Faculty of Computer & Information Sciences
Ain Shams University
Subject: CSC 343 Artificial Intelligence
Year: (3rd year)undergraduate
Academic year: 2nd term 2019-2020

Research Topic (3)

Title: Evolutionary Computation, Classification and Search

1. Introduction

- **A* algorithm:**

  The A* algorithm is one of the most popular algorithms used in many applications, some of its popular applications is:

  a) Game development

     Used in games that requires path finding and puzzle.

  b) Traffic applications

     Used to find the shortest path from a point to another.

- **KNN Algorithm:**

  The KNN algorithm is a classifier that is often used in a variety of applications such as

  a) Economic forecasting
  b) Data compression
  c) Genetics

  And there are many more complex evolutions of this a classifier such as

  Artificial Neural Networks (ANN) and **Support Vector Machines (SVM).**

- **Genetic Algorithm (GA):**

  Genetic Algorithm is mainly used in optimization problems, but it is also used in other applications such as:

  a) DNA Analysis
  b) Economics

c) Neural Networks

d) Image Processing

And many other applications in different fields in real life.

2. The algorithms

2.1. A*

2.1.1. The main steps of the algorithm:

Main containers in the class SearchAlgorithm:

I. nodes: a 2D array of nodes represents the maze.

II. open: a list contains childes of each node in the full path.

III. close: a list contains the visited nodes (nodes in the path).

1) Represent the maze string into 2D array of nodes, each node has {ID, neighbors, edge cost, parent, g(n), h(n), heuristic function (F(n))}
2) Define the Start node and Goal node.
3) Add Start node to the Open list.
4) Update the neighbors of the current node and add them to the Open list sorted by huresticFn.
5) Add the current node to the Close list and remove it from Open list.
6) If the current node is the Goal node then return the ID of nodes in the Close list (the path).
7) Else pop the first node in the Open list, then repeat from step 4.

2.1.2. The implementation of the algorithm:

- Class node represents each node in the maze

```
class Node:
    id =-1
    up =-1
    down =-1
    left = -1
    right = -1
    previousNode = -1
    edgeCost = -1
    gOfN = -1 # total edge cost
    hOfN = -1 # heuristic value
    heuristicFn = -1

    def __init__(self, value, id, up, down, left, right, prevNode, edgeCost, gOfN, hOfN,fOfN):
        self.value = value
        self.id = id
        self.up=up
        self.down=down
        self.left=left
        self.right=right
        self.previousNode=prevNode
        self.edgeCost=edgeCost
        self.gOfN=gOfN
        self.hOfN=hOfN
        self.heuristicFn=fOfN
```

- Attributes in class SearchAlgorithms

```
class SearchAlgorithms:

    nodes = [] #2D array of nodes
    startNode = None #[value, X, Y]
    goalNode = None #[value, X, Y]
    open = []
    close = []
    path = []  # Represents the correct path from start node to the goal node.
    fullPath = []  # Represents all visited nodes from the start node to the goal node.
    totalCost = None
```

- Constructor of class SearchAlgorithm

```
def __init__(self, mazeStr,edgeCost = None):
    column = mazeStr.split(' ')
    for i in range(len(column)):
        self.nodes.append([])
        row = column[i].split(',')
    # -- finding startNode and endNode --
        if self.startNode is None:
            self.GetStartAndGoal(mazeStr.index('S'),mazeStr.index('E'), len(column), len(column[0]), mazeStr)
    # -- represent the maze as 2D array of nodes --
        for j in range(len(row)):
            self.nodes[i].append(Node(row[j], i*len(row)+j, None, None, None, None,
                                -1, -1, 0, self.Heuristic(j,i), sys.maxsize ))
            #edge cost
            self.nodes[i][j].edgeCost = edgeCost[i*len(row)+j]
            #left
            if j is not 0:
               self.nodes[i][j].left = [i, j-1]
            #right
            if j is not len(row)-1:
                self.nodes[i][j].right = [i, j+1]
            #up
            if i is not 0:
                self.nodes[i][j].up = [i-1, j]
                #down
                self.nodes[i-1][j].down = [i,j]
```

- A* Function

1)

```
def AstarManhattanHeuristic(self):
    #seting start node
    self.nodes[self.startNode[1]][self.startNode[2]].heuristicFn = \
        self.nodes[self.startNode[1]][self.startNode[2]].hOfN + self.nodes[self.startNode[1]][self.startNode[2]].gOfN

    #append start node to Open list
    self.open.append(self.nodes[self.startNode[1]][self.startNode[2]])

    prevNode = None
    while len(self.open) != 0:
    # -- sorting the Open list ordered by heuristicFn --
        self.open.sort(key= lambda node: node.heuristicFn)

    # -- pop the least order node to get its neighbours and append it to Close list --
        currentNode = self.open.pop(0)
        # ==handel the case of trying a wrong path (if the selected path was blocked)==
        if prevNode is not None and currentNode.previousNode is not prevNode.id:
            while currentNode.previousNode is not prevNode.id:
                prevNode = self.close.pop()
```

2)

```
# -- append the current node to Close list and append its id to the fullPath --
    self.close.append(currentNode)
    self.fullPath.append(currentNode.id)

# -- if the goal is reached --
    if currentNode.value == self.goalNode[0]:
        for obj in self.close:
            self.path.append(obj.id)
        self.totalCost = currentNode.gOfN
        break

    prevNode = currentNode
return self.fullPath, self.path, self.totalCost
```

3)

```
# -- update attributes of neighbours of the current node ( update parent, gOfN, FoFN), and append them to Open list --
    if currentNode.up != None:
        up = self.nodes[currentNode.up[0]][currentNode.up[1]]
        if(currentNode.heuristicFn < up.heuristicFn):
            up.previousNode = currentNode.id
            up.gOfN = currentNode.gOfN +  up.edgeCost
            up.heuristicFn = up.gOfN + up.hOfN
            self.open.append(up)
    if currentNode.down != None:
        down = self.nodes[currentNode.down[0]][currentNode.down[1]]
        if (currentNode.heuristicFn < down.heuristicFn):
            down.previousNode = currentNode.id
            down.gOfN =  down.edgeCost + currentNode.gOfN
            down.heuristicFn = down.gOfN + down.hOfN
            self.open.append(down)
    if currentNode.left != None:
        left = self.nodes[currentNode.left[0]][currentNode.left[1]]
        if (currentNode.heuristicFn < left.heuristicFn):
            left.previousNode = currentNode.id
            left.gOfN = currentNode.gOfN +  left.edgeCost
            left.heuristicFn = left.gOfN + left.hOfN
            self.open.append(left)
    if currentNode.right != None:
        right = self.nodes[currentNode.right[0]][currentNode.right[1]]
        if (currentNode.heuristicFn < right.heuristicFn):
            right.previousNode = currentNode.id
            right.gOfN = currentNode.gOfN +  right.edgeCost
            right.heuristicFn = right.gOfN + right.hOfN
            self.open.append(right)
```

- Define Start node and Goal node

```
'''
    this function defines the index of the Start node, and Goal node in
    the 2D array of nodes
'''
def GetStartAndGoal(self, startIndex, goalIndex, strRowSize, str):

    startTmpX = goalTmpX = startNodeX = goalNodeX = 0

    startNodeY = int(startIndex / (strRowSize+1))
    goalNodeY = int(goalIndex / (strRowSize+1))
    if startNodeY == 0:
        for i in range(strRowSize-1):
            if str[i] == 'S':
                startNodeX = startTmpX
                break
            startTmpX+=1
    else:
        for i in range(startNodeY*(strRowSize+1), startNodeY*strRowSize + (strRowSize+1)):
            k = str[i]
            if str[i] == 'S':
                startNodeX = startTmpX
                break
            if str[i] != ',': startTmpX+=1

if goalNodeY == 0:
    for i in range(strRowSize-1):
        if str[i] == 'E':
            goalNodeX = goalTmpX
            break
        goalTmpX+=1
else:
    for i in range(goalNodeY*(strRowSize+1), goalNodeY*strRowSize + (strRowSize)):
        k = str[i]
        if str[i] == 'E':
            goalNodeX = goalTmpX
            break
        if str[i] != ',': goalTmpX+=1

self.startNode = ['S', startNodeY, startNodeX]
```

- Calculate h(n)

```
"""
    this function calculates the h(n) of a node
    given its index in the 2D array of nodes
"""
def Heuristic(self, x, y):
    return abs(x - self.goalNode[2]) + abs(y - self.goalNode[1])
```

2.1.3. Sample run (the output)

```
**ASTAR with Manhattan Heuristic **
Full Path:[0, 7, 14, 21, 22, 29, 1, 2, 9, 16, 17, 18, 25, 32, 31]
Path is: [1, 2, 9, 16, 17, 18, 25, 32, 31]
Total Cost: 30
```

2.2. K-Nearest Neighbors

2.2.1. The main steps of the algorithm:

1) Split the dataset into 2 sets (Train set, and Test set).
2) Calculating and store the Euclidean distance between each data Point in the test set and each data point in the train set.
3) Sort the distances Ascending.
4) Get labels of the first K data points.
5) Classify the test point based on most classes in the first K nearest points.
6) Calculate the accuracy by comparing the predicted classes with the test classes

   (Accuracy = count of correct predictions).

2.2.2. The implementation of the algorithm:

- Calculating the Euclidean distance

```
# region KNN
class KNN_Algorithm:

    def __init__(self, K):
        self.K = K

    def euclidean_distance(self, p1, p2):
        sum =0
        for i in range(len(p1)-1):
            sum += (p1[i] - p2[i])**2
        return sqrt(sum)
```

- KNN Function

```
def KNN(self, X_train, X_test, Y_train, Y_test):
    yPred =[]
    for p1 in X_test:
        destencesList = []
        neighbours = []
        classesOfNeighbours = {0:0, 1:0}
    #1- Calculating euclidean distance
        for p2 in  range(len(X_train)):
            destencesList.append([p2,self.euclidean_distance(p1,X_train[p2])])
    #2- sorting distances
        destencesList.sort(key= lambda p: p[1])
        #3- get the first K elements
        for i in range(self.K):
            pointIndex = destencesList[i][0]
            neighbours.append([pointIndex, Y_train[pointIndex]])
    #4- Predection
        # -- for integer classes only --
        for i in range(self.K):
            classesOfNeighbours[neighbours[i][1]] += 1
        yPred.append(max(set(classesOfNeighbours), key= classesOfNeighbours.get))

        # -- for multiple classes --
        # for i in range(self.K):
        #     if  neighbours[i][1] not in classesOfNeighbours:
        #         classesOfNeighbours.update({neighbours[i][1]: 0})
        #     classesOfNeighbours[neighbours[i][1]] += 1
        # yPred.append(max(set(classesOfNeighbours), key= classesOfNeighbours.get))
```

- Calculating Accuracy

```
#5- calculating accuracy
    for i in range(len(yPred)):
        c = 0
        if yPred[i] == Y_test[i]:
            c+=1
    return c*100/len(Y_test)*100
```

2.2.3. Sample run (the output)

```
KNN Accuracy: 87.71929824561403
```

2.3. Genetic Algorithm

2.3.1. The main steps of the algorithm

1) Randomly get an initial population of individuals, each individual is a list of genes, each gene is a list of cities.
2) Get the fitness value for each individual in the population.
3) Select pair of individuals from current population, parent chromosomes are selected based on a probability related to their fitness using Roulette Wheel Selection.
4) Apply genetic operators (crossover and mutation) to create a pair of offspring individuals.
5) Repeat from step 3 until the size of the new population reaches the given population size.
6) Go to step 2 and repeat with the new population until reaching the given number of generations.
7) Select the individual with least fitness value from the final generation.

2.3.2. The implementation of the algorithm:

- Attributes of the class

```
# region GeneticAlgorithm
class GeneticAlgorithm:
    Cities = [1, 2, 3, 4, 5, 6]
    DNA_SIZE = len(Cities)
    POP_SIZE = 20
    GENERATIONS = 5000
    Pc = 0.9
    Pm = 0.01
```

- Creates the initial population randomly

```
"""
   Return a list of POP_SIZE individuals, each randomly generated via iterating
   DNA_SIZE times to generate a string of random characters with random_char().
"""

def random_population(self):
    pop = []
    for i in range(1, 21):
        x = r.sample(self.Cities, len(self.Cities))
        if x not in pop:
            pop.append(x)
    return pop
```

- Calculate the fitness value

```
"""
calculate the fitness value of a given DNA (individual)
using the function Cost()
"""
def fitness(self, dna):
    sum = 0
    for i in range(len(dna)):
        if i == len(dna)-1:
            sum += self.cost(dna[i], dna[0])
            break
        sum += self.cost(dna[i], dna[i+1])
    return sum
```

- Randomly select pair of individuals using Roulette Wheel Selection

```
"""
- Chooses a random element from items, where items is a list of tuples in
   the form (item, weight (cumulative fitness)).
- weight determines the probability of choosing its respective item.
 """
#Roulate Wheel (chooses based on cumulative fitness)
def weighted_choice(self, items):
    weight_total = sum((item[1] for item in items))
    n = r.uniform(0, weight_total)
    for item, weight in items:
        if n < weight:
            return item
        n = n - weight
    return item
```

- Apply the mutation operation

```
"""
   For each gene in the DNA, there is a 1/mutation_chance chance that it will be
   switched out with a random character. This ensures diversity in the
   population, and ensures that is difficult to get stuck in local minima.
"""

def mutate(self, dna, random1, random2):
   if random1 <= self.Pm:
       index = int(random2*len(dna))
       mutateList = dna[index:]
       tmp = dna[:index]
       random.shuffle(mutateList)
       tmp+=mutateList
       return tmp
   return dna
```

- Apply the crossover operation

```
"""
apply the crossover operation on a pair of DNAs by splitting
each of them at a random index and swapping the splited parts
"""
def crossover(self, dna1, dna2, random1, random2):
   if random1 <= self.Pc:
       index = int(random2*len(dna1))
       b1 = dna1[index:]
       b2 = dna2[index:]
       NewDNA1 = dna1[:index]
       NewDNA2 = dna2[:index]
       for i in range(len(b1)):
           if b2[i] not in NewDNA1:
               NewDNA1.append(b2[i])
           if b1[i] not in NewDNA2:
               NewDNA2.append(b1[i])
       if len(NewDNA1) is not len(dna1):
           for i in b1:
               if i not in NewDNA1:
                   NewDNA1.append(i)
       if len(NewDNA2) is not len(dna2):
           for i in b2:
               if i not in NewDNA2:
                   NewDNA2.append(i)

       return NewDNA1, NewDNA2
   return dna1, dna2
```

2.3.3. Sample run (the output)

```
[5, 3, 2, 1, 6, 4]
58
```

3. Discussion

I will discuss the code of each function in the 3 algorithms,

1) A* Algorithm

A. *Constructor of the class*

```
def __init__(self, mazeStr,edgeCost = None):
```

in this function I make 2 main tasks:

i. Find the Start and Goal node in the Maze string using the function

```
GetStartAndGoal(self, startIndex, goalIndex, strRowSize, str):
```

ii. Represent the maze in a 2D array of nodes by splitting the Maze string into Columns without in between spaces

```
column = mazeStr.split(' ')
```

then split each column into rows without commas in between

```
row = column[i].split(',')
```

then create a new node for each cell in the row and append it to 2D array of nodes.

```
self.nodes[i].append(Node(row[j], i*len(row)+j, None, None, None, None,
                          -1, -1, 0, self.Heuristic(j,i), sys.maxsize ))
```

then add the Edge cost, and neighbors for this node.

If there is a way to enhance efficiency of this function it will be by enhancing the algorithm of representing the maze into 2D array of nodes or enhancing the algorithm of getting the start and goal node.

B. *The A* function*

```
AstarManhattanHeuristic(self)
```

First step in this function is to set the attributes of the Start node and append it to the Open list, then loop as long as the Open list is not empty, or the Goal node is not reached.

In this loop I make 3 main tasks:

i. sorting the Open list ordered by heuristicFn
ii. pop the least order node to get its neighbors and append it to Close list
iii. update attributes of neighbors of each visited node (update parent, gOfN, FoFN), and append them to Open list

C. Find Start and Goal node

```
GetStartAndGoal(self, startIndex, goalIndex, strRowSize, str)
```

In this function I use the index of Start node and Goal node in the maze string to get their X, Y indices in the 2D array of nodes.
this function can be enhanced by enhancing the algorithm of searching for the start and the goal node.

2) KNN Algorithm

A. The KNN function

```
KNN(self, X_train, X_test, Y_train, Y_test)
```

This function has a one main loop that iterates on the data points in the test set, in this loop there are main 5 tasks:

i. Calculating Euclidean distance

Between each test data point and all train data points and store them.

ii. sorting distances ascending.

iii. Get the nearest K elements and store them

iv. Prediction

Predict the class of each test data point based on majority of the train classes.

v. calculating accuracy

Compare the predicted classes with the test classes, the accuracy will be the number of correct predicted class.

This function can be enhanced by generalize the algorithm of prediction to be suitable for multiple classes prediction not only 2 class.

3) Genetic Algorithm

A. Calculate fitness function

```
fitness(self, dna)
```

this function calculates the fitness value for a given DNA by summing the cost between each 2 consecutives cities.

B. Apply mutate operation

```
mutate(self, dna, random1, random2)
```

there is a condition in this function which is the random number must be less than the probability of mutation

```
if random1 <= self.Pm:
```

in this if condition body the DNA genes will be shuffled starting from a random index.

C. Apply crossover operation

```
crossover(self, dna1, dna2, random1, random2)
```

the crossover operation will be applied if the random number is less than or equal the probability of crossover operation by split both of the 2 DNAs at a random index and swap them together taking into consideration of avoiding repeating genes.

This algorithm can be enhanced by enhancing the algorithm of applying crossover operation without repeated genes and enhancing the mutation function.

4. References

[1] https://brilliant.org/wiki/a-star-search/#a-star-search

[2] https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/#how-does-knn-work

[3] https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_application_areas.htm

[4] https://www.geeksforgeeks.org

[5] https://www.w3schools.com

[6] Lab 2

[7] Lab 5 (KNN)

[8] Genetic Algorithm Lab