**Report**

**Introduction**

The game of Jump-It consists of a single player and a board of integers, were each integer represents the cost to move to that integer’s position. The game boards themselves consist of a set of n integers where the first element in the board is always 0 and the rest of the integers are always positive. The main objective of the game is to move the player character from the first cell to the last cell with the lowest total cost. This experiment implemented a genetic algorithm to generate Jump-It problem solutions with lowest total cost, and compare the result with the Dynamic Programming solution to test the overall performance of our algorithm. The player character will always start the game in the first cell, with a total cost of 0, and will have two types of moves available: The player can either move to the adjacent cell or jump over the adjacent cell to land on the next cell. The total cost of a game of Jump-It is the sum of the costs of the visited cells. The goal of this game is to get the lowest total cost.

**Background**

Genetic algorithm (GA) is adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetics. It is commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection. It reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. In a genetic algorithm, the set of genes of an individual can be represented using the binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.

**Implementation**

We used binary representation for our candidate solutions, and the genotype consists of bit strings, and the phenotype means the solution of the Jump- it problems. This experiment begins with creating a set of individuals which is called a population. Each individual is a solution to the Jump-it problem. We encoded a board with 1s and 0s, where a 1 corresponds to a visited cell and a 0 corresponds to an unvisited cell, and the length of board equal to the bits of binary number. We random generate a binary list which includes the random binary combination. According to the rule of this game, the player have to end with last position, which means the last position always have to be visited, so we set up the last cell of the board always equal to 1. Also, the player have to start to jump from first position and the cost of first position is always equal to 0, so we strip out the first cell on the board when we calculate the total cost. In addition, every candidate solution cannot have two consecutive 0s, so we build our own “Pairwise Testing” function to check if there are any adjacent 0s appear in the candidate solution, if yes, we dropped this candidate to make sure all our selected population are valid to this Jump-it problem.

The fitness function determines how fit an individual is (the total cost of an individual is lower than any other individuals). It gives a fitness score to each individual, the lower a total cost of an individual has the better fitness score it will get. The probability that an individual will be selected for reproduction is based on its fitness score. In the parent selection step, we used Roulette Wheel method for selecting parents. This method is used to associate a probability of selection with each individual chromosome. If  is the fitness of individual *i* in the population, its probability of being selected is where N is the number of individuals in the population. In order to select the best parents, Roulette Wheel method is able to find the candidate solution with the highest probability. Moreover, in order to enhance the diversity in select the parents. We randomly generated threshold values, instead of always choosing highest probability, we compared the fitness probabilities with the threshold values, as long as the fitness probabilities greater than or equal to the threshold value, we return the parents

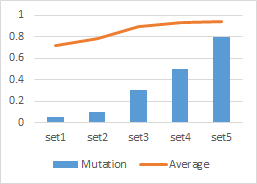
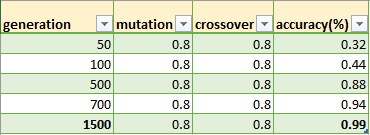
In the parent crossover step, we used crossover on both selected parents. We randomly chose a crossover point, and split both parents to two parts, and swapped one of the parts between the two parents. The resulting combination are the children. However, a generated child would have two consecutive 0s sometimes, which is not allowed in the Jump-It game. In order to prevent this from happening in crossover, we did try different crossover points, and clone one of the two parents when children not valid. In addition, in order to preserving and introducing diversity. Mutation allow the algorithm to avoid local minima by preventing the population of chromosomes from becoming too similar to each other, we used flip bit mutation operator takes the chosen genome and inverts the bits (i.e. if the genome bit is 1, it is changed to 0 and vice versa). Moreover, we compare GA method to Dynamic Programming to check the overall accuracy.

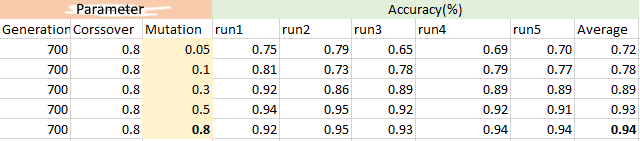
**Experimental Setup and Results**

In the experiment, we used two samples as the input files. One is small which only includes 5 jump-it boards and the other is much larger including 100 jump-it boards to give us a better test sample. We discussed our result based on the larger file in this section as shown in Table 1 and Table 2. We started the max generation from 50 to 700, the overall accuracy ranged from 31% to 94% with setting up crossover rate equal to 0.8 and mutation rate equal 0.05. We noticed that the accuracy have been improved with increasing the generation size. We also enhance the generation size up to 1500, and the result reach up to 99% accuracy, which is really good performance, but the computational cost is more expensive.

In our algorithm implementation, we generated the population size randomly. However, it is really easy to lead to local optimal, in order to avoid this happen, we increased the mutation rate and analyzed the relationship between mutation rate and accuracy. We chose 5 different mutation rate with same generation size and crossover rate, for each setting up, we run the experiment 5 times, and calculated the average accuracy of all runs. We observed that when mutation rate equal to 0.05, the average accuracy only around 72%, but it go up to 94% when mutation rate equal to 0.8.(as shown in Table 3). We can conclude that mutation rate does help improve the performance and it can avoid the local optimal problem.

Table 1 Table 2

****

**** Table 3

**Extra Implementation**

We modified our selection method to tournament selection method, it will select the winner of each tournament (the one with the best fitness). The fitness of individuals to use as selection criterion, and it will return a list of selected individuals. If the tournament size is larger, weak individuals have a smaller chance to be selected because if a weak individual is selected to be in a tournament, there is a higher probability that a stronger individual is also in that tournament. We have also implemented random selection method, which returns randomly two genomes from the gene pool. For this particular Jump it problem, tournament selection has higher accuracy than random selection when the generation size is small, Table 4 show the comparison of tournament selection and random selection in small generation size. Table 5 show the comparison of tournament selection and random selection in larger generation size. However, when the generation size increased, tournament selection did not make any improvement, it performed worse than random selection.

Table 4

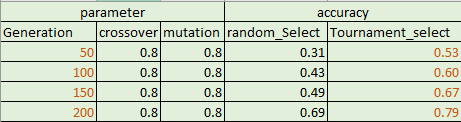
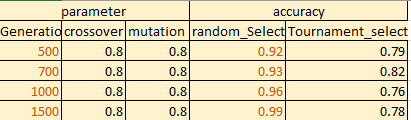


Table 5

****

**Conclusion**

In this experiment, we implemented a genetic algorithm on Jump-it board game, we used binary presentation to analyze this board game solution. Our genetic algorithm including selection, mutation, and crossover function, which are all helpful to improve the performance. We compared our algorithm with Dynamic Programming solution to test the overall accuracy, the best result reach up to 99% accuracy when we using random selection and the generation size is large.

**Reference**

1. Eiben, A. E. et al (1994). "Genetic algorithms with multi-parent recombination". PPSN III: Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: 78–87. ISBN 3-540-58484-6.  
  
2. Miller, B.L. and Goldberg, D.E., 1995. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, *9*(3), pp.193-212.  
3.  "XI. Crossover and Mutation*". http://www.obitko.com/:* Marek Obitko*. Retrieved 2011-04-07*.

**Code**

"""==============================================================

COURSE: CSC 635, Homework 4

PROGRAMMER: Junya Zhao and Rohan Saha

Trace: \\trace\Class\CSC-535-635\001\Rohan2728\HW4

DATE: 5/1/2018

DESCRIPTION: To implement a Genetic Algorithm, for jump it game

FILES: hw4.py

DATASET: input1.txt and input2.txt

=============================================================="""

# ---------------------------------Imports--------------------------------------

import itertools as it

import random

# ------------------------------------------------------------------------------

# ---------------------------------Variables------------------------------------

global cost, path

cost = [] # global table to cache results - cost[i] stores minimum cost of playing the game starting at cell i

path = [] # global table to store path leading to cheapest cost

# ------------------------------------------------------------------------------

# ---------------------------------Class----------------------------------------

class GeneticAlgorithm:

probCrossover = 0.0

probMutation = 0.0

maxGeneration = 0

board = []

population = []

fitness = []

def \_\_init\_\_(self, gameBoard, nMaxGeneration, pCrossover, pMutation):

self.population = []

self.fitness = []

self.board = gameBoard

self.probCrossover = pCrossover

self.probMutation = pMutation

self.maxGeneration = nMaxGeneration

'''Initializes Genetic Algorithm'''

def geneticAlgorithm(self):

self.createPopulation()

list(map(lambda c: self.fitness.append(self.generateFitness(c)), self.population))

for i in range(self.maxGeneration):

'''Selection Roulette Wheel'''

parent1 = self.selectionRouletteWheel()

parent2 = self.selectionRouletteWheel()

child1, child2 = self.crossover(parent1, parent2) # Do crossover

mChild1, mChild2 = self.mutation(child1, child2) # Do mutation

child1Fitness = self.generateFitness(mChild1)

child2Fitness = self.generateFitness(mChild2)

# Replaces the least fittest individuals from the gene pool

if child1Fitness > self.generateFitness(parent1) or child1Fitness > self.generateFitness(parent2):

sP1 = min(self.fitness)

sP1\_index = self.fitness.index(sP1)

if self.population.count(mChild1) == 0:

self.population[sP1\_index] = mChild1

self.fitness[sP1\_index] = child1Fitness

if child2Fitness > self.generateFitness(parent1) or child2Fitness > self.generateFitness(parent2):

sp2 = min(self.fitness)

sP2\_index = self.fitness.index(sp2)

if self.population.count(mChild2) == 0:

self.population[sP2\_index] = mChild2

self.fitness[sP2\_index] = child2Fitness

'''Roulette Wheel selection function'''

def selectionRouletteWheel(self):

totalSum = sum(self.fitness)

rWheel = list(map(lambda f: f/totalSum, self.fitness))

s = 0

pick = random.random()

for i, r in enumerate(rWheel):

s += r

if pick <= s:

parent = self.population[i]

break

return parent

'''Single point crossover function'''

def crossover(self, parent1, parent2):

child1, child2 = [], []

counter = 0

if random.random() <= self.probCrossover:

while counter < len(parent1):

c1, c2 = self.doCrossover(parent1, parent2)

counter += 1

if self.pairwiseTesting(c1) and self.pairwiseTesting(c2):

child1 = c1

child2 = c2

break

if counter >= len(parent1):

child1 = parent1

child2 = parent2

else:

child1 = parent1

child2 = parent2

return child1, child2

def doCrossover(self, parent1, parent2):

crossOverPoint = int(len(parent1)/2)

c1 = parent1[:crossOverPoint] + parent2[crossOverPoint:]

c2 = parent2[:crossOverPoint] + parent1[crossOverPoint:]

return c1, c2

'''Random index mutation function'''

def mutation(self, child1, child2):

mChild1, mChild2 = [], []

counter = 0

if random.random() <= self.probMutation:

while counter < len(child1):

mC1, mC2 = self.doMutation(child1, child2)

counter += 1

if self.pairwiseTesting(mC1) and self.pairwiseTesting(mC2):

mChild1 = mC1

mChild2 = mC2

break

if counter >= len(child1):

mChild1 = child1

mChild2 = child2

else:

mChild1 = child1

mChild2 = child2

return mChild1, mChild2

def doMutation(self, child1, child2):

mChild1, mChild2 = child1[:], child2[:]

index = random.randint(0, len(child1) - 2)

if child1[index] == 0:

mChild1[index] = 1

else:

mChild1[index] = 0

if child2[index] == 0:

mChild2[index] = 1

else:

mChild2[index] = 0

return mChild1, mChild2

'''Evalution function to get the fitness value of each genome'''

def generateFitness(self, chromosome):

cost = 0

for index, value in enumerate(chromosome):

if value == 1:

cost += self.board[index]

return 1/cost

'''Generates candidate solutions or the gene pool'''

def createPopulation(self):

n = len(self.board)

for i in range(n\*3):

pop = [random.randint(0, 1)]

for j in range(n-2):

if pop[-1] == 0:

pop.append(1)

else:

pop.append(random.randint(0, 1))

pop.append(1)

if self.population.count(pop) == 0:

self.population.append(pop)

'''Function to validate that genome doesnot have simultaneous 0 values'''

def pairwiseTesting(self, candSol):

isSame = True

x, y = it.tee(candSol)

next(y, None)

for i, j in zip(x, y):

if i == 0 and j == 0:

isSame = False

break

return isSame

'''Gets the max fittest value from the gene pool, afer max generation is reached'''

def getFittestSolution(self):

max\_fittest = max(self.fitness)

fittest = self.population[self.fitness.index(max\_fittest)]

max\_fittest = round(1 / max\_fittest)

print("Minimum Cost (fitness): ", max\_fittest)

path\_contents = "0"

print("path showing indices of visited cells:", end=" ")

print(0, end="")

for i in range(len(fittest)):

if fittest[i] == 1:

print(" ->", i + 1, end="")

path\_contents += " -> " + str(self.board[i])

print("\npath showing contents of visited cells: ", path\_contents)

return max\_fittest

# ------------------------------------------------------------------------------

# ---------------------------------Functions------------------------------------

"""

Dynamic Programming solution to the jump-It problem

The solution finds the cheapest cost to play the game along with the path leading

to the cheapest cost

"""

def jumpIt(board):

# Bottom up dynamic programming implementation

# board - list with cost associated with visiting each cell

# return minimum total cost of playing game starting at cell 0

n = len(board)

cost[n - 1] = board[n - 1] # cost if starting at last cell

path[n - 1] = -1 # special marker indicating end of path "destination/last cell reached"

cost[n - 2] = board[n - 2] + board[n - 1] # cost if starting at cell before last cell

path[n - 2] = n - 1 # from cell before last, move into last cell

# now fill the rest of the table

for i in range(n - 3, -1, -1):

# cost[i] = board[i] + min(cost[i+1], cost[i+2])

if cost[i + 1] < cost[i + 2]: # case it is cheaper to move to adjacent cell

cost[i] = board[i] + cost[i + 1]

path[i] = i + 1 # so from cell i, one moves to adjacent cell

else:

cost[i] = board[i] + cost[i + 2]

path[i] = i + 2 # so from cell i, one jumps over cell

return cost[0]

def displayPath(board):

# Display path leading to cheapest cost - method displays indices of cells visited

# path - global list where path[i] indicates the cell to move to from cell i

cell = 0 # start path at cell 0

print("path showing indices of visited cells:", end=" ")

print(0, end="")

path\_contents = "0" # cost of starting/1st cell is 0; used for easier tracing

while path[cell] != -1: # -1 indicates that destination/last cell has been reached

print(" ->", path[cell], end="")

cell = path[cell]

path\_contents += " -> " + str(board[cell])

print()

print("path showing contents of visited cells:", path\_contents)

'''Calculates the overall accuracy of Genetic algorithm against Dynamic Programming'''

def Accuracy():

correct = 0

for dp\_cost, ga\_cost in accuracyResult:

if dp\_cost == ga\_cost :

correct += 1

overall\_accuracy = (correct / len(accuracyResult)) \* 100

print("GA Overall Accuracy: ", overall\_accuracy, "%")

# ------------------------------------------------------------------------------

# ---------------------------------Program Main---------------------------------

if \_\_name\_\_ == "\_\_main\_\_":

# f = open("input1.txt", "r") # input1.txt

f = open("input2.txt", "r") # input2.txt

accuracyResult = []

for line in f:

lyst = line.split() # tokenize input line, it also removes EOL marker

lyst = list(map(int, lyst))

# DP solution

cost = [0] \* len(lyst) # create the cache table

path = cost[:] # create a table for path that is identical to path

min\_cost = jumpIt(lyst)

print("game board:", lyst)

print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")

print("DP Solution")

print("minimum cost: ", min\_cost)

displayPath(lyst)

print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")

print("GA Solution")

maxGeneration = 700

probCrossover = 0.8

probMutation = 0.8

ga = GeneticAlgorithm(lyst[1:], maxGeneration, probCrossover, probMutation)

ga.geneticAlgorithm()

max\_fittest = ga.getFittestSolution()

print("===========================================================================================")

print()

accuracyResult.append((min\_cost, max\_fittest))

Accuracy()

# ---------------------------------End of Program-------------------------------