Assignment 3

AI 2016 – Minor Robotica

- **It will be graded.**
- **Execute in groups of one or two persons.**
- **Create one PDF per group with the answers to the questions on paper and include a short but clear explanation of what you programmed.**
- **You will only alter the files search.py and searchAgents.py**
- **Combine the PDF + your altered search.py code file in one ZIP file with the name:**
  AI2015 USC ASTAR lastname1-studentnr1 lastname2-studentnr2.zip
  so e.g.:
  AI2015 USC ASTAR deKok-500223133 Stolk-500131323.zip
- **Upload the zip-file to the DLWO-Dropbox of this course.**
  Other naming conventions will not be graded.

- **More info on the Python Pacman problem:**
  http://inst.eecs.berkeley.edu/~cs188/pacman/search.html

Assignments:

This assignment continues on the previous (BFS/DFS) assignment's code base. Please extend your search.py file with the required additions and also update the `searchAgents.py` file.

Please make use of the functionalities and structures provided in `util.py`
Here you can find FIFOs, priority queues etc.

Hint: you can benefit from the dictionary capabilities of Python, for example in maintaining dictionaries coupling a node to its parent node and keeping track of the action that brought you here.

Assignment 3

Uniform Cost Search

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider `mediumDottedMaze` and `mediumScaryMaze`.

By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behaviour in response.

Question 3a (2 points)

- **Implement the uniform-cost graph search algorithm in the `uniformCostSearch` function in `search.py`.**
- **Describe your design choices and approach of your implementation in a brief report.**

We encourage you to look through `util.py` for some data structures that may be useful in your implementation. You should now observe successful behaviour in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Note: You should get very low and very high path costs for the `StayEastSearchAgent` and `StayWestSearchAgent` respectively, due to their exponential cost functions (see `searchAgents.py` for details).

A* search

Question 3b i (3 points)

- **Implement A* graph search in the empty function `aStarSearch` in `search.py`. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The `nullHeuristic` heuristic function in `search.py` is a trivial example and always gives a heuristic value of `0`.**
- **Describe your design choices and approach of your implementation in a brief report.**

You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic. This is implemented already as `manhattanHeuristic` in `searchAgents.py`:

```
python pacman.py -l bigMaze -z .5 -p SearchAgent –a
fn=astar,heuristic=manhattanHeuristic
```

You should see that A* finds the optimal solution slightly faster than uniform cost search (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly). What happens on `openMaze` for the various search strategies?

Question 3b ii (1 point)

- **Describe your findings of the implementation. How does the selected heuristic affect the search paths?**
- **What is the behaviour of A* when using the `nullHeuristic`? Does this behaviour match (or comes close to) any other search algorithm?**

Finding All the Corners

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.

In *corner mazes*, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like `tinyCorners`, the shortest path does not always go to the closest food first! *Hint*: the shortest path through `tinyCorners` takes 28 steps.

Note: Make sure to complete the BFS search of week 2 before working on this question, because this question builds upon your answer of last week.

Question 3c (2 points)

Implement the `CornersProblem` search problem in `searchAgents.py`. Because the goal of this search problem is different, you will need to choose a newly designed state representation that encodes all the information necessary to detect whether all four corners have been reached (in `searchAgents.py`).

In this assignment, your search agent should solve:

```
python pacman.py -l tinyCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem
```

To receive full credit, you need to define an abstract state representation that *does not* encode irrelevant information (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pacman `GameState` as a search state. Your code will be very, very slow if you do (and also wrong).

Hint: The only parts of the game state you need to reference in your implementation are the starting Pacman position and the location of the four corners.

Our implementation of `breadthFirstSearch` expands just under 2000 search nodes on `mediumCorners`. However, heuristics (used with A* search) can reduce the amount of searching required.

Finding All the Corners With Heuristic

Question 3d (2p bonus)

Note: Make sure to complete Question 3b i before working on Question 3d, because this question builds upon the answers of Question 3b i.

Implement a non-trivial (no null heuristic), consistent heuristic for the `CornersProblem` in `cornersHeuristic`.

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

Note: `AStarCornersAgent` is a shortcut for

```
-p SearchAgent -a
fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be *admissible*, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be *consistent*, it must additionally hold that if an action has cost *c*, then taking that action can only cause a drop in heuristic of at most *c*.

Remember that admissibility isn't enough to guarantee correctness in graph search -- you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore, it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible

heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in in f-value. Moreover, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).