[Topic 2 Algorithms]

General instructions:

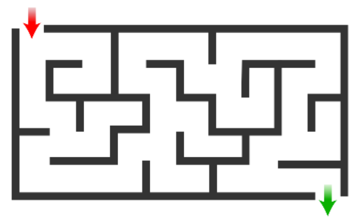
Regarding your task:

1. Download the template file (Topic2.py).
2. **Don't rename** the python file.
3. You should submit **the same python** file in addition to your documentation.
4. Submit **only running** code that you have tested before.
5. **Compressed** files (.zip/.rar) are **not allowed.**
6. Please, read the documentation carefully.
7. Clear the output after being displayed for multiple runs.
8. This project will be **auto graded**.
9. Copying or Getting code from online resources (including YouTube) is considered as cheating case.
10. **Do not change** any class functions signature (parameters, or order).
11. You can add any extra attributes, functions or classes you need as long as the **main structure is left as it is**.
12. **Implement** the given functions.
13. **Install any missing library in your package which is imported in the file and use Python 3.6.**

Plagiarism checking will be applied. Research is subject to rejection in such case. All submissions will be checked for plagiarism automatically.

Algorithm (1)

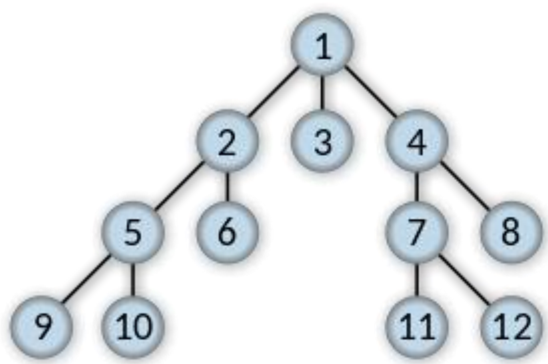
1. In this project, you are expected to solve a 2-D maze using DFS. A maze is path typically from start node 'S' to Goal node 'E'.



Input: 2D maze represented as a string.

Output: the full path from Start node to End node (Goal Node), direct path to go from Start to End directly.

Example: let's say the end node is 6 in case of DFS.



Full Path: 1, 2, 5, 9, 10, 6.

Path: 1, 2, 6.

The input and output are explained below. Your code should be generic for any dimension of a given maze.

Maze: 'S,.,.,#,.,.,. .,#,.,.,.,#,. .,#,.,.,.,.,. .,.,#,#,.,.,. #,.,#,E,.,#,.'

- Maze is a string, rows are separated by **space** and columns are separated by **comma** ','.
- The board is read **row wise**, the nodes are numbered **0-based** starting the leftmost node.
- You have to create your own board **as a 2D array (NO 1D ARRAY ALLOWED)** of **Nodes**.

```
S . . # . . .
. # . . . # .
. # . . . . .
. . # # . . .
# . # E . # .
```

Topic2.py file has search algorithms region

The search algorithms region contains two classes:

a. Class Node represents a cell in the board of game. You can add extra attributes, but you do not delete current attributes or neglect them.

```python
class Node:
    id = None  # Unique value for each node.
    up = None  # Represents value of neighbors (up, down, left, right).
    down = None
    left = None
    right = None
    previousNode = None  # Represents value of neighbors.


    def __init__(self, value):
        self.value = value
```

b. Class Search Algorithms:

```python
class SearchAlgorithms:
    ''' * DON'T change Class, Function or Parameters Names and Order
        * You can add ANY extra functions,
          classes you need as long as the main
          structure is left as is '''
    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.


    def __init__(self, mazeStr, edgeCost=None):
        ''' mazeStr contains the full board
         The board is read row wise,
        the nodes are numbered 0-based starting
        the leftmost node'''
        pass

    def DFS(self):
        # Fill the correct path in self.path
        # self.fullPath should contain the order of visited nodes
        # self.path should contain the direct path from start node to goal node
        return self.fullPath, self.path
```

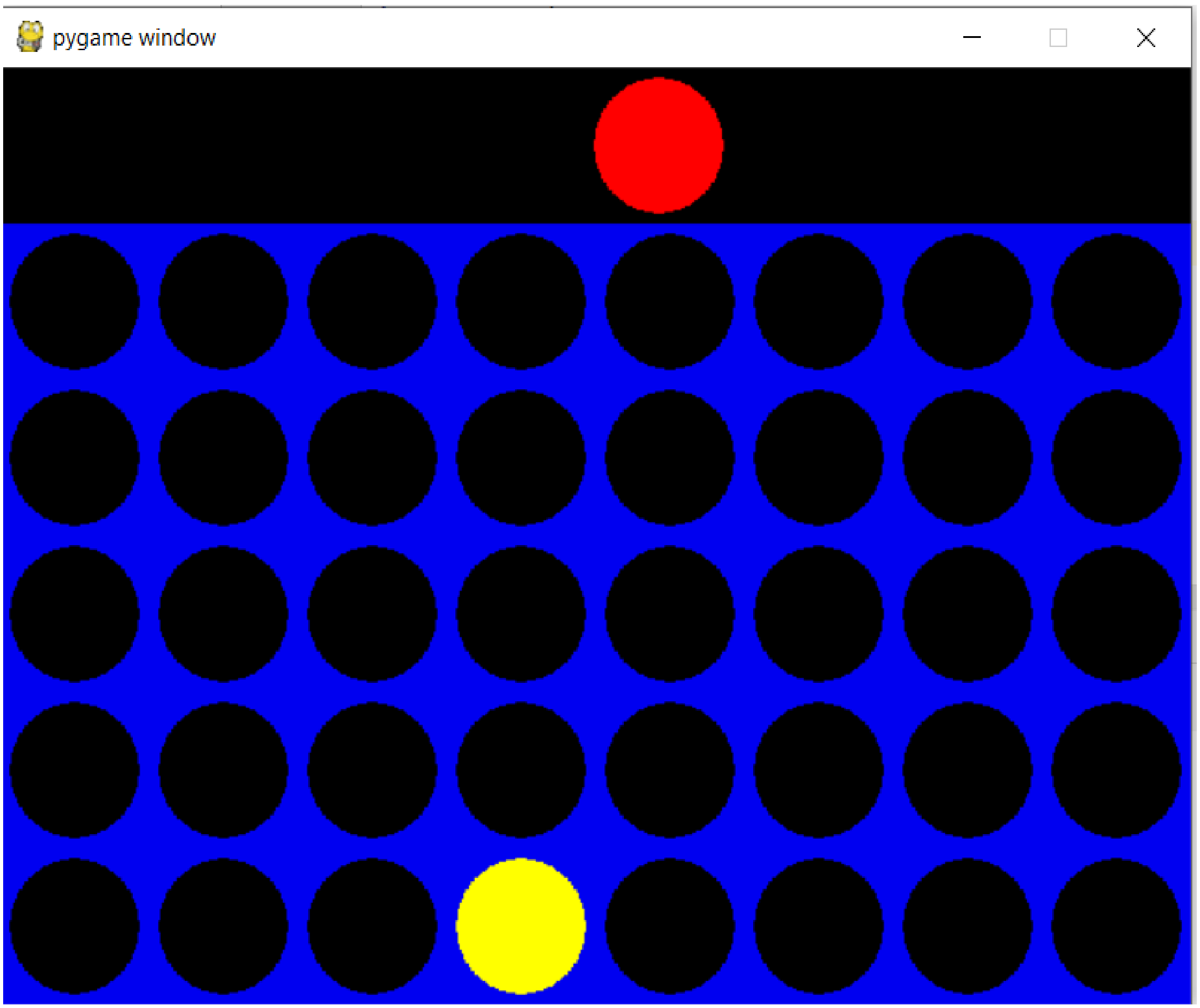
3. The Main Function for search algorithm:

```python
def SearchAlgorithm_Main():
    searchAlgo = SearchAlgorithms('S,.,.,#,.,.,. .,#,.,.,.,#,. .,#,.,.,.,.,. .,.,#,#,.,.,. #,.,#,E,.,#,.')
    fullPath, path = searchAlgo.DFS()
    print('**DFS**\n Full Path is: ' + str(fullPath) +'\n Path is: ' + str(path))
```

Algorithm (2)

2. Implement Connect 3 using Alpha Beta algorithm, this game is played on a vertical board which has 8 columns and 5 rows. Each column has a hole in the upper part of the board, where pieces are introduced. There is a window for every square, so that pieces can be seen from both sides.

Both players have a set of 20 thin pieces (like coins); each of them uses a different color. The board is empty at the start of the game. GUI is already implement, you need to **implement only Alpha beta function**.



Class Hierarchy Functions:

- Gaming
  - __init__(self)
  - AlphaBeta(self, board, depth, alpha, beta, currentPlayer)
  - create_board(self)
  - draw_board(self, board)
  - drop_piece(self, board, row, col, piece)
  - evaluate_window(self, window, piece)
  - get_next_open_row(self, board, col)
  - get_valid_locations(self, board)
  - is_terminal_node(self, board)
  - is_valid_location(self, board, col)
  - pick_best_move(self, board, piece)
  - print_board(self, board)
  - score_position(self, board, piece)
  - winning_move(self, board, piece)

1. __init__ function: constructor to initialize some attributes.
2. AlphaBeta function: needs to be implemented.
3. Create board function: creates a new empty board when game is started.
4. Drop_piece function: set a coin of a position (row, column) with a piece of game (human player or AI player).
5. Evaluate Window function: helps in calculating score of position (utility function).
6. Get_next_open_row function: get the first valid row to be play in.
7. Get_valid_locations function: helps to find the empty valid locations to play in.
8. Is_terminal_node function: check if it’s a leaf node.
9. Is_valid_location function: check if it’s an empty location to play in.
10. Pick_best_move function: check all valid moves and check the best one that grantees to win.
11. Print_board: prints the board in console.
12. Score_position: calculates the utility value.
13. Winning_move: checks if the any player wins.

Algorithm (3)

3. Implement K-Means algorithm on a dataset that holds a diagnosis for the eyes of patients.

- The diagnosis is based on the following features:
  1. Age: (0) young, (1) adult.
  2. Prescription: (0) myope, (1) hypermetrope.
  3. Astigmatic: (0) no, (1) yes.
  4. Tear production rate: (0) normal, (1) reduced.
- The output classes are:
  1. Need contact lenses (1): the patient should be fitted with a special type of contact lenses.
  2. No contact lenses (0): the patient should not be fitted with a
  3. Special type of contact lenses.
- Dataset Sample:

Age	Prescription	Astigmatic	Tear Production Rate	Diagnosis
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0

- The Number of max iterations = 100
- Instead of using random centroids use the first k items from the training set
- Calculate the distance once using Manhattan and once using Euclidean

1. Euclidean: Take the square root of the sum of the squares of the differences of the coordinates.
   - For example, if x = (a, b) and y=(c, d) , the Euclidean distance between x and y is

$$\sqrt{(a-c)^2+(b-d)^2}.$$

2. Manhattan: Take the sum of the absolute values of the differences of the coordinates.
   - For example, if x = (a, b) and y=(c, d) , the Euclidean distance between x and y is

$$|a - c| + |b - d|$$

The output should be the centroids of the classes using Euclidean and Manhattan distances.

Topic2.py file has K-means region

1. The Kmeans region contains four classes:
   a. Data Item class for dataset

```python
class DataItem:
    def __init__(self, item):
        self.features = item
        self.clusterId = -1

    def getDataset():
        data = []
        data.append(DataItem([0, 0, 0, 0]))
        data.append(DataItem([0, 0, 0, 1]))
        data.append(DataItem([0, 0, 1, 0]))
        data.append(DataItem([0, 0, 1, 1]))
        data.append(DataItem([0, 1, 0, 0]))
        data.append(DataItem([0, 1, 0, 1]))
        data.append(DataItem([0, 1, 1, 0]))
        data.append(DataItem([0, 1, 1, 1]))
        data.append(DataItem([1, 0, 0, 0]))
        data.append(DataItem([1, 0, 0, 1]))
        data.append(DataItem([1, 0, 1, 0]))
        data.append(DataItem([1, 0, 1, 1]))
        data.append(DataItem([1, 1, 0, 0]))
        data.append(DataItem([1, 1, 0, 1]))
        data.append(DataItem([1, 1, 1, 0]))
        data.append(DataItem([1, 1, 1, 1]))
        return data
```

b. Cluster:

```
class Cluster:
        def __init__(self, id, centroid):
            self.centroid = centroid
            self.data = []
            self.id = id

        def update(self, clusterData):
            self.data = []
            for item in clusterData:
                self.data.append(item.features)
            tmpC = np.average(self.data, axis=0)
            tmpL = []
            for i in tmpC:
                tmpL.append(i)
            self.centroid = tmpL
```

c. Similarity Distance

```
class SimilarityDistance:
        def euclidean_distance(self, p1, p2):
            pass
        def Manhattan_distance(self, p1, p2):
            pass
```

d. Clustering K-means algorithm class:

```
class Clustering_kmeans:
    def __init__(self, data, k, noOfIterations, isEuclidean):
        self.data = data
        self.k = k
        self.distance = SimilarityDistance()
        self.noOfIterations = noOfIterations
        self.isEuclidean = isEuclidean

    def initClusters(self):
        self.clusters = []
        for i in range(self.k):
            self.clusters.append(Cluster(i, self.data[i * 10].features))

    def getClusters(self):
        self.initClusters()
        pass
```

2. The Main Function for k-means algorithm:

```
def Kmeans_Main():
    dataset = DataItem.getDataset()
    # 1 for Euclidean and 0 for Manhattan
    clustering = Clustering_kmeans(dataset, 2, len(dataset),1)
    clusters = clustering.getClusters()
    for cluster in clusters:
        for i in range(4):
            cluster.centroid[i] = round(cluster.centroid[i], 2)
        print(cluster.centroid[:4])
```