Problem Set 2

To work on this problem set, you will need to get the code, much like you did for the first two problem sets.

Your answers for the problem set belong in the main file `lab2.py`.

Search

Explanation

This section is an explanation of the system you'll be working with. There aren't any problems to solve. Read it carefully anyway.

We've learned a lot about different types of search in lecture the last several weeks. This problem set will involve implementing several search techniques. For each type of search 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 a class, defined in `search.py` that has lists `.nodes` and `.edges` and a dictionary `.heuristic`. Nodes are just string names, but edges are dictionaries that contain each edge's `"NAME"`, `"LENGTH"`, and two endpoints, specified as `"NODE1"` and `"NODE2"`.

The heuristic is a dictionary, for each possible goal node, mapping each possible start node to a heuristic value.

An example graph would be made like this:

```
Graph(edges=[ { 'NAME': 'e1', 'LENGTH': 10, 'NODE1': 'Forbidden 3rd Floor
Corridor', 'NODE2': 'Common Room' },
              { 'NAME': 'e2', 'LENGTH': 4, 'NODE1': 'Common Room', 'NODE2':
'Kitchens' } ],
      heuristic={'Common Room':{ 'Forbidden 3rd Floor Corridor': 10,
                                 'Common Room': 15,
                                 'Kitchens': 20 },
                 'Kitchens':{'Forbidden 3rd Floor Corridor': 7,
                             'Common Room': 8}
```

In this graph representation, there are three nodes (Forbidden 3rd Floor Corridor, Common Room, and Kitchens), two edges, and heuristic values specified for getting to the Common Room or getting to the Kitchens. One of the edges connects the Forbidden 3rd Floor Corridor with the Common Room, and the other connects the Common Room with the Kitchens.

The representation for an entire graph, described above, is mainly used in the tester. Your search procedures will receive the graph object and you can access the data within the graph by using the procedures defined in `search.py`, and summarized below:

Helper functions

These functions will help you work with the graph representation:

- `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.are_connected(node1, node2)`: Return True iff 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 iff there is an edge between every two adjacent nodes in the list, False otherwise
- `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.

In addition, you're expected to know how to access elements in lists and dictionaries at this point. For some portions of this lab, you may want to use lists like either stacks or queues, as documented at <http://docs.python.org/tut/node7.html>.

You also may need to sort Python lists. Python has built-in sorting functionality, documented at <http://wiki.python.org/moin/HowTo/Sorting>. If you read that document, recall that Solaris-Athena* computers (the purple Athena* computers, not the black ones) still have an older version of Python prior to v2.4 installed.

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**. 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 while others are ordered by path distance. 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.

Extending a path in the agenda

In this problem set, a path consists of a list of node names. When it comes time to extend a new path, a path is selected from the agenda. The last node in the path is identified as the node to be extended. The nodes that connect to the extended node, the adjacent nodes, are the possible extensions to the path. Of the possible extensions, the following nodes are NOT added:

- nodes that already appear in the path.
- nodes that have already been extended (if an extended-nodes set is being used.)

As an example, if node *A* is connected to nodes *S*, *B*, and *C*, then the path ['S', 'A'] is extended to the new paths ['S', 'A', 'B'] and ['S', 'A', 'C'].

The paths you create should be new objects. If you try to extend a path by *modifying* (or "mutating") the existing path, for example by using list `.append()`, you will probably find yourself in a world of hurt.

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.

Exiting the search

Non-optimal searches such as DFS, BFS, Hill-Climbing and Beam **may** exit either:

- when it finds a path-to-goal in the agenda
- when a path-to-goal is first removed from the agenda.

Optimal searches such as branch and bound and A* **must** always exit:

- When a path-to-goal is first removed from the agenda.

For the sake of consistency, you should implement all your searches to exit:

- **When a path-to-goal is first removed from the agenda.**

Multiple choice

This section contains the first graded questions for the problem set. The questions are located in lab2.py and let you check your knowledge of different types of search. You should, of course, try to answer them before checking the answers in the tester.

Basic Search

The first two types of search to implement are breadth-first search and depth-first search. When necessary, use backtracking for the search.

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 input to the functions are:

- `graph`: The graph
- `start`: The name of the node that you want to start traversing from
- `end`: 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).

Hill Climbing

Hill climbing is very similar to depth first search. There is only a slight modification to the ordering of paths that are added to the agenda. For this part, implement the following function:

```
def hill_climbing(graph, start, goal):
```

The hill-climbing procedure you define here *should* use backtracking, for consistency with the other methods, even though hill-climbing typically is not implemented with backtracking.

Beam Search

Beam search is very similar to breadth first search. There is modification to the ordering of paths in the agenda. The agenda at any time can have up to **k** paths of length **n**; **k** is also known as the **beam width**. **n** correspond to the level of the search graph. You will need to sort your paths by the graph heuristic to ensure that only the top k paths at each level are in your agenda. You may want to use an array or dictionary to keep track of paths of different lengths.

For this part, implement the following function:

```
def beam_search(graph, start, goal, beam_size):
```

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 are branch and bound and 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.

Branch and Bound

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 branch_and_bound(graph, start, goal):
```

A*

You're almost there! You've used heuristic estimates to speed up search and edge lengths to compute optimal paths. Let's combine the two ideas now to get the A* search method. In A*, the path with the least (heuristic estimate + path length) is taken from the queue to extend. A* always uses an extended list -- make sure to use one.

```
def a_star(graph, start, goal):
```

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 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.

In lecture and tutorials, you've seen examples of graphs that have admissible heuristics that were not consistent. Have you seen graphs with consistent heuristics that were not admissible? Why is this so? For this part, complete the following functions, which return True iff the heuristics for the given goal are admissible or consistent, respectively, and False otherwise:

```
def is_admissible(graph, goal):

def is_consistent(graph, goal):
```

Survey

Please answer these questions at the bottom of your `lab2.py` file:

- How many hours did this problem set take?
- Which parts of this problem set, if any, did you find interesting?
- Which parts of this problem set, if any, did you find boring or tedious?

(We'd ask which parts you find confusing, but if you're confused you should really ask a TA.)

Checking your submission

When you're finished with your lab, don't forget to submit your code and run the online unit tests!

6.034 Artificial Intelligence

Fall 2010