CS 188 Fall 2022 — Introduction to Artificial Intelligence — Midterm

- You have 110 minutes.
- The exam is closed book, no calculator, and closed notes, other than one double-sided cheat sheet that you may reference.
- For multiple choice questions,

  ☐ means mark **all options** that apply

  ○ means mark a **single choice**

First name	
Last name	
SID	
Name and SID of person to the right	
Name and SID of person to the left	
Discussion TAs (or None)	

Honor code: "As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others."

By signing below, I affirm that all work on this exam is my own work, and honestly reflects my own understanding of the course material. I have not referenced any outside materials (other than one double-sided cheat sheet), nor collaborated with any other human being on this exam. I understand that if the exam proctor catches me cheating on the exam, that I may face the penalty of an automatic "F" grade in this class and a referral to the Center for Student Conduct.

Signature: ______________________________

Point Distribution

Q1.	Potpourri	22
Q2.	Haunted House	17
Q3.	Rocket Science	16
Q4.	Golden Bear Years	13
Q5.	Games	8
Q6.	Indiana Jones & the Kingdom of the Crystal Skull	12
Q7.	Reinforcement Learning	12
	Total	100

Q1. [22 pts] Potpourri

(a) Evgeny has an arbitrary search problem. Assume all costs are positive.

(i) [1 pt] If he runs uniform cost graph search on the problem, he is guaranteed to get the optimal solution.

[✓ True] ○ False

True. Uniform-cost graph search will consider all the paths with cost 0, then all the paths with cost 1, then all the costs with path 2, etc. until finding the optimal (least-cost) solution. (Note the assumption that all costs are positive, so we don't have to worry about negative-cost solutions.)

(ii) [1 pt] Jason takes the problem and designs a heuristic function h such that for all states s, $h(s) > 0$. If he runs greedy graph search on the problem using the heuristic, he is guaranteed to get the optimal solution.

○ True [✓ False]

False. Greedy search with an arbitrary heuristic is not guaranteed to get the optimal solution.

(iii) [1 pt] Regardless of your answer for the previous parts, suppose both uniform cost graph search and greedy graph search return the same optimal path. Jason claims that using his heuristic, A* graph search is guaranteed to return the optimal path for Evgeny's search problem. Is he correct?

○ Yes, because the priority value for A* search on a given state is the sum of priority values for UCS and greedy search.

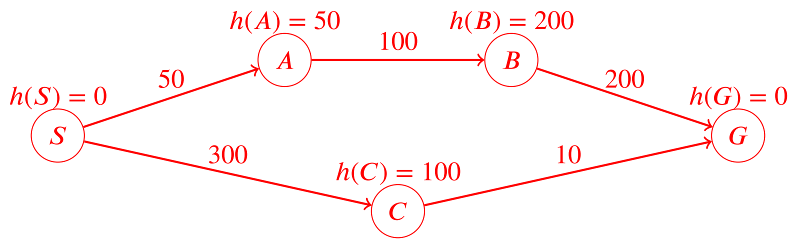
○ Yes, but not for the reason above.

[✓ No, because if the heuristic is not consistent, then A* graph search is not guaranteed to be optimal.]

○ No, but not for the reason above.

UCS and greedy search could return an optimal solution with an inadmissible or inconsistent heuristic (e.g. if there's only one solution). However, A* graph search is not guaranteed to return the optimal solution if the heuristic is not consistent.

A counterexample here. The optimal path is S-C-G but A* will return S-A-B-G.



(b) [2 pts] In a CSP, what could happen to the domain of an arbitrary variable after enforcing arc consistency? Select all that apply.

[✓ The domain could remain unchanged.]

[✓ The domain size could be smaller than before (but not empty).]

[✓ The domain could be empty.]

○ None of the above

When enforcing arc consistency, values in a variable's domain may or may not be removed. It's possiblet hat no values get removed from the domain, some values get removed, or all values get removed.

(c) Consider a general CSP with n variables of domain size d. We would like to use a search tree to solve the CSP, where all the complete assignments (and thus all the solutions) are leaf nodes at depth n.

(i) [2 pts] How many possible complete assignments are there for this CSP?

○ $O(n^2d^3)$

○ $O(n \cdot d)$

○ $O(n^2)$

○ $O(n^2d)$

○ $O(n^d)$

[✓ $O(d^n)$]

○ None of the above

There are d^n complete assignments in CSP because each variable has d values in its domain, and there are n variables.

(ii) [2 pts] How many leaves does the search tree have?

○ $O(n^2 \cdot d^n)$	● $O(n! \cdot d^n)$
○ $O(d! \cdot n^d)$	○ Same as the answer to the previous subpart
○ $O(d^2 \cdot n^d)$	○ None of the above

The branching factor at the top level would be nd because there is a choice of n variables to assign first, and each variable could be assigned to one of a choice of d values. At the second level, there is a choice of $n-1$ remaining variables to assign, and each variable could still be assigned to any of the d values, for a branching factor of $d(n-1)$. At each level, there is one fewer variable to assign, but each variable can always be assigned to any of the d values.

In total, the number of leaves is the product of all the branching factors. (Consider a tree where the root has 3 children, i.e. branching factor 3, and each of the children has 2 more children, i.e. branching factor 2. This tree has $2 \times 3 = 6$ total leaves.)

There are n levels of the tree (one level for each variable to assign.) The branching factors are $dn, d(n-1), d(n-2), \dots, d(1)$. The product of the branching factors is $n! \times d^n$.

Another way to arrive at the answer is to note that each leaf of the search tree is a complete assignment. However, a complete assignment could be obtained by assigning each variable in an arbitrary order. (Recall that each node of the search tree corresponds to a sequence of actions, and in CSPs, an action in the search problem is assigning a value to a variable.) Each complete assignment can be assigned in one of $n!$ orders (n choices of the first variable to assign, $n-1$ choices of the second variable to assign, etc.). There are d^n assignments, and each assignment can be assigned in one of $n!$ orders, for a total of $n! \times d^n$ leaves.

(d) Suppose you are playing a zero-sum game in which the opponent plays optimally. There are no chance nodes in the game tree.

(i) [1 pt] If you want to maximize your utility when playing against this opponent, which algorithm would be the best fit? Assume all the algorithms don't have a limit on search depth.

● Minimax	○ Monte Carlo Tree Search
○ Expectimax	○ Not enough information

The game is zero-sum, which means that your opponent's utility is the negative of your utility.

Since your opponent is playing optimally, they're trying to maximize their utility. Equivalently, your opponent is trying to minimize your utility. Minimax is the algorithm that models your opponent as minimizing your utility.

(ii) [2 pts] Select all true statements about pruning game trees.

- ☐ Using alpha-beta pruning allows you to choose an action with a higher value compared to not using alpha-beta pruning.
- ☐ Alpha-beta pruning ensures that all the nodes in the game tree have their correct values.
- ☐ Alpha-beta pruning will always prune at least one node/leaf in a minimax game tree.
- ■ Alpha-beta pruning does not work on expectimax game trees with unbounded values.
- ○ None of the above

Option 1: False. Alpha-beta pruning does not affect the action chosen at the root node. (We got a couple clarification requests about what it means to "choose an action"–recall that in game trees, we only keep track of the action chosen at the root node, not actions chosen at intermediate nodes.)

Option 2: False. If nodes get pruned, some of the intermediate nodes in the tree may take on incorrect values. Only the root node is guaranteed to have the correct value.

Option 3: False. In the worst case, alpha-beta pruning might not be able to prune any nodes at all.

Option 4: True. In most cases, alpha-beta pruning is able to prune at least one node. Only specific worst-case orderings of leaf nodes actually cause alpha-beta pruning to prune no nodes.

(e) [2 pts] Select all true statements about alpha-beta pruning.

- ☐ Alpha-beta pruning affects the action selected at the root node.
- ■ The order in which nodes are pruned affects the number of nodes explored.
- ☐ The order in which nodes are pruned affects the action selected at the root node.

🟥 In most cases, pruning reduces the number of nodes explored.

○ None of the above

Option 1: True. Minimax doesn't involve taking samples. MCTS relies on randomly performing playouts of games from a given state.

Option 2: False. In the worst case, alpha-beta pruning might not prune any nodes, and every node in the search tree would need to be visited.

Option 3: True. Minimax without alpha-beta pruning will search every node in the search tree.

Option 4: False. Alpha-beta pruning is an optimization added to standard minimax (which doesn't prune any nodes at all), so minimax cannot explore fewer nodes than minimax with alpha-beta pruning.

Alpha-Beta pruning only prunes away branches that cannot possibly influence the final decision. Therefore the third option is incorrect. In addition, alpha-beta pruning doesn't affect the actions selected at the root node. Thus option 1 is incorrect.

(f) [2 pts] Select all true statements about minimax, minimax with alpha-beta pruning, and Monte Carlo tree search (MCTS).

🟥 Among the three search algorithms, only MCTS is a sample-based search algorithm.

☐ Neither minimax with alpha-beta pruning nor MCTS will ever search the tree exhaustively (i.e. visit every node in the search tree).

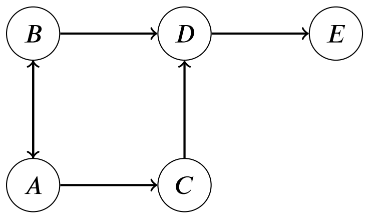
🟥 Minimax is an exhaustive search algorithm (i.e. it visits every node in the search tree).

☐ When the game tree is small, minimax can explore fewer nodes than minimax with alpha-beta pruning.

○ None of the above

Although Alpha-beta pruning sometimes prunes those that are guaranteed not to be an optimal state for the current player, there are cases that it cannot prune anything. MiniMax always explore all paths. Therefore the last option is incorrect.

(g) The following graph defines an MDP with deterministic transitions. Each transition produces a constant reward $r = 1$. The discount factor is $\gamma = 0.5$. The game ends once the state E is reached (there are no actions available from E).



$$\begin{tikzcd} B \arrow[r] \arrow[d, leftrightarrow] & D \arrow[r] & E \\ A \arrow[r] & C \arrow[u] & \end{tikzcd}$$

Note: The formula for the sum of an infinite geometric series is $\sum_{i=0}^{\infty} x^i = 1 + x + x^2 + x^3 + ... = \frac{1}{1-x}$

(i) [1 pt] What is $V^*(D)$, the optimal value at state D?

$\boxed{1}$

This MDP is small enough that we can solve for the optimal values by inspection. From state D, the only possible action is to move to E, which gives immediate reward of 1. From E, no further rewards are available. In total, the expected discounted reward from starting at state D and moving optimally is 1. (Note that the reward of 1 is not discounted, since it is the result of the immediate next action from D.)

(ii) [1 pt] What is $V^*(A)$, the optimal value at state A?

$\boxed{2}$

This MDP is small enough that we can solve for the optimal values by inspection. Since all transition rewards are positive, we'd like to stay in this MDP as long as possible. From A, the optimal policy is to infinitely move between A and B. Using the formula for an infinite geometric series, this infinite sequence of actions gives expected discounted reward of $1 + 0.5 + 0.5^2 + \ldots = 1/(1 - 0.5) = 2$.

(iii) [1 pt] How many iteration(s) are needed for policy iteration to converge in the **worst case** (for any initial policy π_0)? In other words, select the minimum k such that $\pi_k = \pi^*$ for the worst π_0.

○ 0
○ 1
[✓ 2]
○ 3
○ ∞ (never converges in the worst case)
○ None of the above

Note that the policy from C, D, and E is fixed, because there is only one action available from C and D, and there are no actions available from E.

By inspection, the optimal policy from A is to go to B, and the optimal policy from B is to go to A. In the worst case, the initial policy could sub-optimally move right for both A and B.

To run policy iteration, we evaluate the values of the initial policy. The initial policy from A would go to C, then, D, then E. This sequence gives expected discounted reward of $V^{\pi_0}(A) = 1 + 0.5 + 0.25 = 1.75$. The initial policy of B would go to D, then, E, for an expected discounted reward of $V^{\pi_0}(B) = 1 + 0.5 = 1.5$. Similar reasoning gives $V^{\pi_0}(C) = 1.5$ (two actions to E), $V^{\pi_0}(D) = 1$ (one action to E), and $V^{\pi_0}(E) = 0$ (no actions available from E).

Then, we can run policy improvement and see how the policy from A and B changes (remember that the policy from C, D, and E is fixed).

From A, moving right to C, then following π_0 would give expected discounted reward of $1 + 0.5V^{\pi_0}(C) = 1.75$.

From A, moving up to B, then following π_0 would give expected discounted reward of $1 + 0.5V^{\pi_0}(B) = 1.75$.

Since moving up and right give equal reward, the optimal policy from A could be either up or right. Let's imagine the worst-case scenario and suppose that $\pi_1(A) = \rightarrow$.

From B, moving right to D, then following π_0 would give expected discounted reward of $1 + 0.5V^{\pi_0}(D) = 1.5$.

From B, moving down to A, then following π_0 would give expected discounted reward of $1 + 0.5V^{\pi_0}(A) = 1.875$.

The better action from B is to move down to A, so $\pi_1(B) = \downarrow$.

At this point, we don't have the optimal policy yet, so we can run one more iteration of policy iteration. In policy evaluation, the only value that changes is $V^{\pi_1}(B) = 1 + 0.5 + 0.75 + 0.375 = 2.625$, for the sequence of actions BACDE.

From A, moving right to C, then following π_1 still gives expected discounted reward of 1.75.

From A, moving up to B, then following π_1 gives expected discounted reward of $1 + 0.5(2.625) = 2.3125$.

The better action from A is to move up to B, so $\pi_2(A) = \uparrow$.

After two iterations of policy iteration, we've found the optimal policy.

(iv) [1 pt] How many iteration(s) are needed for value iteration to converge if $V_0(s) = 0$ for all states s? In other words, select the minimum k such that $V_k(s) = V^*(s)$.

○ 0
○ 1
○ 2
○ 3
[✓ ∞ (never converges)]
○ None of the above

Recall that $V_k(s)$ is the expected discounted reward for starting at state s and acting optimally for k moves.

In this MDP, we know that the optimal policy from A is to travel between state A and B forever. Thus, no matter how large k is, $V_k(A)$ and $V_k(B)$ will never equal $V^*(A)$ and $V^*(B)$.

Another way to reason this out is to note that $V^*(A) = 1 + 0.5 + 0.5^2 + \ldots = 2$ is the result of an infinite summation. Each term in the summation represents one extra time step of reward. Value iteration would add one term to sum in each iteration, and there are an infinite number of terms, so value iteration will never converge to the exact value of $V^*(A)$.

(h) [2 pts] Select all true statements about reinforcement learning.

☐ Direct Evaluation requires knowing the transition function of the underlying MDP.
[✓ TD Learning itself does not learn the optimal value/policy.]
☐ The optimal Q-value $Q^*(s, a)$ is the expected discounted reward for following the optimal policy starting at state s.
[✓ Q-learning can learn the optimal policy using only transition data from random policies.]
○ None of the above

Option 1: False. Direct Evaluation is model-free, so it doesn't need the model. In general, reinforcement learning methods like direct evaluation are used when you don't know the transition function of the underlying MDP.

Option 2: True. TD Learning only evaluates the current policy. Recall that TD learning computes a running average of rewards from the episodes we see. TD learning never uses a max term to compute the optimal action or values of any state or Q-state.

Option 3: False. The optimal Q value is the expected discounted reward for starting at state s, taking action a (even if a is sub-optimal), *then* following the optimal policy after the first action.

Option 4: True. Q Learning is off-policy, so the behavioral policy can be random as long as it reaches every (s,a) pair infinitely often.

Q2. [17 pts] Haunted House

Mr. and Mrs. Pacman have found themselves inside of a haunted house with H floors. Each floor is a rectangular grid of dimension $M \times N$. Mr. Pacman and Mrs. Pacman have been separated, and must find each other. There is one staircase per floor, all located on the same square in the $M \times N$ grid. At each timestep, Mr. Pacman or Mrs. Pacman can move *North*, *East*, *South*, *West*, *Up*, or *Down* (they can take *Up* or *Down* only if they are at a staircase). However, Mr. Pacman can only travel L levels of stairs before he has to rest for W turns. Mr. and Mrs. Pacman alternate turns.

(a) **(i)** [1 pt] What is the maximum branching factor?

$\boxed{6}$ The maximum branching factor is 6 (North, West, South, East, up, down) At each turn either Mr. or Mrs. Pacman can move.

Note that the branching factor is not $6^2 = 36$, because the two Pacmans move in separate turns. If both Pacmans moved at once, then the branching factor would be 36, because the list of possible actions would include every possible pair of actions that Mr. and Mrs. Pacman take.

(ii) [2 pts] Assume that all actions have a cost of 1, except for *Up* and *Down*, which have a cost of 2. Which of the following search algorithms will be guaranteed to return the solution with the fewest number of actions? Select all that apply.

☐ Depth-First Search
■ Breadth-First Search
■ Uniform-Cost Search
○ None of the above

By definition, depth-first search has no guarantees of returning the solution with the fewest number of actions.

By definition, breadth-first search is guaranteed to return the solution with the fewest number of actions.

In general, UCS will not return the solution with the fewest number of actions. However, because Mr. and Mrs. Pacman have to travel the same number of stairs regardless of the path chosen to meet on the same floor, every solution will have the same number of cost-2 actions. (For example, if Mr. Pacman is on floor 2 and Mrs. Pacman is on floor 6, any solution must have a total of 4 Up/Down actions.) UCS returns the solution with the lowest cost. Since all solutions have the same number of cost-2 actions, UCS will return the solution with the fewest number of cost-1 actions, which is also the solution with the fewest number of total actions.

(iii) [6 pts] For **a(iii) and a(iv)** only, assume there are K knights that Mr. and Mrs. Pacman have to avoid, located throughout the house. Since the knights are wearing heavy armor, they cannot move. Fill in the blanks such that S evaluates to the minimal state space size. Each blank can include numbers (including 0) or variables in the problem statement.

$$S = A \cdot 6^B \cdot M^C \cdot N^D \cdot H^E \cdot K^F$$

A = $\boxed{2 \cdot (L + W)}$ B = $\boxed{0}$ C = $\boxed{2}$ D = $\boxed{2}$ E = $\boxed{2}$ F = $\boxed{0}$

A $= 2 \cdot (L + W)$. We need a factor of 2 to keep track of whether Mr. or Mrs. Pacman moves next. Also, we need a counter to keep track of how many stairs Pacman can still travel, or how many steps Pacman still needs to rest for. Note that we can actually use one counter to keep track of both of these, because only one counter will ever be used at a time; Pacman is either still able to travel for some number of stairs, or is unable to travel and must rest for some number of turns. (As a concrete example, you could store a number ranging from $-W$ to $+L$. If the number is negative (between $-W$ and -1), it represents the number of turns Pacman still needs to rest. If the number is positive (between 1 and L), it represents the number of stairs Pacman can still travel. This number can take on one of $L + W$ values.

B = 0. There's no power of 6 needed in this state space.

$C = D = E = 2$. Mr. Pacman can be in one of MNH squares. Mrs. Pacman can be in one of MNH squares. We need to keep track of both of their locations in every state, which results in a factor of $(MNH)^2$.

F = 0. Note that we don't need to store any information about the knights in our states, because they're located in a constant location throughout the problem (like the walls in classic Pacman). Instead, we could encode the hard-coded locations of the knights in the code logic for our successor function and/or goal test.

In summary: $S = 2 \cdot (L + W) \cdot (MNH)^2$

(iv) [2 pts] Assume that all actions have a cost of 1, except for *Up* and *Down*, which have a cost of 2. Which of the following search algorithms will be guaranteed to return the solution with the fewest number of actions? Select all that apply.

- ☐ Depth-First Search
- [✓ Breadth-First Search]
- [✓ Uniform-Cost Search]
- ○ None of the above

Adding the knights doesn't change the fact that every solution needs the same number of cost-2 Up/Down actions. From the same reasoning as (a)(ii), BFS and UCS will return the path with the fewest number of actions, which is equivalently the shortest/cheapest path.

(b) For each modification to the original problem, **write the term X such that the new minimal state space size S' is $X \cdot S$.** Each subpart below is **independent** (the modifications are not combined).

(i) [2 pts] Mr. and Mrs. Pacman discover two elevators that start on floor 1 and floor H respectively. On each turn, each elevator moves one level up and down, respectively. When an elevator reaches the top or bottom floor, it reverses direction. This motion is repeated for all time steps.

$$X = (2 \cdot H)$$

. One way to see this is to note that the elevator positions cycle every $2H$ turns, so we just need a counter telling us how far into the length-$2H$ cycle we're currently in.

Another way to see this is to note that the location first elevator completely determines the location of the second elevator. (For example, if the first elevator is at the top, you know that the second elevator must be at the bottom.) Thus we can store a counter between 1 and H telling us the location of the first elevator. However, we also need to know if the elevators are moving toward each other, or away from each other. We need a Boolean flag to keep track of this, which adds a factor of 2.

(ii) [2 pts] Mr. and Mrs. Pacman find out that there are indistinguishable ghosts inside the house. There are G ghosts, where **G is much smaller than** $M \cdot N \cdot H$. The ghosts move all at once randomly after each turn, and there can only be one ghost in a single square.

$$X = (M \cdot N \cdot H)^G$$

Since G is relatively small, it makes sense to store a list of ghost locations. Each ghost can be in one of MNH squares on the board, and there are G ghosts.

(iii) [2 pts] Same as the previous subpart, but now there are more ghosts.

Specifically, $G > M \cdot N \cdot H \cdot \frac{\log(2)}{\log(M \cdot N \cdot H)}$. (You don't need this expression to solve this problem.)

$$X = 2^{N*M*H}$$

As G becomes increasingly large, it makes sense to store a list of Booleans indicating whether each square has a ghost. There are MNH squares, and each square needs a Boolean flag indicating whether it contains a ghost.

Addendum (you don't need this to solve the problem): When does it become more efficient to use the Boolean list instead of the list of locations? We can compare the two state spaces and figure out a value of G when it becomes more efficient to use the Boolean list.

$$2^{N*M*H} < (N * M * H)^G$$

We can take the log of both side (monotonically increasing functions)

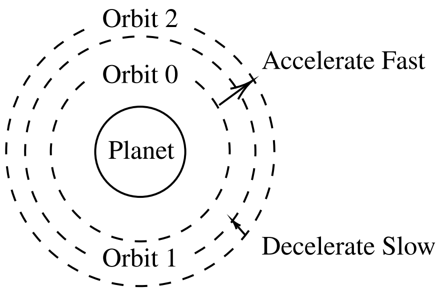
$$\log 2^{N*M*H} < \log (N * M * H)^G$$

$$N * M * H * \log (2) < G * \log (N * M * H)$$

$$N * M * H * \frac{\log (2)}{\log (N * M * H)} < G$$

Q3. [16 pts] Rocket Science

We are in a spaceship, orbiting around a planet. The actions available in this search problem are listed below:



Action	Result	Cost
Accelerate Fast	Move up 2 orbits	$5 + 3k$
Accelerate Slow	Move up 1 orbit	$5 + k$
Decelerate Fast	Move down 2 orbits	$5 + 3k$
Decelerate Slow	Move down 1 orbit	$5 + k$

For all subparts, assume that we are performing A* search, and **a solution exists from any state**. Select whether the provided heuristic is admissible and whether the provided heuristic is consistent, **for any choice of** $k > 0$.

(a) Suppose our goal is a specific orbit.

(i) [2 pts] h_1 = min # of actions needed to go to the goal orbit if **only Slow actions** are allowed

[✓ Admissible] [✓ Consistent] ○ Neither

First, note that h_1 is equivalently the distance, measured in number of orbits, between our current orbit and the goal orbit. (For example, if we're in orbit 5 and the goal is orbit 8, then $h_1 = 3$, because it would take 3 slow actions to reach the goal.)

We can travel at most 2 orbits in one action, so if we're h_1 orbits away from the goal, then we need at least $h_1/2$ actions to reach the goal. Each action costs 5, so the total cost must be at least $5(h_1/2) = 2.5h_1$.

We have $h_1 < 2.5h_1 \leq$ true cost, so h_1 is admissible.

Note that each action costs at least 5. If we need x actions to reach the goal, our cost is at least $5x$. However, our heuristic would be x. Thus the heuristic always underestimates the true cost: $h_1 = x < 5x <$ true cost.

Intuitively, two neighboring states can only differ by at most 2 orbits. This means that the heuristic difference between two neighboring states is at most 2. However, the true cost difference between two neighboring states must be at least 5, since each action has cost greater than 5. Thus the heuristic difference always underestimates the true cost difference, so we have a consistent heuristic.

Formally, consider neighboring states s_1 and s_2. $|h_1(s_1) - h_1(s_2)| \leq 2 \leq 5 \leq cost(s_1, s_2)$. For any arbitrary pair of states, we can construct the optimal single-step path, and add the above inequality for each step and get that this holds for any pair of states. Thus, this is also consistent.

(ii) [2 pts] $h_2 = h_1 \cdot (5 + k)$

☐ Admissible ☐ Consistent [✓ Neither]

The idea behind this heuristic is to multiply the orbit distance by a lower bound on the cost per step (each action costs at least $5 + k$).

Intuitively, this idea fails to consider that the fast action actually moves you to the goal with half as many actions. We can make this concrete by designing a counterexample involving some Fast actions. Suppose that we're in orbit 5, and the goal orbit is 15. Then $h_2 = 10(5 + k)$. The true cost would be to take five Fast actions to orbit 15, then one slow action to orbit 8, for a true cost of $5(5 + 3k)$.

Now, we can write an inequality to see if there's a value of k for which the heuristic overestimates the true cost: $10(5 + k) > 5(5 + 3k)$. We can expand to $50 + 10k > 25 + 15k$, then rearrange to $25 > 5k$ and $k < 5$. We can also check by plugging in $k = 1$, which gives a heuristic of $10(5 + 1) = 60$, but a true cost of $5(5 + 3) = 40$.

Since the heuristic might overestimate the true cost, it's inadmissible. By definition, an inadmissible heuristic is also inconsistent.

(iii) [2 pts] $h_3 = \min\left[h_1 \cdot (5 + k),\ m\right]$, $\quad m = \begin{cases} 0.5 \cdot h_1 \cdot (5 + 3k) + 0.5 \cdot (5 - k) & \text{if } h_1 \text{ is odd} \\ 0.5 \cdot h_1 \cdot (5 + 3k) & \text{otherwise} \end{cases}$

[✓ Admissible]   [✓ Consistent]   ○ Neither

We know that $h_1 \cdot (5+k)$ is the orbit distance multiplied by the cost of a slow action. In other words, this is the cost of a solution only using Slow actions.

Intuitively, it seems like the $0.5 \cdot h_1 \cdot (5+3k)$ term is doing something with Fast actions, given that $5+3k$ is the cost of a Fast action. The $0.5 \cdot h_1$ term appears to be accounting for Fast actions by halving the orbit distance: if we're h_1 orbits from the goal, we'd need approximately $0.5 \cdot h_1$ Fast actions to reach the goal. Thus $0.5 \cdot h_1 \cdot (5+3k)$ is approximately the cost to reach the goal if we only use Fast actions.

Next, note the cases where h_1 is even and odd, which might clue you into the fact that $0.5 \cdot h_1 \cdot (5+3k)$ fails to account for being an odd number of actions away from the goal. If the orbit distance is odd, we can use Fast actions to get close to the goal 2 orbits at a time, but we'd eventually need one Slow action to actually reach the goal.

Now we can verify our intuition by computing the true cost to the goal if we use as many Fast actions as possible. If the orbit distance is odd, we can take $0.5 \cdot (h_1 - 1)$ Fast actions, followed by a single Slow action to reach the goal. (Try writing out an example if you're having trouble deriving this expression.) Multiplying by costs, we get a cost of $0.5 \cdot (h_1 - 1) \cdot (5+3k)$ for the Fast actions, plus $1 \cdot (5+k)$ for the Slow action. Rearranging terms, we get:

$$\begin{aligned}
& 0.5 \cdot (h_1 - 1) \cdot (5+3k) + 1 \cdot (5+k) \\
= \; & 0.5 \cdot h_1 \cdot (5+3k) + 0.5 \cdot (-1) \cdot (5+3k) + 1 \cdot (5+k) \\
= \; & 0.5 \cdot h_1 \cdot (5+3k) - 2.5 - 1.5k + 5 + k \\
= \; & 0.5 \cdot h_1 \cdot (5+3k) + 2.5 - 0.5k \\
= \; & 0.5 \cdot h_1 \cdot (5+3k) + 0.5 \cdot (5-k)
\end{aligned}$$

If the orbit distance is even, then we need $0.5 \cdot h_1$ Fast actions to reach the goal, for a total cost of $0.5 \times h_1 \times (5+3k)$.

In summary, h_3 is the minimum of $h_1 \times (5+k)$, the cost to the goal with only Slow actions, and m, the cost to the goal with only Fast actions (plus possibly one Slow action if h_1 is odd). Note that all Slow actions or all Fast actions (plus possibly one Slow) are the only two possible optimal solutions. If one Fast action is cheaper than two Slow actions, then the optimal solution is to use as many Fast actions as possible. If two Slow actions is cheaper than one Fast action, then the optimal solution is to use all Slow actions.

Thus h_3 is actually the exact optimal cost for this search problem. The exact cost to solve a search problem is admissible and consistent.

(iv) [2 pts] Which of the following heuristics will find the optimal solution the fastest?

○ h_1 ○ h_2 [✓ h_3]   ○ None of the above

h_3 is the exact cost, so it is the best possible heuristic for finding the optimal solution the fastest. (Recall that heuristics can be arranged in a hierarchy, where 0 is the worst heuristic and the true cost is the best heuristic.)

(b) Now, suppose there are many planets in space. Each planet can have a different number of orbits. Our goal is a specific orbit of a specific planet.

Now, the only actions available are $\{Accelerate\ Slow, Decelerate\ Slow, Transition\}$. The spaceship can only transition to a different planet from the current planet's outermost orbit, and it will land in the outermost orbit of the destination planet. The *Transition* action has a cost of $5+2k$.

(i) [2 pts] $h_4 = \begin{cases} 0 & \text{if orbiting the correct planet} \\ 1 & \text{otherwise} \end{cases}$

[✓ Admissible]   [✓ Consistent]   ○ Neither

If we're orbiting the correct planet, $h_4 = 0$, which is always an underestimate of the true cost.

If we're orbiting the incorrect planet, $h_4 = 1$. We'll need at least one Transition action to reach the correct planet, so the true cost is at least 5. $h_4 = 1 < 5 <$ true cost, so h_4 underestimates the true cost and therefore is admissible.

The maximum change in this heuristic is 1. The minimum cost of an action is 5, so this heuristic will never overestimate the cost of an action. Therefore, h_4 is consistent as well.

(ii) [2 pts] $h_5 = h_4 \cdot$ (min # of actions needed to go to the outermost orbit of the current planet)

[✓ Admissible]   [✓ Consistent]   Neither

Note that h_4 serves as an indicator variable (if you're not familiar with an indicator variable, think "if statement"): if $h_4 = 1$ (we're not orbiting the right planet), then h_5 is the minimum number of actions needed to reach the outermost orbit. If $h_4 = 0$ (we're orbiting the correct planet), then $h_5 = 0$.

If we're orbiting the correct planet, then $h_5 = 0$ is always an underestimate of the true cost.

If we're not orbiting the correct planet, then we first have to reach the outermost orbit of the current planet in order to take the Transition action. Therefore, the cost to reach the outermost orbit is an underestimate of the true cost (we have to reach the outermost orbit, then take extra actions that can only increase the cost).

Note that we can only use Slow actions to reach the outermost orbit (there are no more Fast actions). Each Slow action costs at least 5, so if we need x actions to reach the outermost orbit, the cost of reaching the outermost orbit is at least $5x$. Therefore, the number of actions to reach the outermost orbit is an underestimate of the true cost to reach the outermost orbit. h_5 is an underestimate of the true cost to reach the outermost orbit, which is itself an underestimate of the true cost to the goal orbit, so h_5 is admissible.

The maximum change in this heuristic is 1; if we take an action on the wrong planet, the number of actions to the outermost orbit can change by at most 1. If we're in the outermost orbit ($h_5 = 0$) and we transition to the correct planet ($h_5 = 0$), the heuristic changes by 0. If we're orbiting the correct planet, the heuristic always changes by 0. From the previous part, the minimum cost of an action is 5, so this heuristic will never overestimate the cost of an action. Therefore, h_5 is consistent.

(iii) [2 pts] $h_6 = (1 - h_4) \cdot (\text{min \# of actions needed to go to the goal orbit})$

[✓ Admissible]   Consistent   Neither

Similar to the previous part, h_4 is being used as an indicator variable. If we're orbiting the correct planet, h_6 is the minimum number of actions to reach the goal orbit. If we're not orbiting the correct planet, $h_6 = 0$.

If we're not orbiting the correct planet, $h_6 = 0$ is always an underestimate of the true cost.

If we're orbiting the correct planet, the number of actions to the goal state is an underestimate of the true cost to the goal state, because each action costs at least 5. (The reasoning is similar to the previous part.) Therefore, h_5 is admissible.

When considering whether h_6 is consistent, it helps to look for a situation where h_6 changes by a large amount, and then compare the change in h_6 with the actual cost of the action. We know that h_6 is high in outer orbits of the goal planet, and $h_6 = 0$ on non-goal planets. Consider a case where the goal planet has a huge number of orbits, and we're in the outermost orbit. Then h_6 is some huge number in the outermost orbit of the goal planet. If we transition out of the goal planet, h_6 will change from the huge number to 0. This arbitrarily large number could overestimate the true cost of the action, which is $5 + 2k$. Therefore, h_6 is not consistent.

(iv) [2 pts] $h_7 = \min(h_5, h_6)$

[✓ Admissible]   [✓ Consistent]   Neither

Note that at least one of h_5, h_6 is 0 at all times, because one is multiplied by h_4 and the other by $1 - h_4$ and h_4 can only be 0 or 1.

In particular, when we orbit the correct planet, $h_6 = 0$, so $\min(h_5, h_6) = 0$. When we orbit the incorrect planet, $h_5 = 0$, so $\min(h_5, h_6) = 0$.

Therefore, $h_7 = 0$, which is admissible (always underestimates the true cost) and consistent (never changes by more than 0, so never changes by more than an action cost).

Q4. [13 pts] Golden Bear Years

Pacman will be a freshman at University of Perkeley, so he is planning his 4 years in college. He loves playing video games, so he will only choose courses from the computer science department: $CS_1, CS_2, \ldots, CS_8$.

In this problem, the variables are the 8 courses, and their domains are the 4 years to take them. The constraints are listed below:

- CS_1 and CS_2 should be taken in Year 1.
- CS_3 should be taken in Year 3.
- CS_5 should be taken in Year 4.
- CS_5 and CS_6 should be taken in the same year.
- CS_5 and CS_7 should be taken in adjacent years, but it doesn't matter which course is taken first.
- CS_7 and CS_8 shouldn't be taken in the same year.

(a) [1 pt] How many binary constraint(s) are there in the above problem?

○ 0 ○ 1 ○ 2 [✓ 3] ○ 4 ○ 5

The last three constraints are binary constraints.

The first constraint can be decomposed into two unary constraints. You can check that CS_1 is taken in Year 1 without checking any other variables, and you can check that CS_2 is taken in Year 1 without checking any other variables.

The second and third constraints each only involve one variable, so they're unary constraints.

The fourth constraint requires checking both CS_5 and CS_6. The fifth constraint requires checking both CS_5 and CS_7. The sixth constraint requires checking both CS_7 and CS_8. All of these are binary constraints.

(b) [8 pts] Pacman has to take all 8 courses in 4 years, so he decides to run backtracking search with arc consistency. Select the values in the domains that will be **removed** after enforcing **unary constraints** and **arc consistency**. If no values are removed from the domain, select "No values removed" (write N).

(1) CS_1: ☐ 1 [✓ 2] [✓ 3] [✓ 4] ○ N No values removed

(2) CS_2: ☐ 1 [✓ 2] [✓ 3] [✓ 4] ○ N No values removed

(3) CS_3: [✓ 1] [✓ 2] ☐ 3 [✓ 4] ○ N No values removed

(4) CS_4: ☐ 1 ☐ 2 ☐ 3 ☐ 4 [✓ N No values removed]

(5) CS_5: [✓ 1] [✓ 2] [✓ 3] ☐ 4 ○ N No values removed

(6) CS_6: [✓ 1] [✓ 2] [✓ 3] ☐ 4 ○ N No values removed

(7) CS_7: [✓ 1] [✓ 2] ☐ 3 [✓ 4] ○ N No values removed

(8) CS_8: ☐ 1 ☐ 2 [✓ 3] ☐ 4 ○ N No values removed

First, we enforce unary constraints on CS_1, CS_2, CS_3, and CS_5:

CS1	1			
CS2	1			
CS3			3	
CS4	1	2	3	4
CS5				4
CS6	1	2	3	4
CS7	1	2	3	4
CS8	1	2	3	4

To enforce arc consistency, note that there are only 3 binary constraints and therefore 6 arcs that we need to check to begin with:

- $CS_5 \rightarrow CS_6$
- $CS_6 \rightarrow CS_5$
- $CS_5 \rightarrow CS_7$
- $CS_7 \rightarrow CS_5$
- $CS_7 \rightarrow CS_8$
- $CS_8 \rightarrow CS_7$

Enforcing $CS_5 \rightarrow CS_6$ causes no values to be removed from CS_5's domain.

Enforcing $CS_6 \rightarrow CS_5$ causes all values but 4 to disappear from CS_6's domain. Since we changed CS_6, we have to re-add any arcs pointing at CS_6 to the queue. The only arc pointing at CS_6 that has a constraint is $CS_5 \rightarrow CS_6$. Now, we have:

Domains:

CS1	1			
CS2	1			
CS3			3	
CS4	1	2	3	4
CS5				4
CS6				4
CS7	1	2	3	4
CS8	1	2	3	4

Queue:

- $CS_5 \rightarrow CS_7$
- $CS_7 \rightarrow CS_5$
- $CS_7 \rightarrow CS_8$
- $CS_8 \rightarrow CS_7$
- $CS_5 \rightarrow CS_6$

Next, enforcing $CS_5 \rightarrow CS_7$ removes no values from CS_5's domain.

Enforcing $CS_7 \rightarrow CS_5$ removes all values but 3 from CS_7's domain. Since we changed CS_7, we have to re-add any arcs pointing at CS_7 to the queue. The only arc pointing at CS_7 that has a constraint is $CS_5 \rightarrow CS_7$ (there's also $CS_8 \rightarrow CS_7$, but that's already on the queue). Now, we have:

Domains:

CS1	1			
CS2	1			
CS3			3	
CS4	1	2	3	4
CS5				4
CS6				4
CS7			3	
CS8	1	2	3	4

Queue:

- $CS_7 \rightarrow CS_8$
- $CS_8 \rightarrow CS_7$
- $CS_5 \rightarrow CS_6$
- $CS_5 \rightarrow CS_7$

Next, enforcing $CS_7 \rightarrow CS_8$ causes no values to be removed from CS_7's domain.

Enforcing $CS_8 \rightarrow CS_7$ causes 3 to be removed from CS_8's domain. Since we changed CS_8, we have to re-add any arcs pointing at CS_8 to the queue. The only arc pointing at CS_8 that has a constraint is $CS_7 \rightarrow CS_8$. Now, we have:

Domains:

CS1	1			
CS2	1			
CS3			3	
CS4	1	2	3	4
CS5				4
CS6				4
CS7			3	
CS8	1	2		4

Queue:

- $CS_5 \rightarrow CS_6$
- $CS_5 \rightarrow CS_7$
- $CS_7 \rightarrow CS_8$

At this point, the remaining three arcs in the queue are consistent, so we can remove them from the queue after checking them.

(c) [2 pts] Now, suppose University of Perkeley has provided n total CS courses (n is much larger than 8). Pacman doesn't worry about graduation, so he can spend d years at the school (d is much larger than 4). After taking the prerequisite courses (CS_1 to CS_8), the school has no other constraints on the courses Pacman can take. Now, what will be the time complexity of the AC-3 arc consistency algorithm?

- ◯ The time complexity is $O(n^2d^3)$ and cannot be further reduced.
- ◯ The time complexity is generally $O(n^2d^3)$, and can be reduced to $O(n^2d^2)$.
- [✓ The time complexity can be less than $O(n^2d^2)$.]

In the worst case, arc consistency takes time $O(n^2d^3)$, but can be reduced to $O(n^2d^2)$. However, this is the worst-case scenario, where we have to check every single arc (there are up to $O(n^2)$ arcs, and each arc takes $O(d^2)$ time), and we have to re-check every arc $O(d)$ times (once for each time a value is removed from a variable's domain).

In the CSP described, where there are very few constraints and lots of variables, the number of arcs we have to check (and re-check) can be lower than $O(n^2d^2)$.

(d) [2 pts] Suppose that Pacman is running local search to find a solution to this CSP, and currently has the following variable assignment. For the next iteration, he wants to change the assignment of one variable, which will lead to the fewest number of unsatisfied constraints remaining. Fill in the blank for the variable and the value it should change to.

$CS_1 : 3 \quad CS_2 : 2 \quad CS_3 : 4 \quad CS_4 : 4$

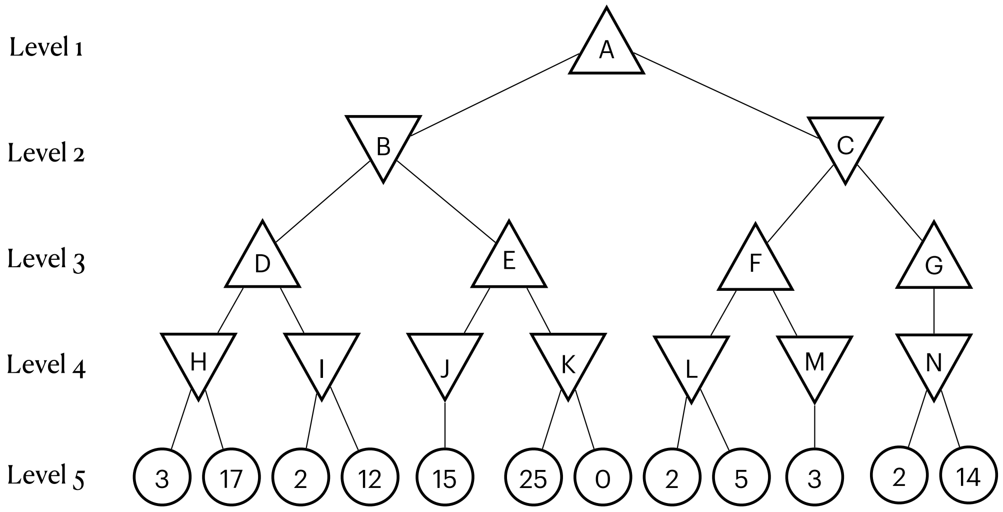
$CS_5 : 4 \quad CS_6 : 1 \quad CS_7 : 2 \quad CS_8 : 4$

The variable: $\boxed{CS_5}$ will be assigned to the value: $\boxed{1}$

Changing the value of CS5 from 4 to 1 could resolve constraint 4 and constraint 5, and it won't create any new constraints.

Q5. [8 pts] Games

(a) Consider the following game tree.



(i) [1 pt] What is the minimax value at node A?

$\boxed{3}$

$H = 3, I = 2 \rightarrow D = 3.$

$J = 15, K = 0 \rightarrow E = 15.$

$L = 2, M = 3 \rightarrow F = 3.$

$N = 2 \rightarrow G = 2.$

$B = \min(3, 15) = 3.$ $C = \min(3, 2) = 2.$

$A = \max(3, 2) = 3.$

(ii) [3 pts] Which branches will be pruned after running minimax search with alpha-beta pruning? For instance, if the edge between node A and node B is pruned, write $A - B$. If the edge between H and 3 is pruned, write $H - 3$. List the pruned branches from left to right. If a branch from an upper level is pruned, you don't have to list the branches below that.

$\boxed{I - 12, E - K, L - 5, C - G}$

$H - 3$ and $H - 17$ must be checked.

After $I - 2$, we know that $I \leq 2$. Given a choice between $H = 3$ and $I \leq 2$, the maximizer E will always prefer H, so we can stop looking at children of I. Thus $I - 12$ is pruned.

After $J - 15$, we know that $E \geq 15$. Given a choice between $E \geq 15$ and $D = 3$, the minimizer B will always prefer D, so we can stop looking at children of E. Thus $E - K$ is pruned.

After $L - 2$, we know that $L \leq 2$. If the right node of L is smaller or equal to 2, then it will be chosen by L, but this value will not eventually be chosen by A because we know that $B \geq 3$; If it is larger than 2, then 2 will be chosen.

Then we check out $M = 3$. We know that $C \leq 3$ and $B \geq 3$. Thus, the root node would prefer B over A, so $C - G$ will be pruned.

(iii) [2 pts] Suppose all the min nodes in layer 4 are changed to max nodes and all the max nodes in layer 3 are changed to min nodes. In other words, we have max, min, min, max as levels 1, 2, 3, 4 in the game tree. We then run minimax search with alpha-beta pruning.

Which of the following statements is true?

○ The same set of leaf nodes will be pruned, because this is still a minimax problem and running alpha-beta pruning will result in the same set of leaf nodes to be pruned.

[✓ ● A different set of leaf nodes will be pruned.]

◯ No leaf nodes can be pruned, because pruning is only possible when the minimizer and maximizer alternate at each level.

◯ None of the above

A different set of nodes will be pruned. We first draw the new game tree. We see that level 2 and level 3 can be merged into one level since they are both minimizers. All of the principles of minimax with alpha-beta pruning still applies to the new tree. In the new tree, the right-most three nodes can be pruned.

(iv) [2 pts] Now, consider another game tree which has the same structure as the original game tree shown above. This modified game tree can take on any values at the leaf nodes. What is the minimum and the maximum number of leaf nodes that can be pruned after running alpha-beta pruning?

Minimum leaf nodes pruned: $\boxed{0}$ Maximum leaf nodes pruned: $\boxed{5}$

Minimum number of nodes that can be pruned is 0 because there could always be such leaf values that requires all leaf nodes to be checked. Maximum number of nodes that can be pruned is 5 because the nodes with values 3, 17, 2, 15, 2, 5, 3 in the diagram will always have to be checked no matter what values the leaf nodes take on. Another assignment option for maximum pruning is "3, 17, 2, 15, 2, 5, 2"(e.g. B=3, L=100, the left node of N=1).

Q6. [12 pts] Indiana Jones & the Kingdom of the Crystal Skull

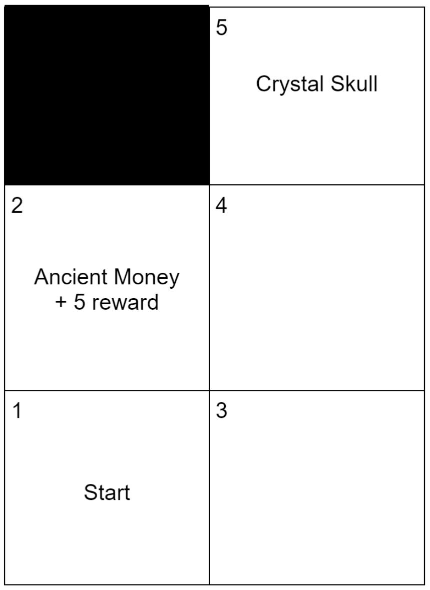
Help Indiana Jones find the crystal skull that was hidden somewhere at the old frozen lake!

For all parts of the problem, assume that value iteration begins with all states initialized to zero, and $\gamma = 1$.

(a) Suppose that we are performing value iteration on the grid world MDP below. Indiana starts at the bottom-left part of the lake labeled 1.

Indiana found an ancient manuscript detailing the rules of this MDP. **(All of these rules are also described in the transition table below.)**

- Indiana can only go up or right, unless the action would cause Indiana to move off the board or into the black square.
- Because the lake is frozen, if Indiana moves up from Square 3, it's possible (with probability x) that Indiana slips and actually moves two squares up.
- From Squares 1, 2, and 4, Indiana deterministically moves one square in the chosen direction.
- Once Indiana reaches Square 5, the only action available is Exit, which gives reward 100. (There are no actions available after exiting.)
- Square 2 contains ancient money, so any action that moves Indiana into Square 2 gives reward +5.
- All other transitions give reward -4 if Indiana moves two squares, and -10 if Indiana moves one square.



s	a	s'	$T(s,a,s')$	$R(s,a,s')$
1	Up	2	1.0	+5
1	Right	3	1.0	-10
2	Right	4	1.0	-10
3	Up	4	$1-x$	-10
3	Up	5	x	-4
4	Up	5	1.0	-10
5	Exit	End	1.0	100

(i) [2 pts] Find the optimal state values for Square 2 and Square 3. Your answer can be in terms of x.

$V^*(2)$: $\boxed{80}$ $V^*(3)$: $\boxed{80+16x}$

To compute $V^*(2)$, note that the optimal policy is to go right (in fact, that's the only available action from 2). After moving right, Indiana reaches 4, where the optimal policy is to move up (in fact, that's the only available action from 4). After moving up, Indiana reaches 5, where the optimal policy is to exit. In total, this deterministic path has reward $V^*(2) = -10 + -10 + 100 = 80$.

To compute $V^*(3)$, note that the optimal policy is to go up (in fact, that's the only available action from 3).

Case I: With probability $1-x$, we get reward -10 and land in 4. From 4, the remaining moves in the MDP are deterministic: we move from 4 to 5 and exit. In total, we earn reward $-10 - 10 + 100 = 80$, and Case I gives expected reward of $(1-x)(80) = 80 - 80x$.

Case II: With probability x, we get reward -4 and land in 5. From 5, we exit and earn reward 100, and Case II gives expected reward of $x(-4 + 100) = 96x$.

In total, $V^*(3) = 80 - 80x + 96x = 80 + 16x$.

(ii) [2 pts] For which values of x would Indiana prefer to go right on Square 1?

$\boxed{\frac{15}{16}} < x$

Going right at Square 1 lands Indiana in Square 3, which gives immediate reward -10 and future expected reward from 3 of $80 + 16x$. Going up at Square 1 lands Indiana in Square 2, which gives immediate reward +5 and future expected reward from 2 of 80.

In order to prefer going right, we need $-10 + 80 + 16x > 5 + 80$.

Simplifying, we get $70 + 16x > 85 \rightarrow 16x > 15 \rightarrow x > 15/16$.

(b) [6 pts] The rest of this question is independent of the previous subpart.

Seeing how Indiana figures out the path to the skull, the ancient powers of the skull start to confuse Indiana.

Now, when Indiana takes an action, **the action is changed to a different action** according to a particular probability distribution denoted $p(a'|s,a)$. Given state s and action a, the action is changed to a' with probability $p(a'|s,a)$. To confuse him further, Indiana will receive reward as if his action did not change at all.

Indiana tries to adopt those changes to Q-value iteration.

$$\text{Regular MDP:}\quad Q(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \max_{a'} Q(s',a')\right]$$

$$\text{New formulation:}\quad Q(s,a) \leftarrow \textbf{(i) (ii) (iii) (iv)}\left[\textbf{(v)} + \textbf{(vi)}\right]$$

(i)	○ 1	[✓ $\sum_{a'}$]	○ $argmax_a$
(ii)	[✓ $p(a'\|s,a)$]	○ $T(s,a,s')$	○ $\sum_{s'}$
(iii)	○ $T(s,a',s')$	[✓ $\sum_{s'}$]	○ 1
(iv)	[✓ $T(s,a',s')$]	○ $p(a'\|s,a)$	○ γ
(v)	○ $R(s,a',s')$	○ γ	[✓ $R(s,a,s')$]
(vi)	[✓ $\gamma * max_{a''}Q(s',a'')$]	○ $\gamma * Q(s',a')$	○ $\gamma * argmax_a Q(s',a)$

$$Q(s,a) \leftarrow \sum_{a'} p(a'|s,a) \sum_{s'} T(s,a',s')\left[R(s,a,s') + \gamma * max_{a''}Q(s',a'')\right]$$

Because we don't know what action will be taken, we need to take a weighted average over the possible actions that are taken. This introduces the $\sum_{a'} p(a'|s,a)$ term.

Once the action is taken, we have to sum over the possible states resulting from the action. This results in the $\sum_{s'}$ term (this reasoning is just like in regular MDPs).

To take a weighted average over the possible states resulting from the action, we need to use a' (since we're taking the changed action). This introduces the $T(s,a',s')$ term.

After landing in state s', we receive reward as if we had taken our original action (a). This introduces the $R(s,a,s')$ term.

After landing in state s', we want to act optimally from s'. This is unchanged from the standard MDP equation, but we already used the variable a' in our equation, so we've introduced a new variable a''. In other words, in standard MDPs, a' was used to represent the best action from s'. However, in modified MDPs, a' was already used to represent the changed actions, so we use a'' instead to represent the best action from s'. This introduces the $\gamma * \max_{a''} Q(s',a'')$ term.

(c) [2 pts] The rest of this question is independent of the previous subpart.

Indiana's enemies are not sleeping and they also are trying to get the skull power. They are also trying Q-value iteration. They heard old stories about how people approaching the lake lose control of their movement.

Consider a modified MDP where the agent can choose any action a, but the next state s' will be determined regardless of the action. In other words, $T(s,a,s')$ can be changed into $T(s,s')$, but $R(s,a,s')$ stays the same. Looking at the policy extraction step, Indiana's enemies noted that to get the policy, they do not even need any Q-values or V-values.

Write an expression for deriving the optimal policy **without using any Q-values or V-values**.

$$\text{Regular MDP:}\quad \pi(s) \leftarrow argmax_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma * max_{a'}Q(s',a')\right]$$

$$\boxed{\pi(s) \leftarrow argmax_a \textstyle\sum_{s'} T(s,s')R(s,a,s') \qquad \text{or} \qquad \pi(s) \leftarrow argmax_a \textstyle\sum_{s'} R(s,a,s')}$$

Since "the next state will be determined regardless of the action," we cannot control the successor s' and, therefore, to plan for the future. Thus, we remove the $\gamma * \max_{a'} Q(s',a')$ term and just focus on the reward of current action, which is $R(s,a,s')$. The answer can be $argmax_a \sum_{s'} R(s,a,s')$. Because $T(s,s')$ is the same for a given s, $\pi(s) \leftarrow argmax_a \sum_{s'} T(s,s')R(s,a,s')$ can also be considered as a correct answer.

Q7. [12 pts] Reinforcement Learning

In this class, we have studied several reinforcement learning (RL) methods. In this problem, we will examine properties of several extensions of these approaches.

(a) [4 pts] Select all true statements about Q-learning.

For this subpart, assume that during Q-learning, we visit every state-action pair an infinite number of times.

- ■ At convergence, Q-values obtained from Q-learning will lead to the optimal policy of the MDP, π^*.
- ■ At convergence, Q-learning always finds a deterministic policy.
- ☐ Q-learning cannot converge in any MDP when $\gamma = 1$.
- ☐ Assume that you derive a deterministic policy from a Q-function Q. The policy is denoted by $\pi(s) = \arg\max_a Q(s,a)$. Then, the learned Q-values $Q(s,a)$ for $a \neq \pi(s)$ represent the expected discounted reward of a valid policy in the MDP.
- ■ When running policy iteration, the $Q(s,a)$ values in the table of Q-values at intermediate iterations represent the correct Q-function for some policy in the MDP.
- ○ None of the above

The first statement is true, the second statement is true since the policy returned by Q-learning: $\pi(a|s) = \arg\max_a Q(s,a)$ chooses only one action at every state and hence is deterministic. For the third statement, note that one can construct trivial MDPs, where the reward is 0 at each state, for every action, where Q-learning would not diverge. For the fourth statement, the answer is false: even at convergence, the Q-values at actions not chosen by the arg-max policy are not the estimates of long-term return. For the fifth statement, policy iteration computes Q^π for any given policy, and the improves the policy against it.

(b) In this subpart, we will consider how standard Q-learning will behave in two different types of MDPs:

- Infinite-horizon MDP **with terminals**: there are certain states, denoted as $\mathcal{S}_T$, such that if the agent reaches these states, it terminates and gets no reward.
- Infinite-horizon MDP, **without terminals**: there are no states where the MDP terminates.

Formally, for any state s in the MDP, we define a function $\tau(s) \in \{0,1\}$. In other words, $\tau(s) = 0$ or $\tau(s) = 1$ for all s. $\tau(s) = 1$ implies that $s \in \mathcal{S}_T$ (i.e. s is a terminal state).

(i) [2 pts] Fill in the blank for the Q-learning update with terminations, in terms of $\tau(s)$.

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[R(s,a) + \gamma \boxed{} \max_{a'} Q(s',a')\right]$$

$1 - \tau(s')$

Recall that in standard Q-learning, the $\gamma \max_{a'} Q(s',a')$ term represents the expected discounted future reward for starting in state s' (where you end up after taking the action), and acting optimally after that.

If we arrive at an s' that is a terminal state, then the expected discounted future reward should be set to 0. If we arrive at an s' that is not a terminal state, then the expected discounted future reward should be the same as in standard Q-learning.

We need a factor that causes the future reward term to be 0 if s' is a terminal state, and 1 if s' is not a terminal state. Note that $1 - \tau(s') = 0$ if s' is a terminal state (and $\tau(s') = 1$), and $1 - \tau(s') = 1$ if s' is not a terminal state (and $\tau(s') = 0$).

(ii) [4 pts] Consider two MDPs, $\mathcal{M}_1$ and $\mathcal{M}_2$, which are identical, except that their reward functions are shifted versions of each other:

- For every state-action pair (s, a) in $\mathcal{M}_1$, $0 \leq R_1(s, a) \leq 1$
- For every state-action pair (s, a) in $\mathcal{M}_2$, $R_2(s, a) = R_1(s, a) - 2.0$

Assume there are no cycles in the MDP (i.e. you cannot keep accumulating reward infinitely).

True or false: The policies found by Q-learning on $\mathcal{M}_1$ and $\mathcal{M}_2$ **without terminals** are **not** identical.

- ○ True, because reward functions are not the same in the two MDPs.
- ○ True, because Q-learning would not accumulate negative reward.
- ○ True, but not for the reasons above.
- [✓ False, shifting a reward function does not affect the optimal policy.]
- ○ False, because the reward function is now negative.
- ○ False, but not for the reasons above.

False. Shifting the rewards for all states in an infinite horizon MDP does not change the policy found by Q-learning. Note that to compute the optimal policy, Q-learning takes the max over all the Q-values in the resulting state s'. Shifting all rewards by some constant amount doesn't change the action chosen by the max function.

True or false: The policies found by Q-learning on $\mathcal{M}_1$ and $\mathcal{M}_2$ **with terminals** are **not** identical.

- ○ True, because reward functions are not the same in the two MDPs.
- [✓ True, because there are MDPs where we might terminate instead of attaining negative reward.]
- ○ True, but not for the reasons above.
- ○ False, shifting a reward function does not affect the optimal policy.
- ○ False, because the reward function is now negative.
- ○ False, but not for the reasons above.

For this part, note that since we have made the rewards negative, the resulting policy will prefer early terminations, so policies found by Q-learning in the two MDPs will be different.

Another way to arrive at this answer is to note that the terminal states have expected future reward of 0. When we shift the rewards of all the other states/actions, we don't shift the rewards of the terminal states, so the terminal states can go from being worse than the other states/actions, to being better than the other states/actions.

(c) [2 pts] Consider this modification to the Q-learning update:

$$Q(s, a) \leftarrow \alpha \cdot \left(R(s, a, s') + \gamma \frac{\sum_{a'} Q(s', a')}{\|a'\|} \right) + (1 - \alpha) \cdot Q(s, a)$$

where $\|a'\|$ is the number of actions available from state s'.

If we run Q-learning with this update on an MDP, the learned policy is obtained by: $\pi'(s) = \max_a Q(s, a)$.

Would $\pi'(s)$ be the optimal policy?

- ○ Yes, but only if the MDP has deterministic transitions.
- ○ Yes, but only if the MDP does not have cycles.
- ○ Yes, but only if we visit every state-action pair infinitely often, and we decrease the learning rate appropriately.
- ○ No, $\pi'(s)$ will be the least optimal policy.
- [✓ No, $\pi'(s)$ will be an arbitrary policy that may be random, optimal or sub-optimal.]
- ○ No, $\pi'(s)$ will be a greedy policy that always maximizes the immediate next reward (ignoring future rewards).

The only change between the standard Q-learning update and the modified update is the term multiplied by γ. $\max_{a'} Q(s', a')$ has been replaced by $\frac{\sum_{a'} Q(s', a')}{\|a'\|}$.

Instead of taking the maximum Q-value from state s', we're taking the average of all the Q-values from state s'.

The max term is needed in order to compute the maximum expected reward starting from the new state s'. By replacing the max term with an average, we're calculating the value of each episode to be the immediate reward from that episode, plus the expected reward for choosing an action at random from the new state s'. Without the max term, we've lost the part of the Q-learning update that computes the best value (the value of acting optimally) from the new state s'.

This will likely produce a policy that does not attain the optimal policy, but may still be optimal in some MDPs (e.g., if the MDP only has self-loops), and may just be sub-optimal in other MDPs.