CSC 665 - Pacman Search

This project implements different search algorithms for Pacman agent to find its path to the goal.

Overview

- Link to the project's original specs
- By utilizing a general graph search and providing it the correspoding search strategy, different search algorithms will be able to reuse that logic to find the solution.
- The search logic can be found in `search/search.py`
- The Node class can be found in `search/node.py`

How to run

- Download or git clone this repo if not already
- In terminal, go to directory:
    - `cd path/to/CSC665-pacman-search/search`
- To test Depth First Search (DFS):
    - `python pacman.py -l tinyMaze -p SearchAgent`
- To test Breadth First Search (BFS):
    - `python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs`
- To test Uniform Cost Search (UCS):
    - `python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs`
- To test A* search:
    - `python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic`

Implementations

General graph search

```
def graphSearch(problem, fringe):
```

- Given the search problem and fringe as arguments, the general graph search function will search through the successors of a problem to find a goal. The argument fringe should be an empty queue. Also, it will return a sequence of actions (directions) from the start to goal state.

- The function starts with the start state of the problem and pushes it to the fringe (partial plan for the search strategy to explore next).
- By looping through the fringe and check until it's empty, or when there's no more candidate to explore, the function will do either 2 things:
    - Checking if the current state is already the goal state. If so, return the list of actions that gets to this state as a result.
    - Choose the corresponding state and node to expand corresponding to the strategy. In addition, it makes sure to check if the current and the to-be-expanded states are not yet added to the list of visited states in order to avoid checking the same states again (which will result in infinite loop!)

Search strategies

1. DFS

   ```
   def depthFirstSearch(problem):
   ```

   - Search the deepest nodes in the search tree first, using Stack as the strategy.

2. BFS

   ```
   def depthFirstSearch(problem):
   ```

   - Search the shallowest nodes in the search tree first, using Queue as the strategy.

3. UCS

   ```
   def depthFirstSearch(problem):
   ```

   - Search the node of least total cost first, using PriorityQueueWithFunction as the strategy. The lambda function that helps achieve this strategy takes in consideration of the current node's path cost.

4. A* Search

   ```
   def depthFirstSearch(problem):
   ```

   - Search the node that has the lowest combined cost and heuristic first, using PriorityQueueWithFunction as the strategy. The lambda function that helps achieve this strategy takes in consideration of the current node's path cost.