus f(x)\right\rangle$ term. If $f(x) = 0$, then it is just $\left|0\right\rangle - \left|1\right\rangle$. However, if $f(x) = 1$, we have

$$\left|0\oplus f(x)\right\rangle - \left|1\oplus f(x)\right\rangle = \left|0\oplus 1\right\rangle - \left|1\oplus 1\right\rangle = \left|1\right\rangle - \left|0\right\rangle = -\left(\left|0\right\rangle - \left|1\right\rangle\right),$$

because $1\oplus 1 = 0$. In both cases, we can write

$$\left|0\oplus f(x)\right\rangle - \left|1\oplus f(x)\right\rangle = (-1)^{f(x)}\left(\left|0\right\rangle - \left|1\right\rangle\right),$$

because $(-1)^0 = 1$ and $(-1)^1 = -1$.

Note how, thanks to these transformations, there is information about the value $f(x)$ coded into the amplitude of the state now. As you will soon see, this is a key ingredient of the algorithm.

If we take this to our expression for the state after the oracle application, we get

$$O_f\left(\frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}|x\rangle|-\rangle\right) = \frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}|x\rangle\left(|0\oplus f(x)\rangle - |1\oplus f(x)\rangle\right) =$$
$$\frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}|x\rangle\left(|0\rangle - |1\rangle\right) = \frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}|x\rangle\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) =$$
$$\frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}|x\rangle|-\rangle\,.$$

Notice how the application of O_f has introduced a relative phase in some of the states $|x\rangle$ of the superposition. This technique is called **phase kickback**, because we have only used the register in state $|-\rangle$ to create the phase but it ends up affecting the whole state. It is used in other famous quantum methods such as the Deutsch-Jozsa and Simon's algorithms (see the book by Yanofsky and Mannucci [47] for an excellent explanation of these methods).

As we have proved, the phase that goes with the basis state $|x\rangle$ depends only on $f(x)$ and it is 1 if $f(x) = 0$ and -1 if $f(x) = 1$. In this way, we say that we have **marked** those elements that satisfy the conditions that we are interested in, that is, those elements x such that $f(x) = 1$. Remarkably, we have done this with just one call to O_f, exploiting the possibility of evaluating it in superposition. That is an exponential number of function evaluations with just one call! It sounds like magic, doesn't it?

However, although after applying O_f we have somehow separated the elements x that satisfy $f(x) = 1$ from the rest, we do not seem to be closer to finding one of them. If we measure the state as it is, the probability of measuring an x such that $f(x) = 1$ is the same as it was before applying O_f. The phase that we have introduced has an absolute value equal to 1 and, consequently, does not affect the measurement probability.

But, wait! There is more to Grover's algorithm. There is another circuit block that we apply after O_f: it's called **Grover's diffusion operator** and we will use it to increase the probability of measuring the marked states. Describing its inner workings in full detail would take us astray from our path — for that, we recommend checking out Dancing with

Qubits [7], by Robert Sutor, which offers a perfect explanation of its behaviour — but let's at least give a quick overview of what it does.

Grover's diffusion operator implements an operation called **inversion about the mean**. This may sound complicated, but in fact it is quite simple. First, the average value m of all the amplitudes of the states is computed. Then, every amplitude a is replaced with $2m - a$. After this transformation, the positive amplitudes will be a little bit smaller, but the negative ones will be a little bit bigger. This is why the technique used by Grover's algorithm is called **amplitude amplification**. Again, we recommend you checking Sutor's book [7] for a detailed description of how this operation works.

So, after this first application of Grover's diffusion operator, the amplitudes of the elements that we are interested in finding are a little bit larger. But, in general, this will still not be enough to guarantee a high probability of measuring one of them. For this reason, we will need to mark the elements again with O_f and then apply the diffusion operator once more. We will repeat this procedure, applying first O_f and then the diffusion operator, several times until the probability of measuring one of the states we are looking for is high enough (close to 1). And that is the moment when we can measure the whole state and observe the result to, hopefully, obtain one element that satisfies the conditions.

But how many times should we apply O_f followed by the diffusion operator? This is a crucial point in Grover's algorithm that we will study in more detail in the next subsection.

6.1.3 Probability of finding a marked element

As we have just seen, when using Grover's algorithm, we are repeatedly applying for a certain number of times the quantum oracle given to us followed by the diffusion operator. Of course, we would like the number of repetitions to be as small as possible — so that the algorithm runs faster — while guaranteeing a high probability of finding one of the marked elements. How can we go about this?

One possible approach in order to analyze the behaviour of Grover's algorithm could be studying the properties of the inversion about the mean operation that we mentioned in

the previous subsection. However, there is a better way. It turns out that the combination of O_f and Grover's diffusion operator acts like a rotation in a two-dimensional space. We will not give the full details — check the lecture notes by John Watrous [46] for a very thorough and readable explanation — but, if we have n-bit strings and there is only one marked element x_1, it can be proved that the state that we reach after m applications of O_f followed by the diffusion operator is

$$\cos(2m+1)\theta\,|x_0\rangle + \sin(2m+1)\theta\,|x_1\rangle\,,$$

where

$$|x_0\rangle = \sum_{x\in\{0,1\}^n, x\neq x_1} \sqrt{\frac{1}{2^n-1}}\,|x\rangle$$

and $\theta \in (0, \pi/2)$ is such that

$$\cos\theta = \sqrt{\frac{2^n-1}{2^n}}, \qquad \sin\theta = \sqrt{\frac{1}{2^n}}.$$

Notice that $|x_0\rangle$ is just the uniform superposition of the states $|x\rangle$ such that $f(x) = 0$. Then, what we want to obtain is a state in which $\sin(2m+1)\theta$ is close to 1, because then we would have a high probability of finding x_1 when we measure. For that, ideally, we would like to have

$$(2m+1)\theta \approx \frac{\pi}{2},$$

because $\sin \pi/2 = 1$.

Solving for m, we obtain

$$m \approx \frac{\pi}{4\theta} - \frac{1}{2}.$$

What is more, we know that $\sin\theta = \sqrt{1/2^n}$, so, for a big enough n, we will have

$$\theta \approx \sqrt{\frac{1}{2^n}}$$

and then we can choose

$$m = \left\lfloor \frac{\pi}{4} \sqrt{2^n} \right\rfloor,$$

that is, the biggest integer that is less than or equal to $(\pi/4)\sqrt{2^n}$.

Notice that there are exactly 2^n elements but only one of them satisfies the conditions we are interested in. This means that, with a classical algorithm, if we can only use f to check if an element x is the one we are looking for — that is, to check if $f(x) = 1$ — then we would need about $2^n/2$ calls to f on average to find x. However, with Grover's algorithm, we only need about $\sqrt{2^n}$. That is a quadratic speedup!

Nevertheless, there is a subtlety here. In the classical setting, if we use f more times, the probability of finding the marked element increases. But with Grover's algorithm, if m is not selected wisely, we can overshoot and actually decrease the success probability instead of increasing it!

This sounds baffling. How is it possible that by searching more we find ourselves with less possibilities of finding the hidden element? The key is that, as we have shown, the probability of measuring x_1 is $(\sin(2m+1)\theta)^2$. This function is periodic and oscillates between 0 and 1, so after reaching values close to 1, it goes back down to 0.

Let's illustrate this with an example. In *Figure 6.3*, we consider the case $n = 4$ and we show how the probability of finding exactly one marked element changes as we vary the number of Grover iterations m, from 0 to 20. In this case, $\lfloor(\pi/4)\sqrt{2^n}\rfloor$ is 3 and, as you can see, the success probability with $m = 3$ is close to 1. However, for $m = 5$ the probability has decreased dramatically, and for $m = 6$ it is nearly 0.

This shows that we need to be very careful when selecting the number of iterations m in Grover's algorithm. For the case in which there is only one marked element, we have obtained a good choice for m. But what if there is more than one marked element? It turns out — check the lecture notes by John Watrous [46] — that if there are k marked elements,

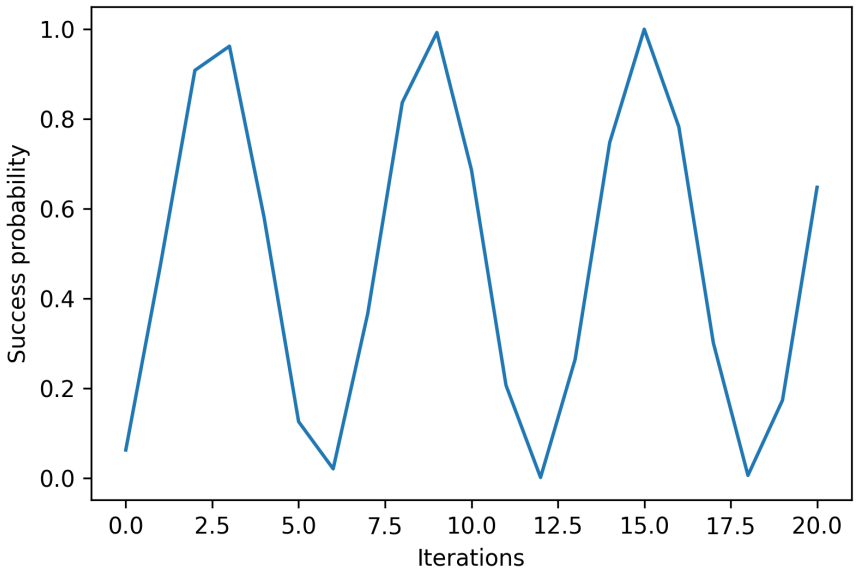

Figure 6.3: Probability of finding one marked element among 16 when using Grover's algorithm with a number of iterations that varies from 0 *to* 20

we can repeat our previous reasoning and show that a good value for m is

$$m = \left\lfloor \frac{\pi}{4} \sqrt{\frac{2^n}{k}} \right\rfloor,$$

provided that k is small compared to 2^n. If k is not small compared to 2^n, don't worry; then the probability of finding a marked element just by choosing at random is $k/2^n$, which will be sizeable, so you wouldn't even need a quantum computer in the first place.

This solves our problem if we know how many marked elements there are. But, in the most general case, we may lack that information. In that circumstance, we can apply the results of a very useful paper by Boyer, Brassard, Høyer, and Tapp [48]. They showed that by choosing m at random in a range that increases dynamically, we can still be guaranteed that will find a marked element with high probability while keeping the average number of iterations as $O(\sqrt{2^n})$ (see *Appendix C, Computational Complexity*, for a refresher on asymptotic notation).

In fact, they proved that the probability of finding a marked element with their method is at least $1/4$. This might seem unimpressive, but we can easily see how that is more than enough. Indeed, the probability of not finding a marked element is then no more than $3/4$. So, suppose that we repeat the process 1000 times. Then, the probability of failure is at most $(3/4)^{1000}$, which is extremely low. In fact, the chance of a meteorite hitting your quantum computer while running your circuits is much, much bigger than that!

So far in this section, we have covered all that we need to know in order to apply Grover's algorithm in search problems. However, our main goal is solving optimization problems. We explore the connection between both tasks in the next subsection.

6.1.4 Finding minima with Grover's algorithm

Optimization problems are obviously related to search problems. In fact, when solving an optimization problem, we are trying to find a value with a special property: it should be a minimum or maximum among all the possible values. This connection was exploited by Dürr and Høyer in a 1996 paper [49] in which they introduced a quantum algorithm, based on Grover's search, to find minima of functions. The main idea behind the algorithm is quite straightforward. Suppose we want to find a minimum of a function g that is computed over binary strings of length n. We select one such string x_0 at random and we compute $g(x_0)$. Now we apply Grover's algorithm with an oracle that, on input x, returns 1 if $g(x) < g(x_0)$ and 0 otherwise. If the element that we measure after applying Grover's search, call it x_1, really achieves a value that is lower than $g(x_0)$, we replace x_0 with it and repeat the process but now with an oracle that checks the condition $g(x) < g(x_1)$. If not, we keep using x_0. We repeat this process several times and we return the element with the lowest value among the ones that we have considered.

There are a couple of details that we need to flesh out here. The first one is how to construct the oracles. In general, of course, it will depend on the function g. For that reason, in the next section we will focus on circuits that we can use with the Dürr-Høyer algorithm to solve QUBO and HOBO problems.

On the other hand, we should take care of the number of iterations that we will use in each application of Grover's algorithm and, also, of the number of times that we need to repeat the procedure for selecting a new element and constructing a new oracle. The original paper by Dürr and Høyer gives all the details, but let's just mention that it uses the method proposed by Boyer, Brassard, Høyer, and Tapp [48] that we explained in the previous subsection, and it guarantees that a minimum will be found with a probability of at least $1/2$ with a number of calls to the oracle that is $O(\sqrt{2^n})$.

With this, we have now covered all the concepts that we need in order to apply this search method to solve QUBO and HOBO problems. We will devote the next section to explaining how to construct quantum oracles for these kinds of problems.

6.2 Quantum oracles for combinatorial optimization

As we have seen, the Dürr-Høyer algorithm can be used to find the minimum of a function g with high probability and with a quadratic speedup over brute force search. However, in order to use it, we need a quantum oracle that, given binary strings x and y, checks whether $g(x) < g(y)$.

In our case, we are interested in functions g that can appear in QUBO and HOBO problems. This means that g will be a polynomial with real coefficients and binary variables, and we could implement the quantum oracle with a straightforward approach: design a classical circuit for it using AND, OR, and NOT gates, and then simulate the classical gates with the Toffoli quantum gate, as we showed in *Section 1.5.2*.

However, in 2021, Gilliam, Woerner, and Gonciulea, introduced an improved way of implementing quantum oracles for QUBO and HOBO problems in a paper titled *Grover adaptive search for constrained polynomial binary optimization* [50].

In this section, we will study in detail the techniques that they proposed and how to use them to implement our quantum oracles. We will start by considering the case in which all the coefficients of the polynomial are integer numbers and, then, we will extend our study

to the most general case when the coefficients are real numbers. But, before we get to that, we need to take a brief detour to talk about one of the most important subroutines in all of quantum computing: the **quantum Fourier transform**.

6.2.1 The quantum Fourier transform

The quantum Fourier transform (usually abbreviated as **QFT**) is, beyond any doubt, one of the most useful tools in quantum computing. It is an essential part of Shor's algorithm for integer factorization [6] and it is behind the speedups of other famous quantum algorithms such as HHL [14].

We will use the QFT to help us implement the arithmetical operations that we need to compute the values of the polynomial function of our QUBO and HOBO problems. We could, for instance, implement these operations in a basis representation. As an example, we might design a unitary transformation taking $|x\rangle\,|y\rangle\,|0\rangle$ to $|x\rangle\,|y\rangle\,|x+y\rangle$, where x and y are binary numbers and $x+y$ is their addition. However, this could involve a big number of one- and two-qubit gates.

Instead, we will use the approach proposed by Gilliam, Woerner, and Gonciulea in [50] and we will compute the arithmetical operations using the state amplitudes. We will explain in detail how to do that in the next subsections. But, before that, we will study how to use the QFT to recover information from the amplitudes of a quantum state.

The QFT on m qubits is defined as the unitary transformation that takes the basis states $|j\rangle$ to

$$\frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} e^{\frac{2\pi i j k}{2^m}} |k\rangle\,,$$

where i is the imaginary unit. Its action is extended to the rest of the states by linearity.

We will not study the properties of the QFT in detail. For that, you can refer to *Dancing with Qubits*, by Robert Sutor [7]. However, we need to know that the QFT can be implemented with a number of one- and two-qubit gates that is quadratic in m. This is an exponential speedup over the best algorithm that we have for the analogous classical operation (the **discrete Fourier transform**).

For example, the circuit for the QFT on three qubits is shown in *Figure 6.4*. In it, the rightmost gate, which acts on the top and bottom qubits, is the SWAP gate. As we mentioned in *Section 1.4.3*, this gate swaps the states of two qubits and it can be implemented by means of CNOT gates. Moreover, this QFT circuit uses the **phase gate**, denoted by $P(\theta)$. This is a parametrized gate that depends on an angle θ and whose coordinate matrix is

$$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}.$$

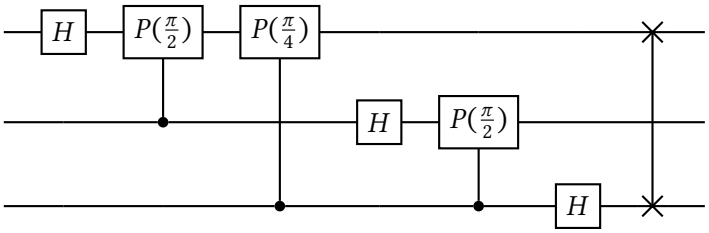

Figure 6.4: Circuit for the quantum Fourier transform on 3 *qubits*

Important note

The phase gate is very similar to the R_Z gate that we introduced in *Section 1.3.4*. In fact, when applied on its own to a qubit, $P(\theta)$ is equivalent to $R_Z(\theta)$ up to an unimportant global phase. However, in the QFT circuit, we are using a controlled version of the phase gate and the global phase becomes a relative one, which is not unimportant at all!

As we have seen, the QFT acts by introducing phases of the form $e^{2\pi ijk/2^m}$ when it is applied on basis states $|j\rangle$. Nevertheless, for the purposes of our computations, we are more interested in recovering the values j from those phases. For that, we will need the **inverse quantum Fourier transform**, usually denoted $\text{QFT}^\dagger$. Of course, its action is the inverse

of that of the QFT, meaning that it takes a state such as

$$\frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} e^{\frac{2\pi i j k}{2^m}} |k\rangle$$

to the basis state $|j\rangle$.

The circuit for the inverse QFT can be obtained from that of the QFT by reading the circuit backwards and using the inverse of each gate we find. For example, the circuit for the inverse QFT on 3 qubits is shown in *Figure 6.5*. Notice that the inverse of $P(\theta)$ is $P(-\theta)$, while the H and SWAP gates are their own inverses.

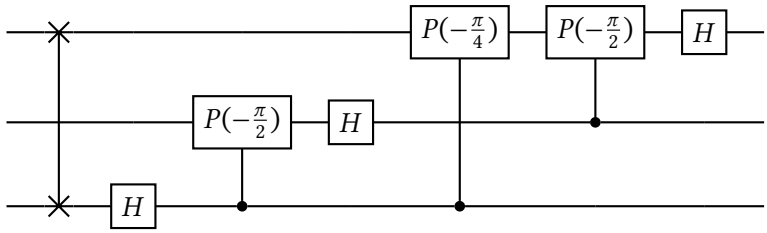

Figure 6.5: Circuit for the inverse quantum Fourier transform on 3 qubits

When designing a quantum oracle to minimize a function g, our goal will be to perform the computation in such a way that the $g(x)$ values appear as exponents in the amplitudes of our states so that we can later recover them by means of the inverse QFT. This may sound like a difficult endeavour, but as we will show in the following subsections, we already have all the tools that we need in order to succeed. We will start by showing how to encode integer values in exactly the way that we require.

6.2.2 Encoding and adding integer numbers

As will become apparent soon in this section, the most convenient way of working with integer numbers in the context of GAS oracles is using their **two's complement** representation. In it, we can encode numbers from -2^{m-1} to $2^{m-1} - 1$ by using m-bit strings.

Positive numbers are represented in the usual way for binary numbers, but a negative number x is represented by $2^m - x$.

For instance, if $m = 4$, we represent 3 by 0011, and -5 by 1011 (which is the binary representation of $11 = 16 - 5$). One advantage of this representation is that the most significant bit indicates the sign of the encoded number: positive numbers always start with 0, while negative numbers start with 1.

Another perk of two's complement representation is that, with it, we can compute additions involving both positive and negative numbers by simply performing regular binary addition and discarding the last carry-out, if it exists. For instance, if we add 0011 (which is 3) and 1011 (which is -5), we obtain 1110 which is, indeed, the encoding of -2 (because $14 = 16-2$). Similarly, if we add 0110 (which is 6) and 1100 (which is -4) we obtain 0010 (after discarding the last carry-out), which is 2, as expected. These facts about two's complement arithmetic will be very helpful in implementing our quantum oracle, as we show next.

Exercise 6.3

Using two's complement with 5 qubits, represent 10 and -7 and perform their addition.

As we have mentioned in the previous subsection, when computing $g(x)$ with an oracle, we are interested in obtaining the state

$$\frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} e^{\frac{2\pi i g(x) k}{2^m}} |k\rangle$$

so that we can then apply the inverse QFT to get $|g(x)\rangle$. We will achieve this step by step.

Notice that $g(x)$ is always a sum of products of integer values. So, let's first deal with integer addition, and leave multiplication for the next subsection.

Following the notation of [51], we will call the state

$$\frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} e^{\frac{2\pi i j k}{2^m}} |k\rangle$$

the **phase encoding** of j. Then, for our purposes, it is enough to be able to know how to prepare the phase encoding of 0 and to know how to add a given integer l to the phase encoding of any other integer. In that way, we can start from 0 and add all the terms in the polynomial expression of g one by one.

Preparing the phase encoding of 0 could not be easier. We just need to apply the Hadamard gate to each and every qubit that we are using to represent the integer values. In this way, we will obtain the state

$$\frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} |k\rangle = \frac{1}{\sqrt{2^m}} \sum_{k=0}^{2^m-1} e^{\frac{2\pi i 0 k}{2^m}} |k\rangle ,$$

which is, indeed, the phase encoding of 0.

Suppose now that we have a state that phase-encodes j and we want to add l to it. We first assume that l is non-negative and, later, we will deal with negative numbers. To add l in phase encoding, we just need to apply the gates shown in *Figure 6.6*.

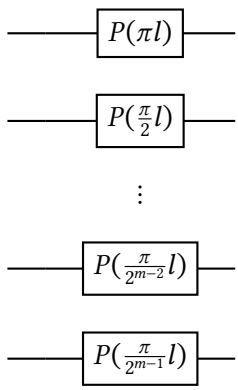

Figure 6.6: Circuit for adding l to a state in phase encoding when we have m qubits

Indeed, when we apply those gates to a basis state $|k\rangle$, we obtain $e^{2\pi ikl/2^m}|k\rangle$. To prove it, just notice how, if the h-th qubit of $|k\rangle$ is 1, the circuit of *Figure 6.6* adds a phase of value $e^{\pi il/2^h} = e^{2^{m-h}\pi il/2^m}$ (we start counting qubits from 0) and no phase otherwise. When we sum all these phases over the qubits of $|k\rangle$ that have value 1, we obtain exactly $e^{2\pi ilk/2^m}$. Thus, by linearity, when we apply the circuit to the phase encoding of j, we get

$$\frac{1}{\sqrt{2^m}}\sum_{k=0}^{2^m-1} e^{\frac{2\pi ijk}{2^m}} e^{\frac{2\pi ilk}{2^m}} |k\rangle = \frac{1}{\sqrt{2^m}}\sum_{k=0}^{2^m-1} e^{\frac{2\pi i(j+l)k}{2^m}} |k\rangle\,,$$

which is the phase encoding of $j + l$, as desired.

So, this works beautifully for non-negative numbers. But, what about negative ones? It turns out that, if l is negative, we can again use the very circuit in *Figure 6.6* — no further adjustments required. The key observation is that, for any integer $0 \leq h \leq m-1$, it holds that

$$e^{\frac{\pi i(2^m+l)}{2^h}} = e^{\frac{\pi il}{2^h}} e^{\frac{\pi i 2^m}{2^h}} = e^{\frac{\pi il}{2^h}} e^{\pi i 2^{m-h}} = e^{\frac{\pi il}{2^h}},$$

because $m - h > 0$, making 2^{m-h} even and implying $e^{\pi i 2^{m-h}} = 1$.

This means that if we plug in l or $2^m + l$ in the gates of *Figure 6.6*, we obtain exactly the same circuit. Thus, we can work with the two's complement representation of l instead of l and the results for the addition that we proved previously for non-negative integers will also hold for negative integers. The only concern could be that, when adding in two's complement a positive and a negative number, we get a carry-out (like, for instance, when we added 6 and -4 in a previous example). However, in that case, the carry-out will give us an even power of two and, again, the corresponding phase will be 1, leaving the result unchanged. Effectively, we are performing arithmetic modulo 2^m, so we are safe. Notice, nevertheless, that, if we add two positive or two negative integers and we get a carry-out, then we will get a wrong result — in this case, modular arithmetic turns against us!

Important note

You always need to use a number of qubits that is large enough to represent, in two's complement, any integer number that may arise from the computations. If you are working with a polynomial $g(x)$, you can simply add up the absolute value of all the coefficients in $g(x)$ to obtain a constant K. Then, you can choose any m such that $-2^{m-1} \leq -K \leq K \leq 2^{m-1} - 1$. If you want to be even more precise, you can select K as the maximum between the sum of all positive coefficients and the sum of the absolute value of all the negative coefficients.

As an example, in *Figure 6.7*, we present a circuit that prepares the phase representation of 0, adds 3 to it, and then adds -5 (or, equivalently, subtracts 5). Notice that some of the gates could be simplified. For instance, $P(3\pi)$ is just $P(\pi)$. We could also merge consecutive P gates into single gates by adding their angles together (for instance, $P(-5\frac{\pi}{2})P(3\frac{\pi}{2}) = P(-2\frac{\pi}{2}) = P(\pi)$). For the sake of clarity, throughout this section, we will keep the gates in their original form, without any simplification.

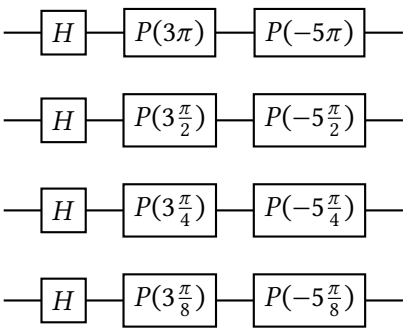

Figure 6.7: Circuit for preparing the phase representation of 0*, adding* 3 *to it and then subtracting* 5

Exercise 6.4

Derive a circuit that prepares the phase representation of 0, adds 6 to it and then subtracts 4. Use 4 qubits.

We now have the first ingredient that we need in order to compute the $g(x)$ polynomial: adding integers in phase encoding. In the next subsection, we will learn how to deal with the product of binary variables.

6.2.3 Computing the whole polynomial

You may be tempted to think that performing the multiplications that we need to compute our polynomial $g(x)$ will be much harder than performing the additions. But not quite! Let's look into this.

All the variables that we are considering are binary, and this means that, when we perform a multiplication such as x_0x_1, we always obtain either 0 or 1 as a result. Thus, if $g(x)$ is, for example, $3x_0x_1 - 2x_1x_2 + 1$, we will need to add 1 always (because it is the independent term and, as the name suggests, does not depend on the value of the variables), but we will only need to add 3 when both x_0 and x_1 are 1 and we will only need to subtract 2 when both x_1 and x_2 take value 1.

Does this sound familiar? Well, it should, because these computations that we have described correspond, precisely, to the application of controlled operations. Therefore, in order to calculate the contribution of a term such as $3x_0x_1$, we can use the circuit that we derived in the previous subsection to add 3 in phase encoding, but with each gate controlled by both x_0 and x_1. Notice that there is nothing special in using just two qubits as the controls, so we could also consider polynomials with terms such as $-2x_0x_2x_4$ or $5x_1x_2x_3x_5$.

To better illuminate these techniques, in *Figure 6.8*, we show a circuit that computes $3x_0x_1 - 2x_1x_2 + 1$. The first column of gates prepares the phase encoding of 0. The second one adds the independent term of the polynomial. The next one adds 3, but only if $x_0 = x_1 = 1$ (that is why all the gates are controlled by the $|x_0\rangle$ and $|x_1\rangle$ qubits). Similarly, the last column subtracts 2, but only when $x_1 = x_2 = 1$.

There are a couple of technical details to discuss about the circuit in *Figure 6.8*. First, we have adopted the usual convention of setting all the one-qubit gates that are controlled by the same qubits in a single column. In fact, we could consider them as a single multi-qubit

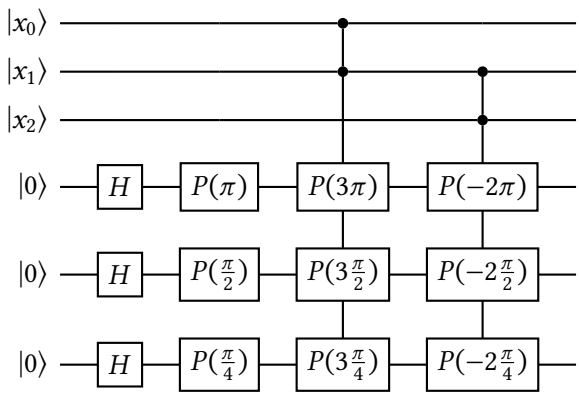

Figure 6.8: Circuit for computing $3x_0x_1 - 2x_1x_2 + 1$ in phase encoding

gate, but in some quantum computers you may need to separate them and apply them in sequence (in any case, this is something that the transpiler should take care of, don't worry). Also, notice that these gates are multi-controlled, but — using techniques like the ones described in *Section 4.3* of [16] — you can transform them into a combination of one- and two-qubit gates with Toffoli gates, which, in turn, can be decomposed into just one- and two-qubit gates.

Exercise 6.5

Design a circuit for computing $x_1x_2 - 3x_0 + 2$ in phase encoding. Use multi-qubit and multi-controlled gates.

So now we know how to compute, in phase encoding, the values of polynomials on binary variables with integer coefficients. But what about the case in which the coefficients are real numbers? We have two options to deal with that situation. The first one is to approximate them by using fractions with the same denominator. For instance, if your coefficients are 0.25 and -1.17, you can represent them by $25/100$ and $-117/100$. Then, you can multiply the whole polynomial by 100 without changing the variable values at which the minimum is attained and work with 25 and -117, which are integers. The other option is to use

the real numbers directly in the encoding. For instance, in the circuit of *Figure 6.6*, you would use l even if it is not an integer. In this case, you will work with a superposition of approximations of the real coefficient, with the better approximations having the larger amplitudes (see the discussion in [50] for all the details).

This completes our discussion on how to compute, in phase encoding, the value of any polynomial on binary variables. However, we are not quite done yet! In the next subsection, we will use our newly-acquired knowledge to finally implement the oracles that we need for the GAS algorithm.

6.2.4 Constructing the oracle

So far in this section we have covered a lot of ground. However, we should not forget what our final goal is: we want to implement an oracle that, given x and y, returns whether $g(x) < g(y)$ or not. This is what we need in order to use the **Dürr-Høyer** algorithm to find a minimum of g. In the previous subsection, we showed how to build a circuit that, given x, computes $g(x)$ in phase encoding. For the sake of simplicity, in the circuits that we will use in this subsection, we will denote the sequence of gates that implements $g(x)$, excluding the initial column of H gates, by just a big box with $g(x)$ inside. In a similar way, when we need to use the QFT or its inverse, we will use a box labeled QFT or $\text{QFT}^\dagger$.

Using this notation, an oracle to determine whether $g(x) < g(y)$ can be implemented by using the circuit depicted in *Figure 6.9*.

Let's explain bit by bit the elements of the circuit. First, notice that the upper qubits are reserved for the inputs x and y and, consequently, are registers of n qubits each. Next, we have m auxiliary qubits that we will use to compute the values of the polynomials (as we mentioned previously, you need to select m so that it is big enough to store all the intermediate results). Finally, the bottom qubit will store the result of checking whether $g(x) < g(y)$.

From what we have studied in this section and under the assumption that all the coefficients in g are integers, we know that the state just before the CNOT gate is $|x\rangle\,|y\rangle\,|g(x) - g(y)\rangle\,|0\rangle$.

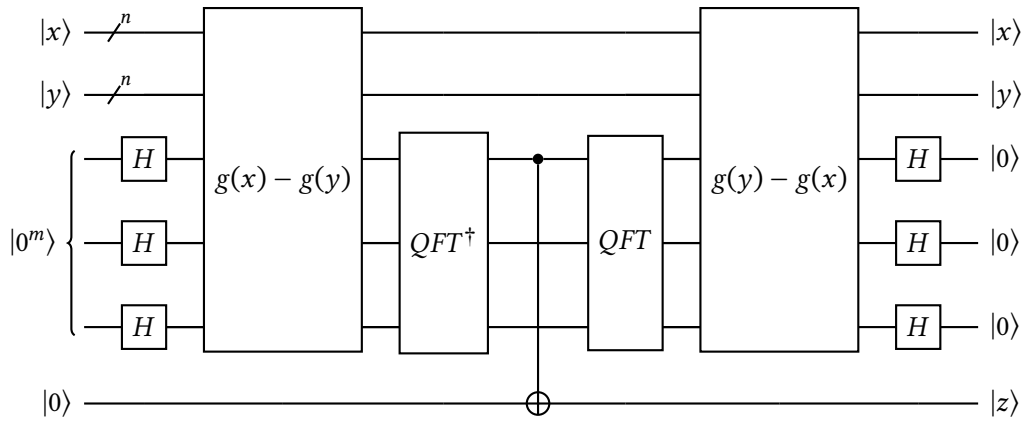

Figure 6.9: Oracle to determine whether $g(x) < g(y)$

Now, if $g(x) < g(y)$, then $g(x) - g(y) < 0$ and the most significant bit of $g(x) - g(y)$ will be 1, because we are working with two's complement representation. Thus, when we apply the CNOT gate, we will set the bottom qubit to $|1\rangle$ if $g(x) < g(y)$, and we will leave it in state $|0\rangle$ otherwise. This is the value that we will denote $|z\rangle$.

It would be natural to think that we could end the circuit after applying the CNOT gate. After all, we have already computed the result that we needed: z will be 1 if $g(x) < g(y)$ and it will be 0 otherwise. Nevertheless, we need to set the m auxiliary qubits back to $|0\rangle$. This is the value that is expected for the correct behaviour of the subsequent applications of the oracle (remember that we are using Grover's algorithm, so there will be several repetitions of the oracle circuit). What is more, we also need to set these qubits back to $|0\rangle$ to disentangle them from the rest of the qubits in the circuit. If they remain entangled, they may prevent the rest of the circuit from working correctly.

The process of setting the qubits back to $|0\rangle$ is known as **uncomputation** and it is a very important technique in many quantum algorithms. Since all quantum gates are reversible, we cannot just "erase" the content of some qubits (that would be extremely irreversible, because we would be forgetting the original values and it would be impossible to restore them). We need to perform the same computations that we carried out, but in reverse: hence

the name "uncomputation." In our case, we use the QFT to go back to phase encoding and then we add $g(y)-g(x)$, which, of course, is the inverse of adding $g(x)-g(y)$. Consequently, after the $g(y) - g(x)$ gate, the auxiliary qubits contain the phase encoding of 0 and, when we apply the column of H gates, we obtain $|0\rangle$, as desired.

We have, finally, completed our construction of the oracle that we need for GAS. However, there are a couple of additional details that may be useful in practice. On the one hand, notice that, in each application of Grover's algorithm in GAS, the value of y is fixed (it is x_0, the best solution that we have found by then). Thus, we can simplify the design of the oracle in *Figure 6.9* by eliminating the qubits reserved for $|y\rangle$, computing $g(x_0)$ with a classical computer, and using $g(x) - g(x_0)$ and $g(x_0) - g(x)$ in the gates that compute the values of the polynomial.

On the other hand, using techniques similar to the ones that we have studied in this section, we can create oracles to check whether polynomial constraints are met or not. For instance, if one of the constraints in our problem is $3x_0 - 2x_0x_1 < 3$, we can easily adapt our oracle construction to check whether that condition is met. Thus, we do not always need to transform our optimization problems into a pure QUBO form, but we can keep (some of) the constraints and check them directly. This might be more convenient than working with penalty terms in some cases.

But enough of theoretical considerations for now. In the next section, we will explain how to use GAS in Qiskit in order to solve combinatorial optimization problems.

6.3 Using GAS with Qiskit

If you want to practice what you have learned in this chapter about Grover's search, the Dürr-Høyer algorithm, and the construction of oracles, you can try to implement your own version of GAS in Qiskit from scratch. It is not a difficult project and it can be very satisfactory. However, there is no need for that. In the Qiskit Optimization module, you can find a ready-to-use implementation of Grover Adaptive Search (we will be using **version 0.4.0** of the package). Let's see how to use it.

One additional advantage of working with Qiskit's GAS implementation is that it accepts the optimization problem format that we used with QAOA in *Section 5.2.2*. The simplest way of using it is by defining a QUBO problem like the one that we can create with the following piece of code:

```
from qiskit_optimization.problems import QuadraticProgram
qp = QuadraticProgram()
qp.binary_var('x')
qp.binary_var('y')

qp.minimize(linear = {'x':2,'y':2}, quadratic = {('x','y'):-3})

print(qp.export_as_lp_string())
```

The output of the execution is the following:

```
\ This file has been generated by DOcplex
\ ENCODING=ISO-8859-1
\Problem name: CPLEX

Minimize
 obj: 2 x + 2 y + [ - 6 x*y ]/2
Subject To

Bounds
 0 <= x <= 1
 0 <= y <= 1

Binaries
 x y
End
```

As you surely recognize, this is the type of problem that we have been extensively working with in the last few chapters. To solve it with GAS in Qiskit, we need to define a `GroverOptimizer` object as follows:

```
from qiskit_optimization.algorithms import GroverOptimizer
from qiskit import Aer
from qiskit.utils import algorithm_globals, QuantumInstance
seed = 1234
algorithm_globals.random_seed = seed
quantum_instance = QuantumInstance(Aer.get_backend("aer_simulator"),
    shots = 1024, seed_simulator = seed, seed_transpiler=seed)
grover_optimizer = GroverOptimizer(num_value_qubits = 3, num_iterations=2,
    quantum_instance=quantum_instance)
```

Notice that we have set seed values for reproducibility and we have created a quantum instance based on the Aer simulator. Of course, if you want to use a real quantum computer, you just need to create the quantum instance from one of the quantum devices, as we have seen in previous chapters. Then, we have defined a `GroverOptimizer` object that uses 3 qubits to represent the values of the polynomial (what we have denoted as m in the previous section) and that stops the execution if it has seen no improvement in 2 consecutive iterations (the `num_iterations` parameter). Notice that 3 qubits are enough to represent all the possible values of our polynomial in two's complement, but 2 qubits would be too few.

To use this `GroverOptimizer` object to solve our problem, we can run the following instructions:

```
results = grover_optimizer.solve(qp)
print(results)
```

This will give us the following output:

```
fval=0.0, x=0.0, y=0.0, status=SUCCESS
```

This is, indeed, the optimal solution to the problem, as you can check by trying the 4 possible options. That was easy, wasn't it?

Exercise 6.6

Write the code needed to use GAS in Qiskit to find the solution of the QUBO problem with binary variables x, y, and z and objective function $3x + 2y - 3z + 3xy$.

But what if you want to solve a more complicated problem? It turns out that the Grover Optimizer class also can work with problems with constraints. Imagine that we define a problem with the following instructions:

```
qp = QuadraticProgram()
qp.binary_var('x')
qp.binary_var('y')
qp.binary_var('z')
qp.minimize(linear = {'x':2}, quadratic = {('x','z'):1, ('z','y'):-2})
qp.linear_constraint(linear = {'x':2, 'y':-1, 'z':1},
    sense ="<=", rhs = 2)
print(qp.export_as_lp_string())
```

If we execute the code, we obtain the following output, which corresponds to a quadratic program with linear constraints:

```
\ This file has been generated by DOcplex
\ ENCODING=ISO-8859-1
\Problem name: CPLEX

Minimize
 obj: 2 x + [ 2 x*z - 4 y*z ]/2
Subject To
 c0: 2 x - y + z <= 2
```

```
Bounds
 0 <= x <= 1
 0 <= y <= 1
 0 <= z <= 1

Binaries
 x y z
End
```

We could create a `GroverOptimizer` object and directly use its `solve` method with `qp`. Then, the `GroverOptimizer` object will convert the constrained problem into a QUBO one and solve it. Easy peasy. However, there is a small problem: how can we know how many qubits we should use for the polynomial values? Since we don't know the penalty terms that will be introduced in the conversion, we don't know the coefficients of the polynomial. Of course, we could use a big enough value to be sure that there will be no problems, but that will make the execution slower, especially in the simulator. And if we use too few qubits, our results could be erroneous.

For that reason, we recommend converting the problem first into QUBO form and then solving it with GAS. In this way, we can more accurately determine the number of qubits that we need. For instance, for the problem that we have just defined, we can obtain the transformed QUBO problem with the following instructions:

```
from qiskit_optimization.converters import QuadraticProgramToQubo
qp_to_qubo = QuadraticProgramToQubo()
qubo = qp_to_qubo.convert(qp)
print(qubo.export_as_lp_string())
```

The output is the following:

```
\ This file has been generated by DOcplex
\ ENCODING=ISO-8859-1
```

```
\Problem name: CPLEX

Minimize
 obj: - 46 x + 24 y - 24 z - 24 c0@int_slack@0 - 48 c0@int_slack@1 + [ 48 x^2
      - 48 x*y + 50 x*z + 48 x*c0@int_slack@0 + 96 x*c0@int_slack@1 + 12 y^2
      - 28 y*z - 24 y*c0@int_slack@0 - 48 y*c0@int_slack@1 + 12 z^2
      + 24 z*c0@int_slack@0 + 48 z*c0@int_slack@1 + 12 c0@int_slack@0^2
      + 48 c0@int_slack@0*c0@int_slack@1 + 48 c0@int_slack@1^2 ]/2 + 24
Subject To

Bounds
 0 <= x <= 1
 0 <= y <= 1
 0 <= z <= 1
 0 <= c0@int_slack@0 <= 1
 0 <= c0@int_slack@1 <= 1

Binaries
 x y z c0@int_slack@0 c0@int_slack@1
End
```

As you can see, this is now a bona fide QUBO problem. Moreover, by inspecting the polynomial coefficients, we can notice that 10 qubits, for instance, are enough to store the polynomial values. Thus, we can solve the problem with the following piece of code:

```
grover_optimizer = GroverOptimizer(10,
    num_iterations=4, quantum_instance=quantum_instance)
results = grover_optimizer.solve(qubo)
print(results)
```

If we run it, we obtain the following, which is indeed the solution to the problem:

```
fval=-2.0, x=0.0, y=1.0, z=1.0, c0@int_slack@0=0.0, c0@int_slack@1=1.0,
status=SUCCESS
```

However, this involves the slack variables used in the transformation. If you don't want to see them, you can alternatively run GAS on the original problem, now that we know how many qubits to use:

```
grover_optimizer = GroverOptimizer(10, num_iterations=4,
    quantum_instance=quantum_instance)
results = grover_optimizer.solve(qp)
print(results)
```

In this case, the output is the following:

```
fval=-2.0, x=0.0, y=1.0, z=1.0, status=SUCCESS
```

This is exactly the same solution that we obtained with the transformed problem, but now without the slack variables.

This is all you need to know if you want to use GAS in Qiskit. In the next chapter, we will study the **Variational Quantum Eigensolver**, a generalization of QAOA that will allow us to solve many interesting optimization problems.

Summary

In this chapter, we have learned about Grover's search algorithm and how it can be adapted to find minima of functions with the Dürr-Høyer algorithm. We have also learned about quantum oracles and their role in these two methods.

After that, we learned how to perform arithmetic in phase encoding and how to retrieve the results by using the mighty Quantum Fourier Transform. We also studied how to use all these techniques to implement oracles that can be used in Grover's Adaptive Search to solve combinatorial optimization problems.

Finally, we also learned how to use GAS with Qiskit to obtain solutions of both QUBO problems and constrained quadratic programs.

Now, get ready for the next chapter: we will be studying the Variational Quantum Eigensolver and some of its most important applications!