Parts (c), (d), and (e) Consider a search problem where all edges have cost 1 and the optimal solution has cost C. Let h be a heuristic which is $\max\{h^* - k, 0\}$, where h^* is the actual cost to the closest goal and k is a nonnegative constant.

(c) (2 pt) Circle all of the following that are true (if any).

(i) h is admissible.

(ii) h is consistent.

(iii) A* tree search (no closed list) with h will be optimal.

(iv) A* graph search (with closed list) with h will be optimal.

(d) (1 pt) Which of the following is the most reasonable description of how much more work will be done (= how many more nodes will be expanded) with heuristic h compared to h^*, as a function of k?

(i) Constant in k

(ii) Linear in k

(iii) Exponential in k

(iv) Unbounded

Now consider the same search problem, but with a heuristic h' which is 0 at all states that lie along an optimal path to a goal and h^* elsewhere.

(e) (2 pt) Circle all of the following that are true (if any).

(i) h' is admissible.

(ii) h' is consistent.

(iii) A* tree search (no closed list) with h' will be optimal.

(iv) A* graph search (with closed list) with h' will be optimal.

1 תשובה

Parts (c), (d), and (e) Consider a search problem where all edges have cost 1 and the optimal solution has cost C. Let h be a heuristic which is $\max\{h^* - k, 0\}$, where h^* is the actual cost to the closest goal and k is a nonnegative constant.

(c) (2 pt) Circle all of the following that are true (if any).

(i) h is admissible. h^* is admissible and $h \leq h^*$.

(ii) h is consistent. h^* is consistent and subtracting a constant from both sides does not change the underlying inequality.

(iii) A* tree search (no closed list) with h will be optimal. Tree search requires admissibility.

(iv) A* graph search (with closed list) with h will be optimal. Graph search requires admissibility and consistency.

(d) (1 pt) Which of the following is the most reasonable description of how much more work will be done (= how many more nodes will be expanded) with heuristic h compared to h^*, as a function of k?

(i) Constant in k

(ii) Linear in k

(iii) Exponential in k

(iv) Unbounded

At $k = 0$, only the d nodes on an optimal path to the closest goal are expanded for search depth (= optimal path length) d. At $k = \max(h)$, the problem reduces to uninformed search and BFS expands b^d nodes for branching factor b. In general, all nodes within distance k of the closest goal will have heuristic $h = 0$ and uninformed search may expand them. Note that search reduces to BFS since A^* with $h = 0$ is UCS and in this search problem all edges have cost 1 so path cost = path length.

Now consider the same search problem, but with a heuristic h' which is 0 at all states that lie along an optimal path to a goal and h^* elsewhere.

(e) (2 pt) Circle all of the following that are true (if any).

(i) h' is admissible.

(ii) h' is consistent.

(iii) A* tree search (no closed list) with h' will be optimal.

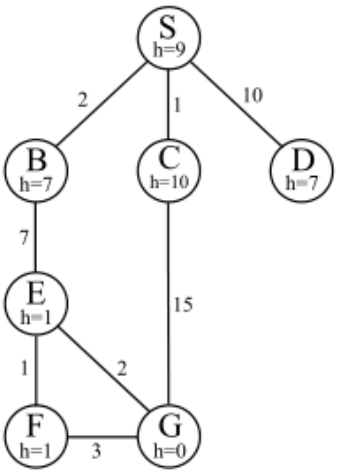
(iv) A* graph search (with closed list) with h' will be optimal.

h' is not consistent as needed for optimality of graph search. Consistency is violated by any edge connecting a state outside of the optimal path to the optimal path since h' drops faster than the reduction in the true cost.

שאלה 2

1. (12 points) Search

Consider the search graph shown below. S is the start state and G is the goal state. All edges are bidirectional.



For each of the following search strategies, give the path that would be returned, or write *none* if no path will be returned. If there are any ties, assume alphabetical tiebreaking (i.e., nodes for states earlier in the alphabet are expanded first in the case of ties).

(a) (1 pt) Depth-first graph search

(b) (1 pt) Breadth-first graph search

(c) (1 pt) Uniform cost graph search

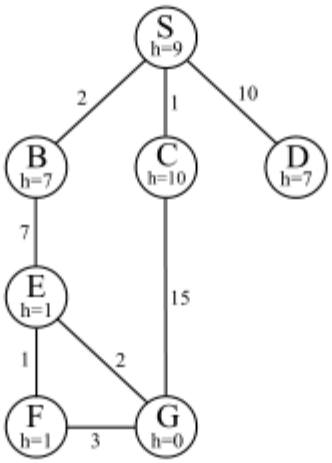
(d) (1 pt) Greedy graph search

(e) (2 pt) A* graph search

תשובה 2

1. (12 points) Search

Consider the search graph shown below. S is the start state and G is the goal state. All edges are bidirectional.



For each of the following search strategies, give the path that would be returned, or write *none* if no path will be returned. If there are any ties, assume alphabetical tiebreaking (i.e., nodes for states earlier in the alphabet are expanded first in the case of ties).

(a) (1 pt) Depth-first graph search

S-B-E-F-G

(b) (1 pt) Breadth-first graph search

S-C-G

(c) (1 pt) Uniform cost graph search

S-B-E-G

(d) (1 pt) Greedy graph search

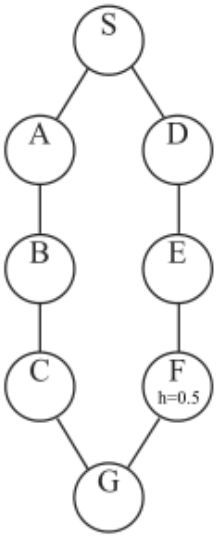
S-B-E-G

(e) (2 pt) A* graph search

S-B-E-G

שאלה 3

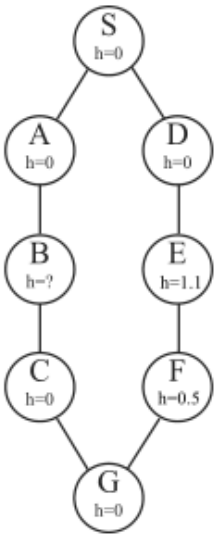
For the following question parts, all edges in the graphs discussed have cost 1.



(f) **(3 pt)** Suppose that you are designing a heuristic h for the graph on the right. You are told that $h(F) = 0.5$, but given no other information. What ranges of values are possible for $h(D)$ if the following conditions must hold? Your answer should be a range, e.g. $2 \leq h(D) < 10$. You may assume that h is nonnegative.

i. h must be admissible

ii. h must be admissible and consistent



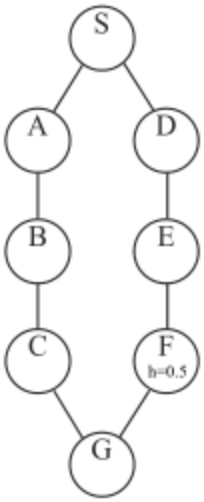
(g) **(3 pt)** Now suppose that $h(\text{F}) = 0.5$, $h(\text{E}) = 1.1$, and all other heuristic values except $h(\text{B})$ are fixed to zero (as shown on the right). For each of the following parts, indicate the range of values for $h(\text{B})$ that yield an admissible heuristic AND result in the given expansion ordering when using A* graph search. If the given ordering is impossible with an admissible heuristic, write *none*. Break ties alphabetically. Again, you may assume that h is nonnegative.

i. B expanded before E expanded before F

ii. E expanded before B expanded before F

תשובה 3

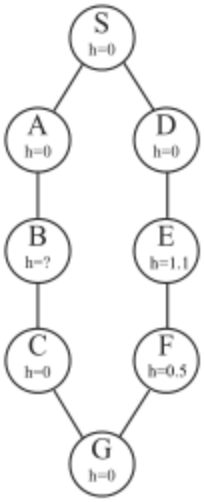
For the following question parts, all edges in the graphs discussed have cost 1.



(f) **(3 pt)** Suppose that you are designing a heuristic h for the graph on the right. You are told that $h(F) = 0.5$, but given no other information. What ranges of values are possible for $h(D)$ if the following conditions must hold? Your answer should be a range, e.g. $2 \leq h(D) < 10$. You may assume that h is nonnegative.

i. h must be admissible
$0 \leq h(D) \leq 3$
The path to goal from D is 3.

ii. h must be admissible and consistent
$0 \leq h(D) \leq 2.5$
In order for $h(E)$ to be consistent, it must hold that $h(E) - h(F) \leq 1$, since the path from E to F is of cost 1. Similarly, it must hold that $h(D) - h(F) = h(D) - 0.5 \leq 2$, or $h(D) \leq 2.5$.



(g) **(3 pt)** Now suppose that $h(\text{F}) = 0.5$, $h(\text{E}) = 1.1$, and all other heuristic values except $h(\text{B})$ are fixed to zero (as shown on the right). For each of the following parts, indicate the range of values for $h(\text{B})$ that yield an admissible heuristic AND result in the given expansion ordering when using A* graph search. If the given ordering is impossible with an admissible heuristic, write *none*. Break ties alphabetically. Again, you may assume that h is nonnegative.

i. B expanded before E expanded before F
$0.0 \leq h(\text{B}) \leq 1.1$

ii. E expanded before B expanded before F
$1.1 < h(\text{B}) \leq 1.5$

שאלה 4

2. (12 points) Formulation: Holiday Shopping

You are programming a holiday shopping robot that will drive from store to store in order to buy all the gifts on your shopping list. You have a set of N gifts $G = \{g_1, g_2, \ldots g_N\}$ that must be purchased. There are M stores, $S = \{s_1, s_2, \ldots s_M\}$ each of which stocks a known inventory of items: we write $g_k \in s_i$ if store s_i stocks gift g_k. Shops may cover more than one gift on your list and will never be out of the items they stock. Your home is the store s_1, which stocks no items.

The actions you will consider are travel-and-buy actions in which the robot travels from its current location s_i to another store s_j in the fastest possible way and buys whatever items remaining on the shopping list that are sold at s_j. The time to travel-and-buy from s_i to s_j is $t(s_i, s_j)$. You may assume all travel-and-buy actions represent shortest paths, so there is no faster way to get between s_i and s_j via some other store. The robot begins at your home with no gifts purchased. You want it to buy all the items in as short a time as possible and return home.

For this planning problem, you use a state space where each state is a pair (s, u) where s is the current location and u is the set of unpurchased gifts on your list (so $g \in u$ indicates that gift g has not yet been purchased).

(a) (1 pt) How large is the state space in terms of the quantities defined above?

(b) (4 pt) For each of the following heuristics, which apply to states (s, u), circle whether it is admissible, consistent, neither, or both. Assume that the minimum of an empty set is zero.

(neither / admissible / consistent / both) The shortest time from the current location to any other store: $\min_{s' \neq s} t(s, s')$

(neither / admissible / consistent / both) The time to get home from the current location: $t(s, s_1)$

(neither / admissible / consistent / both) The shortest time to get to any store selling any unpurchased gift: $\min_{g \in u}(\min_{s':g \in s'} t(s, s'))$

(neither / admissible / consistent / both) The shortest time to get home from any store selling any unpurchased gift: $\min_{g \in u}(\min_{s':g \in s'} t(s', s_1))$

(neither / admissible / consistent / both) The total time to get each unpurchased gift individually: $\sum_{g \in u}(\min_{s':g \in s'} t(s, s'))$

(neither / admissible / consistent / both) The number of unpurchased gifts times the shortest store-to-store time: $|u|(\min_{s_i, s_j \neq s_i} t(s_i, s_j))$

You have waited until very late to do your shopping, so you decide to send an swarm of R robot minions to shop in parallel. Each robot moves at the same speed, so the same store-to-store times apply. The problem is now to have all robots start at home, end at home, and for each item to have been bought by at least one robot (you don't have to worry about whether duplicates get bought). Hint: consider that robots may not all arrive at stores in sync.

(c) (4 pt) Give a minimal state space for this search problem (be formal and precise!)

One final task remains: you still must find your younger brother a stuffed Woozle, the hot new children's toy. Unfortunately, no store is guaranteed to stock one. Instead, each store s_i has an initial probability p_i of still having a Woozle available. Moreover, that probability drops exponentially as other buyers scoop them up, so after t time has passed, s_i's probability has dropped to $\beta^t p_i$. You cannot simply try a store repeatedly; once it is out of stock, that store will stay out of stock. Worse, you only have a single robot that can handle this kind of uncertainty! Phrase the problem as a single-agent MDP for planning a search policy for just this one gift (no shopping lists). You receive a single reward of +1 upon successfully buying a Woozle, at which point the MDP ends (don't worry about getting home); all other rewards are zeros. You may assume a discount of 1.

(d) (3 pt) Give a minimal state space for this MDP (be formal and precise!)

תשובה 4.1

2. (12 points) Formulation: Holiday Shopping

You are programming a holiday shopping robot that will drive from store to store in order to buy all the gifts on your shopping list. You have a set of N gifts $G = \{g_1, g_2, \ldots g_N\}$ that must be purchased. There are M stores, $S = \{s_1, s_2, \ldots s_M\}$ each of which stocks a known inventory of items: we write $g_k \in s_i$ if store s_i stocks gift g_k. Shops may cover more than one gift on your list and will never be out of the items they stock. Your home is the store s_1, which stocks no items.

The actions you will consider are travel-and-buy actions in which the robot travels from its current location s_i to another store s_j in the fastest possible way and buys whatever items remaining on the shopping list that are sold at s_j. The time to travel-and-buy from s_i to s_j is $t(s_i, s_j)$. You may assume all travel-and-buy actions represent shortest paths, so there is no faster way to get between s_i and s_j via some other store. The robot begins at your home with no gifts purchased. You want it to buy all the items in as short a time as possible and return home.

For this planning problem, you use a state space where each state is a pair (s, u) where s is the current location and u is the set of unpurchased gifts on your list (so $g \in u$ indicates that gift g has not yet been purchased).

(a) (1 pt) How large is the state space in terms of the quantities defined above?

$M \times 2^N$. You are in one of M places (simple index from 1 to M), and have not purchased some subset of N items (binary vector of size N).

(b) (4 pt) For each of the following heuristics, which apply to states (s, u), circle whether it is admissible, consistent, neither, or both. Assume that the minimum of an empty set is zero.

([neither] / admissible / consistent / both) The shortest time from the current location to any other store: $\min_{s' \neq s} t(s, s')$

(neither / admissible / consistent / [both]) The time to get home from the current location: $t(s, s_1)$

(neither / admissible / consistent / [both]) The shortest time to get to any store selling any unpurchased gift: $\min_{g \in u}(\min_{s':g \in s'} t(s, s'))$

(neither / [admissible] / consistent / both) The shortest time to get home from any store selling any unpurchased gift: $\min_{g \in u}(\min_{s':g \in s'} t(s', s_1))$

([neither] / admissible / consistent / both) The total time to get each unpurchased gift individually: $\sum_{g \in u}(\min_{s':g \in s'} t(s, s'))$

([neither] / admissible / consistent / both) The number of unpurchased gifts times the shortest store-to-store time: $|u|(\min_{s_i, s_j \neq s_i} t(s_i, s_j))$

Remember, a consistent heuristic doesn't decrease from state to state by more than it actually costs to get from state to state. And of course, a heuristic is admissible if it is consistent. If you're confused, remember: the problem defines the minimum of an empty set as 0.

i. This heuristic does not return 0 in the goal state $(s_1, \emptyset)$, since it gives the minimum distance to any store *other than the current one.*

ii. We'll always need to get home from any state; the distance to home from home is 0; and this heuristic does not decrease by more than it costs to get from state to state.

iii. We'll always need to get that last unpurchased item, and taking the min distance store guarantees that we underestimate how much distance we actually have to travel. It is consistent because the heuristic never diminishes by more than what is travelled.

iv. We'll always need to get home from getting the last unpurchased item, and taking the min underestimates the actual requirement. What makes this heuristic inconsistent is that when we visit the last store to pick up the last unfinished item, the value of the heuristic goes to 0. Let's say the graph looks like this: s_3 `--1-->` s_2 `----5----->` s_1, with s_2 containing the last item. From s_3, the heuristic is 5, but from s_2, the heuristic is now 0, meaning that traveling from s_3 to s_2 decreases the heuristic by 5 but the actual cost is only 1.

v. This can overestimate the actual amount of work required.

vi. Same.

תשובה 4.2

You have waited until very late to do your shopping, so you decide to send an swarm of R robot minions to shop in parallel. Each robot moves at the same speed, so the same store-to-store times apply. The problem is now to have all robots start at home, end at home, and for each item to have been bought by at least one robot (you don't have to worry about whether duplicates get bought). Hint: consider that robots may not all arrive at stores in sync.

(c) (4 pt) Give a minimal state space for this search problem (be formal and precise!)

We need the location of each robot at each time. At a given time, a robot can either be at one of M stores, or in any of $(T-1)M$ transition locations, where T is the maximum travel distance between two stores. Thus, the location of each robot takes $(MT)^R$. We also need the set of items purchased (2^N). Therefore, the size of each state is: $(MT)^R \times 2^N$.

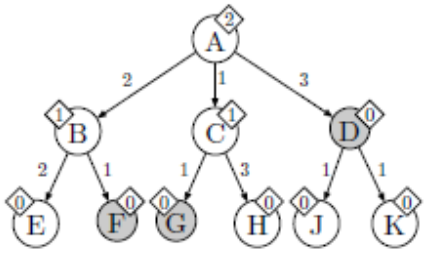
One final task remains: you still must find your younger brother a stuffed Woozle, the hot new children's toy. Unfortunately, no store is guaranteed to stock one. Instead, each store s_i has an initial probability p_i of still having a Woozle available. Moreover, that probability drops exponentially as other buyers scoop them up, so after t time has passed, s_i's probability has dropped to $\beta^t p_i$. You cannot simply try a store repeatedly; once it is out of stock, that store will stay out of stock. Worse, you only have a single robot that can handle this kind of uncertainty! Phrase the problem as a single-agent MDP for planning a search policy for just this one gift (no shopping lists). You receive a single reward of $+1$ upon successfully buying a Woozle, at which point the MDP ends (don't worry about getting home); all other rewards are zeros. You may assume a discount of 1.

(d) (3 pt) Give a minimal state space for this MDP (be formal and precise!)

Which stores have been checked: 2^M
Whether Woozle has been bought: 2
Current time: T.
We may also want to keep track of the current location (M), but since there is no reward for traveling, we don't have to model that aspect of the problem.

Q1. [10 pts] Search: Algorithms

Consider the state space search problem shown to the right. A is the start state and the shaded states are goals. Arrows encode possible state transitions, and numbers by the arrows represent action costs. Note that state transitions are directed; for example, $A \rightarrow B$ is a valid transition, but $B \rightarrow A$ is not. Numbers shown in diamonds are heuristic values that estimate the optimal (minimal) cost from that node to a goal.



For each of the following search algorithms, write down the nodes that are removed from fringe in the course of the search, as well as the final path returned. Because the original problem graph is a tree, the tree and graph versions of these algorithms will do the same thing, and you can use either version of the algorithms to compute your answer.

Assume that the data structure implementations and successor state orderings are all such that *ties are broken alphabetically*. For example, a partial plan $S \rightarrow X \rightarrow A$ would be expanded before $S \rightarrow X \rightarrow B$; similarly, $S \rightarrow A \rightarrow Z$ would be expanded before $S \rightarrow B \rightarrow A$.

(a) [2 pts] **Depth-First Search (ignores costs)**

Nodes removed from fringe:

Path returned:

(b) [2 pts] **Breadth-First Search (ignores costs)**

Nodes removed from fringe:

Path returned:

(c) [2 pts] **Uniform-Cost Search**

Nodes removed from fringe:

Path returned:

(d) [2 pts] **Greedy Search**

Nodes removed from fringe:

Path returned:

(e) [2 pts] **A* Search**

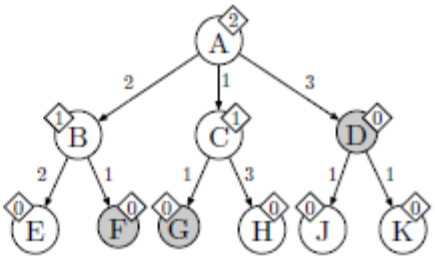
Nodes removed from fringe:

Path returned:

תשובה 5

Q1. [10 pts] Search: Algorithms

Consider the state space search problem shown to the right. A is the start state and the shaded states are goals. Arrows encode possible state transitions, and numbers by the arrows represent action costs. Note that state transitions are directed; for example, $A \rightarrow B$ is a valid transition, but $B \rightarrow A$ is not. Numbers shown in diamonds are heuristic values that estimate the optimal (minimal) cost from that node to a goal.



For each of the following search algorithms, write down the nodes that are removed from fringe in the course of the search, as well as the final path returned. Because the original problem graph is a tree, the tree and graph versions of these algorithms will do the same thing, and you can use either version of the algorithms to compute your answer.

Assume that the data structure implementations and successor state orderings are all such that *ties are broken alphabetically*. For example, a partial plan $S \rightarrow X \rightarrow A$ would be expanded before $S \rightarrow X \rightarrow B$; similarly, $S \rightarrow A \rightarrow Z$ would be expanded before $S \rightarrow B \rightarrow A$.

(a) [2 pts] **Depth-First Search (Ignores costs)**

Nodes removed from fringe: A, B, E, F

Path returned: A, B, F

(b) [2 pts] **Breadth-First Search (Ignores costs)**

Nodes removed from fringe: A, B, C, D

Path returned: A, D

(c) [2 pts] **Uniform-Cost Search**

Nodes removed from fringe: A, C, B, G

Path returned: A, C, G

(d) [2 pts] **Greedy Search**

Nodes removed from fringe: A, D

Path returned: A, D

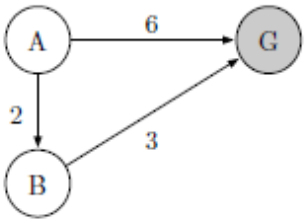
(e) [2 pts] **A* Search**

Nodes removed from fringe: A, C, G

Path returned: A, C, G

Q2. [6 pts] Search: Heuristic Function Properties

For the following questions, consider the search problem shown on the left. It has only three states, and three directed edges. A is the start node and G is the goal node. To the right, four different heuristic functions are defined, numbered I through IV.



	h(A)	h(B)	h(G)
I	4	1	0
II	5	4	0
III	4	3	0
IV	5	2	0

(a) [4 pts] **Admissibility and Consistency**

For each heuristic function, circle whether it is admissible and whether it is consistent with respect to the search problem given above.

	Admissible?		Consistent?	
I	Yes	No	Yes	No
II	Yes	No	Yes	No
III	Yes	No	Yes	No
IV	Yes	No	Yes	No

(b) [2 pts] **Function Domination**

Recall that *domination* has a specific meaning when talking about heuristic functions.

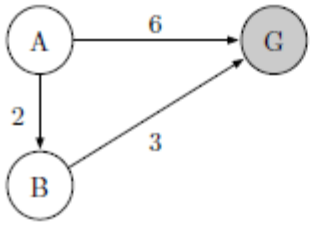
Circle all true statements among the following.

1. Heuristic function III dominates IV.
2. Heuristic function IV dominates III.
3. Heuristic functions III and IV have no dominance relationship.

4. Heuristic function I dominates IV.
5. Heuristic function IV dominates I.
6. Heuristic functions I and IV have no dominance relationship.

Q2. [6 pts] Search: Heuristic Function Properties

For the following questions, consider the search problem shown on the left. It has only three states, and three directed edges. A is the start node and G is the goal node. To the right, four different heuristic functions are defined, numbered I through IV.



	h(A)	h(B)	h(G)
I	4	1	0
II	5	4	0
III	4	3	0
IV	5	2	0

(a) [4 pts] **Admissibility and Consistency**

For each heuristic function, circle whether it is admissible and whether it is consistent with respect to the search problem given above.

	Admissible?		Consistent?	
I	[Yes]	No	Yes	[No]
II	Yes	[No]	Yes	[No]
III	[Yes]	No	[Yes]	No
IV	[Yes]	No	Yes	[No]

II is the only inadmissible heuristic, as it overestimates the cost from B: $h(B) = 4$, when the actual cost to G is 3.

To check whether a heuristic is consistent, ensure that for all paths, $h(N) - h(L) \leq \text{path}(N \to L)$, where N and L stand in for the actual nodes. In this problem, $h(G)$ is always 0, so making sure that the direct paths to the goal ($A \to G$ and $B \to G$) are consistent is the same as making sure that the heuristic is admissible. The path from A to B is a different story.

Heuristic I is not consistent: $h(A) - h(B) = 4 - 1 = 3 \not\leq \text{path}(A \to B) = 2$.
Heuristic III is consistent: $h(A) - h(B) = 4 - 3 = 1 \leq 2$
Heuristic IV is not consistent: $h(A) - h(B) = 5 - 2 = 3 \not\leq 2$

(b) [2 pts] **Function Domination**

Recall that *domination* has a specific meaning when talking about heuristic functions.

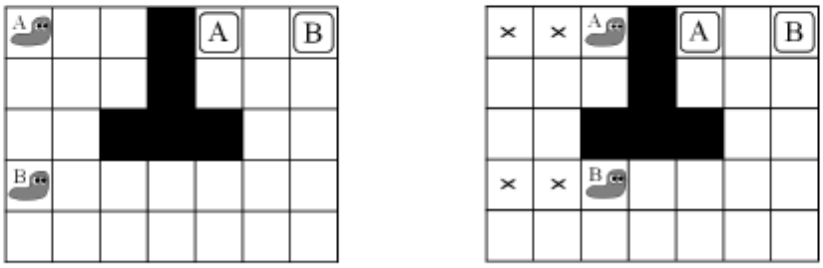
Circle all true statements among the following.

1. Heuristic function III dominates IV.
2. Heuristic function IV dominates III.
3. [Heuristic functions III and IV have no dominance relationship.]
4. Heuristic function I dominates IV.
5. [Heuristic function IV dominates I.]
6. Heuristic functions I and IV have no dominance relationship.

For one heuristic to dominate another, *all* of its values must be greater than or equal to the corresponding values of the other heuristic. Simply make sure that this is the case. If it is not, the two heuristics have no dominance relationship.

Q3. [8 pts] Search: Slugs

You are once again tasked with planning ways to get various insects out of a maze. This time, it's slugs! As shown in the diagram below to the left, two slugs A and B want to exit a maze via their own personal exits. In each time step, both slugs move, though each can choose to either stay in place or move into an adjacent free square. The slugs cannot move into a square that the other slug is moving into. In addition, the slugs leave behind a sticky, poisonous substance and so they cannot move into any square that *either* slug has ever been in. For example, if both slugs move right twice, the maze is as shown in the diagram below to right, with the x squares unpassable to either slug.



You must pose a search problem that will get them to their exits in as few time steps as possible. You may assume that the board is of size N by M; all answers should hold for a general instance, not simply the instance shown above. (You do not need to generalize beyond two slugs.)

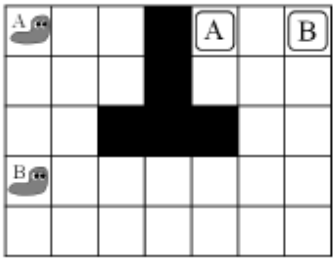
(a) [3 pts] How many states are there in a minimal representation of the space? Justify with a brief description of the components of your state space.

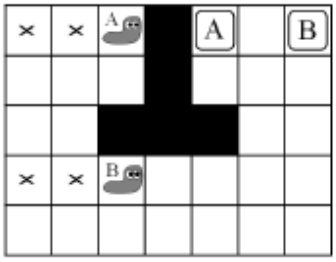
(b) [2 pts] What is the branching factor? Justify with a brief description of the successor function.

(c) [3 pts] Give a non-trivial admissible heuristic for this problem.

Q3. [8 pts] Search: Slugs

You are once again tasked with planning ways to get various insects out of a maze. This time, it's slugs! As shown in the diagram below to the left, two slugs A and B want to exit a maze via their own personal exits. In each time step, both slugs move, though each can choose to either stay in place or move into an adjacent free square. The slugs cannot move into a square that the other slug is moving into. In addition, the slugs leave behind a sticky, poisonous substance and so they cannot move into any square that *either* slug has ever been in. For example, if both slugs move right twice, the maze is as shown in the diagram below to right, with the x squares unpassable to either slug.





You must pose a search problem that will get them to their exits in as few time steps as possible. You may assume that the board is of size N by M; all answers should hold for a general instance, not simply the instance shown above. (You do not need to generalize beyond two slugs.)

(a) [3 pts] How many states are there in a minimal representation of the space? Justify with a brief description of the components of your state space.

$2^{MN}(MN)^2$

The state includes a bit for each of the MN squares, indicating whether the square has been visited (2^{MN} possibilities). It also includes the locations of each slug (MN possibilities for each of the two slugs).

(b) [2 pts] What is the branching factor? Justify with a brief description of the successor function.

$5 \times 5 = 25$ for the first time step, $4 \times 4 = 16$ afterwards.

At the start state each slug has at most five possible next locations (North, South, East, West, Stay). At all future time steps one of those options will certainly be blocked off by the snail's own trail left at the previous time step. Only 4 possible next locations remain.

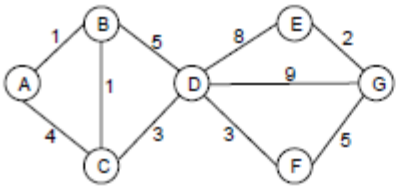
We accepted both 25 and 16 as correct answers.

(c) [3 pts] Give a non-trivial admissible heuristic for this problem.

max(maze distance of bug A to its exit, maze distance of bug B to its exit)

Many other correct answers are possible.

Q1. [9 pts] Search



Node	h_1	h_2
A	9.5	10
B	9	12
C	8	10
D	7	8
E	1.5	1
F	4	4.5
G	0	0

Consider the state space graph shown above. A is the start state and G is the goal state. The costs for each edge are shown on the graph. Each edge can be traversed in both directions. Note that the heuristic h_1 is consistent but the heuristic h_2 is not consistent.

(a) [4 pts] **Possible paths returned**

For each of the following graph search strategies (*do not answer for tree search*), mark which, if any, of the listed paths it could return. Note that for some search strategies the specific path returned might depend on tie-breaking behavior. In any such cases, make sure to mark *all* paths that could be returned under some tie-breaking scheme.

Search Algorithm	**A-B-D-G**	**A-C-D-G**	**A-B-C-D-F-G**
Depth first search			
Breadth first search			
Uniform cost search			
A* search with heuristic h_1			
A* search with heuristic h_2			

(b) Heuristic function properties

Suppose you are completing the new heuristic function h_3 shown below. All the values are fixed except $h_3(B)$.

Node	A	B	C	D	E	F	G
h_3	10	?	9	7	1.5	4.5	0

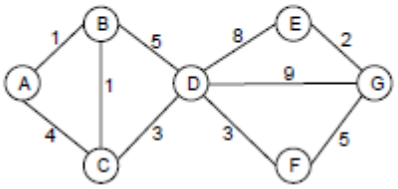
For each of the following conditions, write the set of values that are possible for $h_3(B)$. For example, to denote all non-negative numbers, write $[0, \infty]$, to denote the empty set, write $\varnothing$, and so on.

(i) [1 pt] **What values of $h_3(B)$ make h_3 admissible?**

(ii) [2 pts] **What values of $h_3(B)$ make h_3 consistent?**

(iii) [2 pts] **What values of $h_3(B)$ will cause A* graph search to expand node A, then node C, then node B, then node D in order?**

Q1. [9 pts] Search



Node	h_1	h_2
A	9.5	10
B	9	12
C	8	10
D	7	8
E	1.5	1
F	4	4.5
G	0	0

Consider the state space graph shown above. A is the start state and G is the goal state. The costs for each edge are shown on the graph. Each edge can be traversed in both directions. Note that the heuristic h_1 is consistent but the heuristic h_2 is not consistent.

(a) [4 pts] **Possible paths returned**

For each of the following graph search strategies (*do not answer for tree search*), mark which, if any, of the listed paths it could return. Note that for some search strategies the specific path returned might depend on tie-breaking behavior. In any such cases, make sure to mark *all* paths that could be returned under some tie-breaking scheme.

Search Algorithm	**A-B-D-G**	**A-C-D-G**	**A-B-C-D-F-G**
Depth first search	x	x	x
Breadth first search	x	x	
Uniform cost search			x
A* search with heuristic h_1			x
A* search with heuristic h_2			x

The return paths depend on tie-breaking behaviors so any possible path has to be marked. DFS can return any path. BFS will return all the shallowest paths, i.e. A-B-D-G and A-C-D-G. A-B-C-D-F-G is the optimal path for this problem, so that UCS and A* using consistent heuristic h_1 will return that path. Although, h_2 is not consistent, it will also return this path.

(b) Heuristic function properties

Suppose you are completing the new heuristic function h_3 shown below. All the values are fixed except $h_3(B)$.

Node	A	B	C	D	E	F	G
h_3	10	?	9	7	1.5	4.5	0

For each of the following conditions, write the set of values that are possible for $h_3(B)$. For example, to denote all non-negative numbers, write $[0, \infty]$, to denote the empty set, write $\varnothing$, and so on.

(i) [1 pt] **What values of $h_3(B)$ make h_3 admissible?**

To make h_3 admissible, $h_3(B)$ has to be less than or equal to the actual optimal cost from B to goal G, which is the cost of path B-C-D-F-G, i.e. 12. The answer is $0 \leq h_3(B) \leq 12$

Filling in the numbers shows this results in the condition: $9 \leq h_3(B) \leq 10$

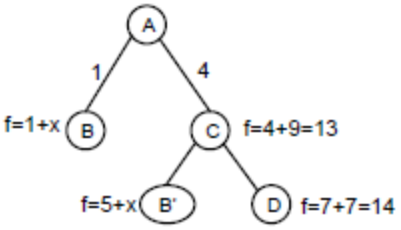
(ii) [2 pts] **What values of $h_3(B)$ will cause A* graph search to expand node A, then node C, then node B, then node D in order?**

The A* search tree using heuristic h_3 is on the right. In order to make A* graph search expand node A, then node C, then node B, suppose $h_3(B) = x$, we need

$1 + x > 13$

$5 + x < 14$ (*expand B'*) *or* $1 + x < 14$ (*expand B*)

so we can get $12 < h_3(B) < 13$



Q4. [8 pts] Search

(a) [4 pts] The following implementation of graph search may be incorrect. Circle all the problems with the code.

```
function GRAPH-SEARCH(problem, fringe)
    closed ← an empty set,
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop
        if fringe is empty then
            return failure
        end if
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem,STATE[node]) then
            return node
        end if
        ADD STATE[node] TO closed
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end loop
end function
```

1. Nodes may be expanded twice.
2. The algorithm is no longer complete.
3. The algorithm could return an incorrect solution.
4. None of the above.

(b) [4 pts] The following implementation of A* graph search may be incorrect. You may assume that the algorithm is being run with a consistent heuristic. Circle all the problems with the code.

```
function A*-SEARCH(problem, fringe)
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop
        if fringe is empty then
            return failure
        end if
        node ← REMOVE-FRONT(fringe)
        if STATE[node] IS NOT IN closed then
            ADD STATE[node] TO closed
            for successor IN GETSUCCESSORS(problem, STATE[node]) do
                fringe ← INSERT(MAKE-NODE(successor), fringe)
                if GOAL-TEST(problem,successor) then
                    return successor
                end if
            end for
        end if
    end loop
end function
```

1. Nodes may be expanded twice.
2. The algorithm is no longer complete.
3. The algorithm could return an incorrect solution.
4. None of the above.

Q4. [8 pts] Search

(a) [4 pts] The following implementation of graph search may be incorrect. Circle all the problems with the code.

```
function GRAPH-SEARCH(problem, fringe)
    closed ← an empty set,
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop
        if fringe is empty then
            return failure
        end if
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem,STATE[node]) then
            return node
        end if
        ADD STATE[node] TO closed
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end loop
end function
```

1. Nodes may be expanded twice.
2. The algorithm is no longer complete.
3. The algorithm could return an incorrect solution.
4. None of the above.

The stated algorithm is equivalent to tree search. In graph search, nodes added to the "closed" list should not be expanded again. Since this algorithm does not do that, it can get stuck in a loop and that is why it is not complete.

(b) [4 pts] The following implementation of A* graph search may be incorrect. You may assume that the algorithm is being run with a consistent heuristic. Circle all the problems with the code.

```
function A*-SEARCH(problem, fringe)
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop
        if fringe is empty then
            return failure
        end if
        node ← REMOVE-FRONT(fringe)
        if STATE[node] IS NOT IN closed then
            ADD STATE[node] TO closed
            for successor IN GETSUCCESSORS(problem, STATE[node]) do
                fringe ← INSERT(MAKE-NODE(successor), fringe)
                if GOAL-TEST(problem,successor) then
                    return successor
                end if
            end for
        end if
    end loop
end function
```

1. Nodes may be expanded twice.
2. The algorithm is no longer complete.
3. The algorithm could return an incorrect solution.
4. None of the above.

The stated algorithm expands fewer nodes to find a goal, but it does not always find the optimal goal in terms of cost. Note that "incorrect" means that it "not optimal" here.

Q7. [33 pts] Short Answer

Each true/false question is worth 1 point. Leaving a question blank is worth 0 points. **Answering incorrectly is worth −1 point.**

(a) Assume we are running A^* graph search with a consistent heuristic h. Assume the optimal cost path to reach a goal has a cost c^*. Then we have that

(i) [*true* or *false*] All nodes n reachable from the start state satisfying $g(n) < c^*$ will be expanded during the search.

(ii) [*true* or *false*] All nodes n reachable from the start state satisfying $f(n) = g(n) + h(n) < c^*$ will be expanded during the search.

(iii) [*true* or *false*] All nodes n reachable from the start state satisfying $h(n) < c^*$ will be expanded during the search.

(b) Running A^* *graph search* with an inconsistent heuristic can lead to suboptimal solutions. Consider the following modification to A^* graph search: replace the closed list with a cost-sensitive closed list, which stores the f-cost of the node along with the state ($f(n) = g(n) + h(n)$).

Whenever the search considers expanding a node, it first verifies whether the node's state is in the cost-senstive closed list and only expands it if either (a) the node's state is not in the cost-sensitive closed list, or (b) the node's state is in the cost-sensitive closed list with a higher f-cost than the f-cost for the node currently considered for expansion.

If a node is expanded because it meets criterion (a), its state and f-cost get added to the cost-sensitive closed list; if it gets expanded because it meets criterion (b), the cost associated with the node's state gets replaced by the current node's f-cost. Which of the following statements are true about the proposed search procedure?

(i) [*true* or *false*] The described search procedure finds an optimal solution if h is admissible.

(ii) [*true* or *false*] The described search procedure finds an optimal solution if h is consistent.

(iii) [*true* or *false*] Assuming h is admissible (but possibly inconsistent), the described search procedure will expand no more nodes than A^* tree search.

(iv) [*true* or *false*] Assuming h is consistent, the described search procedure will expand no more nodes than A^* graph search.

(c) Let H_1 and H_2 both be admissible heuristics.

(i) [*true* or *false*] $\max(H_1, H_2)$ is necessarily admissible

(ii) [*true* or *false*] $\min(H_1, H_2)$ is necessarily admissible

(iii) [*true* or *false*] $(H_1 + H_2)/2$ is necessarily admissible

(iv) [*true* or *false*] $\max(H_1, H_2)$ is necessarily consistent

(d) Let H_1 be an admissible heuristic, and let H_2 be an inadmissible heuristic.

(i) [*true* or *false*] $\max(H_1, H_2)$ is necessarily admissible

(ii) [*true* or *false*] $\min(H_1, H_2)$ is necessarily admissible

(iii) [*true* or *false*] $(H_1 + H_2)/2$ is necessarily admissible

(iv) [*true* or *false*] $\max(H_1, H_2)$ is necessarily consistent

Q7. [33 pts] Short Answer

Each true/false question is worth 1 point. Leaving a question blank is worth 0 points. **Answering incorrectly is worth −1 point.**

(a) Assume we are running A^* graph search with a consistent heuristic h. Assume the optimal cost path to reach a goal has a cost c^*. Then we have that

(i) [*true* or *false*] All nodes n reachable from the start state satisfying $g(n) < c^*$ will be expanded during the search.

(ii) [*true* or *false*] All nodes n reachable from the start state satisfying $f(n) = g(n) + h(n) < c^*$ will be expanded during the search.

(iii) [*true* or *false*] All nodes n reachable from the start state satisfying $h(n) < c^*$ will be expanded during the search.

(b) Running A^* *graph search* with an inconsistent heuristic can lead to suboptimal solutions. Consider the following modification to A^* graph search: replace the closed list with a cost-sensitive closed list, which stores the f-cost of the node along with the state ($f(n) = g(n) + h(n)$).

Whenever the search considers expanding a node, it first verifies whether the node's state is in the cost-senstive closed list and only expands it if either (a) the node's state is not in the cost-sensitive closed list, or (b) the node's state is in the cost-sensitive closed list with a higher f-cost than the f-cost for the node currently considered for expansion.

If a node is expanded because it meets criterion (a), its state and f-cost get added to the cost-sensitive closed list; if it gets expanded because it meets criterion (b), the cost associated with the node's state gets replaced by the current node's f-cost. Which of the following statements are true about the proposed search procedure?

(i) [*true* or *false*] The described search procedure finds an optimal solution if h is admissible.

(ii) [*true* or *false*] The described search procedure finds an optimal solution if h is consistent.

(iii) [*true* or *false*] Assuming h is admissible (but possibly inconsistent), the described search procedure will expand no more nodes than A^* tree search.

(iv) [*true* or *false*] Assuming h is consistent, the described search procedure will expand no more nodes than A^* graph search.

(c) Let H_1 and H_2 both be admissible heuristics.

(i) [*true* or *false*] $\max(H_1, H_2)$ is necessarily admissible

(ii) [*true* or *false*] $\min(H_1, H_2)$ is necessarily admissible

(iii) [*true* or *false*] $(H_1 + H_2)/2$ is necessarily admissible

(iv) [*true* or *false*] $\max(H_1, H_2)$ is necessarily consistent

(d) Let H_1 be an admissible heuristic, and let H_2 be an inadmissible heuristic.

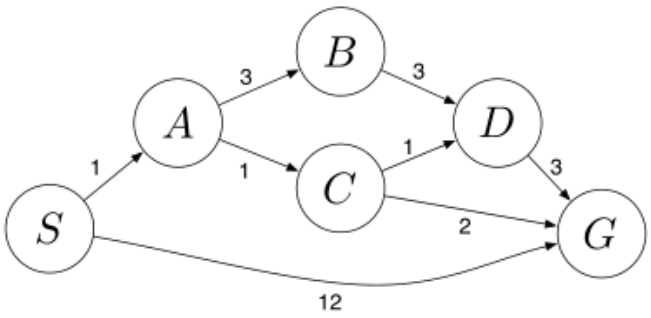
(i) [*true* or *false*] $\max(H_1, H_2)$ is necessarily admissible

(ii) [*true* or *false*] $\min(H_1, H_2)$ is necessarily admissible

(iii) [*true* or *false*] $(H_1 + H_2)/2$ is necessarily admissible

(iv) [*true* or *false*] $\max(H_1, H_2)$ is necessarily consistent

1. (12 points) Search



Answer the following questions about the search problem shown above. Break any ties alphabetically. For the questions that ask for a path, please give your answers in the form '$S - A - D - G$.'

(a) (2 pt) What path would breadth-first graph search return for this search problem?

(b) (2 pt) What path would uniform cost graph search return for this search problem?

(c) (2 pt) What path would depth-first graph search return for this search problem?

(d) (2 pt) What path would A* graph search, using a consistent heuristic, return for this search problem?

(e) (4 pt) Consider the heuristics for this problem shown in the table below.

State	h_1	h_2
S	5	4
A	3	2
B	6	6
C	2	1
D	3	3
G	0	0

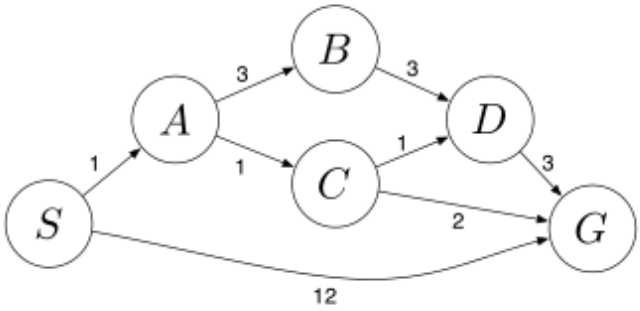
i. (1 pt) Is h_1 admissible? **Yes** **No**

ii. (1 pt) Is h_1 consistent? **Yes** **No**

iii. (1 pt) Is h_2 admissible? **Yes** **No**

iv. (1 pt) Is h_2 consistent? **Yes** **No**

1. (12 points) Search



Answer the following questions about the search problem shown above. Break any ties alphabetically. For the questions that ask for a path, please give your answers in the form '$S - A - D - G$.'

(a) (2 pt) What path would breadth-first graph search return for this search problem?

$S - G$

(b) (2 pt) What path would uniform cost graph search return for this search problem?

$S - A - C - G$

(c) (2 pt) What path would depth-first graph search return for this search problem?

$S - A - B - D - G$

(d) (2 pt) What path would A* graph search, using a consistent heuristic, return for this search problem?

$S - A - C - G$

(e) (4 pt) Consider the heuristics for this problem shown in the table below.

State	h_1	h_2
S	5	4
A	3	2
B	6	6
C	2	1
D	3	3
G	0	0

i. (1 pt) Is h_1 admissible? **Yes** **No**

ii. (1 pt) Is h_1 consistent? **Yes** **No**

iii. (1 pt) Is h_2 admissible? **Yes** **No**

iv. (1 pt) Is h_2 consistent? **Yes** **No**

An admissible heuristic must underestimate or be equal to the true cost.
A consistent heuristic must satisfy $h(N) - h(L) \leq \text{path}(N \to L)$ for all paths and nodes N and L.

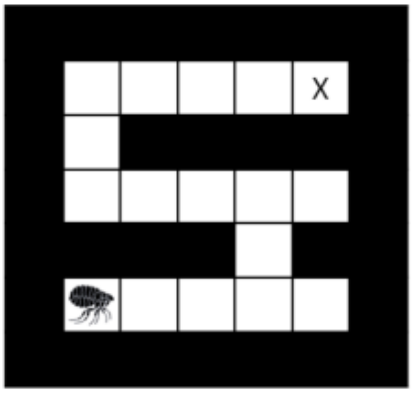
$h1$ overestimates the cost $S \to G$ as 5 when it is 4, so it is inadmissible.
$h1$ is not consistent because $h(S) - h(A) \leq \text{path}(S \to A)$ is violated as $5 - 3 \leq 1$.
$h2$ does not overestimate costs and is admissible.
$h2$ is not consistent because $h(S) - h(A) \leq \text{path}(S \to A)$ is violated as $4 - 2 \leq 1$.

2. (12 points) Hive Minds: Redux

Let's revisit our bug friends from assignment 2. To recap, you control one or more insects in a rectangular maze-like environment with dimensions $M \times N$, as shown in the figures below. At each time step, an insect can move North, East, South, or West (but not diagonally) 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 (as denoted by the black squares), but the map is known.

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

(a) (6 pt) The Flea



You now control a single flea as shown in the maze above, which must reach a designated target location X. However, in addition to moving along the maze as usual, your flea can jump on top of the walls. When on a wall, the flea can walk along the top of the wall as it would when in the maze. It can also jump off of the wall, back into the maze. Jumping onto the wall has a cost of 2, while all other actions (including jumping back into the maze) have a cost of 1. Note that the flea can only jump onto walls that are in adjacent squares (either north, south, west, or east of the flea).

i. (2 pt) Give a *minimal* state representation for the above search problem.

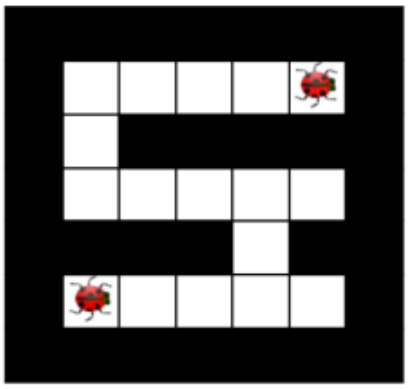
ii. (2 pt) Give the size of the state space for this search problem.

iii. (2 pt) Is the following heuristic admissible? **Yes** **No**

$$h_{\text{flea}} = \text{the Manhattan distance from the flea to the goal.}$$

If it is *not* admissible, provide a nontrivial admissible heuristic in the space below.

(b) (6 pt) Long Lost Bug Friends



You now control a pair of long lost bug friends. You know the maze, but you do not have any information about which square each bug starts in. You want to help the bugs reunite. You must pose a search problem whose solution is an all-purpose sequence of actions such that, after executing those actions, both bugs will be on the same square, regardless of their initial positions. Any square will do, as the bugs have no goal in mind other than to see each other once again. Both bugs execute the actions mindlessly and do not know whether their moves succeed; if they use an action which would move them in a blocked direction, they will stay where they are. Unlike the flea in the previous question, bugs *cannot* jump onto walls. Both bugs can move in each time step. Every time step that passes has a cost of one.

i. (2 pt) Give a *minimal* state representation for the above search problem.

ii. (2 pt) Give the size of the state space for this search problem.

iii. (2 pt) Give a nontrivial admissible heuristic for this search problem.

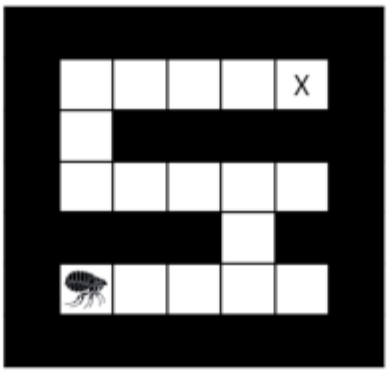
תשובה 12.1

2. (12 points) Hive Minds: Redux

Let's revisit our bug friends from assignment 2. To recap, you control one or more insects in a rectangular maze-like environment with dimensions $M \times N$, as shown in the figures below. At each time step, an insect can move North, East, South, or West (but not diagonally) 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 (as denoted by the black squares), but the map is known.

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

(a) (6 pt) The Flea



You now control a single flea as shown in the maze above, which must reach a designated target location X. However, in addition to moving along the maze as usual, your flea can jump on top of the walls. When on a wall, the flea can walk along the top of the wall as it would when in the maze. It can also jump off of the wall, back into the maze. Jumping onto the wall has a cost of 2, while all other actions (including jumping back into the maze) have a cost of 1. Note that the flea can only jump onto walls that are in adjacent squares (either north, south, west, or east of the flea).

i. (2 pt) Give a *minimal* state representation for the above search problem.

The state is the location of the flea as an (x, y) coordinate. The map is known, including walls and the goal, and the actions of the flea depend only on its location.

ii. (2 pt) Give the size of the state space for this search problem.

The state space is $M \times N$. The flea can occupy any free location in a given maze, and any square might be free or a wall in a maze, so any of the $M \times N$ locations are possible.

iii. (2 pt) Is the following heuristic admissible? **Yes** **No**

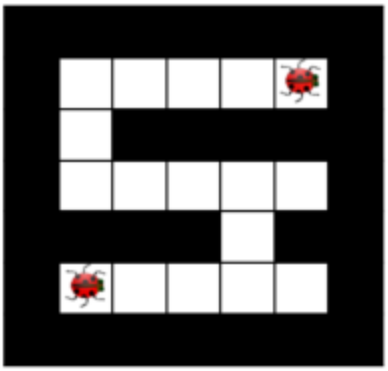
$$h_{\text{flea}} = \text{the Manhattan distance from the flea to the goal.}$$

It is yielded by the relaxed problem where the flea passes through walls. It never overestimates because 1. a wall can never decrease the length of a path to the goal and 2. the cost of the flea jumping up a wall (2) is higher than the cost of moving.

If it is *not* admissible, provide a nontrivial admissible heuristic in the space below.

תשובה 12.2

(b) (6 pt) Long Lost Bug Friends



You now control a pair of long lost bug friends. You know the maze, but you do not have any information about which square each bug starts in. You want to help the bugs reunite. You must pose a search problem whose solution is an all-purpose sequence of actions such that, after executing those actions, both bugs will be on the same square, regardless of their initial positions. Any square will do, as the bugs have no goal in mind other than to see each other once again. Both bugs execute the actions mindlessly and do not know whether their moves succeed; if they use an action which would move them in a blocked direction, they will stay where they are. Unlike the flea in the previous question, bugs *cannot* jump onto walls. Both bugs can move in each time step. Every time step that passes has a cost of one.

i. (2 pt) Give a *minimal* state representation for the above search problem.

The state is a list of boolean variables, one for each position in the maze, which marks whether the position could contain a bug. There is no need to separately keep track of the bugs since their starting positions are not known; to ensure they meet only a single square must be possible for both.

ii. (2 pt) Give the size of the state space for this search problem.

The size is 2^{MN} since every of the $M \times N$ possible maze positions must be considered and every position has a boolean variable. A full state is the product of the individual position states, which are binary valued for the base of 2.

iii. (2 pt) Give a nontrivial admissible heuristic for this search problem.

h_{friends} = the maximum Manhattan distance of all possible pairs of points the bugs can be in.
This is never an overestimate because the number of steps to join the insects with certainty is at least the shortest path (with no obstacles) between their farthest possible locations from one another. Remember that the starting locations are unknown so the bugs cannot simply be controlled to move toward each other.

3. (12 points) A* Graph Search

```
function A* GRAPH SEARCH(problem)
    fringe ← an empty priority queue
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    closed ← an empty set
    ADD INITIAL-STATE[problem] to closed
    loop
        if fringe is empty then
            return failure
        end if
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then
            return node
        end if
        for successor in GETSUCCESSORS(problem, STATE[node]) do
            if successor not in closed then
                ADD successor to closed
                fringe ← INSERT(MAKE-SUCCESSOR-NODE(successor, node), fringe)
            end if
        end for
    end loop
end function
```

The above implementation of A graph search may be incorrect!* In the list below circle all of the problems that the bugs may cause when executing graph search and justify your answer. Note that the fringe is a priority queue. Nodes are inserted into the fringe using the standard key for A*, namely $f = g + h$. h is a consistent heuristic.

(a) The GetSuccessors function could be called multiple times on the same state.

(b) The algorithm is no longer complete.

(c) The algorithm could return a suboptimal solution.

(d) The implementation is incorrect, but none of the above problems will be caused.

(e) The implementation is correct.

To receive any credit you must briefly justify your answer in space below.

3. (12 points) A* Graph Search

```
function A* Graph Search(problem)
    fringe ← an empty priority queue
    fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
    closed ← an empty set
    Add Initial-State[problem] to closed
    loop
        if fringe is empty then
            return failure
        end if
        node ← Remove-Front(fringe)
        if Goal-Test(problem, State[node]) then
            return node
        end if
        for successor in GetSuccessors(problem, State[node]) do
            if successor not in closed then
                Add successor to closed
                fringe ← Insert(Make-Successor-Node(successor, node), fringe)
            end if
        end for
    end loop
end function
```

The above implementation of A graph search may be incorrect!* In the list below circle all of the problems that the bugs may cause when executing graph search and justify your answer. Note that the fringe is a priority queue. Nodes are inserted into the fringe using the standard key for A*, namely $f = g + h$. h is a consistent heuristic.

(a) The GetSuccessors function could be called multiple times on the same state.

(b) The algorithm is no longer complete.

(c) The algorithm could return a suboptimal solution.

(d) The implementation is incorrect, but none of the above problems will be caused.

(e) The implementation is correct.

To receive any credit you must briefly justify your answer in space below.

The bug is the insertion of *successor* into *closed* at time of insertion of a node into the fringe, rather than at the time that node gets popped from the fringe. As a consequence of this bug, the first path encountered to a state will put that state in the closed list. This can cause suboptimality as we only have the guarantee that a state has been reached optimally once a node reaching it gets popped off the fringe.

(a) is False as when a node that reaches a state s is placed in the fringe, that state s is also put on the closed list. This means never in the future can a node be placed on the fringe that ends in that same state s, and hence the same state s can be the argument to GetSuccessors at most once.

(b) is False. A* tree search is complete. The difference is that the above algorithm will cut off parts of the tree search whenever it has placed a node on the fringe in the past that ends in the same state. So compared to tree search we only lose copies of subtrees that we are covering. Hence the above algorithm is complete.

(c) is True. See explanation at beginning of solution.

(d) is False.

(e) is False.

(a) **(7 pt)** Consider a graph search problem where for every action, the cost is at least ϵ, with $\epsilon > 0$. Assume the used heuristic is consistent.

True **False** Depth-first graph search is guaranteed to return an optimal solution.
True **False** Breadth-first graph search is guaranteed to return an optimal solution.
True **False** Uniform-cost graph search is guaranteed to return an optimal solution.
True **False** Greedy graph search is guaranteed to return an optimal solution.
True **False** A* graph search is guaranteed to return an optimal solution.
True **False** A* graph search is guaranteed to expand no more nodes than depth-first graph search.
True **False** A* graph search is guaranteed to expand no more nodes than uniform-cost graph search.

(b) **(2 pt)** Iterative deepening is sometimes used as an alternative to breadth first search. Give one advantage of iterative deepening over BFS, and give one disadvantage of iterative deepening as compared with BFS. Be concise and specific!

(c) **(2 pt)** Consider two different A* heuristics, $h_1(s)$ and $h_2(s)$, that are each admissible. Now, combine the two heuristics into a single heuristic, using some (not yet specified) function g. Give the choice for g that will result in A* expanding a minimal number of nodes while still guaranteeing admissibility. Your answer should be a heuristic function of the form $g(h_1(s), h_2(s))$.

(d) **(3 pt)** Let $h_1(s)$ be an admissible A* heuristic. Let $h_2(s) = 2h_1(s)$. Then:

True **False** The solution found by A* tree search with h_2 is guaranteed to be an optimal solution.
True **False** The solution found by A* tree search with h_2 is guaranteed to have a cost at most twice as much as the optimal path.
True **False** The solution found by A* graph search with h_2 is guaranteed to be an optimal solution.

non-optimal solution when popped is its f-cost. The prefix of the optimal path to the goal has an f-cost of $g + h_0 = g + 2h_1 \leq 2(g + h_1) \leq 2C^*$, with C^* the optimal cost to the goal. Hence we have that $\bar{g} \leq 2C^*$ and the found path is at most twice as long as the optimal path.

True **False** The solution found by A* graph search with h_2 is guaranteed to be an optimal solution.
False. h_2 is not guaranteed to be admissible and graph search further requires consistency for optimality.
same cost, which is not the case here.

True **False** Uniform-cost graph search is guaranteed to return an optimal solution.
True. UCS expands paths in order of least total cost so that the optimal solution is found.

True **False** Greedy graph search is guaranteed to return an optimal solution.
False. Greedy search makes no guarantees of optimality. It relies solely on the heuristic and not the true cost.

True **False** A* graph search is guaranteed to return an optimal solution.
True, since the heuristic is consistent in this case.

True **False** A* graph search is guaranteed to expand no more nodes than depth-first graph search.
False. Depth-first graph search could, for example, go directly to a sub-optimal solution.

True **False** A* graph search is guaranteed to expand no more nodes than uniform-cost graph search.
True. The heuristic could help to guide the search and reduce the number of nodes expanded. In the extreme case where the heuristic function returns zero for every state, A* and UCS will expand the same number of nodes. In any case, A* with a consistent heuristic will never expand more nodes than UCS.

(b) **(2 pt)** Iterative deepening is sometimes used as an alternative to breadth first search. Give one advantage of iterative deepening over BFS, and give one disadvantage of iterative deepening as compared with BFS. Be concise and specific! Advantage: iterative deepening requires less memory (limited to the current depth). Disadvantage: iterative deepening repeats computations and therefore can require additional run time.

(c) **(2 pt)** Consider two different A* heuristics, $h_1(s)$ and $h_2(s)$, that are each admissible. Now, combine the two heuristics into a single heuristic, using some (not yet specified) function g. Give the choice for g that will result in A* expanding a minimal number of nodes while still guaranteeing admissibility. Your answer should be a heuristic function of the form $g(h_1(s), h_2(s))$.
$g = \max(h_1(s), h_2(s))$. Consider the true cost $h^*(s)$. For admissibility, both $h_1(s) \leq h^*(s)$ and $h_2(s) \leq h^*(s)$, and their max can be no larger. $h^*(s)$ leads to expanding the minimal number of nodes: in particular it expands the nodes of an optimal path to a goal and no more. g is the closest heuristic to h^* of guaranteed admissibility by h_1 and h_2 and so expands a minimal number of nodes for functions over h_1, h_2.

(d) **(3 pt)** Let $h_1(s)$ be an admissible A* heuristic. Let $h_2(s) = 2h_1(s)$. Then:

True **False** The solution found by A* tree search with h_2 is guaranteed to be an optimal solution.
False. h_2 is not guaranteed to be admissible since only one side of the admissibility inequality is doubled.

True **False** The solution found by A* tree search with h_2 is guaranteed to have a cost at most twice as much as the optimal path.
True. In A* tree search we always have that as long as the optimal path to the goal has not been found, a prefix of this optimal path has to be on the fringe. Hence, if a non-optimal solution is found, then at time of popping the non-optimal path from the fringe, a path that is a prefix of the optimal path to the goal is sitting on the fringe. The cost $\bar{g}$ of a

Q7. [28 pts] A* Search: Parallel Node Expansion

Recall that A* graph search can be implemented in pseudo-code as follows:

```
1: function A*-Graph-Search(problem, fringe)
2:     closed ← an empty set
3:     fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
4:     loop do
5:         if fringe is empty then return failure
6:         node ← Remove-Front(fringe)
7:         if Goal-Test(problem, State[node]) then return node
8:         if State[node] is not in closed then
9:             add State[node] to closed
10:            child-nodes ← Expand(node, problem)
11:            fringe ← Insert-All(child-nodes, fringe)
```

You notice that your successor function (Expand) takes a very long time to compute and the duration can vary a lot from node to node, so you try to speed things up using parallelization. You come up with A*-Parallel, which uses a "master" thread which runs A*-Parallel and a set of $n \geq 1$ "workers", which are separate threads that execute the function Worker-Expand which performs a node expansion and writes results back to a shared fringe. The master thread issues non-blocking calls to Worker-Expand, which dispatches a given worker to begin expanding a particular node.[1] The Wait function called from the master thread pauses execution (sleeps) in the master thread for a small period of time, e.g., 20 ms. The *fringe* for these functions is in shared memory and is always passed by reference. Assume the shared *fringe* object can be safely modified from multiple threads.

A*-Parallel is best thought of as a modification of A*-Graph-Search. In lines 5-9, A*-Parallel first waits for some worker to be free, then (if needed) waits until the fringe is non-empty so the worker can be assigned the next node to be expanded from the fringe. If all workers have become idle while the fringe is still empty, this means no insertion in the fringe will happen anymore, which means there is no path to a goal so the search returns failure. (This corresponds to line 5 of A*-Graph-Search). Line 16 in A*-Parallel assigns an idle worker thread to execute Worker-Expand in lines 17-19. (This corresponds to lines 10-11 of A*-Graph-Search.) Finally, lines 11-13 in the A*-Parallel, corresponding to line 7 in A*-Graph-Search is where your work begins. Because there are workers acting in parallel it is not a simple task to determine when a goal can be returned: perhaps one of the busy workers was just about to add a really good goal node into the fringe.

```
1: function A*-Parallel(problem, fringe, workers)
2:     closed ← an empty set
3:     fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
4:     loop do
5:         while All-Busy(workers) do Wait
6:         while fringe is empty do
7:             if All-Idle(workers) and fringe is empty then
8:                 return failure
9:             else Wait
10:        node ← Remove-Front(fringe)
11:        if Goal-Test(problem, State[node]) then
12:            if Should-Return(node, workers, fringe) then
13:                return node
14:        if State[node] is not in closed then
15:            add State[node] to closed
16:            Get-Idle-Worker(workers).Worker-Expand(node, problem, fringe)

17: function Worker-Expand(node, problem, fringe)
18:     child-nodes ← Expand(node, problem)
19:     fringe ← Insert-All(child-nodes, fringe)
```

[1] A non-blocking call means that the master thread continues executing its code without waiting for the worker to return from the call to the worker.

שאלה 15.2

Consider the following possible implementations of the SHOULD-RETURN function called before returning a goal node in A*-PARALLEL:

I
```
function SHOULD-RETURN(node, workers, fringe)
    return true
```

II
```
function SHOULD-RETURN(node, workers, fringe)
    return ALL-IDLE(workers)
```

III
```
function SHOULD-RETURN(node, workers, fringe)
    fringe ← INSERT(node, fringe)
    return ALL-IDLE(workers)
```

IV
```
function SHOULD-RETURN(node, workers, fringe)
    while not ALL-IDLE(workers) do WAIT
    fringe ← INSERT(node, fringe)
    return F-COST[node] == F-COST[GET-FRONT(fringe)]
```

For each of these, indicate whether it results in a complete search algorithm, and whether it results in an optimal search algorithm. Give a brief justification for your answer (answers without a justification will receive zero credit). Assume that the state space is finite, and the heuristic used is consistent.

(a) **(i)** [4 pts] **Implementation I**

Optimal? Yes / No. Justify your answer:

Complete? Yes / No. Justify your answer:

(ii) [4 pts] **Implementation II**

Optimal? Yes / No. Justify your answer:

Complete? Yes / No. Justify your answer:

(iii) [4 pts] **Implementation III**

Optimal? Yes / No. Justify your answer:

Complete? Yes / No. Justify your answer:

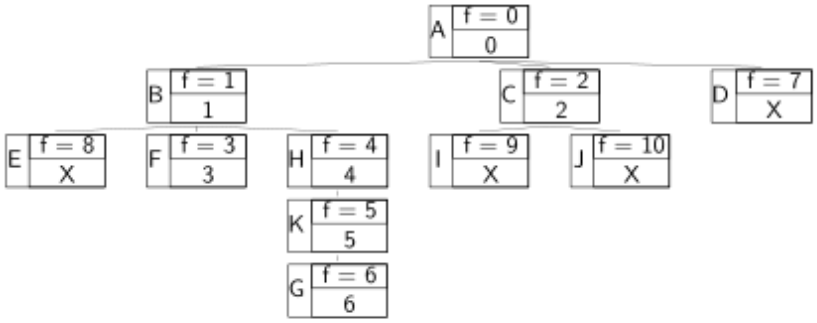
(iv) [4 pts] **Implementation IV**

Optimal? Yes / No. Justify your answer:

Complete? Yes / No. Justify your answer:

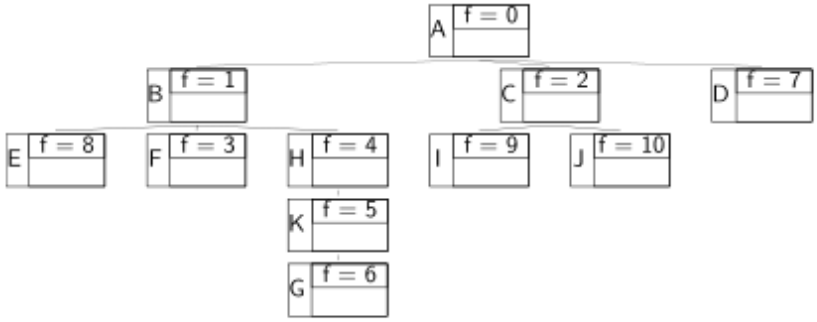
(b) Suppose we run A*-Parallel with implementation **IV** of the Should-Return function. We now make a new, additional assumption about execution time: Each worker takes exactly one time step to expand a node and push all of the successor nodes onto the fringe, independent of the number of successors (including if there are zero successors). All other computation is considered instantaneous for our time bookkeeping in this question.

A*-Parallel with the above timing properties was run with a single (1) worker on a search problem with the search tree in the diagram below. Each node is drawn with the state at the left, the f-value at the top-right ($f(n) = g(n) + h(n)$), and the time step on which a worker expanded that node at the bottom-right, with an 'X' if that node was not expanded. G is the unique goal node. In the diagram below, we can see that the start node A was expanded by the worker at time step 0, then node B was expanded at time step 1, node C was expanded at time step 2, node F was expanded at time step 3, node H was expanded at time step 4, node K was expanded at time step 5, and node G was expanded at time step 6. Nodes D, E, I, J were never expanded.

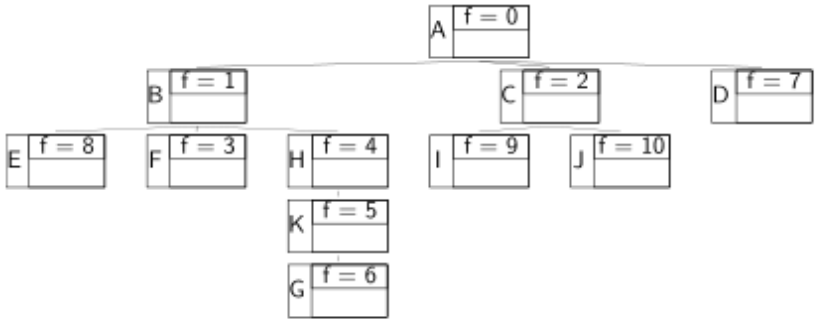


In this question you'll complete similar diagrams by filling in the node expansion times for the case of two and three workers. Note that now multiple nodes can (and typically will!) be expanded at any given time.

(i) [6 pts] Complete the node expansion times for the case of two workers and fill in an 'X' for any node that is not expanded.



(ii) [6 pts] Complete the node expansion times for the case of three workers and fill in an 'X' for any node that is not expanded.



תשובה 15.1

Consider the following possible implementations of the SHOULD-RETURN function called before returning a goal node in A*-PARALLEL:

I
```
function SHOULD-RETURN(node, workers, fringe)
    return true
```

II
```
function SHOULD-RETURN(node, workers, fringe)
    return ALL-IDLE(workers)
```

III
```
function SHOULD-RETURN(node, workers, fringe)
    fringe ← INSERT(node, fringe)
    return ALL-IDLE(workers)
```

IV
```
function SHOULD-RETURN(node, workers, fringe)
    while not ALL-IDLE(workers) do WAIT
    fringe ← INSERT(node, fringe)
    return F-COST[node] == F-COST[GET-FRONT(fringe)]
```

For each of these, indicate whether it results in a complete search algorithm, and whether it results in an optimal search algorithm. Give a brief justification for your answer (answers without a justification will receive zero credit). Assume that the state space is finite, and the heuristic used is consistent.

(a) (i) [4 pts] **Implementation I**

Optimal? Yes / [No]. Justify your answer:

Suppose we have a search problem with two paths to the single goal node. The first path is the optimal path, but nodes along this path take a really long time to expand. The second path is suboptimal and nodes along this path take very little time to expand. Then this implementation will return the suboptimal solution.

Complete? [Yes] / No. Justify your answer:

PARALLEL-A* will keep expanding nodes until either (a) all workers are idle (done expanding) and the fringe is empty, or (b) a goal node has been found and returned (this implementation of SHOULD-RETURN returns a goal node unconditionally when found). So, like standard A*-GRAPH-SEARCH, it will search all reachable nodes until it finds a goal.

(ii) [4 pts] **Implementation II**

Optimal? Yes / [No]. Justify your answer:

Not complete (see below), therefore not optimal.

Complete? Yes / [No]. Justify your answer:

Suppose there is just one goal node and it was just popped off the fringe by the master thread. At this time a worker can still be busy expanding some other node. When this happens this implementation returns false and we've "lost" this goal node because we've already pulled it off the fringe, and a goal node will never be returned since this was the only one.

(iii) [4 pts] **Implementation III**

Optimal? Yes / [No]. Justify your answer:

Optimality is not guaranteed. Suppose there is just a single node on the fringe and it is a suboptimal goal node. Suppose further that a single worker is currently working on expanding the parent of an optimal goal node. Then the master thread reaches line 10 and pulls the suboptimal goal node off the fringe. It then begins running GOAL-TEST in line 11. At some point during the execution of GOAL-TEST, the single busy worker pushes the optimal goal node onto the fringe and finishes executing WORKER-EXPAND, thereby becoming idle. Since it was the only busy worker when it was expanding, we now have ALL-IDLE(*workers*) and when the master thread finishes executing the goal test and runs SHOULD-RETURN, the ALL-IDLE check will pass and the suboptimal goal node is returned.

Complete? [Yes] / No. Justify your answer:

All goal nodes will be put back into the fringe, so we never throw out a goal node. Because the state space is finite and we have a closed set, we know that all workers will eventually be idle. Given these two statements and the argument from completeness of Implementation I that all reachable nodes will be searched (until a goal node is returned), we can guarantee a goal node will be returned.

(iv) [4 pts] **Implementation IV**

Optimal? [Yes] / No. Justify your answer:

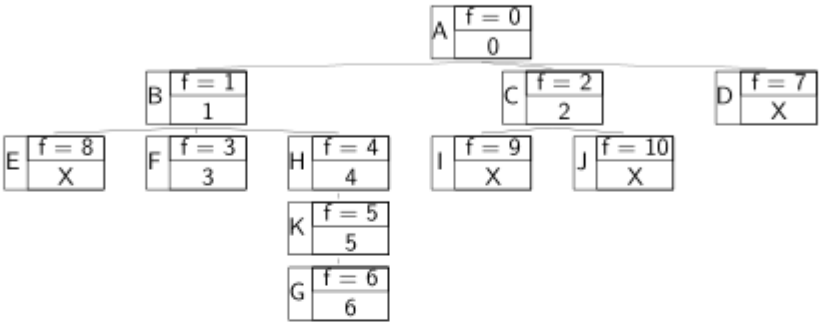
This implementation guarantees that an optimal goal node is returned. After WAITing for all the workers to become idle, we know that if there are any unexpanded nodes with lower F-cost than the goal node we are currently considering returning they will now be on the fringe (by the consistent heuristic assumption). Then, we re-insert the node into the fringe and return it only if it has F-cost equal to the node with the lowest F-cost in the fringe after the insertion. Note that even if it was *not* the lowest F-cost node in the fringe this time around, this might still be the optimal goal node. But not to worry; we have put it back into the fringe ensuring that it can still be returned once we have expanded all nodes with lower F-cost.

Complete? [Yes] / No. Justify your answer:

Optimal (see above), therefore complete.

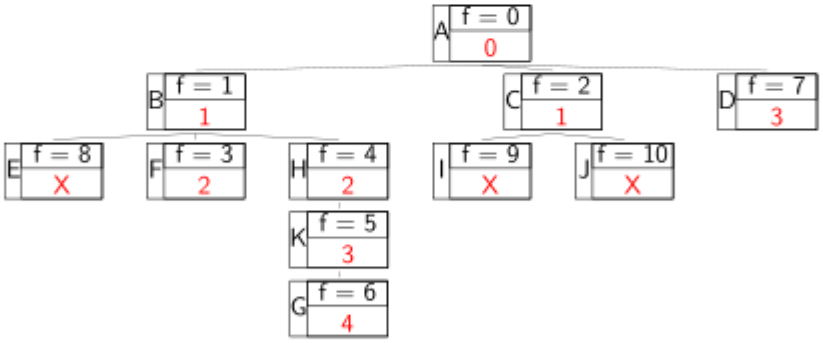
(b) Suppose we run A*-Parallel with implementation **IV** of the Should-Return function. We now make a new, additional assumption about execution time: Each worker takes exactly one time step to expand a node and push all of the successor nodes onto the fringe, independent of the number of successors (including if there are zero successors). All other computation is considered instantaneous for our time bookkeeping in this question.

A*-Parallel with the above timing properties was run with a single (1) worker on a search problem with the search tree in the diagram below. Each node is drawn with the state at the left, the f-value at the top-right ($f(n) = g(n) + h(n)$), and the time step on which a worker expanded that node at the bottom-right, with an 'X' if that node was not expanded. G is the unique goal node. In the diagram below, we can see that the start node A was expanded by the worker at time step 0, then node B was expanded at time step 1, node C was expanded at time step 2, node F was expanded at time step 3, node H was expanded at time step 4, node K was expanded at time step 5, and node G was expanded at time step 6. Nodes D, E, I, J were never expanded.

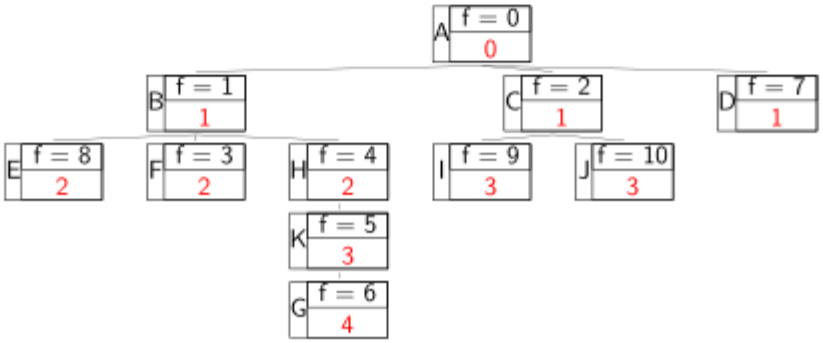


In this question you'll complete similar diagrams by filling in the node expansion times for the case of two and three workers. Note that now multiple nodes can (and typically will!) be expanded at any given time.

(i) [6 pts] Complete the node expansion times for the case of two workers and fill in an 'X' for any node that is not expanded.



(ii) [6 pts] Complete the node expansion times for the case of three workers and fill in an 'X' for any node that is not expanded.



Q1. [10 pts] All Searches Lead to the Same Destination

For all the questions below assume :

- All search algorithms are *graph* search (as opposed to tree search).
- $c_{ij} > 0$ is the cost to go from node i to node j.
- There is only one goal node (as opposed to a set of goal nodes).
- All ties are broken alphabetically.
- Assume heuristics are consistent.

Definition: Two search algorithms are defined to be ***equivalent*** if and only if they expand the same nodes in the same order and return the same path.

In this question we study what happens if we run uniform cost search with action costs d_{ij} that are potentially different from the search problem's actual action costs c_{ij}. Concretely, we will study how this might, or might not, result in running uniform cost search (with these new choices of action costs) being equivalent to another search algorithm.

(a) [2 pts] Mark *all* choices for costs d_{ij} that make running **Uniform Cost Search** algorithm with these costs d_{ij} *equivalent* to running **Breadth-First Search**.

- ◯ $d_{ij} = 0$
- ◯ $d_{ij} = \alpha$, $\alpha > 0$
- ◯ $d_{ij} = \alpha$, $\alpha < 0$
- ◯ $d_{ij} = 1$
- ◯ $d_{ij} = -1$
- ◯ None of the above

(b) [2 pts] Mark *all* choices for costs d_{ij} that make running **Uniform Cost Search** algorithm with these costs d_{ij} *equivalent* to running **Depth-First Search**.

- ◯ $d_{ij} = 0$
- ◯ $d_{ij} = \alpha$, $\alpha > 0$
- ◯ $d_{ij} = \alpha$, $\alpha < 0$
- ◯ $d_{ij} = 1$
- ◯ $d_{ij} = -1$
- ◯ None of the above

תשובה 16.1

Q1. [10 pts] All Searches Lead to the Same Destination

For all the questions below assume :

- All search algorithms are *graph* search (as opposed to tree search).
- $c_{ij} > 0$ is the cost to go from node i to node j.
- There is only one goal node (as opposed to a set of goal nodes).
- All ties are broken alphabetically.
- Assume heuristics are consistent.

Definition: Two search algorithms are defined to be ***equivalent*** if and only if they expand the same nodes in the same order and return the same path.

In this question we study what happens if we run uniform cost search with action costs d_{ij} that are potentially different from the search problem's actual action costs c_{ij}. Concretely, we will study how this might, or might not, result in running uniform cost search (with these new choices of action costs) being equivalent to another search algorithm.

(a) [2 pts] Mark *all* choices for costs d_{ij} that make running **Uniform Cost Search** algorithm with these costs d_{ij} *equivalent* to running **Breadth-First Search**.

- ○ $d_{ij} = 0$
- ● $d_{ij} = \alpha,\ \alpha > 0$
- ○ $d_{ij} = \alpha,\ \alpha < 0$
- ● $d_{ij} = 1$
- ○ $d_{ij} = -1$
- ○ None of the above

Beadth First search expands the node at the shallowest depth first. Assigning a constant positive weight to all edges allows to weigh the nodes by their depth in the search tree.

(b) [2 pts] Mark *all* choices for costs d_{ij} that make running **Uniform Cost Search** algorithm with these costs d_{ij} *equivalent* to running **Depth-First Search**.

- ○ $d_{ij} = 0$
- ○ $d_{ij} = \alpha,\ \alpha > 0$
- ● $d_{ij} = \alpha,\ \alpha < 0$
- ○ $d_{ij} = 1$
- ● $d_{ij} = -1$
- ○ None of the above

Depth First search expands the nodes which were most recently added to the fringe first. Asssigning a constant negative weight to all edges essentially allows to reduce the value of the most recently nodes by that constant, making them the nodes with the minimum value in the fringe when using uniform cost search.

(c) [2 pts] Mark *all* choices for costs d_{ij} that make running **Uniform Cost Search** algorithm with these costs d_{ij} *equivalent* to running **Uniform Cost Search** with the original costs c_{ij}.

- ◯ $d_{ij} = c_{ij}^2$
- ◯ $d_{ij} = 1/c_{ij}$
- ◯ $d_{ij} = \alpha\ c_{ij}, \qquad \alpha > 0$
- ◯ $d_{ij} = c_{ij} + \alpha, \qquad \alpha > 0$
- ◯ $d_{ij} = \alpha\ c_{ij} + \beta, \quad \alpha > 0,\ \beta > 0$
- ◯ None of the above

(d) Let $h(n)$ be the value of the heuristic function at node n.

(i) [2 pts] Mark *all* choices for costs d_{ij} that make running **Uniform Cost Search** algorithm with these costs d_{ij} *equivalent* to running **Greedy Search** with the original costs c_{ij} and heuristic function h.

- ◯ $d_{ij} = h(i) - h(j)$
- ◯ $d_{ij} = h(j) - h(i)$
- ◯ $d_{ij} = \alpha\ h(i), \quad \alpha > 0$
- ◯ $d_{ij} = \alpha\ h(j), \quad \alpha > 0$
- ◯ $d_{ij} = c_{ij} + h(j) + h(i)$
- ◯ None of the above

(ii) [2 pts] Mark *all* choices for costs d_{ij} that make running **Uniform Cost Search** algorithm with these costs d_{ij} *equivalent* to running **A* Search** with the original costs c_{ij} and heuristic function h.

- ◯ $d_{ij} = \alpha\ h(i), \quad \alpha > 0$
- ◯ $d_{ij} = \alpha\ h(j), \quad \alpha > 0$
- ◯ $d_{ij} = c_{ij} + h(i)$
- ◯ $d_{ij} = c_{ij} + h(j)$
- ◯ $d_{ij} = c_{ij} + h(i) - h(j)$
- ◯ $d_{ij} = c_{ij} + h(j) - h(i)$
- ◯ None of the above

(c) [2 pts] Mark *all* choices for costs d_{ij} that make running **Uniform Cost Search** algorithm with these costs d_{ij} *equivalent* to running **Uniform Cost Search** with the original costs c_{ij}.

- ○ $d_{ij} = c_{ij}^2$
- ○ $d_{ij} = 1/c_{ij}$
- ● $d_{ij} = \alpha\ c_{ij}, \quad \alpha > 0$
- ○ $d_{ij} = c_{ij} + \alpha, \quad \alpha > 0$
- ○ $d_{ij} = \alpha\ c_{ij} + \beta, \quad \alpha > 0,\ \beta > 0$
- ○ None of the above

Uniform cost search expands the node with the lowest cost-so-far $= \sum_{ij} c_{ij}$ on the fringe. Hence, the relative ordering between two nodes is determined by the value of $\sum_{ij} c_{ij}$ for a given node. Amongst the above given choices, only for $d_{ij} = \alpha\ c_{ij},\ \alpha > 0$, can we conclude,
$\sum_{ij \in \text{ path}(n)} d_{ij} \geq \sum_{ij \in \text{ path}(m)} d_{ij} \iff \sum_{ij \in \text{ path}(n)} c_{ij} \geq \sum_{ij \in \text{ path}(m)} c_{ij}$, for some nodes n and m.

(d) Let $h(n)$ be the value of the heuristic function at node n.

(i) [2 pts] Mark *all* choices for costs d_{ij} that make running **Uniform Cost Search** algorithm with these costs d_{ij} *equivalent* to running **Greedy Search** with the original costs c_{ij} and heuristic function h.

- ○ $d_{ij} = h(i) - h(j)$
- ● $d_{ij} = h(j) - h(i)$
- ○ $d_{ij} = \alpha\ h(i), \quad \alpha > 0$
- ○ $d_{ij} = \alpha\ h(j), \quad \alpha > 0$
- ○ $d_{ij} = c_{ij} + h(j) + h(i)$
- ○ None of the above

Greedy search expands the node with the lowest heuristic function value $h(n)$. If $d_{ij} = h(j) - h(i)$, then the cost of a node n on the fringe when running uniform-cost search will be $\sum_{ij} d_{ij} = h(n) - h(start)$. As $h(start)$ is a common constant subtracted from the cost of all nodes on the fringe, the relative ordering of the nodes on the fringe is still determined by $h(n)$, i.e. their heuristic values.

(ii) [2 pts] Mark *all* choices for costs d_{ij} that make running **Uniform Cost Search** algorithm with these costs d_{ij} *equivalent* to running **A* Search** with the original costs c_{ij} and heuristic function h.

- ○ $d_{ij} = \alpha\ h(i), \quad \alpha > 0$
- ○ $d_{ij} = \alpha\ h(j), \quad \alpha > 0$
- ○ $d_{ij} = c_{ij} + h(i)$
- ○ $d_{ij} = c_{ij} + h(j)$
- ○ $d_{ij} = c_{ij} + h(i) - h(j)$
- ● $d_{ij} = c_{ij} + h(j) - h(i)$
- ○ None of the above

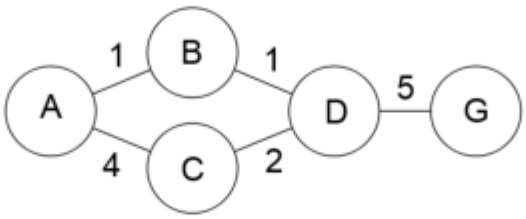
A^* search expands the node with the lowest $f(n) + h(n)$ value, where $f(n) = \sum_{ij} c_{ij}$ is the cost-so-far and h is the heuristic value. If $d_{ij} = c_{ij} + h(j) - h(i)$, then the cost of a node n on the fringe when running uniform-cost search will be $\sum_{ij} d_{ij} = \sum_{ij} c_{ij} + h(n) - h(start) = f(n) + h(n) - h(start)$. As $h(start)$ is a common constant subtracted from the cost of all nodes on the fringe, the relative ordering of the nodes on the fringe is still determined by $f(n) + h(n)$.

Q2. [18 pts] Dynamic A* Search

After running A* graph search and finding an optimal path from start to goal, the cost of one of the edges, $X \to Y$, in the graph changes. Rather than re-running the entire search, you want to find a more efficient way of finding the optimal path for this new search problem.

You have access to the fringe, the closed set and the search tree as they were at the completion of the initial search. In addition, you have a *closed node map* that maps a state, s from the closed set to a list of nodes in the search tree ending in s which were not expanded because s was already in the closed set.

For example, after running A* search with the null heuristic on the following graph, the data structures would be as follows:



Fringe: {} Closed Node Map: {A:[ ], B:[ ], C:[ ], D:[(A-C-D, 6)]}

Closed Set: {A, B, C, D} Search Tree:

{A : [(A-B, 1), (A-C, 4)],
A-B : [(A-B-D, 2)],
A-C : [ ],
A-B-D : [(A-B-D-G, 7)],
A-B-D-E : [ ]}

For a general graph, for each of the following scenarios, select the choice that finds the correct optimal path and cost *while expanding the fewest nodes*. Note that if you select the 4^{th} choice, you must fill in the change, and if you select the last choice, you must describe the set of nodes to add to the fringe.

In the answer choices below, if an option states some nodes will be added to the fringe, this also implies that the final state of each node gets cleared out of the closed set (indeed, otherwise it'd be rather useless to add something back into the fringe). You may assume that there are no ties in terms of path costs.

Following is a set of eight choices you should use to answer the questions on the following page.

i. The optimal path does not change, and the cost remains the same.
ii. The optimal path does not change, but the cost increases by n
iii. The optimal path does not change, but the cost decreases by n
iv. The optimal path does not change, but the cost changes by _____
v. The optimal path for the new search problem can be found by adding the subtree rooted at X that was expanded in the original search back onto the fringe and re-starting the search.
vi. The optimal path for the new search problem can be found by adding the subtree rooted at Y that was expanded in the original search back onto the fringe and re-starting the search.
vii. The optimal path for the new search problem can be found by adding all nodes for each state in the *closed node map* back onto the fringe and re-starting the search.
viii. The optimal path for the new search problem can be found by adding some other set of nodes back onto the fringe and re-starting the search. Describe the set below.

(a) [3 pts] Cost of $X \to Y$ is increased by $n, n > 0$, the edge is on the optimal path, and was explored by the first search.

○ i ○ ii ○ iii ○ iv, Change:
○ v ○ vi ○ vii ○ viii, Describe the set below:

(b) [3 pts] Cost of $X \to Y$ is decreased by $n, n > 0$, the edge is on the optimal path, and was explored by the first search.

○ i ○ ii ○ iii ○ iv, Change:
○ v ○ vi ○ vii ○ viii, Describe the set below:

(c) [3 pts] Cost of $X \to Y$ is increased by $n, n > 0$, the edge is not on the optimal path, and was explored by the first search.

○ i ○ ii ○ iii ○ iv, Change:
○ v ○ vi ○ vii ○ viii, Describe the set below:

(d) [3 pts] Cost of $X \to Y$ is decreased by $n, n > 0$, the edge is not on the optimal path, and was explored by the first search.

○ i ○ ii ○ iii ○ iv, Change:
○ v ○ vi ○ vii ○ viii, Describe the set below:

(e) [3 pts] Cost of $X \to Y$ is increased by $n, n > 0$, the edge is not on the optimal path, and was not explored by the first search.

○ i ○ ii ○ iii ○ iv, Change:
○ v ○ vi ○ vii ○ viii, Describe the set below:

(f) [3 pts] Cost of $X \to Y$ is decreased by $n, n > 0$, the edge is not on the optimal path, and was not explored by the first search.

○ i ○ ii ○ iii ○ iv, Change:
○ v ○ vi ○ vii ○ viii, Describe the set below:

תשובה 17.1

(a) [3 pts] Cost of $X \to Y$ is increased by $n, n > 0$, the edge is on the optimal path, and was explored by the first search.

○ i ○ ii ○ iii ○ iv, Change:
○ v ○ vi ○ vii ● viii, Describe the set below:

The combination of all the nodes from the *closed node map* for the final state of each node in the subtree rooted at Y plus the node ending at Y that was expanded in the initial search. This means that you are re-exploring every path that was originally closed off by a path that included the edge $X \to Y$.

(b) [3 pts] Cost of $X \to Y$ is decreased by $n, n > 0$, the edge is on the optimal path, and was explored by the first search.

○ i ○ ii ● iii ○ iv, Change:
○ v ○ vi ○ vii ○ viii, Describe the set below:

The original optimal path's cost decreases by n because $X \to Y$ is on the original optimal path. The cost of any other path in the graph will decrease by at most n (either n or 0 depending on whether or not it includes $X \to Y$). Because the optimal path was already cheaper than any other path, and decreased by at least as much as any other path, it must still be cheaper than any other path.

(c) [3 pts] Cost of $X \to Y$ is increased by $n, n > 0$, the edge is not on the optimal path, and was explored by the first search.

● i ○ ii ○ iii ○ iv, Change:
○ v ○ vi ○ vii ○ viii, Describe the set below:

The cost of the original optimal path, which is lower than the cost of any other path, stays the same, while the cost of any other path either stays the same or increases. Thus, the original optimal path is still optimal.

(d) [3 pts] Cost of $X \to Y$ is decreased by $n, n > 0$, the edge is not on the optimal path, and was explored by the first search.

○ i ○ ii ○ iii ○ iv, Change:
○ v ○ vi ○ vii ○ viii, Describe the set below:

The combination of the previous goal node and the node ending at X that was expanded in the initial search.

There are two possible paths in this case. The first is the original optimal path, which is considered by adding the previous goal node back onto the fringe. The other option is the cheapest path that includes $X \to Y$, because that is the only cost that has changed. There is no guarantee that the node ending at Y, and thus the subtree rooted at Y contains $X \to Y$, so the subtree rooted at X must be added in order to find the cheapest path through $X \to Y$.

(e) [3 pts] Cost of $X \to Y$ is increased by $n, n > 0$, the edge is not on the optimal path, and was not explored by the first search.

● i ○ ii ○ iii ○ iv, Change:
○ v ○ vi ○ vii ○ viii, Describe the set below:

This is the same as part (c).

(f) [3 pts] Cost of $X \to Y$ is decreased by $n, n > 0$, the edge is not on the optimal path, and was not explored by the first search.

● i ○ ii ○ iii ○ iv, Change:
○ v ○ vi ○ vii ○ viii, Describe the set below:

Assuming that the cost of $X \to Y$ remains positive, because the edge was never explored, the cost of the path to X is already higher than the cost of the optimal path. Thus, the cost of the path to Y through X can only be higher, so the optimal path remains the same.

If you allow edge weights to be negative, it is necessary to find the optimal path to Y through X separately. Because the edge was not explored, a node ending at X was never expanded, so the negative edge would still never be seen unless the path was found separately and added onto the fringe. In this case, adding this path and the original goal path, similar to (d), would find the optimal path with the updated edge cost.