[AI-PACKAGE]

Milestone 1

General instructions:

Regarding your AI-Package:

1. Initially, create a new project named 'AI-Package'.
2. With each milestone, you will add a folder with the Task name.
3. After you finish writing your code:
    - Open the folder shared with your team on GoogleDrive (the one with your team number)
    - Upload all the project files with the same hierarchy.
    - Submit **only running** code that you have tested before.
4. Compressed files (.zip/.rar) are **not allowed.**
5. The Submission of package is only through **your shared folder on google drive.**

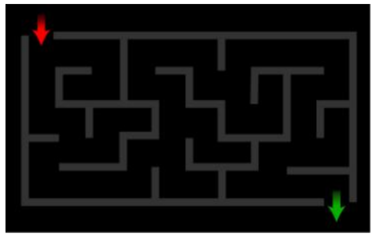
Don't delete any previous milestones

Regarding the search algorithms submission [Milestone 1]:

1. Add a new folder named 'SearchAlgorithms' in the 'AI-Package' project.
2. Add the shared template file named 'SearchAlgorithms.py' to 'SearchAlgorithms' folder.
3. Your code should be written **only** in "SearchAlgorithms.py'" file under 'SearchAlgorithms' folder.
4. Please, read code documentation carefully.
5. Your code should be generic for any dimension of a given maze.
6. This milestone will be **autograded**.

This package is intended for team work contribution. Sharing ideas or part of the answers is considered plagiarism and will not be tolerated.
All submissions will be checked for plagiarism automatically.

In this milestone, you are expected to solve a 2-D maze using
Depth limited search, bidirectional search and best first search. A maze is path typically from start node 'S' to Goal node 'E'.



Input: 2D maze represented as a string and Edge Cost (in case of UCS and A*)

Output:

i. Path: the path to go from Start to End
ii. Full Path: the path of all visited nodes.
iii. Total Cost (only in UCS and A*)

The input and output are explained below.

```
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 of **Nodes**.

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

1- Depth limited search algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit

DLS pseudocode:

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(*problem*.INITIAL-STATE), *problem*, *limit*)

function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution, or failure/cutoff
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  else if *limit* = 0 **then return** *cutoff*
  else
    cutoff_occurred? ← false
    for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
      child ← CHILD-NODE(*problem*, *node*, *action*)
      result ← RECURSIVE-DLS(*child*, *problem*, *limit* − 1)
      if *result* = *cutoff* **then** *cutoff_occurred?* ← true
      else if *result* ≠ *failure* **then return** *result*
    if *cutoff_occurred?* **then return** *cutoff* **else return** *failure*

You Have to implement the Recursive -DLS Here.

```python
def DLS(self):
    # Fill the correct path in self.path
    # self.fullPath should contain the order of visited nodes
    return self.path, self.fullPath
```

And it`s calling is in main function:

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

2- Bidirectional search algorithm:

Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.
you can use any search technique such as BFS-DFS.

BDS pseudocode:

```
BIDIRECTIONAL_SEARCH
Q_I.Insert(x_I) and mark x_I as visited
Q_G.Insert(x_G) and mark x_G as visited
while Q_I not empty and Q_G not empty do
    if Q_I not empty
        x ← Q_I.GetFirst()
        if x = x_G or x ∈ Q_G
            return SUCCESS
        forall u ∈ U(x)
            x' ← f(x, u)
            if x' not visited
                Mark x' as visited
                Q_I.Insert(x')
            else
                Resolve duplicate x'
    if Q_G not empty
        x' ← Q_G.GetFirst()
        if x' = x_I or x' ∈ Q_I
            return SUCCESS
        forall u^-1 ∈ U^-1(x')
            x ← f^-1(x', u^-1)
            if x not visited
                Mark x as visited
                Q_G.Insert(x)
            else
                Resolve duplicate x
return FAILURE
```

Figure 2.7: A general template for bidirectional search.

You have to implement your code here:

```
def BDS(self):
    # Fill the correct path in self.path
    # self.fullPath should contain the order of visited nodes
    return self.path, self.fullPath, self.totalCost
```

And the calling of the function is:

```
searchAlgo = SearchAlgorithms('S,.,.,#,.,.,. .,#,.,.,.,#,. .,#,.,.,.,.,. .,.,#,#,.,.,. #,.,#,E,.,#,.')
path, fullPath = searchAlgo.BDS()
print('**BFS**\nPath is: ' + str(path) + '\nFull Path is: ' + str(fullPath) + '\n\n')
```

3- Best first search algorithm:

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function. Were, h(n)= estimated cost from node n to the goal. [f(n) = h(n)]

BFS pseudocode:

```
Best-first search {

 closed list = [ ]
 open list = [start node]

    do {
            if open list is empty then{
                    return no solution
            }
            n = heuristic best node
            if n == final node then {
                    return path from start to goal node
            }
            foreach direct available node do{
                    if node not in open and not in closed list do {
                            add node to open list
                            set n as his parent node
            }
            delete n from open list
            add n to closed list
    } while (open list is not empty)
 }
```

You have to implement your code here:

```
def BFS(self):
    # Fill the correct path in self.path
    # self.fullPath should contain the order of visited nodes
    return self.path, self.fullPath
```

The calling of the function is here:

```
searchAlgo = SearchAlgorithms('S,.,.,#,.,.,. .,#,.,.,.,#,. .,#,.,.,.,.,. .,.,#,#,.,.,. #,.,#,E,.,#,.', [0, 15, 2, 100, 60, 35, 30, 3
                                                                                                  , 100, 2, 15, 60, 100, 30, 2
                                                                                                  , 100, 2, 2, 2, 40, 30, 2, 2
                                                                                                  , 100, 100, 3, 15, 30, 100, 2
                                                                                                  , 100, 0, 2, 100, 30])
path, fullPath, TotalCost = searchAlgo.BFS()
print('** UCS **\nPath is: ' + str(path) + '\nFull Path is: ' + str(fullPath) + '\nTotal Cost: ' + str(
    TotalCost) + '\n\n')
```

heuristicValue: [0, 15, 2, 100, 60, 35, 30, 3,
100, 2, 15, 60, 100, 30, 2,
100, 2, 2, 2, 40, 30, 2, 2,
100, 100, 3, 15, 30, 100, 2,
100, 0, 2, 100, 30]

heuristicvalue is a list, will be passed for BFS.

Each Node has an Heuristic value that represents the cost from any current node to the goal node.

The template code is explained below.

SearchAlgorithms.py file:

It contains two classes:

b. Class Node represents a cell in the board of game.

```
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.
    edgeCost = None  # Represents the cost on the edge from any parent to this node.
    gOfN = None  # Represents the total edge cost
    hOfN = None  # Represents the heuristic value
    heuristicFn = None  # Represents the value of heuristic function

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

2) Class SearchAlgorithms:

   I. Do not change class functions, parameters, or order
   II. You can add any extra attributes, functions or classes you need as long as the main structure is left as it is.
   III. Implement the given functions.

```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.
    totalCost = -1  # Represents the total cost in case using UCS, AStar (Euclidean or Manhattan)

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

    def DLS(self):
        # Fill the correct path in self.path
        # self.fullPath should contain the order of visited nodes
        return self.path, self.fullPath

    def BDS(self):
        # Fill the correct path in self.path
        # self.fullPath should contain the order of visited nodes
        return self.path, self.fullPath

    def BFS(self):
        # Fill the correct path in self.path
```

Javapoint.com can help you to understand more about the mentioned algorithms …. You feel free to write your pseudocode and implement it according to your understanding.