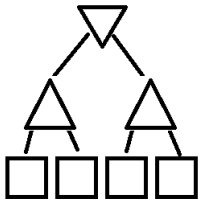
1 Adversarial Search (Minimax+Expectimax Pruning)

1. Consider the following generic tree, where the triangle pointing down is a minimizer, the triangles pointing up are maximizers, and the square leaf nodes are terminal states with some value that has not been assigned yet:



For each leaf node, is it possible for that leaf node to be the leftmost leaf node pruned? If yes, give an example of terminal values that would cause that leaf to be pruned. If not, explain. How might your answers change if the total range of the values were fixed and known?

Solution (from left to right):

The first node can never be pruned with no other information. Using alpha-beta reasoning, alpha and beta are still -inf and +inf, meaning there's no chance of pruning.

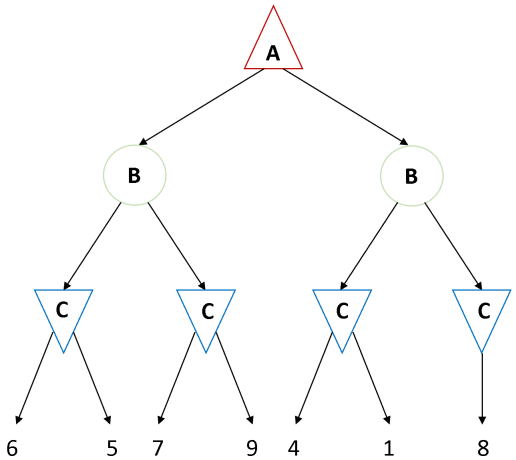
The second node also can never be pruned; no value of the first node can ever be greater than the infinity in the beta node to cause pruning of this node.

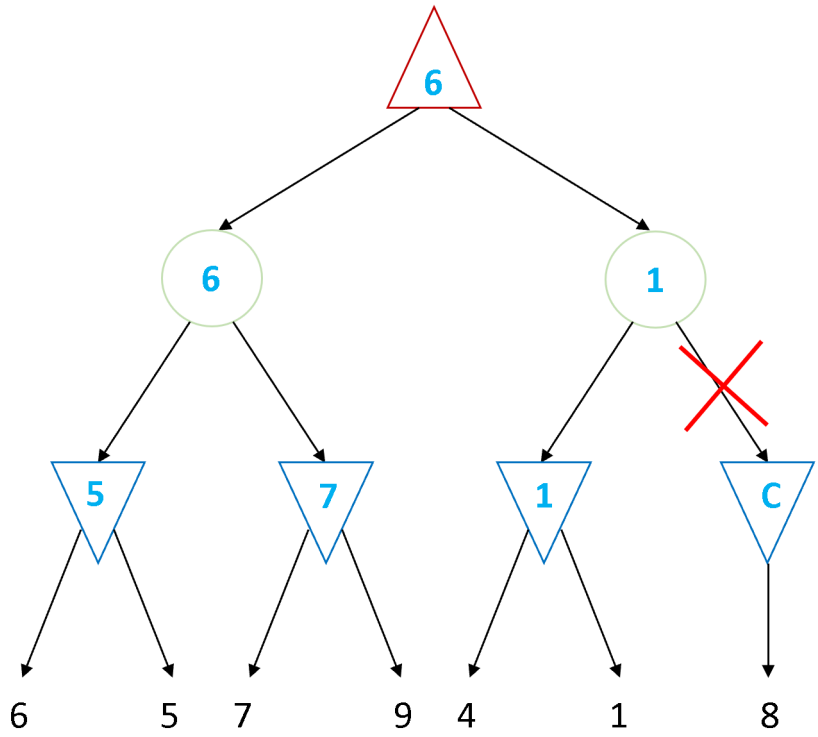
The third node cannot be the first pruned. Intuitively, if the third node were the first pruned, that would mean that the entire right tree would be unexplored.

The fourth node can be pruned. If the values of the first three leaf nodes from left to right were 1, 1, 9, we know that minimizer will not choose the right path, so this leaf is pruned. Maybe trace through why this is!

If the range were fixed, we would find that the first node still can never be pruned, the second node might be (if the first node had value = max), the third node can be (if both the first two nodes had value = min), and the fourth node can be for the same reasoning as above.

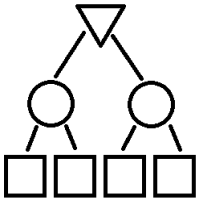
2. Recall that a chance node has the expected value of its children, and let each child have an equal probability of being chosen.
Perform pruning on the following game tree and fill in the values at letter nodes. A is a maximizer, B is a chance node, and C is a minimizer. Assume that values can only be in the range 0-9 (inclusive).





Solution:

3. Now repeat the exercise from question 1 with the following tree, where the maximizer nodes have been replaced with expectimax nodes. Assume the range of values is unknown. How would your answer change if the range of values were fixed and known (say, between 0-9)?



Solution:

With no knowledge of range, no node can be pruned. At any given state, in finding the value, we are always faced with the possibility of finding a node several orders of magnitude smaller than those seen before.

With some knowledge of a fixed range (let's say 0-9 for concreteness), this does change our answer a bit. We are still not able to prune the first two leaf nodes, but the third and fourth are able to be pruned.

If the first two nodes were both 0, then there is no need to go to the next branch. The minimizer is already doing its best!

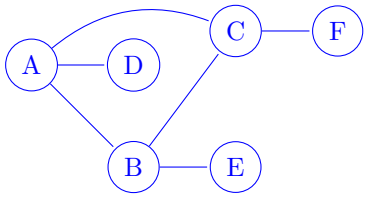
If the first two nodes were 3, and the value of the third node was 9, we can prune the fourth leaf node. There is no way for the fourth node to cause the value of the random node to be less than 3. We can thus prune.

2 CSPs

You've generously saved a row of 6 seats in Rashid for 6 of your 15-281 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 (for now, ignore any alldiff constraints):

- 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 running AC-3 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 via AC-3. Then some value is assigned to B. Which domains will and will not 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 A's domain. This means that D's domain won't change because the only arc that might cause a change, $D \to A$ will never be enforced. However, the domains of C, E and F do change.

[1] not correlated with their grades

6. Suppose the actual constraints of this CSP are as follows:

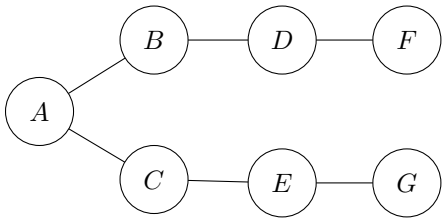
   Both C and E want to sit next to B.
   A wants to sit next to D, but not next to B or C.
   F and C had a falling out over whether AI or blockchain was cooler, so there needs to be at least 2 seats between them.

   B gets to class first and sits down in seat 3. Run AC-3 to determine the final seating arrangement (more space on next page to work it out).

   In order: F, E, B, C, D, A. There may be alternate solutions.

7. You are now trying a brand new algorithm to solving CSPs by enforcing arc consistency via AC-3 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. For a single variable with d values remaining, it is possible to backtrack $d-1$ times in the worst case. For the following constraint graph, assume each variable has a 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:

   $3(d-1)$

   Note that the given constraint graph represents a tree-structured CSP. Therefore, any order of assignments which assigns variables from root to leaves in some topological ordering of the nodes guarantees that we do not have to backtrack given that root-to-leaf arcs are consistent.
   Note: As we didn't cover tree-structured CSPs in lecture, they will not be on the exam.

   ABCDEFG and GECABDF are both orderings which satisfy the above description, which means we essentially initially enforce arc consistency, then assign values to nodes from root to leaves without any backtracking.

   GFEDBCA is not such an 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 $d-1$ 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 $3(d-1)$ times.

3 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 and non-negative
Depth-First	If we are performing graph search	Never
Iterative Deepening	Always	When all edge costs are equal and non-negative
A*	Always	When an admissible (tree search) or consistent (graph search) heuristic is used

2. Consider a dynamic A* search in which after running A* graph search and finding an optimal path from start to goal (assuming there's only one goal), the cost of one of the edges $X \to Y$ 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 X was explored by the initial search.

The optimal path could change if the original cost is no longer the cheapest when adding the amount of n.
We would re-explore all previously expanded nodes with the new edge cost of X → Y. If Y is explored, we end the search for all paths that will end at Y in the previous search add the goal node together with the previous optimal path cost of Y → Goal to the frontier, then we keep searching down until we pop the goal node out of the frontier. This means that you are re-exploring every path that was originally blocked by a path that included the edge X → Y. If the cheapest path to Y is discovered, then we already know the optimal path and path cost of $Y \to$ Goal

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

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 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 X 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 X was explored by the initial search.

We would put previously expanded node (with their path and path cost) on the frontier and search for X, as well as the optimal path. If X is expanded this time, we clear all paths on the frontier except for the original optimal path and the cheapest path leading to X plus the edge $X \to Y$. Then we do A* search with this new frontier.

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 frontier. 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 optimal path leading to X must be found 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 the edge weights c can't go negative.)

Assuming that the cost of $X \to Y$ remains non-negative, 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.

4 Local Search

4.1 Warm Up

1. Define completeness and optimality for local search problems.

   A local search algorithm is complete if it always finds a goal if one exists, and optimal if it always finds a global minimum/maximum.

2. What are two advantages of local search?

   Two advantages of local search algorithms are that they don't use very much memory and can potentially be used to find reasonable solutions to large scale problems.

3. What are two disadvantages of local search techniques?

   Two disadvantages of local search algorithms are that they typically do not store the path to the goals state and cannot keep track of multiple paths to the goal state since they do not maintain a search tree.

4. What are four functions used in genetic algorithms?

   The four functions used in genetic algorithms (that we would need to define) are the fitness function, the selection function, the crossover function, and the mutation function.

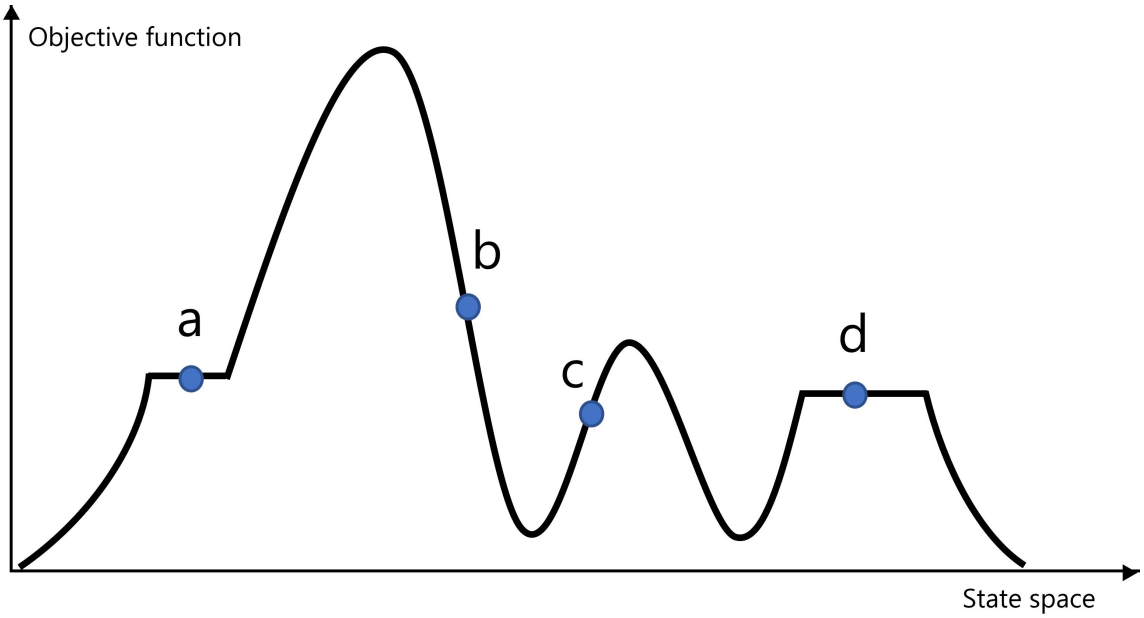
5. What two local search techniques does simulated annealing combine?

   Simulated annealing combines the local search techniques of random walk and hill climbing.

6. What is one disadvantage of beam search and how can we resolve it?

   One disadvantage of beam search is that the k states we maintain may lack sufficient diversity since the states chosen by beam search may quickly end up concentrated in a small region of the state space. We can resolve this by using stochastic beam search and randomly choosing which k offspring to maintain based on their fitness.

4.2 Practice



Consider how each of the following searches performs in state space above. Recall that in the context of local search, our goal is to find the state that optimizes the objective function.

1. Hill-climbing search with start state c

   Does it terminate? If so, where?

   Does it find the global maximum?

   It terminates at the local maximum (to the right of c). It is not complete.

2. Random-restart hill climbing with randomly generated initial states a, d, then b

   Does it terminate? If so, where?

   Does it find the global maximum?

   Random-restart hill climbing conducts a series of searches. For start states a and d, the search will terminate immediately since there are no better neighboring states. (Then it will randomly restart at some random start state.)
   For start state b, the search will reach the global maximum, so it is complete. In general, random-restart hill climbing is "trivially complete with probability approaching 1, because it will eventually generate a goal state as the initial state." (AIMA Chapter 4, page 124)

3. Stochastic hill-climbing that allows sideways moves with start state d

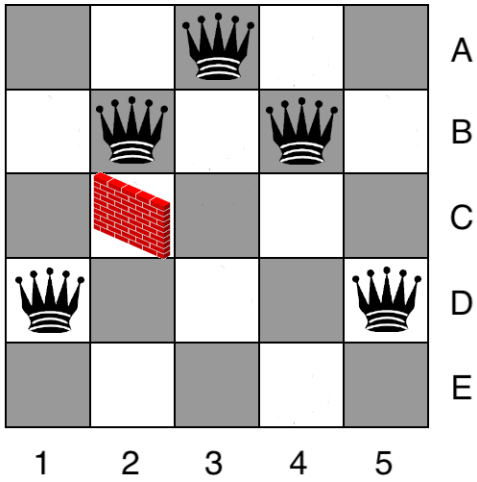
   Does it terminate? If so, where?

   Does it find the global maximum?

This search will not terminate. Since we allow sideways movement and d is on a flat local maximum, stochastic hill-climbing will choose randomly from one of its neighboring states with the same objective function value. This will result in an endless loop. It is not complete.

4.3 5-QUEENS Revisited

Now, let's revisit the 5-QUEENS problem. Our goal is to try to place 5 queens on a 5x5 chessboard with no conflict between any two queens. However, this time, there is a twist: our chessboard has a wall!



If the wall blocks the path between two queens (horizontal, vertical, or diagonal) then those two queens will not be in conflict. The wall cannot be moved, and a queen cannot pass through the wall. Below is some pseudocode to try to solve the 5-QUEENS problem with hill climbing.

```
function 5-QUEENS-HILL-CLIMBING(problem)
    current_state = problem.INITIAL-STATE
    loop do
        neighbor = state with lowest # of conflicts between queens obtained by making
            one legal chess move from current_state
        if neighbor.num_conflicts >= current_state.num_conflicts
            return current_state
        current_state = neighbor
```

Using the given starting state, pseudocode, and the rules regarding walls, answer the following questions.

1. Run 5-QUEENS-HILL-CLIMBING on the given start state until completion. Where do the queens end up? Is this a goal state?

   The queens will end up at D1, B2, E3, A4, and D5. This is not a goal state because there is a conflict between D1 and D5.
   Our starting state has 4 conflicts. If we move A3 to E3, we see that the new state only has 2 conflicts, which is the lowest number of conflicts among all the neighbors. From here, we can move B4 to A4 to reduce the number of conflicts to 1. This is no conflict between this new position and D1 due to the brick wall. From here, we cannot take any steps to reduce the number of conflicts.

2. If we remove the brick wall, will 5-QUEENS-HILL-CLIMBING end up in a global optima?

   If we remove the brick wall, 5-QUEENS-HILL-CLIMBING will not end up in a global optima. It will be at a state whose value is not less than its neighbors and is not a global optima. The state and all its neighboring states will have 2 or more conflicts.

5 Linear Programming

1. True/False

   (a) The cost vector points in the direction of decreasing cost.

   False - The cost vector points in the direction of increasing cost.

   (b) As the magnitude of c increases, the distance between the contour lines of the objective $c^T x$ increases as well.

   False - an increase in the magnitude of c implies that a lesser distance needs to be traversed in order to incur the same increase in cost. This means that the distance between contour lines actually decreases.

   (c) A maximizing LP with exactly one constraint will always have a maximum objective value of ∞.

   False - If the cost vector is perpendicular to the one constraint, then all the points on the constraint will have an equal, maximum, and bounded (i.e., not ∞) objective value.

   (d) The Simplex algorithm will always visit strictly fewer vertices in a graph than using vertex enumeration.

   False - In the worst case, Simplex may end up visiting all the intersections between constraints in a graph. If we start off at a point with a high objective value and each neighboring vertex is better than the last, we end up visiting all of them.

2. The 281 TAs are throwing a big Valentine's day party for all the students. They will have a chocolate fountain and lots of fruit punch but they need help deciding how much of each to buy. Assume they can buy any fraction of an ounce. The staff has a budget of \$100. An ounce of chocolate is \$1 while an ounce of fruit punch is \$0.50. The staff wants to make sure there's enough for all the students so they need at least a total of 80 ounces of chocolate and fruit punch combined. Lastly, the 281 locker is small, and can only hold up to 120 ounces of items. Chocolate brings 3 units of happiness per ounce and fruit punch brings 1 unit of happiness per ounce, and the TAs want to maximize student happiness.

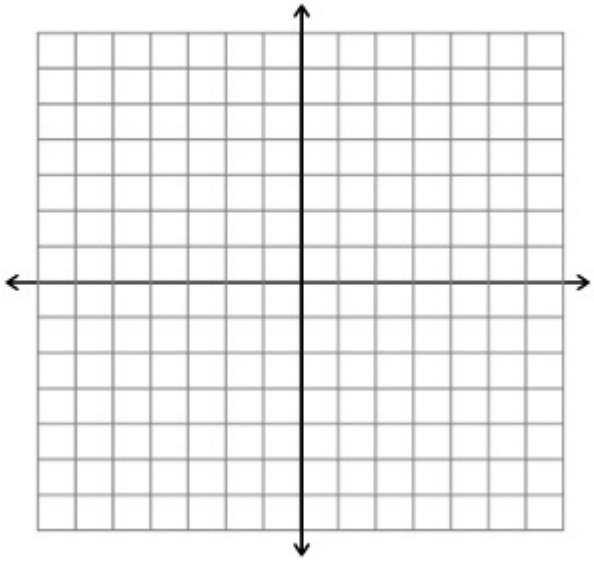
   (a) What are the variables needed to formulate this as a linear programming problem?

   x_1 is ounces of chocolate and x_2 is ounces of fruit punch

   (b) What are necessary constraints needed to formulate this as a linear programming problem?

   $x_1 + 0.5x_2 \leq 100$
   $x_1 + x_2 \geq 80$
   $x_1 + x_2 \leq 120$
   $x_1 \geq 0$
   $x_2 \geq 0$

   (c) Plot this linear optimization problem in the graph below and find the optimal solution. Be sure to draw in the cost vector.



The optimal solution would be to buy 100 ounces of chocolate and 0 ounces of fruit punch. The graph and feasible region should look something like the following.