CS 188 Spring 2014 — Introduction to Artificial Intelligence — Midterm 1

- You have approximately 2 hours and 50 minutes.
- The exam is closed book, closed notes except your one-page crib sheet.
- Mark your answers ON THE EXAM ITSELF. If you are not sure of your answer you may wish to provide a *brief* explanation. All short answer sections can be successfully answered in a few sentences AT MOST.

First name	
Last name	
SID	
edX username	

First and last name of student to your left	
First and last name of student to your right	

For staff use only:

Q1.	All Searches Lead to the Same Destination	/10
Q2.	Dynamic A* Search	/18
Q3.	CSPs	/16
Q4.	Variants of Trees	/15
Q5.	More Games and Utilities	/10
Q6.	Faulty Keyboard	/9
Q7.	Deep inside Q-learning	/8
Q8.	Bellman Equations for the Post-Decision State	/14
	Total	/100

THIS PAGE IS INTENTIONALLY LEFT BLANK

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

For all the questions below assume :

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

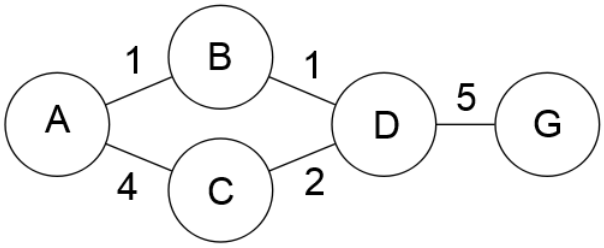
A^* search expands the node with the lowest $f(n) + h(n)$ value, where $f(n) = \sum_{ij} c_{ij}$ is the cost-so-far and h is the heuristic value. If $d_{ij} = c_{ij} + h(j) - h(i)$, then the cost of a node n on the fringe when running uniform-cost search will be $\sum_{ij} d_{ij} = \sum_{ij} c_{ij} +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

Q2. [18 pts] Dynamic A* Search

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

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

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



Fringe: {}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This is the same as part (c).

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

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

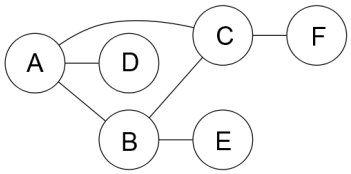
Assuming that the cost of $X \to Y$ remains positive, because the edge was never explored, the cost of the path to X is already higher than the cost of the optimal path. Thus, the cost of the path to Y through X can only be higher, so the optimal path remains the same.

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

Q3. [16 pts] CSPs

(a) The graph below is a constraint graph for a CSP that has only binary constraints. Initially, no variables have been assigned.

For each of the following scenarios, mark all variables for which the specified filtering might result in their domain being changed.



(i) [1 pt] A value is assigned to A. Which domains might be changed as a result of running forward checking for A?

○ A ● B ● C ● D ○ E ○ F

Forward checking for A only considers arcs where A is the head. This includes $B \to A$, $C \to A$, $D \to A$. Enforcing these arcs can change the domains of the tails.

(ii) [1 pt] A value is assigned to A, and then forward checking is run for A. Then a value is assigned to B. Which domains might be changed as a result of running forward checking for B?

○ A ○ B ● C ○ D ● E ○ F

Similar to the previous part, forward checking for B enforces the arcs $A \to B$, $C \to B$, and $E \to B$. However, because A has been assigned, and a value is assigned to B, which is consistent with A or else no value would have been assigned, the domain of A will not change.

(iii) [1 pt] A value is assigned to A. Which domains might be changed as a result of enforcing arc consistency after this assignment?

○ A ● B ● C ● D ● E ● F

Enforcing arc consistency can affect any unassigned variable in the graph that has a path to the assigned variable. This is because a change to the domain of X results in enforcing all arcs where X is the head, so changes propagate through the graph. Note that the only time in which the domain for A changes is if any domain becomes empty, in which case the arc consistency algorithm usually returns immediately and backtracking is required, so it does not really make sense to consider new domains in this case.

(iv) [1 pt] A value is assigned to A, and then arc consistency is enforced. Then a value is assigned to B. Which domains might be changed as a result of enforcing arc consistency after the assignment to B?

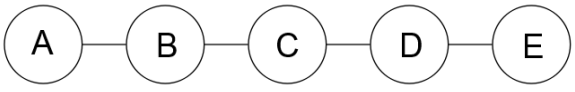
○ A ○ B ● C ○ D ● E ● F

After assigning a value to A, and enforcing arc consistency, future assignments and enforcing arc consistency will not result in a change to As domain. This means that Ds domain won't change because the only arc that might cause a change, $D \to A$ will never be enforced.

(b) You decide to try a new approach to using arc consistency in which you initially enforce arc consistency, and then enforce arc consistency every time you have assigned an even number of variables.

You have to backtrack if, after a value has been assigned to a variable, X, the recursion returns at X without a solution. Concretely, this means that for a single variable with d values remaining, it is possible to backtrack up to d times. For each of the following constraint graphs, if each variable has a domain of size d, how many times would you have to backtrack in the worst case for each of the specified orderings?

(i) [6 pts]



A-B-C-D-E: 0

A-E-B-D-C: $2d$

C-B-D-E-A: 0

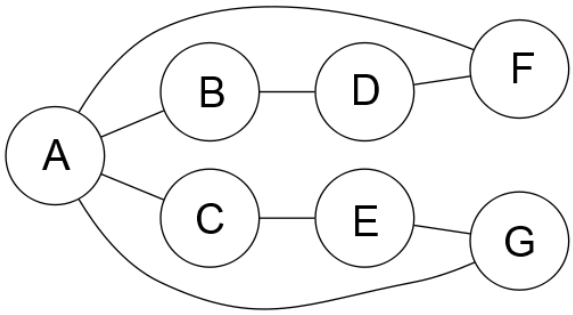
If no solution containing the current assignment exists on a tree structured CSP, then enforcing arc consistency will always result in an empty domain. This means that running arc consistency on a tree structured

CSP will immediately tell you whether or not the current assignment is part of a valid solution, so you can immediately start backtracking without further assignments.

$A - B - C - D - E$ and $C - B - D - E - A$ are both linear orderings of the variables in the tree, which is essentially the same as running the two pass algorithm, which will solve a tree structured CSP with no backtracking.

$A - E - B - D - C$ is not a linear ordering, so while the odd assignments are guaranteed to be part of a valid solution, the even assignments are not (because arc consistency was not enforced after assigning the odd variables). This means that you may have to backtrack on every even assignment, specifically E and D. Note that because you know whether or not the assignment to E is valid immediately after assigning it, the backtracking behavior is not nested (meaning you backtrack on E up to d times without assigning further variables). The same is true for D, so the overall behavior is backtracking $2d$ times.

(ii) [6 pts]



A-B-C-D-E-F-G: d^2

F-D-B-A-C-G-E: $d^4 + d$

C-A-F-E-B-G-D: d^2

$A - B - C - D - E - F - G$: The initial assignment of A, B might require backtracking on both variables, because there is no guarantee that the initial assignment to A is a valid solution. Because A is a cutset for this graph, the resulting graph consists of two trees, so enforcing arc consistency immediately returns whether the assignments to A and B are part of a solution, and you can begin backtracking without further assignments.

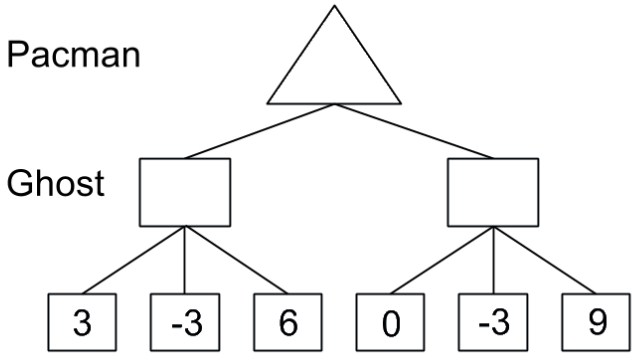
$F - D - B - A - C - G - E$: Until A is assigned, there is no guarantee that any of the previous values assigned are part of a valid solution. This means that you may be required to backtrack on all of them, resulting in d^4 times. Furthermore, the remaining tree is not assigned in a linear order, so further backtracking may be required on G (similar to the second ordering above) resulting in a total of $d^4 + d$.

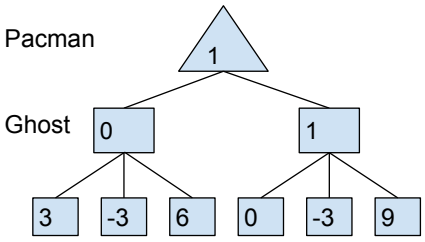
$C - A - F - E - B - G - D$: This ordering is similar to the first one. except that the resulting trees are not being assigned in linear order. However, because each tree only has a single value assigned in between each run of arc consistency, no backtracking will be required (you can think of each variable as being the root of the tree, and the assignment creating a new tree or two where arc consistency has been enforced), resulting in a total of d^2 times.

Q4. [15 pts] Variants of Trees

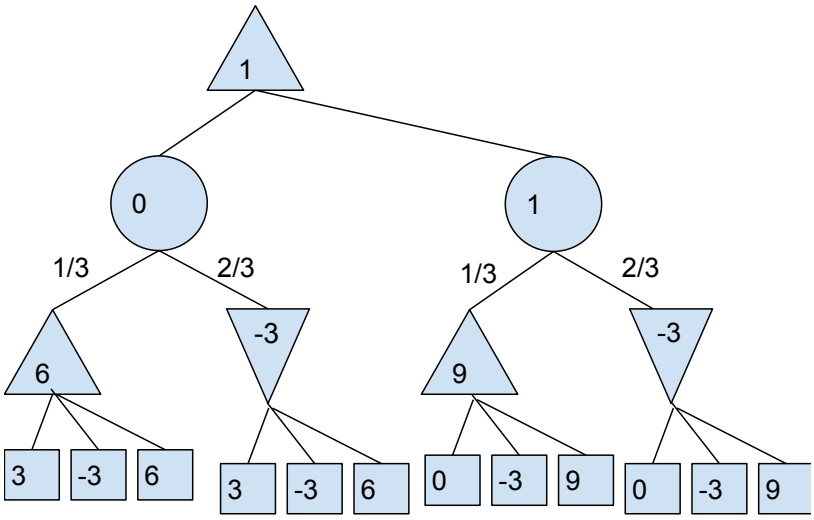
(a) Pacman is going to play against a careless ghost, which makes a move that is optimal for Pacman $\frac{1}{3}$ of the time, and makes a move that that minimizes Pacman's utility the other $\frac{2}{3}$ of the time.

(i) [2 pts] Fill in the correct utility values in the game tree below where Pacman is the maximizer:

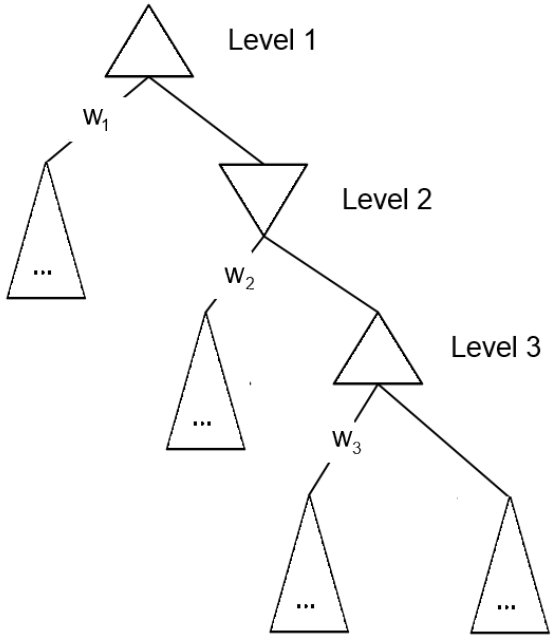




(ii) [2 pts] Draw a complete game tree for the game above that contains only max nodes, min nodes, and chance nodes.



(b) Consider a modification of alpha-beta pruning where, rather than keeping track of a single value for α and β, you instead keep a list containing the best value, w_i, for the minimizer/maximizer (depending on the level) at each level up to and including the current level. Assume that the root node is always a max node. For example, consider the following game tree in which the first 3 levels are shown. When the considering the right child of the node at level 3, you have access to w_1, w_2, and w_3.



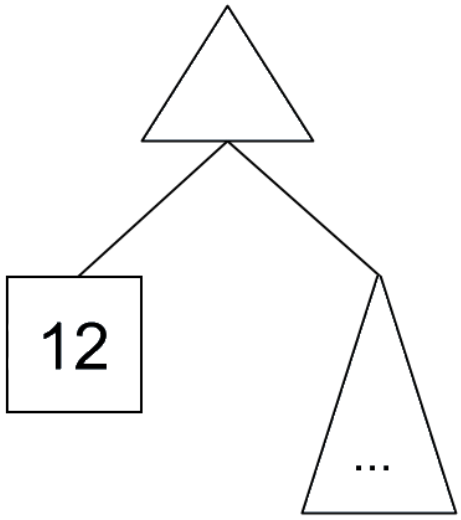
(i) [1 pt] Under this new scenario, what is the pruning condition for a max node at the n^{th} level of the tree (in terms of v and $w_1...w_n$)? $v > min_i(w_i)$, where i is even;

(ii) [1 pt] What is the pruning condition for a min node at the n^{th} level of the tree?
$v < max_i(w_i)$, where i is odd;

(iii) [2 pts] What is the relationship between α, β and the list of $w_1...w_n$ at a max node at the n^{th} level of the tree?

- ◯ $\sum_i w_i = \alpha + \beta$
- ◯ $\max_i w_i = \alpha, \quad \min_i w_i = \beta$
- ◯ $\min_i w_i = \alpha, \quad \max_i w_i = \beta$
- ◯ $w_n = \alpha, \quad w_{n-1} = \beta$
- ◯ $w_{n-1} = \alpha, \quad w_n = \beta$
- ◯ None of the above. The relationship is $\beta = min(w_2, w_4, w_6...), \quad \alpha = max(w_1, w_3, w_5...)$

(c) Pacman is in a dilemma. He is trying to maximize his overall utility in a game, which is modeled as the following game tree.



The left subtree contains a utility of 12. The right subtree contains an unknown utility value. An oracle has told you that the value of the right subtree is one of -3, -9, or 21. You know that each value is equally likely, but without exploring the subtree you do not know which one it is.

Now Pacman has 3 options:

1. Choose left;
2. Choose right;
3. Pay a cost of c $= 1$ to explore the right subtree, determine the exact utility it contains, and then make a decision.

(i) [3 pts] What is the expected utility for option 3? $12 * \frac{2}{3} + 21 * \frac{1}{3} - 1 = 14$

(ii) [4 pts] For what values of c (for example, $c > 5$ or $-2 < c < 2$) should Pacman choose option 3? If option 3 is never optimal regardless of the value for c, write None.
According to the previous part, we have $15 - c > 12$, then we will choose option 3.
Therefore, $c < 3$

Q5. [10 pts] More Games and Utilities

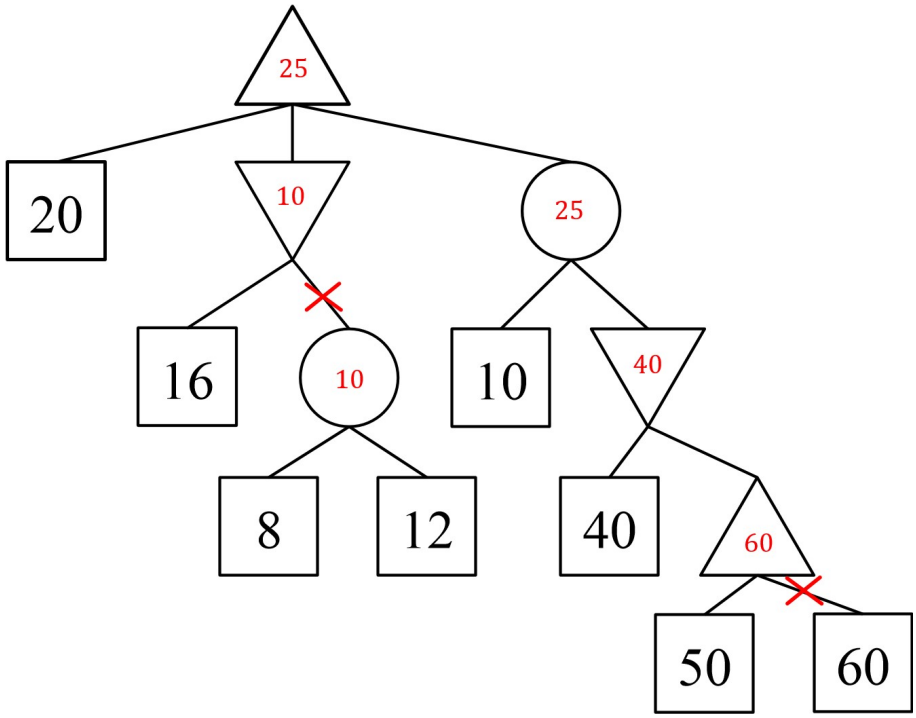
(a) Games. Consider the game tree below, which contains maximizer nodes, minimizer nodes, and chance nodes. For the chance nodes the probability of each outcome is equally likely.

(i) [3 pts] Fill in the values of each of the nodes.

(ii) [4 pts] Is pruning possible?

◯ No. Brief justification: _____

◯ Yes. Cross out the branches that can be pruned.



(b) Utilities. Pacman's utility function is $U(\$x) = \sqrt{x}$. He is faced with the following lottery: $[0.5, \$36\ ;\ 0.5, \$64]$. Compute the following quantities:

(i) [1 pt] What is Pacman's expected utility?

$EU([0.5, \$36\ ;\ 0.5,\ \$64]) =$
The utility of a lottery can be computed via $EU(L) = \sum_{x \in L} p(x)U(x)$
Hence, $EU([0.5, \$36\ ;\ 0.5,\ \$64]) = 0.5U(\$36) + 0.5U(\$64) = 0.5(6) + 0.5(8) = 7$

(ii) [1 pt] What is Equivalent Monetary Value for this lottery?

$EMV([0.5, \$36\ ;\ 0.5, \$64]) =$
The equivalent monetary value is the amount of money you would pay in exchange for the lottery.
Since the utility of the lottery was calculated to be 7, then we just need to find the X such that $U(\$X) = 7$
Answer: $X = \$49$

(iii) [1 pt] What is the maximum amount Pacman would be willing to pay for an insurance that guarantees he gets \$64 in exchange for giving his lottery to the insurance company?

Solve for X in the equation: $U(64 - \$X) = U(Lottery) = 7$
Answer: $X = \$15$

Q6. [9 pts] Faulty Keyboard

Your new wireless keyboard works great, but when the battery runs out it starts behaving strangely. All keys other than $\{a, b, c\}$ stop working. The $\{a, b, c\}$ keys work as intended with probability 0.8, but could also produce either of the other two different characters with probability 0.1 each. For example, pressing a will produce 'A' with probability 0.8, 'B' with probability 0.1, and 'C' with probability 0.1. After 11 key presses the battery is completely drained and no new characters can be produced.

The naïve state space for this problem consists of three states $\{A, B, C\}$ and three actions $\{a, b, c\}$.

For each of the objectives below specify a reward function on the naïve state space such that the optimal policy in the resulting MDP optimizes the specified objective. The reward can only depend on current state, current action, and next state; it cannot depend on time.

If no such reward function exist, then specify how the naïve state space needs to be expanded and a reward function on this expanded state space such that the optimal policy in the resulting MDP optimizes the specified objective. The size of the extended state space needs to be minimal. You may assume that $\gamma = 1$.

There are many possible correct answers to these questions. We accepted any reward function that produced the correct policy, and any state-space extension that produced a state-space of minimal asymptotic complexity. (Big O complexity of the state space size in terms on the horizon and the number of keys and characters available)

(a) [3 pts] Produce as many B's as possible before the first C.

● $R(s, a, s') =$ 1 if $s' = B$, 0 otherwise For this part, we want to achieve the policy of always taking action b. Many reward functions gave this policy (giving positive reward for taking action b achieved the same thing). Notice that we do not need a bit for whether C has been pressed or not, because it will not affect our optimal policy or transitions.
Another correct reward function might be $R(s, a, s') = 1$ if a='b', 0 otherwise.

○ The state-space needs to be extended to include: _____

Reward function over extended state space: _____

(b) [3 pts] Produce a string where the number of B's and the number of C's are as close as possible.

○ $R(s, a, s') =$ _____

● The state-space needs to be extended to include: a counter D for the difference between the number of B's and the number of C's. D = #B-#C = number of B's minus number of C's in s. New state-space size: $3 \cdot 23$

Reward function over extended state space: $R(s, a, s') = -|D(s')|$

For this part, you need to keep track of the difference between the number of B's you have seen so far and the number of C's you have seen so far. A significant number of people wrote that you can explicitly store both the counts of B and C. While this is a valid state space, it is not minimal, since it shouldn't matter whether you are, say, in a situation where you pressed 4 B's and 4 C's, and a situation where you pressed 2 B's and 2 C's.

The reward function above is one of many that can reproduce the behavior specified. This reward function basically assigns a score of 1 for any transition that decreases the counter, a score of 0 for any transition that doesn't change the counter, and a score of -1 for any transition that increases the counter by 1.

A significant number of people had the right idea for the reward function, but only wrote down an equation that relates number of B's and number of C's. (example: $-|\#B - \#C|$). While close, it's ambiguous in the sense that it's not clear if these numbers are referring to the state s or s', which are both parameters to the reward function.

Another correct but non-minimal extension of the state space is one where we add both a counter for the number of B' and a counter for the number of C's. In this case the size of the state-space would be: $3 \cdot 12 \cdot 12$. Another possible state space extension that leads to a smaller state space (but with the same complexity as our answer) can be obtained by noticing that once $|D(s)| > 5$ the policy for the remaining time steps is determined. Therefore we could augment the state space with $\max(-6, \min(6, D))$ instead of $D(s)$.

(c) [3 pts] Produce a string where the first character re-appears in the string as many times as possible.

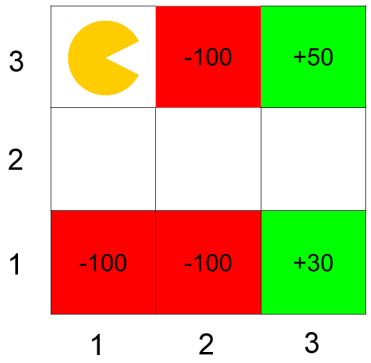
○ $R(s, a, s') =$ _____

● The state-space needs to be extended to include: character produced after the first keypress
The size of the state-space is now $3 \cdot 3$.
first(s) = first character produced on the way to state s.
Note that this new variable added to the state-space remains constant after the first character is produced.

Reward function over extended state space: $R(s, a, s') = 1$ if $a = \text{lowercase}(\text{first}(s))$, 0 otherwise

Q7. [8 pts] Deep inside Q-learning

Consider the grid-world given below and an agent who is trying to learn the optimal policy. Rewards are only awarded for taking the *Exit* action from one of the shaded states. Taking this action moves the agent to the Done state, and the MDP terminates. Assume $\gamma = 1$ and $\alpha = 0.5$ for all calculations. All equations need to explicitly mention γ and α if necessary.



(a) [3 pts] The agent starts from the top left corner and you are given the following episodes from runs of the agent through this grid-world. Each line in an Episode is a tuple containing (s, a, s', r).

Episode 1	Episode 2	Episode 3	Episode 4	Episode 5
(1,3), S, (1,2), 0	(1,3), S, (1,2), 0	(1,3), S, (1,2), 0	(1,3), S, (1,2), 0	(1,3), S, (1,2), 0
(1,2), E, (2,2), 0	(1,2), E, (2,2), 0	(1,2), E, (2,2), 0	(1,2), E, (2,2), 0	(1,2), E, (2,2), 0
(2,2), E, (3,2), 0	(2,2), S, (2,1), 0	(2,2), E, (3,2), 0	(2,2), E, (3,2), 0	(2,2), E, (3,2), 0
(3,2), N, (3,3), 0	(2,1), Exit, D, -100	(3,2), S, (3,1), 0	(3,2), N, (3,3), 0	(3,2), S, (3,1), 0
(3,3), Exit, D, +50		(3,1), Exit, D, +30	(3,3), Exit, D, +50	(3,1), Exit, D, +30

Fill in the following Q-values obtained from direct evaluation from the samples:

$Q((3,2), \text{N}) =$ 50 $\quad Q((3,2), \text{S}) =$ 30 $\quad Q((2,2), \text{E}) =$ 40

Direct evaluation is just averaging the discounted reward after performing action a in state s.

(b) [3 pts] Q-learning is an online algorithm to learn optimal Q-values in an MDP with unknown rewards and transition function. The update equation is:

$$Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1}, a'))$$

where γ is the discount factor, α is the learning rate and the sequence of observations are $(\cdots, s_t, a_t, s_{t+1}, r_t, \cdots)$. Given the episodes in (a), fill in the time at which the following Q values first become non-zero. Your answer should be of the form (**episode#,iter#**) where **iter#** is the Q-learning update iteration in that episode. If the specified Q value never becomes non-zero, write *never*.

$Q((1,2), \text{E}) =$ (4,2) $\quad Q((2,2), \text{E}) =$ (3,3) $\quad Q((3,2), \text{S}) =$ (3,4)

This question was intended to demonstrate the way in which Q-values propagate through the state space. Q-learning is run in the following order - observations in ep 1 then observations in ep 2 and so on.

(c) [2 pts] In Q-learning, we look at a window of (s_t, a_t, s_{t+1}, r_t) to update our Q-values. One can think of using an update rule that uses a larger window to update these values. Give an update rule for $Q(s_t, a_t)$ given the window $(s_t, a_t, r_t, s_{t+1}, a_{t+1}, r_{t+1}, s_{t+2})$.

$Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \gamma r_{t+1} + \gamma^2 \max_{a'} Q(s_{t+2}, a'))$
(Sample of the expected discounted reward using r_{t+1})

$Q(s_t, a_t) = (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma((1-\alpha)Q(s_{t+1}, a_{t+1}) + \alpha(r_{t+1} + \gamma \max_{a'} Q(s_{t+2}, a'))))$
(Nested Q-learning update)

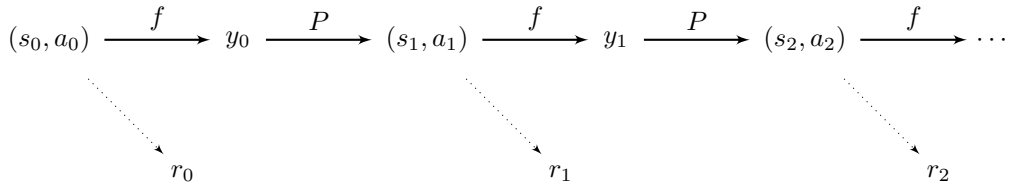
$Q(s_t, a_t) = (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max((1-\alpha)Q(s_{t+1}, a_{t+1}) + \alpha(r_{t+1} + \gamma \max_{a'} Q(s_{t+2}, a')), \max_{a'} Q(s_{t+1}, a')))$
(Max of normal Q-learning update and one step look-ahead update)

Q8. [14 pts] Bellman Equations for the Post-Decision State

Consider an infinite-horizon, discounted MDP (S, A, T, R, γ). Suppose that the transition probabilities and the reward function have the following form:

$$T(s, a, s') = P(s'|f(s, a))$$
$$R(s, a, s') = R(s, a)$$

Here, f is some deterministic function mapping $S \times A \to Y$, where Y is a set of states called *post-decision states*. We will use the letter y to denote an element of Y, i.e., a post-decision state. In words, the state transitions consist of two steps: a deterministic step that depends on the action, and a stochastic step that does not depend on the action. The sequence of states (s_t), actions (a_t), post-decision-states (y_t), and rewards (r_t) is illustrated below.



You have learned about $V^\pi(s)$, which is the expected discounted sum of rewards, starting from state s, when acting according to policy π.

$$V^\pi(s_0) = E\left[R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \dots\right]$$
$$\text{given } a_t = \pi(s_t) \text{ for } t = 0, 1, 2, \dots$$

$V^*(s)$ is the value function of the optimal policy, $V^*(s) = \max_\pi V^\pi(s)$.

This question will explore the concept of computing value functions on the post-decision-states y. [1]

$$W^\pi(y_0) = E\left[R(s_1, a_1) + \gamma R(s_2, a_2) + \gamma^2 R(s_3, a_3) + \dots\right]$$

We define $W^*(y) = \max_\pi W^\pi(y)$.

(a) [2 pts] Write W^* in terms of V^*.
$W^*(y) =$

- ● $\sum_{s'} P(s' \mid y) V^*(s')$
- ○ $\sum_{s'} P(s' \mid y)[V^*(s') + \max_a R(s', a)]$
- ○ $\sum_{s'} P(s' \mid y)[V^*(s') + \gamma \max_a R(s', a)]$
- ○ $\sum_{s'} P(s' \mid y)[\gamma V^*(s') + \max_a R(s', a)]$
- ○ None of the above

Consider the expected rewards under the optimal policy.

$$\begin{aligned} W^*(y_0) &= E\left[R(s_1, a_1) + \gamma R(s_2, a_2) + \gamma^2 R(s_3, a_3) + \dots \mid y_0\right] \\ &= \sum_{s_1} P(s_1 \mid y_0) E\left[R(s_1, a_1) + \gamma R(s_2, a_2) + \gamma^2 R(s_3, a_3) + \dots \mid s_1\right] \\ &= \sum_{s_1} P(s_1 \mid y_0) V^*(s_1) \end{aligned}$$

V^* is time-independent, so we can replace y_0 by y and replace s_1 by s', giving

$$W^*(y) = \sum_{s'} P(s' \mid y) V^*(s')$$

[1] In some applications, it is easier to learn an approximate W function than V or Q. For example, to use reinforcement learning to play Tetris, a natural approach is to learn the value of the block pile *after* you've placed your block, rather than the value of the pair (current block, block pile). TD-Gammon, a computer program developed in the early 90s, was trained by reinforcement learning to play backgammon as well as the top human experts. TD-Gammon learned an approximate W function.

(b) [2 pts] Write V^* in terms of W^*.
$V^*(s) =$

- ○ $\max_a[W^*(f(s,a))]$
- ○ $\max_a[R(s,a) + W^*(f(s,a))]$
- ● $\max_a[R(s,a) + \gamma W^*(f(s,a))]$
- ○ $\max_a[\gamma R(s,a) + W^*(f(s,a))]$
- ○ None of the above

$$\begin{aligned}
V^*(s_0) &= \max_{a_0} Q(s_0, a_0) \\
&= \max_{a_0} E\left[R(s_0,a_0) + \gamma R(s_1,a_1) + \gamma^2 R(s_2,a_2) + \dots \mid s_0, a_0\right] \\
&= \max_{a_0} \left(E\left[R(s_0,a_0) \mid s_0,a_0\right] + E\left[\gamma R(s_1,a_1) + \gamma^2 R(s_2,a_2) + \dots \mid s_0,a_0\right]\right) \\
&= \max_{a_0} \left(R(s_0,a_0) + E\left[\gamma R(s_1,a_1) + \gamma^2 R(s_2,a_2) + \dots \mid f(x_0,a_0)\right]\right) \\
&= \max_{a_0} \left(R(s_0,a_0) + \gamma W^*(f(s_0,a_0))\right)
\end{aligned}$$

Renaming variables, we get

$$V^*(s) = \max_a \left(R(s,a) + \gamma W^*(f(s,a))\right)$$

(c) [4 pts] Recall that the optimal value function V^* satisfies the Bellman equation:

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

which can also be used as an update equation to compute V^*.
Provide the equivalent of the Bellman equation for W^*.

$W^*(y) =$ $\sum_{s'} P(s'|y) \max_a \left(R(s',a) + \gamma W^*(f(s',a))\right)$

The answer follows from combining parts (a) and (b)

(d) [3 pts] Fill in the blanks to give a policy iteration algorithm, which is guaranteed return the optimal policy π^*.

- Initialize policy $\pi^{(1)}$ arbitrarily.
- For $i = 1, 2, 3, \dots$
  - Compute $W^{\pi^{(i)}}(y)$ for all $y \in Y$.
  - Compute a new policy $\pi^{(i+1)}$, where $\pi^{(i+1)}(s) = \arg\max_a$ ____(1)____ for all $s \in S$.
  - If ____(2)____ for all $s \in S$, **return** $\pi^{(i)}$.

Fill in your answers for blanks (1) and (2) below.

(1)
- ○ $W^{\pi^{(i)}}(f(s,a))$
- ○ $R(s,a) + W^{\pi^{(i)}}(f(s,a))$
- ● $R(s,a) + \gamma W^{\pi^{(i)}}(f(s,a))$

◯ $\gamma R(s,a) + W^{\pi^{(i)}}(f(s,a))$

◯ None of the above

(2) $\pi^{(i)}(s) = \pi^{(i+1)}(s)$

Policy iteration performs the following update:

$$\pi^{(i+1)}(s) = \arg\max_a Q^{\pi^{(i)}}(s,a)$$

Next we express Q^π in terms of W^π (similarly to part b):

$$\begin{aligned} Q^\pi(s_0,a_0) &= E\left[R(s_0,a_0) + \gamma R(s_1,a_1) + \gamma^2 R(s_2,a_2) + \dots \mid s_0, a_0\right] \\ &= R(s_0,a_0) + \gamma E\left[R(s_1,a_1) + \gamma R(s_2,a_2) + \dots \mid f(s_0,a_0)\right] \\ &= R(s_0,a_0) + \gamma W^\pi(f(s_0,a_0)) \end{aligned}$$

(e) [3 pts] In problems where f is known but $P(s'|y)$ is not necessarily known, one can devise reinforcement learning algorithms based on the W^* and W^π functions. Suppose that an the agent goes through the following sequence of states, actions and post-decision states: $s_t, a_t, y_t, s_{t+1}, a_{t+1}, y_{t+1}$. Let $\alpha \in (0,1)$ be the learning rate parameter.

Write an update equation analogous to Q-learning that enables one to estimate W^*

$$W(y_t) \leftarrow (1-\alpha)W(y_t) + \alpha\left(\max_a \left(R(s_{t+1},a) + \gamma W(f(s_{t+1},a))\right)\right)$$

Recall the motivation for Q learning: the Bellman equation satisfied by the Q function is $Q(s,a) = E_{s'}[R(s,a) + \gamma \max_{a'} Q(s',a')]$. Q learning uses an estimate of the right-hand side of this equation obtained from a single sample transition s, a, s'. That is, we move $Q(s,a)$ towards $R(s,a) + \gamma \max_{a'} Q(s',a')$.

We have obtained a Bellman-like equation for W^* in part c.

$$\begin{aligned} W^*(y) &= \sum_{s'} P(s'|y) \max_{a'} \left(R(s',a') + \gamma W^*(f(s',a'))\right) \\ &= E_{s'}\left[\max_a \left(R(s',a) + \gamma W^*(f(s',a))\right)\right] \end{aligned}$$

If we don't know $P(s'|y)$, we can still estimate the expectation above from a single transition (y, s') using the expression inside the expectation: $\max_a \left(R(s',a) + \gamma W^*(f(s',a))\right)$. Doing a partial update with learning rate parameter α, we get

$$W(y_t) \leftarrow (1-\alpha)W(y_t) + \alpha \max_a \left(R(s_{t+1},a) + \gamma W(f(s_{t+1},a))\right)$$