CS 188 Fall 2024 — Introduction to Artificial Intelligence — Midterm

Solutions last updated: Monday, October 28th

- You have 110 minutes.
- The exam is closed book, no calculator, and closed notes, other than one double-sided cheat sheet that you may reference.
- For multiple choice questions,
  - ☐ means mark **all options** that apply
  - ◯ means mark a **single choice**

First name	
Last name	
SID	
Name 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."

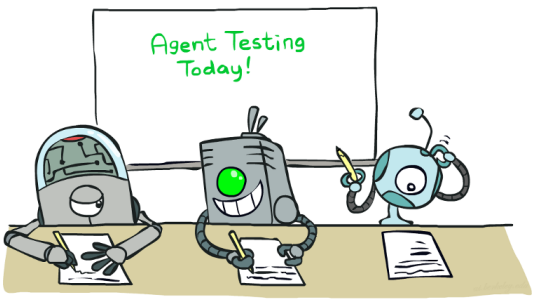
By signing below, I affirm that all work on this exam is my own work, and honestly reflects my own understanding of the course material. I have not referenced any outside materials (other than my cheat sheets), nor collaborated with any other human being on this exam. I understand that if the exam proctor 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.	Word Game	19
Q2.	Easter Island	15
Q3.	Pacman.io	19
Q4.	Pranav's Phunky Puzzle	14
Q5.	Spelunking	16
Q6.	Potpourri: Arbitrary Abstraction	17
	Total	100

It's testing time for our CS188 robots!
Circle your favorite robot below.
(ungraded, just for fun)



The following clarifications were given on the exam; the main exam started at 7:10 and ended at 9:00.

[7:43] Q1(c) option 3: Two pairs are the same if and only if they have the same letters in the same order (so OT does not match with TO)

[8:03] Q4(c)-(e) - The goal is to build a rectangle, which means in the 4x1 grid, D and E cannot go in positions 1 or 4.

[8:05] Q2(d) and Q2(e): O(n) excludes the obstacle that Edgar is standing on.

[8:38] "Flat edge" definition: A has 3 flat edges, along the top, left, and bottom. D has 2 flat edges.

Q1. [19 pts] Word Game

Consider the following word game. You start with 4 letters in a sequence, and you want to transform the sequence into a goal sequence by swapping adjacent letters. On each turn, you can choose one of the following actions, each with cost 1:

- Swap the 1st and 2nd letters.
- Swap the 2nd and 3rd letters.
- Swap the 3rd and 4th letters.

The objective of the game is to get to the goal sequence with the minimum number of swaps. Here is an example:

Goal: *HATS*			
H	*S*	*A*	*T*
Swap *A* and *S*			
H	*A*	*S*	*T*
Swap *T* and *S*			
H	*A*	*T*	*S*

When we call the successor function on a given state, we add the successor states onto the fringe in the following order:

1. The state after swapping the 1st and 2nd letters.
2. The state after swapping the 2nd and 3rd letters.
3. The state after swapping the 3rd and 4th letters.

For example, when we call the successor function on state "ABCD", we add [BACD, ACBD, ABDC] on the fringe, in that order.

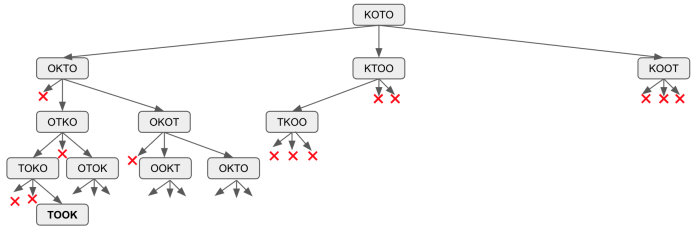
For parts (a) and (b), consider the following start sequence and goal sequence:

Goal: *TOOK*			
K	*O*	*T*	*O*

(a) [4 pts] What is the resulting solution from running BFS graph search?

- ● KOTO → OKTO → OTKO → **TOKO** → TOOK
- ○ KOTO → OKTO → OTKO → **OTOK** → TOOK
- ○ KOTO → **KTOO** → TKOO → TOKO → TOOK
- ○ BFS fails to find a solution

See the tree below. We are doing graph search so repeat nodes are not considered. Arrows indicate that a successor function was called on a node and that its children are added to the stack. A red x indicates that the node is a repeat and ignored after it was popped.



(b) [4 pts] Running DFS results in this solution: KOTO → KOOT → OKOT → OKTO → OTKO → OTOK → TOOK

How many states were *expanded* to find this solution during DFS graph search?

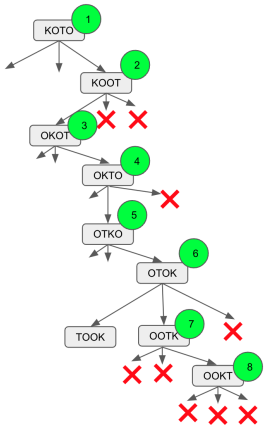
Hint: You expand a state when you call the successor function on that state.

○ 4 ○ 5 ○ 6 ○ 7 ● 8 ○ 9 ○ 10

See the tree below. We are doing graph search so repeat nodes are not considered. Arrows indicate that a successor function was called on a node and that its children are added to the stack. A red x indicates that the node is a repeat and ignored after it was popped.

The green circle indicates the i^{th} expanded node.

The problem specifies that children are added to the stack in this manner: if we call the successor function on "ABCD", we add [BACD, ACBD, ABDC]. This is visualized left to right in the graph below. This means that when using DFS, the right most node (ABDC) is popped first.



(c) [4 pts] Select all admissible heuristics.

☐ The number of letters not in the right place.

☐ The sum of distances for each letter between its current position and the closest identical letter in the goal sequence.

☐ For each adjacent pair (i.e., 1st and 2nd letters, 2nd and 3rd letters, 3rd and 4th letters), count the number of pairs that do not match the corresponding pair in the goal sequence.

■ The minimum distance from any letter to the closest identical letter in the goal sequence.

○ None of the above.

Number of misplaced letters: Not admissible because two misplaced letters can be corrected with a single swap. Ex Goal is ABCD, and current configuration is BACD. Heuristic will provide a value of 2.

Sum of letter distances: Not admissible because if over estimates the number of swaps required. Ex Goal is ABCD, and current configuration is BACD. Heuristic will provide a value of 2.

Number of non-matching adjacent pairs: Not admissible consider the following counter example. Ex. Goal is ABCD, current configuration is BACD. Pairs that don't match the goal is BA and AC (heuristic of 2), but a single swap resolves this

Min letter distance: Admissible. Fixing the position of the letter closest to its desired location always underestimates the number of swaps requried for fix the entire word.

For the rest of this question, we change the cost of each action as follows:

- Swap 1st and 2nd letters: Cost 1.
- Swap 2nd and 3rd letters: Cost 2.
- Swap 3rd and 4th letters: Cost 3.

If two states on the fringe have the same priority, break ties alphabetically. For example, if ABCD and BACD have the same priority, pop ABCD off the fringe first.

(d) [3 pts] Which statements are true about running UCS graph search on this game, assuming a solution exists?

- [x] UCS will always find a solution.
- [x] UCS will always return the lowest-cost solution.
- [ ] UCS will always expand fewer states than BFS.
- ○ None of the above

Uniform cost search will always find the cost optimal solution (and therefore always find a solution). However, since uniform cost search will explore the minimum cost solutions first (i.e. have a preference of swapping the 1st and 2nd letters). UCF has no guarantees when compared with BFS.

Consider the following heuristic: The maximum distance from any letter to its position in the goal sequence.

For example, for the goal "ABCD," the state "CDBA" has a heuristic of 3, because A has distance 3, B has distance 1, C has distance 2, and D has distance 2.

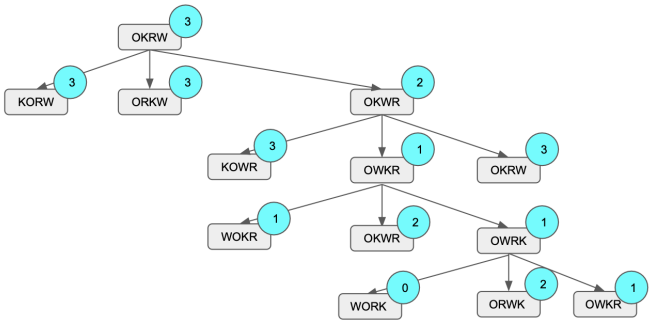
For part (e), consider the following start sequence and goal sequence:

Goal: *WORK*			
O	*K*	*R*	*W*

(e) [4 pts] Which solution is returned by greedy graph search?

- ○ OKRW → **ORKW** → ORWK → OWRK → WORK
- ● OKRW → OKWR → OWKR → **OWRK** → WORK
- ○ OKRW → OKWR → OWKR → **WOKR** → WORK

See the tree below. The number in the blue circle indicates the heuristic value. Greedy search always chooses the lowest heuristic value to expand next, tie-breaking using alphabetical order (as specified in the problem statement).



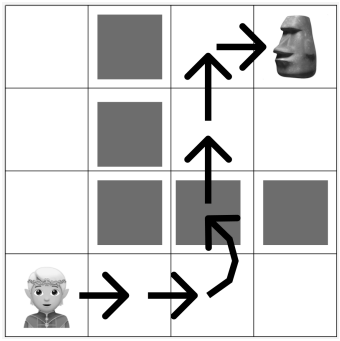
Q2. [15 pts] Easter Island

Edgar is an elf on Easter Island, seeking Pietru the Paradise Dweller—an ancient Moai statue—for midterm wisdom.

Easter Island is represented as an $N \times N$ grid, with k randomly placed obstacles. At each step, Edgar can either move up, down, left, or right (if unobstructed). He also has 3 jumps to use during the journey.

A jump is an action that puts him on top of an obstacle adjacent to him in the direction he chooses. For example, in the figure to the right, Edgar jumps north from $(3, 0)$ to the top of the obstacle at $(3, 1)$. **He may continue to walk on top of adjacent obstacles until he descends**. Descending does not require him to use a jump.

Edgar knows the location of all obstacles, as well as his starting position and the position of the Moai statue on the grid. Edgar also knows what the legal moves are, at every position. If there are no legal moves—for example, if he's surrounded by obstacles with no jumps remaining—his mission ends. All actions have a cost of 1, and he cannot remain still.



In this example, straight arrows are regular moves and the curved arrow is a jump.

(a) [3 pts] Edgar is comparing DFS, BFS, and A* Search using the trivial heuristic $h(n) = 0$. Assume the branching factor is 4, and the nearest solution is 4 moves away. Select all true statements.

Hint: You expand a state when you call its successor function.

- ☐ A* Search will find the solution by expanding fewer states than BFS.
- ☐ A* Search will always expand $4^0 + 4^1 + 4^2 = 21$ states or fewer.
- ■ BFS will always find a solution at an equal or shallower depth than DFS.
- ○ None of the above.

With $h(n) = 0$, A* Search behaves like Breadth-First Search in a graph with uniform costs, expanding states level by level. Therefore, A* Search must evaluate all states up to depth 3, totaling $1 + 4 + 16 + 64 = 85$ states before considering the earliest solution state at depth 4. BFS will always find the shallowest solution first.

(b) [2 pts] Edgar uses A* Search with an admissible heuristic. What is the worst-case time complexity in terms of the branching factor b and the depth of the shallowest solution d?

- ○ $O(n)$, where n is the number of states
- ● $O(b^d)$
- ○ $O(d^b)$
- ○ $O(b)$

In the worst case, A* Search has exponential time complexity $O(b^d)$, where b is the branching factor and d is the depth of the solution.

(c) [3 pts] Edgar now uses greedy graph search with heuristic $h(n) = E(n)/100$, where $E(n)$ is his Euclidean distance to the statue at state n. Select all true statements.

- ☐ Greedy graph search is complete, because it only evaluates legal moves.
- ☐ Greedy graph search will always find a solution with $h(n)$, because $h(n)$ is an admissible heuristic.
- ☐ Greedy graph search will always find a shorter path than A* search with $h(n)$ as its heuristic.
- ● None of the above

Greedy search is incomplete. Greedy search does find a solution with $V(n)$ here, but it is not because $V(n)$ is an admissible heuristic. A* star will always find the shortest path.

Edgar wants to design an admissible heuristic function $h(n)$. He has the following functions:

1. $M(n)$ = the Manhattan distance from Edgar to the goal for state n.
2. $E(n)$ = the Euclidean distance from Edgar to the goal for state n.
3. $O(n)$ = the smallest number of obstacles that lie along any path of distance $M(n)$ between Edgar and the goal for state n.
4. $J(n)$ = the number of jumps remaining.

(d) [2 pts] Which of the following are **admissible** heuristics?

- [x] $M(n)$
- [x] $E(n)$
- [x] $O(n)$
- [ ] $J(n)$
- ○ None of the above

An admissible heuristic never overestimates the true cost to reach the goal. Manhattan distance and Euclidean distance are admissible in a grid with allowable movements up, down, left, or right. Option C is $< M(n)$, and therefore admissible. The number of jumps is not admissible (if Edgar is immediately next to the goal and still has 2 jumps remaining, this heuristic is an overestimate).

(e) [4 pts] Which of the following are **admissible** heuristics?

- [x] $\min(O(n), J(n))$
- [x] $E(n)/3$
- [ ] $O(n) + 1$
- [ ] $M(n) + O(n)$
- ○ None of the above

An admissible heuristic must never overestimate the true cost to reach the goal. Option 1 is the minimum between 2 already admissible heuristics, which makes it admissible. Option 2 simplifies to $1/3$ of the Euclidean distance (which itself is already admissible). Option 3 is inadmissible (imagine he is immediately next to the goal standing on an obstacle, this heuristic returns 2 when the real cost is 1). Option 4 is inadmissible (you can imagine using the same example that disqualifies option 3).

(f) [1 pt] Edgar decides to use the simulated annealing algorithm with an initial temperature of 100 and a cooling schedule of $T_i = \frac{T_0}{1+i}$, where i is the number of iterations. Approximately what is the algorithm's temperature after 50 iterations?

- ○ 0
- ● 2
- ○ 10
- ○ 50

Using the cooling schedule: $T_{50} = \frac{100}{1+50} = \frac{100}{51} \approx 1.96$, which is approximately 2.

Q3. [19 pts] Pacman.io

Xavier and Matei are playing a new online game called Pacman.io.

They each move their own Pacman around a 20×20 grid, taking turns, with actions {UP, DOWN, LEFT, RIGHT, STOP}.

To start, 100 food dots are placed around the grid in **random locations**. After a food is eaten by either Pacman, another food is **immediately** placed in a random empty grid location.

Both Pacmen start at power level 1 and gain 1 power level for each food eaten.

A player wins if their Pacman reaches power level 100 or "eats" the other player's Pacman.

"Eating" occurs as follows: When both Pacmen occupy the same grid location, the Pacman with higher power level eats the other one. If they have the same power level, nothing occurs, and both Pacmen occupy the same position.

Xavier represents this game with a game tree.

(a) [4 pts] Which of the following are valid state representations for this problem?

- ☐ 100 booleans for food, and each Pacman's (x, y) coordinates and power level.
- ☐ A list of visited food locations, and each Pacman's (x, y) coordinates and power level.
- ■ A list of current food locations, and each Pacman's (x, y) coordinates and power level.
- ☐ Each Pacman's (x, y) coordinates and power level only.
- ○ None of the above.

A: This is invalid as there are only 100 booleans, and regardless of state, there is always 100 food on the grid as the grid will be replaced after eating.

B: This does not keep track of the current locations of food on the grid.

C: This keeps track of all the required variables for the problem.

D: This does not keep track of food in any way.

(b) [4 pts] Xavier chooses to represent the state with 400 booleans for the food locations (one for each grid location), each Pacman's (x, y) coordinates and their power levels. What is the size of the state space using this state representation?

Represent the answer in the form:

$$A^B \times B^C$$

Fill in the boxes for A, B, and C using only positive integers.

A: 40000 B: 2 C: 400

A= # Pacman Coordinates × # Power Levels, B=# of Pacmen, C=# of Food Bools

Other equivalent solutions are also accepted for full credit, examples:

- $4^{200} \times 200^4$ or $200^4 \times 4^{200}$
- $400^4 \times 4^{198}$
- $20^8 \times 8^{132}$
- $10000^2 \times 2^{404}$
- $100^4 \times 4^{202}$

(c) [4 pts] Xavier considers running depth-limited minimax. In general, which of the following are valid reasons for running depth-limited minimax **instead of** a full minimax search?

- [x] Searching the full tree is often computationally infeasible due to a large branching factor.
- [ ] Using an evaluation function on the leaf nodes in depth-limited minimax is more accurate than computing the utilities for the entire minimax tree.
- [x] A suboptimal agent is better modeled by a depth-limited minimax tree.
- [ ] Depth-limited minimax does alpha-beta pruning for us.
- ◯ None of the above.

A: True, and is one of the more common reasons to limit the depth of game trees.

B: False, an evaluation function *approximates* the utility of states.

C: True, because we use an evaluation function, we are making an approximation. This can reflect suboptimality in the minimax optimality of the agents and/or opponents.

D: False, depth-limited minimax does "prune" the game tree by removing states beyond a certain depth, but this is not "alpha-beta pruning".

(d) [5 pts] Xavier needs a function to compute the utility at each leaf node in the minimax tree.

- Case 1: Xavier's Pacman has a higher power level than Matei's. For two states with the same power levels, the function should prefer states where the Pacmen are closer.
- Case 2: Matei's Pacman has a higher power level than Xavier's. For two states with the same power levels, the function should prefer states where the Pacmen are farther apart.

Write an evaluation function that behaves as above, using these terms:

- Manhattan Distance between Xavier and Matei's Pacmen, $MH(p_x, p_m)$.
- Power level of Xavier's Pacman, ℓ_x.
- Power level of Matei's Pacman, ℓ_m.

Your function cannot be piecewise—write one expression for all states.

You may assume that $\ell_x \neq \ell_m$ and $MH(p_x, p_m) \neq 0$.

Note: An example of "two states with the same power levels" is a pair of states where $\ell_x = 10$ and $\ell_m = 5$ in both states.

Generally, there are two viable functions containing $MH(p_x, p_m)$: $MH(p_x, p_m)$ and $\frac{1}{MH(p_x, p_m)}$

(Case 1) If we use $MH(p_x, p_m)$, we need to multiply this by some $f(\ell_x, \ell_m)$ which satisfies the following requirements:

$$\text{If } \ell_m - \ell_x > 0 : f > 0$$
$$\text{If } \ell_m - \ell_x < 0 : f < 0$$

(Case 2) If we use $\frac{1}{MH(p_x, p_m)}$, we need to multiply this by some $f(\ell_x, \ell_m)$ which satisfies the following requirements (note that this is the opposite of the above case):

$$\text{If } \ell_m - \ell_x > 0 : f < 0$$
$$\text{If } \ell_m - \ell_x < 0 : f > 0$$

Examples of such functions include:

(Case 1): $\text{sign}(\ell_m - \ell_x) MH(p_x, p_m)$ or $(\ell_m - \ell_x) MH(p_x, p_m)$ or $\left(\frac{\ell_m}{\ell_x} - 1\right) MH(p_x, p_m)$

(Case 2): $\text{sign}(\ell_x - \ell_m) \frac{1}{MH(p_x, p_m)}$ or $(\ell_x - \ell_m) \frac{1}{MH(p_x, p_m)}$ or $\left(\frac{\ell_x}{\ell_m} - 1\right) \frac{1}{MH(p_x, p_m)}$

(e) [2 pts] Suppose there are now a known, arbitrary number of Pacmen on a grid of known, arbitrary size. After a Pacman is eaten, the eater will gain power level equal to the power level of the eaten Pacman. The win conditions are to reach power level P_{max} first or eat all other Pacmen.

Which of the following setups represents this new scenario the best for Xavier's Pacman?

◯ Run minimax with all other Pacmen as minimizer nodes. The evaluation function is the power level of Xavier's Pacman.

◯ Run expectimax with all other Pacmen as a expectation nodes. The evaluation function is power level of Xavier's Pacman.

● Construct a game tree with each Pacman maximizing their own utility. The evaluation function for each Pacman is their own power level.

A: False, as the other Pacmen are not necessarily trying to minimize the power level of Xavier's Pacman.

B: False, as we should not make any assumption about the minimax optimality of the other agents.

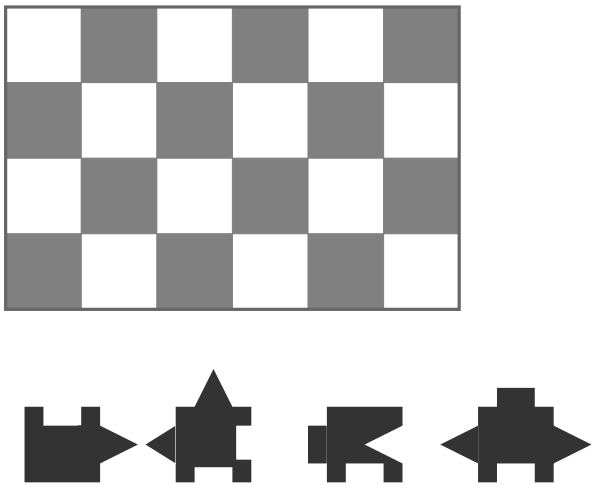
C: True, a multi-agent tree with each agent maximizing their own utility represents both the independence and the adversariality of the Pacmen with respect to the win conditions.

Q4. [14 pts] Pranav's Phunky Puzzle

Pacman is a big fan of jigsaw puzzles and designs a hard puzzle:

- There is an $M \times N$ grid, where each element in the grid fits exactly one puzzle piece.
- There are 5 different pieces. Each piece may be used more than once. Each piece may be rotated by 0°, 90°, 180°, or 270°.
- If a piece has a flat edge, that edge must be touching the edge of the grid.
- The goal is completely fill the grid by placing pieces, forming a $M \times N$ rectangle.

An example grid and pieces are shown below:



(a) [2 pts] Suppose we define the CSP variables to be each grid position. What is the size of the domain for each variable before filtering?

- ○ 5
- ○ 10
- ○ 15
- ● 20

Each grid has 5 possible pieces to choose from, and you can rotate each piece in 1 of 4 directions. So the domain size is $5 * 4 = 20$

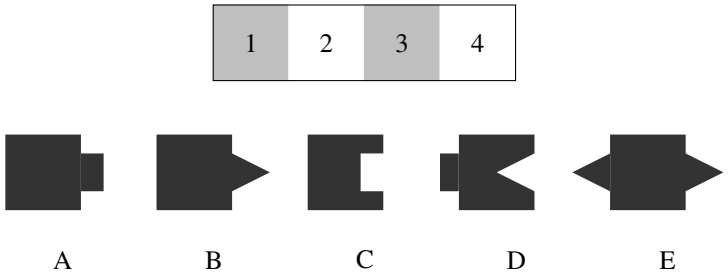
(b) [2 pts] If we have d pieces, rather than 5, what is the time complexity of running backtracking search on this puzzle without using any optimizations?

- ○ $O(dMN)$
- ● $O\left((4d)^{MN}\right)$
- ○ $O\left(d^{MN}\right)$
- ○ $O\left((MN)^{4d}\right)$

In general, backtracking runs in at most d^n time where d is the maximum size of a variable's domain, and n is the number of variables you have. Here, we have $4 * d$ values for each variable, and there are $m * n$ total squares on the grid so the final runtime is $O((4d)^{MN}$

For the rest of the question, we modify the puzzle so that pieces can only be rotated 0° or 180°.

Consider the following puzzle, with a 4×1 grid, and the following 5 pieces:



Reminders: Pieces may be reused. If a piece has a flat edge, that edge must be touching the edge of the grid. For example, piece A cannot go in position 2 or 3.

Pacman solves the puzzle using backtracking search. First, he assigns puzzle piece C to position 1.

Note: If there is **any orientation** of a piece that would fit in a position, include that piece in the domain of that position.

(c) [4 pts] After applying unary constraints, apply arc consistency to $(2 \rightarrow 1)$ and then $(4 \rightarrow 2)$.
What pieces are in the domains of positions 2 and 4?

Domain of position 2:

- [ ] A
- [ ] B
- [ ] C
- [x] D
- [ ] E
- ○ None of the above

Domain of position 4:

- [x] A
- [x] B
- [x] C
- [ ] D
- [ ] E
- ○ None of the above

Position 2: First note that the only pieces that can fit in square 2 are pieces D and E. After placing piece D in position 1, we see that only piece D can fit into piece C.

Position 4: Before applying any binary constraints, the only pieces that fit in square 4 are A, B, and C. After placing piece C in position 2, we see that this does not touch square 4, meaning that it's impossible to eliminate any variables from position 4's domain due to not having any binary constraints between position 1 and position 4

(d) [4 pts] Continuing from the previous subpart, Pacman applies arc consistency on $(2 \rightarrow 3)$ and then $(3 \rightarrow 2)$.
What pieces are in the domains of positions 2 and 3?

Domain of position 2:

- [ ] A
- [ ] B
- [ ] C
- [x] D
- [ ] E
- ○ None of the above

Domain of position 3:

- [ ] A
- [ ] B
- [ ] C
- [ ] D
- [x] E
- ○ None of the above

Position 2: We see that the only element in square 2's domain is D. And the only elements in 3's domain are D and E. Of these two, E fits together with D so we can't eliminate D from square 2's domain.

Position 3: Again, 3 can either contain piece E or piece D. Of these, we can eliminate D because piece D cannot connect to itself. However, piece E can so we keep E in 3's domain.

(e) [2 pts] Consider the domains you computed in the previous subparts.

According to the MRV heuristic, which variable should we assign next? Break ties by picking the lower number.

- (●) 2
- ( ) 3
- ( ) 4

Of the unassigned variables, 2 and 3 both only have a domain of 1. Since the MRV heuristic wants to choose the variable with the smallest domain, we have a tie between these two squares. Of these two, since $2 < 3$ we can use our tie breaking rule to select 2.

Q5. [16 pts] Spelunking

Pacman is mining gemstones in deep, dark caves with an autonomous robot!

Each cave is a separate MDP. The caves have the same state spaces, but not necessarily the same transition and reward functions.

Every time the robot takes an action a, it wirelessly sends a sample to Pacman: (s, a, s', r), where s is the state it was in, a is the action it took, s' is the next state, and r is the reward it received.

The reward is +10 if the robot discovers a gemstone, and 0 otherwise.

The state transition function is deterministic, and the state and action space is finite. You know nothing else about the state and action spaces, the state transition function, or the reward function.

Pacman wants to use Q-learning on the samples to learn a policy π.

Each subpart is separate and independent.

(a) [4 pts] Pacman sends his robot into Cave A, using a random policy. Select all settings that can individually cause π, the learned policy, to be **suboptimal**.

- [x] The robot runs until it visits all states once.
- [x] The robot runs until it discovers all the gemstones in the cave.
- [ ] Pacman sets α to 1, and reduces it slowly to a very small number.
- [ ] Pacman adds an exploration bonus to the reward, which approaches zero when a state is visited a large number of times.
- ○ None of the above.

In regards to the first two options: In Q-learning theory, we will converge to an optimal solution no matter what policy generates our samples, as long as the following condition is fulfilled: in the limit of infinite samples, we sample all states infinite times. That just means we need complete exploration, and we need to see each sample multiple times. As such, option 1 and 2 are correct: they would cause our learned policy to be suboptimal.

Option 3: As long as your learning rate is reduced to a small amount, Q-learning will remain effective. This option is not correct; doing option 3 would not affect our policy's optimality.

Option 4: In the limit of infinite samples, if your exploration bonus approaches zero, then the rewards we see will also approach the true unmodified reward. Option 4 does not affect our policy's optimality.

(b) [2 pts] Pacman collects samples from Cave B and runs Q-learning to learn π, the optimal policy in Cave B.

Then, Pacman follows π in Cave C. What do we expect to happen?

- ○ π is the optimal policy.
- ○ In expectation, π performs better than a random policy, but it is not optimal.
- ● In expectation, π performs about as well as a random policy.

Recall that caves have the same state spaces, but not the same transition nor reward functions. Q-learning finds a policy that is dependent on the transition and reward function (by definition), so we have no guarantees if we go to a different cave. The correct answer is option 3.

(c) [4 pts] Pacman rates caves by their *danger*, D, from 1 (safest) to 100 (most dangerous), and sets the discount factor as $\gamma = \frac{D}{100}$ before running Q-learning in each cave. What effect does this have on a robot following the learned policy, π?
Select all that apply.

- [x] If $D = 100$, the robot will try to collect all gemstones.
- [ ] If $D = 1$, the robot will move randomly.
- [x] In more dangerous caves, future rewards will be more important for the robot.

☐ The horizon scales inversely with D; that is, the more dangerous the cave, the shorter the horizon.

◯ None of the above.

Option 1: Yes. If $D = 100$, then the discount factor is 1. If this is the case, then our horizon (the number of timesteps under which our policy will look at) will be infinite. When this is the case, you would want to gain all gemstones.

Option 2: No. Discount factor of 0.01 just means that future rewards are heavily discounted, and a majority of the policy is aimed towards the immediate reward. This doesn't mean a random policy, however; we're still aimed at maximizing reward.

Option 3: Yes. This is the correct interpretation of higher discount factor. As your discount factor approaches 1, then your horizon increases; the number of steps that you look in the future effectively increases, and you will consider more future rewards.

Option 4: No. The horizon scales propoprtionally with D. The higher the discount factor, the longer the horizon.

For the next three subparts, Josh proposes the new algorithm described below:

1. Josh collects an extremely large amount of samples using a random policy, π_{rand}.
2. Josh computes Q_{rand} by running Q-learning with $\alpha = 0.5$, $\gamma = 0.99$, and the samples from step 1.
3. Josh extracts a policy from Q_{rand} to get π_{smart}.
4. Josh collects an extremely large amount of samples using π_{smart}.
5. Josh initializes Q_{smart} to Q_{rand}.
   He then runs Q-learning to update Q_{smart} using $\alpha = 0.1$, $\gamma = 0.99$, and the samples from step 4.
6. Josh extracts a policy from Q_{smart} to get π_{final}.

(d) [2 pts] Q_{rand} is not optimal. Why?

- ○ Q-learning with randomly generated samples does not give an optimal Q-function.
- ○ The discount, γ, is too high.
- ● The learning rate, α, is too high.

Option 1 is false; Q-learning can succeed with randomly generated samples. That's actually a strength of the algorithm.

Option 2 is false. Discount factor redefines the optimal policy; ie. what is optimal depends on your discount factor.

Option 3 is true. With $\alpha = 0.5$, half of the Q-value comes from the most recent observed sample, which is very unstable.

(e) [2 pts] One way to view this algorithm is that step 1 focuses on **(A)**, and step 4 focuses on **(B)**.

- ○ **(A)**: exploitation, **(B)**: exploration
- ● **(A)**: exploration, **(B)**: exploitation
- ○ **(A)**: finding good states, **(B)**: finding bad states
- ○ **(A)**: finding good actions, **(B)**: finding bad actions

A completely random policy is like setting ϵ to 1, for fully random exploration.

Because the second policy is following Q values that we learned, there is no exploration occurring (no randomness).

(f) [2 pts] Is Q_{smart} more accurate (i.e., closer to Q^*), compared to Q_{rand}?

- ○ For all states, Q_{smart} is strictly more accurate than Q_{rand}.
- ● For all states, Q_{smart} is at least as accurate as Q_{rand}.
- ○ Q_{smart} is more accurate in some states, and less accurate in other states.
- ○ Q_{smart} is identical to Q_{rand}.

Option 2 is correct. Step 5 means that the only state ation pairs that get updated during Q-learning will be states that the "exploitating" policy π_{smart} visits. As α is lower, it will have less noisy updates to a subset of states. Q_{smart} will be better for the states that get visited, and the same as Q_{rand} for states that don't get visited.

Q6. [17 pts] Potpourri: Arbitrary Abstraction

Arbitrary means you know the definitions we discussed in class and the details mentioned in the problem, and nothing else.

(a) [2 pts] Consider an arbitrary two-agent, zero-sum game. We use a complete minimax game tree to decide the action we take. Is it possible to achieve more utility by taking another action?

- ○ Always possible
- ● Sometimes possible
- ○ Never possible

Only B is correct; in the case that your opponent is suboptimal, you could achieve more utility by selecting a different action.

For the next four subparts, consider an arbitrary, deterministic, *sparse* MDP, with a single terminal state.

In a sparse MDP, the reward function $R(s, a, s')$ returns zero for all transitions, (s, a, s'), except for a single transition that ends at the terminal state.

For every two states, there exists a sequence of actions that connect them, i.e. the state space graph is connected. You know nothing else about the transition function, $T(s, a, s')$, other than that it is deterministic.

Suppose there are N non-terminal states, and for every state, you can take M actions.

(b) [1 pt] If we run value iteration, when is the **earliest** possible iteration k that value iteration converges—the smallest k where $V_k = V^*$?

○ 0	● 2	○ M	○ $N \times M$
○ 1	○ N	○ $N + M$	○ N^M

(c) [1 pt] When is the **latest** possible iteration k that value iteration converges—the largest k where $V_k = V^*$, but $V_{k-1} \neq V^*$?

○ 0	○ 2	○ M	○ $N \times M$
○ 1	● N	○ $N + M$	○ N^M

(d) [1 pt] If we run policy iteration, when is the **earliest** possible iteration i that policy iteration converges—the smallest i where $\pi_i = \pi^*$?

Assume that we initialize with a random policy.

● 0	○ 2	○ M	○ $N \times M$
○ 1	○ N	○ $N + M$	○ N^M

(e) [1 pt] When is the **latest** possible iteration i that policy iteration converges—the latest i where $\pi_i = \pi^*$, but $\pi_{i-1} \neq \pi^*$?

Assume that we initialize with a random policy.

○ 0	○ 2	○ M	○ $N \times M$
○ 1	● N	○ $N + M$	○ N^M

b) Suppose that the state s that is the starting location of the terminal transition is connected to all $N - 1$ other non-terminal states. Value iteration will therefore reach the optimal value in one iteration for state s. All other non-terminal states will be updated at iteration 2.

c) The largest k is a chain of nodes, where at $k = 1$, one state updates, $k = 2$, a second state updates, all the way to $k = N$ nodes.

d) For policy iteration, it is technically possible that the random initialization, π_0, is the correct one!

e) The latest i is the same structure as described for value iteration. Suppose that on initialization, all the actions are wrong; they get updated one by one through the chain.

(f) [2 pts] Consider three random variables, A, B, C. We know that A and B are conditionally independent given C.

Select two terms that are equivalent.

- ☐ $P(A)$
- ■ $P(A|C)$
- ■ $P(A|B,C)$
- ☐ $P(A|B)$
- ○ All the terms are different.

By definition, A and B are conditionally independent given C means that $P(A|C) = P(A|B,C)$; knowing the information of B does not give you more information about A, because they are independent.

For the next three subparts, consider two arbitrary MDPs, M_X and M_Y. They have the same state space and action space, but each MDP has its own unknown transition function, T_X and T_Y respectively. We run model-based learning on both MDPs to get $\hat{T}_X, \hat{T}_Y$.

M_Z is a third arbitrary MDP that has the same state and action spaces as M_X and M_Y, but has a weighted average transition function, $T_Z = k_X T_X + k_Y T_Y$, where $k_X + k_Y = 1$. You do not know k_X and k_Y.

Let's explore how we might "reuse" our transition functions $\hat{T}_X, \hat{T}_Y$ to estimate $\hat{T}_Z$.

(g) [2 pts] Blinky gives us a sample, (s, a, s'), taken from M_Z.

Suppose that $\hat{T}_X(s,a,s') >> \hat{T}_Y(s,a,s')$, and $\hat{k}_X = \hat{k}_Y = 0.5$.

$A >> B$ means that A is much greater than B.

How should we change the weights, $\hat{k}_X$ and $\hat{k}_Y$, to correspond with Blinky's information?

- [x] $\hat{k}_X$ should increase.
- [ ] $\hat{k}_X$ should decrease.
- [ ] $\hat{k}_Y$ should increase.
- [x] $\hat{k}_Y$ should decrease.
- ○ None of the above.

If we observe a sample much more likely to come from one probability distribution $\hat{T}_X$ then another $\hat{T}_Y$, then intuitively we believe that Blinky's transition function mix contains more of $\hat{T}_X$. $\hat{k}_X$ should increase, and $\hat{k}_Y$ should decrease.

Aside: In the limit of infinite samples, seeing Blinky's sample in higher proportion would lead the Maximum Likelihood Estimation for k_X, k_Y to shift towards k_X. Although we are only looking a single sample, under the concepts of Q-learning we covered in class, the proportion we see a sample in the long run informs our belief of the transition dynamics; ie. the more likely a transition is, the more samples that correspond to that transition will appear.

(h) [4 pts] Pacman suggests that we should use the following learning update for $\hat{k}_X$:

$$\hat{k}_X \leftarrow \hat{k}_X + \alpha\Big(\hat{T}_X(s,a,s') - \hat{T}_Z(s,a,s')\Big)$$

Select all true statements about this update.

- [x] $\hat{k}_X$ increases when a sample is more likely from M_X than M_Z.
- [ ] $\hat{k}_X$ increases when a sample is more likely from M_Z than M_X.
- [ ] $\hat{k}_X$ does not change when $\hat{k}_X$ is correct, i.e. matches Blinky's true value.
- [ ] $\hat{k}_X$ can only increase.
- ○ None of the above.

Only option 1 is correct. $\hat{T}_X$ corresponds to M_X, and $\hat{T}_Z$ corresponds to M_Z.

For option 3, note that even if $\hat{k}_X$ is correct, it's possible to see a sample that "came from" M_Y. The difference in the large parentheses would be non-zero.

Option 4 is incorrect. It can decrease if the probability specified by $\hat{T}_Z$ is larger than $\hat{T}_X$.

(i) [3 pts] What else needs to be done in addition to Pacman's update in the previous subpart to correctly update $\hat{k}_X$?

Hint: There is a one-word answer, but you can answer with up to 8 words. What kind of function is $\hat{T}_Z$?

Normalization	k_X and k_Y need to sum to one.

The question asks about how $\hat{k}_X$ gets updated. When you apply this learning update for $\hat{k}_X$, you may increase or decrease $\hat{k}_X$. Conceptually, you must ensure that the implied transition function $\hat{T}_Z$ is still valid. That means that $\hat{k}_X + \hat{k}_Y$ need to sum to one, and each one needs to be in the bounds of $[0, 1]$. Normalization is the correct way to do this; $\hat{k}_X = \frac{\hat{k}_X}{\hat{k}_X + \hat{k}_Y}$.

The hint refers to the fact that $\hat{T}_Z$ is a transition function, which outputs probabilities. Probabilities have to sum to one and are bounded between [0, 1].

———

Aside: A common answer was setting $\hat{k}_Y$ to $1 - \hat{k}_X$. This very close to right, but not exactly. (It also doesn't change $\hat{k}_X$ as asked by the question). The key error is that you cannot skip independently updating $\hat{k}_Y$.

Consider if our estimate for $\hat{k}_X$ approaches 1, and then we receive a sample that is impossible under T_X. Under Pacman's update, $\hat{T}_X = \hat{T}_Z$ since $\hat{k}_X = 1$. Then our update will not change $\hat{k}_X$. (Even if we say that $\hat{k}_X$ is very close but less than 1, the update will be extremely tiny, and empirically terrible). We need to update $\hat{k}_Y$ independently; ie. copy the equation, but replace instances of X with Y. Then, if we recieved an impossible sample for T_X but a sample that looks like it's from T_Y, we could update $\hat{k}_Y$ appropriately.

Conceptually, this problem is similar weight-learning with approximate Q-learning. We should conduct individual updates to our weights. In this case, our features specify a probability distribution, so they should also be normalized.