CS 188 Spring 2023 Introduction to Artificial Intelligence Midterm

- 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.
- For multiple choice questions,

  ☐ means mark **all options** that apply

  ○ means mark a **single choice**

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 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.	Search: Project Groups	16
Q2.	Search: Color a Dinosaur	18
Q3.	Logic: Map Coloring	18
Q4.	CSPs: The Zookeeper	15
Q5.	Games	16
Q6.	MDPs: Flying Pacman	17
	Total	100

Q1. [16 pts] Search: Project Groups

You're taking a class with n students, and you want to find a 4-person project group (you and 3 other students). You decide to formulate your problem as a search problem, as follows:

- **State space**: An unordered set of students in the group so far. For example, {you, Inky, Blinky} and {you, Blinky, Inky} are equivalent states.
- **Successor function**: You pick someone from the class (who is not already in your group), and add them to your group. If your group already has 4 students, there are no further successor states.
- **Start state**: The project group is just you, and nobody else.
- **Goal test**: Your group has exactly 4 students, and everyone in the group gets along. (You can assume that the goal test is somehow able to determine whether a group gets along.)

Consider drawing the search **graph** for this problem.

(a) [2 pts] How many nodes are in the search graph?

Hint: Recall that $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ is the number of ways to choose k elements from a set of n elements. Also, $0! = 1$.

○ $(n-1)$
○ $(n-1)^4$
○ $\binom{n-1}{3}$
[✓ $\binom{n-1}{0} + \binom{n-1}{1} + \binom{n-1}{2} + \binom{n-1}{3}$]
○ Infinitely many

Recall that each node of the search graph represents a state, so this question is asking how many states are in this problem.

The state space is a set of people in the group so far. This set could contain 1 person (you), or 2 people (you and one other), or 3 people (you and two others), or 4 people (you and three others). Therefore, we need to count up how many possible groups (always including yourself) of size up to 4 could be made from the class of n students.

There is $\binom{n-1}{0} = 1$ possible 1-person group: it's the group with just you.

There are $\binom{n-1}{1} = (n-1)$ possible 2-person groups: you, plus any of the $n-1$ other people in class.

There are $\binom{n-1}{2}$ possible 3-person groups: you, plus some (unordered) group of 2 people among the $n-1$ others in class.

There are $\binom{n-1}{3}$ possible 4-person groups: you, plus some (unordered) group of 2 people among the $n-1$ others in class.

Note that there are no states with groups with more than 4 people. The successor function will not generate groups with more than 4 people.

(b) [2 pts] Does the search graph contain any cycles?

○ Yes, the cycles allow us to backtrack from groups that don't pass the goal test.
○ Yes, the successor function allows us to move back-and-forth between two adjacent states.
○ No, because search graphs by definition never contain cycles.
[✓ No, because it is only possible to move to states with more students in the group.]

Each node in the graph represents a state, and each directed edge represents an action that takes you from one state to another state.

In this problem's successor function, the only possible action is to go from a smaller group to a larger group. It is not possible to take an action that goes from a larger group back to a smaller group.

In other words, if you took all the nodes (states) in the graph and ordered them by number of people in the group so far, all the directed edges would point from left to right.

Now, consider drawing the search **tree** for this problem.

(c) [2 pts] What is the maximum branching factor for this search tree?

- [✓ $(n-1)$]
- ○ $(n-1)^4$
- ○ $\binom{n-1}{3}$
- ○ $\binom{n-1}{0} + \binom{n-1}{1} + \binom{n-1}{2} + \binom{n-1}{3}$
- ○ Could be infinite

The branching factor represents the number of actions available from a state.

From the root (just you in the group), there is a choice of $n-1$ people to add to your group. Therefore, there are $n-1$ possible actions from the root, and the branching factor from the root is $n-1$.

Future nodes deeper in the tree will have fewer choices of students to add to your group. For example, at depth 1, where the group has two people (you and one other), there is only a choice of $n-2$ people to add to your group. Therefore, the branching factors of nodes deeper in the tree will be less than $n-1$, and the maximum branching factor is from the node at the root.

(d) [2 pts] What is the maximum depth of this search tree, assuming the root node is at depth 0?

- ○ 1
- ○ 2
- [✓ 3]
- ○ $n-1$
- ○ $(n-1)^4$
- ○ Could be infinite

Each extra layer of the tree corresponds to an additional action, so the maximum depth of the tree corresponds to the maximum number of actions that could be taken in this problem.

It takes 3 actions to create a group of 4 people, and after that, there are no more actions or successor states available.

Now, suppose that you want k students in your project group ($k < n$).

Also, suppose that any group of k students passes the goal test.

(e) [2 pts] As n and k grow very large, which search algorithm will expand fewer nodes to find a solution?

- [✓ Depth-first tree search]
- ○ Breadth-first tree search
- ○ Not enough information

In this problem, all the solutions to the problem are at depth $k-1$ in the search tree, because it takes $k-1$ actions to build a group of k students.

Also, any node at depth $k-1$ is a solution to the problem, because any group of k students is a valid solution.

BFS will look through all the nodes at depth 1 (groups of 2), then all the nodes at depth 2 (groups of 3), then all the nodes at depth 3 (groups of 4), etc. before ever looking at a node at depth $k-1$ (group of k).

On the other hand, DFS will repeatedly expand the deepest node, expanding a single depth-0 node, then a single depth-1 node, then a single depth-2 node, etc. until it pops the first depth $k-1$ node off the fringe and returns it as a solution.

(f) [2 pts] Approximately how many nodes will be expanded by the search algorithm you selected before it finds a solution?

- [✓ $O(k)$]
- ○ $O(n)$
- ○ $O(nk)$
- ○ $O(k^n)$

○ $O(n^k)$

As discussed in the previous subpart, DFS will expand one node at each depth before finding a solution. The depth of the tree is $O(k)$, so that's how many nodes DFS will expand.

Now, suppose that some pairs of students in the class cannot work together because they dislike each other. We'd like to modify our search problem definition to account for this.

(g) [4 pts] Which of the following changes would, if implemented individually, ensure that any complete search algorithm still finds a correct solution, if one exists?

- ■ Modify the goal test to check that no such pairs exist in the state being tested.
- ☐ Modify the successor function so that it only adds students who can work with anybody.
- ■ Modify the successor function so that it only adds students who can work with anybody already in the group.
- ☐ Require that states be ordered lists rather than sets.
- ○ None of the above. A search algorithm cannot solve this problem. You would need a CSP algorithm.

The additional information about which students cannot work together could be added to the goal test (first option) or the successor function (third option).

The second option fails because it will prevent the algorithm from finding some solutions. For example, if half the class refuses to work with the other half and vice versa, a solution still exists in either half if it is big enough. However, the successor function will be unable to generate any new successors, because everybody in the class has at least one person they cannot work with (i.e. there is no person that can work with anybody else in class).

Making states ordered lists does not encode any information about which pairs of students cannot work together, so the last option is false.

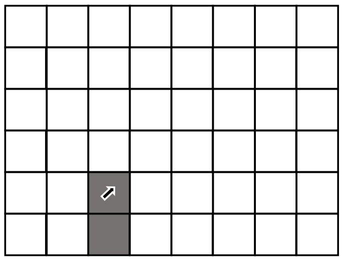
Q2. [18 pts] Search: Color a Dinosaur

Consider a computer screen with a grid of squares.

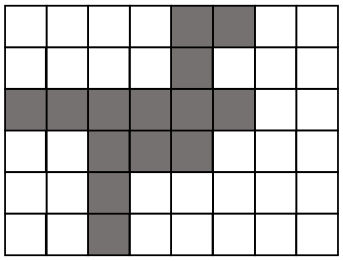
We start with a blank grid and the mouse cursor in the bottom-left square. At each time step, we have several actions to choose from, each costing 1:

- Move the mouse cursor to an adjacent square (north, south, east, west).
- Click the mouse to fill in the currently selected square.

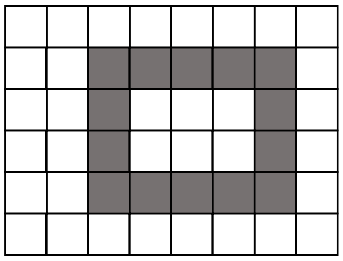
For example, starting from the bottom-left, the action sequence East, East, Click, North, Click produces the grid in Figure 1(a).



(a) Configuration after East, East, Click, North, Click. The arrow denotes the mouse cursor.



(b) A possible goal configuration. The cursor location is ignored in the goal test.



(c) Another possible goal configuration.

Figure 1: Examples for the coloring problem.

We are provided with a list of squares, and we'd like to color all of those squares to form a picture, such as the dinosaur in Figure 1(b) or the rectangle in Figure 1(c), using the fewest number of actions. (Don't laugh! Pixels are expensive!)

A square is *unfinished* if it still needs to be colored.

For each piece of information listed below, select the part of the search problem definition where that information should be stored.

(a) [1 pt] The mouse cursor starts in the bottom-left square.

○ State space ○ Goal test

○ Successor function [✓ Start state]

The information about how the world looks when the search problem starts is encoded in the start state.

(b) [1 pt] The current location of the mouse cursor.

[✓ State space] ○ Goal test

○ Successor function ○ Start state

The current location of the mouse cursor changes depending on the sequence of actions we've taken so far. This belongs in the state space.

(c) [1 pt] The list of squares colored so far.

[✓ State space] ○ Goal test

○ Successor function ○ Start state

The list of squares colored so far changes depending on the sequence of actions we've taken so far. This belongs in the state space.

(d) [1 pt] The original list of squares that need to be colored.

○ State space

○ Successor function

[✓ Goal test]

○ Start state

In our goal test, we need to check if the current state (list of squares colored so far) has matches the desired picture (original list of the squares that need to be colored).

For each heuristic provided, select whether it is admissible, and whether it is consistent.

(e) [3 pts] h_1 = Number of unfinished squares

[✓ Both admissible and consistent] ○ Consistent, but not admissible
○ Admissible, but not consistent ○ Neither admissible nor consistent

We need to click at least this many more times, so the true cost must be at least as great than this (since we need to also add extra actions to move the mouse around).

This heuristic drops by at most 1 after each action; each action costs 1; so the triangle inequality is satisfied and the heuristic is consistent.

In other words, the difference in heuristics between two adjacent states will never exceed 1. Therefore, the difference in heuristics between two adjacent states will never overestimate the true cost between those two states (which is always 1).

(f) [3 pts] h_2 = 2 times the number of unfinished squares

○ Both admissible and consistent ○ Consistent, but not admissible
○ Admissible, but not consistent [✓ Neither admissible nor consistent]

If there is one unfinished square and the cursor is already in the right location, then the true cost is 1 but the heuristic says 2, so it is not admissible. Since consistent implies admissible, it cannot be consistent either.

(g) [3 pts] h_3 = maximum Manhattan distance between any two unfinished squares

○ Both admissible and consistent ○ Consistent, but not admissible
[✓ Admissible, but not consistent] ○ Neither admissible nor consistent

We need to at least cover the maximum distance between any two unfinished squares. Therefore this is an underestimate of the true cost and admissible.

The true action cost is 1. The maximum Manhattan distance between any pair of unfinished squares could drop drastically when a square is finished, which means that this heuristic might overestimate the true cost of an action and does not satisfy the triangle inequality.

(h) [2 pts] Consider h_1 and h_3. Select all true statements:

☐ h_1 dominates h_3.
☐ h_3 dominates h_1.
[✓ Neither h_1 nor h_3 dominates the other.]
[✓ If h_1 and h_3 are admissible, h^* dominates $\max(h_1, h_3)$.]
○ None of the above.

Example where $h_1 > h_3$: Consider a 4x4 block that still needs to be fully colored. Then $h_1 = 16$, but $h_3 = 1$.

Example where $h_3 > h_1$: Suppose we still need to color two squares, spaced 100 squares apart from each other. Then $h_1 = 2$, but $h_2 = 100$.

Because $h_1 > h_3$ is not always true, we cannot say that h_1 dominates h_3. Similarly, because $h_3 > h_1$ is not always true, we cannot say that h_3 dominates h_1.

If h_1 and h_3 are both admissible, then $\max(h_1, h_3)$ is admissible. By definition, h^* is the largest admissible heuristic. Therefore, we have $h^* > \max(h_1, h_3)$, which means that h^* dominates $\max(h_1, h_3)$.

(i) [3 pts] Consider another heuristic:

h_4 = Manhattan distance to nearest unfinished square + 2 times the number of unfinished squares

Select all true statements:

- ☐ Greedy **tree** search with h_4 will always find a solution (possibly not optimal) if one exists.
- 🟥 Greedy **graph** search with h_4 will always find a solution (possibly not optimal) if one exists.
- 🟥 If the goal is the rectangle in Figure 1(c), greedy tree search with h_4 will return the optimal solution.
- ○ None of the above.

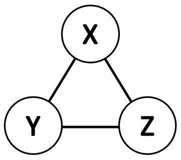
The first option is false. Consider a case with two unfinished squares, both very far away. Greedy search might move to a square, and then avoid clicking it (because that would make h_4 increase by a lot), and move towards the other square. Then, once the cursor reaches the other square, greedy search will again avoid clicking this square (which would increase h_4 by a lot), and start moving back toward the original square. The path explored by greedy search will move back and forth between the two squares forever and never terminate.

If the goal is to draw a rectangle, this greedy search will immediately move the mouse to the nearest unfinished square, and then click on that square. Then, this search will move to an adjacent unfinished square (we know there has to be an adjacent unfinished square because this is a rectangle), and click that square. This search will repeatedly move to an adjacent unfinished square and click that square, moving around the perimeter of the rectangle until the entire rectangle is drawn. This strategy of moving to the closest unfinished square, then clicking around the rectangle, happens to be the optimal solution.

For any general picture, this greedy search may not find the optimal solution. For example, with a solid rectangle with K squares, the optimal solution is the distance to the rectangle plus $2K - 1$ actions to color it efficiently (e.g., in concentric layers). (Side note: We get $2K - 1$ from K clicks, and $K - 1$ moves to reach all the unfinished squares, since if you move efficiently, every move will get you to an unfinished square.) Greedy might color a line across the middle of the rectangle; then, no matter what, it will eventually have to cross over the already-colored middle line to finish the pattern, which is inefficient.

Q3. [18 pts] Logic: Map Coloring

We'd like to express the map coloring problem in propositional logic. Consider the following map:



We'd like to assign colors to nodes X, Y, and Z such that no two adjacent nodes have the same color.

We have three colors available to use: R (Red), G (Green), and B (Blue).

Let the proposition symbol N_C represent node N being assigned color C. For example, $X_R = true$ means that node X is colored Red.

(a) [2 pts] How many symbols are needed to express this problem in propositional logic?

Your answer should be an integer.

9

For each of the 3 nodes, we need 3 symbols (one per color).

The full list of symbols is: $X_R, X_G, X_B, Y_R, Y_G, Y_B, Z_R, Z_G, Z_B$.

(b) [1 pt] For a general map coloring problem with n nodes and c colors, how many symbols are needed to express the problem in propositional logic?

Your answer should be an expression, possibly in terms of n and/or c.

nc

For each of the n nodes, we need c symbols (one per color).

Now we'll formulate several logical sentences that represent this map coloring problem.

(c) [2 pts] Below is a sentence that represents part of this map coloring problem. Fill in the blanks with $\vee$ or $\wedge$.

$$(X_R \_\_\_\_ X_G \_\_\_\_ X_B) \_\_\_\_ (Y_R \_\_\_\_ Y_G \_\_\_\_ Y_B) \_\_\_\_ (Z_R \_\_\_\_ Z_G \_\_\_\_ Z_B)$$

$$(X_R \vee X_G \vee X_B) \wedge (Y_R \vee Y_G \vee Y_B) \wedge (Z_R \vee Z_G \vee Z_B)$$

(d) [2 pts] What does the sentence in the previous subpart represent? You can answer in 10 words or fewer.

Each node must be colored.

In English: $(X_R \vee X_G \vee X_B)$ says that at least one of X_R, X_G, X_B must be true, i.e. X must take on at least one of the colors. The clauses for Y and Z follow the same reasoning.

We AND the three clauses together because all three conditions must be true: X must take on at least one color, and Y must take on at least one color, and Z must take on at least one color.

(e) [2 pts] Write a logical sentence that encodes the constraint "X and Y cannot have the same color". Your sentence does not need to be in CNF.

$\neg(X_R \wedge Y_R) \wedge \neg(X_G \wedge Y_G) \wedge \neg(X_B \wedge Y_B)$

In English: $\neg(X_R \wedge Y_R)$ says that $(X_R \wedge Y_R)$, the case where X and Y are both Red, cannot be true. The clauses for "both can't be Green" and "both can't be Blue" follow the same reasoning.

We AND the three clauses together because all three conditions must be true: X and Y can't both be Red, and they can't both be Green, and they can't both be Blue.

Alternate answer (listing the allowed pairwise assignments to X and Y):

$(X_R \wedge Y_G) \vee (X_G \wedge Y_R) \vee (X_R \wedge Y_B) \vee (X_B \wedge Y_R) \vee (X_G \wedge Y_B) \vee (X_B \wedge Y_G)$

(f) [2 pts] Suppose we've additionally added sentences that encode "X and Z cannot have the same color," and "Y and Z cannot have the same color."

This problem is still missing some sentence(s). Describe the missing sentence(s) (you can answer in 10 words or fewer):

$\boxed{\text{Each node can have at most one color.}}$

Note that the sentence in subpart (c) only guarantees that each node takes on *at least* one color (and not zero colors). It could be the case that a node takes on two colors, but this sentence is still satisfied: for example, X_R and X_G could both be true. The sentence would still evaluate to true, but this would correspond to X being colored both Red and Green, which isn't allowed.

Therefore, we need a sentence that says each node can have *at most* one color (and not more than one color).

(g) [1 pt] How many symbols are assigned to *true* in an assignment that solves the problem?

Your answer should be an integer.

$\boxed{3}$

For each of the 3 nodes, exactly one of the symbols (corresponding to one of the colors) will be assigned to true, which tells us what color is assigned to that node.

Pacman would like to prove the following statement: "If X is colored Red in this problem, then Y will not be colored Red," using only a SAT solver.

(h) [3 pts] In addition to the sentences defining the problem from subparts (c), (e), and (f), which other clauses should be part of the input to the SAT solver to ensure that Pacman gets the right answer? Select all that apply.

- [✓] (X_R)
- ☐ $(\neg X_R)$
- [✓] (Y_R)
- ☐ $(\neg Y_R)$
- ☐ $(\neg X_R \vee \neg Y_R)$
- ☐ $(X_R \vee Y_R)$
- ○ None of the above.

We want to prove that $KB \models \alpha$, where α is $(X_R \implies \neg Y_R)$. To do that, we give the SAT solver $KB \wedge \neg\alpha$ and show it's unsatisfiable. Now

$$KB \wedge \neg\alpha \equiv KB \wedge \neg(X_R \implies \neg Y_R) \equiv (KB \wedge X_R \wedge Y_R)$$

so we add both X_R and Y_R.

Another way to think of this solution is to note that we're proving the statement via contradiction. In other words, we're trying to find a scenario where X is colored Red and Y is also colored Red. This corresponds to the two clauses selected in the solution. If the SAT solver is unable to find such a contradiction, then we know that the statement is true. However, if the SAT solver does find a scenario where X is Red and Y is Red, then we've found a contradiction and the statement must be false.

(i) [3 pts] Select all true statements:

- [✓] Every discrete, finite CSP can be represented as a SAT problem.
- [✓] Every SAT problem can be represented as a CSP.

[✓ A correct SAT representation of a discrete, finite CSP has exactly the same number of satisfying assignments as the CSP has distinct solutions.]

○ None of the above.

The process we used to convert the map coloring problem to a SAT problem can be generalized to all discrete, finite CSPs. We create symbols representing every possible assignment to every variable. Then we add sentences saying that each variable can only take on exactly one value. Finally, we write sentences with the symbols that encode the constraints between variables. We know that it will always be possible to encode constraints with the symbols because in the worst case, we could explicitly write the constraint by listing out all possible assignments that satisfy the constraint, and then express this list of possible assignments as a logical sentence.

To represent a SAT problem as a CSP, we make the symbols into variables. Each variable takes on either true or false as values. Then, we encode the clauses in the SAT problem as constraints in the CSP.

If we use the process described above to represent a CSP as a SAT problem, then each solution to the CSP corresponds to exactly one satisfying assignment of the SAT problem. In other words, for each satisfying assignment we find, we can read off that assignment to get exactly one assignment to all the variables that satisfies all constraints, i.e. one solution to the CSP.

Q4. [15 pts] CSPs: The Zookeeper

You are a newly appointed zookeeper, and your first task is to find rooms for all of the animals.

The zoo has three animals: the Iguana (I), Jaguar (J), and Koala (K).

Each animal needs to be assigned to one of four rooms: the North room (N), East room (E), South room (S), or West room (W), subject to the following constraints:

1. The jaguar cannot share a room with any other animal.
2. The iguana and koala must be in different rooms.
3. The koala can only be in the East room or the South room.

(a) [2 pts] Consider the first constraint: "The jaguar cannot share a room with any other animal."

Can this constraint be expressed using only binary constraints on the three variables I, J, K?

- ○ Yes, it can be expressed as 1 binary constraint.
- [✓ Yes, it can be expressed as 2 different binary constraints.]
- ○ Yes, it can be expressed as 4 different binary constraints.
- ○ No, this is necessarily a unary constraint.
- ○ No, this is necessarily a higher-order constraint.

Recall that a binary constraint relates two variables (in this problem, two animals). We can write this constraint as two different binary constraints: The jaguar cannot share a room with the iguana. The jaguar cannot share a room with the koala. $J \neq I$ and $J \neq K$.

(b) [3 pts] Suppose we enforce unary constraints, and then assign the jaguar to the South room. The remaining values in each domain would be:

Iguana:	North, East, South, West
Jaguar:	South
Koala:	East, South

In the table below, mark each value that would be **removed** by running forward-checking after this assignment.

	North	East	South	West
Iguana	☐	☐	🟥	☐
Koala		☐	🟥	

The main constraint relating the jaguar to the other variables is the first constraint. Since the jaguar has been assigned to the South room, no other animal can be assigned to the South room. Forward checking does not propagate any further.

Formally, forward checking considers every other variable in relation to the variable that was just assigned (here, that's J).

Consider I and J. Only the first constraint is relevant with this pair of variables. Looking through the domain of I, the only value in I's domain that is inconsistent with the newly-assigned value of J is South.

Similarly, consider K and J. Only the first constraint is relevant with this pair of variables. Looking through the domain of K, the only value in K's domain that is inconsistent with the newly-assigned value of J is South.

Forward checking does not check pairs of variables that have not been assigned yet, such as I and K. It only checks the pairs n and J, where n could be all other variables.

(c) [3 pts] Regardless of your answer to the previous subpart, suppose we've done some filtering and the following values remain in each variable's domain:

Iguana:	East, West
Jaguar:	South
Koala:	East

Are all the arcs in this CSP consistent?

- ○ Yes, all arcs are consistent.
- ○ No, only $K \to I$ is not consistent.
- [✓ No, only $I \to K$ is not consistent.]
- ○ No, $K \to I$ and $I \to K$ are both not consistent.
- ○ No, only $J \to I$ is not consistent.
- ○ No, only $I \to J$ is not consistent.
- ○ No, $J \to I$ and $I \to J$ are both not consistent.

The $I \to K$ arc is not consistent. If we assign Iguana to East, there is no valid assignment to Koala that would satisfy the constraints (because of constraint 2).

All of the other arcs in this CSP are consistent. We can go through and check the arcs listed as options in this subpart:

$K \to I$ is consistent because if we assign East at K, we can still assign West at I.

$J \to I$ is consistent because if we assign South at J, we can still assign East or West at I.

$I \to J$ is still consistent because if we assign East at I, South is a valid assignment at J. Also, if we assign West at I, South is a valid assignment at J.

For completeness, here are the remaining arcs:

$J \to K$ is consistent. If we assign South at J, it's fine to assign East at K.

$K \to J$ is consistent. If we assign East at K, it's fine to assign South at J.

The constraints, repeated here for your convenience:

1. The jaguar cannot share a room with any other animal.
2. The iguana and koala must be in different rooms.
3. The koala can only be in the East room or the South room.

(d) [2 pts] Regardless of your answer to the previous subpart, suppose we start over and just enforce the third constraint. Then the remaining values in each domain are:

Iguana:	North, East, South, West
Jaguar:	North, East, South, West
Koala:	East, South

What does the minimum remaining values (MRV) heuristic suggest doing next?

- ○ Assign North or West to a variable next.
- ○ Assign East or South to a variable next.
- [✓ Assign a value to Koala next.]
- ○ Assign a value to Iguana or Jaguar next.

MRV tells us which variable to assign next. In particular, MRV suggests assigning the variable with the fewest remaining values in its domain, which here is Koala (with just 2 values left in its domain, compared to 4 values left in the domains of the other variables).

MRV does not tell us which value to assign next, so we can rule out options 1 and 2 (which suggest values to assign).

(e) [2 pts] Again, consider the CSP after just enforcing the third constraint:

Iguana:	North, East, South, West
Jaguar:	North, East, South, West
Koala:	East, South

Which assignment would the least constraining value (LCV) heuristic prefer?

- [✓ Assign North to Jaguar.]
- ○ Assign East to Jaguar.
- ○ LCV is indifferent between these two assignments.

LCV suggests assigning the value that deletes the fewest values in the domains of other variables.

Assuming we use forward-checking to filter, assigning North to Jaguar causes Iguana to lose North as an option. Assigning East to Jaguar causes both Iguana and Koala to lose East as an option.

Assigning North to Jaguar deletes the fewest values, so LCV gives this assignment higher priority.

Intuitively, North is a less constraining option than East for J, because North leaves K with two options, while East only leaves K with one option. Both North and East for J leave I with three options.

(f) [3 pts] Suppose we add another constraint:

"The West room can contain at most one animal."

Can this constraint be expressed using only binary constraints on the three variables I, J, K?

- ○ Yes, it can be expressed as 1 binary constraint.
- ○ Yes, it can be expressed as 2 different binary constraints.
- [✓ Yes, it can be expressed as 3 different binary constraints.]
- ○ No, this is necessarily a higher-order constraint.

Recall that a binary constraint relates two variables (in this problem, two animals). We can write this constraint as three different binary constraints: $\neg(J = w \wedge I = w)$ and $\neg(J = w \wedge K = w)$ and $\neg(K = w \wedge I = w)$.

In other words, we go through all pairs of variables, and enforce that no pair of variables can both be assigned to West. There are 3 possible pairs of variables that need to be considered in this problem.

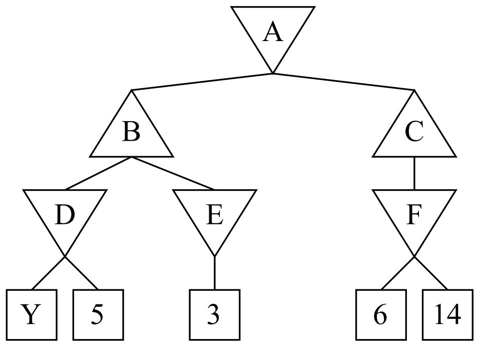
“The iguana and jaguar cannot both be in the West room.”

“The jaguar and koala cannot both be in the West room.”

“The iguana and koala cannot both be in the West room.”

Q5. [16 pts] Games

Consider the following game tree where upward triangle nodes are max nodes, and downward triangle nodes are min nodes:



(a) [1 pt] If $Y = 4$, what is the minimax value at the root node A?

$\boxed{4}$

$D = 4, E = 3, F = 6$

$B = 4, C = 6$

$A = 4$

(b) [3 pts] What range of values for Y would cause the minimax value at the root A to be equal to Y?

Your answer should be an inequality using $\leq$, such as $Y \leq 188$, or $188 \leq Y \leq 288$, or $388 \leq Y \leq 388$, or "impossible" (if there are no such values for Y).

$\boxed{3 \leq Y \leq 5}$

A useful way to work through these questions is to consider different values of Y, and eventually generalize to different ranges of Y. For example, you could start with a very negative value like $Y = -1000$ and see if the minimax value at A is equal to Y. Then, you could start generalizing by noting that every value of $Y < 3$ actually results in the same behavior (there was nothing special about -1000 compared to any other negative number; the important thing was that it was less than all the other terminal node values).

If $Y < 3$, then the value at D will be equal to Y. Therefore, we have $D < 3$. Then, the max node at B will choose the branch to E (to get value 3) and never choose the branch D (which gets value < 3). In summary, if $Y < 3$, the branch to Y will not be chosen, and the value at the root cannot be Y.

The only way for the max node at B to choose the branch to D over the branch to E is if $D \geq 3$. Therefore, we at least have $Y \geq 3$. (In other words, since we ruled out all $Y < 3$ values in the previous paragraph, we've narrowed down our range to $Y \geq 3$.)

The next interesting value to consider is $Y = 5$, since this is the point at which different branches start getting chosen. (In other words, $Y = 3$, $Y = 3.001$, $Y = 4$, etc. will not change the branches chosen.) In particular, when Y exceeds 5, D start to choose the 5 branch instead of the Y branch. In summary, if $Y > 5$, then D will choose the 5 branch instead of Y, which prevents Y from being the value at the root. Therefore, we also have $Y \leq 5$.

At this point, we've narrowed down our range to $3 \leq Y \leq 5$. The last step is to verify that all the values in this range actually cause the root value to equal Y.

From above, we already reasoned that $3 \leq Y \leq 5$ will cause the value at B to be Y. We know that $C = 6$, so if $3 \leq Y \leq 5$, the min node at A will choose B over C. This confirms that the root value at A will be Y (and not 6 from choosing C), and we're done.

(c) [4 pts] Assume that $Y = 4$. If we run minimax with alpha-beta pruning, visiting nodes from left to right, which terminal nodes will we **not** visit because of pruning? Select all that apply.

☐ Y ☐ 6
☐ 5 ☐ 14
☐ 3 [✓ None of the above (we visit all terminal nodes).]

One useful way to work through alpha-beta pruning is to think: if hypothetically, this terminal node had a very positive or very negative value, would that affect the branches chosen in the tree? If the value at the terminal node could change the branches chosen, then we have to check that terminal node and we cannot prune it.

We have to visit all the terminal nodes in the left-most branch (Y and 5) to determine the value of D first. Without checking this first set of terminal nodes, we don't have any bounds on the min or max nodes to prune with.

We have to visit the terminal node 3. This value could have been something very positive like 1000, which would have changed the value at B. (B would have chosen to go to E instead of D.)

We have to visit the terminal nodes 6 and 14, because these could have been very negative values like -1000, which would have changed the value at A. (A would have chosen to go to C instead of B.)

(d) [2 pts] When is the statement $A \leq \max(Y, 14)$ true?

[✓ Always true.] ○ Sometimes true. ○ Never true.

The quickest way to note that this is true is to note that A must eventually take on one of the terminal node values, since we're just repeatedly applying mins and maxes to the terminal node values.

Therefore, C is upper-bounded by the largest of the terminal node values, which happens to be $\max(Y, 14)$.

(e) [2 pts] The root of this tree is a min node. You would like to represent this same game with a new tree that has a max node at the root.

Select all changes that you would need to make in the new tree.

[✓ Convert all min nodes to max nodes.]
[✓ Convert all max nodes to min nodes.]
☐ Add a layer of chance nodes to the new tree.
[✓ Multiply each of the terminal node values by -1.]
○ None of the above - it is impossible to represent this game with a new tree with a max node at the root.

This game has two agents: one agent is represented by the min nodes, and the other agent is represented by the max nodes.

This game is zero-sum. In a zero-sum game, the min and max agents have exactly opposite utilities. We denote this in the game tree by writing out the utilities for one player (the max agent) at the terminal nodes. Then, we assume that the max agent picks higher numbers and the min agent picks lower numbers. This correctly represents the min agent's utility because the min agent's utility is the negative of the numbers at the terminal nodes, and maximizing the negative of the numbers shown is equivalent to minimizing the numbers shown. As a concrete example: the min agent prefers 3 over 5. This represents the fact that "3" on the tree is actually worth -3 to the min agent, and "5" on the tree is actually worth -5 to the min agent. Therefore, the min agent prefers the node labeled 3 because it gives the min agent higher utility.

There was nothing special about which agent we chose to be the max agent and which agent we chose to be the min agent; the important thing was just that their utilities are opposites of each other. We can reinterpret the min agent as a max agent, and reinterpret the max agent as a min agent, without changing the game being represented. This involves converting all min nodes to max nodes, and converting all max nodes to min nodes.

In the original tree, the min agent was trying to minimize the numbers in the tree. Because we've swapped this agent to a max agent looking to maximize the numbers in the tree, we can multiply all the numbers in the tree by -1 to preserve the same ordering of preferences. (As a concrete example: in the original tree, the min agent preferred 3 over 5. In the new tree, this is now a max agent that prefers -3 over -5.)

Multiplying the numbers by -1 also preserves the same ordering of preferences for the max agent who has now been swapped into a min agent. In the original tree, the max agent preferred 5 over 3. In the new tree, this now a min agent that prefers -5 over -3.

(f) [4 pts] Suppose the max nodes B and C are replaced with chance nodes that select actions uniformly at random. What range of values for Y would cause the value at the root A to be equal to Y?

Your answer should be an inequality using $\leq$, such as $Y \leq 188$, or $188 \leq Y \leq 288$, or $388 \leq Y \leq 388$, or "impossible" (if there are no such values for Y).

$\boxed{3 \leq Y \leq 3}$

Solution 1:

For $Y > 5$, $D = 5$, $B = 4$, and $A = 4$, so there is no value of Y such that $C = Y$.

For $Y \leq 5$, $D = Y$ and $B = (Y + 3)/2$. This is always less than 6, so $A = (Y + 3)/2$. For $A = Y$, we must have $Y = (Y + 3)/2$, from which we obtain $Y = 3$.

Solution 2:

First, note that because B and C are chance nodes, it is no longer the case that A is always exactly equal to one of the terminal node values. In particular, B is always going to be the average of 3 and the value at D. Therefore, we either have $B = (3 + 5)/2$, if 5 is selected at D, or we have $B = (3 + Y)/2$, if Y is selected at D.

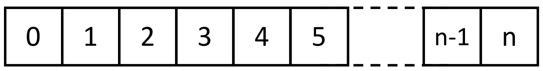
In the case of $B = (3 + 5)/2 = 4$, the value at A will also be 4. We're looking for cases where the value at A is equal to Y, so we check if it's possible that $Y = 4$ here. However, in this case, Y could not have been 4, because we assumed that D chose 5 over Y (and that would not happen if $Y = 4$). You could also just plug $Y = 4$ into the tree and notice that the root value is not 4.

In the case of $B = (3 + Y)/2$, the value at A could also potentially be $(3 + Y)/2$ (depending on how this value compares to 6, the value at C). As before, we check if it's possible that $Y = (3 + Y)/2$. Solving this equation gives us $Y = 3$. Plugging this into the tree, we see that A indeed chooses $B = 3$ over $C = 6$.

Finally, we can confirm that only one value of Y works, not a range. If we changed Y even a little bit to $Y = 3.001$, then the value at B (and later, A) would be the average $(3 + 3.001)/2$, which is not equal to $Y = 3.001$.

Q6. [17 pts] MDPs: Flying Pacman

Pacman is in a 1-dimensional grid with squares labeled 0 through n, inclusive, as shown below:



Pacman's goal is to reach square n as cheaply as possible. From state n, there are no more actions or rewards available.

At any given state, if Pacman is not in n, Pacman has two actions to choose from:

- **Run**: Pacman deterministically advances to the next state (i.e. from state i to state $i + 1$). This action costs Pacman \$1.
- **Fly**: With probability p, Pacman directly reaches state n. With probability $1 - p$, Pacman is stuck in the same state. This action costs Pacman \$2.

(a) [3 pts] Fill in the blank boxes below to define the MDP. i represents an arbitrary state in the range $\{0, \dots, n - 1\}$.

s	a	s'	$T(s, a, s')$	$R(s, a, s')$
i	Run	$i + 1$	1.0	-1
i	Fly	i	$1 - p$	-2
i	Fly	n	p	-2

s' is the resulting successor state after taking an action from a certain state. From state i, taking Fly either keeps you stuck in state i (row 2), or takes you directly to state n (the blank in row 3).

$T(s, a, s')$ represents the probability of starting in s, taking action a, and landing in s'.

Row 1: The action Run is deterministic, so the probability of being in i, taking action Run, and landing in $i + 1$ is 100% or 1.0.

Rows 2-3: Being in i and taking action Fly lands you in state i with probability $1 - p$, and lands you in state n with probability p.

Note that $R(s, a, s')$ is negative because we're trying to minimize costs. Representing costs as negative reward forces an MDP (which maximizes rewards) to minimize those costs.

As a concrete example: this MDP prefers reward -1 over reward -2 because $-1 > -2$. This is consistent with the problem statement, where Pacman would prefer to pay \$1 over paying \$2.

If we left the rewards positive, this MDP would prefer reward $+2$ over reward $+1$. This would be inconsistent with Pacman's preference of spending as little money as possible.

For the next three subparts, assume that $\gamma = 1$.

Let π_R denote the policy of always selecting Run, and π_F denote the policy of always selecting Fly.

Compute the values of these two policies. Your answer should be an expression, possibly in terms of n, p, and/or i.

(b) [2 pts] What is $V^{\pi_R}(i)$?

$\boxed{i - n}$

This expression represents the expected reward of starting in state i and taking action Run indefinitely.

If we start in i and Run indefinitely, we will deterministically move to $i + 1, i + 2, \dots$ until we arrive in state n. This takes $n - i$ Run actions.

Each Run action incurs a reward of -1, and there's no discount ($\gamma = 1$), so the total reward of following this policy is $(-1) \cdot (n - i) = i - n$.

(c) [2 pts] What is $V^{\pi_F}(i)$?

Hint: Recall that the mean of a geometric distribution with success probability p is $\sum_{k=1}^{\infty} k(1-p)^{k-1}p = 1/p$.

$$\boxed{-2/p}$$

We're asked to calculate the expected total reward of starting in state i and taking action Fly indefinitely.

At each time step, with probability p, we land in state n, and with probability $1-p$, we stay in the same place. On average, using the formula given, it will take $1/p$ Fly actions before one succeeds and we reach state n. Each of those actions costs -2, and there's no discount, so the total reward of following this policy is $(-2)\cdot(1/p) = -2/p$.

Formally, to see how the formula results in $1/p$ Fly actions on average, we can write out all the possible outcomes of trying Fly indefinitely:

Case I: Our first Fly action succeeds (takes us to n). This happens with probability p, and results in a total of 1 action taken.

Case II: Our first Fly action fails (we're stuck), but our second Fly action succeeds. This happens with probability $(1-p)p$, where the $1-p$ comes from failing once and the p comes from succeeding. This results in a total of 2 actions taken.

Case III: We fail twice, and succeed on our third try. This happens with probability $(1-p)^2p$, and results in 3 actions taken.

Case IV: We fail three times, and succeed on our fourth try. This happens with probability $(1-p)^3p$, and results in 4 actions taken.

There are infinitely many cases, corresponding to failing more and more times before we succeed.

In total, the expected number of tries before we succeed comes from multiplying each possible number of actions with the probability of requiring that number of actions:

$[1\cdot p] + [2\cdot(1-p)p] + [3\cdot(1-p)^2p] + [4\cdot(1-p)^3p] + \dots$

Writing this as an infinite sum actually yields the exact formula provided to us:

$\sum_{k=1}^{\infty} k(1-p)^{k-1}p = 1/p$

This tells us that on average, it takes $1/p$ Fly actions before one succeeds. This should match your intuition: for example, if Fly succeeds 10% of the time, we expect to try 10 times on average before succeeding once.

Notice that the total reward is independent of i (there's no i in the final expression). This should match your intuition that no matter which state you're in, the value of taking action Fly indefinitely is the same, because the Fly action behaves exactly the same regardless of which state you're in (either you're stuck in the same place, or you directly reach n).

(d) [4 pts] Given the results of the two previous subparts, we can now find the optimal policy for the MDP.

Which of the following are true? Select all that apply. (Hint: consider what value of i makes $V^{\pi_R}(i)$ and $V^{\pi_F}(i)$ equal.)

Note: $\lceil x \rceil$ is the smallest integer greater than or equal to x.

- ☐ If $p < 2/n$, Fly is optimal for all states.
- [✓] If $p < 2/n$, Run is optimal for all states.
- ☐ If $p \geq 2/n$, Fly is optimal for all $i \geq \lceil n - 2/p \rceil$ and Run is optimal for all $i < \lceil n - 2/p \rceil$.
- [✓] If $p \geq 2/n$, Run is optimal for all $i \geq \lceil n - 2/p \rceil$ and Fly is optimal for all $i < \lceil n - 2/p \rceil$.
- ○ None of the above.

Following the hint, we set $V^{\pi_R}(i) = V^{\pi_F}(i)$, and we get $i - n = -2/p$. Solving for i, we get $i = n - 2/p$.

i represents one of the states in the problem, so it should be an integer. We can use the ceiling function to round up i to the nearest integer: $i = \lceil n - 2/p \rceil$.

What does this value of i represent? At state i, $V^{\pi_R}(i) \approx V^{\pi_F}(i)$, so this i tells us approximately where it is equally good to always Run or always Fly.

At this point, we have to figure out whether it's better to Run or Fly for the other values of i. The answer choices suggest that for all $i \geq \lceil n - 2/p \rceil$, it's always better to Fly or Run. Similarly, for all $i < \lceil n - 2/p \rceil$, it's always better to Fly or Run.

Without looking at the answer choices, you can also use intuition to realize the same point. For low values of i, the value of Run is low (because you're far from the end), and for higher values of i, the value of Run gets higher (because you're closer to the end). However, no matter which i you're at, the value of Fly is the same (as we discussed in the previous subpart).

——

Mathematically, $V^{\pi_R}(i) = i - n$ as a function of i is increasing linearly; for higher values of i, $V^{\pi_R}(i)$ increases as well. You could draw this as an upward-sloping line on a graph of state i (x-axis) vs. value (y-axis).

However, $V^{\pi_F}(i) = -2/p$ as a function of i is a constant (it doesn't depend on i). On the same graph of state vs. value, you'd have a flat line.

The upwards-sloping line crosses the flat line at $i = n - 2/p$.

In summary: for higher states $i \geq \lceil n - 2/p \rceil$, we have $V^{\pi_R}(i) > V^{\pi_F}(i)$, which means that it's better to Run in these states.

For lower states $i < \lceil n - 2/p \rceil$, we have $V^{\pi_R}(i) < V^{\pi_F}(i)$, which means that it's better to Fly in these states.

This should match your intuition: if you're close to the end, Run is better. If you're very far away, Fly is better.

——

The final thing we have to check is the $p < 2/n$ condition that the answer choices suggest.

If we plug in $p = 2/n$, we get $i = \lceil n - 2/(2/n) \rceil = \lceil n - n \rceil = 0$.

If we plug in some lower value like $p = 1/n$, we get $i = \lceil n - 2/(1/n) \rceil = \lceil n - 2n \rceil = -2n$.

What does this i value represent? It represents the first state where Run becomes better than Fly. But if $p < 2/n$, we start getting negative values of i! For example, when $p = 1/n$, the first state where Run is better than Fly is $-2n$, which isn't a valid state! In other words, for all states greater than $-2n$, it's better to Run than Fly. This actually means that for all states in the MDP (0 through $n - 1$, which are all non-negative and thus greater than $-2n$), Run is better than Fly.

This should match your intuition: if the probability of succeeding with Fly is way too small, then it's better to always Run, no matter where you're at.

Regardless of your answers to the previous parts, consider the following modified transition and reward functions (which may not correspond to the original problem). As before, once Pacman reaches state n, no further actions or rewards are available.

For each modified MDP and discount factor, select whether value iteration will converge to a finite set of values.

(e) [2 pts] $\gamma = 1$

s	a	s'	$T(s,a,s')$	$R(s,a,s')$
i	Run	$i+1$	1.0	+5
i	Fly	$i+1$	1.0	+5

[✓ Value iteration converges]
○ Value iteration does not converge
○ Not enough information to decide

In these questions, it helps to recall that the value of a state is the expected, discounted sum of rewards for starting in that state and acting optimally. Value iteration is trying to compute a value for every state in the MDP.

Value iteration converges if the value at every state is finite. If the value at some states is infinite, the value iteration will never converge.

——

In this subpart, n is an absorbing state: no matter how you act in this MDP, you eventually have to reach n in a finite number of time steps. Therefore, you only have a limited number of time steps to accumulate reward in this MDP, so the expected discounted sum of rewards in this MDP is finite.

Intuitively, Pacman is only allowed to move forward, so Pacman will inevitably reach n and run out of rewards to collect.

(f) [2 pts] $\gamma = 1$

s	a	s'	$T(s,a,s')$	$R(s,a,s')$
i	Run	$i+1$	1.0	+5
i	Fly	$i-1$	1.0	+5

○ Value iteration converges
[✓ Value iteration does not converge]
○ Not enough information to decide

The expected discounted sum of rewards in this MDP could be infinite. Intuitively, Pacman could use Run and Fly to move between states i and $i+1$ indefinitely, collecting infinite reward.

(g) [2 pts] $\gamma < 1$

s	a	s'	$T(s,a,s')$	$R(s,a,s')$
i	Run	$i+1$	1.0	+5
i	Fly	$i-1$	1.0	+5

[✓ Value iteration converges]
○ Value iteration does not converge
○ Not enough information to decide

The only difference between this subpart and the previous subpart is the discount factor, which is now less than 1.

Pacman can still use Run and Fly to move between states i and $i+1$ indefinitely, but future rewards are discounted. This causes the expected discounted sum of rewards to form an infinite geometric series with common ratio $\gamma < 1$, which converges.

Concretely, the expected discounted sum of rewards for starting somewhere and alternating Run and Fly to move back and forth forever is: $5 + 5\gamma + 5\gamma^2 + \ldots = 5/(1-\gamma)$, which is a finite value.