Implementing a Quantum Genetic Algorithm

P A Ayantha Randika
ponnampe@ualberta.ca

1 Introduction

In this project, I have attempted to implement a genetic algorithm to run on a quantum environment. I have referred to two published articles by Yang, Wang, & Jiao (2004) and Wang *et al.* (2013) to implement their quantum genetic algorithm in Microsoft Q#[1] language. The first section of this report describes classical GA and the second section describes the QGA suggested in the papers. The third section describes the implementation of the simulation of the QGA. The fourth section describes the implementation attempt of QGA using Q#. The final section presents the conclusions derieved from the project.

2 Classical Genetic Algorithm

Genetic Algorithms(GA) are a set of heuristic algorithms which comes under the umbrella of evolutionary algorithms. In this context by adding the "classical," I try to distinguish the set of GA which uses classical computer architecture for the implementation. GAs were inspired by the theory of natural selection by Charles Darwin and first suggested and investigated by Holland (1975). In a mathematical sense, GAs are function optimisers and an excellent candidate to solve NP-hard problems. Most interesting properties of GAs are efficiency, easily programmability and extream robustness regarding the input data (Meyer-Baese & Schmid 2014). GAs have three main concepts: fitness function, crossover and mutation.

2.1 Population

The population of a simple GA consists of a large number of chromosomes. These chromosomes are represented in the implementations as strings of bits. Each bit of a chromosome can be considered as a gene. Figure 1 depicts the concepts of genes, chromosomes and population in a simple GA.

[1]https://docs.microsoft.com/en-us/quantum/language/?view=qsharp-preview

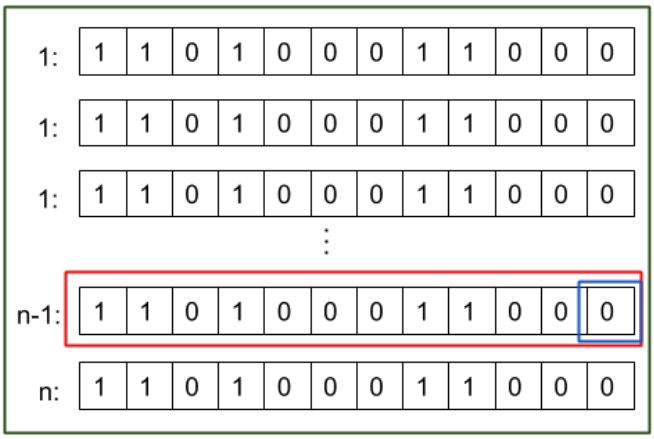

Figure 1: In this figure the area circled by a blue line depicts one bit which represents a single gene in the chromosome. Circle by the red line is a chromosome, which is represented by a string of bits. Population is circled by the green line and it consists on n chromosomes.

2.2 Fitness function

Fitness function is which assess the ability of each in the population to solve the main problem. It usually assigns a score to every individual based on their ability. This score is used to mimic natural selection by eliminating the individual with lower scores allowing high scorers to thrive.

2.3 Crossover

Crossover is an essential genetic operator used in GAs. Crossover is used to combine the genetic information of two parents and create a new generation of offspring. There are different crossover techniques used in GAs. The most simple crossover method is to use each parent's genes according to a fixed percentage. Therefore final offspring will have n% from one parent and n-1% from the other parent.

2.4 Mutation

The mutation is another genetic operator used in GAs. In the simplest form, the mutation is achieved by flipping the bits of chromosomes according to a probability distribution. Mutation in GAs prevent pre-mature convergence and it maintains the diversity from one generation to the next.

2.5 Algorithm

The initial step of the GA is to initialise a population with random genes. Then using the fitness evaluation function, the fitness of each chromosome is evaluated. Based on the fitness score a subset of chromosomes are selected, and a new generation is created through crossover and mutations are added. If the algorithm meets the criteria, it is stopped or otherwise the process is repeated. The process is depicted as a flow chart in Figure 2.

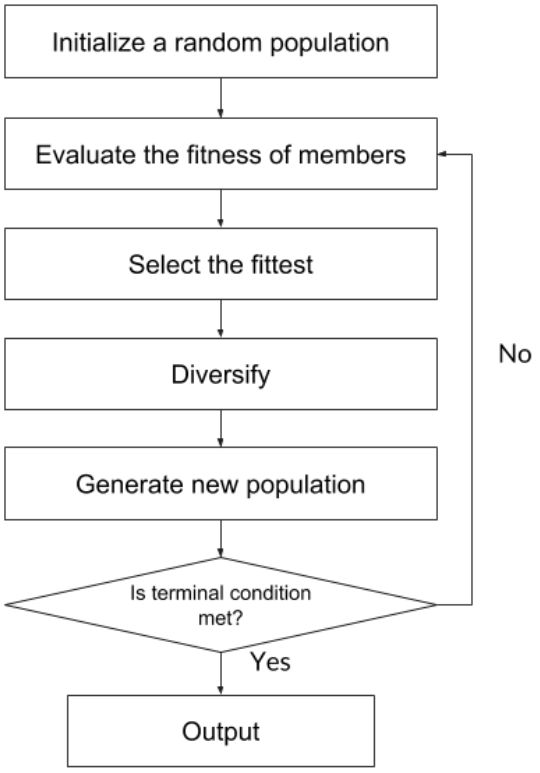

Figure 2: Flow chart of a GA

3 Motivation for a Quantum Solution

As described in the previous section, every chromosome in the population is a string of bits. The purpose of the algorithm is to search combinations of these bits until it finds a right solution. However, if we can use a string Qubits, every member of the population can be represented at once by using the superposition of these Qubits. Therefore, instead of going through several generations we would be able to find the fittest bit combination which solves the problem. This process appears somewhat closer to Grover's algorithm.

The paper by Udrescu, Prodan, & Vlăduţiu (2006) appears to use my initial intuition of Grover's algorithm to implement Quantum Genetic Algorithm (QGA). However, due to the complexity of their approach, I have searched further and found the paper by Yang, Wang, & Jiao (2004), which suggests a reasonably simple QGA approach using quantum chromosomes. The article by Wang *et al.* (2013) enhances the initial approach suggested earlier. Due to the simple method, I have decided to investigate the possibility of implementing these algorithms in Q# language; therefore it can run in an actual quantum environment.

4 Quantum Genetic Algorithm

The QGA suggested by Yang, Wang, & Jiao (2004), uses relatively simple approach. Instead of classical chromosomes, it uses quantum chromosomes, which means using Qubits instead of classical bits. However, this algorithm does not use my initial intuition of using Grover's algorithm. In this approach, a population of chromosomes are initiated using Qubits. If we consider each Qubit to have amplitudes α and β, the Qubit register of a population of n chromosomes with k genes looks as following.

$$\left[\begin{bmatrix}\alpha\\ \beta\end{bmatrix}_{11}\begin{bmatrix}\alpha_2\\ \beta_2\end{bmatrix}_{12}\cdots\begin{bmatrix}\alpha_3\\ \beta_3\end{bmatrix}_{ij}\cdots\begin{bmatrix}\alpha\\ \beta\end{bmatrix}_{(n-1)(k-1)}\begin{bmatrix}\alpha\\ \beta\end{bmatrix}_{nk}\right]$$

Compared with classical GA, this QGA does not use the concepts of crossover and mutation directly. Instead, it follows a somewhat different approach using rotation gates. $U(\theta)$ depicts the matrix for the rotation.

$$U(\theta)=\begin{bmatrix}cos\theta_i & -sin\theta_i\\ sin\theta_i & cos\theta_i\end{bmatrix}$$

At the initialisation, a random rotation angle is applied to the Qubits. Rotation of Qubits of new generations is based on pre-defined adjustment strategy which is based on the fittest chromosome of the previous generation. Table 1 depicts the adjustment strategy suggested by Wang *et al.* (2013). The $\Delta|theta$ is usually around 0.001π. In this adjustment strategy, one can notice that it uses amplitudes of the best chromosome and the currently considered chromosome.

Figure 3 depicts the complete QGA as a flow chart.According to Wang *et al.* (2013), this QGA has

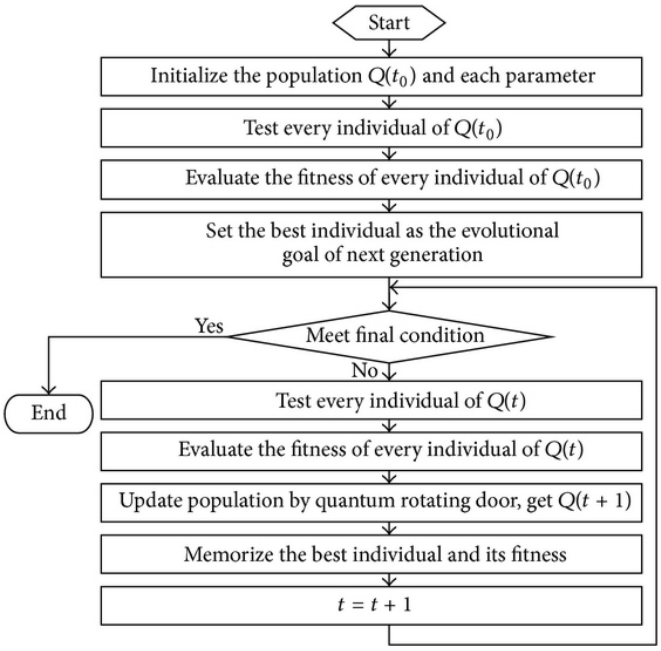

Figure 3: Flow chart of QGA algorithm

x_i	$best_i$	$f(x)>f(best)	$\Delta\theta_i$	$s(\alpha_i, \beta_i)$			
				$\alpha_i\beta_i > 0$	$\alpha_i\beta_i < 0$	$\alpha_i = 0$	$\beta_i = 0$
0	0	FALSE	0	0	0	0	0
0	0	TRUE	0	0	0	0	0
0	1	FALSE	$\Delta\theta_i$	+1	-1	0	±1
0	1	TRUE	$\Delta\theta_i$	-1	+1	±1	0
1	0	FALSE	$\Delta\theta_i$	-1	+1	±1	0
1	0	TRUE	$\Delta\theta_i$	+1	-1	0	±1
1	1	FALSE	0	0	0	0	0
1	1	TRUE	0	0	0	0	0

Table 1: Adjustment strategy to decide the rotating angle

more complex encoding mode and its evolutional approach can cover a wider area than conventional GAs. Furthermore, they claim that this has properties of self-organising, self-adaptive and self-learning.

5 Implementation

As the first approach, a simulation of the QGA is implemented using Python[2] programming language, and then the possibility of implementing it using the Q# language is investigated. The simple function

$$f(x) = |[(x-5)/(2+\sin(x)]|$$

was chosen as the function to optimize. Both simulation and implementation were created with chromosomes containing four genes. Therefore it is represented using a array of four bits(or Qubits). Which means the range of values chromosome can take are 0 -15, integers. In this range, $x = 11$ yields the maximum value for (x). Figure 4 depicts the behaviour of $f(x)$. Therefore the final winning chromosome we are expecting is as follows.

$$\begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}$$

5.1 Python Simulation Implementation

Python version 3.6 and the Numpy package (Oliphant 2006) were used to implement the simulation. As the first step of the simulation Hadamard gate, rotation gate and measurement function were created. The implementation is depicted in Listing 1. A Python list of size two containing amplitudes of a Qubit is used to represent Qubits in the simulation. Hadamard im-

[2]https://docs.python.org/3.6/reference/

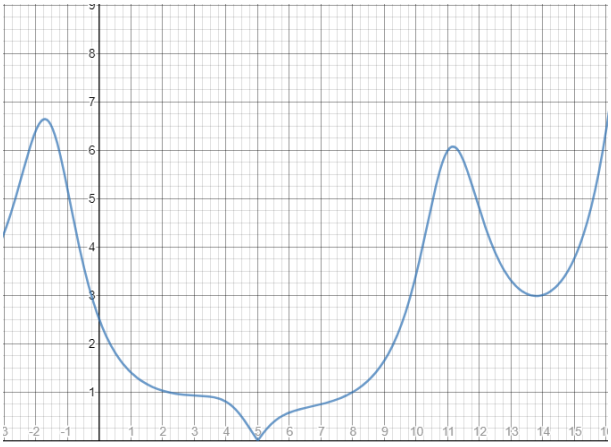

Figure 4: Behaviour of $f(x)$

plementation takes a Qubit as the argument and returns Qubit with Hadamard operation applied. Implemented rotate function also takes a Qubit and theta angle as arguments and returns a Qubit with rotation applied. Implementation of measurement function takes in a Qubit and a threshold. If amplitude zero is larger than equal to the threshold, it returns 0, and otherwise, it returns 1.

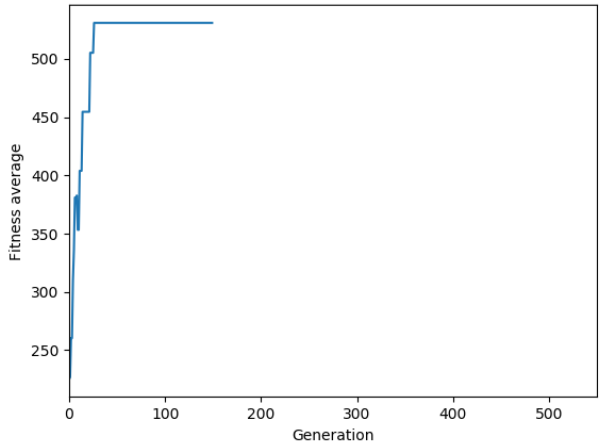

Figure 5: Average fitness score plotted with generation.

After that, functions to choose the winner and diversify the population are implemented. The *choose_winner* function (depicted in Listing 2) takes the entire population as the argument and selects the fittest chromosome of this generation using the helper function *fitness_evaluation*. Here, 0.5 is used as the measurement threshold. The diversify function (depicted in Listing 3) rotates the Qubits based on the fittest chromosome of the previous generation. Instead of the suggested adjustment strategy depicted in Table 1, I have used a straightforward strategy in this function. The $\Delta\theta$ value used in this function is 0.01π.

Using the functions mentioned above with several other functions to connect them, simulations were run with different combinations of population sizes and maximum number of generations. Figure 5 depicts a plot showing average fitness score plotted against generation, in 10 chromosome population with 200 generations. Final average fitness was 459.33 of this run and when the program finished seven chromosomes contained value 11. Running the program, again and again, generated considerably similar results and therefore I assumed that this simulation converges to the optimal value.

5.2 Q# Implementation

Since the simulation was successful, I have investigated the possibility of implementing the same algorithm in Q# language. The first issue I have faced while implementing the algorithm in Q# was implementing the adjustment strategy. The suggested adjustment strategy requires access to amplitudes (α, β) of the Qubits. Observing amplitudes is not possible in Q# environment due to limitations in actual hardware implementations. Therefore I have used the same simple adjustment strategy used in the simulation. The second issue I have faced was the fitness evaluation. Qubits should be measured to evaluate fitness. When it's measured Qubit's state collapses. To overcome this, I have used external data structures to record the rotation angle of Qubits to use them to initialise a new set of chromosomes conating same values as the previous generation.

Listing 4 and Listing 5 show the complete Q# implementation of the discussed QGA, created with the limitations mentioned above. Similar to the simulation, chromosomes with four genes are used. Listing 4 contains helper functions: Rotate, prepareInitAngles, Eval and getFitness. Rotate function rotates the Qubits in the given angles. The prepareInitAngles function creates a set of random angles to be used when rotating the Qubits of the first generation. The getFitness function returns the fitness score of the given chromosome using the helper function eval. Listing 5 contains the QGA function which acts as the primary function in this program. It manages the creation of new generations and diversification of chromosomes.

After running this program with four chromosomes (Four chromosomes are 16 Qubits. It is not possible to increase the number of chromosomes further due to my hardware limitations) for different amounts of

Listing 1: Hadamard gate rotation gate and measurement function implemented using Python

```
def Hadamard (Qubit):
    r=math.sqrt(2.0)
    Qubit_out = [(Qubit[0] + Qubit[1])/r, (Qubit[0] - Qubit[1])/r]
    return Qubit_out

def rotate (Qubit, theta):
    Qubit_out = [Qubit[0]*math.cos(theta) - Qubit[1]*math.sin(theta),
                 Qubit[0]*math.sin(theta) + Qubit[1]*math.cos(theta)]
    return Qubit_out

def Measure(Qubit, threshold):
    if Qubit[0] >= threshold:
        return 0
    return 1
```

Listing 2: Choosing the winner

```
def fitness_evaluation(chromosome):
    x = Measure(chromosome[0], 0.5)*8 + Measure(chromosome[1], 0.5)*4
    + Measure(chromosome[2], 0.5)*2 + Measure(chromosome[3], 0.5)
    y= np.fabs((x-5)/(2+np.sin(x)))
    return y*100

def choose_winner (population):
    for chromosome in population:
        if fitness_evaluation(chromosome) >= fitness_evaluation(GLOBAL_WINNER):
            GLOBAL_WINNER = chromosome
```

Listing 3: Diversify function

```
def divesify (population):
    for chromosome in population:
        if fitness_evaluation(chromosome) < fitness_evaluation(GLOBAL_WINNER):
            for i in range(0, 4):
                if chromosome[i]==0 and GLOBAL_WINNER[i]==1:
                    rotate(chromosome[i], 0.03141592653)
                if chromosome[i]==1 and GLOBAL_WINNER[i]==0:
                    rotate(chromosome[i], -0.03141592653)
```

generations, it did not appear to converge. With more scrutiny, I found that rotating two different Qubits in the same angle does not produce the same outcome. Due to this reason even though my implementation maintains rotation angles of Qubits in external data structures, it cannot re-create the winner chromosome to have the previous value.

6 Conclusion

Evan though the QGA considered can be implemented successfully implemented as a simulation in a classical computer, to the best of my knowledge it is not possible to apply it on a real quantum environment. The adjustment strategy cannot be implemented because it needs access to amplitudes of the Qubits. Fitness evaluation cannot be performed since measurement collapses the quantum state. Even using external data structures to record rotation angles to avoid this issue does not succeed because the Qubits cannot be copied.

References

Holland, J. H. 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press. second edition, 1992.

Meyer-Baese, A., and Schmid, V. 2014. Chapter 5 - genetic algorithms. In Meyer-Baese, A., and Schmid, V., eds., *Pattern Recognition and Signal Analysis in Medical Imaging (Second Edition)*. Oxford: Academic Press, second edition edition. 135 – 149.

Oliphant, T. 2006–. NumPy: A guide to NumPy. USA: Trelgol Publishing.

Udrescu, M.; Prodan, L.; and Vlăduţiu, M. 2006. Implementing quantum genetic algorithms: A solution based on grover's algorithm. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, 71–82. New York, NY, USA: ACM.

Wang, H.; Liu, J.; Zhi, J.; and Fu, C. 2013. The improvement of quantum genetic algorithm and its application on function optimization.

Yang, S.; Wang, M.; and Jiao, L. 2004. A genetic algorithm based on quantum chromosome. In *Proceedings 7th International Conference on Signal Processing, 2004. Proceedings. ICSP '04. 2004.*, volume 2, 1622–1625 vol.2.

Listing 4: Q# helper function implementations

```
operation Rotate(q: Qubit[], angles: Double[]) : Unit {
    H(q[0]);
    Ry(angles[0], q[0]);

    H(q[1]);
    Ry(angles[1], q[1]);

    H(q[2]);
    Ry(angles[2], q[2]);

    H(q[3]);
    Ry(angles[3], q[3]);
}

operation prepareInitAngles(genome: Int, count: Int) : Double[] {
    mutable all_angles = new Double[genome*count];
    for (index in 0 .. genome*count-1) {
        set all_angles[index] = RandomReal(2)*90.0*0.01745;
    }
    return all_angles;
}

open Microsoft.Quantum.Extensions.Math;
open Microsoft.Quantum.Extensions.Convert;
operation Eval(x : Int) : Double{
    let y = AbsD((ToDouble (x)-5.0)/(2.0+Sin(ToDouble (x))));
    return y;
}

operation getFitness(member : Result[]) : Double{
    mutable a0 = 0;
    mutable a1 = 0;
    mutable a2 = 0;
    mutable a3 = 0;
    if (member[0]==One){
        set a0 = 1;
    }
    if (member[1]==One){
        set a1 = 1;
    }
    if (member[2]==One){
        set a2 = 1;
    }
    if (member[3]==One){
        set a3 = 1;
    }

    let x = 8*a0 + 4*a1 + 2*a2 + a3;
    return Eval(x);
}
```

Listing 5: Q# QGA function implementation

```
operation QGA (genome: Int, count: Int, gen: Int) : Result[] {
    mutable winner_angles = new Double[genome];
    mutable all_angles = prepareInitAngles(genome, count);
    mutable r = new Result[genome*count];
    mutable winner_index = -1;
    mutable eval = 0.0;
    let delta_theta = 0.03141592653;

    using (population = Qubit[genome*count]){
        Rotate(population, all_angles);

        set winner_index = -1;
        set eval = 0.0;
        set r =  MultiM (population);
        for (index in 0 .. count-1) {
            let temp = [r[index], r[index+1], r[index+2], r[index+3]];
            let temp_eval = getFitness(temp);
            if (eval < temp_eval) {
                set eval = temp_eval;
                set winner_index = index;
            }
        }
        ResetAll(population);
    }

    for (generation in 0 .. gen) {
    Message($"all_anlges[0] = {all_angles[0]}");
        for (index in 0 .. count-1) {
            if (index != winner_index) {
                if (r[index] == One and r[winner_index] == Zero) {
                    set all_angles[index] = all_angles[index] - delta_theta;
                }
                if (r[index + 1] == One and r[winner_index + 1] == Zero) {
                    set all_angles[index + 1] = all_angles[index + 1] - delta_theta;
                }
                if (r[index + 2] == One and r[winner_index + 2] == Zero) {
                    set all_angles[index + 2] = all_angles[index + 2] - delta_theta;
                }
                if (r[index + 3] == One and r[winner_index + 3] == Zero) {
                    set all_angles[index + 3] = all_angles[index + 3] - delta_theta;
                }

                if (r[index] == Zero and r[winner_index] == One) {
                    set all_angles[index] = all_angles[index] + delta_theta;
                }
                if (r[index + 1] == Zero and r[winner_index + 1] == One) {
                    set all_angles[index + 1] = all_angles[index + 1] + delta_theta;
                }
                if (r[index + 2] == Zero and r[winner_index + 2] == One) {
                    set all_angles[index + 2] = all_angles[index + 2] + delta_theta;
                }
                if (r[index + 3] == Zero and r[winner_index + 3] == One) {
                    set all_angles[index + 3] = all_angles[index + 3] + delta_theta;
                }
            }
        }
```

```
        using (population = Qubit[genome*count]){
            Rotate(population, all_angles);

            set winner_index = -1;
            set eval = 0.0;
            set r =  MultiM (population);
            for (index in 0 .. count-1) {
                let temp = [r[index], r[index+1], r[index+2], r[index+3]];
                let temp_eval = getFitness(temp);
                if (eval < temp_eval) {
                    set eval = temp_eval;
                    set winner_index = index;
                }
            }
            ResetAll(population);
        }
    Message($"Winner is: {winner_index} and eval: {eval} and generation: {generation}");
    }

    Message($"Final Winner is: {winner_index}");
    return [r[winner_index], r[winner_index+1], r[winner_index+2], r[winner_index+3]];
}
```