Fakulteit Ingenieurswese, Bou-omgewing & IT

Faculty of Engineering, Built Environment & IT

EAI 320: Artificial Intelligence

Imtiaz Mukadam

U13083113

Assignment 3: Genetic Algorithm

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Introduction

Genetic Algorithms (or GA's) is a space search technique used in computer science to search large, possibly infinite, search spaces for various desirable solutions. Unlike conventional graph and tree searches, the GA starts with a set of proposed solutions whereby the algorithm is then run as to 'optimize' the solutions under the specific constraints of the problem. To investigate the efficacy and process of Genetic Algorithms, we are tasked with an optimisation problem in which we are to find the optimal location for a hospital in an area with given factors such as distance, number of emergencies and time.

We are tasked to first find the optimal solution by means of finding a minimum of a surface plot using the provided cost function. We then implement a Genetic algorithm to attempt to converge to the minimum location while investigating the effects of its parameters such as mutation rate, number of parents and number of generations to be produced.

We are provided with the location gird that includes 2 bridges which will need to be considered for the problem. We are also given the array w that holds the number of emergencies per location. For convenience, and for the purpose of this investigation, we will assume that the arrays x index corresponds to the grid's x location and the same applies for the y location. To clarify we assume that the array is not graphically mapped to the grid.

The cost function was given as:

$$C_{loc} = \sum_{i}^{16} \sum_{j}^{16} w_{i,j} \times distance$$

The fitness function is given by:

$$f(c) = 2.5 + 4.4 \times distance$$

Methodology

All methods are implemented in the Class GeneticAlgorithms. After instantiating an object of the class, two main functions, plotCostFunction() and geneticAlgorithm() can be called to start the respective processes.

Surface Plot

The surface plot graphs the cost function equation where $distance$ is a function of two sets of coordinates that returns the distance between 2 points that takes into account the 2 bridges in the grid. The function getDistance(x_i, y_i, Xloc, Yloc) returns the distance by first assessing whether a bridge needs to be crossed and then chooses the bridge that minimizes the distance. In this way, the cost function is optimised.

The plotCostFunction() plots the cost function implemented by a subclass called costFunction(). The function costFunction() sums the cost of all grid locations for a proposed hospital location and is given by:

```
def costFunction(self, x, y):
    res = 0
    for i in range(16):
        for j in range(16):
            res += self.w[i][j] * self.getDistance(i, j, x, y)
    if res < self.minCost:
        self.minCost = res
        self.minLocation = [x, y]
    return res
```

Figure 1 - costFuntion() of class GeneticAlgorithm

As can be seen above, costFunction() also notes the minimum cost and its corresponding location. This information is latter used in our GA as the goal location.

The distance function is implemented as follows:

```
def getDistance(self, x_i, y_i, x_loc, y_loc):
    if self.inSameHalfOfMap(x_i, x_loc):
        return self.euclideanDistance(x_i, y_i, x_loc, y_loc)
    else:
        return self.findBridgedEuclideanDistance(x_i, y_i, x_loc, y_loc)
```

Figure 2 getDistance() function of Class GeneticAlgorithm

The distance function first checks if the locations are in the same half of the map, in which case it would just return the Euclidean distance between the 2 points. If a bridge needs to be crossed, the 'bridged' Euclidean distance is returned.

Genetic Algorithm

The genetic algorithm is implemented under the parameters specified in the constructor of the class. The number of parents, number of generations and mutation rate can all be specified as constants in the constructor. These values will later be changed to assess the GA.

The chromosomes of the algorithm will encode a possible hospital location encoded as an 8-bit binary number where the first 4 bits represent the x-coordinates and the second 4-bits represents the y-coordinates. Thus each chromosome will have 2 genes containing a 4-bit binary string given as follows:

$$chromosome = [['xxxx'], ['yyyy']]$$

$$chromosome[0] = ['xxxx'], chromosome[1] = ['yyyy']$$

The algorithm implements a 50/50 crossover where 2 parents create 2 children with swapped x-coordinates and swapped y-coordinates. The population number is a function of the number of parents given as:
$populationSize = numberOfParents \times 2\ (numberOfparents - 1) + numberOfParents$. This implies that the algorithm run with 5 parents will have a population size of 45. The crossover will result in 40 children which, along with the parents, will constitute the population size of 45 for the following generation. Thus a smaller amount of parents will result in a smaller population size.

The main function geneticAlgorithm() contain various helper functions. The algorithm makes use of a priority queue which can conveniently sort the population according to fitness after the fitness has been determined. Therefore, we decided to implement the population in a priority queue that lists the chromosomes as a tuple given as $(fitness, chromosome)$. Thus, after the fitness has been determined, the fittest chromosomes can be conveniently popped from the queue. Some of the more important functions are:

- performGeneticAlgorithm(currentGen) – this function begins the process of the determining the fitness currentGen, obtaining the fittest parents and performing the crossover-mutation process. The nextGen is then returned as a priority queue.
- getGenerationFitness(currentGen) – the fitness of currentGen is calculated and the entire currentGen is returned in a priority queue with items as $(fitness, chromosome)$
- crossover(fitChromosomes) – The fitChromosomes are the most fit parents popped from the current generation. The function then performs the cross over with a chance of mutation which, if successful, randomises the bits in a specific coordinate of a child. The function then returns an entire new population which constitutes the next generation.

The process represented by the functions above repeats for a given number of generations. After the algorithm is complete, the fittest chromosome in the last generation is used as the fittest location found using the algorithm.

```
def geneticAlgorithm(self):
    currentGen = self.initialGeneration()
    for i in range(self.numberOfGenerations):
        currentGen = self.performGeneticAlgorithm(currentGen)
    currentGen = self.getGenerationFitness(currentGen)
    self.GA_result = currentGen.get()
    self.printResults()
```

Figure 3 main GA function

The function above calls:

```
def performGeneticAlgorithm(self, currentGen):
    fittestChromosomes = []
    currentGen = self.getGenerationFitness(currentGen)

    for i in range(self.numberOfParents):
        fittestChromosomes.append(currentGen.get())

    return self.crossOver(fittestChromosomes)
```

which returns a new generation given by the cross over function as:

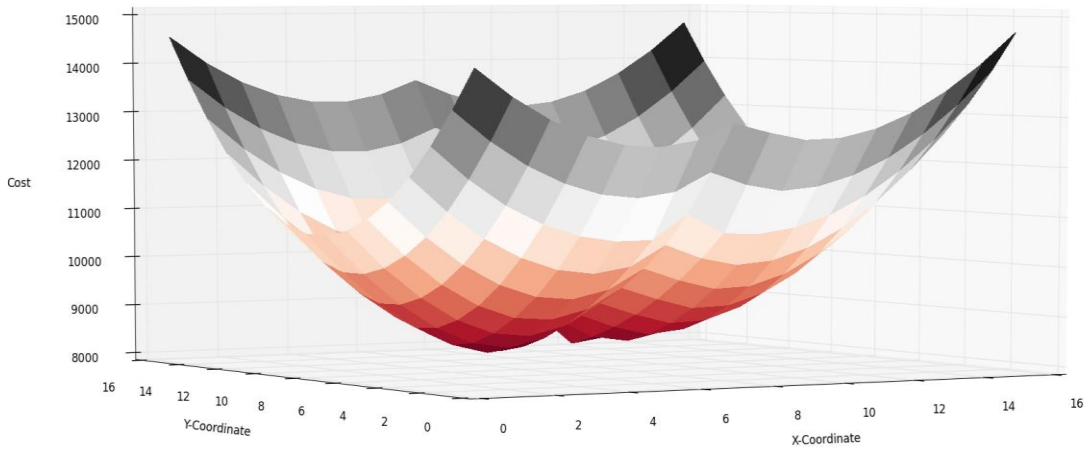
```
def crossOver(self, fitChromosomes):
    nextGen = Queue.PriorityQueue()
    for i in range(len(fitChromosomes)):
        nextGen.put((0, fitChromosomes[i][1]))
        for j in range(len(fitChromosomes)):
            if i != j:  # don't cross the same ones
                nextGen.put((0, self.getCrossOver(fitChromosomes[i], fitChromosomes[j])))
                nextGen.put((0, self.getCrossOver(fitChromosomes[j], fitChromosomes[i])))
    return nextGen
```

Figure 4 crossOver() function which returns the new generation

Various other helper functions are called in the algorithm and can be viewed in the attached code files.

Results

Surface plot



The surface plot displayed a bowl-like surface with various local minima. The global minimum was calculated to be 8198.1903327118816 with a location of [6, 9]. Note that the results differ when switching the indexes of the w array, at which the minimum is at location [9, 7]. This result will now be used in the GA as a proposed location to converge to.

Genetic Algorithm

The results of various runs of the algorithm with different parameters are presented by prints from the python console.

2% mutation rate:

Changing number of parents:

```
In[3]: x.geneticAlgorithm()
Goal location: [6, 9]
Number of generations: 5
Number of parents: 2
Mutation rate: 2.0 %
Optimal fitness: 2.5.

GA RESULTS
GA location found: [ 7 ,  8  ]
GA goal chromosome fitness:
8.72253967444
```

```
Goal location: [6, 9]
Number of generations: 5
Number of parents: 5
Mutation rate: 2.0 %
Optimal fitness: 2.5.

GA RESULTS
GA location found: [ 6 ,  8  ]
GA goal chromosome fitness:
6.9
```

```
Goal location: [6, 9]
Number of generations: 5
Number of parents: 10
Mutation rate: 2.0 %
Optimal fitness: 2.5.

GA RESULTS
GA location found: [ 6 ,  9  ]
GA goal chromosome fitness:
2.5
```

Changing number of generations:

```
Goal location: [6, 9]
Number of generations: 2
Number of parents: 5
Mutation rate: 2.0 %
Optimal fitness: 2.5.

GA RESULTS
GA location found: [ 6 ,  10  ]
GA goal chromosome fitness:
6.9
```

```
Goal location: [6, 9]
Number of generations: 10
Number of parents: 5
Mutation rate: 2.0 %
Optimal fitness: 2.5.

GA RESULTS
GA location found: [ 6 ,  9  ]
GA goal chromosome fitness:
2.5
```

0.1% mutation rate:

Changing number of generations:

```
Goal location: [6, 9]
Number of generations: 5
Number of parents: 5
Mutation rate: 0.1 %
Optimal fitness: 2.5.

GA RESULTS
GA location found: [ 6 ,  10  ]
GA goal chromosome fitness:
6.9
```

```
Goal location: [6, 9]
Number of generations: 2
Number of parents: 5
Mutation rate: 0.1 %
Optimal fitness: 2.5.

GA RESULTS
GA location found: [ 5 ,  9  ]
GA goal chromosome fitness:
6.9
```

```
Goal location: [6, 9]
Number of generations: 10
Number of parents: 5
Mutation rate: 0.1 %
Optimal fitness: 2.5.

GA RESULTS
GA location found: [ 5 ,  8  ]
GA goal chromosome fitness:
8.72253967444
```

Discussion

The surface plot presented us with the optimal location of the hospital in which we would investigate the efficacy of genetic algorithms by trying to converge to the proposed location. The search tree of the problem is relatively complex due to factors such as bridge locations and emergency density, thus the use of GA here is justified.

We see from our results that given a large enough population with a large enough number of generation iterations, the GA will easily converge to the proposed location. The efficacy of the GA is greatly influenced by both the size of the population and the number of parents. Naturally, the higher the amount of iterations, the higher the chance of finding a better chromosome. However, by inspection of the chromosomes after a large number of iterations, we find that the chromosome pool becomes stagnant, and for this particular problem, it quickly becomes a pool of 100% duplicated chromosomes. This could result in a total loss of the fittest possible chromosome. With this in mind, mutation becomes a powerful component of the GA to introduce variation in a stagnant pool. Should the pool become stagnant, a higher mutation rate will break repetition and allow the algorithm to, in a sense, have a longer search lifetime.

A balance between these 3 factors becomes crucial in implementing an effective fast converging genetic algorithm. With these results, a typical effective GA has a mutation rate of 2% and iterates for around 5-10 generations. This with a population size of 45 almost always proved to converge to the fittest chromosome.

Conclusion

A genetic algorithm does not seem to have a typical format. For different problems, not only will one need a specific cross over method, a different mutation method relating to the cross over must be implemented. This, along with the general GA parameters of mutation rate, population size, number of parents etc. allows GA's to be versatile tools that fit various with large search spaces. Naturally, however, due to its random nature, the GA is not always guaranteed to produce optimal results. With careful implementation and parametrizing, it can be one of the most practical and powerful search techniques for many search problems.