Richard Robinson
Artificial Intelligence
Project 1 Part 1

Items included:

node.py: node interactions and some state information
search.py: contains the search functions, and graph search
searchAgent.py: contains states, successors, costs, and is where the search functions are inserted into.

Statement of Problem:

Implement graph search and
1) Depth-first search
2) Breadth-first search
3) Uniform Cost search
4) A* search

For the simple Pacman search problem with just one item to pick up as a goal state

Implementation:

Created a Node class within node.py. This contains the state of the node (has x,y coordinates), parent (for finding the path details), action, and cumulative cost of the paths. Expand returns a list of successors. This Node class is necessary for the implementations of our search solutions.

Within the search.py class there is a graph search. This takes a problem (what is being solved) and a fringe (can be many different data structures). The fringe's purpose to find potential solution paths to the problems. This graph search works by starting in the start state indicated by the problem, pushing the first node onto the data structure. While there are paths to explore in the fringe they are expanded upon, and exhausted until either the goal is reached or there are no paths left. Care is made to prevent loops by indicating visited nodes.

By formatting the graph search this way it can be applied to various differing search applications easily. This allows for easy addition of features for what is desired.

The searching methods applied were all very short and concise, using components from the util.py file.

Depth first: passes the Stack data structure to the graph search

```
return graphSearch(problem, util.Stack())
```

Breadth first: passes the Queue data structure to the graph search

```
return graphSearch(problem, util.Queue())
```

Uniform cost: passes a Priority queue with a lambda function using the cumulative path cost to the graph search

```
return graphSearch(problem, util.PriorityQueueWithFunction( lambda node:
node.path_cost ))
```

aStar search: passes Priority queue with lambda function using cumulative path and the heuristic found

from the nodes x,y state and the given problem placed into the heuristic function.

```
return graphSearch(problem, util.PriorityQueueWithFunction(
    lambda node: node.path_cost + heuristic(node.state, problem)))
```

Conclusions:

This project was a wonderful project. It was kind of confusing rummaging through various parts of the code and tracing one component which expands upon another to another and another and another. I personally wanted to discard what was written and make my own parser for levels, finding successors, and states initially. It took me some time to find all the components I needed and put them together, but once I did there was a certain beauty to it. Many components expanded upon previous in small, but meaningful ways. This allows for a ton of reusability of what is desired as well as being able to build upon select components.

The coding organization in this project was significantly different than my own and it was a good thing being exposed to different coding styles. The graph search, problem, and agent components were particularly interesting. One day I hope to being capable of making solutions without some prior backbone that elegant. I have never seen anything quite like this before in any project I have worked on, or even considered some the solutions guided by this project.

I learned a lot about python and some great coding components through this project.