1 Linear Programming (Conceptual Practice)

These questions were algebraic interpretations of the polls from LP lecture (polls 3 and 4 from lecture 6, 1 and 2 from lecture 7). Pat explained the polls using graphical representations, so this hopefully provides another way of looking at things.

Consider a linear program with a singular constraint of the form $a_1x_1 + a_2x_2 \leq b$ and cost vector $\boldsymbol{c} = \begin{bmatrix}-2 & 3\end{bmatrix}$. You are given the following list of possible points (x_1, x_2) to consider:

$$(5, -5), (-5, 5), (0, 5), (5, 0), (-5, -5), (5, 5)$$

1. Which of the points would maximize the objective value? Which of the points would minimize the objective value?

   Remember that objective value essentially means total cost, or the value of $\boldsymbol{c}^T\boldsymbol{x}$ - you should be familiar with both terms. Excluding (0, 5) and (5, 0), all our points have the same magnitude, so this becomes a matter of picking the right direction for $\boldsymbol{x}$. (-5, 5) is the maximum, (5, -5) is the minimum.

2. Suppose you are given $\boldsymbol{a} = \begin{bmatrix}5 & -4\end{bmatrix}$. What are the values $a_1x_1 + a_2x_2$ for each of the points? If $b = 9$, which points are feasible?

   In order, the values for each are 45, -45, -20, 25, -5, 5. (-5, 5), (0, 5), (-5, -5), (5, 5) are feasible.

3. With the same $\boldsymbol{a}$ from 2., which of our points is feasible and minimizes total cost? Going outside the given list of points, how would we decrease total cost? What is our **minimum** objective value?

   (-5, -5); we decrease total cost by moving along the line $5x_1 - 4x_2 = b$ with x_1 generally tending towards $+\infty$ and x_2 generally tending towards $-\infty$ (making $\boldsymbol{c}^T\boldsymbol{x}$ more negative, as c_1 is negative and c_2 positive).
   (Note that since we have to respect the constraint line, moving (x_1, x_2) in this general direction actually results in both x_1 and x_2 tending towards $-\infty$.)

4. Now with $\boldsymbol{a} = \begin{bmatrix}2 & -3\end{bmatrix}$, how would we decrease total cost? What is our **minimum** objective value?

   We decrease total cost again by traveling in the direction to make cost more negative (with x_1 tending towards $+\infty$ and x_2 tending towards $-\infty$). However, once we hit the line $2x_1 - 3x_2$ (which we cannot cross in order to stay feasible), we find that traveling along the line yields the same cost value of -9 for all points.

5. In general, when would minimizing LPs with a singular constraint not have a minimum objective at $-\infty$?

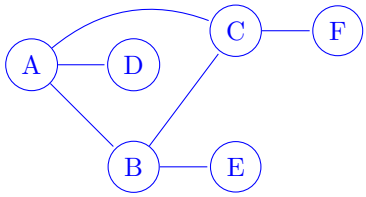
   When the cost vector is perpendicular to the constraint line $a_1x_1 + a_2x_2 = b$ (in other words, when the cost vector points opposite to the $\boldsymbol{a}$ vector), no matter where you move along the constraint line you will get the same cost.

2 CSPs (Arc Consistency, Search)

You've generously saved a row of 6 seats in Rashid for 6 of your 15-381 classmates (A-F[1]), and are now trying to figure out where each person will be seated. You know the following pairs of people have some kind of binary constraint between them:

- A, B
- A, C
- A, D
- B, C
- B, E
- C, F

1. Draw the constraint graph to represent this CSP.



2. Some value is assigned to A. Which domains could change as a result of running forward checking for A?

   B, C, D. 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.

3. Some value is assigned to A, and then forward checking is run for A. Then some value is assigned to B. Which domains could change as a result of running forward checking for B?

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

4. Some value is assigned to A. Which domains could change as a result of enforcing arc consistency after this assignment?

   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.

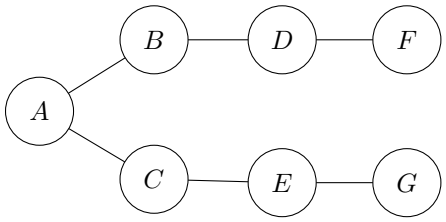
5. Some value is assigned to A, and then arc consistency is enforced. Then some value is assigned to B. Which domains could change as a result of enforcing arc consistency after B's assignment?

   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 D's domain wont change because the only arc that might cause a change, $D \to A$ will never be enforced.

[1] not correlated with their grades

6. You are now trying a brand new algorithm to solving CSPs by enforcing arc consistency initially, then **after every even-numbered assignment** of variables (after assigning 2 variables, then after 4, etc.).

   You have to backtrack if, after assigning a value to variable X, there are no constraint-satisfying solutions. Mathematically, for a single variable with d values remaining, it is possible to backtrack d times in the worst case. For each of the following constraint graphs, assume each variable has domain of size d. How many times would you have to backtrack in the worst case for the specified orderings of assignments?



   (a) ABCDEFG:

   0

   (b) GECABDF:

   0

   (c) GFEDCBA:

   $3d$

   ABCDEFG and GECABDF are both linear orderings of the variables in the tree, which is essentially the same as initially enforcing arc consistency then assigning values to nodes from root to leaves (which, as we saw in HW4, will not backtrack if a solution exists).

   GFEDBCA 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 F, D, and B. Note that because you know whether or not the assignment to F is valid immediately after assigning it, the backtracking behavior is not nested (meaning you backtrack on F up to d times without assigning further variables - otherwise, we would have a worst-case number of backtracks of n^3). The same is true for D and B, so the overall behavior is backtracking $3d$ times.

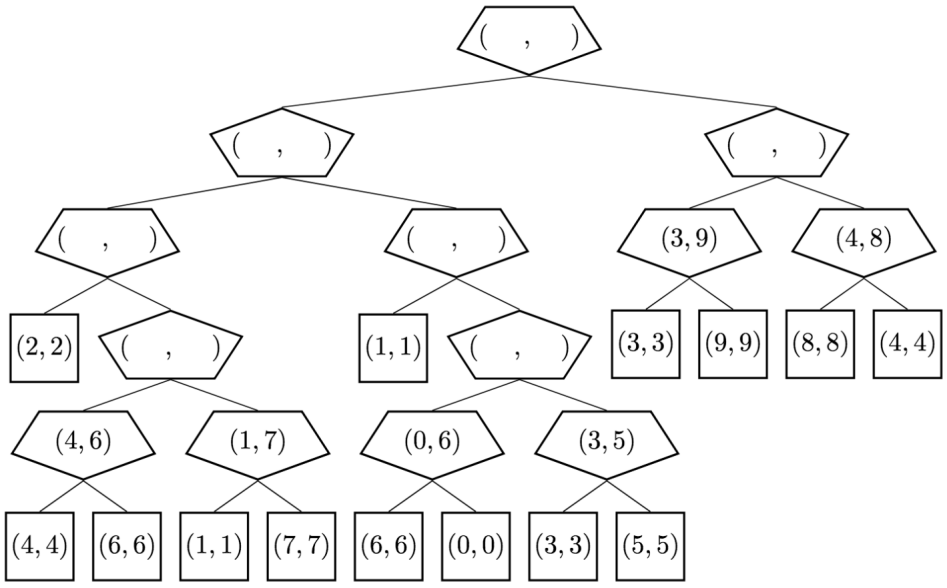
3 Games (Pruning)

Consider a two-player game between Pacman and a ghost (Casper) in which both agents alternate moves. As usual, Pacman tries to maximize his own utility. Unlike the usual minimizer ghost, Casper is friendly and **helps Pacman** by maximizing his utility. **Pacman is unaware** of this and acts as if Casper is playing against him. **Casper knows Pacman is misinformed** and acts accordingly.

1. For the game tree for this game, the value pair at each node is (u, v), where u is the value of the subtree as determined by Pacman, and v is the value of the subtree as determined by Casper.
   For terminal states, $u = v =$ the utility of that state.

   For example, in the subtree below with values (4, 6), Pacman believes Casper would choose the left action (with value = 4), but Casper actually chooses the right action (with value = 6), since that is better for Pacman.

   Fill in the remaining (u, v) values in the tree for this game below, where Casper is the root. Casper nodes are upside-down pentagons, and Pacman nodes are rightside-up pentagons.



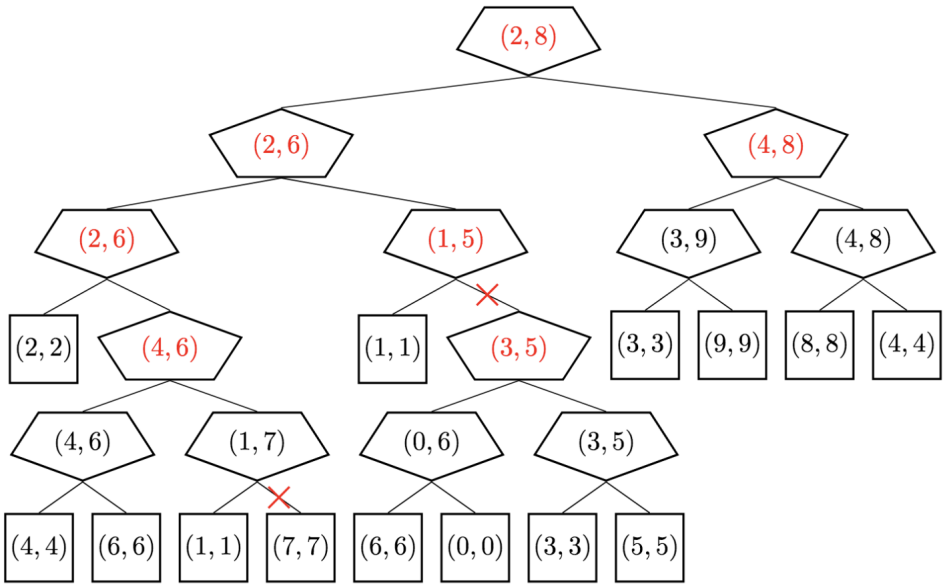
(See below for graphical explanation of answer)

The u value of Pacman's nodes is the maximum of the u values of the immediate children nodes since Pacman believes that the values of the nodes are given by u. The v value of Pacman's nodes is the v value from the child node that attains the maximum u value since, during Pacman's turn, he determines the action that is taken.

The u value of the ghost nodes is the minimum of the u values of the immediate children nodes since Pacman believes Casper would choose the action that minimizes his utility. The v value of the Casper nodes is the maximum of the v values of the immediate children nodes since, during his turn, she chooses the action that maximizes Pacman's utility.

The value of this game, where the goal is to act optimally given the limited information, is 8. Notice that the u values are minimax values since Pacman believes he is playing a minimax game.

2. In the game tree above, put an 'X' on the branches that can be pruned and do not need to be explored when **Casper** computes the value of the tree. Assume that children are visited left-to-right.



Two branches can be pruned and they are marked on the tree above. Branches coming down from Pacman's nodes can never be pruned since the v value from one of the children nodes might be needed by the ghost node above Pacman's, even if the u value is no longer needed. For instance, if the game was simply minimax, the branch between the nodes with values (4,8) would have been pruned. However, notice that in the modified game, the value 8 needed to be passed up the tree.

On the other hand, branches coming down from the ghost nodes can be pruned if we can rule out that in the previous turn Pacman would pick the action leading to this node. For instance, the branch above the leave with values (7,7) can be pruned since Pacman's best u value on path to root is 4 by the time this branch is reached, but the ghost node already explored a subtree with a u value of 1.

3. What would the value of the entire game tree be if Pacman knew that Casper is friendly?

9. It would be the max over the entire tree.

4 Search (Algorithms & Properties)

1. When can we guarantee completeness and optimality (if ever) for each of the following search algorithms we've seen in class? For each algorithm, indicate under what conditions it is complete and/or optimal.

Algorithm	**Complete**	**Optimal**
Breadth-First		
Depth-First		
Iterative Deepening		
A*		

Algorithm	**Complete**	**Optimal**
Breadth-First	Always	When all edge costs are equal
Depth-First	Never	Never
Iterative Deepening	Always	When all edge costs are equal
A*	Always	When an admissible (trees) or consistent (graphs) heuristic is used

2. Consider a dynamic A* search where after running A* graph search and finding an optimal path from start to goal, the cost of one of the edges $X \rightarrow Y$ in the graph changes. Instead of re-running the entire

search, you want to find a more efficient way of returning the optimal path for this new search problem.

For each of the following changes, describe how the optimal path cost would change (if at all). If the optimal path itself changes, describe how to find the new optimal path. Denote c as the original cost of $X \to Y$, and assume $n > 0$.

(a) c is increased by n, $X \to Y$ is on the optimal path, and was explored by the initial search.

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 Y .

(b) c is decreased by n, $X \to Y$ is on the optimal path, and was explored by the initial search.

The original optimal paths 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 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) c is increased by n, $X \to Y$ is not on the optimal path, and was explored by the initial search.

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) c is decreased by n, $X \to Y$ is not on the optimal path, and was explored by the initial search.

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 Y .

(e) c is increased by n, $X \to Y$ is not on the optimal path, and was not explored by the initial search.

This is the same as part (c).

(f) c is decreased by n, $X \to Y$ is not on the optimal path, and was not explored by the initial search.

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.

5 Agents and Environments

Consider the following 3 agent classes:

```
class A:
    def act(percept):
        return fA()
```

```
class B:
    def act(percept):
        return fB(percept)
```

```
class C:
    percepts, initially []
    def act (percept):
       push(percepts, percept)
       return fC(percepts)
```

In each of the agents, the function f is some arbitrary, possibly randomized, function of its inputs with no internal state of its own.

1. For each of the following sets of properties, list the agents for which there exists some f such that the agent is perfectly rational in every possible environment with those properties.

   (a) Fully observable, deterministic, discrete, single-agent, static

   B, C. For a fully observable environment, only the current percept is required for an optimal decision. Because the environment is static, computation is not an issue. Note that Agent A cannot make optimal decisions because it always makes the same decision (or samples a decision from the same probability distribution), having no internal state.

   (b) Partially observable, deterministic, discrete, single-agent, static

   C. Agent B, the reflex agent, cannot always function optimally in a partially observable environment.

   (c) Partially observable, stochastic, discrete, single-agent, static

   None of the agents can be optimal for an arbitrary dynamic environment, because we can make the environment complex enough to render optimal decisions infeasible for any finite-speed machine.

2. **True/False:** There exists an environment in which every agent is rational. Describe such an environment, or explain why it cannot exist.

   True. There are many possible examples; perhaps the easiest is the environment whose performance measure is always 0; also environments with only one available action per state; etc.