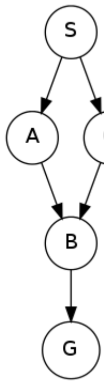
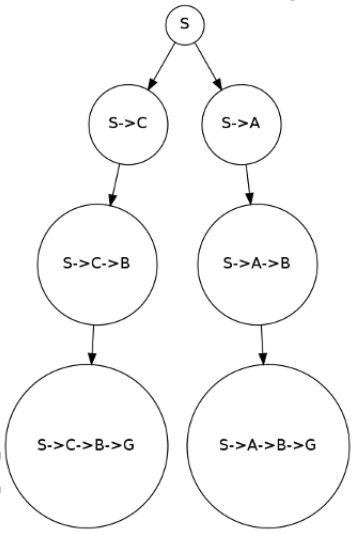
1. (6 points) Search Trees

   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.



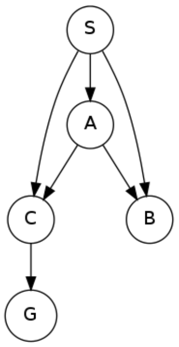
Answer. **7**

Explanation. The complete search tree would look like this.



2. (6 points) Depth-First Graph 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.



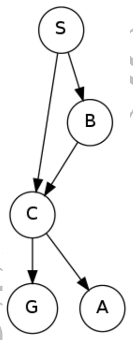
Please enter the final path returned by depth-first graph search on the line below. Your answer should be a string with S as your first character and G as your last character. Don't include arrows or spaces in your submission. For example, if you believe the path is S→X→G, please enter SXG on the line.

Answer. ***SACG***

Explanation.
Step 1: Expand S; Fringe: S-A, S-B, S-C; Closed Set: S.
Step 2: Expand S-A; Fringe: S-A-B, S-A-C, S-B, S-C; Closed Set: S, A
Step 3: Expand S-A-B; Fringe: S-A-C, S-B, S-C; Closed Set: S, A, B
Step 4: Expand S-A-C; Fringe: S-A-C-G, S-B, S-C; Closed Set: S, A, B, C
Step 5: Expand S-A-C-G, finding the goal

3. (6 points) Breadth-First Graph 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.



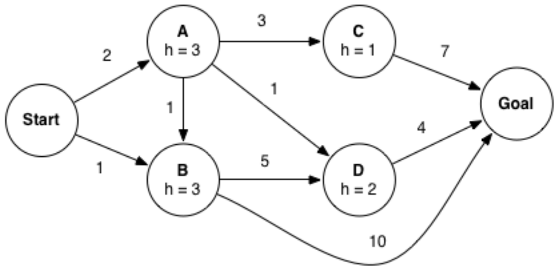
Please enter the final path returned by breadth-first graph search on the line below. Your answer should be a string with S as your first character and G as your last character. Don't include arrows or spaces in your submission. For example, if you believe the path is S→X→G, please enter SXG on the line.

Answer. ***SCG***

Explanation.
Step 1: Expand S; Fringe: S-B, S-C; Closed Set: S
Step 2: Expand S-B; Fringe: S-C, S-B-C; Closed Set: S, B
Step 3: Expand S-C; Fringe: S-B-C, S-C-A, S-C-G; Closed Set: S, B, C
Step 4: Pop S-B-C from our fringe, but do not expand it, because C is in our closed set;
Fringe: S-C-A, S-C-G; Closed Set: S, B, C
Step 5: Expand S-C-A; Fringe: S-C-G; Closed Set: S, B, C, A
Step 6: Expand S-C-G, finding the goal

4. (7 points) A* Graph 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).



(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

(b) What path does A* graph search return?

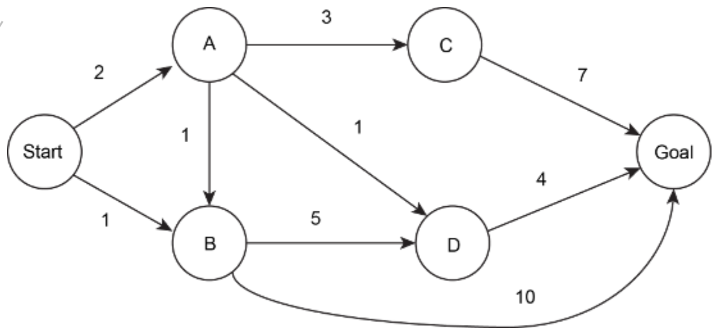
☐ Start-A-C-Goal ☐ Start-B-Goal ✓ ***Start-A-D-Goal***

☐ Start-A-B-Goal ☐ Start-A-B-D-Goal Explanation.

Step 1: Expand S; Fringe: (S-A, 5), (S-B, 4); Closed Set: S
Step 2: Expand S-B; Fringe: (S-A, 5), (S-B-D, 8), (S-B-G, 11); Closed Set: S, B
Step 3: Expand S-A; Fringe: (S-B-D, 8), (S-B-G, 11), (S-A-D, 5), (S-A-C, 6); Closed Set: S, B, A
Step 4: Expand S-A-D; Fringe: (S-B-D, 8), (S-B-G, 11), (S-A-C, 6), (S-A-D-G, 7); Closed Set: S, B, A, D
Step 5: Expand S-A-C; Fringe: (S-B-D, 8), (S-B-G, 11), (S-A-D-G, 7), (S-A-C-G, 13); Closed Set: S, B, A, D, C
Step 6: Expand S-A-D-G, finding the goal; Closed Set: S, B, A, D, C, G; Return: Start-A-D-Goal

5. (7 points) Uniform-Cost Graph Search

Consider the graph below. Arcs are labeled with their weights. 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).



(a) In what order are states expanded by Uniform Cost 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

(b) What path does uniform cost search return?

☐ Start-A-C-Goal ☐ Start-B-Goal [✓ ***Start-A-D-Goal***]

☐ Start-A-B-Goal ☐ Start-A-B-D-Goal

Explanation.

Step 1: Expand S; Fringe: (S-A, 2), (S-B, 1); Closed Set: S

Step 2: Expand S-B; Fringe: (S-A, 2), (S-B-D, 6), (S-B-G, 11); Closed Set: S, B

Step 3: Expand S-A; Fringe: (S-B-D, 6), (S-B-G, 11), (S-A-D, 3), (S-A-C, 5);
Closed Set: S, B, A

Step 4: Expand S-A-D; Fringe: (S-B-D, 6), (S-B-G, 11), (S-A-C, 5), (S-A-D-G, 7);
Closed Set: S, B, A, D

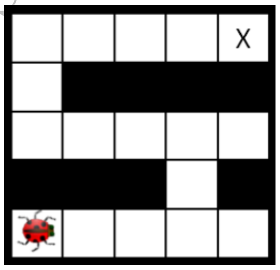
Step 5: Expand S-A-C; Fringe: (S-B-D, 6), (S-B-G, 11), (S-A-D-G, 7), (S-A-C-G, 12);
Closed Set: S, B, A, D, C

Step 6: Pop S-B-D from our fringe, but do not expand it, because D is in our closed set;
Fringe: (S-B-G, 11), (S-A-D-G, 7), (S-A-C-G, 12); Closed Set: S, B, A, D, C

Step 7: Expand S-A-D-G, finding the goal; Closed Set: S, B, A, D, C, G;
Return: Start-A-D-Goal

6. (6 points) Hive Minds: Lonely Bug

Introduction. 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 either (a) move into an adjacent square if that square is currently free, or (b) 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.

Problems. For this problem, you control a single insect as shown in the maze above, 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?

- ☐ 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.

Explanation. The position tuple is enough to calculate the goal test and successor functions. Goal test: $(x, y) = Goal$; Successor: Similar to Pacman successors. EAST changes (x, y) to $(x + 1, y)$, for example.

(b) What is the size of the state space?

√ MN ☐ $(MN)^2$ ☐ 2^{MN} ☐ M^N ☐ N^M ☐ $\max(M, N)$

Explanation. There are MN total values that the position (x, y) can take.

(c) 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.

Explanation.

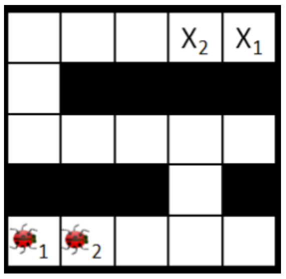
Option 1: If there were no walls, Manhattan distance is the true cost to the goal. Therefore, if there are walls, the Manhattan distance will always be an underestimate, so it is admissable.

Option 2: Euclidean distance will always be less than Manhattan distance, so it is admissable.

Option 3: Consider if the insect is 1 action from the goal, but has already taken 5 steps. This heuristic will return 5, which is an overestimate of the true cost of 1, so not admissable.

7. (7 points) 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 the smallest 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.

Explanation. Given the position of every insect, we can formulate the goal test and successor function.

Goal test: $\forall i \in \{1, 2, ..., K\} : (x_i, y_i) = Goal_i$

Successor: Similar to Pacman successors. EAST changes (x_i, y_i) to $(x_i + 1, y_i)$, for example.

(b) What is the size of the above state space?

☐ MN ☐ 2^{MN} ☐ KMN ✓ $\binom{MN}{k}$ ☐ $(MN)^K 2^K$ ☐ $(MN)^K 2^{MN}$ ☐ $2^K MN$ ☐ 2^{MNK}

Explanation. There are MN choices for each position tuple, and K total tuples, so at first approximation, we might claim the total number of states is $(MN)^K$. However, accounting for the fact that no two insects occupy the same square, in fact, we have $\binom{MN}{k}$ total states: MN states to choose from for the first insect, $MN - 1$ for the second, ..., $MN - K + 1$ states to choose from for the K-th insect.

(c) 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.

Explanation.

Option 1: Consider the state where every insect is right next to its target location. Because you can move every insect in one turn, the true cost from this state to the goal is 1. However, the heuristic would return K, which is an overestimate, so not admissable.

Option 2: Consider the same state as above. The heuristic would again return K, so not admissable.

Option 3: The cost will never be less than the distance from the furthest insect from the goal. The cost could be greater because the insect's path could be obstructed by another insect, so admissable.

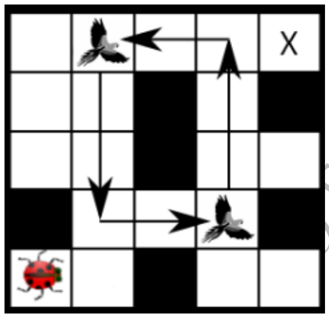
Option 4: Admissable, for the same reason as option 3.

Option 5: Consider the same state as in option 1. The heuristic would again return K, so not admissable.

8. (7 points) 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).



(a) 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.

Explanation. You need the (x, y) position for the goal test, and, in addition to the position of the insect, the bird positions need to be known to compute successors. To know the bird positions it is sufficient to know the time modulo T.

Goal test: $(x, y) = Goal$

Successor: All of the actions from the last problem, and an additional BIRD action which is only available when $(x, y) = (x_b(t), y_b(t))$ and moves (x, y) to $(x_b(t + 1), y_b(t + 1))$.

(b) Which of the following is the size of the state space?

☐ MN √ MNT ☐ MNB ☐ $MNTB$ ☐ $(MN)^{B+1}$ ☐ $2^{MN}MN$ ☐ $(MN)^{B+1}2^B$

Explanation. The position tuple (x, y) can take MN values, and the time t can take T values, so the total number of states is MNT.

(c) 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.***

Explanation.

Option 1: Consider the state where an insect is three squares from the goal, but is on the same spot as a bird that will fly to the goal. If the insect takes the BIRD action, it will move to the goal in 1 step. However, the heuristic would return 3, so not admissible.

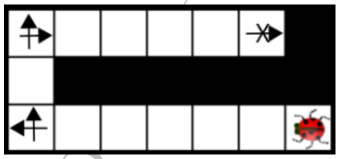
Option 2: Consider the same state as above. The heuristic would return 3, so not admissible.

Option 3: Consider the state where an insect is two squares from a bird, but one grid from the goal. The heuristic would return 2, while the true cost is 1, so not admissible.

Option 4: An insect can travel at most 3 grids per move (via a bird), so dividing the Manhattan distance by 3 gives us an admissible heuristic.

9. (6 points) 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):



(a) 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.

Explanation. You need the (x, y) position for the goal test, and both the (x, y) position and direction for successors.

Goal test: $(x, y) = Goal$

Successor: You can either move as far as possible in the direction, or you can change directions.

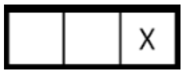
(b) What is the size of the state space?

☐ MN ☐ $\max(M, N)$ ☐ $\min(M, N)$ √ $4MN$ ☐ $(MN)^2$ ☐ $(MN)^4$ ☐ 4^{MN}

Explanation. The position (x, y) can take on MN different values, and there are 4 directions, so the total state space is $4MN$.

10. (7 points) 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 is guaranteed to 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.



(a) 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.¡/text¿
- ☐ 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.

Explanation. The state is a two dimensional array *bool*, which stores a boolean variable for every position (x, y) at location $bool[x][y]$.

Goal test: $bool[x_{goal}][y_{goal}] = True$ and all other $bool[x][y] = False$

Successor: Actions available to us are NORTH,SOUTH,EAST,WEST. The action WEST, for example will move us from state *bool* to a new state $bool_{next}$ in the following way:

```
for x in range(width):
for y in range(height):
bool_next[x][y] = bool[x+1][y] or (bool[x][y] and is_wall((x-1,y)))
```

(b) What is the size of the state space?

☐ MN ☐ MNT ✓ 2^{MN} ☐ $(MN)^T$ ☐ e^{2MN} ☐ ∞

Explanation. There are MN total booleans, and 2 values for each boolean, so there are 2^{MN} total possible states.

(c) Which of the following are admissible heuristics?

- ☐ Total number of possible locations the insect might be in.
- ✓ ***The maximum of Manhattan distances to the exit square from each possible location the insect could be in.***
- ✓ ***The minimum of Manhattan distances to the exit square from each possible location the insect could be in.***

Explanation.

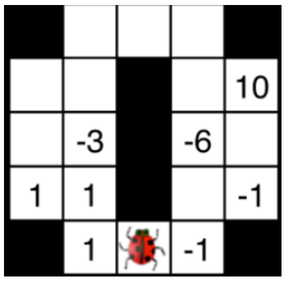
Option 1: Consider the maze above. From the beginning state, we are 2 moves away from the exit square. However, the heuristic would suggest 3, which is not admissable.

Option 2: Our all-purpose solution must work for the possible insect location that is furthest from the exit square, so the solution must be cost at least as much as the cost from that location, so this heuristic is admissable.

Option 3: Admissable, for the same reason as above.

11. (7 points) Hive Minds: Time Limit

In this problem, the ladybug has a limited number of timesteps remaining. For each timestep, there is no moving penalty, but the remaining time will decrease by one. The ladybug will gain or lose points when it lands on positive or negative values, respectively. The ladybug must move to a new square for each timestamp, and the ladybug cannot move to a square that it has already visited. Your goal in this problem is to find the optimal score (higher is better) for a given timestep limit.



(a) What is the optimal score for a timestep of 2? *Answer.* ***2***

(b) What is the optimal score for a timestep of 5? *Answer.* ***8***

(c) What is the optimal score for a timestep of 8? *Answer.* ***8***

(d) What is the optimal score for a timestep of 11? *Answer.* ***13***

12. (7 points) Early Goal Checking Graph Search

Recall the GRAPH-SEARCH algorithm (Fig. 1).

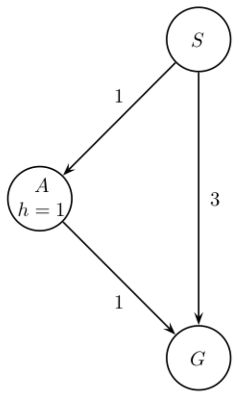
```
function GRAPH-SEARCH(problem, fringe, strategy) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe, strategy)
        if GOAL-TEST(problem, 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
```

Figure 1: General algorithm for GRAPH-SEARCH.

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.



(a) 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.

Explanation. The fringe starts out with the start state, S. We expand S. The successors of S are S-A and S-G. The path S-G passes the goal test, so the algorithm returns S-G.

(b) If using EARLY-GOAL-CHECKING-GRAPH-SEARCH with an A* 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.

Explanation. Like in the previous question, the fringe starts out with the start state, S. We expand S. The successors of S are S-A and S-G. The path S-G passes the goal test, so the algorithm returns S-G.

(c) Assume you run EARLY-GOAL-CHECKING-GRAPH-SEARCH with the Uniform Cost node expansion strategy, 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.

Explanation. The solutions for this question are in the next part of this question.

(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.

√ ***The EXPAND function can be called at most once for each state.***

√ ***The algorithm is complete.***

☐ The algorithm will return an optimal solution.

Explanation. For both UCS and A* with EARLY-GOAL-CHECKING-GRAPH-SEARCH, the first two choices are correct. The EXPAND function can be called at most once for each state because, like in 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. Completeness is also guaranteed, meaning that if there exists a solution to the search problem, the algorithm will find it. Adding the goal-check condition early does not change the completeness of the algorithm.

However, optimality is not guaranteed. The bug is that EARLY-GOAL-CHECKING-GRAPH-SEARCH checks if the child-node satisfies the goal test at the time of insertion onto the fringe, rather than at the time the node is popped from the fringe. As a consequence of this bug, a suboptimal path can be returned.

For the special case of breadth-first graph search this modification will not affect its property of finding a plan that reaches a goal state that requires a minimal number of actions while at the same time giving a minor improvement in running time for breadth-first graph search. (Think about why this is the case!)

Keep in mind, however, that for consistency across DFS, BFS, UCS, and A* in this class we use the generally applicable version of graph search presented in lecture. For the projects you should use the version presented in lecture (reproduced at the top of this page).

13. (7 points) In the GRAPH-SEARCH algorithm (Fig. 2), 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,

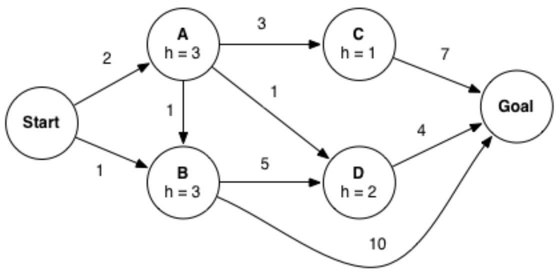
```
function GRAPH-SEARCH(problem, fringe, strategy) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe, strategy)
        if GOAL-TEST(problem, 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
```

Figure 2: General algorithm for 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 in Fig. 2 with the highlighted block in the code 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.



(a) 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.)

- ☐ $S \to A \to D \to G$
- √ $S \to B \to G$
- ☐ $S \to A \to C \to G$
- ☐ $S \to B \to D \to G$
- ☐ $S \to A \to B \to D \to G$

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.

(b) 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.

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.

14. (7 points) 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.***

Explanation. The EXPAND function can be called at most once per state. Before a node is expanded it is added to the closed set and never expanded again.

The memory-efficient 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].

The memory-efficient algorithm will return an optimal solution path. The original graph search algorithm is guaranteed to return an optimal solution path, and the updates made to the algorithm only remove nodes from consideration when replacing them with a node that ends in the same state and has lower cost.

Tip for projects. Memory-efficient-graph-search is the version Russell and Norvig (3rd ed) present for uniform-cost-graph-search. For the projects make sure to use the version shown above.

15. (7 points) 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.

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

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

Explanation. Consistency ensures that the first time we expand a state, the cost to that state is optimal. This is important for A^* graph search, because the first time a state is expanded, it will go in the closed set, and it will NEVER be expanded again.

(b) Select all true statements about A^*-CSCS.

- $\surd$ ***If h is admissible, then A^*- CSCS finds an optimal solution.***
- $\surd$ ***If h is consistent, then A^*- CSCS finds an optimal solution.***
- $\surd$ ***If h is admissible, then A^*-CSCS will expand at most as many nodes as A^* tree search.***
- $\surd$ ***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.
- $\surd$ ***If h is consistent, then A^*- CSCS will expand at most as many nodes as A^* graph search.***

Explanation.

There are two changes between A^*-CSCS and normal A^* graph search.

1) The closed set of states is now a dictionary: (key, value) = (state, cost of state)

2) We will expand a state S if S is not in the closed set OR the current cost of S is better than the last best cost of S we've found

We do not need consistency for A^*-CSCS, because we can expand a state once suboptimally, and then expand it again in a more optimal way. So both are true.

A^* tree search will expand every state that comes off of the fringe, while A^*-CSCS will only expand the states that meet both conditions. Therefore, A^* tree search will always expand more nodes than A^*-CSCS, regardless of the heuristic value. So both are true.

When we have a consistent heuristic, the new condition in A^*-CSCS (the current cost of S is better than the last best cost of S we've found) will never be true, because every time we expand a state S, the cost to that state would be optimal. Therefore, given a consistent heuristic, A^*-CSCS will expand the same states as A^* graph search.

When we have an admissible heuristic, A^*-CSCS may re-expand some of the states in the closed set, while A^* graph search will never re-expand any states, so A^*-CSCS will expand more states.

So admissible is false, consistent is true.