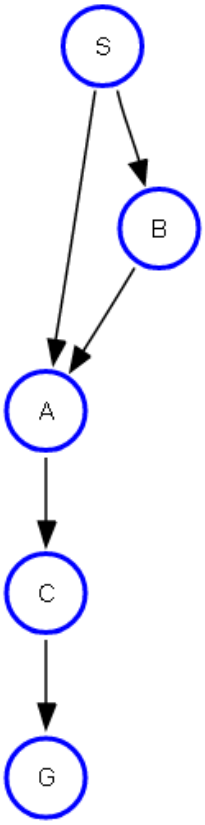
QUESTION 1: SEARCH TREES

How many nodes are in the complete search tree for the given state space graph? The start state is S. You may find it helpful to draw out the search tree on a piece of paper.

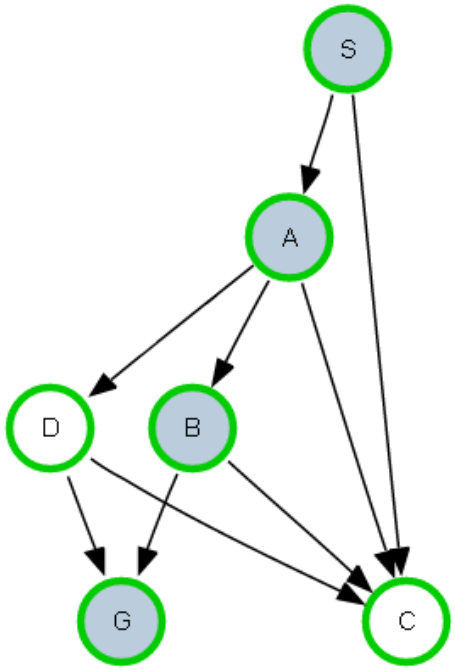


8 ✔

QUESTION 2: BREADTH-FIRST SEARCH

Consider a breadth-first graph search on the graph below, where S is the start and G is the goal state. Assume that ties are broken alphabetically (so a partial plan S->X->A would be expanded before S->X->B and S->A->Z would be expanded before S->B->A). You may find it helpful to execute the search on scratch paper.

Select the final path returned by breadth-first graph search. To select a path, click on each of the nodes in the state-space graph that are part of the path.

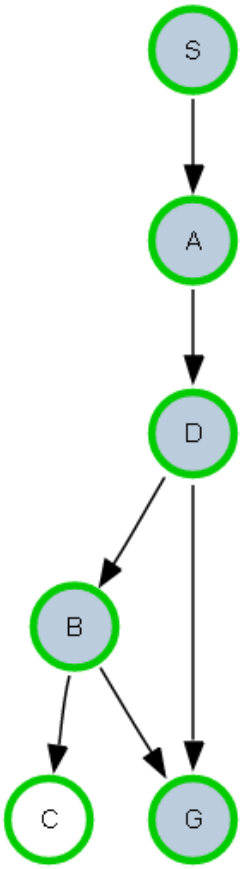


States you correctly identified are marked with a solid green circle, and the states you identified incorrectly are marked with a dotted red circle. ✔

QUESTION 3: DEPTH-FIRST SEARCH

Consider a depth-first graph search on the graph below, where S is the start and G is the goal state. Assume that ties are broken alphabetically (so a partial plan S->X->A would be expanded before S->X->B and S->A->Z would be expanded before S->B->A). You may find it helpful to execute the search on scratch paper.

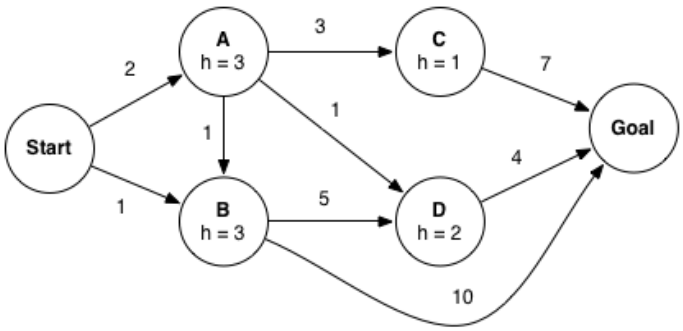
Select the final path returned by depth-first graph search. To select a path, click on each of the nodes in the state-space graph that are part of the path.



States you correctly identified are marked with a solid green circle, and the states you identified incorrectly are marked with a dotted red circle. ✔

QUESTION 4: 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 (so a partial plan S->X->A would be expanded before S->X->B and S->A->Z would be expanded before S->B->A.



In what order are states expanded by A* graph search? You may find it helpful to execute the search on scratch paper.

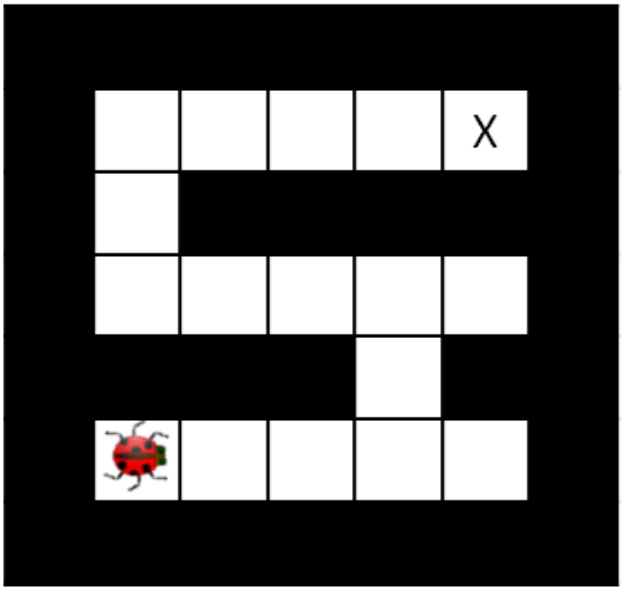
- ✔ Start, A, B, C, D, Goal
- Start, A, C, Goal
- Start, B, A, D, C, Goal
- Start, A, D, Goal
- Start, A, B, Goal
- Start, B, A, D, B, C, Goal

What path does A* graph search return?

- ✔ Start-A-C-Goal
- Start-B-Goal
- Start-A-D-Goal
- Start-A-B-Goal
- Start-A-B-D-Goal

HIVE MINDS

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

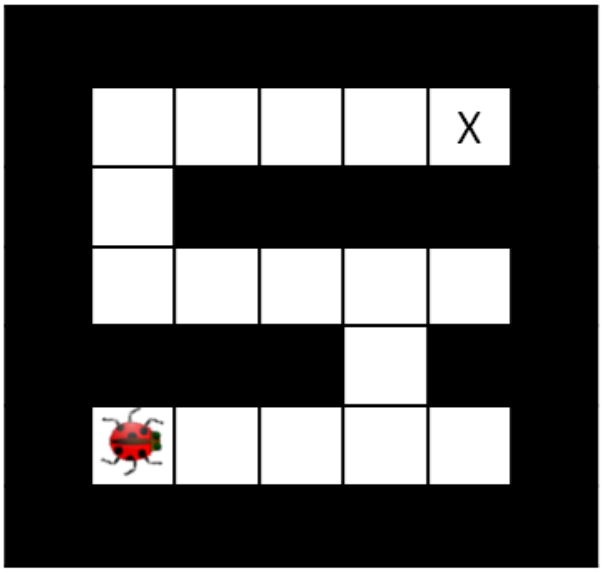


At each time step, an insect can move into an adjacent square if that square is currently free, or the insect may stay in its current location. Squares may be blocked by walls, but the map is known. **Optimality is always in terms of time steps; all actions have cost 1 regardless of the number of insects moving or where they move**.

For each of the five questions, you should answer for a general instance of the problem, not simply for the example maps shown.

QUESTION 5: 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.



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

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

What is the size of the state space?

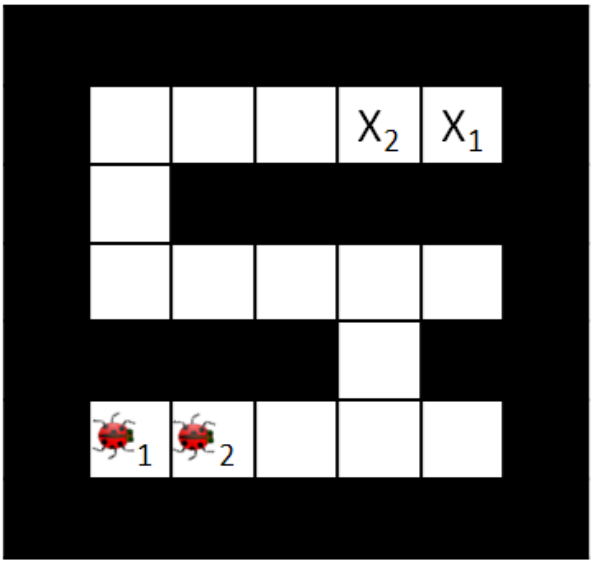
- ◉ MN
- ○ $(MN)^2$
- ○ 2^{MN}
- ○ M^N
- ○ N^M
- ○ $\max(M, N)$

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

- ☑ Manhattan distance from the insect's location to the hive.
- ☑ Euclidean distance from the insect's location to the hive.
- ☐ Number of steps taken by the insect.

QUESTION 6: 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.



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

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

What is the size of the state space?

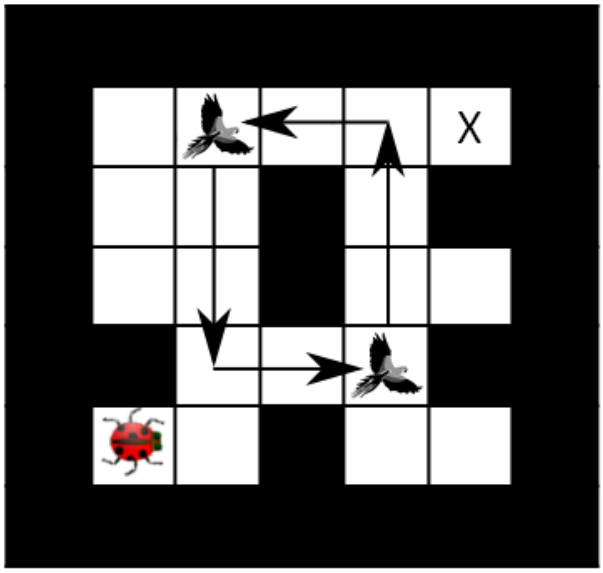
- ○ MN
- ○ 2^{MN}
- ○ KMN
- ◉ $(MN)^K$
- ○ $(MN)^K 2^K$
- ○ $(MN)^K 2^{MN}$
- ○ $2^K MN$
- ○ 2^{MNK}

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

- ☐ Sum of Manhattan distances from each insect's location to its target location.
- ☐ 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.
- ☑ Max of Manhattan distances from each insect's location to its target location.
- ☑ 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.
- ☐ Number of insects that have not yet reached their target location.

QUESTION 7: HIVE MINDS: MIGRATING BIRDS

You again control a single insect, but there are B birds flying along *known* paths. Specifically, at time t each bird b will be at position $(x_b(t), y_b(t))$. The tuple of bird positions repeats with period T. Birds might move **up to 3 squares** per time step. An example is shown below, but keep in mind that you should answer for a general instance of the problem, not simply the map and path shown below.



Your insect *can* share squares with birds and it can even hitch a ride on them! On any time step that your insect shares a square with a bird, the insect may either move as normal or move directly to the bird's next location (either action has cost 1, even if the bird travels farther than one square).

Which of the following is a minimal state representation?

- ○ A tuple (x, y) giving the position of the insect.
- ○ A tuple (x, y) giving the position of the insect, plus a tuple of bird positions (x_b, y_b) giving the location of each bird.
- ◉ A tuple (x, y) giving the position of the insect, plus an integer $r = t \bmod T$ where t is the time step.
- ○ A tuple (x, y) giving the position of the insect, plus B boolean variables indicating whether each of the birds is carrying an insect passenger.
- ○ A tuple (x, y) giving the position of the insect, plus a tuple of bird positions (x_b, y_b) giving the location of each bird, plus B boolean variables indicating whether each of the birds is carrying an insect passenger.

Which of the following is the size of the state space?

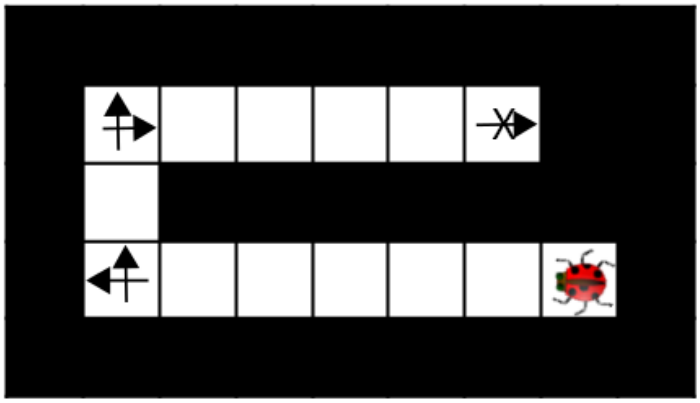
- ○ MN
- ◉ MNT
- ○ MNB
- ○ $MNTB$
- ○ $(MN)^{B+1}$
- ○ $2^{MN}MN$
- ○ $(MN)^{B+1}2^B$

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

- ☐ Cost of optimal path to target in the simpler problem that has no birds.
- ☐ Manhattan distance from the insect's current position to the target.
- ☐ Manhattan distance from the insect's current position to the nearest bird.
- ☑ Manhattan distance from the insect's current position to the target divided by three.

QUESTION 8: HIVE MINDS: 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 vth 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):



Which of the following is a minimal state representation?

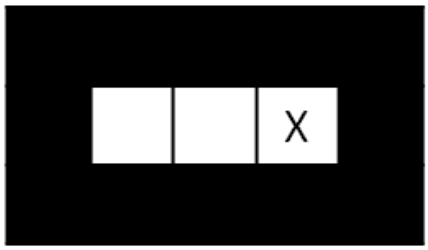
- ✔ ○ A tuple (x, y) giving the position of the insect.
- ◉ A tuple (x, y) giving the position of the insect, plus the direction the insect is facing.
- ○ A tuple (x, y) giving the position of the insect, plus an integer representing the number of direction changes necessary on the optimal path from the insect to the goal.
- ○ A tuple (x, y) giving the position of the insect, plus an integer t representing the number of time steps that have passed.

What is the size of the state space?

- ✔ ○ MN
- ○ $\max(M, N)$
- ○ $\min(M, N)$
- ◉ $4MN$
- ○ $(MN)^2$
- ○ $(MN)^4$
- ○ 4^{MN}

QUESTION 9: HIVE MINDS: LOST AT NIGHT

It is night and you control a single insect. You know the maze, but you do not know what square the insect will start in. You must pose a search problem whose solution is an all-purpose sequence of actions such that, after executing those actions, the insect will be on the exit square, regardless of initial position. The insect executes the actions mindlessly and does not know whether its moves succeed: if it uses an action which would move it in a blocked direction, it will stay where it is. For example, in the maze below, moving right twice guarantees that the insect will be at the exit regardless of its starting position.



Which of the following state representations could be used to solve this problem?

- ✔ ○ A tuple (x, y) representing the position of the insect.
- ○ A tuple (x, y) representing the position of the insect, plus a list of all squares visited by the insect.
- ○ An integer t representing how many time steps have passed, plus an integer b representing how many times the insect's motion has been blocked by a wall.
- ◉ A list of boolean variables, one for each position in the maze, indicating whether the insect could be in that position.
- ○ A list of all positions the insect has been in so far.

What is the size of the state space?

- ✔ ○ MN
- ○ MNT
- ◉ 2^{MN}
- ○ $(MN)^T$
- ○ e^{2MN}
- ○ The state space is infinite.

Which of the following are admissible heuristics?

- ✔ ☐ Total number of possible locations the insect might be in.
- ☑ The maximum of Manhattan distances to the goal from each possible location the insect could be in.
- ☑ The minimum of Manhattan distances to the goal from each possible location the insect could be in.

QUESTION 10: EARLY GOAL CHECKING GRAPH SEARCH

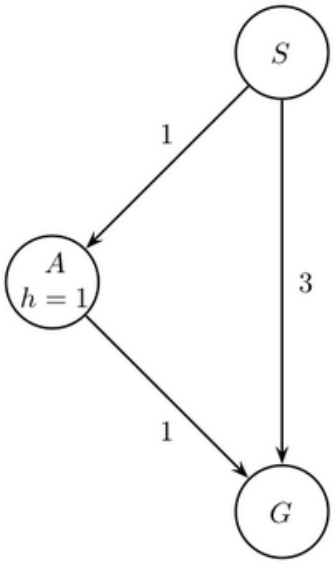
Recall from lecture the general algorithm for GRAPH-SEARCH reproduced below.

```
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:

```
function EARLY-GOAL-CHECKING-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
                if GOAL-TEST(problem, STATE[child-node]) then return child-node
                fringe ← INSERT(child-node, fringe)
            end
    end
```

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.



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

- (•) S-G
- ( ) S-A-G
- ( ) EARLY-GOAL-CHECKING-GRAPH-SEARCH will not find a solution path.

QUESTION 11: LOOKAHEAD GRAPH SEARCH

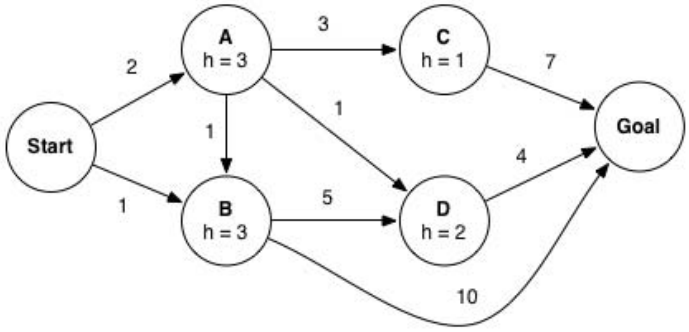
Recall from lecture the general algorithm for Graph Search reproduced below.

```
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
```

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
```

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.



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.)

- ○ $S \to A \to D \to G$
- ◉ $S \to B \to G$
- ○ $S \to A \to C \to G$
- ○ $S \to B \to D \to G$
- ○ $S \to A \to B \to D \to G$

Check

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

- ☑ The EXPAND function can be called at most once for each state.
- ☑ The algorithm is complete.
- ☐ The algorithm will return an optimal solution.

QUESTION 12: MEMORY EFFICIENT 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
```

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 alternate version of GRAPH-SEARCH, MEMORY-EFFICIENT-GRAPH-SEARCH, which saves memory by (a) not adding node *n* to the fringe if STATE[*n*] is in the closed set, and (b) checking if there is already a node in the fringe with last state equal to STATE[*n*]. If so, rather than simply inserting, it checks whether the old node or the new node has the cheaper path and then accordingly leaves the fringe unchanged or replaces the old node by the new node.

By doing this the fringe needs less memory, however insertion becomes more computationally expensive.

More concretely, MEMORY-EFFICIENT-GRAPH-SEARCH is shown below with the changes highlighted.

```
function MEMORY-EFFICIENT-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 ← SPECIAL-INSERT(child-node, fringe, closed)
            end
    end


function SPECIAL-INSERT(node, fringe, closed) return fringe
    if STATE[node] not in closed set then
        if STATE[node] is not in STATE[fringe] then
            fringe ← INSERT(node, fringe)
        else if STATE[node] has lower cost than cost of node in fringe reaching STATE[node] then
            fringe ← REPLACE(node, fringe)
```

Now, we've produced a more memory efficient graph search algorithm. However, in doing so, we might have affected some properties of the algorithm. Assume you run MEMORY-EFFICIENT-GRAPH-SEARCH with the A* node expansion strategy and a consistent heuristic, select all statements that are true.

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

QUESTION 13: A*-CSCS

Recall that a dictionary, also known as a hashmap, works as follows:
Inserting a key-value pair into a dictionary when the key is not already in the dictionary adds the pair to the dictionary:

```
dict ← an empty dictionary
dict["key"] ← "value"
print dict["key"]
→ "value"
```

Updating the value associated with a dictionary entry is done as follows:

```
dict["key"] ← "new value"
print dict["key"]
→ "new value"
```

We saw that for A^* *graph* search to be guaranteed to be optimal the heuristic needs to be consistent. In this question we explore a new search procedure using a dictionary for the closed set, A^* -graph-search-with-Cost-Sensitive-Closed-Set (A^* - CSCS).

```
function A*-CSCS-GRAPH-SEARCH(problem, fringe, strategy) return a solution, or failure
    closed ← an empty dictionary
    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 or COST[node] < closed[STATE[node]] then
            closed[STATE[node]] ← COST[node]
            for child-node in EXPAND(node, problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

Rather than just inserting the last state of a node into the closed set, we now store the last state paired with the cost of the node. Whenever A^* - CSCS considers expanding a node, it checks the closed set. Only if the last state is not a key in the closed set, or the cost of the node is less than the cost associated with the state in the closed set, the node is expanded.

For **regular** A^* **graph search** which of the following statements are true?

- ☐ If h is admissible, then A^* graph search finds an optimal solution.
- ☑ If h is consistent, then A^* graph search finds an optimal solution.

Check

In each of the following parts, select all true statements about A^* **-CSCS**

- ☑ If h is admissible, then A^* - CSCS finds an optimal solution.
- ☑ If h is consistent, then A^* - CSCS finds an optimal solution.

Check

- ☑ If h is admissible, then A^* - CSCS will expand at most as many nodes as A^* **tree** search.
- ☑ If h is consistent, then A^* - CSCS will expand at most as many nodes as A^* **tree** search.

- ☐ If h is admissible, then A^* - CSCS will expand at most as many nodes as A^* **graph** search.
- ☑ If h is consistent, then A^* - CSCS will expand at most as many nodes as A^* **graph** search.