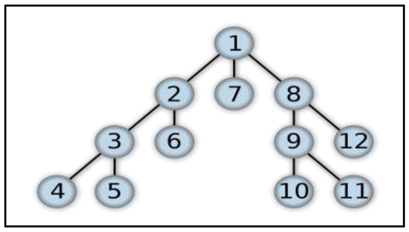
Faculty of Computer & Information Sciences
Ain Shams University
Subject: CSC 343 Artificial Intelligence
Year: (3rd year)undergraduate
Academic year: 2nd term 2019-2020

Research Topic (2)

Title: Clustering, Gaming and Search

تحذير هام: علي الطالب عدم كتابة اسمه أو كتابة اي شيء يدل علي شخصيته

1. Introduction
    a. Depth-first search Applications: -
        i. Depth-first-Search (The algorithm works in $O(m+n)$ time where n is the number of vertices and m is the number of edges.)is An Algorithm of searching in a graph of tree data Structure in a Story Of This Algorithm in by searching At The Start of root And It is The Top in the tree or graph and go deeply in the tree or graph to make a path ,then backtrack until it finds the main path that you want . The algorithm does this until the correct graph has been explored.
        ii. Applications:
            1. Analyzing Networks
            2. Scheduling
            3. Mapping Routes
            4. Finding Spanning Trees are Graph Problems
            5. traveling-salesman problem.
            6. The Ford-Fulkerson algorithm.
            7. topological sorting.



b. Alpha–beta Applications:

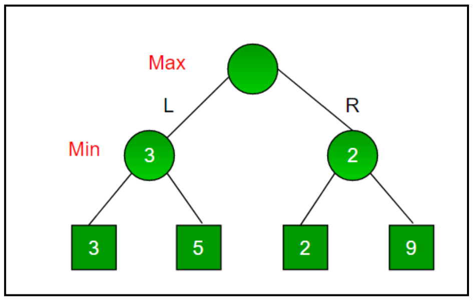
   i. Alpha–beta-pruning is An Upgrade of Minmax Algorithm that is used in decision making and game theory to find the optimal move for a player.

   ii. **Alpha:** The best (highest value) the path of Maximizer. The initial value of alpha is **-∞.**

   iii. **Beta:** The best (lowest value) the path of Minimizer. The initial value of beta is **+∞.**

   iv. Applications:

      1. Tic-Tac-Toe
      2. Mancala
      3. Chess
      4. Backgammon



c. . K-means Applications:

   i. **K means** algorithm is an iterative algorithm that is Do a partition of a dataset into
   non-overlapping subgroups (clusters)

   ii. Applications:

      1. market segmentation
      2. mage segmentation
      3. document clustering
      4. image compression

2. The algorithms

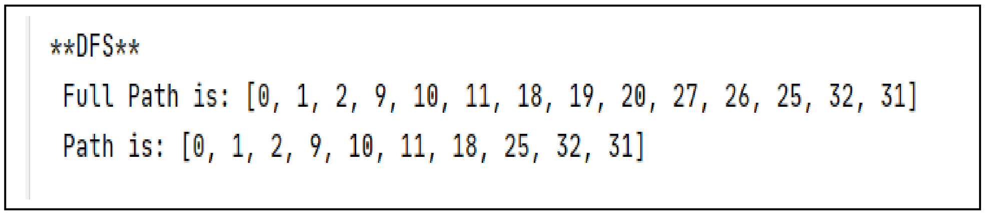
2.1. Depth-first search

2.1.1. The main steps of the algorithm

1. The graph produces the minimum spanning tree and all pair shortest path tree.
2. Detecting cycle in a graph
3. Path Finding by specializing
4. Finding Strongly Connected Components of a graph See this for DFS based algorithm
5. Solving puzzles with just all solutions, like Our maze. DFS could adapted to seek out any solution to the maze by only including nodes on this path within the visited list.

2.1.2. The implementation of the algorithm (your Python code)

2.1.3. Sample run (the output)



```
**DFS**
 Full Path is: [0, 1, 2, 9, 10, 11, 18, 19, 20, 27, 26, 25, 32, 31]
 Path is: [0, 1, 2, 9, 10, 11, 18, 25, 32, 31]
```

2.2. Alpha–beta

2.2.1. The main steps of the algorithm

1. Maximizer go into LEFT side: It is now the minimizers turn. The minimizer now has a choice between x and y. Being the minimizer it will choose the least among both, that is x
2. Maximizer go into RIGHT side: It is now the minimizers turn. The minimizer now has a choice between w and z. He will choose z as it is the least among the two values.

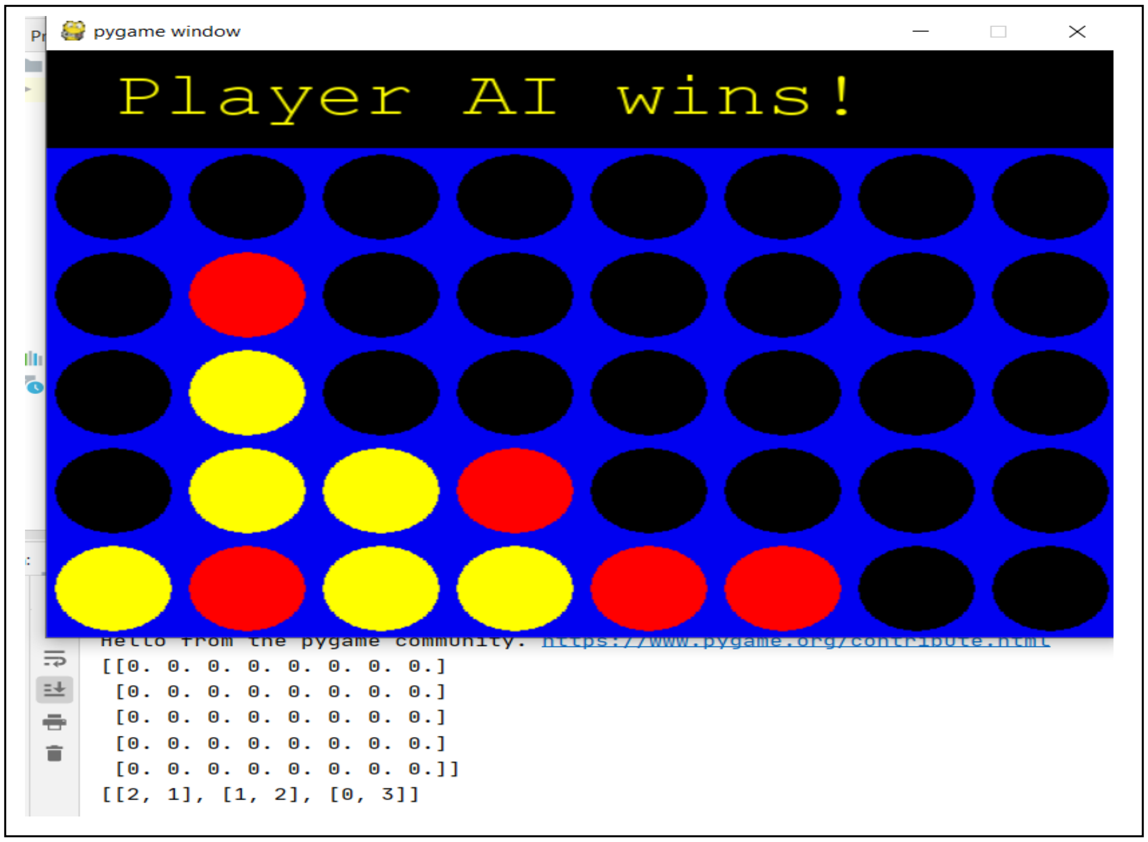
3. Being the maximizer, you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go LEFT and the optimal value is x.

2.2.2. The implementation of the algorithm (your Python code)

```
def AlphaBeta(self, board, depth, alpha, beta, currentPlayer):
    v_locations = self.get_valid_locations(board)
    col = random.choice(v_locations)
    '''Implement here'''
    if depth==0 or self.is_terminal_node(board):
        if self.is_terminal_node(board):
            if self.winning_move(board,self.PLAYER_PIECE):
                                    return(None,-math.inf)
            elif self.winning_move(board,self.AI_PIECE):
                                    return(None,math.inf)
            else:
                                    return (None,0)
       else:
           return (None,self.score_position(board,self.AI_PIECE))
    if currentPlayer:# Max Player The Ai One
        value = -math.inf
        for i in v_locations:
            Row = self.get_next_open_row(board, i)
            board_copy = board.copy()
            self.drop_piece(board_copy,Row,i,self.AI_PIECE)
            new_score = self.AlphaBeta(board_copy,depth -
1,alpha,beta,False)[1]
            if new_score > value:
                col = i
                value = new_score
            alpha = max(alpha,value)
            if alpha >= beta:
                break
        return col, value
```

```
else:  # Min player The Person One
    col = random.choice(v_locations)
    value = math.inf
    for i in v_locations:
        row = self.get_next_open_row(board, i)
        board_copy = board.copy()
        self.drop_piece(board_copy,row,i, self.PLAYER_PIECE)
        new_score = self.AlphaBeta(board_copy, depth - 1, alpha, beta, True)[1]
        if new_score < value:
            value = new_score
            col = i
        beta = min(beta, value)
        if alpha >= beta:
            break
    return col, value
```

2.2.3. Sample run (the output)



```
Hello from the pygame community. https://www.pygame.org/contribute.html
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
[[2, 1], [1, 2], [0, 3]]
```

2.3. K-means

2.3.1. The main steps of the algorithm

1. Specify number of clusters K.
2. Initialize centroids by first shuffling on the dataset and then randomly selecting *K* data points for the Centro ids without replacement.
3. Keep iterating until there is no change to the Centro ids.
4. Compute the sum of Manhattan distance or Euclidean distance between data points and all centroids.
5. Assign each data point to the closest cluster (centroid).
6. Compute the centroids for the clusters by taking the average of all data points that belong to each cluster.

2.3.2. The implementation of the algorithm (your Python code)

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

```
def getClusters(self):
   self.initClusters()
   '''Implement Here'''
   for i in range(self.noOfIterations):
      for item in self.data:
         max_iterat= 100
         for cluster in self.clusters:
            if(self.isEuclidean==0):
               clusterDistance =
self.distance.Manhattan_distance(cluster.centroid, item.features)
            elif(self.isEuclidean==1):
               clusterDistance =
self.distance.euclidean_distance(cluster.centroid,item.features)
            if (clusterDistance < max_iterat):
               item.clusterId = cluster.id
               max_iterat = clusterDistance
         clusterData = [
                     x for x in self.data
                    if x.clusterId == item.clusterId
                   ]
         self.clusters[item.clusterId].update(clusterData)
   return self.clusters
```

2.3.3. Sample run (the output)

in case of euclidean_distance (euclidean_distance==1)

```
[0.5, 0.5, 0.0, 0.5]
[0.5, 0.5, 1.0, 0.5]
```

In case of Manhattan_distance (euclidean_distance==0)

```
[0.5, 0.5, 0.0, 0.5]
[0.5, 0.5, 1.0, 0.5]
```

In case of Manhattan_distance (euclidean_distance==0) but The out But of Commented Code

```
[0.5, 0.5, 0.5, 0.5]
[1, 0, 1, 0]
```

3. Discussion

The Algorithm Order:

Depth-first search (DFS)-> in Worst Case **$O(n)$**.

Alpha–beta pruning -> in Worst case **$O(b^m)$**.

k-means algorithm -> in worst Case **$O(n^2)$.**

The most efficient algorithm is **Depth-first search (DFS)** with the smallest complexity.

The Worst of The Is Alpha-Beta Pruning with Biggest Complexity.

References

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

Intelligence, Artificial. *Gaming Algorithms Lab 4.*

—. *K-Means Lab 8.*