CS232 - Group Quiz Nr 1

Topic: Search

Wellesley Honor Code: By signing my name I attest to having completed this assignment on my own, or as part of true peer-learning team.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Quiz points**

<table>
<thead>
<tr>
<th>Problem</th>
<th>Possible</th>
<th>Awarded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1: Search Trees</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Q2: Depth-first search</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Q3: Breadth-first search</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Q4: A* search</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Q5.1: Lonely bug</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Q5.2: Swarm</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Q6: Early-goal</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Q7: Look-ahead</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>22</strong></td>
<td></td>
</tr>
</tbody>
</table>
Q1: Search Trees

How many nodes are in the complete search tree for the given state space graph? The start state is S. Draw out the search tree to show your work.

Nr. of Nodes: _____

Q2: Depth first search

What path will DFS find from state S to goal G in the given tree? Show the search tree you created. Write the answer as a string, for example, SXG. Ties are broken alphabetically.

Path: ___________

Q3: Breadth first search
What path will BFS find from state $S$ to goal $G$ in the given tree? Show the search tree you created. Write the answer as a string, for example, SXG. Ties are broken alphabetically.

Path: ___________

Q4: A* Search
Consider A* graph search on the graph below. Arcs are labeled with action costs and states are labeled with heuristic values. Assume that ties are broken alphabetically. Execute the search (by building the search tree) on the box below, then answer the following questions.

In what order are states expanded by A* graph search?
What path does A* graph search return?

1. Start-B-Goal
2. Start-A-C-Goal
3. Start-A-B-Goal
4. Start-A-D-Goal
5. Start-A-B-D-Goal

Q5: HIVE MINDS

The next questions share a common setup. You control one or more insects in a rectangular maze-like environment with dimensions M×N, as shown in the figure below.

At each time step, an insect can move into an adjacent square if that square is currently free, or the insect may stay in its current location. 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 questions, you should answer for a general instance of the problem, not simply for the example maps shown.

Q5.1 HIVE MINDS: LONELY BUG

You control a single insect as shown in the maze below, which must reach a designated target location X, also known as the hive. There are no other insects moving around.

A - Which of the following is a minimal correct state space representation?

1. An integer d encoding the Manhattan distance to the hive.
2. A tuple (x,y) encoding the x and y coordinates of the insect.
3. A tuple (x,y,d) encoding the insect's x and y coordinates as well as the Manhattan distance to the hive.
4. This cannot be represented as a search problem.

B - What is the size of the state space?

1. max(M,N)
2. MN
3. (MN)^2
4. 2^{MN}
5. M^N
6. N^M

C - Which of the following heuristics are admissible (if any)?

1. Manhattan distance from the insect’s location to the hive.
2. Euclidean distance from the insect’s location to the hive.
3. Number of steps taken by the insect.
Q5.2 HIVE MINDS: 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 time step all insects move **simultaneously** to a currently free square (or stay in place); adjacent insects cannot swap in a single time step.

A - Which of the following is a minimal correct state space representation?

1. MN booleans $(b_1,b_2,...,b_{MN})$ encoding whether or not an insect is in each square.
2. $K$ tuples $((x_1,y_1),(x_2,y_2),..., (x_K,y_K))$ encoding the $x$ and $y$ coordinates of each insect.
3. $K$ tuples $((x_1,y_1),(x_2,y_2),..., (x_K,y_K))$ encoding the $x$ and $y$ coordinates of each insect, plus $K$ boolean variables indicating whether each insect is next to another insect.
4. $K$ tuples $((x_1,y_1),(x_2,y_2),..., (x_K,y_K))$ encoding the $x$ and $y$ coordinates of each insect, plus MN booleans indicating which squares are currently occupied by an insect.

B - What is the size of the state space?

1. $MN$
2. $KMN$
3. $(MN)^K$
4. $2^{MNK}$
5. $2^KMN$
6. $(MN)^K2^K$
C - Which of the following heuristics are admissible (if any)?

1. Sum of Manhattan distances from each insect's location to its target location.
2. Sum of costs of optimal paths for each insect to its goal if it were acting alone in the environment, unobstructed by the other insects.
3. Max of Manhattan distances from each insect's location to its target location.
4. Max of costs of optimal paths for each insect to its goal if it were acting alone in the environment, unobstructed by the other insects.
5. Number of insects that have not yet reached their target location.

Q6 EARLY GOAL CHECKING GRAPH SEARCH
Recall from lecture the general algorithm for GRAPH-SEARCH reproduced 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
```

With the above implementation a node that reaches a goal state may sit on the fringe while the algorithm continues to search for a path that reaches a goal state. Let's consider altering the algorithm by testing whether a node reaches a goal state when inserting into the fringe. Concretely, we add the line of code highlighted below:
Now, we’ve produced a graph search algorithm that can find a solution faster. However, In doing so we might have affected some properties of the algorithm. To explore the possible differences, consider the example graph below.

A - If using EARLY-GOAL-CHECKING-GRAPH-SEARCH with a Uniform Cost node expansion strategy, which path, if any, will the algorithm return?

1. S-G
2. S-A-G
3. EARLY-GOAL-CHECKING-GRAPH-SEARCH will not find a solution path.
B - If using EARLY-GOAL-CHECKING-GRAPH-SEARCH with an A* node expansion strategy, which path, if any, will the algorithm return?

1. S-G
2. S-A-G
3. EARLY-GOAL-CHECKING-GRAPH-SEARCH will not find a solution path.

C - Assume you run EARLY-GOAL-CHECKING-GRAPH-SEARCH with the Uniform Cost node expansion strategy, select all statements that are true.

1. The EXPAND function can be called at most once for each state.
2. The algorithm is complete.
3. The algorithm will return an optimal solution.

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

1. The EXPAND function can be called at most once for each state.
2. The algorithm is complete.
3. The algorithm will return an optimal solution.

Q7: LOOKAHEAD GRAPH SEARCH

Recall from lecture the general algorithm for Graph Search reproduced 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
    end
```
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.

```
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
    end
```

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.

```
A - If using LOOKAHEAD-GRAPH-SEARCH with an A* node expansion strategy, which path will this algorithm return? (We strongly encourage you to 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.)

1. S→A→D→G
2. S→A→C→G
3. S→B→G
4. S→B→D→G
5. S→A→B→D→G

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

4. The EXPAND function can be called at most once for each state.
5. The algorithm is complete.
6. The algorithm will return an optimal solution.