Written Assignment 1: Solution

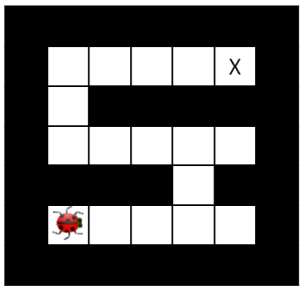
Deadline: January 20th, 2020

Instruction: You may discuss these problems with classmates, but please complete the write-ups individually. (This applies to BOTH undergraduates and graduate students.) Remember the collaboration guidelines set forth in class: you may meet to discuss problems with classmates, but you may not take any written notes (or electronic notes, or photos, etc.) away from the meeting. Your answers must be **typewritten**, except for figures or diagrams, which may be hand-drawn. Please submit your answers (pdf format only) on **Canvas**.

Q1. Hive Minds (30 points)

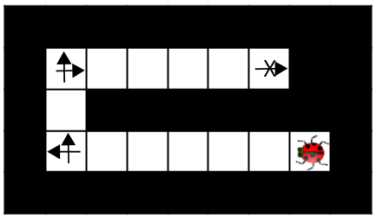
All questions in **Q1** share a common setup. You control one or more insects in a rectangular maze-like environment with dimensions $M \times N$. At each time step, an insect can either (a) move into an adjacent square if that square is currently free, or (b) stay in its current location. Each location is represented as a coordinate (x, y). Squares may be blocked by walls, but the map is known. Optimality is always in terms of time steps; all actions have cost 1 regardless of the number of insects moving or where they move. For each of the following questions, you should answer for a general instance of the problem, not simply for the example maps shown.

Q1.1. Lonely Bug. You control a single insect as shown in the example maze below, which must reach a designated target location X, also known as the hive. There are no other insects moving around. Provide a smallest correct state space representation. What is the size of the state space?



Answer. A tuple (x, y) encoding the (x) and (y) coordinates of the insect. The size of the state space is (MN)

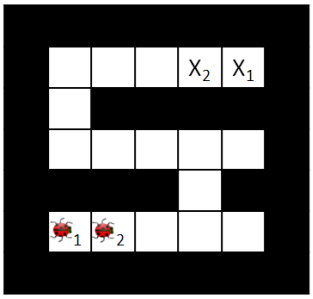
Q1.2. Jumping Bug. Your single insect is alone in the maze again. This time, it has super legs that can take it as far as you want in a straight line in each time step. The disadvantage of these legs is that they make turning slower, so now it takes the insect a time step to change the direction it is facing. Moving v squares requires that all intermediate squares passed through, as well as the v^{th} square, currently be empty. The cost of a multi-square move is still 1 time unit, as is a turning move. As an example, the arrows in the maze below indicate where the insect will be and which direction it is facing after each time step in the optimal (fewest time steps) plan (cost 5):



Provide a smallest correct state space representation. What is the size of the state space?

Answer. A tuple (x, y) giving the position of the insect, plus the direction the insect is facing. The size of the state space is $(4MN)$

Q1.3. Swarm Movement. You control K insects, each of which has a specific target ending location X_k. No two insects may occupy the same square. In each step, all insects move simultaneously to a currently free square (or stay in place); adjacent insects cannot swap in a single time step. Provide a smallest correct state space representation. What is the size of the state space?

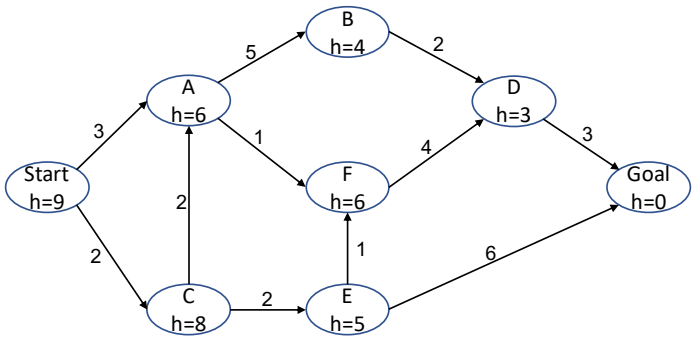


Answer. K tuples $((x_1, y_1), (x_2, y_2), \ldots, (x_K, y_K))$ encoding the coordinates of each insect. The size of the state space is $(MN)^K$

Q2. Graph Search (50 points)

Consider the following graph where `Start` is the start and `Goal` is the goal state. For the following sub-questions, ties are broken in alphabetical order.

1. (8 pts) In what order are states expanded by DFS? What path would DFS return?
   Answer. Order: Start, A, B, D, Goal. Path: Start, A, B, D, Goal



2. (8 pts) In what order are states expanded by BFS? What path would BFS return?
   Answer. Order: Start, A, C, B, E, F, D, Goal. (Note, the answer of Start, A, C, B, F, E, D, Goal. is accepted too) Path: Start, C, E, Goal.

3. (8 pts) In what order are states expanded by UCS? What path would UCS return?
   Answer. Order: Start, C, A, E, F, B, D, Goal. Path: Start, C, E, Goal.

4. (8 pts) In what order are states expanded by Greedy search? What path would Greedy Search return?
   Answer. Order: Start, A, B, D, Goal. Path: Start, A, B, D, Goal

5. (8 pts) In what order are states expanded by A^*? What path would A^* return?
   Answer. Order: Start, A, C, E, F, Goal. Path: Start, C, E, Goal.

6. (5 pts) Is the heuristic in the above graph admissible? If not, provide a minimal set of edges whose costs must be changed along with their new costs in order to make the heuristic admissible.
   Answer. Yes.

7. (5 pts) Is the heuristic in the above graph consistent? If not, provide a minimal set of edges whose costs must be changed along with their new costs in order to make the heuristic consistent.
   Answer. No. Replace the cost of arc (C,E) with the cost ≥ 3

Q3. Look-ahead Graph Search (20 points)

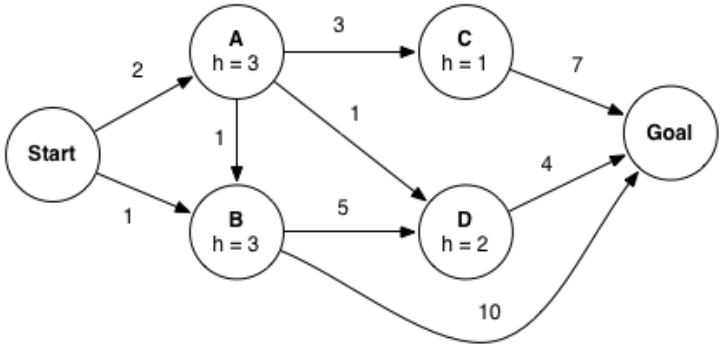
Recall from lecture the general algorithm for Graph Search reproduced below.

Using GRAPH-SEARCH, when a node is expanded it is added to the closed set. This means that even if a node is added to the fringe multiple times it will not be expanded more than once. Consider an alternative version of GRAPH-SEARCH, LOOKAHEAD-GRAPH-SEARCH, which saves memory by using a "fringe-closed-set" keeping track of which states have been on the fringe and only adding a child node to the fringe if the state of that child node has not been added to it at some point. Concretely, we replace the highlighted block above with the highlighted block below.

Now, we've produced a more memory efficient graph search algorithm. However, in doing so, we might have affected some properties of the algorithm. To explore the possible differences, consider the example graph below.

```
function GRAPH-SEARCH(problem, fringe, strategy) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe, strategy)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

```
function LOOKAHEAD-GRAPH-SEARCH(problem, fringe, strategy) return a solution, or failure
    fringe-closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    add INITIAL-STATE[problem] to fringe-closed
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe, strategy)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(node, problem) do
            if STATE[child-node] is not in fringe-closed then
                add STATE[child-node] to fringe-closed
                fringe ← INSERT(child-node, fringe)
        end
    end
```



1. If using LOOKAHEAD-GRAPH-SEARCH with an A* node expansion strategy, which path will this algorithm return? (Suggestion: step through the execution of the algorithm on a scratch sheet of paper and keep track of the fringe and the search tree as nodes get added to the fringe.)

   Answer. S-B-G

   Explanation. Fringe: S

   Fringe-Closed: S

   Expand S.

   Fringe: S-A (f=5), S-B (f=4)

   Fringe-Closed: S, A, B

   Pick path with lowest f-cost. Expand S-¿B.

   Fringe: S-A (f=5), S-B-D (f=8), S-B-G (f=11)

   Fringe-Closed: S, A, B, D, G

   We can keep going, but since we added G to the Fringe-Closed set, we will never add any more paths ending at G onto the fringe, so the final path returned will be the current goal-terminating path on the fringe, which is S-B-G.

2. Assume you run LOOKAHEAD-GRAPH-SEARCH with the A* node expansion strategy and a consistent heuristic, select all statements that are true.

   (a) The EXPAND function can be called at most once for each state. **(True)**

   (b) The algorithm is complete. **True**

   (c) The algorithm will return an optimal solution. **False**

   Explanation. The first two choices are correct. The EXPAND function can be called at most once for each state because the algorithm ensures that for every state, at most one path ending at that state will ever be added to the fringe, which implies that every state gets expanded at most once.

   Like GRAPH-SEARCH, the algorithm is complete. When a node, n, doesn't make it onto the fringe it's always the case that there is another node, m, which is in the fringe or the closed set with STATE[n] = STATE[m].

   However, the algorithm is not guaranteed to return an optimal solution. The bug is that inserting a state into the closed list when a node containing that state is inserted into the fringe means that the first path to that state will be the only one considered. This can cause suboptimality as we can only guarantee that a state has been reached optimally when a node is popped off the fringe.