CS 188 Summer 2021 — Introduction to Artificial Intelligence

Exam Prep 2A

Q1. SpongeBob and Pacman (Search Formulation)

Recall that in Midterm 1, Pacman bought a car, was speeding in Pac-City, and the SpongeBob wasn't able to catch him. Now Pacman has run out of gas, his car has stopped, and he is currently hiding out at an undisclosed location.

In this problem, you are on the SpongeBob side, tryin' to catch Pacman!

There are still p SpongeBob cars in the Pac-city of dimension m by n. In this problem, **all SpongeBob cars can move, with two distinct integer controls: throttle and steering, but Pacman has to stay stationary**. Once one SpongeBob car takes an action which lands him in the same grid as Pacman, Pacman will be arrested and the game ends.

Throttle: $t_i \in \{1, 0, -1\}$, corresponding to {Gas, Coast, Brake}. This controls the **speed** of the car by determining its acceleration. The integer chosen here will be added to his velocity for the next state. For example, if a SpongeBob car is currently driving at 5 grid/s and chooses Gas (1) he will be traveling at 6 grid/s in the next turn.
Steering: $s_i \in \{1, 0, -1\}$, corresponding to {Turn Left, Go Straight, Turn Right}. This controls the **direction** of the car. For example, if a SpongeBob car is facing North and chooses Turn Left, it will be facing West in the next turn.

(a) Suppose you can **only control 1 SpongeBob car**, and have absolutely no information about the remainder of $p-1$ SpongeBob cars, or where Pacman stopped to hide. Also, the SpongeBob cars can travel up to 6 grid/s so $0 \le v \le 6$ at all times.

(i) What is the **tightest upper bound** on the size of state space, if your goal is to use search to plan a sequence of actions that guarantees Pacman is caught, no matter where Pacman is hiding, or what actions other SpongeBob cars take. Please note that your state space representation must be able to represent **all** states in the search space.

(ii) What is the maximum branching factor? Your answer may contain integers, m, n.

(iii) Which algorithm(s) is/are guaranteed to return a path passing through all grid locations on the grid, if one exists?

☐ Depth First Tree Search ☐ Breadth First Tree Search
☐ Depth First Graph Search ☐ Breadth First Graph Search

(iv) Is Breadth First Graph Search guaranteed to return the path with the shortest number of **time steps**, if one exists?

○ Yes ○ No

(b) Now let's suppose you can control **all** p SpongeBob cars at the same time (and know all their locations), but you still have no information about where Pacman stopped to hide

(i) Now, you still want to search a sequence of actions such that the paths of p SpongeBob car combined **pass through all** $m * n$ **grid locations**. Suppose the size of the state space in part (a) was N_1, and the size of the state space in this part is N_p. Please select the correct relationship between N_p and N_1

○ $N_p = p * N_1$ ○ $N_p = p^{N_1}$ ○ $N_p = (N_1)^p$ ○ None of the above

(ii) Suppose the maximum branching factor in part (a) was b_1, and the maximum branching factor in this part is b_p. Please select the correct relationship between b_p and b_1

○ $b_p = p * b_1$ ○ $b_p = p^{b_1}$ ○ $b_p = (b_1)^p$ ○ None of the above

Q2. Search

(a) Rubik's Search

Note: You do not need to know what a Rubik's cube is in order to solve this problem.

A Rubik's cube has about 4.3×10^{19} possible configurations, but any configuration can be solved in 20 moves or less. We pose the problem of solving a Rubik's cube as a search problem, where the states are the possible configurations, and there is an edge between two states if we can get from one state to another in a single move. Thus, we have 4.3×10^{19} states. Each edge has cost 1. Note that the state space graph does contain cycles. Since we can make 27 moves from each state, the branching factor is 27. Since any configuration can be solved in 20 moves or less, we have $h^*(n) \leq 20$.

For each of the following searches, estimate the approximate number of states expanded. Mark the option that is closest to the number of states expanded by the search. Assume that the shortest solution for our start state takes exactly 20 moves. Note that 27^{20} is much larger than 4.3×10^{19}.

(i) DFS Tree Search

Best Case: ○ 20 ○ 4.3×10^{19} ○ 27^{20} ○ ∞ (never finishes)

Worst Case: ○ 20 ○ 4.3×10^{19} ○ 27^{20} ○ ∞ (never finishes)

(ii) DFS graph search

Best Case: ○ 20 ○ 4.3×10^{19} ○ 27^{20} ○ ∞ (never finishes)

Worst Case: ○ 20 ○ 4.3×10^{19} ○ 27^{20} ○ ∞ (never finishes)

(iii) BFS tree search

Best Case: ○ 20 ○ 4.3×10^{19} ○ 27^{20} ○ ∞ (never finishes)

Worst Case: ○ 20 ○ 4.3×10^{19} ○ 27^{20} ○ ∞ (never finishes)

(iv) BFS graph search

Best Case: ○ 20 ○ 4.3×10^{19} ○ 27^{20} ○ ∞ (never finishes)

Worst Case: ○ 20 ○ 4.3×10^{19} ○ 27^{20} ○ ∞ (never finishes)

(v) A* tree search with a perfect heuristic, $h^*(n)$, Best Case

○ 20 ○ 4.3×10^{19} ○ 27^{20} ○ ∞ (never finishes)

(vi) A* tree search with a bad heuristic, $h(n) = 20 - h^*(n)$, Worst Case

○ 20 ○ 4.3×10^{19} ○ 27^{20} ○ ∞ (never finishes)

(vii) A* graph search with a perfect heuristic, $h^*(n)$, Best Case

○ 20 ○ 4.3×10^{19} ○ 27^{20} ○ ∞ (never finishes)

(viii) A* graph search with a bad heuristic, $h(n) = 20 - h^*(n)$, Worst Case

○ 20 ○ 4.3×10^{19} ○ 27^{20} ○ ∞ (never finishes)

(b) Limited A^* Graph Search

Consider a variant of A^* graph search called Limited A^* graph search. It is exactly like the normal algorithm, but instead of keeping all of the fringe, at the end of each iteration of the outer loop, the fringe is reduced to just a certain amount of the best paths. I.e. after all children have been inserted, the fringe is cut down to the a certain length. The pseudo-code for normal A^* graph search is reproduced below, the only modification being an argument W for the limit.

```
1:  function A* GRAPH SEARCH(problem,W)
2:      fringe ← an empty priority queue
3:      fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
4:      closed ← an empty set
5:      ADD INITIAL-STATE[problem] to closed
6:      loop
7:          if fringe is empty then
8:              return failure
9:          end if
10:         node ← REMOVE-FRONT(fringe)
11:         if GOAL-TEST(problem, STATE[node]) then
12:             return node
13:         end if
14:         if STATE[node] not in closed then
15:             ADD STATE[node] to closed
16:             for successor in GETSUCCESSORS(problem, STATE[node]) do
17:                 fringe ← INSERT(MAKE-SUCCESSOR-NODE(successor, node), fringe)
18:             end for
19:         end if
20:         fringe = fringe[0:W]
21:     end loop
22: end function
```

(i) For a positive W, limited A^* graph search is complete.

○ True ○ False

(ii) For a positive W, limited A^* graph search is optimal.

○ True ○ False

(iii) Provide the smallest value of W such that this algorithm is equivalent to normal A^* graph search (i.e. the addition of line 16 makes no difference to the execution of the algorithm).