CSC 535 Data Mining

Assignment 3 Report Collection

Submitted to:

Dr. Jamil Saquer

Author(s):

*Chase Dickerson*

*Jacob Schaum*

**Report**

**Introduction**

For this assignment, we were asked to find a solution to the game “Jump-It”, by using a Genetic Algorithm. Our results were to be compared to a Dynamic Programming solution. For this assignment, we were provided two input files to test our implementation. The two files contained sets of different boards varying in length and in numbers.

**Background**

We used an implementation of a Genetic Algorithm to solve the problem. Genetic Algorithms attempt to computationally mimic the process by which natural selection operates and apply its ideas to solve problems.

**Implementation**

Our implementation followed the traditional Genetic Algorithm. We implemented crossover, mutations, and used a version of roulette wheel. Our crossover function accepts two chromosomes. It has a crossover probability of .5. It is shown in figure 1.

**def** crossover**(**selection1**,** selection2**):**

# Check if previous element in index is 0 for two 0's in a row

crossOverProb **=** 0.5

crossOverIndex **=** **None**

**for** i **in** range**(**len**(**selection1**)):**

randomIndex **=** random**.**randint**(**1**,**len**(**selection1**))**

crossoverChance **=** random**.**random**()**

**if** crossoverChance **<** crossOverProb**:**

crossOverIndex **=** randomIndex

**break**

# cut everyhing from right of index and swap with same in other parent

**if** **not** crossOverIndex **is** **None:**

child1 **=** selection1**[**0**:**crossOverIndex**]** **+** selection2**[**crossOverIndex**:]**

child2 **=** selection2**[**0**:**crossOverIndex**]** **+** selection1**[**crossOverIndex**:]**

**return(**child1**,** child2**)**

**else:**

**return(**selection1**,** selection2**)**

Figure 1: Crossover Function

Our mutate function accepts a child to mutate. Our mutate probability is set to a low value of .02. This will make mutating unlikely but not impossible. Our mutate function is shown in Figure 2.

Figure 2: Mutation

**def** mutate**(**board**):**

newBoard **=** board

mutationProb **=** 0.02

**for** i **in** range**(**len**(**newBoard**)):**

mutateChance **=** random**.**random**()**

**if** mutateChance **<** mutationProb**:**

**if** newBoard**[**i**]** **==** 0**:**

newBoard**[**i**]** **=** 1

**else:**

newBoard**[**i**]** **=** 0

**return** newBoard

**Experimental Setup and Results**

We first start out by testing our results with input1.txt file which contained smaller boards and reasonable numbers. We ran into some issues at the beginning and eventually took out all but one row from the file to just test one board in which we new what the answer was likely to be. Once we eventually worked out the kinks, we decided to test with the input2.txt file which contained bigger boards.

Overall our GA accuracy was very good. We got consistent results of over 80% and most of time were well over 90. Sometimes we would find a lower cost than the DP solution. For input2.txt, one of our accuracies was outputted as “GA Overall Accuracy 0.9635151925900824”.

**Conclusion**

In conclusion, we learned a lot about how Genetic Algorithms work and how they can be useful. We found it interesting on how different the results can be with each run of the algorithm. All in all, Genetic Algorithms can be useful when it comes to data mining.

**Code**

# Program: HW3 Jump-It Game

# File: assn3.py

# Developers: Chase Dickerson and Jacob Schaum

**import** random

**import** math

**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

**def** gaJumpIt**(**board**):**

origBoard **=** board**[**0**:]**

alleleCount **=** len**(**board**)**

# Create parents 1 and d

parent1 **=** generateInitialChromosome**(**alleleCount**)**

parent2 **=** generateInitialChromosome**(**alleleCount**)**

# assign parents to selections

selection1 **=** parent1

selection2 **=** parent2

cost1 **=** 0

cost2 **=** 0

**for** i **in** range**(**0**,**500**):**

# Find two children using crossover

child1**,** child2 **=** crossover**(**selection1**,** selection2**)**

child1 **=** mutate**(**child1**)**

child2 **=** mutate**(**child2**)**

child1 **=** fixConsecutives**(**child1**)**

child2 **=** fixConsecutives**(**child2**)**

# Find the cost of each chromosome

parent1\_cost **=** getCost**(**board**,**selection1**)**

parent2\_cost **=** getCost**(**board**,**selection2**)**

child1\_cost **=** getCost**(**board**,**child1**)**

child2\_cost **=** getCost**(**board**,**child2**)**

total\_cost **=** parent1\_cost **+** parent2\_cost **+** child1\_cost **+** child2\_cost

# Find probabilty of each cost of Chromosome

parent1\_chance **=** math**.**ceil**(**100**\*(**parent1\_cost**/**total\_cost**))**

parent2\_chance **=** math**.**ceil**(**100**\*(**parent2\_cost**/**total\_cost**))**

child1\_chance **=** math**.**ceil**(**100**\*(**child1\_cost**/**total\_cost**))**

child2\_chance **=** math**.**ceil**(**100**\*(**child2\_cost**/**total\_cost**))**

# Find new set of chromosomes

**if** parent1\_chance **<=** parent2\_chance **and** parent1\_chance **<=** child1\_chance **and** parent1\_chance **<=** child2\_chance**:**

selection1 **=** parent1

cost1 **=** parent1\_cost

elif parent2\_chance <= parent1\_chance and parent2\_chance <= child1\_chance and parent2\_chance <= child2\_chance:

selection1 = parent2

cost1 = parent2\_cost

elif child1\_chance <= parent1\_chance and child1\_chance <= parent2\_chance and child1\_chance <= child2\_chance:

selection1 = child1

cost1 = child1\_cost

else:

selection2 = child2

cost2 = child2\_cost

print("GA Solution")

if cost1 < cost2:

print("minimum cost", cost1)

print("path showing indices of visited cells:", displayIndices(origBoard, selection1))

print("path showing contents of visited cells:", displayVisited(origBoard, selection1))

return cost1

else:

print("minimum cost", cost2)

print("path showing indices of visited cells:", displayIndices(origBoard, selection2))

print("path showing contents of visited cells:", displayVisited(origBoard, selection2))

return cost2

def displayIndices(board, selection):

indices = ""

for i in range(len(selection)):

if selection[i] == 1:

if indices == "":

indices += str(i)

elif i < len(selection):

indices += " -> " + str(i)

return indices

def displayVisited(board, selection):

path = ""

for i in range(len(selection)):

if selection[i] == 1:

if path == "":

path += str(board[i])

elif i < len(selection):

path += " -> " + str(board[i])

return path

def fixConsecutives(board):

for i in range(len(board)):

if i != 0 and board[i-1] == 0 and board[i] == 0:

fixChance = random.random()

if fixChance < 0.5:

board[i-1] = 1

else:

board[i] = 1

return board

def mutate(board):

newBoard = board

mutationProb = 0.02

for i in range(len(newBoard)):

mutateChance = random.random()

if mutateChance < mutationProb:

if newBoard[i] == 0:

newBoard[i] = 1

else:

newBoard[i] = 0

return newBoard

def crossover(selection1, selection2):

# Check if previous element in index is 0 for two 0's in a row

crossOverProb = 0.5

crossOverIndex = None

for i in range(len(selection1)):

randomIndex = random.randint(1,len(selection1))

crossoverChance = random.random()

if crossoverChance < crossOverProb:

crossOverIndex = randomIndex

break

# cut everyhing from right of index and swap with same in other parent

if not crossOverIndex is None:

child1 = selection1[0:crossOverIndex] + selection2[crossOverIndex:]

child2 = selection2[0:crossOverIndex] + selection1[crossOverIndex:]

return(child1, child2)

else:

return(selection1, selection2)

def getCost(board, alleles):

cost = 0

for i in range(0, len(alleles)):

if alleles[i] is 1:

cost += board[i]

return cost

def generateInitialChromosome(alleleCount):

chromosome = []

for i in range(alleleCount):

# Ensures there's never 2 zeros in a row

if i != 0 and chromosome[i-1] == 0:

chromosome.append(1)

else:

if random.random() < 0.5:

chromosome.append(0)

else:

chromosome.append(1)

return chromosome

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)

def main():

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

global cost, path

min\_cost = 0

min\_ga\_cost = 0

for line in f:

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

lyst = list(map(int, lyst))

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("cost: ", min\_cost)

displayPath(lyst)

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

min\_ga\_cost += gaJumpIt(lyst)

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

print("GA Overall Accuracy", min\_cost/min\_ga\_cost)

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

main()