Application of nature inspired metaheuristics to the optimization of QAOA circuits.

Mentee Luis Eduardo Martinez Hernandez
Mentor Alberto Maldonado Romo
Mentor Amandeep Bhatia

January 10, 2022

Abstract

This work investigate the potential use of metaheuristics to optimize QAOA circuits. The report contains a summary of the used methods and the experimental study to observe the results. The results indicate that nature inspired metaheuristics can be a good alternative to classical optimizers.

1 Introduction

Metaheuristics are not a novel method for optimization problems. In fact, one of the first metaheuristics was proposed in 1982 with the one called simulated annealing [1]. A metaheuristic is simply a procedure that coordinates the use of heuristics to produce high quality solutions [8]. Where an heuristic is: 'a method which, on the basis of experience or judgement, seems likely to yield a reasonable solution to a problem, but which cannot be guaranteed to produce the mathematically optimal solution.'[8].

Metaheuristics are composed of two phases: exploration and exploitation [1]. In the exploration phase, the metaheuristic searches the solution space of the objective function. While in the exploitation phase the candidates solutions, obtained from the previous phase, are evaluated to find the one with the best quality. Additionally, metaheuristics have several controllable parameters for flexibility, although, it requires the careful configuration of these parameters. [8]

Among the different types of metaheuristics, there is a particular group that will be of interest later on. This group is the population based methods. Some examples of these are: Ant colony optimization (ACO), Particle Swarm Optimization (PSO), Artificial Bee Colony (ABC) and Bat optimization (BA) [1].

On the other hand, the quantum approximation optimization algorithm is a method to produce approximate solutions to combinatorial optimization problems[4]. The algorithm depends on a parameter, usually called p and the algorithm gets a better solution the bigger the parameter is. This method uses a set of parametrized unitary

transformations. To get the right parameters of these transformations, it is a common practice to use a classical optimizers. So the problem that we'll address is to find the performance of some metaheuristics in this optimization process.

2 Nature inspired Metaheuristics

We will consider four metaheuristics for the problem at hand. The chosen metaheuristics are: Bat optimization(BAT), Particle Swarm Optimization (PSO), Artificial Bee Colony (ABC) and Ant colony optimization (ACO). This methods were selected because they have shown good results with other optimization problems, like the optimization of weights for neural networks [7] [6] [1].

2.1 Bat optimization

The key feature of this method is exploiting the fact that usually bats use echolocation to hunt their prey [3]. Echolocation happens when bats emit a sound and then they listen to the bounced sound to identify their prey [3]. And even further, studies show that bats which use echolocation can create a three dimensional scenario of their surrounding, the moving speed of the prey, the distance of the target, among other things [3]. As such, the bat optimization method can be summarized as follows:

Algorithm 1 Bat optimization

procedure BA
    bat population ← initialize a list of bats
    for each bat in bat population ← initialize its pulse rate r, loudness A,
    velocity, initial position, pulse interval and frequency (min and max)
    $t \leftarrow 0$
    while (t < number of iterations):
        best position ← best position in the population
        average loudness ← get average loudness from population
        for each bat in population:
            frequency ← frequency min +
            (frequency max-frequency min)*random number
            velocity ← velocity + (position - best position)*current frequency
            position ← position + velocity
            if random number > current pulse interval **then**
                position ← best position+random number*average loudness
            position ← random position+random number*average loudness
            if random number < loudness and cost < best position cost **then**
                loudness ← random number*loudness
                pulse interval ← initial pulse interval $*$ $(1 - (e^{-t*random}))$
    Return the best position found by the bats

2.2 Particle Swarm Optimization

While bat optimization technique uses bats and their corresponding echolocation to search the solution space, the particle swarm optimization uses the swarm behaviour of particles to optimize an objective function. The swarm behaviour as outlined in [7] must accomplish 5 points. Some of these points are: the swarm must be able to make simple space and time calculations, the swarm must be able to respond to quality factors in the environment, the swarm should not change its mode of behavior every time the environment changes and the swarm must be able to change behavior mode when it's worth the computational price [7]. The key idea behind the algorithm is that the swarm may follow the direction of the best solution found, although the particles may not follow this direction accordingly to a random variable. Additionally, a particle adjust it's velocity accordingly to the proximity of its current objective. The method can be summarized as follows:

Algorithm 2 Particle swarm optimization

```
1:  procedure PSO
2:      swarm ← initialize a list of particles
3:      for each particle in swarm ← initialize its initial position, velocity
4:      and best position found
5:      w ← number in range 0 < w < 1
6:      c1 ← number in range 0 < c1 < 1
7:      c2 ← number in range 0 < c2 < 1
8:      while (t < number of iterations):
9:          best position ← best position in the swarm
10:         for each particle in population:
11:             current position ← current position+velocity
12:             if current position cost < best position found cost then
13:                 best position found ← current position
14:             velocity ← w*velocity+
15:             c1*random number*(best position found-current position)+
16:             c2*random number*(best position found by swarm-current position)
17:     Return the best position found by the swarm
```

2.3 Ant colony optimization

Originally ACO was proposed to solve discrete optimization problems [8]. However, there are proposals to change the nature of the optimizer to one that solves continuous optimization problems. We'll use one of these proposals. Before explaining the method, we'll explain the key feature of the ant colony optimization. Ants communicate with other ants in the colony by a secretion called pheromone [9]. This chemical is produce when the ants in the colony move to the candidate solutions and then they return to the colony. Usually the pheromone trail to indicate the quality of the solution to other ants. Each ants that passes over the pheromone trail reinforces the trail and in consequence, the trail will be more appealing to other ants [8] [2]. Even further, this pheromone trail has the characteristic that evaporates over time. This is a mechanism that allows the ants to forget bad solution and explore new ones [8]. The general method can be summarized as follows:

Algorithm 3 Ant colony optimization

```
1:  procedure ACO
2:      colony ← initialize a list of ants
3:      points ← initialize a list of uniform distributed points
4:      for each ant in colony ← initialize its memory, memory limit and current location
5:      while (t < number of iterations):
6:          for each ant in colony:
7:              if random number > q then
8:                  ant current location ← best position
9:                  store current position in ant memory
10:                 move the ant location accordingly to a gradient optimizer
11:             else
12:                 ant current location ← grab a random point with a probability of
13:                 pointPheromone/totalPheromone
14:                 store current position in ant memory
15:                 move the ant location accordingly to a gradient optimizer
16:         for the best ant in colony:
17:             current location pheromone ← pheromone + (1/error)
18:         for each point:
19:             point pheromone ← pheromone ∗ (1 − p)
20:     Return the best position found by the colony
```

2.4 Artificial bee colony optimization

In the bee colony optimization technique we use bees as an agent to solve the optimization problem. In the proposed method by [6] bees can be one of three types: The worker bees, the onlooker bees and the scout bees. The worker bees go to the position of a potential solution and explore the neighborhood to see if a better solution can be found. The onlooker bees search the best proposals of worker bees accordingly to a random decision and finally the scout bees explore a new solution at random. This interaction can be summarized by the following pseudocode:

Algorithm 4 Artificial Bee Colony optimization

```
1:  procedure ABC
2:      swarm ← initialize a list of worker bees
3:      onlookers ← initialize a list of onlooker bees
4:      for each bee in swarm ← initialize its current location at random
5:      for each bee in onlookers ← initialize its current location at random
6:      while (t < number of iterations):
7:          for each bee in swarm:
8:              new position ← bee position +
9:              random(-a,a)*(bee position[random] - swarm[random].position[random])
10:             if new position cost <  bee position cost then
11:                 bee position ← new position
12:             else
13:                 if solution has not improved in r tries then
14:                     bee position ← lower bound +
15:                      random uniform(0, 1)*(upper bound-lower bound)
16:         for each bee in onlookers:
17:             selected position ← select a position of the bees in swarm using a roulette selection
18:             new position ← selected position +
19:             random(-a,a)*(selected position[random] - swarm[random].position[random])
20:             if new position cost <  bee position cost then
21:                 selected position ← new position
22:     Return the best position found by the colony
```

3 Quantum Approximation Optimization Algorithm

Recently, a new way to address NP-Complete problems has been created. This new method is the quantum approximate optimization algorithm (QAOA). This method consists in exploiting the properties of quantum mechanics to arrive to an approximation to a solution for a NP-Complete problem. The general method can be summarized as follows:

1. Create an initial state that is a uniform superposition over the computational basis states.
2. Apply the unitary transformations $U(\boldsymbol{\gamma}) = e^{-i\gamma}$ and $U(\boldsymbol{\beta}) = e^{-i\beta H_B}$, p times. Where β and α are a set of angles, H_P is the problem Hamiltonian and H_B is the mixing Hamiltonian.
3. Measure in the computational basis and calculate the cost.

After calculating the cost in step 3, it is common to use an optimizer to change the value of the angles in order to get a better result.

Additionally, one form of measuring the overall performance of a QAOA circuit is by calculating the approximation ratio. The approximation ratio is calculated by diving the probability of the solution by the probability of the highest output [5].

3.1 Max-cut

The max-cut is a well know problem in computer science. It consists in finding two disjoint sets in a graph such that the number of edges between nodes in different sets are maximized [4]. For this problem, the proposed mixing Hamiltonian and problem Hamiltonian are taken directly from [5]:

$H_P = \sum_{(i,j)\in E} \frac{1}{2}\left(1 - Z_i Z_j\right)$
$H_B = \sum_{(j=1}^{n} \sigma_j^x$

4 Experimental study

To evaluate the performance of the QAOA circuits while using the nature inspired optimizers we'll compare the execution time, approximation ratio, probability of one solution and the the probability to get any solution with other common optimization techniques such as Nelder-Mead, COBYLA and SLSQP optimizers. The selected problem to compare the performance is the well know max cut problem.

The experiments for the max-cut problem consisted in creating 4-complete to 9 complete graphs. Afterwards, we executed the QAOA instance with p value from 1 to 8 using a qasm simulator with 1000 shots to check the execution time, approximation ratio, probability of one solution and the probability to get any solution. To get the the set of optimal solutions, we executed a brute force algorithm that enumerated all possible answers. For testing the approximation ratio and the probability of getting one correct solution we selected at random one of the solutions from the set of optimal solutions. Additionally, for the qasm simulator test, we decided to get the output of the

circuit with the same parameters 30 times to get more accurate results. The following hyper parameters were used for the optimizers:

Method	Hyperparameters
PSO	particles=20, w=0.4, c1=0.1, c2=0.1, iterations=50
BA	bats=5, iterations=50, alfa= 0.4, gamma=0.4
ACO	points=100, ants=20, q=0.01, evaporation rate=0.9, iterations = 100, local search routine=COBYLA, num iterations for local search=10
ABC	points=100, ants=20, q=0.01, evaporation rate=0.9, iterations = 100
Nelder-Mead	Absolute error in between iterations that is acceptable for convergence=0.0001, maximum iterations=until convergence.
COBYLA	rhobeg= 1.0, maxiter= 1000
SLSQP	maxiter= 100, ftol= 1e-06, eps= 1.4901161193847656e-08

Table 1: Hyperparameters for the optimizers

On the other hand, the tests were executed using a computer with a ryzen 1300 processor with 8gb of RAM. Finally, the tests were ran using qiskit and scipy library in their version 0.29.0 and 1.7.1 respectively.

5 Results

In this section, we'll present the results of the executions described in the experimental study section. The following results show the mean value across the different values of p. For example, the execution time for K4 and COBYLA optimizer shows the mean execution time for the different values of p.

For starters, the experiments lead us to conclude that the execution time is variable between the optimizers. However, the optimizers that take longer times to run are the BA, ABC and Nelder-Mead. We can see that this is presented for every graph that we used for the experiments. Other thing to notice is that ACO takes a longer time to converge compared to the COBYLA optimizer. This is due to the fact that ACO calculates several COBYLA subroutines as part of its optimization process. The last insigth that we have is that COBYLA and PSO are the fastest methods among the classical and nature inspired optimizers. Here is the complete output for this experiments:

Graph	Nelder-Mead	COBYLA	SLSQP	PSO	BA	ACO	ABC
K4	108.30	5.84	61.20	56.37	159.22	72.59	129.95
K5	126.02	6.34	60.99	68.29	197.95	89.39	157.9
K6	147.0	7.85	83.96	80.66	225.97	104.29	190.69
K7	169.8	9.07	88.63	93.13	263.13	121.4	215.82
K8	213.58	11.22	120.69	114.59	321.66	149.53	267.28
K9	246.57	13.32	100.96	133.69	376.24	174.88	309.83

Table 2: Execution time results

As we can see in the following two tables, the results for any optimizer are quite low. However, we can see that PSO gives the best results on K4 with BA optimizer. Another thing to note is that the more nodes the graph has, the more difficult is for the optimizer to get a good probability for a selected solution.

Graph	Nelder-Mead	COBYLA	SLSQP	PSO	BA	ACO	ABC
K4	0.11	0.14	0.08	0.16	0.16	0.14	0.16
K5	0.04	0.05	0.02	0.05	0.05	0.05	0.05
K6	0.03	0.04	0.01	0.04	0.04	0.04	0.04
K7	0.01	0.01	0.00	0.01	0.01	0.01	0.01
K8	0.01	0.01	0.00	0.01	0.01	0.01	0.01
K9	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 3: Ground state probability

Graph	Nelder-Mead	COBYLA	SLSQP	PSO	BA	ACO	ABC
K4	0.02	0.03	0.02	0.03	0.03	0.03	0.03
K5	0.02	0.03	0.01	0.03	0.03	0.03	0.03
K6	0.02	0.03	0.01	0.03	0.03	0.03	0.02
K7	0.01	0.02	0.00	0.02	0.02	0.02	0.02
K8	0.01	0.02	0.0	0.01	0.01	0.02	0.02
K9	0.01	0.01	0.0	0.01	0.01	0.01	0.01

Table 4: Approximation ratio

While the results for one solution may seem bad, the results for getting any solution are more promising. We can observe several things, the first one is that in general the performance of PSO and ABC are good compared to the other optimizers. Other thing to notice is that even that ACO uses a COBYLA subroutine, the metaheuristic doesn't always yield to better results compared to the classical COBYLA method. Another

problem that the previous results showed is that with higher dimensions, the methods tend to lower their performance.

Graph	Nelder-Mead	COBYLA	SLSQP	PSO	BA	ACO	ABC
K4	0.67	0.81	0.49	0.94	0.93	0.85	0.95
K5	0.74	1.00	0.48	0.98	0.98	0.92	0.98
K6	0.59	0.84	0.25	0.8	0.76	0.88	0.75
K7	0.67	0.95	0.16	0.87	0.89	0.89	0.91
K8	0.4	0.79	0.15	0.71	0.66	0.64	0.72
K9	0.53	0.78	0.14	0.81	0.79	0.78	0.73

Table 5: Probability of measuring a solution

6 Conclusions

With the observed results we can conclude that nature inspired metaheuristics are a good alternative to classical optimizers. Between them, PSO and ABC showed the most promising results due to the high probability of measuring a correct solution and low execution time. Additionally, we offer the follow summary of the advantages and disadvantages of the used metaheuristics that we constructed from the results showed in the results section.

Method	Advantages	Disadvantages
PSO	Low execution time, good results (better than SLSQP and Nelder-Mead) and easy to implement	Takes longer execution time compared to COBYLA
BA	Good results (better than SLSQP and Nelder-Mead) and medium difficulty to implement	Takes the longer execution time of all methods
ACO	In some executions gives good results (better than SLSQP and Nelder-Mead)	Difficult to implement, worse results compared to COBYLA and takes a long execution time.
ABC	Good results (better than SLSQP and Nelder-Mead), medium difficulty to implement, lower execution time compared to BA and ACO.	In the experiments it never outperformed COBYLA and takes more time compared to COBYLA.

Table 6: Summary of methods

References

[1] Ashraf Mohamed Hemeida, Somaia Awad Hassan, Al-Attar Ali Mohamed, Salem Alkhalaf, Mountasser Mohamed Mahmoud, Tomonobu Senjyu, Ayman Bahaa El-Din, Nature-inspired algorithms for feed-forward neural network classifiers: A survey of one decade of research, Ain Shams Engineering Journal, Volume 11, Issue 3, 2020, Pages 659-675, ISSN 2090-4479, https://doi.org/10.1016/j.asej.2020.01.007.

[2] Mavrovouniotis, M., Yang, S. Training neural networks with ant colony optimization algorithms for pattern classification. Soft Comput 19, 1511–1522 (2015). https://doi.org/10.1007/s00500-014-1334-5

[3] Yang, Xin-She. (2010). A New Metaheuristic Bat-Inspired Algorithm. 284. 10.1007/978-3-642-12538-6_6.

[4] Edward Farhi and Jeffrey Goldstone and Sam Gutmann. (2014). A Quantum Approximate Optimization Algorithm.

[5] Ryan Hoque. (2020). The Quantum Approximate Optimization Algorithm.

[6] Dervis Karaboga (2010) Artificial bee colony algorithm. Scholarpedia, 5(3):6915.

[7] J. Kennedy and R. Eberhart, "Particle swarm optimization," Proceedings of ICNN'95 - International Conference on Neural Networks, 1995, pp. 1942-1948 vol.4, doi: 10.1109/ICNN.1995.488968.

[8] Silver, E. An overview of heuristic solution methods. J Oper Res Soc 55, 936–956 (2004). https://doi.org/10.1057/palgrave.jors.2601758

[9] Yang, X.-S.: Nature-inspired Metaheuristic Algorithms. Luniver Press, (2008).