

Assignment 1
Priyanka Dhulkhed
Liam O'Brien

Task 1:

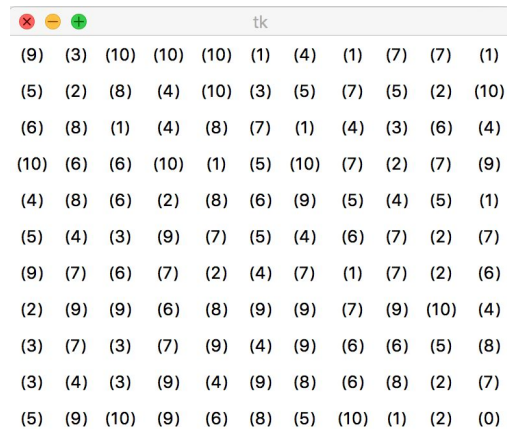
For the first task we have it implemented so the user can upload a text file containing the numbers they want inputted, otherwise they can have a randomly picked $n \times n$ array of size 5, 7, 9, or 11. The program automatically fills the GUI with the numbers that were manually chosen or else it randomly fills it with random numbers from 1 to $(n-1)$. We then generate a 0 cell matrix for task two.

The structure we used to construct the matrix of cells in the puzzle is depicted below.

```
class cell(object):
    def __init__(self, x, y, value, visit):
        self.x = x
        self.y = y
        self.value = value
        self.visit = visit
```

The object cell represents a single cell in the puzzle grid. This object has attributes x (the row-coordinate), y (the column coordinate), value (the actual value of the cell as depicted in the GUI), and visit (the number of steps to visit that cell from the root cell).

The GUI that was created by the program for a 11 x 11 matrix is depicted below:



(9)	(3)	(10)	(10)	(10)	(1)	(4)	(1)	(7)	(7)	(1)
(5)	(2)	(8)	(4)	(10)	(3)	(5)	(7)	(5)	(2)	(10)
(6)	(8)	(1)	(4)	(8)	(7)	(1)	(4)	(3)	(6)	(4)
(10)	(6)	(6)	(10)	(1)	(5)	(10)	(7)	(2)	(7)	(9)
(4)	(8)	(6)	(2)	(8)	(6)	(9)	(5)	(4)	(5)	(1)
(5)	(4)	(3)	(9)	(7)	(5)	(4)	(6)	(7)	(2)	(7)
(9)	(7)	(6)	(7)	(2)	(4)	(7)	(1)	(7)	(2)	(6)
(2)	(9)	(9)	(6)	(8)	(9)	(9)	(7)	(9)	(10)	(4)
(3)	(7)	(3)	(7)	(9)	(4)	(9)	(6)	(6)	(5)	(8)
(3)	(4)	(3)	(9)	(4)	(9)	(8)	(6)	(8)	(2)	(7)
(5)	(9)	(10)	(9)	(6)	(8)	(5)	(10)	(1)	(2)	(0)

Task 2:

The second task looks for all the cells that can be reached from the starting cell until it reaches the goal cell which is hard-coded to 0. It then prints to the terminal the cell matrix that was created and altered to show the possible moves from the starting cell. Any cell that is not able to be reached from the starting cell is replaced with an X. If the puzzle is solvable, we print to the terminal the minimum number of steps that is required to reach the goal cell. If the puzzle is unsolvable, we return the negative value of cells that are never able to be reached from the starting cell.

In order to implement Task 2, the cell structure (depicted above) as well as the following structures were used

```
class node(object):
    def __init__(self, state, parent, action, cost, depth):
        self.state = state
        self.parent = parent
        self.action = action
        self.cost = cost
        self.depth = depth

class act_res(object):
    """docstring for act-res"""
    def __init__(self, action, result):
        self.action = action
        self.result = result

class cellMatrixVal(object):
    def __init__(self, matrix, value):
        self.matrix = matrix
        self.value = value
```

The object node is used to construct and traverse the tree for a breadth first search implementation.

The attributes are :

- state : has the value cell. It represents the state/cell in the puzzle the node is of
- parent : the parent of the node
- action : a set of actions that can be possibly reached from a certain state in the puzzle
- cost : describes the cost of the traversal
- depth : the depth of the node in the tree

The act-res object contains :

- an action (left/right/up/down) that can be performed for a particular cell
- result : the cell that has been reached by performing the action

The cellMatrixVal object contains :

- matrix : any matrix of cells
- value : the evaluation value of that matrix

The function (along with helper methods) performs a breadth first search in order to find an evaluation value of a certain puzzle.

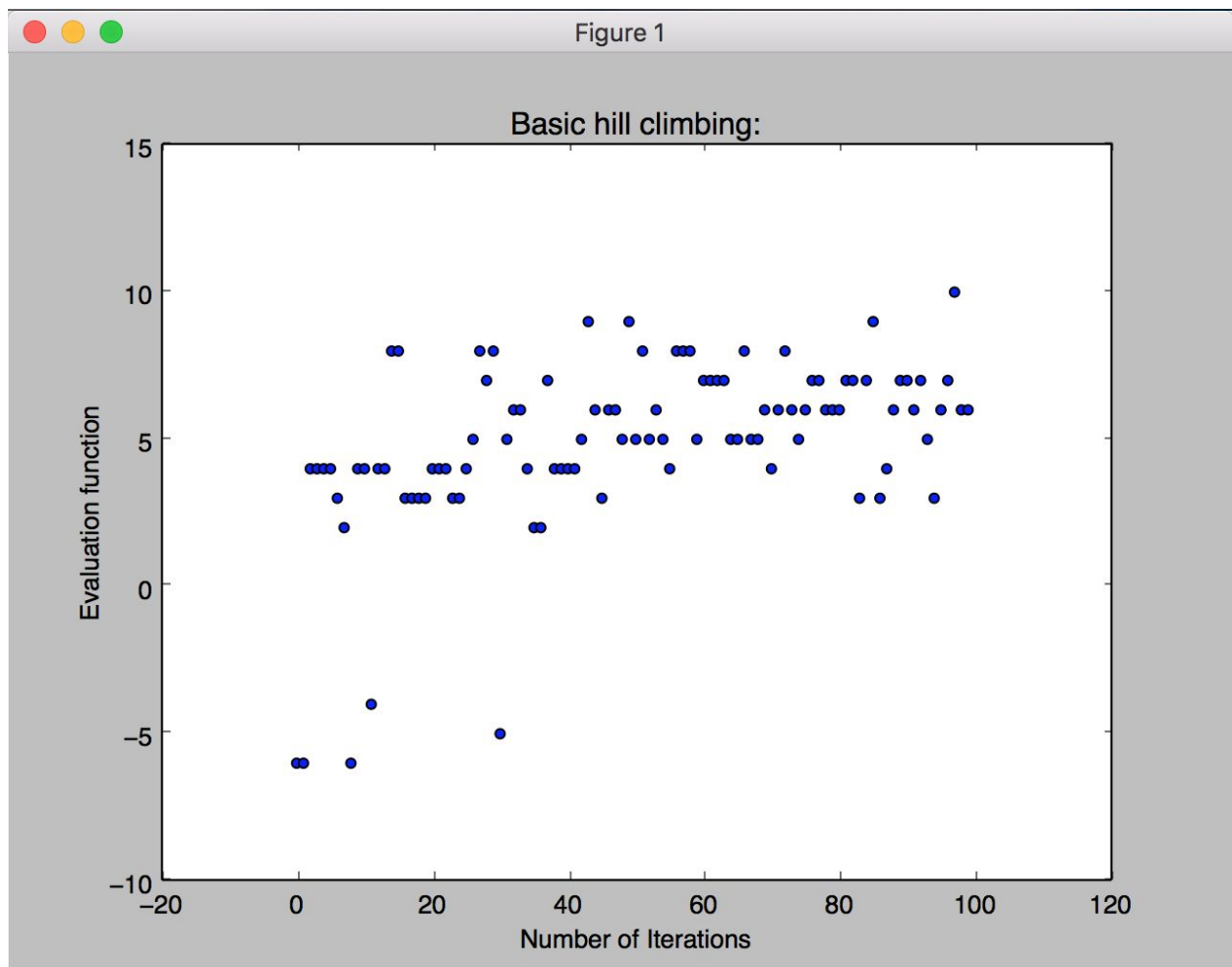
Task 3:

Task 3 implements a basic hill climbing approach. The function that implements this is:

```
def basicHillClimbing(cellMatrix, randomGen, iter):  
    ....  
    return hillClimb
```

The function above takes in the original puzzle, the dimensions of the puzzle, and the number of iterations to run the process over. It returns hillClimb, which is a cellMatVal object that contains the value as well as the result of the function.

After the cell matrix is displayed, we prompt the user to enter the number of times they want the hill climb method to iterate. The program then tries the hill climbing method. For the purpose of demonstration, the following graph displays 100 iterations:



From the graph you can see that the evaluation function increases in a linear trend to the number of iterations. These are the times to compute 100 iterations of the respective matrices:

5 x 5: 1.587 Seconds

7 x 7: 3.246 Seconds

9 x 9: 7.718 Seconds

11 x 11: 18.644 Seconds

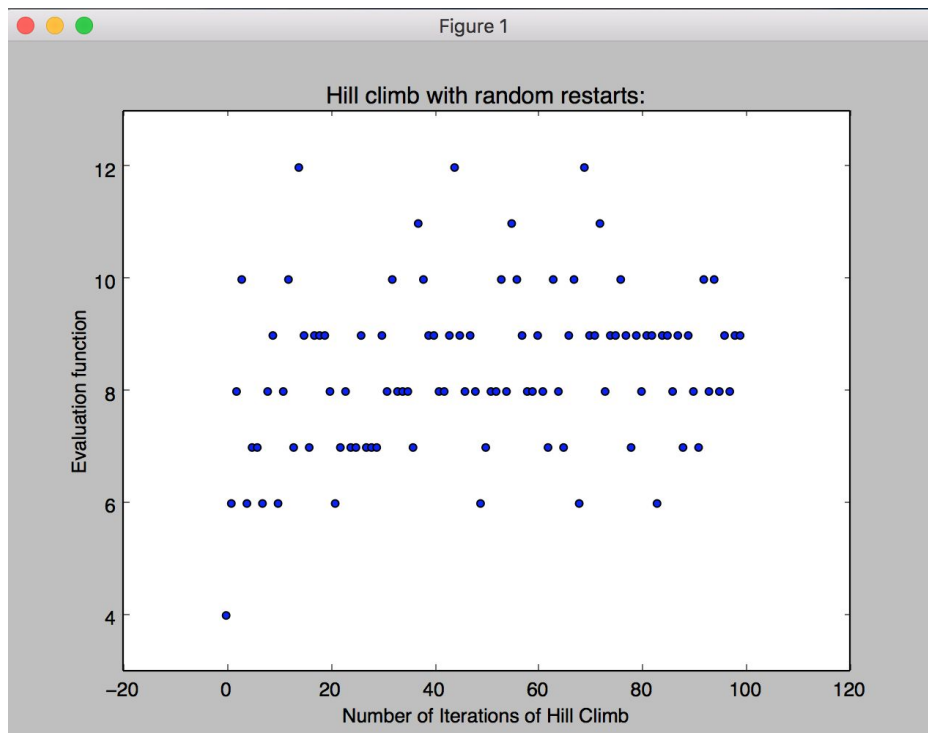
Task 4:

In this task we implement the basic hill climbing approach with the possibility of downhill movements to escape local extrema. The function we implemented is depicted below:

```
def hillClimbRandRestart(cellMatrix, randomGen, iterHillClimb, iterPerHillClimb):  
    ....  
    return max(allVals)
```

This function takes in the number of iterations to perform the hill climbing process and the number of iterations per hill climbing process and returns the value of the puzzle configuration that results in the maximum evaluation.

After we display the first graph we prompt the user for the number of hill climb processes they would like to perform, then we ask them how many iterations per hill climb they would like to perform. For the purpose of demonstration, the following graph displays 100 processes at 100 iterations:



These are the times to compute 100 iterations of the respective matrices:

5 x 5: 1.454 Seconds

7 x 7: 3.179 Seconds

9 x 9: 7.566 Seconds

11 x 11: 17.981 Seconds

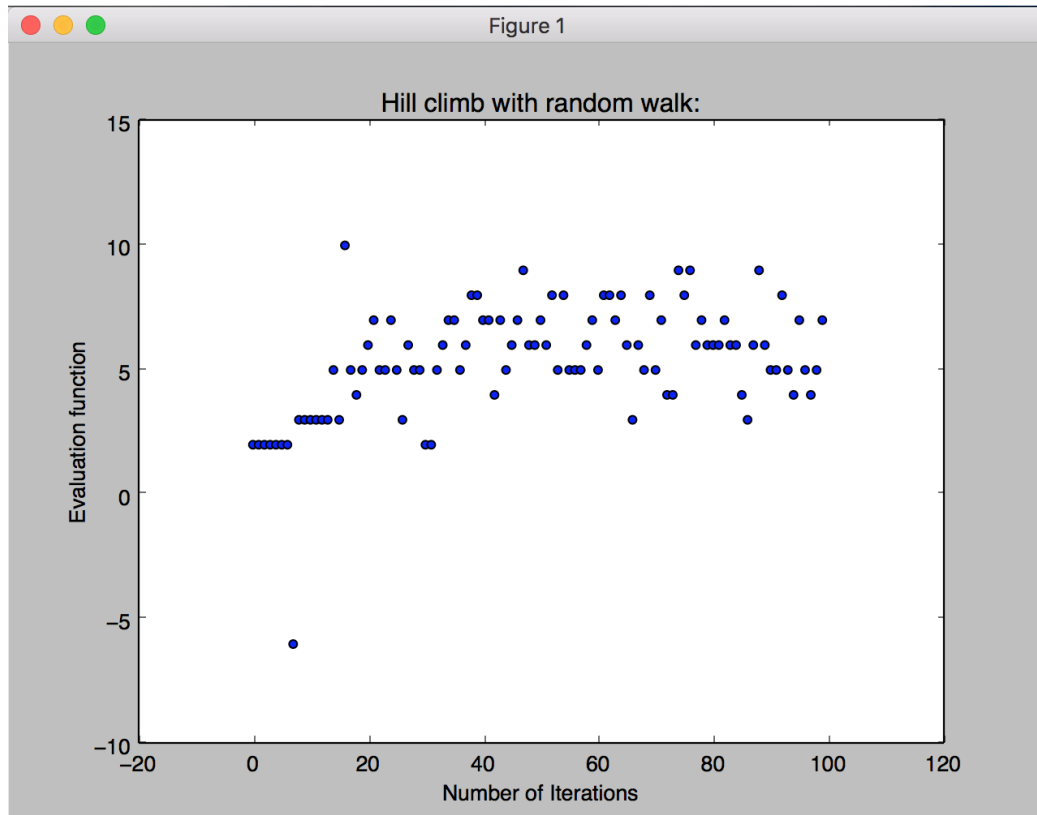
Task 5:

In task 5, we perform the basic hill climbing approach with a possibility for a downward hill movement. The function is depicted below:

```
def basicHillClimbRandWalk(cellMatrix, randomGen, iter, prob):  
    ....  
    return hillClimb
```

This function takes in the number of iterations and the probability for a downward movement and returns the object hillClimb, which contains the optimal puzzle configuration and its value.

After the graph is displayed for task 4 we inform the user that task 5 will perform a random walk with a probability. We then ask the user to enter the number of iterations of hill climbing they would like to perform, then we ask them to enter a number for the probability. For the purpose of demonstration, the following graph displays 100 iterations at a probability of 0.5:



These are the times to compute 100 iterations the respective matrices with a probability of .5%

5 x 5: 1.627 Seconds

7 x 7: 4.575 Seconds

9 x 9: 8.584 Seconds

11 x 11: 14.288 Seconds

Task 6:

In task 6, we perform a simulated annealing. The function is depicted below:

```
def simulatedAnnealing(cellMatrix, randomGen, iter, iTemp, tempDecay):
    currentMat = cellMatrix
    currentVal = evaluateTreeSearch(cellMatrix, randomGen).value
    currentMatVal = cellMatrixVal(currentMat, currentVal)
    T = iTemp
    while(True):
        T = T*tempDecay
        if round(T, 5) == 0:
            return currentMatVal
        randX = random.randint(0, randomGen-1)
        randY = random.randint(0, randomGen-1)
        nextMat = currentMat
        for c in range(randomGen):
```

```

        for d in range(randomGen):
            nexxtMat[c][d].visit == 0
            randCell = random.randint(1, randomGen - 1)
            nexxtMat[randX][randY].value = randCell
            nexxtCellVal = evaluateTreeSearch(nexxtMat, randomGen).value

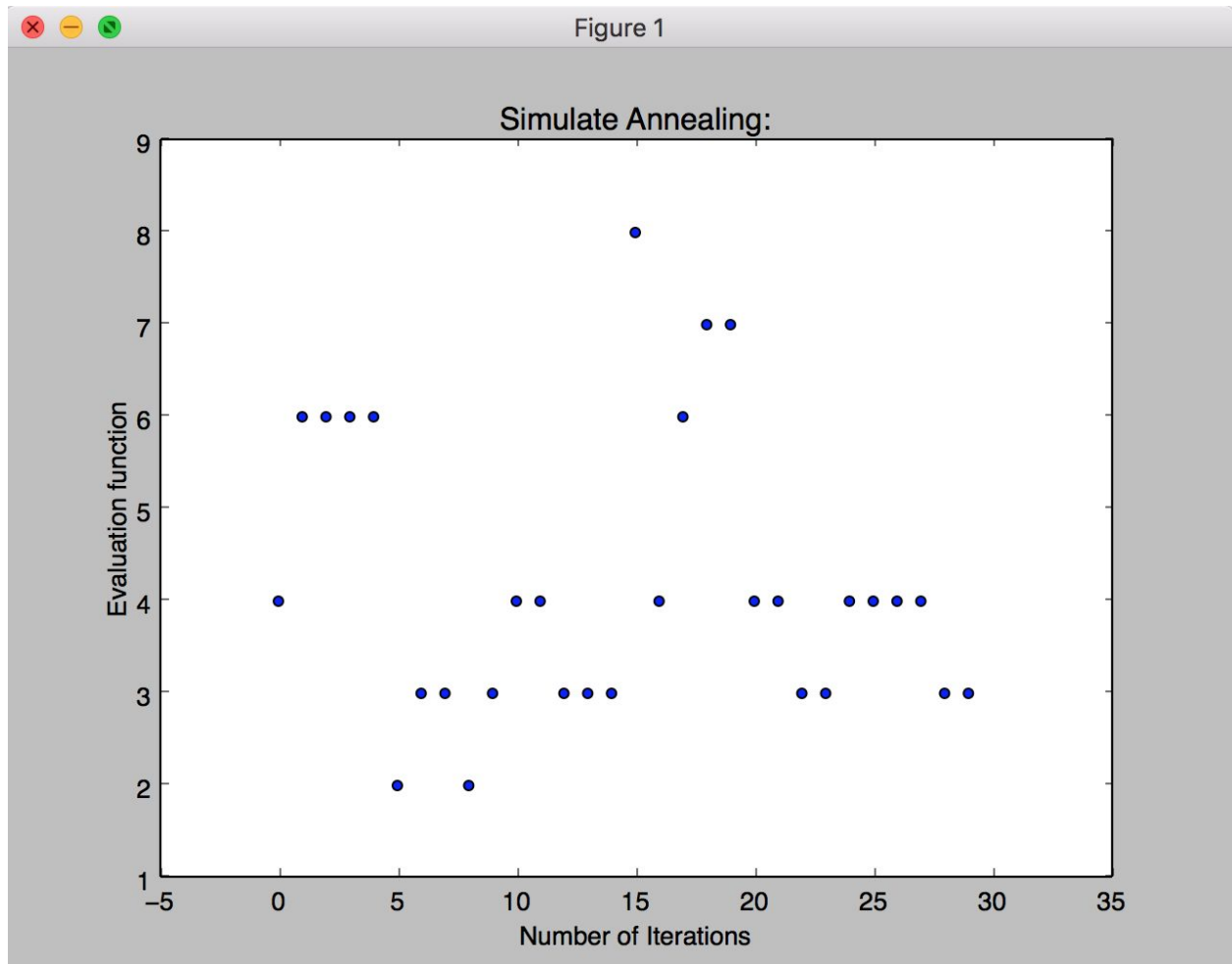
            nexxtMatVal = cellMatrixVal(nexxtMat, nexxtCellVal)

            deltaE = nexxtCellVal - currentVal
            if deltaE > 0:
                currentMat = nexxtMat
                currentVal = nexxtCellVal
                currentMatVal = nexxtMatVal
            else:
                probRand = random.random()
                prob = math.exp(deltaE/T)
                if probRand > prob:
                    currentMat = nexxtMat
                    currentVal = nexxtCellVal
                    currentMatVal = nexxtMatVal

```

Taking in an initial temperature (iTemp), the number of iterations (iter), and the temperature decay rate (tempDecay), the function returns a currentMatVal object that contains the optimal puzzle configuration and the value of that puzzle.

After the graph from Task 5 is displayed, we prompt to user to simulate annealing. We first ask the user to input the number of iterations they want to perform, followed by the initial temperature, and ending with the rate of decay of temperature. For the purpose of demonstration, the following graph displays 30 iterations with an initial temperature of 30 and a temperature decay of 0.4:



The graph displays that as the number of iterations increases, the evaluation function has a general linear trend.

These are the times to compute 100 iterations the respective matrices an initial temperature of 20 and a temperature decay of 0.4:

5 x 5: 0.0297 seconds

7 x 7: 0.0470 seconds

9 x 9: 0.0475 seconds

11 x 11: 0.0633 seconds

100 x 100: 0.6118 seconds

Task 7:

In task 7, we implement a genetic algorithm. The function is depicted below:

```
def geneticAlgo(cellMatrix, randomGen, goodEnoughFitness, numOfParents, numOfIterations):  
    population = generatePopulation(numOfParents, randomGen, cellMatrix)
```

```

    iterat = 0

    while (True):
        iterat = iterat + 1
        if (iterat > numOfIterations):
            theBestFitness = bestFitness(population, randomGen)
            # print "the Best fitness is", theBestFitness
            for ans in population:
                if fitness(ans, randomGen) == theBestFitness:
                    ansMat = OneDto2D(ans, randomGen)
                    ansVal = theBestFitness
                    ansMatVal = cellMatrixVal(ansMat, ansVal)
                    # print "RETURN HEREE"
                    return ansMatVal

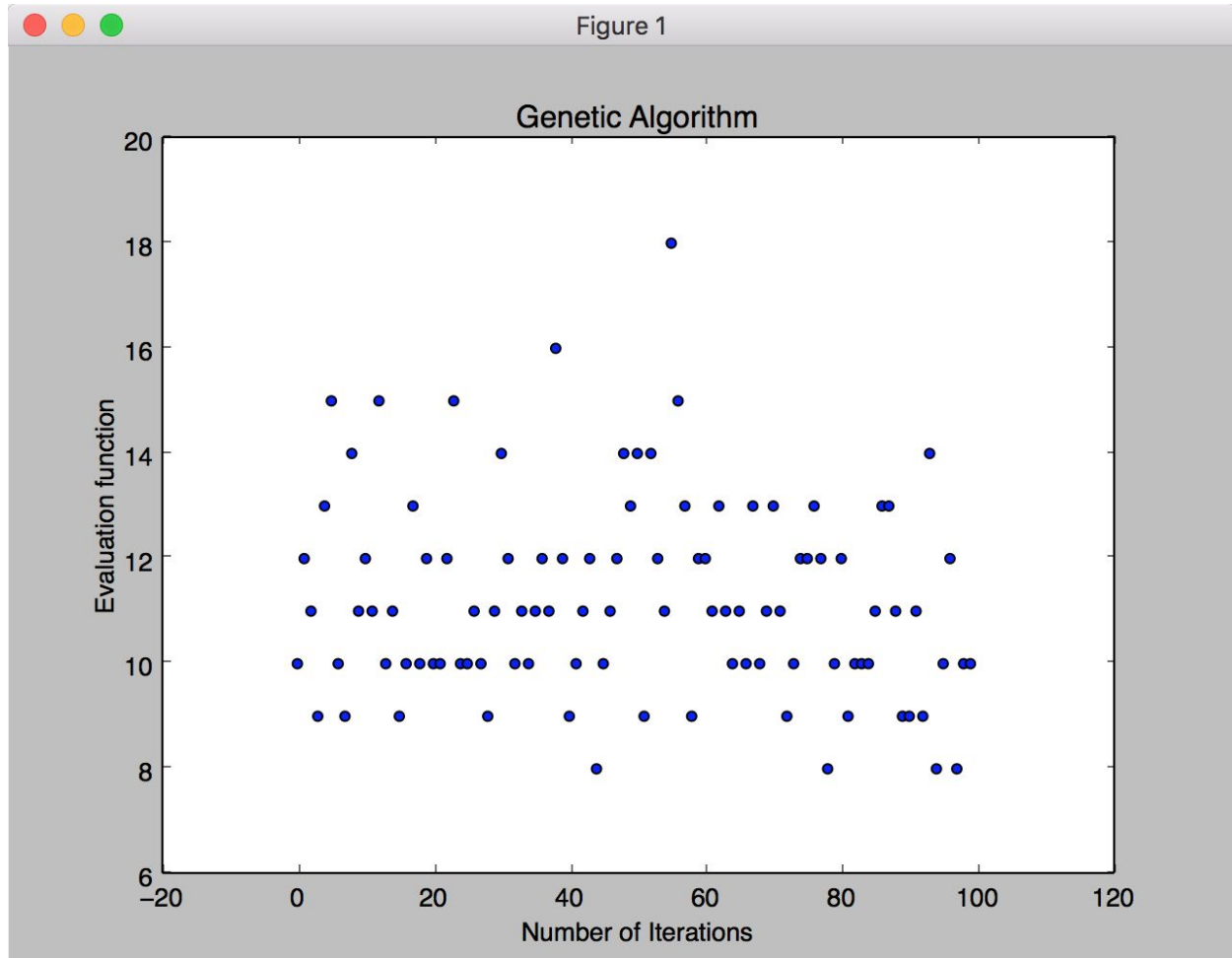
            newPopulation = []
            for i in range(len(population)):
                # print "RUN HERE???"
                xIndex = random.randint(0, len(population)-1)
                yIndex = random.randint(0, len(population)-1)
                child = reproduce(population[xIndex], population[yIndex], randomGen)
                randomProb = random.uniform(0, 1)
                probbRand = random.random()
                if probbRand > randomProb:
                    child = mutate(child, randomGen)
                newPopulation.append(child)
            population = newPopulation
            if bestFitness(population, randomGen) >= goodEnoughFitness:
                for ans in population:
                    if fitness(ans, randomGen) == bestFitness(population, randomGen):
                        ansMat = OneDto2D(ans, randomGen)
                        ansVal = bestFitness(population, randomGen)
                        ansMatVal = cellMatrixVal(ansMat, ansVal)
                        return ansMatVal

            theBestFitness = bestFitness(population, randomGen)
            for ans in population:
                if fitness(ans, randomGen) == theBestFitness:
                    ansMat = OneDto2D(ans, randomGen)
                    ansVal = theBestFitness
                    ansMatVal = cellMatrixVal(ansMat, ansVal)
                    return ansMatVal

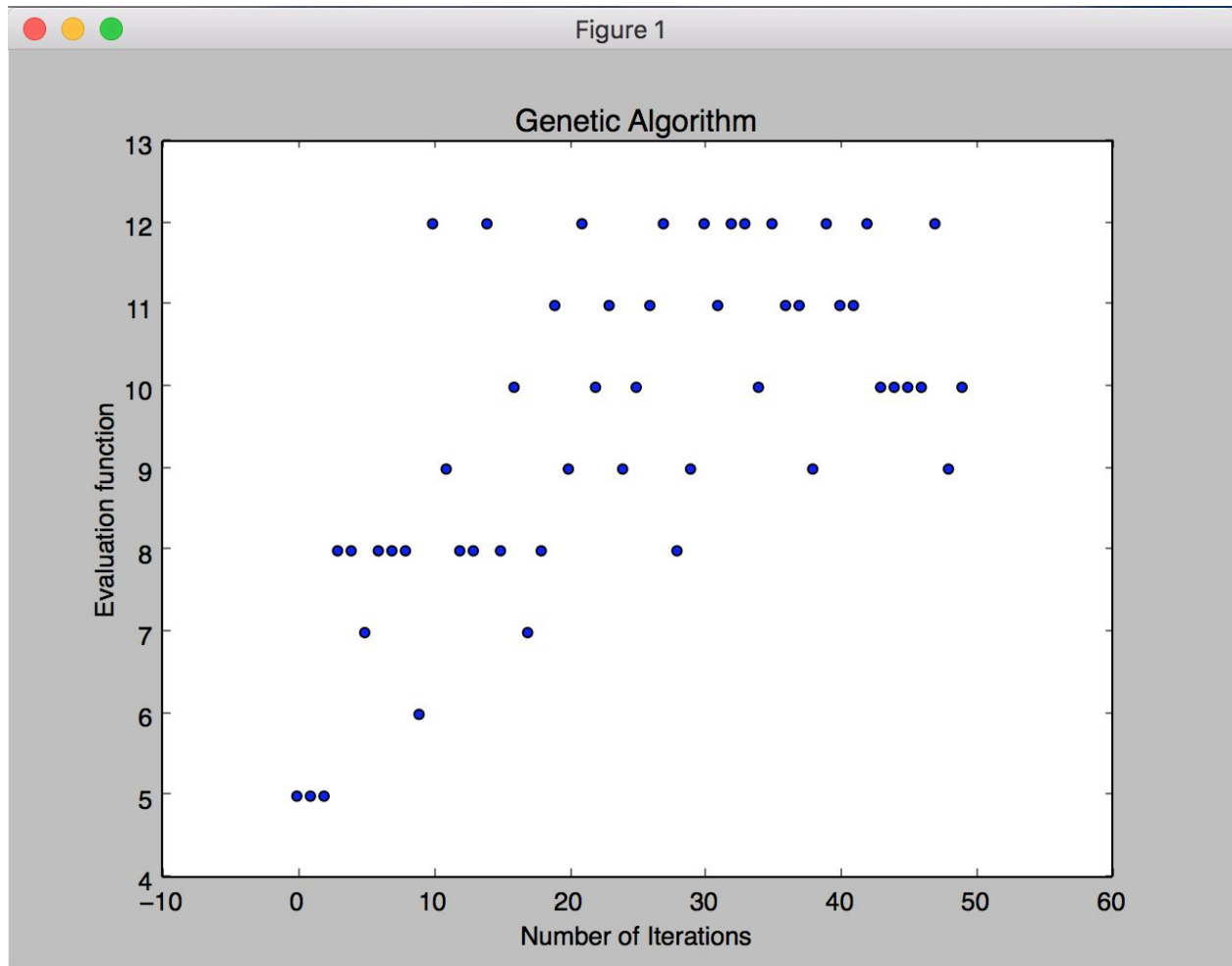
```

This function takes in a number of parents to create a population, a “good enough” fitness, and the number of iterations to let the algorithm run. A population is created with a generatePopulation method, in which the original matrix is included. After the population is created, an empty set newPopulation is created. Two random parents are selected in the set of parents and the function reproduce allows the 2 parents to reproduce and selects one child which has a higher value. With some random probability, this child is mutated with the mutate function. After several children are created, they are put into the set population. Then, we compare the bestFitness of the children in the population to see if it reaches the goodEnoughFitness, and if so, the new matrix configuration and the value are returned in an ansMatVal object. There is a check in the beginning so that if the number of iterations has exceeded the number specified by the user, the best in the existing set is returned.

After task 6, the program informs the user that we will be performing a genetic algorithm. We ask the user to enter the number of parents they would like the algorithm to perform on. Next we ask them to enter a fitness number that is suitable for them. Lastly we ask them the number of iterations they would like to perform. For the purpose of demonstration, the following graph displays 50 parents with a fitness of 5 and 100 iterations:



From this graph it is not clear that there is not any linear trend or any positive progress on the evaluation function over the number of iterations. The following graph goes through the evaluation function over the number of parents:



(Note: Number of iterations on this graph represents the number of parents)

From this last graph it is very clear that as the number of parents increase, the evaluation function performs linearly in respect to the number of parents.