

Solutions last updated: Sunday, October 15

- You have 110 minutes.
- The exam is closed book, no calculator, and closed notes, other than two double-sided cheat sheets that you may reference.
- Anything you write outside the answer boxes or you ~~cross-out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we will grade the worst interpretation.

For questions with **circular bubbles**, you may select only one choice.

- Unselected option (completely unfilled)
- Only one selected option (completely filled)
- Don't do this (it will be graded as incorrect)

For questions with **square checkboxes**, you may select one or more choices.

- You can select
- multiple squares (completely filled)

First name	
Last name	
SID	
Name of person to the right	
Name 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 my cheat sheets), 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	14
Q2. Search: Dwinelle Hall Maze	17
Q3. CSPs: Splitting CSPs on a Constraint	22
Q4. Multi-Agent Search: Faulty Functions	16
Q5. MDPs: One Piece, Zero Reward	15
Q6. Reinforcement Learning: Clumsy Test Taker	16
Total	100

Q1. [14 pts] Potpourri

(a) [1 pt] True or false: Local search has no optimality guarantees, so we always prefer to use A* search over local search.

- True False

False. If we have computation/time limits, local search might be the only way to get a solution under the constraints. Also, if we only need a “reasonably good” solution and not a perfect solution, then local search might work better than A*. Also, A* requires designing a heuristic; if we don’t have one, we may prefer to use local search.

(b) [1 pt] True or false: In Monte Carlo tree search, all else equal, performing more rollouts from a state will generally result in a better estimate for the value of that state.

- True False

True. To estimate the value of a state, we check the fraction of rollouts that result in a win. If we perform more rollouts, we should generally expect our fraction to be a closer approximation of the true value of a state.

(c) [1 pt] True or false: When performing a rollout in Monte Carlo tree search, we must play moves according to the optimal policy.

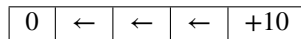
- True False

False. We play moves according to some fast (possibly random) policy.

In the next two subparts, consider the grid world below. Reminder: in a grid world, there is only one action, “Exit”, available from the squares labeled +10 and 0, that gives the rewards of +10 and 0, respectively.

Assumptions: The living reward for every action is 0. All actions succeed with probability 1.0. The discount factor $\gamma = 1$.

We want to run policy iteration, starting at the policy shown below. In case of ties, we will choose left instead of right.

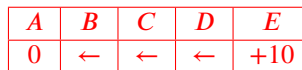


(d) [2 pts] What is the resulting policy after one iteration of policy iteration?

- | | | | | | | | | | | | |
|--|---|---|---|-----|-----|---|---|---|---|---|-----|
| <input type="radio"/> <table border="1" style="border-collapse: collapse; text-align: center; width: 150px;"> <tr><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">←</td><td style="padding: 2px 10px;">←</td><td style="padding: 2px 10px;">←</td><td style="padding: 2px 10px;">+10</td></tr> </table> | 0 | ← | ← | ← | +10 | <input type="radio"/> <table border="1" style="border-collapse: collapse; text-align: center; width: 150px;"> <tr><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">→</td><td style="padding: 2px 10px;">←</td><td style="padding: 2px 10px;">←</td><td style="padding: 2px 10px;">+10</td></tr> </table> | 0 | → | ← | ← | +10 |
| 0 | ← | ← | ← | +10 | | | | | | | |
| 0 | → | ← | ← | +10 | | | | | | | |
| <input checked="" type="radio"/> <table border="1" style="border-collapse: collapse; text-align: center; width: 150px;"> <tr><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">←</td><td style="padding: 2px 10px;">←</td><td style="padding: 2px 10px;">→</td><td style="padding: 2px 10px;">+10</td></tr> </table> | 0 | ← | ← | → | +10 | <input type="radio"/> <table border="1" style="border-collapse: collapse; text-align: center; width: 150px;"> <tr><td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">→</td><td style="padding: 2px 10px;">→</td><td style="padding: 2px 10px;">←</td><td style="padding: 2px 10px;">+10</td></tr> </table> | 0 | → | → | ← | +10 |
| 0 | ← | ← | → | +10 | | | | | | | |
| 0 | → | → | ← | +10 | | | | | | | |

Intuitive answer: Policy improvement does a one-step lookahead. Only the state directly left of the +10 will be able to see the +10 reward from the one-step lookahead.

Complete solution: In this solution, we’ll label the states as follows:



We first run policy evaluation to learn the value of the current policy, π_i . The expected discounted reward for going left is 0, so we have $V^{\pi_i}(B) = V^{\pi_i}(C) = V^{\pi_i}(D) = 0$. We also have $V^{\pi_i}(A) = 0$ and $V^{\pi_i}(E) = 10$ since the only actions available from those two states are to Exit and collect the reward.

Then, we run a one-step look-ahead to compute the new policy, π_{i+1} . The general equation for policy improvement is:

$$\pi_{i+1}(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^{\pi_i}(s')]$$

But in this question, we have deterministic actions, so the expected value computation over s' is unnecessary. We can also plug in $\gamma = 1$. Also, we only have two actions from every state, (\leftarrow and \rightarrow), so we can expand out the argmax expression. Finally, we know the living reward is always 0, so we can ignore the $R(s, a, s')$ term as well for non-exit actions. This simplifies the policy improvement equation to:

$$\pi_{i+1}(s) = \arg \max_a [V^{\pi_i}(\text{state left of } s), V^{\pi_i}(\text{state right of } s)]$$

In words: if the state left of s has higher value, the policy is to go left. Otherwise, if the state right of s has higher value, the policy is to go right. In case of a tie, we'll choose to go left, as the tiebreaker in the question suggests.

Plugging in values for each state:

$$\pi_{i+1}(B) = \arg \max_a [V^{\pi_i}(A), V^{\pi_i}(C)] = \arg \max_a [0, 0] = \leftarrow$$

$$\pi_{i+1}(C) = \arg \max_a [V^{\pi_i}(B), V^{\pi_i}(D)] = \arg \max_a [0, 0] = \leftarrow$$

$$\pi_{i+1}(D) = \arg \max_a [V^{\pi_i}(C), V^{\pi_i}(E)] = \arg \max_a [0, 10] = \rightarrow$$

- (e) [2 pts] Starting at the initial policy shown above (always go left), how many iterations of policy iteration are needed to compute the optimal policy?

In other words, if π_0 is the initial policy, select the minimum k such that $\pi_k = \pi^*$.

- 1 3 More than 4
 2 4

The first iteration changes the policy at the square directly left of +10. The next iteration changes the policy at the square two left of +10. Then, the third and final iteration changes the policy at the square three left of +10.

In the next two subparts, consider an agent performing Q-learning using one of these exploration strategies:

- Strategy A: Epsilon-greedy, where ϵ is set to a fixed value. You can assume ϵ , the probability of choosing a random action, is set such that $0.5 < \epsilon < 1$.
- Strategy B: Epsilon-greedy, where ϵ starts at the same fixed value, but decreases over time.

- (f) [1 pt] What happens if we run Strategy A forever?

- We prioritize exploration forever, and never exploit what we've learned.
 We prioritize exploiting what we know forever, and never explore new states.
 We make random moves every time.

Since $\epsilon < 1$, we are not making random moves all the time. But since we keep ϵ fixed forever, we keep exploring new states and never focus our efforts on exploiting what we know.

- (g) [1 pt] Which strategy will result in higher regret if we act according to that strategy forever?

Hint: Regret is the absolute difference between the expected reward for following the optimal policy, and the expected reward for acting according to the given strategy.

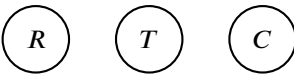
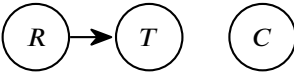
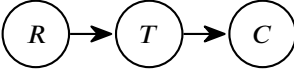
- Strategy A The regret is the same
 Strategy B

In Strategy A, we'll keep doing silly random actions forever, leading to lots of regret accumulating. In Strategy B, we eventually decay ϵ to 0, at which point we'll be acting according to a non-random policy, which should result in lower regret.

Note for future semesters: In this semester, Bayes' Nets were in scope for the midterm, but Bayes' Nets were covered close to the midterm, and students didn't get practice problems on Bayes' Nets before the midterm. Therefore, we explicitly promised students that Bayes' Nets would be lightly tested on the midterm, and the questions on the exam would be answerable just by watching lecture. This Bayes' Net question is not indicative of a Bayes' Net question you would see if the topic were fully in-scope (this question is substantially easier).

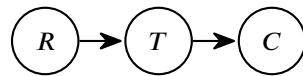
In the next two subparts, consider the following scenario: Rain (R) causes traffic (T). Rain and traffic have no effect on cavity (C).

(h) [3 pts] Select all of the Bayes' nets that can represent a joint distribution with the assertions given above.

- 
- 
- 
- None of the above

- Option 1: False. In this model, rain and traffic are marginally independent, which is not consistent with the scenario.
- Option 2: Using the intuitive causal interpretation of arrows in a Bayes' net, this Bayes' net directly represents the scenario described.
- Option 3: True. Adding an additional arrow only increases the set of joint distributions we can represent. Since this scenario was representable with just the $R \rightarrow T$ arrow, this scenario is still representable when we additionally add the $T \rightarrow C$ arrow.

(i) [2 pts] In this subpart, consider the following Bayes' net:



Suppose R and C are binary random variables (each variable has two possible values), and T is a random variable with three possible values.

Select all true statements about the sizes of the conditional probability tables (CPTs) associated with nodes in the Bayes' nets shown above.

- Every CPT has the same number of entries.
- The CPT associated with C has 6 entries.
- None of the above

- In FA23, Lectures 12-13, we showed how to compute the size of a conditional probability table.
- $P(R)$ has two entries, because R is a random variable with 2 values.
- $P(T|R)$ has 6 entries. We need one entry for every possible combination of T, R values. For each of the 3 values of T , there is an entry for each of the 2 values of R , for a total of $3 \cdot 2 = 6$ entries.
- $P(C|T)$ has 6 entries for a similar reason. For each of the 3 values of T , there are 2 entries (one for each value of C).
- Option 1 is false. $P(R)$ has 2 entries, but $P(T|R)$ and $P(C|T)$ each have 6 entries.
- Option 2 is true. $P(C|T)$ has 6 entries as described above.

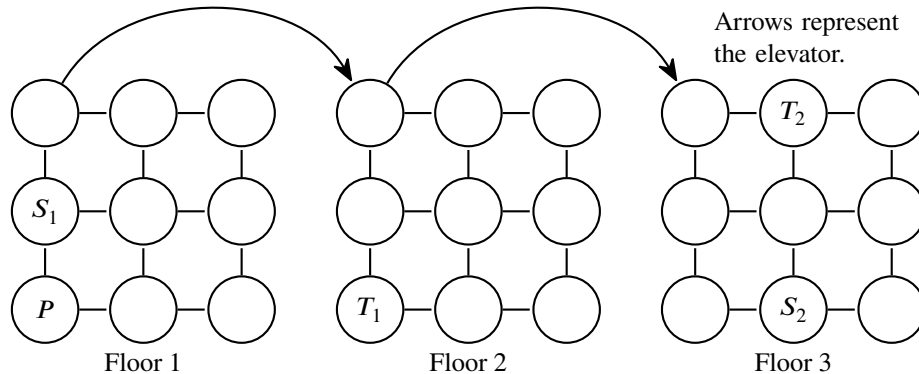
Q2. [17 pts] Search: Dwinelle Hall Maze

Some of Pacman's friends are lost in Dwinelle Hall and need help to make it to their classes!

Consider a 3-dimensional $M \times N \times F$ grid. You can think of this grid as F 2-dimensional $M \times N$ grids stacked on top of each other (where each 2-dimensional grid represents one floor of Dwinelle Hall).

Pacman (labeled P in the diagram) always starts at the bottom-left square of Floor 1. Pacman can take these actions in the grid (all actions have cost 1):

- Move to an adjacent square on the same floor (north, south, east, or west).
- From the top-left corner in any floor, take an elevator to the top-left corner of the floor directly above. The elevator cannot go down.



An example search graph is shown above, with $F = 3$ floors, and a $M \times N = 3 \times 3$ grid on each floor. **Your answers in this question should work for any arbitrary $M \times N \times F$ grid**, not just the example shown.

There are k Pacfriends, and each Pacfriend has a starting and target location (not necessarily unique). In the example above, we've shown two Pacfriends, with starting locations labeled S_1, S_2 , and target locations labeled T_1, T_2 .

Pacman's goal is to escort each Pacfriend from their starting location to their target location.

- When Pacman enters a square with a Pacfriend, Pacman automatically "picks up" the Pacfriend, who will now start to share Pacman's location as Pacman takes actions.
- When Pacman enters a Pacfriend's target location, Pacman automatically "drops off" the Pacfriend, who will stay at the target location.
- Pacman can be escorting (have picked up, but not dropped off) multiple Pacfriends at the same time.

(a) [2 pts] What is the maximum branching factor in this search problem?

4

For non-elevator squares, there are up to 4 actions available.

For the elevator square, there are two same-floor actions available (only two same-floor actions available from the top-left corner), plus one elevator action available, for a total of 3 actions available.

The question asks for the maximum branching factor, and $\max(3, 4) = 4$.

(b) [2 pts] Select all true statements about this search problem.

- Depth-first tree search is guaranteed to find a solution, or report that no solutions exist.
- Breadth-first tree search will expand at least as many nodes as breadth-first graph search.

None of the above

The search graph contains cycles (Pacman could wander in circles on one floor), so DFS could run forever.

Breadth-first graph search will expand fewer nodes because we can avoid expanding the same state repeatedly.

(c) [2 pts] In this subpart, suppose that all Pacfriends' starting and target locations are on Floor 1. Select all true statements about this specific problem.

- Depth-first graph search will not explore any nodes on higher floors.
- Breadth-first graph search will not explore any nodes on higher floors.
- None of the above

DFS could go down a path to a different floor and explore paths on other floors before finding a solution.

BFS could explore nodes on other floors if they are closer than nodes on the first floor.

(d) [4 pts] Select each state space representation (not necessarily the most efficient representation) that could be used to model this search problem.

Each answer choice is a different state space representation.

- Locations of Pacman and every Pacfriend
- Pacman location, and one Boolean variable for each Pacfriend
- Pacman location, and two Boolean variables for each Pacfriend
- Location of every Pacfriend
- None of the above

Option 1: True. The world state is fully specified by the locations of every character (Pacman and Pacfriends).

Option 2: False. At least two boolean values are needed (see next option).

Option 3: True. For each Pacfriend, keep track of whether they have been picked up by Pacman (false = they're in the starting location, true = they're in Pacman's location). Also, keep track of whether they have been dropped off by Pacman (false = they're in Pacman's location or the starting location, true = they're in the target location).

Option 4: False. Without Pacman's location, we couldn't write a valid successor function.

(e) [3 pts] Consider the state space representation: Pacman location, and 5 Boolean variables for each Pacfriend.

(Note: this may or may not be a correct or minimal state space representation.)

How many possible states exist in this representation? Your answer could be in terms of M, N, F, k .

$MNF \cdot 2^{5k}$

Pacman could be in any of the locations in the maze. There are MNF different locations in the maze.

Pacfriends, solution 1: If there are k Pacfriends and we store 5 Boolean values per Pacfriend, then we store $5k$ Boolean values in total. There are $2^{(5k)}$ ways in which the $5k$ Boolean values could be assigned.

Pacfriends, solution 2: Each Pacfriend has 5 Boolean values, which could be assigned in 2^5 different ways. With k Pacfriends, we have $\underbrace{2^5 \cdot 2^5 \dots \cdot 2^5}_{k \text{ times}} = (2^5)^k$ possible assignments to the Boolean values

(f) [4 pts] Select all admissible heuristics for this problem.

- Minimum distance between any Pacfriend's current location and their target location
- Maximum distance between any Pacfriend's current location and their target location
- Number of the highest floor, minus number of the floor Pacman is currently on
- Maximum difference between any Pacfriend's current floor and their target location's floor
- None of the above

Option 1: True. Pacman must eventually travel this distance in order to move the Pacfriend in question (who is closest to their target among all Pacfriends) to its target location. It could help to think about a relaxed problem here, where Pacman only needs to transport a single Pacfriend to its target location. In the relaxed problem, this heuristic is admissible, so this heuristic is also admissible in the standard problem.

Option 2: True. Pacman must eventually travel this distance to move the Pacfriend in question (who is furthest to their target among all Pacfriends) to its target location. Again, the relaxed problem heuristic may be helpful.

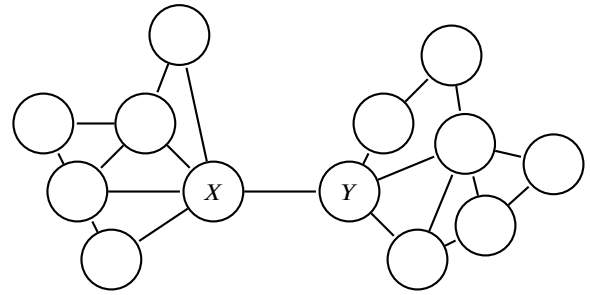
Option 3: False. Consider a case where there are multiple floors, but all the Pacfriends are on Floor 1 (as we saw in one of the earlier subparts). Then, after Pacman is finished escorting the last Pacfriend to its target location, the true cost to the goal is 0, but the heuristic would be nonzero (number of highest floor minus floor 1 would be nonzero). This counterexample shows that the heuristic overestimates the true cost to the goal in at least one case: therefore, the heuristic is not admissible.

Option 4: True. Pacman will eventually need to escort every Pacfriend from its current floor to its target location's floor, which will involve taking the elevator from the current floor to the target location's floor. Since each elevator action costs 1, Pacman's true cost will be greater than or equal to the difference in floors (target location floor minus current floor).

Q3. [22 pts] CSPs: Splitting CSPs on a Constraint

Consider a CSP with $L + R$ variables. There are d possible assignments to each variable. The CSP only has binary constraints.

We notice that if we remove one of the binary constraints, then the CSP can be decomposed into two independent subproblems, with no constraints between the subproblems: a “left” CSP with L variables and a “right” CSP with R variables. Let X and Y be the variables in the left and right subproblems, respectively, connected by the constraint we removed.



An example is shown to the right (but your answers should not assume we’re using this example). If we remove the binary constraint relating X and Y , this CSP can be decomposed into two independent subproblems. The “left” CSP includes X and has 6 variables. The “right” CSP includes Y and has 7 variables.

- (a) [2 pts] Without removing any constraints, we assign a value to one of the variables on the left side of the CSP. Then, we run forward checking.

What is the maximum number of variables whose domains could change as a result of running forward checking? (Do not count the variable whose value was just assigned.)

- 0
 1
 L
 R
 $L + R - 1$

Recall that forward checking only changes the domains of a variable if that variable has a constraint relating it to the variable that was just assigned.

Suppose we assigned the variable on the left side that’s connected to the right side (in the diagram, this variable is labeled X).

This variable could be connected to all $L - 1$ variables on the left side, in addition to the 1 variable on the right side. This gives a maximum of $(L - 1) + 1 = L$ variables whose domains could change as a result of running forward checking.

Side note (not necessary for answering the question), for why the answer is L and not $L - 1$: If we assign one of the variables on the left side that is not connected to the right side (in the diagram, a variable on the left that’s not X), then it would only have up to $L - 1$ other variables that it could be connected to. Since the question asks for the maximum number of variables whose domain could change, the answer is still L .

For the next two subparts: Without removing any constraints, we assign a value to one of the variables on the left side of the CSP. Then, we enforce arc consistency on the entire CSP.

- (b) [3 pts] Select all true statements.

- If the variable we select is not X , the domains of all variables on the right side of the CSP will be unchanged.
- If the variable we select is X , the domains of all variables on the right side of the CSP will be unchanged.
- After enforcing arc consistency, it is guaranteed that we can find a solution to the CSP without performing any more backtracking search.
- None of the above

Options 1-2: False. When enforcing arc consistency, the consequences of changing domains could propagate across multiple constraints, through the $X - Y$ constraint linking the two sides, into the other side of the CSP.

As a concrete example of propagation between the two sides, suppose the example CSP drawn were a map coloring CSP with inequality constraints. A node on the left side connected to X has domain: red, blue, green. The node X has domain: blue, green. The node Y has domain: blue.

We assign the left side node connected to X to the value green. Now, we check the arc from X to the newly-assigned variable, and X loses value green, leaving only blue in X ’s domain. Next, we check the arc $Y \rightarrow X$. This will cause Y to lose value blue (assigning Y to blue leaves no valid assignment in X ’s domain). This example shows that assigning a non- X variable on the left side could cause a variable on the right side (here, Y) to have its domain changed.

Option 3: False. In general, enforcing arc consistency is not sufficient to solve a CSP, and the structure of this CSP does not change that fact.

As a concrete example, we could extend the example from lecture, where each variable has red and blue left in its domain, the CSP is arc-consistent, but no solutions exist. Suppose every variable in the example CSP shown only had red and blue left in its domain after enforcing arc consistency. The CSP is arc consistent, but no solutions exist, and we won't know that no solutions exist unless we run more backtracking search.

(c) [2 pts] If we use the AC3 algorithm to enforce arc consistency, what is the maximum number of arcs we would need to place on the queue at the beginning of the algorithm?

- | | |
|--------------------------------------|--|
| <input type="radio"/> L | <input type="radio"/> $2(L + R)^2 + 2$ |
| <input type="radio"/> R | <input checked="" type="radio"/> $L(L - 1) + R(R - 1) + 2$ |
| <input type="radio"/> $2(L + R) + 2$ | <input type="radio"/> $L^2 + R^2 + 1$ |

On the left side of the CSP, there are L variables. Each of the L variables could be connected to all $L - 1$ other variables on the left side, for a total of $L(L - 1)$ arcs on the left side.

Similarly, each of the R variables on the right side could be connected to all other $R - 1$ variables on the right side, for a total of $R(R - 1)$ arcs on the right side.

Finally, there are 2 arcs spanning the two sides: $X \rightarrow Y$ and $Y \rightarrow X$.

In total, this is $L(L - 1) + R(R - 1) + 2$ arcs.

Note: These terms already account for directed arcs in both directions, because we consider each pair of variables twice in the counting. For a concrete example, if we had some pair of variables i and j , the arc $i \rightarrow j$ is counted when we count all outward arcs from i . Then, the arc $j \rightarrow i$ is also counted when we count all outward arcs from j . We do want to count both of these arcs because in arc consistency, we consider both directed edges between nodes.

Note: The question does not allow us to make any assumptions about the left/right sides of the CSP, so this bound cannot be made any tighter. We have to assume the worst-case, where both sides of the CSP are fully-connected by binary constraints (i.e. every variable is constrained against every other variable in that side).

Note: No other arcs need to be considered, because there are no relevant arcs between a variable on the left side and a variable on the right side (except for the two arcs between X and Y). In other words, we don't need to consider any arcs between the left and right sides (except for the arcs between X and Y), because we know that those arcs will always be consistent, because there are no constraints along those arcs.

(d) [2 pts] Suppose we solve this CSP using backtracking search with no improvements (no filtering, using arbitrary ordering, and no consideration of the CSP structure).

In the worst case, how many assignments do we need to check in order to solve this CSP?

Your expression could be in terms of L , R , and d .

$$d^{L+R}$$

In the worst case, we have to check every possible assignment. Each of the variables has d possible values, and there are $L + R$ variables.

This is the standard (i.e. as seen in lecture) worst-case runtime analysis for solving a CSP using backtracking search.

In the next three subparts, we will use an idea similar to cutset conditioning to solve this CSP more efficiently.

(e) [1 pt] First, we assign these variable(s) in all possible ways:

- X
- All variables directly connected to X and Y by a constraint
- All variables in the smaller side of the CSP
- All variables in the larger side of the CSP

Recall that in cutset conditioning, when we assign some subset of variables in all possible ways, we effectively “remove” those variables from the constraint graph, leaving a residual constraint graph that is simpler to solve.

Following a similar idea, we think about the structure of the CSP in this problem, and which variables should be removed to create a simpler constraint graph. In particular, removing X will cause the residual graph to consist of two independent subproblems, which would be simpler to solve.

Options 1-2 (all variables in one side) are incorrect, because they do not improve the runtime for solving this CSP.

For example, if we choose to remove the right side of the CSP from the graph, we would need to enumerate all possible assignments of the right side. There are $O(d^R)$ possible assignments to the right side of the CSP. For each assignment, you would need to solve the residual CSP (i.e. the entire left side that’s still left), and each residual CSP takes $O(d^L)$ time to solve. In the worst case, we may need to solve every residual CSP (e.g. if there were no solution).

The total runtime is solving $O(d^R)$ different residual CSPs, with runtime of $O(d^L)$ per residual CSP, for a total of $O(d^R d^L) = O(d^{R+L})$, the same as naive backtracking search.

Option 4 (all variables connected to either X or Y by a constraint) is incorrect, because it will not always help us solve this CSP more efficiently, compared to the case where we remove just X .

As an extreme example, suppose that every variable on the left was connected to X by a constraint, and similarly, every variable on the right was connected to Y by a constraint. Then, in this option, the set of variables we’re removing ends up being every variable in the CSP. We end up assigning every variable in the CSP in all possible ways, which creates $O(d^{L+R})$ possible assignments. Effectively, we are checking every assignment to the CSP (as in naive backtracking search), and the residual CSPs left by removing every variable is a trivial empty CSP.

(f) [2 pts] Next, for each assignment, solve the residual CSP (with the assigned variable(s) removed).

What is the runtime to solve each residual CSP, and why?

- $O((L + R - 1)d^2)$, because the residual CSP is tree-structured.
- $O(d^{L-1}d^R)$, because the residual CSP is tree-structured.
- $O(d^{L-1} + d^R)$, because the residual CSP has independent subproblems.
- $O(d^{L-1}d^R)$, because the residual CSP has independent subproblems.

After removing X , we now have two independent subproblems. The left subproblem has $L - 1$ variables, and the right subproblem has R variables.

It takes $O(d^{L-1})$ time to solve the left subproblem in the worst case. (Standard CSP runtime analysis: in the worst case, we have to check all possible assignments.) Likewise, it takes $O(d^R)$ time to solve the right subproblem.

Because the two subproblems are independent, we can simply solve one subproblem (without thinking about the other), and then solve the other subproblem (without thinking about the first). The runtime for doing this is $O(d^{L-1} + d^R)$.

Note: In general, this will be much faster than $O(d^{L+R})$, so this satisfies the question requirement of solving the CSP more efficiently. For example, if each half had 40 binary-valued variables, the naive approach requires checking 2^{80} assignments. By contrast, the cutset-inspired approach requires checking $2^{39} + 2^{40}$ assignments, which is roughly 700,000,000,000 times faster than the naive approach.

Note: We add the two runtimes instead of multiplying them because the two runtimes are problems we are solving separately. For a completely unrelated example, imagine if we had to selection-sort an array of M items, and then selection-sort another separate array of N items. Selection sort takes quadratic time, so the total runtime is $O(M^2 + N^2)$. It is not $O(M^2 N^2)$.

Options 1-2 are incorrect because the residual CSP is not tree-structured. Given what we know about the CSP structure (i.e. two halves connected by a constraint), there’s no efficient method to remove variables and create a residual CSP with a tree structure.

(g) [1 pt] Finally, how do we output a solution to the original CSP?

- Find one solution to one residual CSP.
- Find all solutions to one residual CSP.
- Find one solution to all residual CSPs.
- Find all solutions to all residual CSPs.

As soon as we find one solution to any of the residual CSPs, we have an assignment to all variables except X . We can plug in the value of X that was assigned to create this residual CSP. This allows us to reconstruct a solution to the original CSP, and we are done.

In the rest of the question, we will consider a different algorithm that uses the structure of the CSP to solve it more efficiently.

First, we'll remove the binary constraint to split the CSP into two independent subproblems.

(h) [2 pts] How should we solve the remaining independent subproblems?

- Find one solution to each CSP.
- Find one solution to the smaller CSP, and all solutions to the larger CSP.
- Find all solutions to the smaller CSP, and one solution to the larger CSP.
- Find all solutions to each CSP.

Unlike the previous algorithm, finding one solution is no longer sufficient, because there is no guarantee that the solution to the left and right subproblems are compatible. In other words, the assignment to X in the left solution and the assignment to Y in the right solution may cause a constraint violation, and we would not discover this while solving the independent subproblems.

Finding one solution for one half, and all the solutions for the other half, is similarly not sufficient. Suppose we found one solution to the left half and all solutions to the right half. It could be possible that the solution to the left CSP assigns a value to X that leaves no valid assignments to Y . In that case, even if we found all the solutions to the right half, we would not be able to reconstruct a solution to the whole CSP. However, another solution to the left half that assigns X differently might be able to be combined with a solution to the right half to form a solution to the whole CSP. Therefore, finding one solution to one half of the CSP is also not sufficient to solve the entire CSP.

The only option here that would allow us to solve the entire CSP is to find all solutions to both halves of the CSP.

(i) [1 pt] Suppose we have a solution to the left subproblem, and a solution to the right subproblem. Which variables do we need to consider to check whether the solutions can be combined into a solution of the original CSP?

- All $L + R$ variables
- Only X and Y
- All variables in the smaller subproblem
- All variables connected to X and Y (including X and Y)

We only need to check the binary constraint between X and Y to know if two subproblem solutions are compatible with each other.

We don't need to re-check the variables in each side because the question statement already says we have a solution to each subproblem, which implies that all the constraints in the left half and right half of the CSP (i.e. all except the constraint between X and Y) have already been satisfied by the subproblem solutions.

Finally, how do we solve the original CSP after solving the independent subproblems? Fill in the details of the algorithm described below.

- Create a new hash map (e.g. Python dictionary) for the left CSP.
- For each solution we found to the left CSP (possibly only one): add a key-value pair to the left subproblem’s hash map.
- Create a second new hash map for the right CSP.
- For each solution we found to the right CSP (possibly only one): add a key-value pair to the right subproblem’s hash map.
- Perform a nested iteration: for each key of the left hash map, iterate through all the keys of the right hash map.

In this part of the algorithm, we have up to $O(d^L)$ solutions to the left subproblem, and up to $O(d^R)$ solutions to the right subproblem. We need to find a pair of solutions, one to the left subproblem and one to the right subproblem, that have can be recombined into a solution to the full CSP (i.e. with X and Y assignments that don’t violate the constraint between them).

A naive solution would check each of the $O(d^L)$ left solutions against all of the $O(d^R)$ solutions, for total runtime of $O(d^L d^R)$, the same as naive backtracking search.

For better efficiency, we notice two things: first, we only need a single solution to the overall CSP. Second, the only constraint left to check at this point is the constraint between X and Y . Therefore, if we have many different left-side solutions that all assign X to the same value, we really only need to consider one of these solutions, and look for a right-side solution with an assignment to Y that doesn’t violate the XY constraint.

Likewise, if we have many different right-side solutions that all assign Y to the same value, we only need one of them to pair with a left-side assignment that assigns X to a compatible value to reconstruct a full solution.

(j) [2 pts] What are the key-value pairs we add to the left hash map?

- | | | |
|----------------------------------|---|---|
| <input type="radio"/> | Key: The solution we found. | Value: The assignment to X in the solution. |
| <input checked="" type="radio"/> | Key: The assignment to X in the solution. | Value: The solution we found. |
| <input type="radio"/> | Key: The solution we found. | Value: 1. |
| <input type="radio"/> | Key: 1. | Value: The solution we found. |

In order to efficiently keep track of one left-side solution per assignment to X , we create a hash map that maps each assignment of X (in the left hash map) to some solution to the left subproblem that uses that assignment.

In other words, for each assignment to X that yields at least one solution to the left subproblem, we map that assignment to some solution to the left subproblem that uses that assignment X .

The right hash map is exactly the same. Keys are assignments to Y . Values are solutions to the problem that use those assignments to Y .

(k) [2 pts] After creating both hash maps, we run a nested iteration through the keys of the left and right hash maps.

For left hash map key x and right hash map key y : which of these conditions tells us that we’ve found a solution to the original CSP?

- x and y are equal.
- x and y are not equal.
- x and y violate a constraint in the original CSP.
- x and y don’t violate a constraint in the original CSP.

If we find a key x (assignment to X) and a key y (assignment to Y) that don’t violate a constraint in the original CSP, and x and y both exist in the hash maps, then that means we have the solution to the left subproblem with $X = x$, and a solution to the right subproblem with $Y = y$.

(l) [2 pts] What is a tight big-O bound on the runtime of the nested iteration through the keys of the left and right hash maps? Your expression could be in terms of L , R , and d (you may not need all variables).

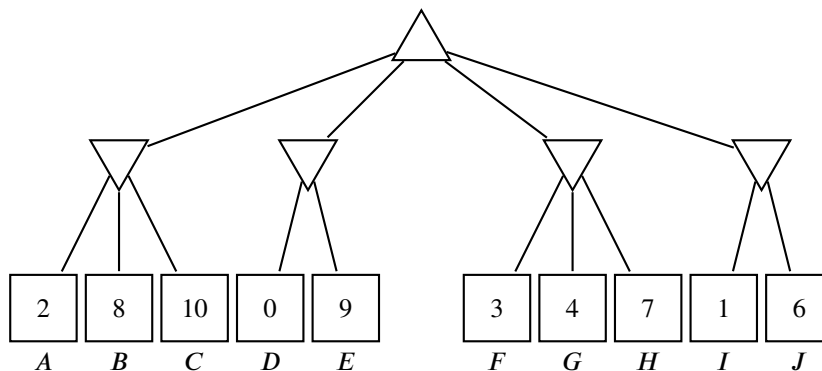
$$O(d^2)$$

There are up to d keys in each hash table, because there are up to d possible assignments for each variable.

Q4. [16 pts] Multi-Agent Search: Faulty Functions

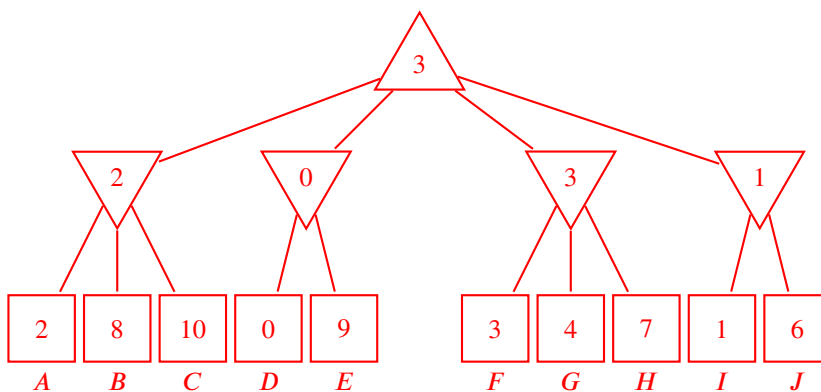
Pacman is in a two-player, zero-sum game, where the players can take turns choosing actions many times before the game ends.

To choose his next action, Pacman runs a depth-limited minimax search, resulting in the game tree below. Upwards-pointing triangles represent maximizer nodes, and downward-pointing triangles represent minimizer nodes.



(a) [1 pt] What is the value at the root node of the tree?

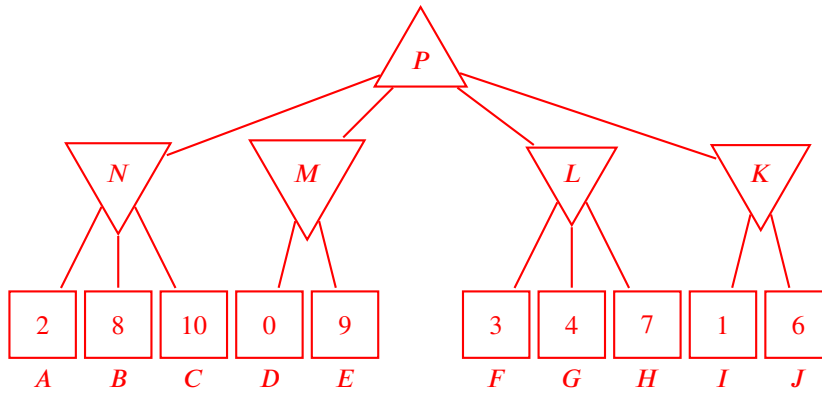
3



(b) [3 pts] Select all branches that would **not** be explored due to alpha-beta pruning, assuming we explore from left to right. (Hint: A **does not** get selected, because when we run alpha-beta pruning, node A does get explored.)

- A
 B
 C
 D
 E
 F
 G
 H
 I
 J

In this solution, we'll label the minimizer and maximizer nodes with letters:



In standard minimax alpha-beta pruning, exploring from left to right, we always have to explore the left-most set of terminal nodes, which are A, B, C here.

At this point, we know that $N = 2$. We also know that $P \geq 2$.

D must be explored. If D and E both had really positive values, this would make M really positive, and would change the action selection at P (would prefer M over N). However, if D and E were both really negative, then M would be really negative, and P would instead prefer N over M . Since the values of D and E can affect the action choice at the root, we need to explore at least one of these nodes. Since we explore left to right, we will start by exploring D .

At this point, in addition to $N = 2$ and $P \geq 2$ from earlier, we also know that $M \leq 0$. This means that P should always prefer $N = 2$ over $M \leq 0$, so we can stop exploring the subtree underneath M now. This means that E does not get explored.

As we explore F, G, H , the best bound we can get on L is $L \leq 3$. In order to stop exploring underneath L , we would need to have $L \leq 2$ in order to guarantee that P is going to choose N and never choose L . However, since we never achieve the $L \leq 2$ bound, we end up needing to explore all of F, G, H .

I must be explored for the same reason that D had to be explored (consider the case where I and J are both very positive, changing the action selection at P to now prefer K).

However, once we explore I , we now have the bound $K \leq 1$. This tells us that P will never prefer $K \leq 1$ and will always choose $N = 2$ instead, so we can stop exploring underneath K . This allows us to not explore J .

In summary: E is not explored, because after seeing $D = 0$ and bounding $M \leq 0$, we know M will never be chosen. Also, J is not explored, because after seeing $I = 1$ and bounding $K \leq 1$, we know K will never be chosen.

Pacman learns that he might have used a faulty evaluation function that sometimes outputs a value 5 greater than the intended value. Formally, if the intended evaluation function is $f(x)$, then the faulty evaluation function $f'(x)$ is defined as:

$$f'(x) = \begin{cases} f(x) & \text{with probability 0.5} \\ f(x) + 5 & \text{with probability 0.5} \end{cases}$$

- (c) [4 pts] Select all true statements. Do not make any assumptions about the original evaluation function $f(x)$ (i.e. it can be any arbitrary evaluation function).

Note: "True minimax value" refers to the minimax value that would be computed with no depth limit.

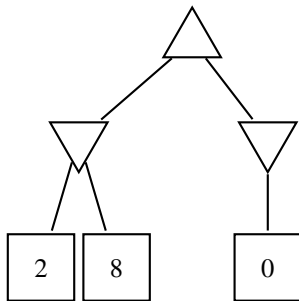
- If the evaluation function is not faulty, Pacman is guaranteed to compute the true minimax value at the root node.
- If the evaluation function is not faulty, Pacman is guaranteed to select the optimal action at the root node.
- If the evaluation function is faulty, the value computed at the root node is guaranteed to be at most 5 away from the true minimax value.
- If the evaluation function is faulty, Pacman is guaranteed to select the optimal action at the root node.
- None of the above

The key insight for this question is: evaluation functions are ultimately estimates of the true minimax of a non-terminal node.

If the original evaluation function $f(x)$ gave wildly inaccurate estimates for the true minimax values, then regardless of whether we use the intended or faulty evaluation function, there are no guarantees of correct minimax value or optimal action selection.

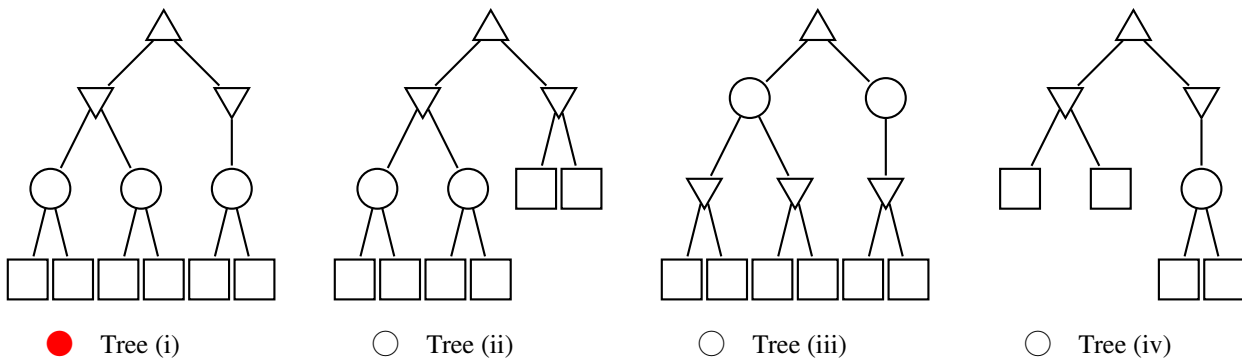
Note: This question implies that nodes A through J are non-terminal nodes where we plugged in an evaluation function in a few places. First, at the top of the question, we said that players can take turns choosing actions *many* times before the game ends. Also, we mentioned that Pacman is running a depth-limited minimax search.

For the rest of the question, assume that Pacman's depth-limited minimax search instead results in the game tree below, and that Pacman is certain that the values in the tree were outputted by the faulty evaluation function.



We would like to draw a new tree that models the situation: Pacman runs depth-limited minimax search, but Pacman does not know if the values computed by the faulty evaluation function are accurate, or 5 higher than intended.

(d) [2 pts] Which of the following modified trees best models the new situation? Circles represent chance nodes.



The extra chance nodes represent the uncertainty over whether Pacman received the intended value, or a value 5 higher than intended.

The ordering of nodes should be: the original maximizer layer, followed by the original minimizer layer, followed by the extra chance nodes. This reflects the order in which events happen in the scenario: Pacman first runs the depth-2 limited search (with the maximizer and minimizer), and then uses the faulty evaluation function (whose uncertainty is represented by the chance nodes).

Option 2 is incorrect because the nodes are layered in the wrong order.

Options 3-4 are incorrect because they only model the uncertainty of the evaluation function in some parts of the tree, but not all parts of the tree where the evaluation function is used. In the original tree, Pacman uses the faulty evaluation function in three places (the three terminal nodes), so we need to replace all three of those terminal nodes with chance nodes representing the uncertainty over the faulty evaluation function.

(e) [2 pts] Which values should exist in the terminal nodes of the modified game tree?

- 0, 2, 5, 8
- 5, 0, 2, 8
- 0, 2, 7, 5, 8, 13
- 5, -3, 0, 2, 3, 8
- 0, 2, 2.5, 3.5, 6.5, 8
- 2.5, -1.5, 0, 1.5, 2, 8

Case 1, the evaluation function is not faulty: the intended value of the evaluation function is the same as the value returned by the faulty function.

Case 2, the evaluation function is faulty: the intended value of the evaluation function is 5 less than the value returned by the faulty function.

For example, if Pacman ran the faulty evaluation function and received the value 8, then the intended value of the evaluation function is either 8, if the evaluation function correctly outputted $f(x)$, or $8 - 5 = 3$, if the evaluation function incorrectly outputted $f(x) + 5$.

The original values in the tree were 0, 2, 8. If we subtract 5 from each of those three values, we get $-5, -3, 3$.

Note: Adding 5 to each of the values provided is incorrect, because the value we are provided is already the output of the faulty function (which may have added 5). In other words, we have already received the value of $f'(x)$, and we never knew the original value of $f(x)$.

While it would be possible to build an alternate search tree where the chance node represents the evaluation function and outputs either $f(x)$ or $f(x) + 5$, this would never be a search tree that Pacman could build. Pacman doesn't know the true value $f(x)$, and if Pacman did know that value, Pacman would simply use that value directly in computation, instead of trying to model uncertainty over the correct $f(x)$ and the incorrect $f(x) + 5$.

(f) [3 pts] Is it possible to model the same situation by only changing the values at terminal nodes, without modifying the structure of the game tree?

Yes

No

If you answered yes, write the values that you would use to replace the terminal node values 0, 2 and 8, in that order. (Your answer should be a list of 3 numbers.)

If you answered no, write a brief justification (one sentence is sufficient).

We can replace each terminal node value with the expected intended value of that terminal node, taking the faulty evaluation function into account.

In other words, instead of replacing each terminal node with a chance node over two further terminal nodes, we directly replace the terminal node's value with the average of the two further terminal nodes' values.

0 is replaced with $(0 + (0 - 5))/2 = -2.5$.

2 is replaced with $(2 + (2 - 5))/2 = -0.5$.

8 is replaced with $(8 + (8 - 5))/2 = 5.5$.

(g) [1 pt] Now, suppose that the faulty function outputs $f(x) + 5$ with probability 0.2, and $f(x)$ with probability 0.8.

Is it still possible to draw a new game tree that models the situation?

Yes

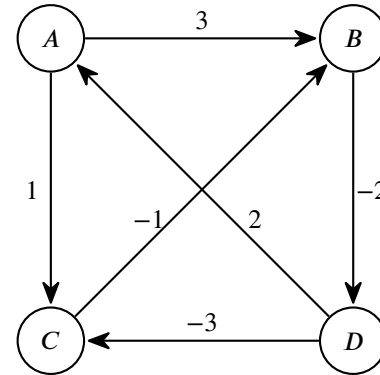
No

Yes. Chance nodes don't necessarily need to have all actions equally likely.

Q5. [15 pts] MDPs: One Piece, Zero Reward

Luffy is trying to find the One Piece by taking actions in the MDP shown to the right.

- From A and D , there are two possible actions (going to two possible successor states). With probability 0.8, the intended successor state is reached. With probability 0.2, the wrong successor state is reached.
- The reward only depends on the state and successor state, not the action. For example, transitioning from A to B receives reward 3, regardless of whether Luffy successfully took action $A \rightarrow B$, or unsuccessfully took action $A \rightarrow C$.
- From B and C , only one action is available, and it always succeeds.



For this entire question, assume that $\gamma = 1$.

(a) [4 pts] Select all true statements.

- $Q(A, A \rightarrow B) > Q(A, A \rightarrow C)$ because the reward for action $A \rightarrow B$ is greater than the reward for action $A \rightarrow C$.
- $V_1(C) = -1$
- $V_1(A) = (0.8 \cdot 3) + (0.2 \cdot 1)$
- $V_1(B) = 0$
- None of the above

Option 1 is false, because the reasoning provided is false. In order to compute Q-values, we need to consider not just the immediate reward, but also future rewards after the immediate action.

For Options 2-4, we are using the intuitive definition of $V_1(s)$ as the best expected reward we can get from starting in state s and acting optimally for 1 time step. (It would be possible, but unnecessarily complicated, to use the Bellman update equation to solve for these values.)

Option 2 is true. In 1 step, the best reward we can get is -1 .

Option 3 is true. In 1 step, the best we can do is take $A \rightarrow B$, which succeeds 0.8 of the time.

Option 4 is false. $V_1(B) = -2$. With one action left from B , there is no way to get a reward of 0 (you can't choose not to take an action in this MDP). You have to take the action $B \rightarrow D$, which gives you reward -2 .

In the rest of the question, consider a modification to this scenario. At every time step, Luffy keeps track of the total cumulative reward accumulated from all actions so far.

If Luffy ever takes an action that causes his total reward to become negative, then Luffy transitions to a terminal state called *End*. There are no further actions or rewards available from this *End* state.

(b) [2 pts] Consider modifying any MDP (not necessarily the one shown above) such that the game ends if the total reward becomes negative.

Luffy suggests that π^* , the optimal policy of the original, unmodified MDP, will also be the optimal policy for this modified problem. Is Luffy correct?

- No, because π^* does not account for total reward and avoiding negative rewards.
- No, because π^* only tells us the values of states, not the optimal actions.
- Yes, because π^* already accounts for total reward and avoiding negative rewards.
- Yes, because even though π^* doesn't account for total reward, π^* maximizes expected rewards, and maximizing reward is always the same as avoiding a negative total reward.

Option 2 is false, because a policy, by definition, tells us what actions to take, not the values of states.

Option 3 is false, because in an unmodified MDP, we don't encode any information about keeping the total reward positive.

Option 4 is false. As a counterexample, consider this MDP:

A	B	C	D	E
+10	-1	0	1	1

Assume, just for this counterexample (not the entire problem), that we earn a square's reward by moving into that square. Also, for this counterexample, assume all actions are deterministic, and $\gamma = 1$.

Assume we want to know the optimal policy if we are in state C and have accumulated total reward of 0 so far.

If we're just trying to maximize expected reward, ignoring total reward, then the optimal policy from C is to go left, collecting reward of $-1 + 10 = 9$ in total.

By contrast, if we also need to avoid negative rewards, then the optimal policy from C is to go right, collecting reward of $1 + 1 = 2$. If we tried to go left from C , we would collect reward of -1 , and the game would immediately end since our total reward is now negative, and we would never be able to collect the $+10$ reward.

In this counterexample, the policy changes depending on whether we want to maximize reward, or maximize reward while avoiding a negative total reward.

(c) [1 pt] In a normal MDP, we have a table of values $V(s)$, one value for every state.

To model this modified situation, we will create a table of values $V(s, x)$. What does x represent?

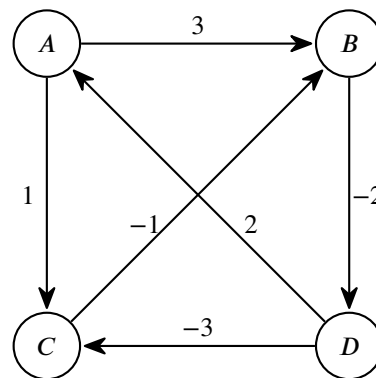
- $x = 0$ if Luffy's total reward is negative, and $x = 1$ otherwise.
- The total reward Luffy has accumulated so far.
- The previous state Luffy was in.
- Luffy's starting state.

Intuitively, we need to keep track of the total reward so far in addition to the state, because our computations change depending on our total reward so far.

For example, again using the counterexample from the previous subpart, being in state C with total reward 100 is different from being in state C with total reward 0. In the former case, the optimal value is 9 by going left. In the latter case, the optimal value is 2 by going right.

The question is reprinted for your convenience below. There is no new information until subpart (d) begins.

- From A and D , there are two possible actions (going to two possible successor states). With probability 0.8, the intended successor state is reached. With probability 0.2, the wrong successor state is reached.
- The reward only depends on the state and successor state, not the action. For example, transitioning from A to B receives reward 3, regardless of whether Luffy successfully took action $A \rightarrow B$, or unsuccessfully took action $A \rightarrow C$.
- From B and C , only one action is available, and it always succeeds.



For this entire question, assume that $\gamma = 1$.

At every time step, Luffy keeps track of the total cumulative reward accumulated from all actions so far.

If Luffy ever takes an action that causes his total reward to become negative, then Luffy transitions to a terminal state called *End*. There are no further actions or rewards available from this *End* state.

(d) [4 pts] Write a modified Bellman equation for this modified MDP.

$$V^*(s, x) = \text{(i) (ii) (iii)} [R(s, s') + \gamma V^*(\text{(iv)}, (\text{v}))]$$

- | | | | | | | | | |
|-------|----------------------------------|--------------|----------------------------------|----------------|----------------------------------|----------------|----------------------------------|--------------------|
| (i) | <input checked="" type="radio"/> | \max_a | <input type="radio"/> | \sum_a | <input type="radio"/> | $\max_{s'}$ | <input type="radio"/> | $\sum_{s'}$ |
| (ii) | <input type="radio"/> | \max_a | <input type="radio"/> | \sum_a | <input type="radio"/> | $\max_{s'}$ | <input checked="" type="radio"/> | $\sum_{s'}$ |
| (iii) | <input type="radio"/> | $T(s, x, a)$ | <input type="radio"/> | $T(s, s')$ | <input type="radio"/> | $T(s, a, s')$ | <input checked="" type="radio"/> | $T(s, x, a, s')$ |
| (iv) | <input type="radio"/> | s | <input checked="" type="radio"/> | s' | <input type="radio"/> | x | <input type="radio"/> | a |
| (v) | <input type="radio"/> | x | <input type="radio"/> | $x - R(s, s')$ | <input checked="" type="radio"/> | $x + R(s, s')$ | <input type="radio"/> | $x \cdot R(s, s')$ |

$$V^*(s, x) = \max_a \sum_{s'} T(s, x, a, s') [R(s, s') + \gamma V^*(s', x + R(s, s'))]$$

In the modified MDP, we still need to maximize over all actions, and take an expected value over the result of taking an action. The first two blanks are unmodified from the standard Bellman equation.

The transition probability still depends on s, a, s' (probability of landing in successor s' depends on the state you are in, s , and the action you select, a). However, the transition probability also depends on x now.

As an example, suppose you are in state C and you take action $C \rightarrow B$. Depending on your total reward so far, you could either end up in B (continuing the game after taking the -1 reward), or *End* (if the -1 reward causes your total reward to go negative and ends the game).

Concretely: $T(C, 100, C \rightarrow B, End) = 0.0$ because if you have total reward 100, then the probability of landing in *End* after taking the $C \rightarrow B$ action is 0.0 (the game will not end if you incur a -1 reward at this point).

However, $T(C, 0, C \rightarrow B, End) = 1.0$ because if you have total reward 0, then the probability of landing in *End* after taking the $C \rightarrow B$ action is 1.0 (you will incur the -1 reward, causing your total reward to become negative, and ending the game).

This example shows that the s, a, s' transition tuple of $(C, C \rightarrow B, End)$ is not enough to uniquely determine the transition probability of this transition. You additionally need to know how much reward has been accumulated so far.

Once we arrive at successor state s' , we need the value of being in state s' and acting optimally. This is the fourth blank.

However, the value function also needs to keep track of the total reward. After taking this s, a, s' transition, our total reward for future time steps has changed from x to $x + R(s, s')$, since we have incurred the immediate reward from the transition we just took. This is the fifth blank.

(e) [3 pts] Select all true statements.

$V_k(End, -1) = 0$ for all k .

$V^*(End, i) = 0$ for all $i < 0$.

$V^*(B, i) = 0$ for all $0 < i < 2$.

None of the above

Option 1 is true. From state *End*, no further actions or rewards are available. Here, k represents the iterations of value iteration (i.e. how many time steps you have left to act optimally), but k is irrelevant because if you are already in the *End* state, and your total reward is -1 (has gone negative), then no matter how many more time steps you have, the total future reward you can accumulate is 0.

Option 2 is true. From state *End*, no further actions or rewards are available. Here, i represents the total reward you have when landing in the *End* state. It doesn't matter whether you're in the *End* state having accumulated -1 or -1000 total reward; either way, the optimal (and only) strategy left from *End* is to do nothing, and collect no future rewards, for an optimal value of 0.

Option 3 is false. The expression $V^*(B, i)$ asks for the value of starting in state *B*, having total reward $i < 2$, and acting optimally indefinitely. At this point, the optimal (and only) policy is to take the $B \rightarrow D$ action, incur a reward of -2 , and then land in the *End* state, from which no further rewards are available. Therefore, $V^*(B, i) = -2$, representing the -2 reward that you will get from starting in this state and acting optimally. (In this MDP, there is no option to take no action, so you are forced to take the only action available, even though it gives negative reward and ends the game.)

(f) [1 pt] For this subpart only, consider a modification to the scenario. The game now ends if Luffy's total reward is less than -1 (instead of ending when the total reward is less than 0).

Is it still possible to model this modified scenario as an MDP and solve it?

Yes

No

The only place where the 0 reward threshold was relevant was when we designed the modified transition function such that transitions with $x + R(s, s') < 0$ (i.e. that cause the total reward to drop below 0) transition to the *End* state instead of the intended successor state.

We could still design a modified transition function so that transitions to *End* only occur when the total reward drops below -1 . The rest of the MDP can still be defined and solved in the same way.

Q6. [16 pts] Reinforcement Learning: Clumsy Test Taker

Pacman is working on an exam with N pages, numbered 1 through N .

From page i , Pacman can try to turn to page $i + 1$ or page $i - 1$, for $2 \leq i \leq N - 1$. On page 1 and page N , Pacman can only try to turn to page 2 and page $N - 1$, respectively.

However, each time Pacman turns a page, Pacman might drop the exam packet and end up on any random page (with equal probability). Pacman does not know the probability of dropping the exam.

(a) [1 pt] What is $|S|$, the size of the state space, in this problem?

Your answer could be in terms of N .

- 0 1 2 N N^2

The most natural way to model this problem as an MDP is to treat each page as a state of the MDP, and the page turns as transitions between states (pages).

There are N pages in the exam, so there are N different states in the state space.

(b) [1 pt] What is $|A|$, an upper-bound on the number of actions per state, in this problem?

- 0 1 2 N N^2

From each state i , there are at most 2 actions available: turning to page $i + 1$ or $i - 1$.

(c) [4 pts] Select the correct space requirements for each reinforcement learning (RL) algorithm. Select one bubble per row.

Model-based learning $|S|$ $|A|$ $|S| \cdot |A|$ $|S|^2 \cdot |A|$

Direct evaluation $|S|$ $|A|$ $|S| \cdot |A|$ $|S|^2 \cdot |A|$

Temporal difference (TD learning) $|S|$ $|A|$ $|S| \cdot |A|$ $|S|^2 \cdot |A|$

Q-learning $|S|$ $|A|$ $|S| \cdot |A|$ $|S|^2 \cdot |A|$

Model-based learning: We need to use our observed episodes to compute estimates of the transition and reward model: $T(s, a, s')$ and $R(s, a, s')$. These tables contain one value per state, per action, per successor state.

Direct evaluation and TD learning both try to learn a value function for a given policy, $V^\pi(s)$. To keep track of this value function, we need to keep track of one value per state.

Q-learning tries to learn a table of Q-values $Q(s, a)$, one value per q-state. We need to keep track of one value per q-state. A q-state is a state-action pair, so we need one value per state, per action.

The rest of the question is independent of the previous subparts.

- (d) [2 pts] When we run TD learning, which samples do we use to update $V^\pi(s)$, for a specific state s ?
- Only samples where the agent is at s , takes an action $\pi(s)$, and lands back in the same state s
 - Only samples where the agent is at s and takes an action $\pi(s)$
 - Only samples where the agent is at some other state s' , takes an action $\pi(s')$, and lands in successor state s
 - All samples, regardless of state and successor state

In TD learning, we keep track of a table of estimated values $V^\pi(s)$, one per state.

The value of a state is the expected discounted reward of starting at that state and acting according to the given policy.

We are able to update our estimate for the value of a state when we observe an episode that starts at that state and acts according to the given policy.

Consider an arbitrary MDP, with unknown transition and reward models. We observe an agent acting according to policy π .

In standard TD learning, we would process each sample one at a time:

$$V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + \alpha [r_i + \gamma V^\pi(s'_i)]$$

- (e) [4 pts] Suppose we want to process k samples in one update instead of k separate updates. Select the most appropriate modified update equation.

Note: The correct update equation will converge to the true value of $V^\pi(s)$, assuming we provide enough samples and set α properly, but you don't need to show this to solve the question.

$$V^\pi(s) \leftarrow (1 - \alpha) \cdot \text{(i)} \cdot \text{(ii)} + \alpha \cdot \text{(iii)} \cdot \text{(iv)} [\text{(v)} + \text{(vi)}]$$

- | | | | | |
|-------|---|--|--|---|
| (i) | <input type="radio"/> 0 | <input checked="" type="radio"/> 1 | <input type="radio"/> k | <input type="radio"/> $1/k$ |
| (ii) | <input type="radio"/> $\sum_s V^\pi(s)$ | <input type="radio"/> $\prod_s V^\pi(s)$ | <input type="radio"/> $\max_s V^\pi(s)$ | <input checked="" type="radio"/> $V^\pi(s)$ |
| (iii) | <input type="radio"/> 0 | <input type="radio"/> 1 | <input type="radio"/> k | <input checked="" type="radio"/> $1/k$ |
| (iv) | <input checked="" type="radio"/> $\sum_{i=1}^k$ | <input type="radio"/> $\prod_{i=1}^k$ | <input type="radio"/> $\max_{1 \leq i \leq k}$ | <input type="radio"/> $\min_{1 \leq i \leq k}$ |
| (v) | <input checked="" type="radio"/> r_i | <input type="radio"/> γr_i | <input type="radio"/> αr_i | <input type="radio"/> r_i^2 |
| (vi) | <input type="radio"/> $V^\pi(s_i)$ | <input type="radio"/> $V^\pi(s'_i)$ | <input type="radio"/> $\gamma V^\pi(s_i)$ | <input checked="" type="radio"/> $\gamma V^\pi(s'_i)$ |

$$V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + \alpha \cdot \frac{1}{k} \sum_{i=1}^k [r_i + \gamma V^\pi(s'_i)]$$

The first term of the sum, where we keep some information about the original $V^\pi(s)$, is unchanged.

In the second term of the sum, we need to introduce not just one sample, but k samples, into our estimate. From the answer choices provided, the only consistent method (i.e. results in convergence toward the true value) to introduce all samples at once is by taking an arithmetic mean over all the samples.

In other words: in regular TD learning, the term $[r_i + \gamma V^\pi(s'_i)]$ represents the estimate of $V^\pi(s)$ based on the sample we just received: the estimated value of being in s is equal to the immediate reward we got from being in s and taking an action, namely r_i , plus the estimated reward from the successor state, namely $V^\pi(s'_i)$.

In the modified equation, the term $\frac{1}{k} \sum_{i=1}^k [r_i + \gamma V^\pi(s'_i)]$ collects all k different estimates of $V^\pi(s)$, one per sample, by taking the average of all the estimates.

Note: For this question, it is not necessary to prove that the equation you provided actually converges to the true value of $V^\pi(s)$. However, it hopefully feels intuitive that averaging k different sample estimates instead of using a single sample estimate still results in convergence to the true value. Any other choice of expression would not result in convergence to the true value.

In the modified TD learning algorithm: we wait to receive k samples that would normally cause an update to $V^\pi(s)$, and then we update $V^\pi(s)$ with all k samples in a single update.

(f) [4 pts] Select all true statements about this modified algorithm. Assume $0 < \alpha < 1$ and $k > 1$.

- Given the same finite set of samples: the values computed with the modified algorithm are always identical to the values computed with the original algorithm.
- More recent sets of k samples influence the value more than older sets of k samples.
- Within a single set of k values, more recent samples will influence the value more than older samples.
- We can learn the optimal policy by storing only $V^\pi(s)$.
- None of the above

Option 1 is incorrect. Consider two samples, and $k = 2$. If we performed two separate updates, then the first sample processed would get multiplied by α (in the first update) and $1 - \alpha$ (in the second update). The second sample processed would only get multiplied by α (in the second update). However, if we performed a single update, then both samples processed would only get multiplied by α .

Option 2 is correct. Each batch of k samples behaves like a single sample in standard TD learning. A more recent batch of k samples will have had fewer $1 - \alpha$ factors multiplied to it, and therefore will contribute more weight to the final estimate. By contrast, an older batch of k samples will have had many $1 - \alpha$ factors multiplied to it (once per update), and therefore will contribute exponentially less weight to the final estimate.

Option 3 is incorrect because samples within each batch are weighted equally, whereas in TD learning, the recent samples are strictly more important (because of the exponential moving average repeatedly de-weighting earlier samples with repeated factors of $1 - \alpha$ each update).

Option 4 is incorrect because we cannot extract the optimal policy from $V^\pi(s)$ without the transition or reward functions.