# Department of Computing

# CS370: Artificial Intelligence

# Class: BSCS-5AB

# Lab 3: Solving Problem by Searching

# Date: 6-10-2016

# Time: 10am-1pm & 2pm-5pm

# Instructor: Dr. Omar Arif

# Lab 3: Solving Problems by Searching

**Implement Questions 4, 5 and 6 from** [**http://ai.berkeley.edu/search.html**](http://ai.berkeley.edu/search.html) **.**

**Note:** You can work in groups of 2 (same group member as in lab 2) but not more than 2.

**Deliverables:** Submit search.py and searchAgents.py (**each member should submit the files on LMS)**

**Deadline:** Before the next lab.

### Question 4 (3 points): A\* search

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.

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 (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 5 (3 points): 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 Question 2 before working on Question 5, because Question 5 builds upon your answer for Question 2.*

Implement the CornersProblem search problem in searchAgents.py. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Now, 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.

### Question 6 (3 points): Corners Problem: Heuristic

*Note: Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.*

Implement a non-trivial, 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).

**Grading:** Your heuristic must be a non-trivial non-negative consistent heuristic to receive any points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll be graded:

|  |  |
| --- | --- |
| **Number of nodes expanded** | **Grade** |
| more than 2000 | 0/3 |
| at most 2000 | 1/3 |
| at most 1600 | 2/3 |
| at most 1200 | 3/3 |

Remember: If your heuristic is inconsistent, you will receive no credit, so be careful!