6.034 Artificial Intelligence, Fall 2006
Prof. Patrick H. Winston

Problem Set 2

This problem set is due Wednesday, October 4th at 11:59 PM. If you have questions about it, ask the TA email list. Your response will probably come from a TA.

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

> The code can be downloaded from the assignments section.

Your answers for the problem set belong in the main file `ps2.scm`.

Some important notes on problem sets

- DrScheme lets you add certain graphical elements, like boxes drawn around code to comment them out, to your code. If you use these, then DrScheme will save the code (by default) in a wacky non-text format that GraderBot doesn't understand. If you use these features, you'll have to use the following menu item to save your code:

  File > Save Other > Save definitions as text...

- There is now a problem set grading policy that tells you how to submit a problem set late, and why your code needs to run without human intervention.
- You may still want to refer to **Using DrScheme** in the assignments section.

1. Search

1.1. 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 (consisting of a list of nodes and a list of edges), a start node, and a goal node.

A graph is a Scheme-like expression that contains the keywords NODES and EDGES. Each node expression consists of a description, called the *node ID*, and a heuristic value that guesses its distance to a goal node. In most places, you will represent the node using simply the node ID.

Each edge expression contains an *edge ID*, a length, and two endpoints (specified as node IDs).

An example of a graph representation looks like this:

```
(define graph
  (list '(NODES
    ("Forbidden 3rd Floor Corridor" 10)
    ("Common Room" 15)
    ("Kitchens" 20))
  '(EDGES
    (e1 10 "Forbidden 3rd Floor Corridor" "Common Room")
    (e2 4 "Common Room" "Kitchens"))))
```

In this graph representation, there are three nodes (Forbidden 3rd Floor Corridor, Common Room, and Kitchens), followed by their respective heuristic values. There are two edges in this graph. One edge 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 nodes and edges as separate parameters. You can access the data within the node and edge expressions by using the procedures defined in `search.scm`.

1.1.1. Helper functions

These functions will help you work with the graph representation:

- `(get-nodes graph)`: Extract the NODES list of a graph.
- `(get-edges graph)`: Extract the EDGES list of a graph.
- `(get-last lst)`: Returns the last item in a list.
- `(get-heuristic node-id nodes)`: Look up the heuristic value of a node by its id.
- `(get-connected-nodes node-id edges)`: Find all nodes that are connected to the given node by an edge.
- `(node-id expr)`: get the node id from a node expression.
- `(node-heuristic expr)`: get the heuristic value from a node expression.
- `(get-edge node-id1 node-id2 edges)`: look for an edge connecting the two given nodes in a list of edges, and return it as an edge expression.
- `(edge-distance edge)`: given an edge expression, get the weight (distance) of that edge.
- `(edge-nodes edge)`: given an edge expression, get a list containing the two nodes it connects.

1.2. The Queue

Different search techniques utilize the queue differently. Some add paths to the beginning of the queue while others add paths to the end of the queue. Some queues 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 queue is accessed and updated.

1.2.1. Extending a path in the queue

In this problem set, a path consists of a list of node id values. When it comes time to extend a new path, a path is selected from the queue. The last node in the path is identified as the node to be extended. All nodes that connect to the extended node are identified as well. These 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 list is being used.)

1.2.2. Adding paths to the queue

Different search techniques require managing the queue in different ways.

Note: (updated Monday, Oct. 2) When you add multiple paths to the queue (that is, when you extend a node), you should add them in the order you got the nodes back from `get-connected-nodes`. Almost everyone is doing this anyway, but we needed to clarify this.

1.3. The Extended List

The extended list is often used to speed up searches. You will be implementing some types of search that use extended lists and others that don't. Whether or not you will use an extended list in the search, keep track of each node that you extend and add it to the end of the extended list. The first node that you extend will be the start node and this should be at the beginning of the extended list.

1.3.1. useExtendedList?

The boolean variable useExtendedList? will determine whether or not you will use an extended list in the search. Remember that an extended list prevents you from extending a node more than one time. So if useExtendedList? is true, a maximum of one of each node id should appear in the extended list. Keeping track of the nodes that you extend is a useful estimate of the time it took to produce a search result. Therefore, also keep track of nodes that you extend even if useExtendedList? is false. In this case, you may extend a node more than once, and it will appear more than one time in the extended list.

1.4. Returning a Search Result

In order to accurately test your search implementation, we want to see which nodes you've extended and the solution path that your search found. You know you are done when the next node you try to extend is the goal node. Add the goal node to the extended list and return the following result:

```
(create-search-solution extended solution-path)
```

These are the parts of the search result:

- extended - a list of node id values for nodes that you have extended. (Note that this list of node ids should begin with the start node and end with the goal node.)
- solution-path - a list of the node ids that make up the final solution path. This path should start with the id value for the start node and end with the id value for the goal node.

1.5. Multiple choice

This section contains the first graded questions for the problem set. The questions are located in ps2.scm and test your knowledge of different types of search.

2. 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.

2.1. Breadth-first Search and Depth-first Search

Your task is to implement the following functions:

```
(define (bfs nodes edges useExtendedList? extended queue start goal)
(define (dfs nodes edges useExtendedList? extended queue start goal)
```

The input to the functions are:

- nodes - a list of the nodes in the graph
- edges - a list of the edges in the graph
- useExtendedList? - a boolean value indicating whether or not you should use an extended list for the search
- extended - a list of the node id values for nodes that have been extended. The first node to be extended should be at the front of the list. All additional nodes that are extended will be added to the end of the list.

- queue - a list of all paths on the queue
- start - the node id value for the start node
- goal - the node id value for the goal node

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

2.2. 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 queue. You might find the function `sort-by-heuristic` in search.scm very useful. For this part, implement the following function:

```
(define (hill-climbing nodes edges useExtendedList? extended queue
start goal)
```

3. 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*. In your solution for this problem, the order that you place paths in the queue does not matter. However, we will still check to make sure that the correct paths are in the queue.

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

```
(define (path-length node-ids edges)
```

The function takes in a list of node-ids that make up a path and the edges in a graph and computes the length of a path. 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.

3.1. 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` helpful. Remember, it does not matter what order you place paths into the queue. For this part, complete the following:

```
(define (branch-and-bound nodes edges useExtendedList? extended queue
start goal)
```

3.2. 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, so that variable has been removed. Return results as before.

```
(define (a-star nodes edges extended queue start goal)
```

4. 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? (This should give you an idea of how to program this problem.) For this part, complete the following function, which returns:

- #t if a graph has heuristics that are admissible and consistent
- #f otherwise

Note: You can assume for this problem that the heuristic value of the goal node is 0. Is your solution guaranteed to be correct if it is not?

```
(define (isAdmissibleAndConsistent? graph)
  'fill-me-in)
```

5. Survey

Please answer these questions at the bottom of your `ps2.scm` 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.)

When you're done, **run the tester** and submit your .scm files to your `6.034-psets/ps2` directory on Athena. If the tester dies with an error when it's run on your code, your submission will not count.

6. Errata

If you find what you think is an error in the problem set, tell the TAs about it.

6.1. Monday, Oct 2

A couple of issues with the problem set have been resolved, so you should download a new copy of `search.scm` and `tester.scm`.

6.1.1. Order of paths on the queue

We forgot to mention that the tester expects you to add paths to the queue in the order that the nodes come from get-connected-nodes. This doesn't affect many people, because almost everyone is doing that anyway.

If you add the paths in the opposite order, you can download the new `tester.scm` which allows the alternate solutions you get that way.

6.1.2. Multiple choice

In the question about hill-climbing search, assume that the search does not backtrack and does not follow edges that make the heuristic worse, and that a "solution" requires finding a path to the node with the lowest (best) heuristic. (These assumptions are commonly true in hill-climbing.)

6.1.3. Bug in `connected?`

`connected?` didn't work; it threw an error if the nodes were not connected. It works in the new `search.scm`.