CS 182 FALL 2022, PROBLEM SET 1

Due: September 27, 2022 11:59pm

This problem set covers Lectures 2, 3, and 4. The topics include Uninformed Search, Informed Search, and Motion Planning.

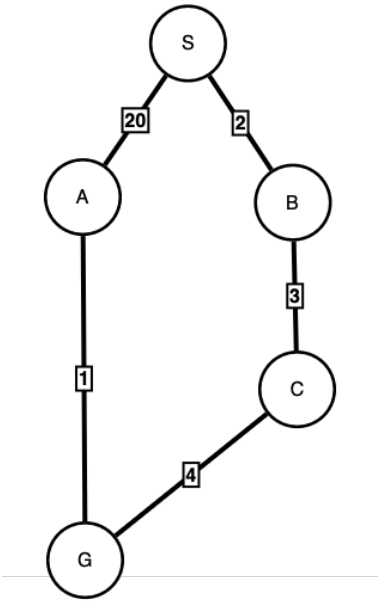
1. *Uninformed and Informed Search.* (7 points)

(1) For each of the following, explain why it's true or provide a counterexample.

(a) (1 point) Breadth-first search is a special case of uniform-cost search.
If uniform-cost search is applied on a problem where the action-costs from state to state are all the same, then the path-cost function (that dictates how nodes are expanded) simply returns the depth at which the node is found at. Hence, nodes are expanded based on the order in which they were first encountered, which is exactly a Breadth-first search.

(b) (1 point) Uniform-cost search is a special case of A* search.
If we let $h(n) = 0$ in the evaluation function $f(n) = g(n)+h(n)$ then $f(n) = g(n)$. $g(n)$ is the path-cost from the inital state to n, so nodes are expanded based on path-cost alone. which is how Uniform-cost search expands nodes.

(c) (1 point) Depth-first search always expands at least as many nodes as A* search with an admissible heuristic.
Consider the following search space with initial state S and goal state G:



Also, define $h(A) = 1, h(B) = 8, h(C) = 6$, and $h(G) = 0$. A depth-first search that expands left-most modes first only expands S and A before encountering G and returning. A* search, on the other hand, expands S, B, and C before expanding G and returning. Hence, depth-first search does not always expand at least as many nodes as A*.

(d) (1 point) Breadth-first search is complete even if zero step costs are allowed.
This is true because Breadth-first search only considers the depth in the tree at which nodes were encountered when choosing what node to expand next, so every node at every level of the tree is still expanded, making this a complete search.

(e) (1 point) Assume that a rook can move on a chessboard any number of squares in a straight line, vertically or horizontally, but cannot jump over other pieces. Manhattan distance is an admissible heuristic for the problem of moving the rook from square A to square B in the smallest number of moves.
Suppose a rook wanted to move from square A to square B and that square A and square B are are on opposite sides of the same column, with no pieces in between. Then, the least number of moves to get the rook from A to B is 1 because the rook can move any number of moves in a straight line, which A and B are on, but the manhattan distance between these two squares is the width of the board - 1. This heuristic overestimates the cost, so it is not admissible.

(2) (2 points) Describe a state space in which iterative deepening search performs much worse than depth-first search (for example, $\Theta(n^2)$ vs. $\Theta(n)$).

2. *Gridworld.* (13 points) Consider the unbounded version of the regular 2D grid. The start state is at the origin, $(0, 0)$, and the goal state is at (u, v). You may assume every edge is bidirectional and has unit cost. You should provide justifications for your answers. Below is an example image showing what this arrangement would look like if $(u, v) = (8, 5)$. Be sure the consider the generic coordinates (u, v) for the questions below.

(1) (1 point) What is the branching factor b in this state space? In other words, for each state, how many neighbors does that state have?

(2) (2 points) Is $h(x, y) = |x - u| + |y - v|$ an admissible heuristic for a state at (x, y)? Is it a consistent heuristic?

(3) (2 points) What is the maximum number of nodes expanded by A* Graph Search using h?[1]

(4) (1 point) Does h remain admissible if some links are removed?

(5) (1 point) Does h remain admissible if some links are added between nonadjacent states?

(6) (6 points) Take $h_2(x, y) = \sqrt{h(x, y)}$ with the original search space. Is h_2 admissible? Is it consistent?

[1] "maximum" refers to worst-case tie-breaking, i.e. when choosing among states with the same desirability as the goal state, assume the goal state is chosen last.

3. *Informed Search.* (20 points) Let $h^*(x)$ denote the cost function that returns the shortest distance between a state x and the nearest goal state t. Let $h(\cdot)$ be a heuristic that overestimates $h^*(\cdot)$ by at most ϵ, meaning that for all states x, $h(x) \leq h^*(x) + \epsilon$. Assume that $h(\cdot)$ assigns 0 to any goal state. Prove that A* *tree search* using h finds a path to the nearest goal state t whose cost is at most ϵ more than an optimal path to the nearest goal state. Formally, if s is the start state, t is the goal state returned by A^* search, and $g(x)$ denotes the lowest cost path from s to x along the explored graph, then show that $g(t) \leq h^*(s) + \epsilon$.

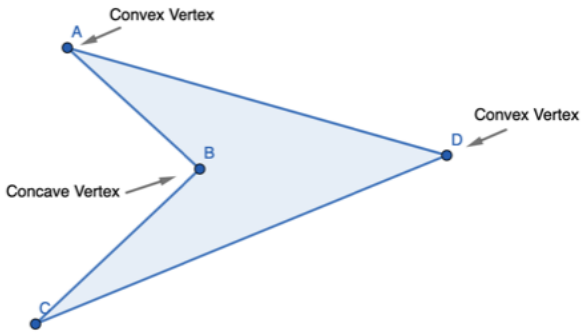


FIGURE 1. Examples of Convex and Concave Vertices

4. *Motion Planning.* (20 points) Let S be a set of disjoint obstacles (simple polygons) in the plane, and let n denote the total number of their edges. Assume that we have a point robot moving on the plane that can touch the edges of the obstacles. (That is, we treat the obstacles as open sets.) The robot starts at the p_{start} position and must get to the p_{goal} position using the shortest collision-free path. In class, we proved that any shortest path between p_{start} and p_{goal} is a polygonal path whose inner vertices are the vertices of the obstacles. You may use this result in your answer to the following questions.

We now consider two different kinds of vertices on obstacles.

- A vertex whose internal angle formed by its two edges is less than 180° is called *convex*.
- A vertex whose internal angle formed by its two edges is at least 180° is called *concave*.

See Fig. 1.

Prove that a shortest path from p_{start} to p_{goal} is a polygonal path where each inner vertex (if any) corresponds to a *convex* vertex of some obstacle.

Hint: Think about what will happen if there is a shortest path from p_{start} to p_{goal} where one inner vertex is a polygon's concave vertex.

5. *Programming.* (35 points) Your friend has a hobby of collecting gnomes and giving them mini-houses within her garden. Moreover, she has a robot that she intends to use to visit each of the gnomes each week to present them with fresh flowers. Your friend wants you to help her program the robot to figure out a fast route to get to every gnome residence. The garden is a 2D rectangular grid divided into discrete squares. Each square is either free, a vegetable (i.e. an obstacle), or a gnome residence. The route must visit every gnome residence at least once. A goal node is any node where the robot has already visited every residence at least once. The size of the grid will vary depending on which part you are working on, but you are guaranteed that there are no more than 5 residences. Your task is to tell the gnome which set of actions it should take to accomplish its goal.

As a visual reference, here is an ASCII rendition of a neighborhood map with 2 residences (labeled R) and some walls (labeled O):

S.O.O
..O.R
O.O.O
.R..R

This will be represented in the code as a list of lists, with blank spaces being 0, obstacles being -1 and residences being 1. The exact structure for the map above is

```
[[0 ,0 , -1 ,0 , -1] ,
[0 ,0 , -1 ,0 ,1] ,
[-1 ,0 , -1 ,0 , -1] ,
[0 ,1 ,0 ,0 ,1]]
```

The start position is given as a tuple of row-column coordinates; the start position is $(0, 0)$ for this grid. The cost is uniformly 1 for every attempted move. The robot can only move in the 4 cardinal directions (up, right, down, left), and cannot move off of the grid. Trying to move into obstacles or off the grid results in staying put.

A node should encompass the current position of the robot, as well as which residences have been visited and which have not. Two nodes with the same position, but different visited residences are different nodes. Note, when adding successors of a search node onto your priority queue, make sure to add them in the following order: up, right, down, left, so that they will be explored in that order. Tie-breaking for A^* should be done by favoring nodes added earlier. Tie-breaking for BFS is the same; however tie-breaking for DFS, if using a stack, should result in adding the successors in the reverse order such that the successors

will be visited in the correct order.

For example inputs and outputs, see `pset1_sample_test_case1.txt`. The first line contains the number of rows r and columns c. The next r lines contain the map information. The last line contains the coordinates for the start state (r_0, c_0). You may also use any of the python standard libraries as you see fit.

(1) (12 points, graded jointly with 5.2) *Defining the search problem.* The goal of this part of the problem is to describe our problem as just an instance of the graph search problem. How do we do this (i.e. what are the nodes and edges of the graph that we can search to find a solution)? In addition to answering this in your writeup, fill in the methods of `GridworldSearchProblem`, which inherits from the abstract class `SearchProblem`.

(2) (4 points, graded jointly with 5.1) *DFS.* For the remaining parts, your implementation of the search strategies should not depend on your implementation of `GridWorldSearchProblem` and should instead work only by calling the methods of `SearchProblem` (i.e. treating the search problem as an abstract class). Implement depth-first search in the function `depthFirstSearch(problem)`. Note that you should only do cycle detection and not re-expand any nodes along a single path; your algorithm should allow for the same node to be visited by different paths. Cycle detection preserves the low memory requirements of DFS.

(3) (4 points) *BFS.* Implement breadth-first search in the `breadthFirstSearch(problem)` function. Note that you should globally keep track of visited nodes and not expand any node more than once.

(4) (6 points, graded jointly with 5.5) *A*.* Implement A* search in the `aStarSearch(problem, heuristic)` function. The `heapq` module is one possible way to implement your priority queue. Alternatively, feel free to use the data structures we have provided. The `heuristic` parameter should be a function that returns a numeric value. The autograder will test your A* function with the `nullHeuristic` that always returns 0. Note that this trivial heuristic is consistent. For debugging, we recommend using the `nullHeuristic` since with this, the `aStarSearch` should do the same thing as the `breadthFirstSearch`.

(5) (3 points, graded jointly with 5.4) *Simple Heuristic.* Now you will implement heuristics, which may depend on how you implemented your states. Your simple heuristic should return the number of residences that haven't been visited. Implement this in the function `simpleHeuristic(state, problem)`. The autograder will test this by passing your heuristic as the `heuristic` parameter in your A* function.

(6) (6 points, 5 bonus points) *Custom Heuristic Challenge.* The simple heuristic is quick to compute, but not very effective in guiding A*. Try to come up with your own heuristic that reduces the number of nodes expanded. Your heuristic should be consistent. In your writeup, you should prove that your custom heuristic is consistent. Implement this in the `customHeuristic(state, problem)` function. The autograder will test this by passing your `customHeuristic(state, problem)` as the `heuristic` parameter in your A* function. You will get full marks by expanding fewer nodes than the simple heuristic. Moreover, we will have a competition with a leaderboard on Gradescope to see who has the best heuristic! Concretely, students will be ranked on the leaderboard to see who expands the least number of nodes over our hidden test cases. We also have a column in the leaderboard for overall execution time as well. Under "leaderboard name", feel free to either use your name or a pseudonym. The top 5 submissions on the leaderboard according to least number of nodes expanded will receive 5 bonus points on this problem set (as well as bragging rights for best heuristic)!

6. Collaboration, Calibration and References. (5 points)

(1) With whom did you work on this problem set? What (if any) references and/or resources did you use beyond the course lecture slides and textbook?

(2) (5 points) Approximately how long did it take you to complete this problem set? Please complete this brief survey worth 5 points, graded for completion.