UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Electrical, Electronic and Computer Engineering

EAI 320 - Intelligent systems

EAI 320 - Practical Assignment 3 Report

Author:
Mohamed Ameen Omar

Student number:
u16055323

March 1, 2018

Contents

List of Figures

List of Tables

1 Introduction

During the third practical for EAI 320, students were tasked to investigate and implement a well known Local search technique known as a Genetic Algorithm. A Local search technique is used in cases wherein the cost of the path to a goal state is irrelevant and the actual goal state is in focus. These techniques, make use of only one node and do not record the path followed by the search to reach the current node [1].

A Genetic Algorithm is part of this class of search techniques. A Genetic Algorithm is used in order to" optimize" a solution to a given problem according to a certain predefined set of criteria, or to find a feasible solution, in an extremely large - almost infinite search space.

A Genetic Algorithm is a type of Local search that was inspired by Darwin's theory of evolution - that the strongest survive and the weak die off. This strength stems from the combining of the genes of both parents of the child. Genetic Algorithms attempt to mimic this by making use of two parent "genes" combining randomly to produce a child. Random mutation (amending of the genes of the children) and random selection of parents based off a "strength" value to merge are all included in this type of search.

Genetic Algorithms are still being researched today and the conditions under which they are most effective in being complete or optimal are not yet fully known. During this practical, students were tasked with evaluating the performance of their implemented Genetic Algorithms on an optimization search problem.

2 Problem Definition

Students were given a 16 x 16 grid representing a particular city as shown in figure 1 below.

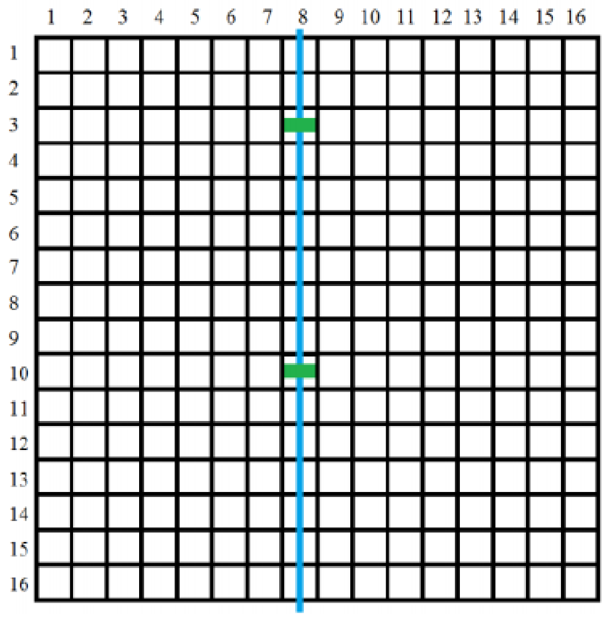
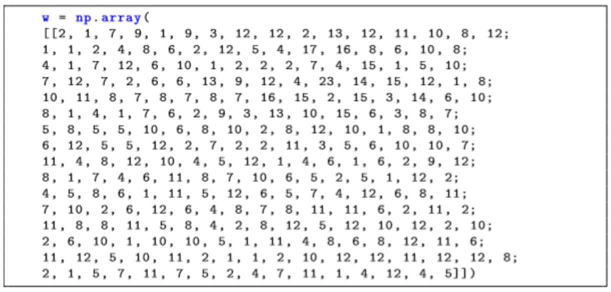


Figure 1: Grid representing the city - provided in the practical specification

A hospital is to be placed in one of the blocks in the city, students are required to find the best or optimal position for the hospital. The optimal position for the new hospital is the one in which the response time to any medical emergency in any part of the city (in any block in the grid) is minimized.

In figure 1, the blue line down column 8 represents a river separating two parts of the city. There are two bridge represented by green lines in the blocks of the bridges. These bridges are located at blocks [3,8] and [9,8]. In order to cross over from one side of the river to the other, a person would first need to go to a bridge and then from the bridge to the destination block, stepping through the river without crossing a bridge is strictly prohibited in this problem.

In order to implement a Genetic algorithm, students are given the results of a survey conducted in the city. This survey evaluated the amount of medical emergencies that occurred in a particular block, over a period of a year in the city. The results are given in the form of a numpy (type of Python library) array shown in figure 2. Each entry in the array, represents the results of the survey for the corresponding block in the grid.



```
w = np.array(
[[2, 1, 7, 9, 1, 9, 3, 12, 12, 2, 13, 12, 11, 10, 8, 12;
1, 1, 2, 4, 8, 6, 2, 12, 5, 4, 17, 16, 8, 6, 10, 8;
4, 1, 7, 12, 6, 10, 1, 2, 2, 2, 7, 4, 15, 1, 5, 10;
7, 12, 7, 2, 6, 6, 13, 9, 12, 4, 23, 14, 15, 12, 1, 8;
10, 11, 8, 7, 8, 7, 8, 7, 16, 15, 2, 15, 3, 14, 6, 10;
8, 1, 4, 1, 7, 6, 2, 9, 3, 13, 10, 15, 6, 3, 8, 7;
5, 8, 5, 5, 10, 6, 8, 10, 2, 8, 12, 10, 1, 8, 8, 10;
6, 12, 5, 5, 12, 2, 7, 2, 2, 11, 3, 5, 6, 10, 10, 7;
11, 4, 8, 12, 10, 4, 5, 12, 1, 4, 6, 1, 6, 2, 9, 12;
8, 1, 7, 4, 6, 11, 8, 7, 10, 6, 5, 2, 5, 1, 12, 2;
4, 5, 8, 6, 1, 11, 5, 12, 6, 5, 7, 4, 12, 6, 8, 11;
7, 10, 2, 6, 12, 6, 4, 8, 7, 8, 11, 11, 6, 2, 11, 2;
11, 8, 8, 11, 5, 8, 4, 2, 8, 12, 5, 12, 10, 12, 2, 10;
2, 6, 10, 1, 10, 10, 5, 1, 11, 4, 8, 6, 8, 12, 11, 6;
11, 12, 5, 10, 11, 2, 1, 1, 2, 10, 12, 12, 11, 12, 12, 8;
2, 1, 5, 7, 11, 7, 5, 2, 4, 7, 11, 1, 4, 12, 4, 5]])
```

Figure 2: Grid representing the city - provided in the practical specification

The average response time for a block is given by $f(d) = 2.4 + 4.5d$ where the variable d, is the Euclidean distance from the hospital to the block. The cost function represents the total cost associated with a particular location chosen for the hospital and is given by:

$$C_{loc} = \sum_{i}^{16} \sum_{j}^{16} \omega_{ij} \cdot (2.4 + 4.5d) \tag{1}$$

Students were required to use the given information in order to compute the distance between a block and the hospital as well as to plot the three dimensional cost surface of the cost function given by equation (1). The global minimum of the cost surface is to be recorded as well in order to evaluate the result of the Genetic Algorithm.

The Genetic algorithm is to find the optimal location for the hospital and all parameters such as population size, mutation rate, etc were left open for students to investigate.

3 Implementation and Methodology

In my implementation, two Python classes were used in order to organize code and make the task easier. All source can be found in Appendix A.

The *member* class contains just two variables, a row and a column variable. These collectively represent the particular block to which we are referring. The *hospitalOptimization* class contains all the other functions and "helper" functions needed in order to implement the Genetic Algorithm and compute the three dimensional surface plot.

A member variable called *grid* is also used in order to store the given numpy array to aid in computing the cost function given. When computing the Euclidean distance between two blocks, a check is first done to determine if the two blocks are on opposite ends of the river, if so, the distance from the current block to a bridge is added to the distance from the same bridge to the detestation block. The same is done for the second bridge and the shorter of the two is used as the Euclidean distance.

3.1 Three dimensional cost plot

A *plotCostSurface()* function within the *hospitalOptimization* class used in order to plot the three dimensional surface plot of the cost function. The *matplotlib* Python libraries were used in order to plot the surface. The surface is shown in figure 5, under results. The *matplotlib "colormap"* function was also used in order to color the plot according the magnitude of the cost of each block.

The x co-ordinate represents the row number and the y co-ordinate represents the column, with indexing beginning from 0 instead of 1. The cost for each block is first computed into a numpy array using the *numpy.array()* library function and the *costFunction()* class method, this method takes in the x and y co-ordinates for a particular block and computes its cost according to the cost function. The function also keeps track of the location that has the smallest cost during each run and stores it in two class variables, *minimumCost* and *minLocation.* Please note that when evaluating whether a location has the lowest cost, all blocks representing the river or a bridge are ignored due to the fact that these locations logically would not be feasible to be used to build a new hospital in. A code snippet of the *costFunction()* class method and the *plotCostSurface()* function are shown in figures 3 and 4 respectively.

```python
#a fucntion that gives the totalCost for a particular location as a hospital
    def costFunction(self, row,col):
        totalCost = 0
        for i in range(0,16):
            for j in range(0,16):
                totalCost += ( self.grid[i][j] * (self.averageResponseTime(i,j,row,col) ) )
        if(self.minimumCost == -1):
            if(col != 7):
                self.minLocation = [row,col]
                self.minimumCost = totalCost
        #minimum must not be in the river or be a bridge
        elif( (self.minimumCost > totalCost) and (col != 7) ):
            self.minLocation = [row,col]
            self.minimumCost = totalCost
        return totalCost
```

Figure 3: Program Source Code for the *costFunction()* class method [Python]

```python
def plotCostSurface(self):
        #row = x
        #col = y
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        x = y = np.arange(0, 16, 1)
        Xmesh, Ymesh = np.meshgrid(x, y)
        z = np.array( [self.costFunction(x, y) for x, y in zip(np.ravel(Xmesh), np.ravel(Ymesh) )] )
        Zmesh = z.reshape(Ymesh.shape)
        ax.plot_surface(Xmesh, Ymesh, Zmesh, rstride=1, cstride=1, cmap=cm.spectral,
                        linewidth=0, antialiased=True)
        ax.set_xlabel('X-Coordinate')
        ax.set_ylabel('Y-Coordinate')
        ax.set_zlabel('Cost')
        ax.set_title("COST SURFACE PLOT\n")
        print("OPTIMAL LOCATION IS :", self.minLocation)
        print("OPTIMAL COST IS: ", self.minimumCost)
        sys.stdout.flush()
```

Figure 4: Program Source Code for the *plotCostSurface* function [Python]

3.2 Genetic Algorithm

In order to perform my Genetic Algorithm the *member* class is used to represent each member of my population stored in a *numpy* array - *myPopulation*. Additional variables such as population size, number of generations and mutation rate are stored in *populationSize*, *numGenerations* and *mutationRate* respectively. The row and column for each member is binary encoded in order to aid in the "crossing over" of chromosomes between two parents when producing a child for the new generation.

The population size has been varied along with the other algorithm parameters in order to investigate the "best" set in finding the correct or most optimal result for the given problem outline in section 2 - Problem Definition. The complete algorithm is run a variable amount of times and the result is compared to the actual solution, thereafter the average success rate of the algorithm is computed and outputted.

First the true optimal location is computed and stored in order to do a comparison later after-which the actual genetic algorithm is run until the specified number of generations has been produced.

Initial Population

The initial population is computed very simply, a random number is generated in the range of 0 to 15 and binary encoded with 8 bits. During this process, when generating the column index, a check is done to ensure that the member is not a bridge or a block in the river. The reason for this, is that the purpose of the algorithm is to find a usable solution, a solution found to be in the river or a bridge is not feasible, since this location would not be appropriate to build a new hospital for obvious reasons. A check is also done to ensure no member is repeated in the initial population in order to have the largest *gene pool* possible. Each new member generated is then stored in the *numpy* array.

Selection of parents and crossing over

After the generation has been produced, new parents are chosen, this is done by making use of the given cost function as a fitness function for each member. One computing the cost function, the member ares sorted from lowest to highest "cost", after-which the lowest cost is reassigned to the member with the highest cost in the generation. This is done in order to produce a weighted average probability for a member to be chosen as a parent for the next generation. The sorting of the members from lowest to highest also aid in retrieving the solution found by each generation or rather each iteration.

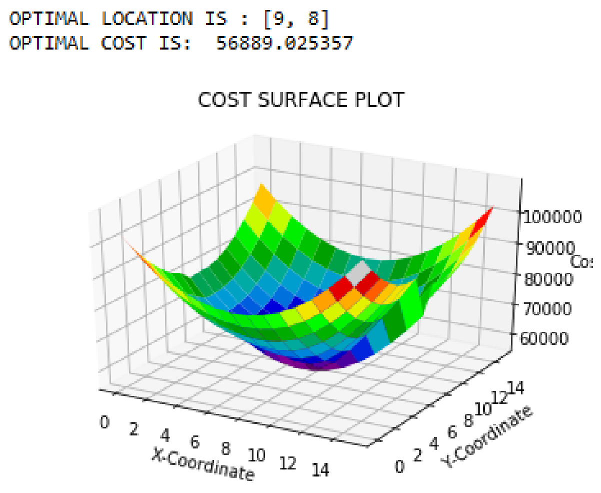
The *numpy.random.choice* from the *numpy* Python binary is used to find the pairs of parents for each generation. The weighted average provabilities are stored in an array and passed in as a parameter into the function, a check is done in order to ensure that no parent in a parent is repeated, meaning that, no parent or member produces a child with itself. The number of parents chosen is half that of the specified population size and each pair of parents produces two children.

In order to produced the children, a random number is generated between 0 and 8 and the bits from one parent are substituted for the other parents are the index produced. The result is also checked, in the case of column "crossing over", if the child produced would be in column 7 (in the grid this value would be 8), if true, this is done again in order to produce another child, since this child is invalid. A random number is also generated and compared to the mutation rate, if certain criteria are met, a mutation to either the row or column (50/50 chance for either) occurs. The mutation implemented is just producing a new random number and changing the row or column of the child to the newly produced row or column.

The process is repeated for every pair of parents until the entire new generation is produced.

4 Results

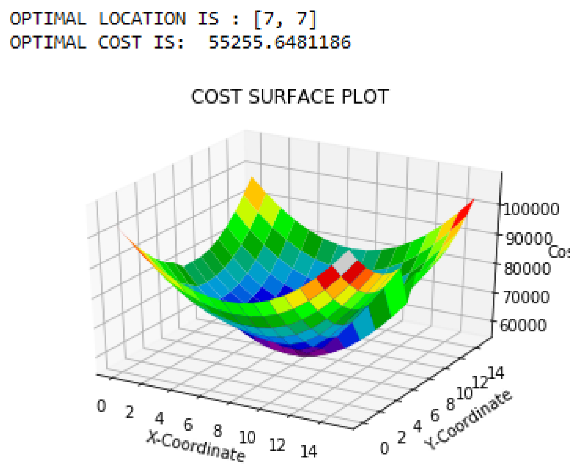
Figure 5 shows the output of the three dimensional cost surface plot as well as the valid optimal solution for the new hospital.



```
OPTIMAL LOCATION IS : [9, 8]
OPTIMAL COST IS:  56889.025357
```

Figure 5: Three dimensional cost surface plot and the valid global minimum

Figure 6 shows the output of the three dimensional cost surface plot as well as the optimal solution for the new hospital if the source code is manipulated to have a location in the river as a solution.



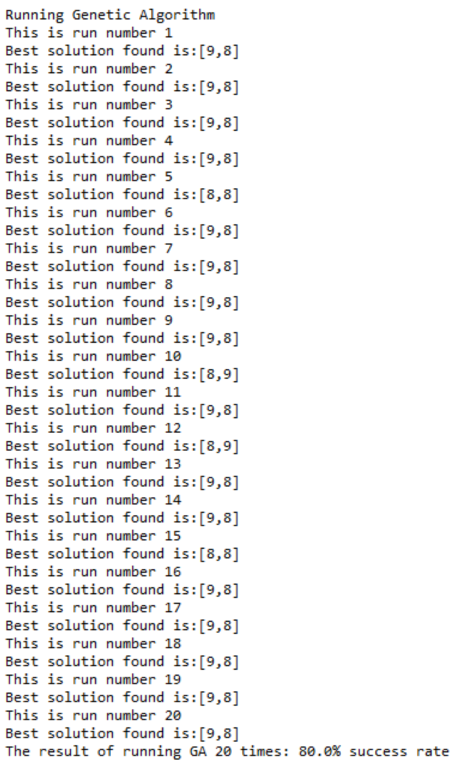
```
OPTIMAL LOCATION IS : [7, 7]
OPTIMAL COST IS:  55255.6481186
```

Figure 6: Three dimensional cost surface plot and the *"TRUE"* global minimum

The table below shows the success rate of the Genetic Algorithm 20 times, with varying Genetic Algorithm parameters. The console outputs for each result is also shown in figures 7 to 12.

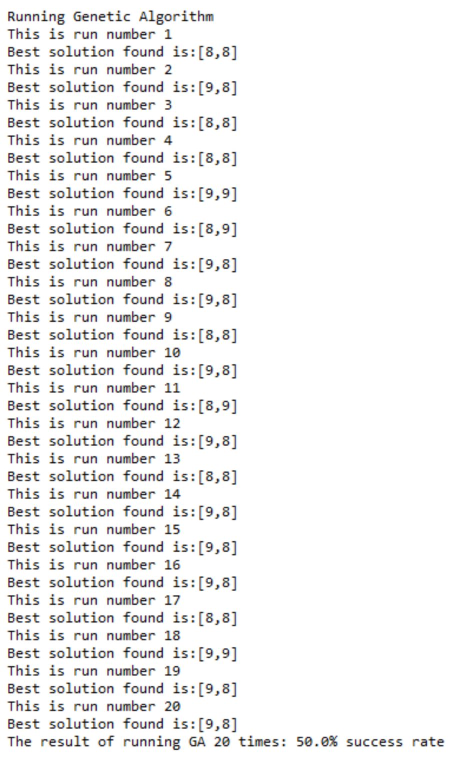
Population Size	120	120	100	80	80	80
Number of Generations	10	10	5	5	10	10
Mutation Rate	1%	20%	1%	1%	1%	20%
Success Rate	80%	50%	55%	75%	55%	45%

Table 1: Table of results



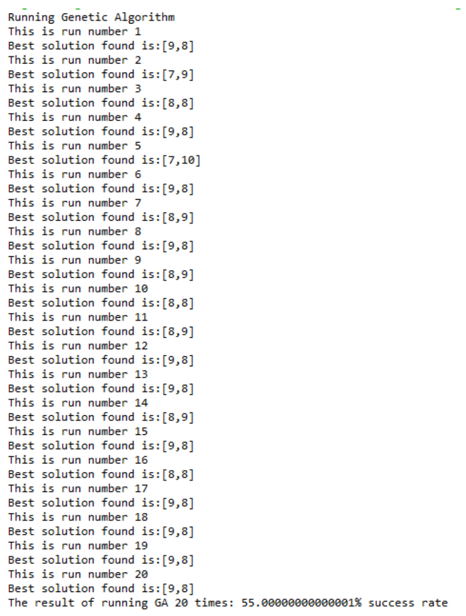
```
Running Genetic Algorithm
This is run number 1
Best solution found is:[9,8]
This is run number 2
Best solution found is:[9,8]
This is run number 3
Best solution found is:[9,8]
This is run number 4
Best solution found is:[9,8]
This is run number 5
Best solution found is:[8,8]
This is run number 6
Best solution found is:[9,8]
This is run number 7
Best solution found is:[9,8]
This is run number 8
Best solution found is:[9,8]
This is run number 9
Best solution found is:[9,8]
This is run number 10
Best solution found is:[8,9]
This is run number 11
Best solution found is:[9,8]
This is run number 12
Best solution found is:[8,9]
This is run number 13
Best solution found is:[9,8]
This is run number 14
Best solution found is:[9,8]
This is run number 15
Best solution found is:[8,8]
This is run number 16
Best solution found is:[9,8]
This is run number 17
Best solution found is:[9,8]
This is run number 18
Best solution found is:[9,8]
This is run number 19
Best solution found is:[9,8]
This is run number 20
Best solution found is:[9,8]
The result of running GA 20 times: 80.0% success rate
```

Figure 7: Population size of 120, 10 generations and a 1% mutation rate



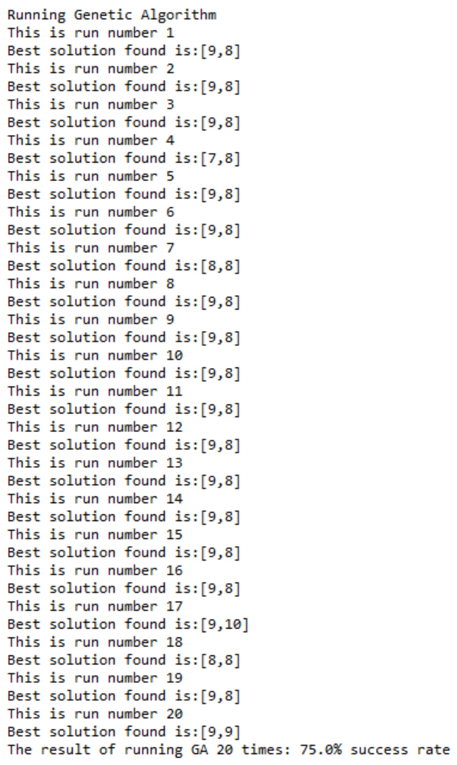
```
Running Genetic Algorithm
This is run number 1
Best solution found is:[8,8]
This is run number 2
Best solution found is:[9,8]
This is run number 3
Best solution found is:[8,8]
This is run number 4
Best solution found is:[8,8]
This is run number 5
Best solution found is:[9,9]
This is run number 6
Best solution found is:[8,9]
This is run number 7
Best solution found is:[9,8]
This is run number 8
Best solution found is:[9,8]
This is run number 9
Best solution found is:[8,8]
This is run number 10
Best solution found is:[9,8]
This is run number 11
Best solution found is:[8,9]
This is run number 12
Best solution found is:[9,8]
This is run number 13
Best solution found is:[8,8]
This is run number 14
Best solution found is:[9,8]
This is run number 15
Best solution found is:[9,8]
This is run number 16
Best solution found is:[9,8]
This is run number 17
Best solution found is:[8,8]
This is run number 18
Best solution found is:[9,9]
This is run number 19
Best solution found is:[9,8]
This is run number 20
Best solution found is:[9,8]
The result of running GA 20 times: 50.0% success rate
```

Figure 8: Population size of 120, 10 generations and a 20% mutation rate



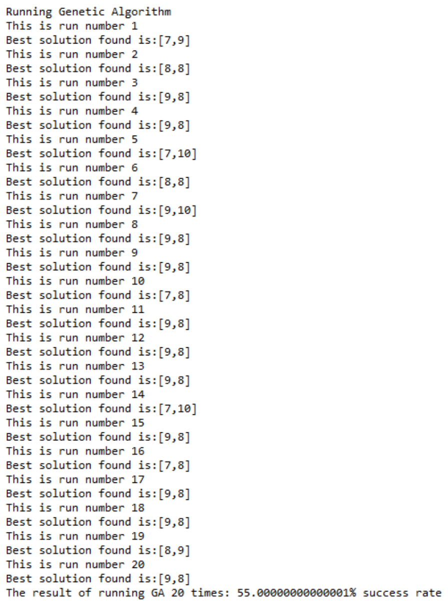
```
Running Genetic Algorithm
This is run number 1
Best solution found is:[9,8]
This is run number 2
Best solution found is:[7,9]
This is run number 3
Best solution found is:[8,8]
This is run number 4
Best solution found is:[9,8]
This is run number 5
Best solution found is:[7,10]
This is run number 6
Best solution found is:[9,8]
This is run number 7
Best solution found is:[8,9]
This is run number 8
Best solution found is:[9,8]
This is run number 9
Best solution found is:[8,9]
This is run number 10
Best solution found is:[8,8]
This is run number 11
Best solution found is:[8,9]
This is run number 12
Best solution found is:[9,8]
This is run number 13
Best solution found is:[9,8]
This is run number 14
Best solution found is:[8,9]
This is run number 15
Best solution found is:[9,8]
This is run number 16
Best solution found is:[8,8]
This is run number 17
Best solution found is:[9,8]
This is run number 18
Best solution found is:[9,8]
This is run number 19
Best solution found is:[9,8]
This is run number 20
Best solution found is:[9,8]
The result of running GA 20 times: 55.00000000000001% success rate
```

Figure 9: Population size of 100, 5 generations and a 1% mutation rate



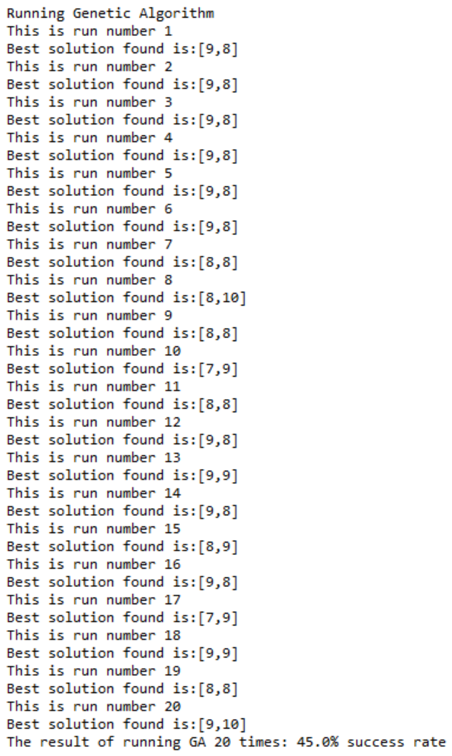
```
Running Genetic Algorithm
This is run number 1
Best solution found is:[9,8]
This is run number 2
Best solution found is:[9,8]
This is run number 3
Best solution found is:[9,8]
This is run number 4
Best solution found is:[7,8]
This is run number 5
Best solution found is:[9,8]
This is run number 6
Best solution found is:[9,8]
This is run number 7
Best solution found is:[8,8]
This is run number 8
Best solution found is:[9,8]
This is run number 9
Best solution found is:[9,8]
This is run number 10
Best solution found is:[9,8]
This is run number 11
Best solution found is:[9,8]
This is run number 12
Best solution found is:[9,8]
This is run number 13
Best solution found is:[9,8]
This is run number 14
Best solution found is:[9,8]
This is run number 15
Best solution found is:[9,8]
This is run number 16
Best solution found is:[9,8]
This is run number 17
Best solution found is:[9,10]
This is run number 18
Best solution found is:[8,8]
This is run number 19
Best solution found is:[9,8]
This is run number 20
Best solution found is:[9,9]
The result of running GA 20 times: 75.0% success rate
```

Figure 10: Population size of 80, 5 generations and a 1% mutation rate



```
Running Genetic Algorithm
This is run number 1
Best solution found is:[7,9]
This is run number 2
Best solution found is:[8,8]
This is run number 3
Best solution found is:[9,8]
This is run number 4
Best solution found is:[9,8]
This is run number 5
Best solution found is:[7,10]
This is run number 6
Best solution found is:[8,8]
This is run number 7
Best solution found is:[9,10]
This is run number 8
Best solution found is:[9,8]
This is run number 9
Best solution found is:[9,8]
This is run number 10
Best solution found is:[7,8]
This is run number 11
Best solution found is:[9,8]
This is run number 12
Best solution found is:[9,8]
This is run number 13
Best solution found is:[9,8]
This is run number 14
Best solution found is:[7,10]
This is run number 15
Best solution found is:[9,8]
This is run number 16
Best solution found is:[7,8]
This is run number 17
Best solution found is:[9,8]
This is run number 18
Best solution found is:[9,8]
This is run number 19
Best solution found is:[8,9]
This is run number 20
Best solution found is:[9,8]
The result of running GA 20 times: 55.00000000000001% success rate
```

Figure 11: Population size of 80, 10 generations and a 1% mutation rate



```
Running Genetic Algorithm
This is run number 1
Best solution found is:[9,8]
This is run number 2
Best solution found is:[9,8]
This is run number 3
Best solution found is:[9,8]
This is run number 4
Best solution found is:[9,8]
This is run number 5
Best solution found is:[9,8]
This is run number 6
Best solution found is:[9,8]
This is run number 7
Best solution found is:[8,8]
This is run number 8
Best solution found is:[8,10]
This is run number 9
Best solution found is:[8,8]
This is run number 10
Best solution found is:[7,9]
This is run number 11
Best solution found is:[8,8]
This is run number 12
Best solution found is:[9,8]
This is run number 13
Best solution found is:[9,9]
This is run number 14
Best solution found is:[9,8]
This is run number 15
Best solution found is:[8,9]
This is run number 16
Best solution found is:[9,8]
This is run number 17
Best solution found is:[7,9]
This is run number 18
Best solution found is:[9,9]
This is run number 19
Best solution found is:[8,8]
This is run number 20
Best solution found is:[9,10]
The result of running GA 20 times: 45.0% success rate
```

Figure 12: Population size of 80, 10 generations and a 20% mutation rate

5 Discussion

The Surface plot is shown in figure 5 with the valid optimal solution the problem proposed. As we can see from figure 5 and figure 6, the actual global minimum of the cost surface is at a location found in the river, this however is not a feasible solution since a hospital cannot be built in a river. The search tree for this problem is extremely complex, due to the tremendous amount of possible state solutions, the river, the bridge locations and the scale of judgment of the optimal solution. Therefore the usage of a Local search, more specifically a Genetic Algorithm is justified.

As we can see from table 1, a balance between population size, number of generations and mutation rate is needed in order for the GA to more often than not converge to the valid optimal solution. However, from figure 7 to 12 it is also evident that when the algorithm does not converge to the *best* solution, it does however converge to a feasible solution, very close to the optimal solution. Thus, even when the best solution is not found, we can have confidence in the solution proposed by the algorithm since it would always find one of, if not the best solution.

From multiple runs of the algorithm, it does require approximately 3-5 generations to find the solution, with more generations being a waste of processing power since it eventually diverges away and converges back to one of the optimal solutions. With a higher generation count and a lower population size, we would see that the *gene pool* would become stagnant and mutation playing an important role to reintroduce variety into the *gene pool* for the following generations to converge to the optimal solution.

A large population size would require more memory and more processing power due to the fact that multiple checks are done on each random number generated for each chromosome increasing the *call stack* size and increasing the amount of computation needed to encode and decode the 8-bit binary representations. The population size also needs to be reasonable, since if the population size of our algorithm is very close to the total number of possible solutions, the usage of a Genetic Algorithm would be futile as a Brute force approach would take a similar time and would guarantee the best result, whereas the same cannot be said for a Genetic Algorithm. This reemphasizes the need to find a balance between the different GA parameters in order to efficiently find a feasible solution (with the hope of it being the optimal) on a similar problem, done on a much larger scale.

If given the opportunity to re-visit my implementation, I would investigate different approaches to the chromosome encoding, since the computation required to encode and decode the binary values is substantial and does slow down the algorithm by noticeable amount. A better mutation method would also be investigated via trial and error since the current implementation is not the most efficient or the best, it simply replaces the row or column of the individual (member) with another random one. If given more time I would also optimize my source code to run faster, since when attempting to run it 100 times with a population size of 100 and a generation count of 1000, I estimated after the third run that it would complete after 9 hours. It is extremely slow and inefficient since for practical problems we require a balance between both accuracy and time complexity.

6 Conclusion

A genetic algorithm does not have a particular format or a ”general” implementation that could be altered slightly in order to solve every problem for which it is used. This type of algorithm is problem specific. The developer would need to tailor every single aspect or element of the algorithm to the problem at hand. The chromosome structure, the population density, the number of generation, the crossing over process, how the mutation is performed, the fitness function used to evaluate each member of the population, the mutation rate and the number of generations would all need to be carefully thought out before implementing a Genetic Algorithm - in order to ensure both accurate and relatively fast solutions. As we can see, this type of search is extremely powerful and can be very accurate when implemented well. It opens up a whole new class of practical problems that can now be solved.

7 Appendix A: Python Code

```
#Mohamed Ameen Omar
#u16055323
#EAI Practical 3
#2018
import math
import numpy as np
import time
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import sys
import random
import matplotlib
from mpl_toolkits.mplot3d import Axes3D
class member:
    def __init__(self,location):
        #stored as binary
        self.row = location[0]
        self.col = location[1]
```

Figure 13: Program Source Code 1 of 12 [Python]

```python
class hospitalOptimization:
    def __init__(self):
        #river is at index 7!!!!!!!!!!
        self.grid = np.array(
                                    [[2, 1, 7, 9, 1, 9, 3, 12, 12, 2, 13, 12, 11,
   10, 8, 12],
                                    [ 1, 1, 2, 4, 8, 6, 2, 12, 5, 4, 17, 16, 8, 6,
    10, 8],
                                    [ 4, 1, 7, 12, 6, 10, 1, 2, 2, 2, 7, 4, 15, 1,
    5, 10],
                                    [ 7, 12, 7, 2, 6, 6, 13, 9, 12, 4, 23, 14, 15,
    12, 1, 8],
                                    [ 10, 11, 8, 7, 8, 7, 8, 7, 16, 15, 2, 15, 3,
   14, 6, 10],
                                    [ 8, 1, 4, 1, 7, 6, 2, 9, 3, 13, 10, 15, 6, 3,
    8, 7],
                                    [ 5, 8, 5, 5, 10, 6, 8, 10, 2, 8, 12, 10, 1,
   8, 8, 10],
                                    [ 6, 12, 5, 5, 12, 2, 7, 2, 2, 11, 3, 5, 6,
   10, 10, 7],
                                    [ 11, 4, 8, 12, 10, 4, 5, 12, 1, 4, 6, 1, 6,
   2, 9, 12],
                                    [ 8, 1, 7, 4, 6, 11, 8, 7, 10, 6, 5, 2, 5, 1,
   12, 2],
                                    [ 4, 5, 8, 6, 1, 11, 5, 12, 6, 5, 7, 4, 12, 6,
    8, 11],
                                    [ 7, 10, 2, 6, 12, 6, 4, 8, 7, 8, 11, 11, 6,
   2, 11, 2],
                                    [ 11, 8, 8, 11, 5, 8, 4, 2, 8, 12, 5, 12, 10,
   12, 2, 10],
                                    [ 2, 6, 10, 1, 10, 10, 5, 1, 11, 4, 8, 6, 8,
   12, 11, 6],
                                    [ 11, 12, 5, 10, 11, 2, 1, 1, 2, 10, 12, 12,
   11, 12, 12, 8],
                                    [ 2, 1, 5, 7, 11, 7, 5, 2, 4, 7, 11, 1, 4, 12,
    4, 5]])
        self.minimumCost = -1
        self.minLocation = []
        self.bridge1 = [2,7] #bridge locations
        self.bridge2 = [9,7] #bridge locations
        self.myPopulation = [] #does not include any locations in river
        self.populationSize = 120
        self.numGenerations = 10
        self.mutationRate = 0.01
        np.random.seed(int(time.time() ))

    #get the distance between two blocks
    def getDistance(self,row,col,row1,col1):
        if(self.sameSide(col,col1) == True):
            return self.getEuclideanDistance(row,col,row1,col1)
        else:
            return self.getDistanceViaBridge(row,col,row1,col1)
```

Figure 14: Program Source Code 2 of 12 [Python]

```python
#returns a boolean whether blocks are on the same side of the river
    def sameSide(self,col,col1):
        if( ( (col <= 7) and (col1 <= 7) )  or ( (col >= 7) and (col1 >=
    7) ) ):
            return True
        else:
            return False

    #gets Euclidean distance between the two
    def getEuclideanDistance(self,row,col,row1,col1):
        distance = 0
        if(row > row1):
            temp = row-row1
            distance += (temp*temp)
        else:
            temp = row1-row
            distance += (temp*temp)
        if(col > col1):
            temp = col-col1
            distance += (temp*temp)
        else:
            temp = col1-col
            distance += (temp*temp)
        return(math.sqrt(distance))
```

Figure 15: Program Source Code 3 of 12 [Python]

```python
#get the distance between two points via a bridge
    def getDistanceViaBridge(self,row,col,row1,col1):
        temp1 = self.getEuclideanDistance(row,col,self.bridge1[0],self.bridge1[1])
        temp1 += self.getEuclideanDistance(self.bridge1[0],self.bridge1[1],row1,col1)
        temp2 = self.getEuclideanDistance(row,col,self.bridge2[0],self.bridge2[1])
        temp2 += self.getEuclideanDistance(self.bridge2[0],self.bridge2[1],row1,col1)
        if(temp1 > temp2):
            return temp2
        else:
            return temp1

    def getClosestBridge(self,row):
        #if distance from my row to bridge1 is greater than didtance to bridge2
        if( math.fabs(row-self.bridge1[0])  > math.fabs(row - self.bridge2[0]) ):
            return self.bridge2
        else:
            return self.bridge1

    #a fucntion that gives the totalCost for a particular location as a hospital
    def costFunction(self, row,col):
        totalCost = 0
        for i in range(0,16):
            for j in range(0,16):
                totalCost += ( self.grid[i][j] * (self.averageResponseTime(i,j,row,col) ) )
        if(self.minimumCost == -1):
            if(col != 7):
                self.minLocation = [row,col]
                self.minimumCost = totalCost
        #minimum must not be in the river or be a bridge
        elif( (self.minimumCost > totalCost) and (col != 7) ):
            self.minLocation = [row,col]
            self.minimumCost = totalCost
        return totalCost
```

Figure 16: Program Source Code 4 of 12 [Python]

```python
#average repsonse time equation given to us
    def averageResponseTime(self, row,col,row1,col1):
        dist = self.getDistance(row,col,row1,col1)
        temp = 2.4 + (4.5 * dist)
        return( temp )

    def plotCostSurface(self):
        #row = x
        #col = y
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        x = y = np.arange(0, 16, 1)
        Xmesh, Ymesh = np.meshgrid(x, y)
        z = np.array( [self.costFunction(x, y) for x, y in zip(np.ravel(
    Xmesh), np.ravel(Ymesh) )] )
        Zmesh = z.reshape(Ymesh.shape)
        ax.plot_surface(Xmesh, Ymesh, Zmesh, rstride=1, cstride=1, cmap=cm
    .spectral,
                        linewidth=0, antialiased=True)
        ax.set_xlabel('X-Coordinate')
        ax.set_ylabel('Y-Coordinate')
        ax.set_zlabel('Cost')
        ax.set_title("COST SURFACE PLOT\n")
        print("OPTIMAL LOCATION IS :", self.minLocation)
        print("OPTIMAL COST IS: ", self.minimumCost)
        sys.stdout.flush()

    #gets best location for the hospital
    def getPrimeLocation(self):
        for i in range(0,16):
            for j in range(0,16):
                self.costFunction(i,j)

    def getInitialPopulation(self):
        for x in range(0,self.populationSize):
            loc = self.getRandomBlock()
            loc = self.verifyLocation(loc) #not repeated
            self.myPopulation.append( member(loc) )
```

Figure 17: Program Source Code 5 of 12 [Python]

```python
#gets a random location for the population
    def getRandomBlock(self):
        return [self.getRandomRow(),self.getRandomCol()]
    #in binary
    def getRandomRow(self):
        row = "{0:b}".format(np.random.randint(0, 16))
        while(len(row) < 8):
            row = "0"+row
        return row
    #in binary
    def getRandomCol(self):
        col = np.random.randint(0,16)
        while(col == 7):
            col = np.random.randint(0,16)
        col = "{0:b}".format(col)
        while(len(col) < 8):
            col = "0" + col
        return col

    #verifies a location is not repeated
    def verifyLocation(self,loc):
        again = True
        if(self.myPopulation != []):
            while(again == True):
                for x in range(0,len(self.myPopulation)):
                    if( (self.myPopulation[x].row == loc[0]) and self.
    myPopulation[x].col == loc[1] ):
                        loc = self.getRandomBlock()
                        break
                again = False
        return loc

    def binaryToInt(self, b = "{0:b}"):
        return int(b,2)
```

Figure 18: Program Source Code 6 of 12 [Python]

```python
#prints the population
    def printPopulation(self):
        for x in range(0,len(self.myPopulation)):
            print(self.binaryToInt(self.myPopulation[x].row), end="")
            print(",",end="")
            print(self.binaryToInt(self.myPopulation[x].col))

    def geneticAlgorithm(self):
        run = 100
        success = 0
        for z in range(0,run):
            self.myPopulation = []
            self.getPrimeLocation()
            for x in range(0,self.numGenerations):
                sys.stdout.flush()
                if(self.myPopulation == []):
                    self.getInitialPopulation()
                invertedCosts = self.sortFitness() #a list of highest to lowest cost
                pairs = self.getBreedingPairs(self.getProbabilities(invertedCosts))#breedingPairs,consecutive not repeated
                newPop = self.breedPairs(pairs)#next generation has been proudued
                self.myPopulation = newPop
            self.sortFitness()
            print("This is run number",z+1)
            print("Best solution found is:", end = "")
            print("[", end = "")
            print(self.binaryToInt(self.myPopulation[0].row), end = "")
            print(",", end = "")
            print(self.binaryToInt(self.myPopulation[0].col), end = "")
            print("]")
            if(self.binaryToInt(self.myPopulation[0].row) == 9 and self.binaryToInt(self.myPopulation[0].col) == 8):
                success += 1
        #self.printPopulation()
        print("The result of running GA", run, end = "")
        print(" times:", (success/run)*100, end = "")
        print("% success rate")
```

Figure 19: Program Source Code 7 of 12 [Python]

```python
#returns a list of "new" costs assosiated with the "new" pop
    def sortFitness(self):
        list = []
        for x in range(0,len(self.myPopulation)):
            row = self.binaryToInt(self.myPopulation[x].row)
            col = self.binaryToInt(self.myPopulation[x].col)
            cost = self.costFunction(row,col)
            list.append((cost,self.myPopulation[x]))
        #its in order of lowest to highest cost
        list = self.sortList(list)
        newPop = []
        invertedCosts = []
        y = len(list) -1
        for x in range(0,len(list)):
            newPop.append( (list[x])[1] )#sort my population in asceding
   order of cost
            invertedCosts.append( (list[y])[0] )
            y = y-1
        self.myPopulation = newPop
        #mypoualtion[0] is now the location with the lowest cost
        return invertedCosts

    #sorts a list of tuples in ascending order
    def sortList(self,list):
        sorted = False
        while(sorted == False):
            swap = False
            for x in range(0,len(list) -1):
                if(list[x][0] > list[x+1][0]):
                    swap = True
                    temp = list[x]
                    list[x] = list[x+1]
                    list[x+1] = temp
            if(swap == False):
                sorted = True
        return list
```

Figure 20: Program Source Code 8 of 12 [Python]

```
#get probabilities based off weighted avergae
    def getProbabilities(self, costs):
        sum = 0
        for x in range(len(costs)):
            sum+= costs[x]
        probs = []
        for x in range(len(costs)):
            prob = costs[x]
            prob = (costs[x]/sum) #probility
            probs.append(prob)
        return probs

    #get random breeding pairs based off of probilities with no repeated
 in a pair
    def getBreedingPairs(self,probs):
        pairs = []
        pairs = np.random.choice(self.myPopulation,self.populationSize,
 True,probs)
        x = 0
        while(x < len(pairs)-1):
            while( (pairs[x].row == pairs[x+1].row) and (pairs[x].col ==
 pairs[x+1].col) ):
                pairs[x] = np.random.choice(self.myPopulation,1,True,probs
 )[0]
            x = x+2
        return pairs
```

Figure 21: Program Source Code 9 of 12 [Python]

```python
def breedPairs(self,pairs):
        x = 0
        list = []
        while(x<len(pairs)):
            rows = self.crossRows(self.myPopulation[x].row,self.
    myPopulation[x+1].row,np.random.randint(0,7))
            cols = self.crossCols(self.myPopulation[x].col,self.
    myPopulation[x+1].col,np.random.randint(0,7))
            newMember = member( [ rows[0],cols[0] ] )
            if(self.shouldMutate() == True):
                newMember = self.mutate(newMember)
            newMember2 = member( [rows[1],cols[1] ])
            if(self.shouldMutate() == True):
                newMember2 = self.mutate(newMember2)
            list.append(newMember)
            list.append(newMember2)
            x = x+2
        return list

    #choose a random position, thats where to cross
    def crossRows(self,row1,row2, index = 7):
        newRow1 = self.getString(row1,row2,index)
        newRow2 = self.getString(row2,row1,index)
        while(self.binaryToInt(newRow1) > 15 or self.binaryToInt(newRow2)
    > 15):
            index = int( np.random.randint(1,7) )
            newRow1 = self.getString(row1,row2,index)
            newRow2 = self.getString(row2,row1,index)
        return([newRow1,newRow2])
```

Figure 22: Program Source Code 10 of 12 [Python]

```python
def crossCols(self,col1,col2,index = 7):
        newCol1 = self.getString(col1,col2,index)
        newCol2 = self.getString(col2,col1,index)
        while(self.binaryToInt(newCol1) == 7 or self.binaryToInt(newCol2) == 7):
             index = np.random.randint(1,7)
             newCol1 = self.getString(col1,col2,index)
             newCol2 = self.getString(col2,col1,index)
        while(self.binaryToInt(newCol1) > 15 or self.binaryToInt(newCol1) > 15):
             again = False
             index = np.random.randint(1,7)
             newCol1 = self.getString(col1,col2,index)
             newCol2 = self.getString(col2,col1,index)
             again = self.binaryToInt(newCol1) == 7 or self.binaryToInt(newCol1) == 7
             while(again == True):
                 index = np.random.randint(1,7)
                 newCol1 = self.getString(col1,col2,index)
                 newCol2 = self.getString(col2,col1,index)
                 again = self.binaryToInt(newCol1) == 7 or self.binaryToInt(newCol1) == 7
        return([newCol1,newCol2])

    def getString(self,s1,s2,end = 5):
      newString = ""
      for x in range(0,end):
          newString = newString + s1[x]
      for x in range(end,len(s2)):
          newString = newString + s2[x]
      return newString

    def shouldMutate(self):
        return(random.randrange(0,100) < (self.mutationRate*100))

    def mutate(self,temp):
        if(random.randrange(0,100) < 50):
            temp.row = self.getRandomRow()
        else:
            temp.col = self.getRandomCol()
        return temp
```

Figure 23: Program Source Code 11 of 12 [Python]

```python
q = hospitalOptimization()
#print(q.costFunction(9,7))
q.plotCostSurface()
#q.getInitialPopulation()
#q.printPopulation()
#q.geneticAlgorithm()
```

Figure 24: Program Source Code 12 of 12 [Python]

8 Bibliography

[1] S. Russel and P. Norvig, *Artificial intelligence : A modern approach*, Third. Pearson, 2010.