
Application of the Quantum Approximate Optimization Algorithm to the MaxCut problem

Quantum Information Project, AP3421-PR, Q2 2019/20

Santiago Vallés-Sanclemente (5041023),
Isacco Gobbi (5161347), Olexiy Fedorets (5152658)
14th February 2020

Contents

1	Introduction	1
2	QAOA	2
3	The MaxCut Problem	3
4	Implementation	4
4.1	Circuit in Qiskit	4
4.2	Code structure	5
4.3	The classical optimizer — <i>Differential Evolution</i>	7
4.4	Backends	8
5	Results	9
5.1	<i>Star</i> graph	9
5.2	<i>Butterfly</i> graph	10
5.3	<i>V9E15</i> graph	13
5.4	Algorithm runtime and performance	15
6	Conclusion	15
	Appendix A Additional results	19
	Appendix B Source code	19

List of Figures

1	A diagram of the full QAOA algorithm (taken from [9]).	3
2	Combination of gates that form the layer unitaries.	5
3	Geometry of IBM's Yorktown Quantum Computer	8
4	Star Graph for 4 nodes and its associated circuit	9
5	Solution for the MaxCut problem for the Star graph using the QAOA algorithm. (a) to (c) show the measurement outcomes of the quantum circuit using different layers (p). (d) to (f) show a histogram with the rounded cost function associated to each measurement outcome. The vertical red lines indicates the average of this value.	10
6	Average cost function measured for different 1-qubit and 2-qubits errors. .	11
7	Butterfly Graph	11
8	Measurement outcomes from the optimized quantum circuit.	12
9	Cost function associated to each measurement outcome. The red vertical line indicates the average.	13
10	The V9E15 graph with the maximum cut which cuts all 15 edges.	14
12	An example of the QAOA circuit with $p = 2$ layers executed on IBMQ's <i>yorktown</i> 5-qubit quantum processor to solve the MaxCut problem for the Butterfly graph.	19

Listings

1	<code>Graph.py</code> — The main file containing the <code>Graph</code> class. 192	<code>main.py</code> — The file from which the <code>Graph</code> is created and the algorithm is executed. 26
---	--	--

1 Introduction

..this is the first time that a quantum algorithm has been proved to achieve a better approximation ratio for a natural NP-hard optimization problem than the best known classical algorithm achieves.
— Scott Aaronson [1]

The Quantum Approximate Optimization Algorithm (QAOA) was proposed E. Farhi and J. Goldstone in 2014 [2] with the goal of approximating solutions to NP-hard combinatorial optimization problems better than what is possible with classical algorithms. A combinatorial optimization problem is defined by finding an object from a finite set such that it is optimal with respect to a given cost function. An algorithm to solve such problems should be able to provably approximate such solution close to the optimum.

In fact, QAOA has potential for a superpolynomial speedup [3] and proved to achieve a better approximation ratio than a classical algorithm [4], until the computer science community caught up with a better classical solution [5]. It is even proposed to be a potential candidate to establish quantum supremacy [6].

QAOA is a hybrid quantum-classical algorithm which combines the preparation of a quantum state with the execution of classical optimization algorithm, similar to the *Variational Quantum Eigensolver* (VQE) [7]

The reason this algorithm is particularly interesting at this time is that it is suitable for *noisy intermediate-scale quantum computers* (NISQ), a term coined by [8], which is the first stage of quantum computers the field is currently heading towards. Algorithms for NISQ are required to have low circuit depth (because of limited gate fidelity) and use a small number of qubits (commonly defined as below 100). As we will see later this algorithm only requires one qubit per variable in the given problem, which already sets a bar hard to be beaten.

A possible problem one could imagine to solve by this algorithm is the time slot occupation problem. Here we have a certain period of time (say 24 hours), divided in slots (of say 4 hours), which we want to occupy by workers who each have specific time slots assigned to them. The optimization problem is to fill as much of the time period as possible (ideally all of it) such that each time slot is occupied by a worker, and ideally only by one.

Another famous application, the *MaxCut* problem, will be introduced in detail in [Section 3](#) and will be analyzed in detail throughout this work.

The main goal of this project is to solve the MaxCut problem for several different graphs by QAOA not only on a simulator but also on real quantum processors, thereby

analyzing its stability in the presence of noise in real devices.

2 QAOA

QAOA aims to find an approximate solution to a combinatorial optimization problem. These kind of problems can be stated as finding the optimal solution from a finite (but extremely large) set of possibilities. Optimality in this context is defined with respect to some target function $C(z)$ of an n -bit string $z \in \{0, 1\}^n$ which needs to be maximized or minimized. In this context we will always treat the optimization as a maximization problem. As any approximation algorithm, QAOA returns a solution that is probably close to optimal, normally within some threshold. More specifically, for an approximate solution z^* : $C_{OPT} \geq C(z^*) \geq \alpha C_{OPT}$

where C_{OPT} is the cost associated with the optimal solution and alpha $\alpha \in [0, 1]$ defines the approximation ratio for the algorithm.

In general the cost function $C(z)$ consists of m clauses which can or cannot be satisfied by the input string z . The cost function can be expressed as:

$$C(z) = \sum_{k=1}^m C_k(z) \quad \text{with} \quad C_k(z) = \begin{cases} +1, & \text{if } z \text{ satisfies clause } k \\ 0, & \text{if } z \text{ does not satisfy clause } k \end{cases} \quad (1)$$

Since QAOA is a hybrid quantum-classical algorithm, $C(z)$ is translated into a cost hamiltonian \mathcal{H} . Any a candidate solution $z \in \{0, 1\}^n$ is mapped into an $n - bit$ string in which each binary variable is represented by a qubit.

The first step of the algorithm is to produce an equal superposition of all n qubit states, i.e. prepare $|+\rangle^{\otimes n}$ [2]. This state is then operated on by p equal layers of gates. Each layer is composed of a *Cost unitary*

$$\hat{U}_C(\gamma) = \exp(-i\gamma C(z)), \quad \gamma_i \in [0, 2\pi] \quad (2)$$

and a *Mixing unitary*

$$\hat{U}_B(\beta_i) = \exp\left(-i\beta \sum_{j=1}^n \hat{X}_j\right), \quad \beta_i \in [0, \pi] \quad (3)$$

where $i \in [1, \dots, p]$ is the layer index. Each layer involves two parameters γ_i and β_i . These represent the duration of the application of the gates. The mixing layer uses \hat{X} -gates (which are non-diagonal in Z-basis) and thus rotates all qubits around the x-axis by an angle β , which allows the exploration of a larger part of the Hilbert space.

In general different layers will have different values of γ and β , therefore for a p layer

circuit we will have in total $2p$ control parameters, namely $\boldsymbol{\beta} = (\beta_1, \dots, \beta_p) \in [0, \pi]^p$, $\boldsymbol{\gamma} = (\gamma_1, \dots, \gamma_p) \in [0, 2\pi]^p$. The prepared state after execution of p gate-layers will then be in the form

$$|\psi(\boldsymbol{\beta}, \boldsymbol{\gamma})\rangle = \hat{U}_B(\beta_p)\hat{U}_C(\gamma_p) \dots \hat{U}_B(\beta_1)\hat{U}_C(\gamma_1) |+\rangle^{\otimes n} \quad (4)$$

The next step is to measure all qubits in computational basis to obtain the expectation value $\langle\psi(\boldsymbol{\beta}, \boldsymbol{\gamma})|C(z)|\psi(\boldsymbol{\beta}, \boldsymbol{\gamma})\rangle$. This encodes the cost function and needs to be maximized, which is performed by feeding the result into a classical optimizer, which returns updated values of the circuit parameters $(\boldsymbol{\beta}, \boldsymbol{\gamma})$. This step will be discussed in detail in [Section 4.3](#) as it is not part of the quantum algorithm. The measurement which gives the highest cost is taken as the solution.

It is shown in [2] that it is guaranteed to find the global maximum of $\langle C(z) \rangle$ for $p \rightarrow \infty$.

A schematic diagram of the full circuit is represented in [Fig. 1](#). An example of a circuit we executed is shown in the appendix [Fig. 12](#).

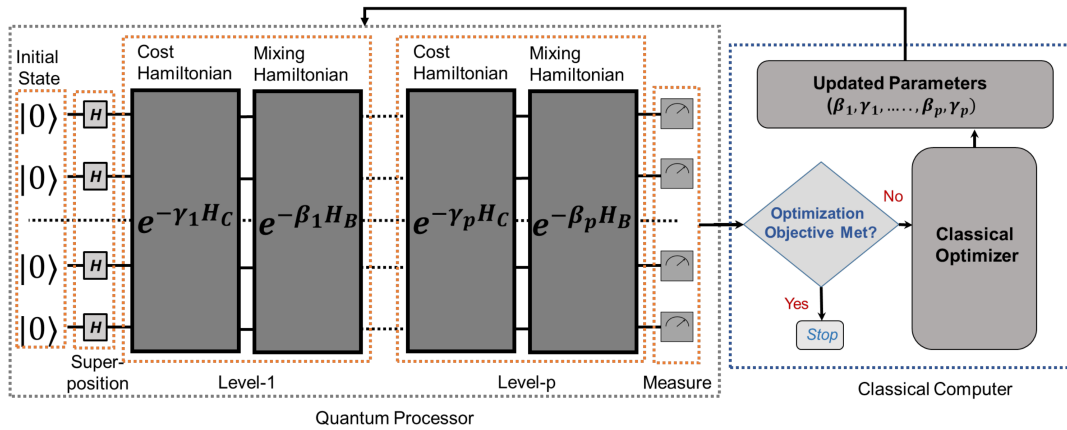


Figure 1: A diagram of the full QAOA algorithm (taken from [9]).

3 The MaxCut Problem

A graph G is defined as a pair $G=(V,E)$ where V is the set of vertices (or nodes) of the graph and E is the set of edges between the nodes. A *cut* is a partition of the vertices set V into two disjoint subsets. Any edge connecting two nodes in the different subsets is said to *cross* the cut and the number of crossed edges determines the *size* of the cut. For any given graph G the **MaxCut** problem consists in finding a cut of maximum size, i.e. a cut that crosses the greatest number of edges.

In general for an n -node graph we can represent any cut in the following way: we assign to each node a value $z_i = \pm 1$: hence node i and node j will belong to the same subset only if they have the same value. Therefore the n -bit binary string $Z = \{z_1, \dots, z_n\}$

with $z_i = \pm 1$ represents the cut in which all nodes with $z_i = +1$ belong to one subset and nodes with $z_i = -1$ to the complementary subset.

It is natural to define the cost function such that given an input string Z (i.e. a cut) the output will be equal to the number of crossed edges. Realizing that partitioning a graph is equivalent to choosing a spin configuration for the Ising model we write the Ising hamiltonian:

$$H(z) = - \sum_{(i,j) \in E} J_{ij} z_i z_j$$

Splitting the sum onto the two subsets, labeled V^+ and V^- and $\delta(V^+)$ the set of edges that connect the two sets

$$\begin{aligned} H(z) &= - \sum_{ij \in E(V^+)} J_{ij} - \sum_{ij \in E(V^-)} J_{ij} + \sum_{ij \in \delta(V^+)} J_{ij} \\ &= - \sum_{ij \in E(G)} J_{ij} + 2 \sum_{ij \in \delta(V^+)} J_{ij} \\ &= N + 2 \sum_{ij \in \delta(V^+)} J_{ij} \end{aligned}$$

Where N is the total number of edges in the original (uncut) graph. The last term on the right is two times the cost hamiltonian, therefore, rearranging the equation we have:

$$C(z) = \frac{1}{2} \sum_{ij \in E} (1 - z_i z_j) \quad (5)$$

setting $J_{ij} = 1$ we obtain the cost function.

This can easily be translated into the cost hamiltonian substituting the bit values z with single qubit gates:

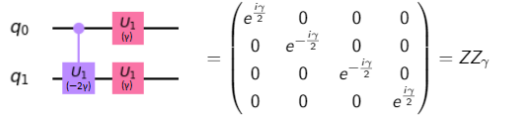
$$\hat{\mathcal{H}} = \frac{1}{2} \sum_{(i,j) \in E} (1 - \hat{Z}_i \otimes \hat{Z}_j) \quad (6)$$

4 Implementation

In this section we explain how has the QAOA algorithm been implemented to solve the MaxCut problem for a given graph. The coding has been performed using the Qiskit module of Python 3.5.6. All the code used for this project can be found in Appendix B.

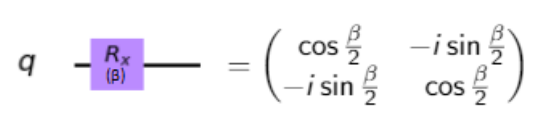
4.1 Circuit in Qiskit

As it was explained in section 2, the circuit associated to the QAOA is formed by a certain number of layers of gates. Each layer must contain a cost unitary and a mixing unitary



$$= \begin{pmatrix} e^{i\gamma/2} & 0 & 0 & 0 \\ 0 & e^{-i\gamma/2} & 0 & 0 \\ 0 & 0 & e^{-i\gamma/2} & 0 \\ 0 & 0 & 0 & e^{i\gamma/2} \end{pmatrix} = ZZ_\gamma$$

(a) Cost unitary for a single edge



$$= \begin{pmatrix} \cos \frac{\beta}{2} & -i \sin \frac{\beta}{2} \\ -i \sin \frac{\beta}{2} & \cos \frac{\beta}{2} \end{pmatrix}$$

(b) Mixing unitary

Figure 2: Combination of gates that form the layer unitaries.

that are defined in Equations (2) and (3). Each of these unitaries will be characterized by a parameter γ and β respectively. Considering the cost function for the MaxCut problem (Equation (5)), the cost unitary for a single edge will be equivalent to the combination of gates shown in Figure 2a. In Figure 2b we have shown the gate associated to the mixing unitary, which will be a rotation of angle β around the x axis.

4.2 Code structure

The central part of the code can be found in the file `Graph.py`. The file `main.py` is used to initialize the program and the file `Edges.txt` contains the information about the edges of the graph that we want to analyze.

The most important piece of the code is the class `Graph`, which contains many public and private methods that are described in the file `Graph.py`. When creating an instance of this class, we generate a graph with the edges specified in the file `Edges.txt` using the Python package `networkx`. In order to initialize this object, we must give as an argument the total number of vertices that form our graph. This way, it is possible to analyze graphs with isolated vertices, which will not appear in the file `Edges.txt`. The private methods `_Assign()` and `_Read_E()` are the ones responsible for these steps and they are automatically called when instancing the class `Graph`.

Once an instance of this class is generated, the public method `Optimizer` can be used to solve the MaxCut problem using the QAOA. The first parameter that this method needs specifies the total number of layers that we want to implement in the QAOA. The other parameters can be used to include noise in our model. We will go back to this later. As it was explained in section 2, in QAOA we need to classically optimize the expectation value of a certain state characterized by the parameters β and γ . For this reason, the method `Optimize()` includes the Differential Evolution (DE) optimization function from the SciPy package. The first argument sent to this method is the function that must be optimized, while the second one is a boundary for the parameters used to achieve this goal. The choice of this classical optimizer will be discussed in Section 4.3. `Optimize()` also includes measurement of the runtime the DE method.

The private method `_Simulate()` is the function that we need to optimize. This method receives the parameters β and γ as an array, the length of which depends on how many layers we have specified when calling the `Optimize()` method. `_Simulate()` uses IBMQ's QASM-simulator¹ to compute the expectation value of a certain state, which is the variable returned by this method and optimized by the DE function. This state is constructed as a quantum circuit in the private method `_Build`. This circuit has as many qubits as nodes the graph has and they are initialized in the $|+\rangle$ state using Hadamard gates. Then, as it was explained in section 4.1, a combination of `u1` and `cu1` gates characterized by the parameter γ is applied in those qubits between which there is an edge. Next, all qubits are rotated around the x-axis an angle β . A for loop is used to repeat these last two steps when we have more than one layer, using different β and γ parameters on each iteration. Once the state is prepared, it is measured in the computational basis and the process starts again until a total of 2048 shots have been simulated. From this data, the `_Simulate()` method is able to compute the expectation value, which is sent back to the DE optimization function. Once the highest expectation value is obtained, it is stored in the public variable `F`. `_Simulate()` also measures the runtime of the quantum circuit, for the case of simulation as well as execution on IBMQ hardware. The runtime results will be discussed in [Section 5.4](#).

As it was mentioned before, the last three arguments sent to the `Optimize()` method can be used to include noise in our simulation. This has been modeled as a 1-qubit depolarizing channel characterized by an error rate given by the third argument and a 2-qubit depolarizing channel with the error rate given by the fourth argument. The second argument is just a boolean variable that must be set to `True` when noise is included in the simulation. These depolarizing channels are associated to the 1-qubit and the 2-qubit gates in the `_Simulate()` method and all the measurements are performed taking them into account.

The class `Graph` also contains the following public methods, which use the `matplotlib` package to represent graphically the results obtained:

- `Plot_G()` : It combines the `matplotlib` and the `networkx` packages to produce a figure with the graph that is being analyzed.
- `Plot_C()` : It produces a figure with the quantum circuit that leads to a maximum in the expectation value. The plots include the optimized β and γ parameters.

¹IBMQ and Qiskit use the *Open Quantum Assembly Language*.

- `Plot_S()`: This method uses the measurement outcomes of the optimized quantum circuit to produce two different figures:
 1. A histogram that shows the raw measurement outcomes normalized to 1, which can be interpreted as the probability to obtain each outcome.
 2. A histogram showing the values obtained for the cost function for each measurement outcome. The average of these values (the public variable F) is also displayed as a red vertical line.

The file `main.py` can also be modified to graphically represent more results, such as the F variable as a function of the error rate in the simulation.

4.3 The classical optimizer — *Differential Evolution*

As seen in Fig. 1, a classical optimization step the end of each iteration of the algorithm, following the measurement of all qubits, is needed. For this step we employ global optimization by *Differential Evolution* (DE) first proposed by Storn and Price [10] and implemented in `scipy.optimize.differential_evolution`. This algorithm essentially applies genetic evolution to a continuous minimization problem, thereby not computing the gradient of the objective function but following a stochastic approach. This approach offers the advantage that the optimization is parallelizable, as the set of candidate parameters can in principle be evaluated simultaneously at each stage.

An initial set of parameters (a *population*) is chosen either at random or by an educated guess. The *fitness* of this parameter set is evaluated by calculating the cost function. In our case, this evaluation is done by the measurement of all qubits at the end of the quantum circuit.

Evolution is simulated by combining the values of existing parameters from the current population according to prefixed criteria (*mutation*, *recombination* etc.) to generate a new population, which will hopefully lead to better fitness. If members of the new population show an improvement (i.e. increased fitness) they are given more weight to serve as a model (*parent*) for the next population. The process is repeated until a tolerance level of the fitness is reached.

This leads to numerous execution of the quantum circuit with different (β, γ) , which requires significant total runtime, as the position in the IBMQ queue is reset after each time the circuit is run with a specific set of (β, γ) . As will be shown later in

It can be foreseen from the above description that the algorithm involves a large set of parameters, finding the optimal choice of which can be quite intricate. The effect of their choice can range from a fast convergence to the global optimum to no convergence

at all (i.e. an infinite loop). The tuning of those is a field of research by itself [11], and we will not go into detail here.

In the end, we found that the parameters which worked best for our type of problem are `updating='immediate', mutation=(0.5,1)2, recombination=0.9, tol=1, maxiter=500`. It is important to note that the chosen method for the classical optimization step is only one of many, there being no consensus in the community on the best choice yet [12]. It may even depend on the type of problem QAOA is applied to [9]. However the differential evolution method has shown satisfying results (despite its drawbacks) both in our as well as in previous studies [9].

4.4 Backends

In order to implement our algorithm it is necessary to access a backend that is able to either run or simulate a quantum circuit. For reasons that will be explained later, we have decided to use IBM's Yorktown 5 qubits Quantum Computer to obtain some of our results. The geometry of this hardware backend and some of its characteristics are depicted in Figure 3.

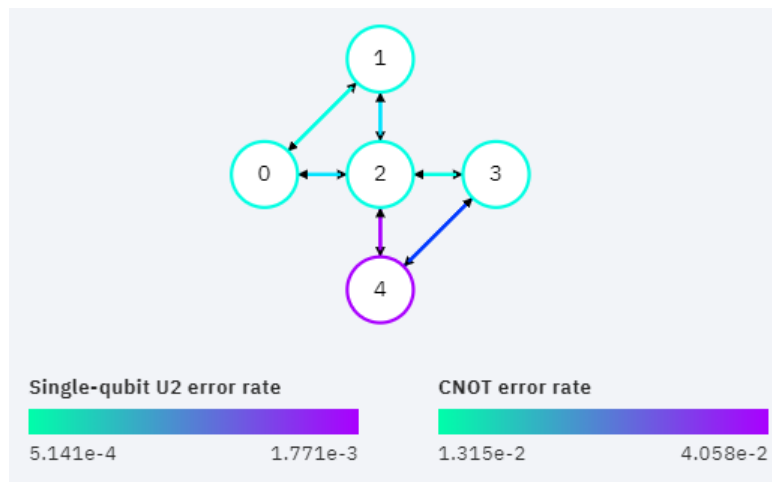


Figure 3: Geometry of IBM's Yorktown Quantum Computer

Most of our circuits have been run locally on Qiskit Aer's QASM simulator. As explained in section 4.2, our code allows us to include gate errors in the form of depolarizing noise. From now on we denote the 1-qubit error rate as ϵ_1 and the 2-qubit error rate as ϵ_2 . When obtaining our results, these will typically take values similar to the ones associated to the Yorktown quantum computer, shown in Figure 3.

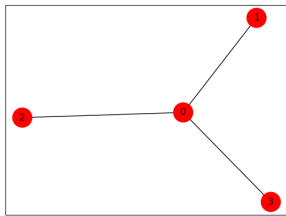
²This also enables a technique called *dithering*, which randomly changes the mutation rate within the given bounds.

5 Results

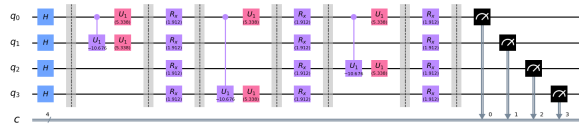
In this section we show the results that have been obtained using the code described in section 4.2. We have studied the MaxCut problem with QAOA for three different graphs: the Star graph the Butterfly graph and the V9E15 graph.

5.1 *Star* graph

The first graph that we have analyzed is the so-called Star Graph with 4 nodes. As it is shown in Figure 4a, this graph contains three links that connect three of its vertices (1,2,3) with the fourth one (0). Figure 4b shows the circuit associated to this graph using QAOA with a single layer.



(a) Star Graph with 4 nodes



(b) Single layer circuit for Star Graph

Figure 4: Star Graph for 4 nodes and its associated circuit

The QAOA for different number of layers has been run on this graph and the results obtained are shown in Figure 5. In Figures 5a to 5c we can see the measurement outcomes of the quantum circuit that has been optimized as explained in section 2. In these figures we can observe that the most likely measurement outcomes are 1000 and 0111. This indicates that the solution subgraphs are the one containing only vertex 0 and the one containing 1,2,3. By looking at Figure 4a we can check with bare eye that this is indeed the correct solution. Comparing the three figures we can observe that those measurement outcomes that do not correspond with a solution become suppressed when using a larger number of layers.

In Figures 5d to 5f we can observe that the average cost function (red line) approaches the maximum value when increasing p . This is also an evidence of the fact that QAOA improves when using a larger number of layers. A representation of the average cost function as a function of the number of layers can be found in the Appendix.

As explained in section 4.2, our code allows us to include noise in our simulation in the form of a depolarizing channel. In order to analyze the influence of this noise over our results, we have studied the dependence of the average cost function with the 1-qubit and the 2-qubit error rate. The results for single layer QAOA are shown on Figure 6, where

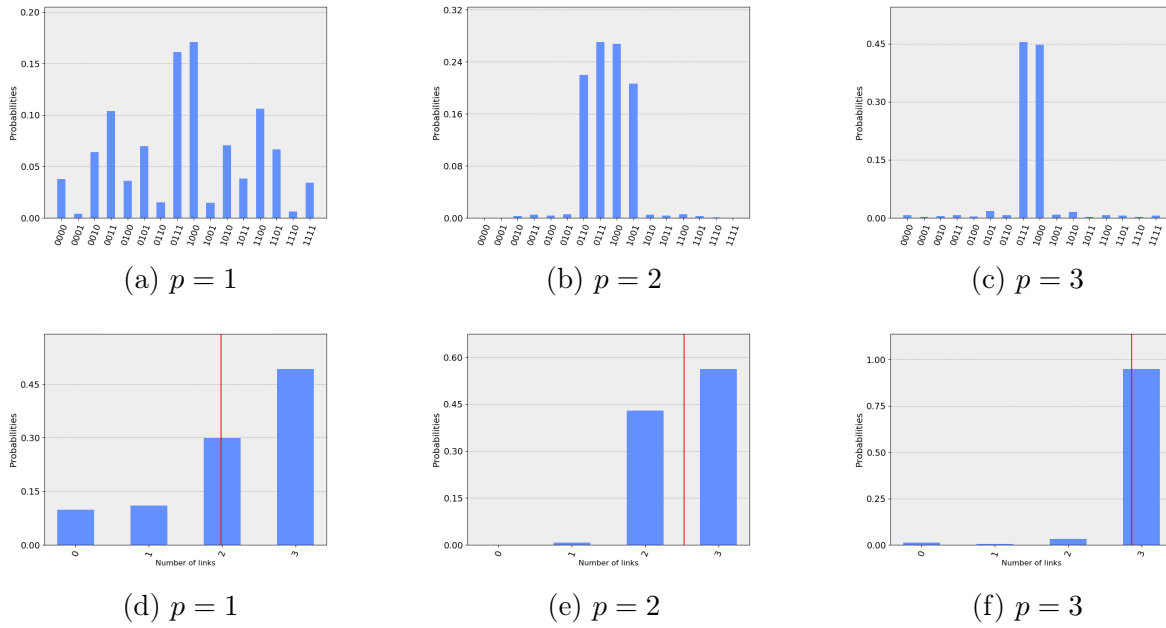


Figure 5: Solution for the MaxCut problem for the Star graph using the QAOA algorithm. (a) to (c) show the measurement outcomes of the quantum circuit using different layers (p). (d) to (f) show a histogram with the rounded cost function associated to each measurement outcome. The vertical red lines indicates the average of this value.

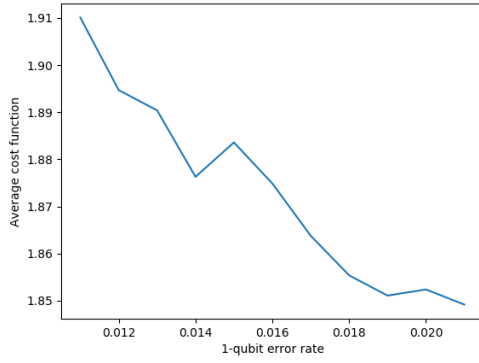
we can see that the average cost function decreases linearly with the noise. Since this is a slow decrease, we can then conclude that our algorithm is quite robust against this type of noise model.

5.2 *Butterfly* graph

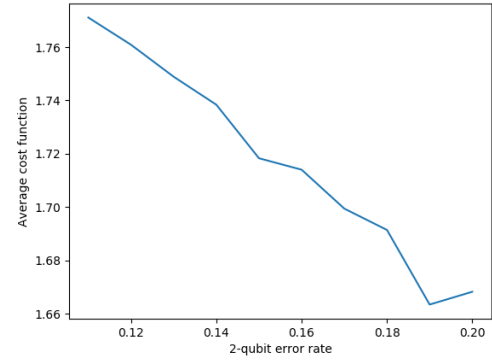
The Butterfly graph consists on five nodes connected by six edges as shown in Figure 7. This structure is especially interesting because the connectivity between qubits of IBM's Yorktown quantum computer follows the same geometry. For this reason, we have analyzed this graph using this quantum computer and we have compared our results with the ones obtained with the qasm simulator, with and without noise.

The QAOA has been applied to the MaxCut problem associated to this graph using 2 layers of gates. In Figure 8 we show the measurement outcomes for the optimized quantum circuit. In Figure 5a, the results obtained with the noiseless qasm simulator are depicted. We can appreciate the same symmetry between the measurement outcomes that was explained for the Star graph. From these results it is clear that there are a total of 18 possible solutions for the MaxCut problem for this graph.

This conclusion is less clear when including gate noise in the form of a depolarizing channel in our simulation, as shown in Figures 8b and 8c. In the former one, it is still possible to distinguish most of the correct measurement outcomes from the incorrect ones,



(a) For a 2-qubit error rate of 0.01



(b) For a 1-qubit error rate of 0.001

Figure 6: Average cost function measured for different 1-qubit and 2-qubits errors.

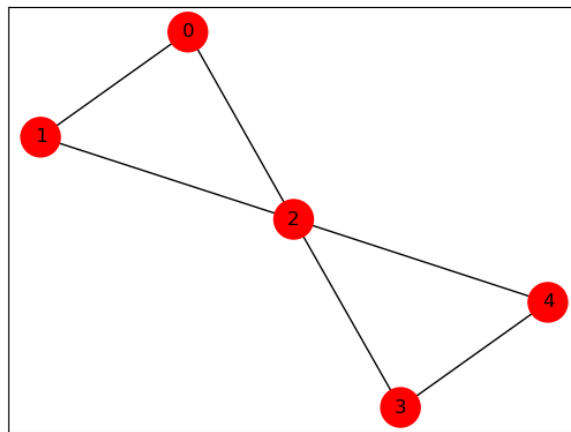


Figure 7: Butterfly Graph

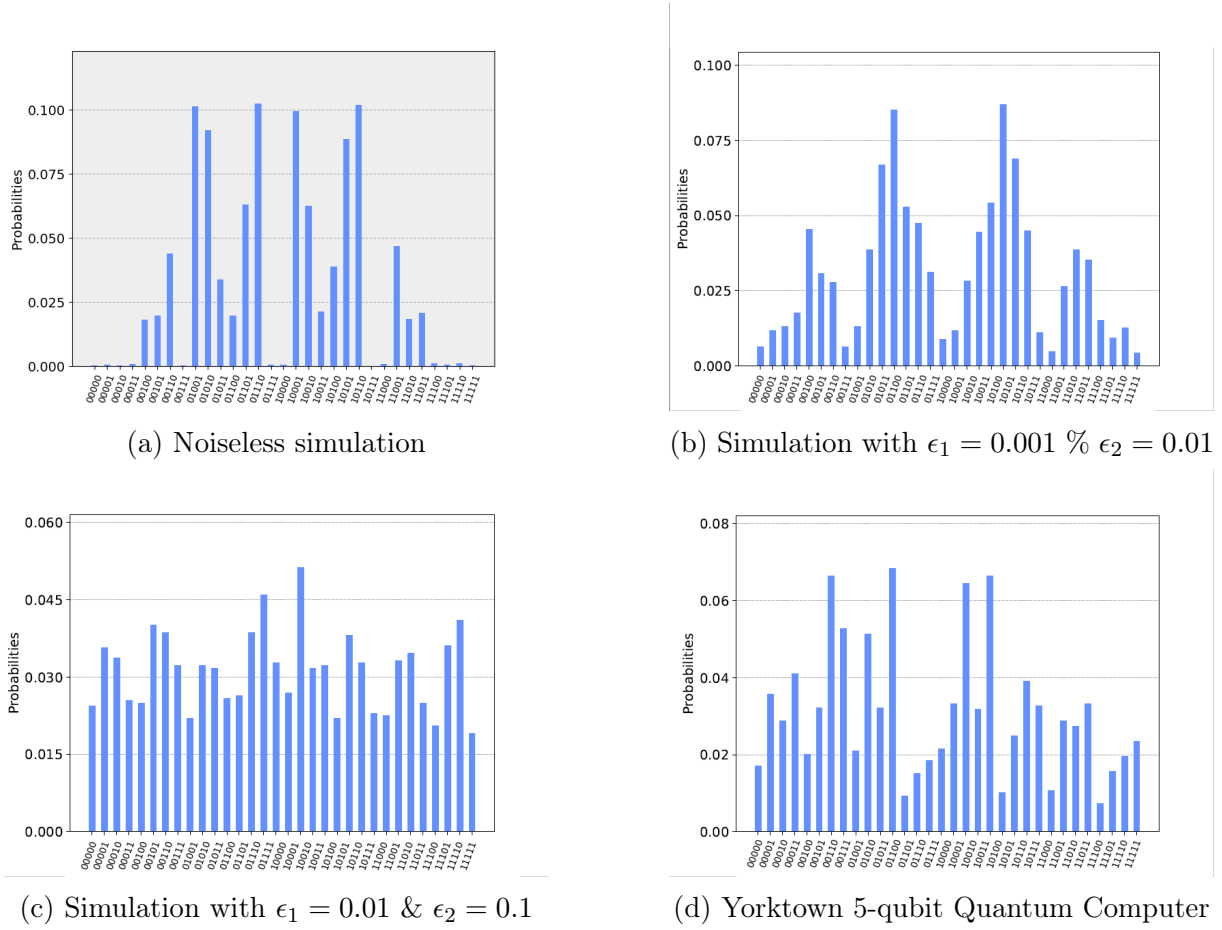
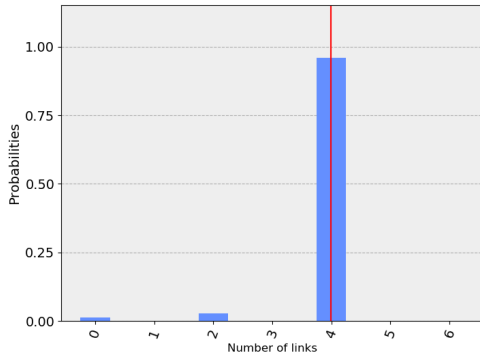


Figure 8: Measurement outcomes from the optimized quantum circuit.

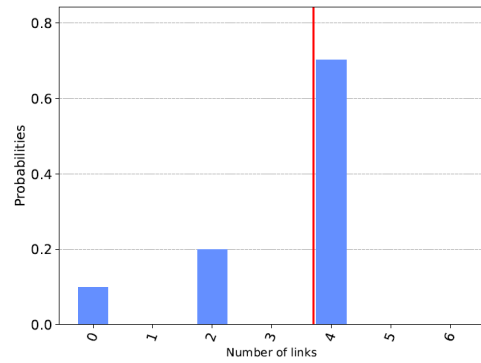
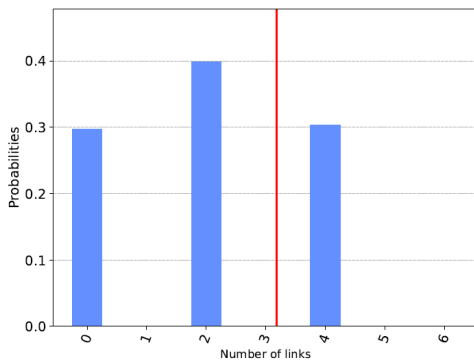
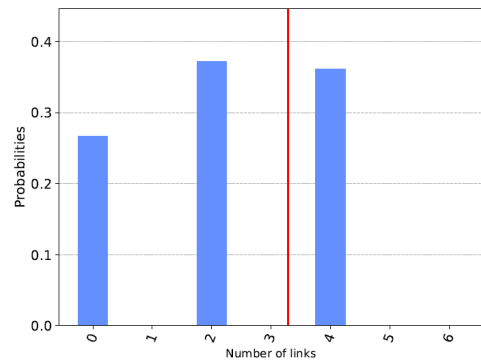
but in Figure 8c it is not possible anymore.

In Figure 8d we show the results obtained with the Yorktown quantum computer. Despite the results being quite noisy, it is still possible to clearly distinguish 4 of the 18 solutions, the ones with probability above 6%. But for the remaining outcomes it is not possible to determine whether they are solutions or not. It is also remarkable that due to the noise it is not possible to observe anymore the symmetry that could be appreciated on Figure 8a.

It is also interesting to analyze the distribution of the values that the cost function takes for each measurement outcome. These results are displayed in Figure 9. In Figure 9a we can see that almost all measurement outcomes had associated the maximum value possible, 4. This is an expected behavior, since those correspond to the ideal noiseless simulation in which we already saw that almost all measurement outcomes correspond to correct results. For the first noisy simulation we still have quite good results in which most of the outcomes have associated a maximum value of the cost function. But for the second simulation and for the hardware backend we see that all the possible values that the cost function can take are equally likely, which indicates that the noise in those



(a) Noiseless simulation

(b) Simulation with $\epsilon_1 = 0.001$ & $\epsilon_2 = 0.01$ (c) Simulation with $\epsilon_1 = 0.01$ & $\epsilon_2 = 0.1$ 

(d) Yorktown 5-qubit Quantum Computer

Figure 9: Cost function associated to each measurement outcome. The red vertical line indicates the average.

systems is too high to extract any conclusion from these results.

5.3 V9E15 graph

We consider a slightly more complicated graph, named V9E15 since it consists in 9 nodes connected with 15 edges. The solution here is not as intuitive as in the previous cases but there exists a cut which crosses all the edges as showed in Figure 10, this subset is represented by the string $\{100111001\}$ or its complementary string.

We run the QAOA with 3 layers for this graph first in a noiseless simulations and with a dephasing with 1-qubit error rate of $\epsilon_1 = 0.01$ and 2-qubit error rate $\epsilon_2 = 0.1$.

Not all the string labels are displayed in the graph axis, nevertheless the correct solutions corresponds to the expected string for noiseless simulation (Graph 11a). Despite the larger number of nodes, the probability peak corresponds to the correct solution and the expectation value is close to the optimal value, showing that in this case as in the previous examples p is large enough to have satisfactory results.

The noisy simulation does not lead to the correct result: this is most likely due to high

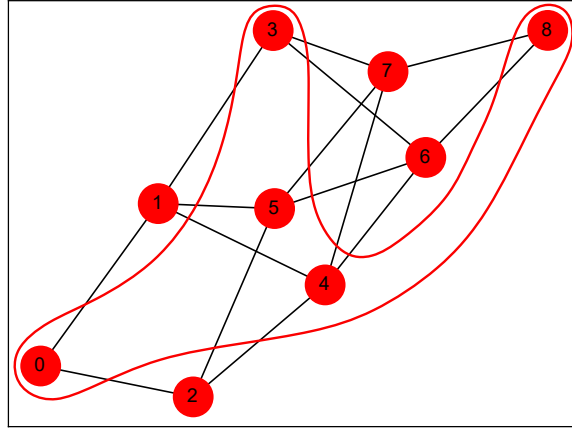
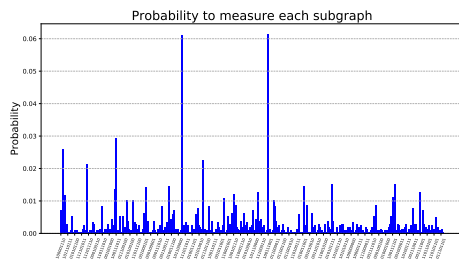
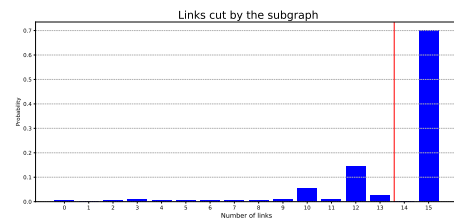


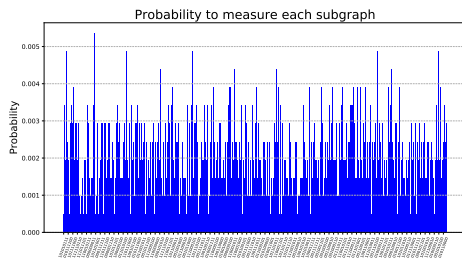
Figure 10: The V9E15 graph with the maximum cut which cuts all 15 edges.



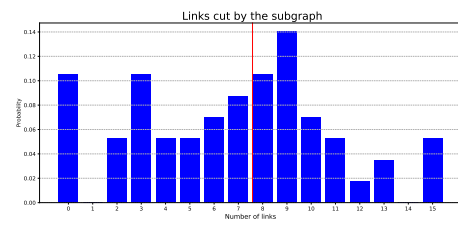
(a) $p = 3$, noiseless



(b) $p = 3$, noiseless



(c) $p = 3$, $\epsilon_1 = 0.01$, $\epsilon_2 = 0.1$



(d) $p = 3$, $\epsilon_1 = 0.01$, $\epsilon_2 = 0.1$

error rates chosen for the simulation. The choice was made in order to better simulate the noise levels of hardware backend as, in the case of the butterfly graph, we showed that the parametrized noise was indeed too low to give results similar actual hardware behavior. Despite the result, it is still useful to investigate a simulation in which the effect of noise is prominent and therefore we decided to include the result.

5.4 Algorithm runtime and performance

As mentioned previously, our analysis showed a couple of bottlenecks. First of all, it became evident when running simulations that most of the runtime (60%-80%) is spent within `scipy.optimize.differential_evolution()`, thus the classical optimizer is the limiting factor in case of simulations. The total runtime of our simulations was typically on the order of a couple hundreds of seconds (longer for more layers). Due to the nature of the classical optimizer it requires several calls of the quantum circuit to converge to the optimum solution of (β, γ) , typically 150 to 300 (more in the case of more layers, which makes sense as the dimension of the solution space grows with p). This becomes a mayor problem when executing the algorithm on hardware backend, as the position in the IBMQ queue gets reset after each run of the quantum circuit. Therefore the execution of the whole algorithm can take anywhere from one night to days.

These aspects indicated that contrary to suggestions in previous works [9] a choice of a different classical optimizer might be better suitable for our application.

6 Conclusion

In this project we have studied the Quantum Approximative Optimization Algorithm by implementing it on the MaxCut problem. To achieve this goal, we have developed a Python code that allows us to solve this problem in an arbitrary graph by running some quantum circuit on Qiskit Aer's QASM simulator. In the Results section we have shown how our code behaves for three different types of graph:

- Star Graph: We have used this simple graph to give some insights into the way that our code works. We also found out that our algorithm for this graph is quite robust under a simple noise model, probably due to the few number of gates used.
- Butterfly Graph: We were able to run the code for this graph in both, the QASM simulator and the Yorktown QC. In general, the results obtained for the hardware backend were too noisy to extract any clear conclusion, although in some cases it was possible to appreciate some successful outcomes. Comparing these results with the ones obtained with the simulator we can conclude that our noise model does not fit very well reality, so it is left as future work to develop a better one.

- V9E15: the simulations for a 9-node graph gave different results: the noiseless simulation was successful giving the expected result while when we tried to implement a simple noise model, it was not able to give the right result. This can be caused by multiple factors, in first instance the noise parameters used were evidently too high and secondly a circuit with a greater number of edges is less tolerant to noise since the number of edges determines the number of operators in the hamiltonian, and hence the number of gate used.

Summing up, we see that our code works very well in the noiseless regime. All the results obtained in this situation give as the most likely outcome the subgraph that cuts the maximum number of edges. But we have also seen that under strong noise or in a real hardware backend this is not fulfilled most of the times. Thus, we hope that in the future new improvements in the hardware backends will allow us to run this code and obtain correct results.

References

- [1] Scott Aaronson. *Shtetl-Optimized* » *Blog Archive* » *Quantum computing news items (by reader request)*. URL: <https://www.scottaaronson.com/blog/?p=2155> (visited on 02/12/2020) (cit. on p. 1).
- [2] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. “A Quantum Approximate Optimization Algorithm”. In: *arXiv:1411.4028 [quant-ph]* (Nov. 14, 2014). arXiv: [1411.4028](https://arxiv.org/abs/1411.4028). URL: <http://arxiv.org/abs/1411.4028> (visited on 11/28/2019) (cit. on pp. 1–3).
- [3] Stephen Jordan. *Quantum Algorithm Zoo*. Dec. 5, 2019. URL: <https://quantumalgorithmzoo.org/> (visited on 02/14/2020) (cit. on p. 1).
- [4] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. “A Quantum Approximate Optimization Algorithm Applied to a Bounded Occurrence Constraint Problem”. In: *arXiv:1412.6062 [quant-ph]* (June 25, 2015). arXiv: [1412.6062](https://arxiv.org/abs/1412.6062). URL: <http://arxiv.org/abs/1412.6062> (visited on 02/14/2020) (cit. on p. 1).
- [5] Boaz Barak, Ankur Moitra, Ryan O’Donnell, Prasad Raghavendra, Oded Regev, David Steurer, Luca Trevisan, Aravindan Vijayaraghavan, David Witmer, and John Wright. “Beating the random assignment on constraint satisfaction problems of bounded degree”. In: *arXiv:1505.03424 [cs]* (Aug. 11, 2015). arXiv: [1505.03424](https://arxiv.org/abs/1505.03424). URL: <http://arxiv.org/abs/1505.03424> (visited on 02/14/2020) (cit. on p. 1).
- [6] Edward Farhi and Aram W. Harrow. “Quantum Supremacy through the Quantum Approximate Optimization Algorithm”. In: *arXiv:1602.07674 [quant-ph]* (Oct. 20, 2019). arXiv: [1602.07674](https://arxiv.org/abs/1602.07674). URL: <http://arxiv.org/abs/1602.07674> (visited on 02/14/2020) (cit. on p. 1).
- [7] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O’Brien. “A variational eigenvalue solver on a photonic quantum processor”. In: *Nat Commun* 5.1 (July 23, 2014), pp. 1–7. ISSN: 2041-1723. DOI: [10.1038/ncomms5213](https://doi.org/10.1038/ncomms5213). URL: <https://www.nature.com/articles/ncomms5213> (visited on 02/14/2020) (cit. on p. 1).
- [8] John Preskill. “Quantum Computing in the NISQ era and beyond”. In: *Quantum* 2 (Aug. 6, 2018), p. 79. DOI: [10.22331/q-2018-08-06-79](https://doi.org/10.22331/q-2018-08-06-79). URL: <https://quantum-journal.org/papers/q-2018-08-06-79/> (visited on 02/14/2020) (cit. on p. 1).
- [9] Mahabubul Alam, Abdullah Ash-Saki, and Swaroop Ghosh. “Analysis of Quantum Approximate Optimization Algorithm under Realistic Noise in Superconducting Qubits”. In: *arXiv:1907.09631 [quant-ph]* (July 13, 2019). arXiv: [1907.09631](https://arxiv.org/abs/1907.09631). URL: <http://arxiv.org/abs/1907.09631> (visited on 12/04/2019) (cit. on pp. 3, 8, 15).

-
- [10] Rainer Storn and Kenneth Price. “Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces”. In: *Journal of Global Optimization* 11.4 (Dec. 1, 1997), pp. 341–359. ISSN: 1573-2916. DOI: [10.1023/A:1008202821328](https://doi.org/10.1023/A:1008202821328). URL: <https://doi.org/10.1023/A:1008202821328> (visited on 02/10/2020) (cit. on p. 7).
 - [11] Rainer Storn. *Differential Evolution Homepage*. URL: http://www1.icsi.berkeley.edu/~storn/code.html#Practical_Advice (visited on 02/14/2020) (cit. on p. 8).
 - [12] Gavin E. Crooks. “Performance of the Quantum Approximate Optimization Algorithm on the Maximum Cut Problem”. In: *arXiv:1811.08419 [quant-ph]* (Nov. 20, 2018). arXiv: [1811.08419](https://arxiv.org/abs/1811.08419). URL: <http://arxiv.org/abs/1811.08419> (visited on 12/04/2019) (cit. on p. 8).

A Additional results

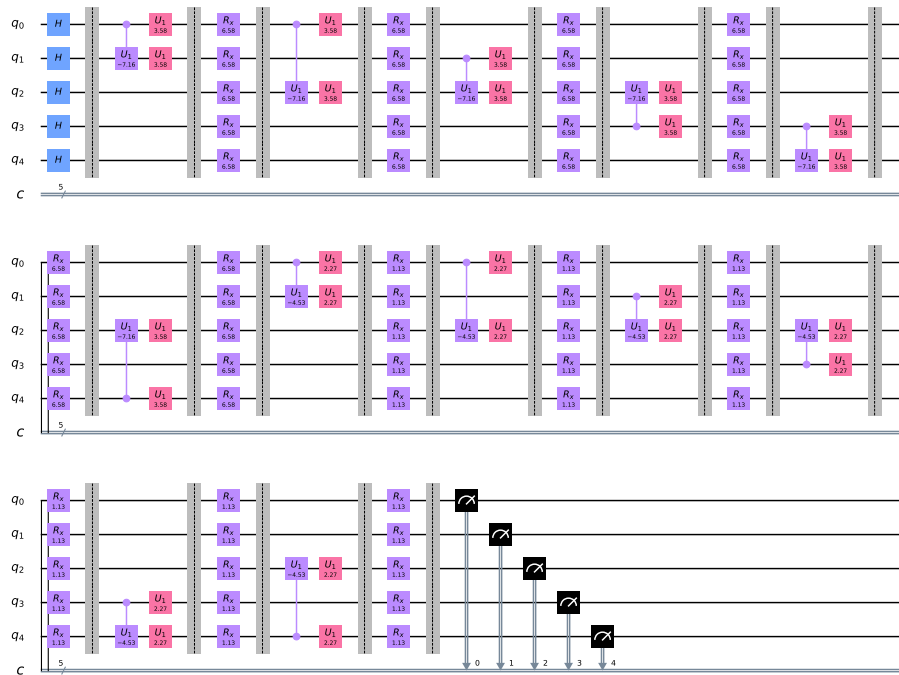


Figure 12: An example of the QAOA circuit with $p = 2$ layers executed on IBMQ's *yorktown* 5-qubit quantum processor to solve the MaxCut problem for the Butterfly graph.

B Source code

Listing 1: `Graph.py` — The main file containing the `Graph` class.

```
#import math tools
import numpy as np

import os.path
import time

# We import the tools to handle general Graphs
import networkx as nx

# We import plotting tools
import matplotlib.pyplot as plt
```

```

from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

# importing Qiskit
from qiskit import Aer, IBMQ
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit, execute
import qiskit.providers.aer.noise as noise

from qiskit.providers.ibmq      import least_busy
from qiskit.tools.monitor      import job_monitor
from qiskit.visualization import plot_histogram

from sympy import Symbol, Matrix, init_printing, pprint, sin, cos, simplify, pi,
↳ zeros, lambdify, ones
from sympy.physics.quantum import TensorProduct as TP
from sympy.physics.quantum.dagger import Dagger as Dag

from scipy.optimize import differential_evolution

class Graph():
    def __init__(self, n):
        self.n = n
        self.vExecutionTime = []
        self.nCircuitCalls = 0

        #Simulation parameters
        self.backend = Aer.get_backend("qasm_simulator")
        self.shots = 2048

        # IBMQ.load_account()
        # print(IBMQ.providers())
        # vBackends = IBMQ.get_provider(group='open').backends()
        # self.backend = vBackends[6] # 'ibmq_qasm_simulator'
        # print(self.backend)

        self.G = nx.Graph()
        self._Assign()

        #Assign nodes and edges to the graph
    def _Assign(self):
        self.V = np.arange(0, self.n, 1)
        self.E = self._Read_E()

        self.G.add_nodes_from(self.V)
        self.G.add_weighted_edges_from(self.E)

```

```

#Read file with edges
def _Read_E(self):
    # x = np.genfromtxt(r'Edges.txt', delimiter=',')
    x = np.genfromtxt(r'V9E15.txt', delimiter=',')
    # x = np.genfromtxt(r'Butterfly.txt', delimiter=',')

    return x

# Generate plot of the Graph
def Plot_G(self):
    colors      = ['r' for node in self.G.nodes()]
    default_axes = plt.axes(frameon=True)
    pos         = nx.spring_layout(self.G)

    nx.draw_networkx(self.G, node_color=colors, node_size=600, alpha=1,
        ↪ ax=default_axes, pos=pos)

    plt.savefig('Graph.pdf', format='pdf', dpi=256)
    plt.clf()

#Optimize gamma and beta using the simulator
def Optimizer(self, p, n, *args):
    bounds = []
    self.noise = n
    self.p = p
    # Error probabilities
    if self.noise:
        self.prob_1 = args[0] # 1-qubit gate
        self.prob_2 = args[1] # 2-qubit gate
    else:
        pass

    for i in range(0,2*p):
        bounds.append((0,2*np.pi))
    tDEtot0 = time.perf_counter()
    result = differential_evolution(self._Simulate, bounds, updating='immediate',
        ↪ mutation=(0.5,1), recombination=0.9, tol=0.1,
        ↪ maxiter=500, workers=6)

    tDEtot1 = time.perf_counter()
    print("Total time for DE optimizer: %.2f s" % (tDEtot1-tDEtot0))
    print("Circuit execution time of %i calls: mean = %.2f s, total = %.2f s"
        ↪ % (len(self.vExecutionTime), np.mean(self.vExecutionTime),
        ↪ np.sum(self.vExecutionTime)))

```

```

print("Actual time spent on DE optimization: %.2f s, which is %.1f %%"
      % ( (tDEtot1-tDEtot0)-np.sum(self.vExecutionTime),
          ↪ 100*((tDEtot1-tDEtot0)-np.sum(self.vExecutionTime))/(tDEtot1-tDEtot0)
          ↪ ) )

print("DE optimization results: ", result.x, -1*result.fun)
self.F = -1*result.fun

# prevent files from overwriting
filename = "counts_"
filenum = 1
while os.path.exists(filename + str(filenum) + ".npz"):
    filenum += 1

np.save("counts_%i" % filenum, self.QAOA_results.get_counts())
np.save("hist_%i" % filenum, [self.hist, self.F])

#Building the circuit
def _Build(self):
    # preapre the quantum and classical resisters
    QAOA = QuantumCircuit(len(self.V), len(self.V))

    # apply the layer of Hadamard gates to all qubits
    QAOA.h(range(len(self.V)))
    QAOA.barrier()

    for i in range(self.p):
        # apply the Ising type gates with angle gamma along the edges in E
        for edge in self.E:
            k = int(edge[0])
            l = int(edge[1])

            QAOA.cu1(-2*self.gamma[i], k, l)
            QAOA.u1(self.gamma[i], k)
            QAOA.u1(self.gamma[i], l)

        # then apply the single qubit X - rotations with angle beta to all
        ↪ qubits
        QAOA.barrier()
        QAOA.rx(2*self.beta[i], range(len(self.V)))
        QAOA.barrier()

    # Finally measure the result in the computational basis
    QAOA.measure(range(len(self.V)),range(len(self.V)))

```

```

self.QAOA = QAOA

#Draw circuit
def Plot_C(self):
    self.QAOA.draw(output='mpl')
    plt.savefig('Circuit.pdf', format='pdf', dpi=256)
    plt.clf()

# Compute the value of the cost function
def cost_function_C(self, x):
    if( len(x) != len(self.V)):
        return np.nan

    C = 0;
    for index in self.E:
        e1 = int(index[0])
        e2 = int(index[1])

        w      = self.G[e1][e2]['weight']
        C = C + w*x[e1]*(1-x[e2]) + w*x[e2]*(1-x[e1])

    return C

#Simulate the circuit
def _Simulate(self, params):
    self.beta = []
    self.gamma = []
    for i in range(self.p):
        self.beta.append(params[2*i])
        self.gamma.append(params[2*i+1])

    self._Build()

    tExec0 = time.perf_counter()
    if self.noise:
        # Depolarizing quantum errors
        error_1 = noise.errors.standard_errors.depolarizing_error(self.prob_1, 1)
        error_2 = noise.errors.standard_errors.depolarizing_error(self.prob_2, 2)

        # Add errors to noise model
        noise_model = noise.NoiseModel()
        noise_model.add_all_qubit_quantum_error(error_1, ['u1', 'u2', 'u3'])
        noise_model.add_all_qubit_quantum_error(error_2, ['cx'])
        basis_gates = noise_model.basis_gates

```

```

        self._Build()
        simulate = execute(self.QAOA, backend=self.backend, shots=self.shots,
                           basis_gates=basis_gates,
                           noise_model=noise_model)

    else:
        self._Build()
        simulate = execute(self.QAOA, backend=self.backend, shots=self.shots)
    # job_monitor(simulate)
    tExec1 = time.perf_counter()
    self.vExecutionTime.append(tExec1-tExec0)
    self.nCircuitCalls += 1

    QAOA_results = simulate.result()
    self.QAOA_results = QAOA_results

    # Evaluate the data from the simulator
    counts = QAOA_results.get_counts()

    avr_C = 0
    max_C = [0,0]
    hist = {}

    for k in range(len(self.G.edges())+1):
        hist[str(k)] = hist.get(str(k),0)

    for sample in list(counts.keys()):

        # use sampled bit string x to compute C(x)
        x = [int(num) for num in list(sample)]
        tmp_eng = self.cost_function_C(x)

        # compute the expectation value and energy distribution
        avr_C = avr_C + counts[sample]*tmp_eng
        hist[str(int(tmp_eng))] = hist.get(str(round(tmp_eng)),0) + counts[sample]

        # save best bit string
        if( max_C[1] < tmp_eng):
            max_C[0] = sample
            max_C[1] = tmp_eng

    self.max_C = max_C
    self.hist = hist
    M1_sampled = avr_C/self.shots
    print("beta = ", self.beta, "gamma = ", self.gamma)

```

```

    return -1*M1_sampled

def Plot_S(self):
    # prevent files from overwriting
    filename = "Simulator_counts_"
    filenum = 1
    while os.path.exists(filename + str(filenum) + ".pdf"):
        filenum += 1

    if self.n > 5:
        figsize = (14,6)
        plt.xticks(fontsize = 7)
    else:
        figsize = (8,6)
        plt.xticks(fontsize = 10)

    plot_histogram(self.QAOA_results.get_counts(),figsize = figsize,bar_labels =
        ↪ False)
    plt.suptitle('Probability to measure each subgraph', fontsize = 20)
    plt.savefig('Simulator_counts_%i.pdf' % filenum, format='pdf', dpi=256)
    plt.xlabel('Measurement outcome')
    plt.clf()

    print('\n --- SIMULATION RESULTS ---\n')
    print('The sampled mean value is M1_sampled = %.02f' % (self.F))
    print('The approximate solution is x* = %s with C(x*) = %d \n' %
        ↪ (self.max_C[0],self.max_C[1]))
    print('The cost function is distributed as: \n')

    # plt.figure(figsize=figsize)
    # plt.bar(self.hist.all().keys(), self.hist.all().values(), color='b')
    plot_histogram(self.hist,figsize = figsize,bar_labels = False)
    plt.axvline(x=self.F, color='r')
    plt.xlabel('Number of links', fontsize = 12)
    plt.title(' Links cut by the subgraph', fontsize = 20)
    plt.savefig('Simulator_counts_%i.pdf' % (filenum+1), format='pdf', dpi=256)
    plt.clf()

    print("Circuit execution time of %i calls: mean = %.2f s, total = %.2f s"
        % (len(self.vExecutionTime), np.mean(self.vExecutionTime),
        ↪ np.sum(self.vExecutionTime)) )
    print("Circuit called %i times" % self.nCircuitCalls)

def plotFromSavedData(self,path):
    (hist, mean) = np.load(path+"hist_1.npy", allow_pickle=True)

```

```

fig, ax = plt.subplots(figsize=(14,6))
ax.set_title(' Links cut by the subgraph', fontsize = 20)
ax.set_ylabel('Probability')
ax.set_xlabel('Number of links', fontsize = 12)
ax.bar(hist.keys(), np.array(list(hist.values()))/np.sum(list(hist.values()))),
    ↪ color='b')
x = np.arange(0,len(hist.values()),1)
mean = np.dot(np.array(list(hist.values())), x) /
    ↪ np.sum(np.array(list(hist.values()))))
ax.axvline(mean, color='r')
ax.grid(which='major', linestyle='--', axis='y')
fig.savefig('hist.pdf', format='pdf', dpi=256)
plt.show()

hist = np.load(path + "counts_1.npy", allow_pickle=True)
fig, ax = plt.subplots(figsize=(12, 6))
# new_x = [1.1 * i for i in range(len(hist.all()))]
# xpos = np.arange(0,len(hist.all()),1)
xpos = np.linspace(-10,4*len(hist.all()),len(hist.all()))
ax.bar(xpos, np.array(list(hist.all().values())) /
    ↪ np.sum(list(hist.all().values()))),
    align='center', width=4, color='b')
# ax.set_xticks(new_x, list(hist.all().keys()) )
# ax.set_xticklabels(hist.all().keys())
ax.set_title('Probability to measure each subgraph', fontsize=20)
ax.set_ylabel('Probability', fontsize=16)
ax.grid(which='major', linestyle='--', axis='y')
ax.xaxis.set_tick_params(width=0.8)
plt.xticks(xpos[::6], list(hist.all().keys())[::6], rotation=70, fontsize=6)
fig.savefig('counts.pdf', format='pdf', dpi=256)
plt.show()

```

Listing 2: main.py — The file from which the `Graph` is created and the algorithm is executed.

```

import numpy as np
import matplotlib.pyplot as plt
from Graph import Graph
import time

# r1 = np.arange(0.001, 0.021, 0.001)
# r2 = np.arange(0.01, 0.21, 0.01)
# M1 = []
# M2 = []

```

```
t0 = time.perf_counter()

#Modify file Edges.txt to introduce the edges of the graph
G = Graph(9) #The argument is the total number of vertices of the graph
G.Plot_G()
#The first argument for Optimizer is p, the number of layers for the QAOA
#The second argument for Optimizer is a boolean to use or not noise in our simulation
#The third and fourth arguments are optional and are the error rate of 1 and 2 qubits

# G.Optimizer(3, True, 1e-2, 1e-1)
G.Optimizer(3, False)

G.Plot_C()
G.Plot_S()

t1 = time.perf_counter()
print("Total execution time: %.2f s" % (t1-t0))

# for i in r1:
#     G.Optimizer(2, True, i, 0.01)
#     M1.append(G.F)
#
# plt.plot(r1,M1)
# plt.xlabel('1-qubit error rate')
# plt.ylabel('Average cost function')
# plt.savefig('Error1.png')
# plt.clf()
#
# for i in r2:
#     G.Optimizer(2, True, 0.001, i)
#     M2.append(G.F)
#
# plt.plot(r2,M2)
# plt.xlabel('2-qubit error rate')
# plt.ylabel('Average cost function')
# plt.savefig('Error2.png')
# plt.clf()
```
