Project 1: Search

Deadline: April 21, 2020

Artificial Intelligence Team

To work on this problem set, you will need to get the code, much like you did for the first problem. Your answers for the problem set belong in the main file project1.py.

Search

Explanation

This problem set will involve implementing several search techniques. For each type of search function you are asked to write, you will get a graph (with a list of nodes and a list of edges and a heuristic), a start node, and a goal node.

A graph is an object of type Graph, defined in search.py, that has lists .nodes and .edges and a dictionary .heuristic (you won't need to access the dictionary directly). All the graphs used by the tester are defined in graphs.py.

A node is just a string representing the name of the node.

An Edge is an object with attributes .name (a string), .node1 (a string, the node at one end of the edge), .node2 (a string, the node at the other end of the edge), and .length (a number).

You will need the following methods:

- graph.get_connected_nodes(node): Given a node name, return a list of all node names that are connected to the specified node directly by an edge.
- graph.get_edge(node1, node2): Given two node names, return the edge that connects those nodes, or None if there is no such edge.

- graph.get_heuristic(start, goal): Given the name of a starting node in the graph and the name of a goal node, return the heuristic value from that start to that goal. If that heuristic value wasn't supplied when creating the graph, then return 0.

You might also want to use the following methods:

- graph.are_connected(node1, node2): Return True if there is an edge running directly between node1 and node2; False otherwise.
- graph.is_valid_path(path): Given 'path' as an ordered list of node names, return True if there is an edge between every two adjacent nodes in the list, False otherwise.

The Agenda

Different search techniques explore nodes in different orders, and we will keep track of the nodes remaining to explore in a list we will call **the agenda** (in class we called this the queue). Some techniques will add paths to the top of the agenda, treating it like a stack, while others will add to the back of the agenda, treating it like a queue. Some agendas are organized by heuristic value, others are ordered by path distance, and others by depth in the search tree. Your job will be to show your knowledge of search techniques by implementing different types of search and making slight modifications to how the agenda is accessed and updated.

The Extended-Nodes Set

An extended-set, sometimes called an "extended list" or "visited set" or "closed list", consists of nodes that have been extended, and lets the algorithm avoid extending the same node multiple times, sometimes significantly speeding up search. You will be implementing types of search that use extended-sets. Note that an extended-nodes set is a set, so if, e.g., you're using a list to represent it, then be careful that a maximum of one of each node name should appear in it. Python offers other options for representing sets, which may help you avoid this issue. The main point is to check that nodes are

not in the set before you extend them, and to put nodes into the extended-set when you do choose to extend them.

Returning a Search Result

A search result is a path which should consist of a list of node names, ordered from the start node, following existing edges, to the goal node. If no path is found, the search should return an empty list, [].

Exiting the search

1. Non-optimal searches such as DFS, BFS **may** exit either:
    - when it finds a path-to-goal in the agenda
    - when a path-to-goal is first removed from the agenda.
2. Optimal searches such as A* must always exit:
    - When a path-to-goal is first removed from the agenda.

    (This is because the agenda is ordered by path length (or heuristic path length), so a path-to-goal is not necessarily the best when it is added to the agenda, but when it is removed, it is guaranteed to have the shortest path length (or heuristic path length).)
3. For the sake of consistency, you should implement all your searches to exit:
    - When a path-to-goal is first removed from the agenda.

Running Tests

For testing your written functions you need to run tester.py

A. Breadth-first Search and Depth-first Search

Your task is to implement the following functions:

- def bfs(graph, start, goal):

- def dfs(graph, start, goal):

The inputs to the functions are:

graph: The graph

start: The name of the node that you want to start traversing from

goal: The name of the node that you want to reach

When a path to the goal node has been found, return the result as explained in the section **Returning a Search Result** (above).

B. Optimal Search

The search techniques you have implemented so far have not taken into account the edge distances. Instead we were just trying to find one possible solution of many. This part of the problem set involves finding the path with the shortest distance from the start node to the goal node. The search types that guarantee optimal solutions is A*.

Since this type of problem requires knowledge of the length of a path, implement the following function that computes the length of a path:

- def path_length(graph, node_names):

The function takes in a graph and a list of node names that make up a path in that graph, and it computes the length of that path, according to the "LENGTH" values for each relevant edge. You can assume the path is valid (there are edges between each node in the graph), so you do not need to test to make sure there is actually an edge between consecutive nodes in the path. If there is only one node in the path, your function should return 0.

C. Uniform Cost Search

Now that you have a way to measure path distance, this part should be easy to complete. You might find the list procedure remove, and/or the Python 'del' keyword, useful (though not necessary). For this part, complete the following:

- def uniform_cost_search(graph, start, goal):

Note : Uniform cost search does not use an extended-set.

D. A*

In A*, the path with the least (heuristic estimate + path length) is taken from the agenda to extend. A* always uses an extended set -- make sure to use one. (Note: If the heuristic is not consistent, then using an extended-set can sometimes prevent A* from finding an optimal solution.)

- def a_star(graph, start, goal):

E. Graph Heuristics

A heuristic value gives an approximation from a node to a goal. You've learned that in order for the heuristic to be admissible, the heuristic value for every node in a graph must be less than or equal to the distance of the shortest path from the goal to that node. In order for a heuristic to be consistent, for each edge in the graph, the edge length must be greater than or equal to the absolute value of the difference between the two heuristic values of its nodes.

For this part, complete the following functions, which return True if the heuristics for the given goal are admissible or consistent, respectively, and False otherwise:

- def is_admissible(graph, goal):
- def is_consistent(graph, goal):

Notes:

1- In all graphs each edge can be traversed in both directions.

2- If you come across this heuristic value

```
heuristic={"G": {'S': 11,
                 'A': 9,
                 'B': 6,
                 'C': 12,
                 'D': 8,
                 'E': 15,
                 'F': 1,
                 'H': 2},
           "H": {'S': 11,
                 'A': 9,
                 'B': 6,
                 'D': 12,
                 'E': 8,
                 'F': 15,
                 'G': 14}}
```

Please be aware this means is heuristic value from S to G is 11 or heuristic value from A to G is 9.

If Goal node is G and you see other values in heuristic dictionary. Ignore them