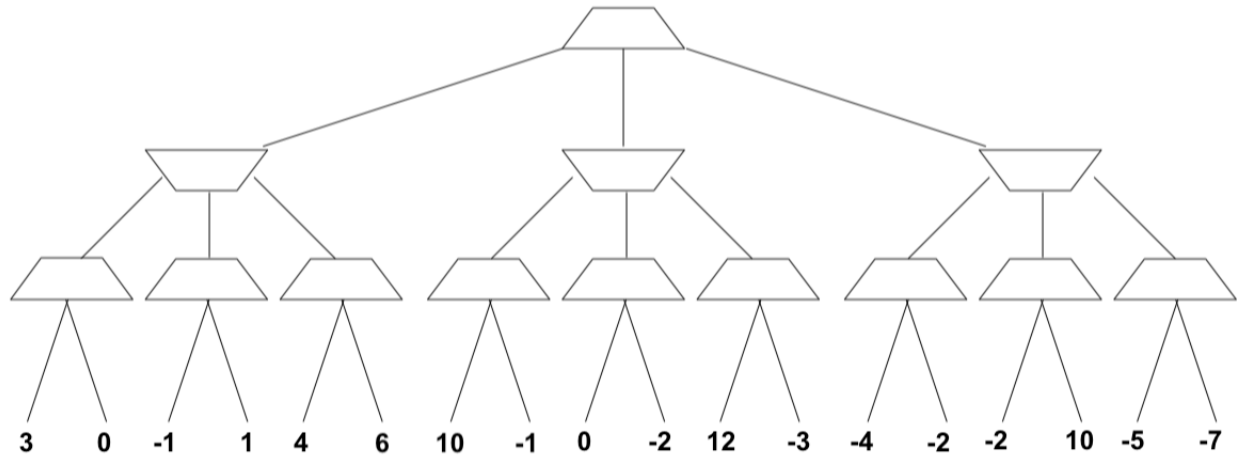
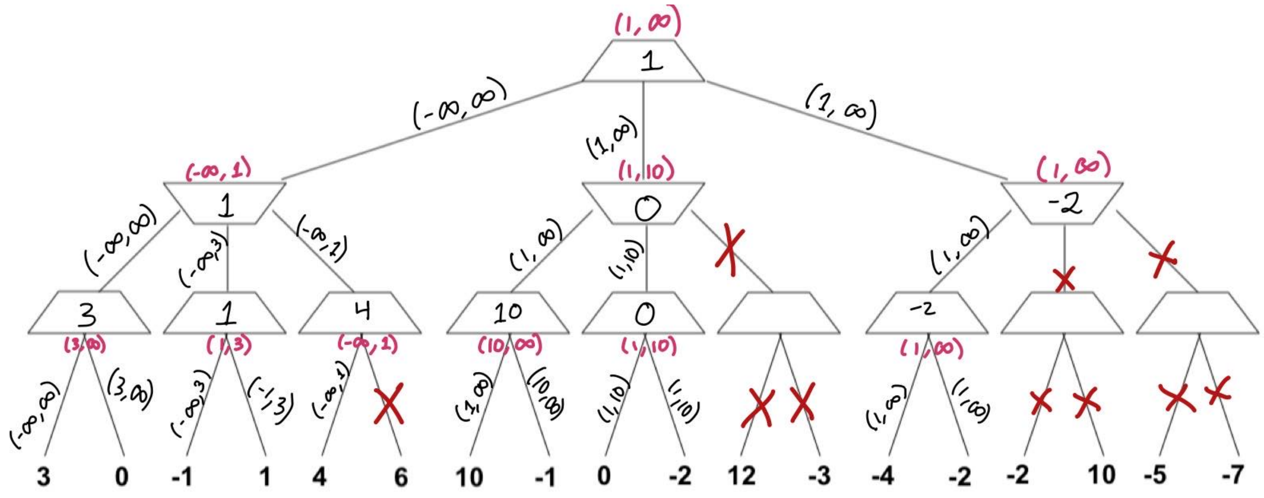
1 Adversarial Search

Consider the following game tree, where the root node is a maximizer. Using alpha beta pruning and visiting successors from left to right, record the values of alpha and beta at each node. Furthermore, write the value being returned at each node inside the trapezoid. Put an 'X' through the edges that are pruned off.



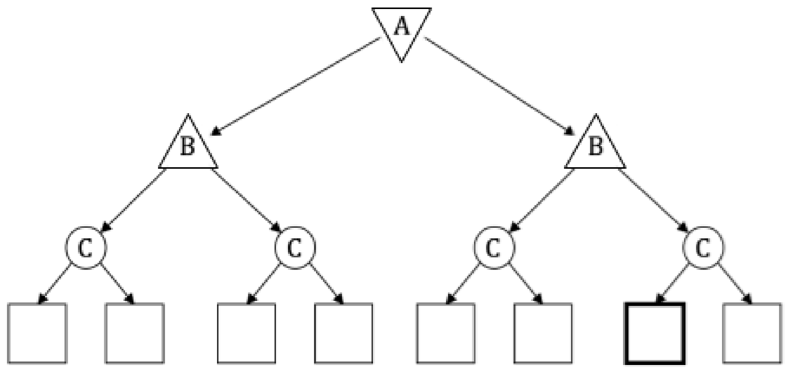
The following picture shows the final (α, β) values at each node in magenta, as well as the (α, β) values passed down upon exploring a subtree in black.

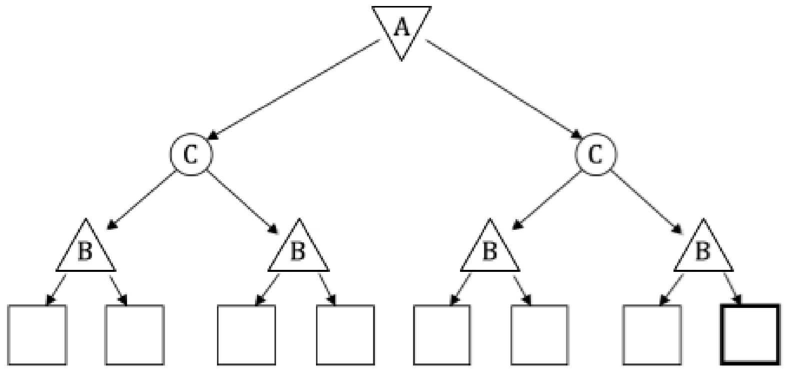


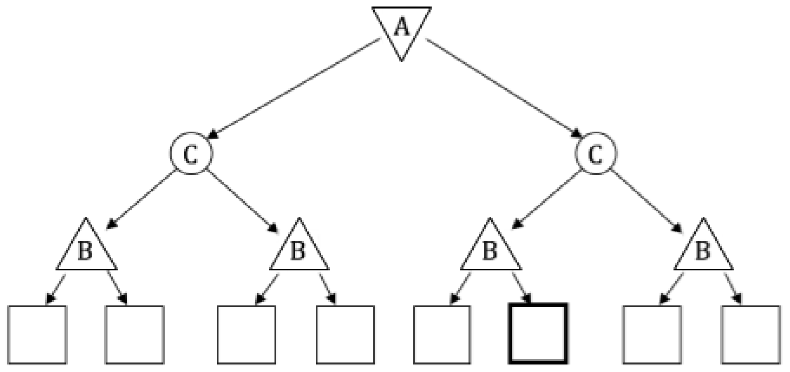
2 Alpha Beta Expectimax

In this question, player A is a minimizer, player B is a maximizer, and C represents a chance node. All children of a chance node are equally likely. Consider a game tree with players A, B, and C. In lecture, we considered how to prune a minimax game tree - in this question, you will consider how to prune an expectimax game tree (like a minimax game tree but with chance nodes). Assume that the children of a node are visited left to right.

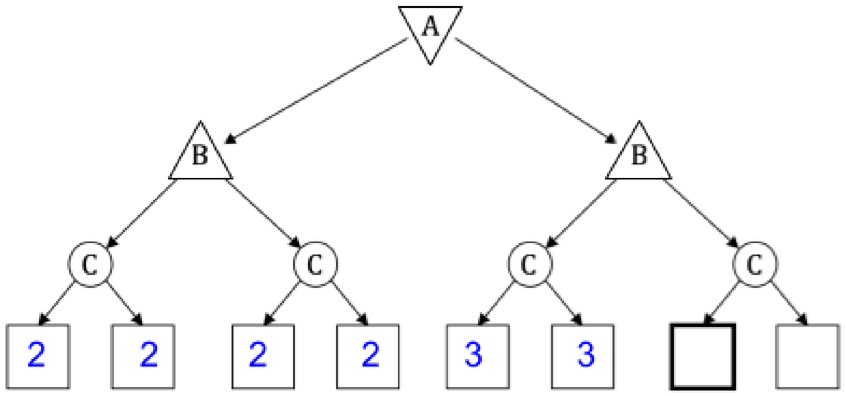
For each of the following game trees, give an assignment of terminal values to the leaf nodes such that the bolded node can be pruned (it doesn't matter if you prune more nodes), or write "not possible" if no such assignment exists. You may give an assignment where an ancestor of the bolded node is pruned (since then the bolded node will never be visited). You should not prune on equality, and your terminal values must be finite.

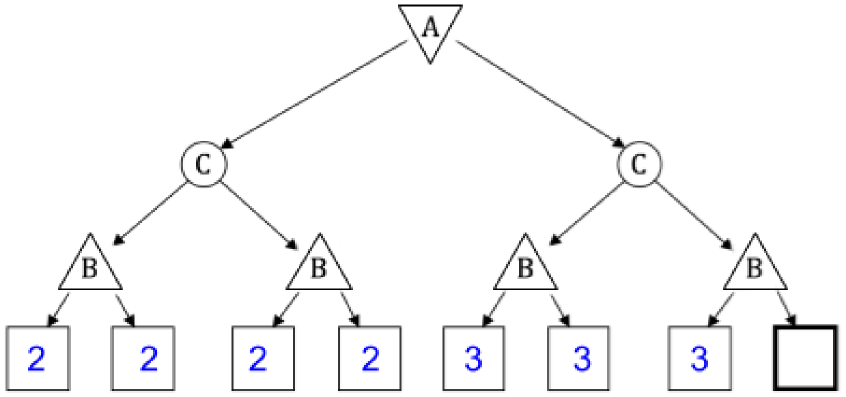


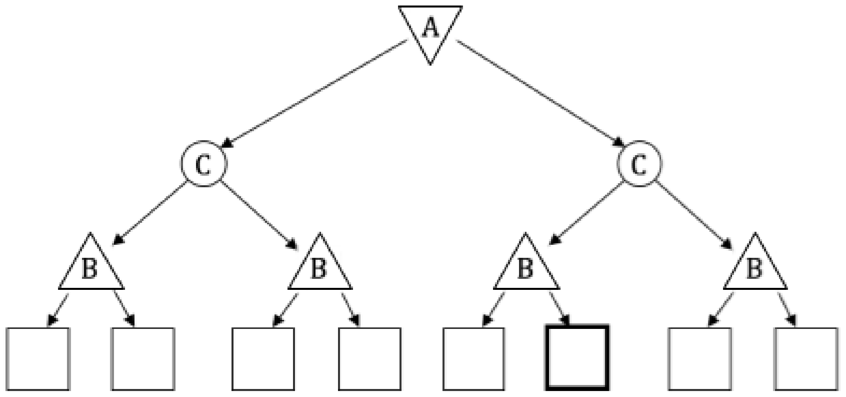




Answers may vary.



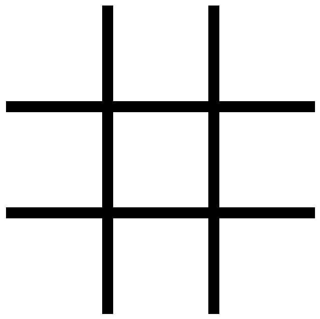


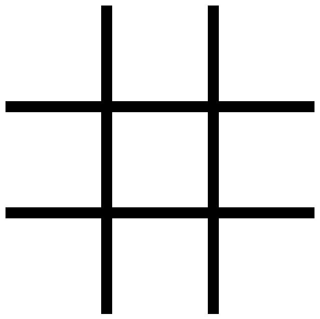


Not possible. At the bolded node, the minimizer can guarantee a score of at most the value of its left subtree. However, since we have not visited all the children of C, there is no bound on the value that C could attain. So, we need to continue exploring nodes until we can put a bound on C's value, which means we must explore the bolded node.

3 XOXO AI Girl [1]

Today, we will be implementing Tic Tac Toe using the Alpha Beta strategy that we have learned in class[2]. Tic Tac Toe is very hard to win when two masters play and often leads to ties which is boring and therefore we will use this strategy to help us become the best players in the world and play as optimally as we can to maximize our chances of winning[3].
To start, feel free to play a couple games of Tic Tac Toe with your neighbor and think of some strategies that you have when you play it!





(a) What are some strategies that you use when you play Tic Tac Toe?

Answers may vary, some examples include: playing center first, playing corners first, etc.

(b) We will now develop an AI to play tic tac toe against the user. Download the starter code from:

https://www.cs.cmu.edu/~15281/recitations/rec3/tictactoe.zip

In this program, a game state is represented as a list of length 9, representing each of the 9 grids from left to right and top to bottom. The list item is either 0 representing an empty space, an "X" which means the user played a piece in that space, or an "O" which means the computer played a piece in that space.

Run the program with the default Tic Tac Toe AI using:

```
python3.6 tictactoe.py
```

This AI uses `chooseFirstAvailable()` to choose a move. Did you win? Think about why this AI is easy to beat.

(c) Look at the function `admin(state,alpha,beta)` which is our implementation of the alpha-beta pruning min function. Complete the corresponding `abmax` function which takes in a game state `state` as well as the `alpha` and `beta` values.

(d) Run tic-tac-toe with the new Alpha-Beta AI that you created. You will need to modify `getComputerAction()` to run your Alpha-Beta code. Did you win? Think about why that is the case.

[1]Harlene came up with the title. If you have some free time, please watch Gossip Girl on Netflix.
[2]For those of you who don't know how to play, please visit this link to learn the rules: https://en.wikipedia.org/wiki/Tic-tac-toe.
[3]Tina claims she really sucks at this game, so if you are in her recitation, please challenge her to a game as it will help boost your confidence.

(e) Alpha-beta pruning cuts down on the number of nodes explored. Can you further reduce the search space?

We can use rotational and reflectional symmetry to remove redundancy in many similar states of the tic-tac-toe board.

(f) Write an evaluation function in `evaluationFunction()` to determine the value of a game state. Recall that the computer is the maximizer.

(g) Now, run tic-tac-toe again with the Alpha-Beta AI that calls your evaluation function. Did you win? The evaluation function is what makes your AI smart. Can you write an evaluation function that is really hard to beat?

4 CSP: Air Traffic Control

We have five planes: A, B, C, D, and E and two runways: international and domestic. We would like to schedule a time slot and runway for each aircraft to either land or take off. We have four time slots: 1, 2, 3, 4 for each runway, during which we can schedule a landing or take off of a plane. We must find an assignment that meets the following constraints:

- Plane B has lost an engine and must land in time slot 1.
- Plane D can only arrive at the airport to land during or after time slot 3.
- Plane A is running low on fuel but can last until at most time slot 2.
- Plane D must land before plane C takes off, because some passengers must transfer from D to C.
- No two aircrafts can reserve the same time slot for the same runway.

(a) Complete the formulation of this problem as a CSP in terms of variables, domains, and constraints (both unary and binary). Constraints should be expressed implicitly using mathematical or logical notation rather than with words. Make sure to specify variables, domains, and constraints.

Variables: A, B, C, D, E for each plane.
Domains: a tuple (*runway type, time slot*) for runway type $\in$ {international, domestic} and time slot $\in \{1, 2, 3, 4\}$.
Constraints:

$$B[1] = 1$$

$$D[1] \geq 3$$

$$A[1] \leq 2$$

$$D[1] < C[1]$$

$$A \neq B \neq C \neq D \neq E$$

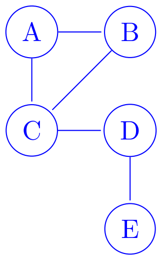
Note here we use B[1] to denote the second value of the tuple assigned to variable B. the time slot value, which is a number in $\{1, 2, 3, 4\}$

For the following parts, we add the following two constraints:

- Planes A, B, and C cater to international flights and can only use the international runway.
- Planes D and E cater to domestic flights and can only use the domestic runway.

(b) The addition of the two constraints above alters the CSP. Specifically, the domain does not need to include the runway type since this information is carried by the variable, and the binary constraints have changed. Determine the new domain and draw the constraint graph for this problem given the original constraints and the two added ones.

Variables: A, B, C, D, E for each plane.
Domain: $\{1, 2, 3, 4\}$
Constraint Graph:



Explanation of Constraints Graph: We can now encode the runway information into the identity of the variable, since each runway has more than enough time slots for the planes it serves. We represent the non-colliding time slot constraint as a binary constraint between the planes that use the same runways.

(c) What are the domains of the variables after running AC-3? Begin by enforcing unary constraints. (Cross out values that are no longer in the domain.)

List of (variable, assignment) pairs:

A	1	2	3	4
B	1	2	3	4
C	1	2	3	4
D	1	2	3	4
E	1	2	3	4

Enforcing arc consistency with AC-3, we have the following domain as a result:

A	~~1~~	2	~~3~~	~~4~~
B	1	~~2~~	~~3~~	~~4~~
C	~~1~~	~~2~~	~~3~~	4
D	~~1~~	~~2~~	3	~~4~~
E	1	2	~~3~~	4

(explanation of process below)
Enforcing unary constraints (in an arbitrary order) first,

1. We cross out 2, 3, 4 from B's domain, adding arcs A → B and C → B to the queue.

2. We cross out 3, 4 from A's domain, adding arcs B → A and C → A to the queue.

3. We cross out 1, 2 from D's domain, adding arcs C → D and E → D to the queue.

Enforcing A → B, we cross out 1 from A's domain; add arcs B → A and C → A to the queue.
Enforcing C → B, we cross out 1 from C's domain; add arcs A → C, B → C, and D → C to the queue.
Enforcing B → A, no domain changes are necessary (all values remaining in B's domain have a consistent corresponding value in A's domain); no arcs are added.
Enforcing C → A, we cross out 2 from C's domain; add arcs A → C, B → C, and D → C to the queue.
Enforcing C → D, we cross out 3 from C's domain; add arcs A → C, B → C, and D → C to the queue.
Enforcing E → D, no domain changes are necessary.
Enforcing B → A, no domain changes are necessary.
Enforcing C → A, no domain changes are necessary.
Enforcing A → C, no domain changes are necessary.
Enforcing B → C, no domain changes are necessary.
Enforcing D → C, we cross out 4 from D's domain (there is no c in C's domain such that $c > 4$); add arcs C → D and E → D to the queue.
Enforcing A → C, no domain changes are necessary.
Enforcing B → C, no domain changes are necessary.
Enforcing D → C, no domain changes are necessary.
Enforcing C → D, no domain changes are necessary.
Enforcing E → D, we cross out 3 from E's domain; add arc D → E to the queue.
Enforcing D → E, no domain changes are necessary.
(phew!)

Note: For a general binary CSP, to enforce arc consistency before assigning any variables, you should add all arcs to the initial queue. For this problem, it can be easily seen that if there are no unary constraints, all the arcs will be consistent before any variable is assigned a value. As a result, we can start with the unary constraints and add arcs only for the related variables after enforcing the unary constraints.

(d) AC-3 can be rather expensive to enforce, and we believe that we can obtain faster solutions using only forward-checking on our variable assignments. Using the Minimum Remaining Values heuristic, perform backtracking search on the graph, breaking ties by picking lower values and characters first. List the (variable, assignment) pairs in the order they occur (including the assignments that are reverted upon reaching a dead end). Enforce unary constraints before starting the search.

List of (variable, assignment) pairs:

(You don't have to use this table)

A	1	2	3	4
B	1	2	3	4
C	1	2	3	4
D	1	2	3	4
E	1	2	3	4

Answer: (B, 1), (A, 2), (C, 3), (C, 4), (D, 3), (E, 1)

5 Discussion Questions

(a) What is the difference between Forward Checking and AC-3?

Forward checking and AC-3 both enforce arc consistency, but forward checking is more limited. Whenever a variable X is assigned, forward checking enforces arc consistency only for arcs that are pointing to X, which will reduce the domains of the neighboring variables in the constraint graph. Forward checking stops at this point, but AC-3 will continue to enforce arc consistency on neighboring arcs until there are no more variables whose domain can be reduced. As a result, FC ensures arc consistency of the assigned variable and its neighbors only, while AC-3 ensures arc consistency for the whole graph.

(b) Why does a tree-structured CSP take $O(nd^2)$ to solve?

If the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time compared to general CSPs, where worst-case time is $O(d^n)$. To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree. Such an ordering is called a topological sort. Any tree with n nodes has $n-1$ arcs, so we can make this graph directed arc-consistent in $O(n)$ steps, each of which must compare up to d possible domain values for two variables, for a total time of $O(nd^2)$.

(c) Why would one use the following heuristics for CSP?

(i) Minimum Remaining Values (MRV)

MRV: "Which variable should we assign next?"

- Fail fast
- We have to assign all variables at some point, so we might as well do hard stuff first (allowing us to prune the search tree faster/realize we need to backtrack)

(ii) Least Constraining Value (LCV)

LCV: "Which value should we try next?"

- We just want one solution.
- We don't try all combinations of value, so we should try ones that are likely to lead to a solution.

(d) How would adversarial search change if the root node is a minimizer instead of a maximizer?

While we can never prune directly off the root node, whether it is minimizer or maximizer, we switch how we treat beta and alpha because we pass down the beta instead of the alpha. While the value starts at positive infinity instead of negative infinity, our update rule for v is $v = min(v, children)$, so the value of the root node can become potentially smaller.