Adversarial Search

CS 3600
Intro to Artificial Intelligence

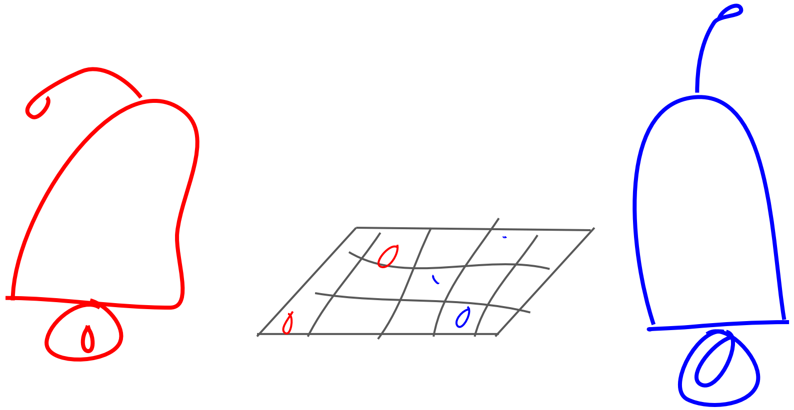
Adding in other agents

Recall, our environments were **single-agent**.

How can you plan a sequence of actions if you have no control over what other agents will do?

Simple **games** are an environment with multiple agents that keeps most of the other restrictions in place

- Checkers, Chess, Go
- Poker, Rock-paper-scissors, Blackjack
- Roulette, Backgammon, Monopoly



Two player, perfect information, zero-sum games

Two player
In general, we could have many players, but many classic games only have two.

Perfect information
All players know the exact state of the game.

Zero-sum
The result (**utility**) for each player at the end of the game is exactly the opposite of the opponent.

These assumptions let us **reason** about the goals of the other agent.

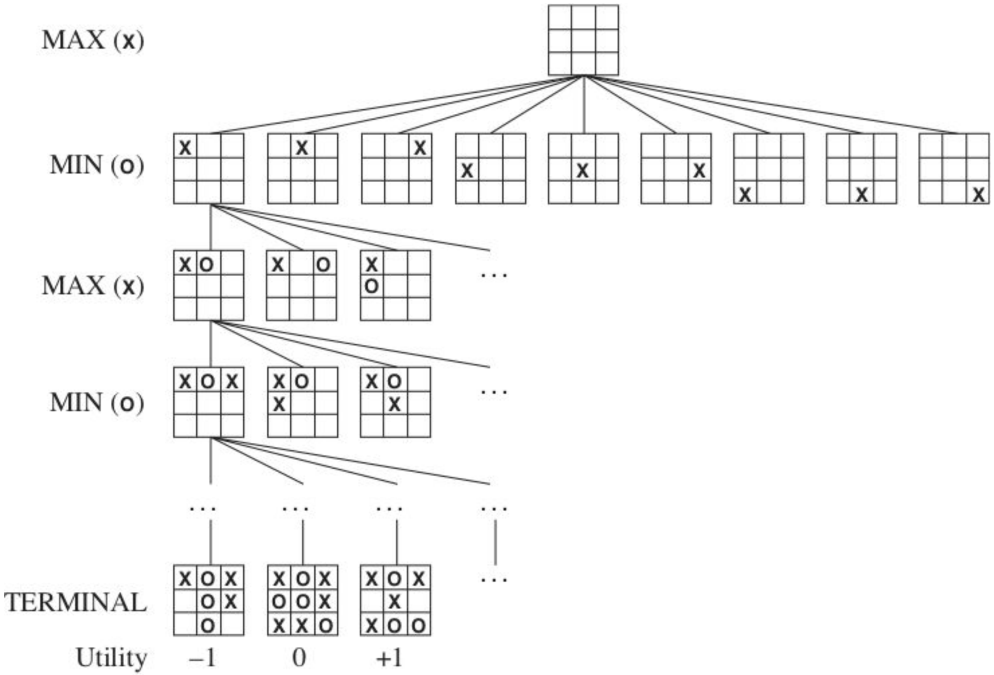
Search trees for 2-player zero-sum games (1)

We can still construct the state space as before, but the other player (**opponent**) gets to make every other move.

In tree form, we choose actions on **alternating layers**.

We are choosing actions that **maximize** our utility, opponent is choosing actions that **minimize** our utility

Idea: keep expanding nodes until we hit a **terminal** state, propagate the final utility back to initial action choice, pick the action with the best utility.

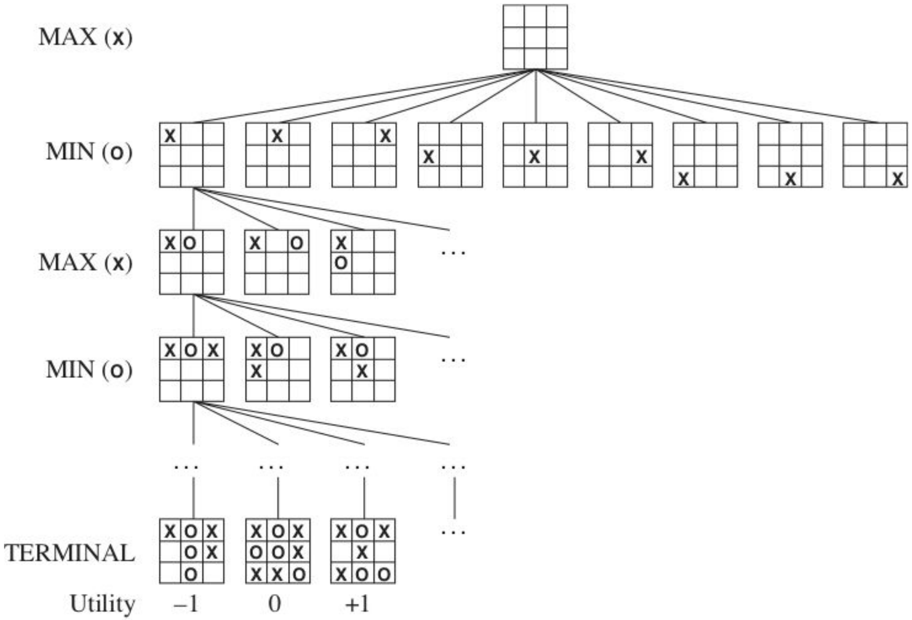


Search trees for 2-player zero-sum games (2)

Notes

- Result of searching this tree is a **single move** (have to wait for opponent's move)
- To pick the **optimal** move, we might need to expand the **entire search tree**
- We are **assuming** the opponent is **rational** (picks the best moves)

A recursive definition of the `Minimax` value of a **state**

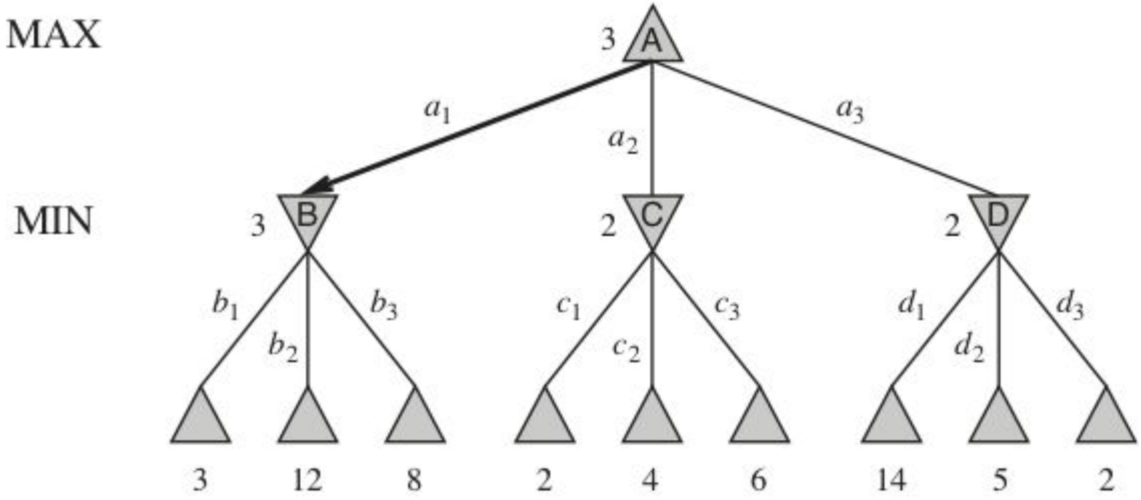


$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

A simple example

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST(s)} \\ \max_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER(s) = MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER(s) = MIN} \end{cases}$$

`Minimax`(B) = 3 (b_1)
`Minimax`(C) = 2 (c_1)
`Minimax`(D) = 2 (d_3)
`Minimax`(A) = 3 (a_1)



The Minimax algorithm

```
def Minimax_Decision(state):
     best_action,best_val = None,-math.inf
     for a in actions(state):
          s_prime = result(a,state)
          a_val = Min_Value(s_prime)
          if a_val>best_val:
               best_val = a_val
               best_action = a
     return best_action
```

```
def Max_Value(state):
     if isTerminal(state):
          return utility(state)
     v = -math.inf
     for a,s in successors(state):
          v = max(v,Min_Value(s))
     return v
```

```
def Min_Value(state):
     if isTerminal(state):
          return utility(state)
     v = math.inf
     for a,s in successors(state):
          v = min(v,Max_Value(s))
     return v
```

The Minimax algorithm - notes

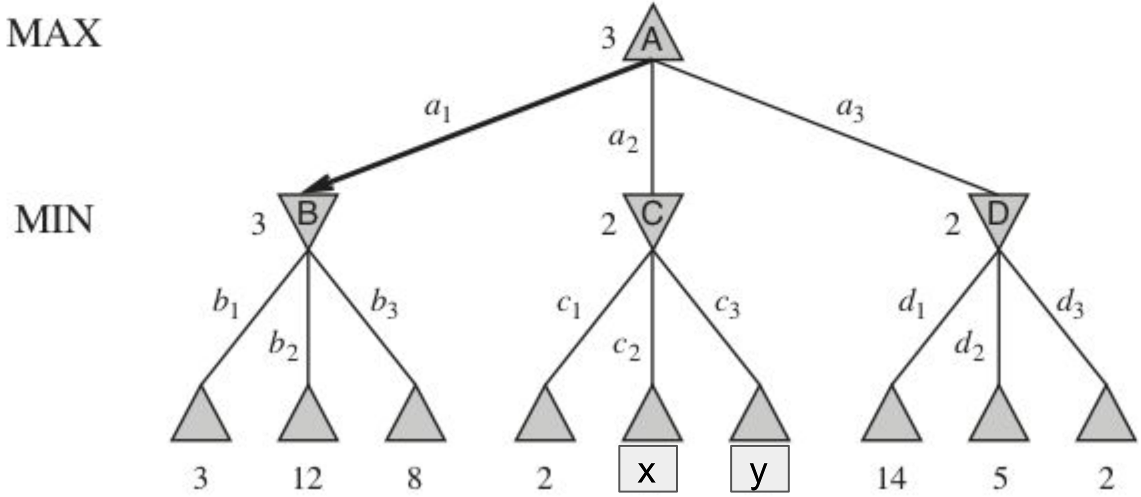
Basically, DFS!

- Complete? **Yes** (if tree is finite)
- Optimal? **Yes** (if opponent is rational)
- Time complexity? **$O(b^m)$** (b: branching factor, m: depth of tree)
- Space complexity? **$O(m*b)$**

For chess, $b \cong 35$, $m \cong 100$

Expanding the full tree isn't going to work!

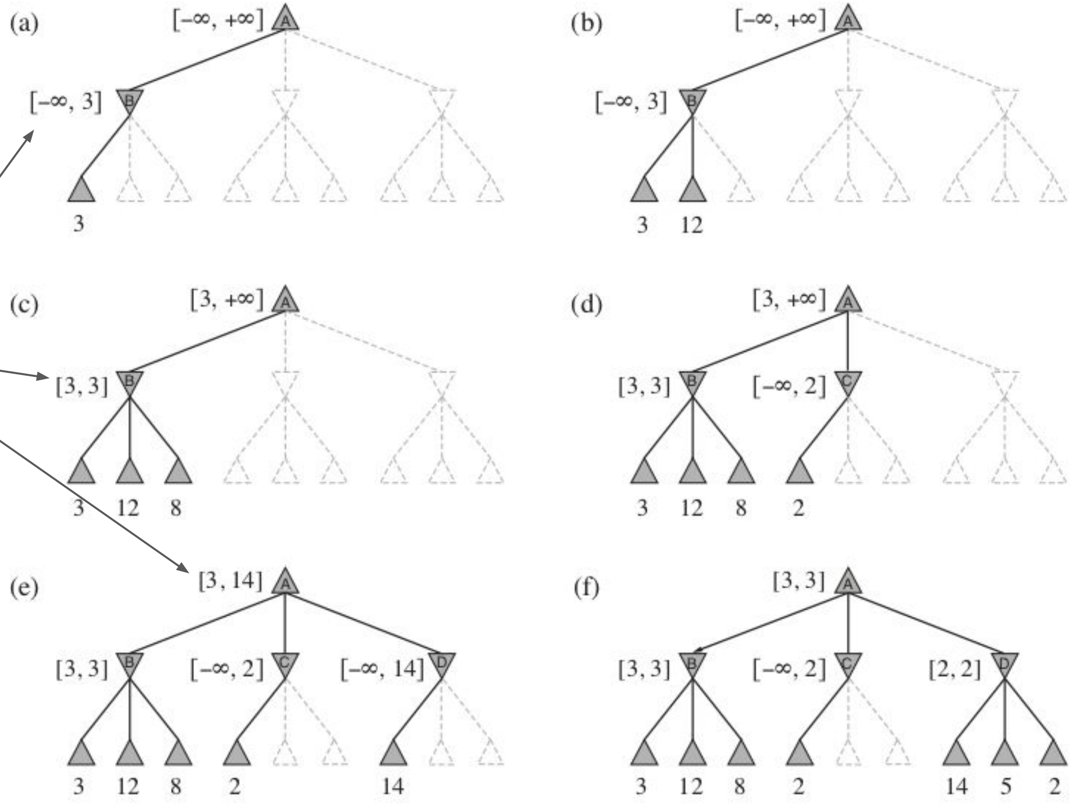
Do we always need to expand every node? (1)



$$
\begin{aligned}
\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3, z, 2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\
&= 3
\end{aligned}
$$

Do we always need to expand every node? (2)

Range of possible Minimax values



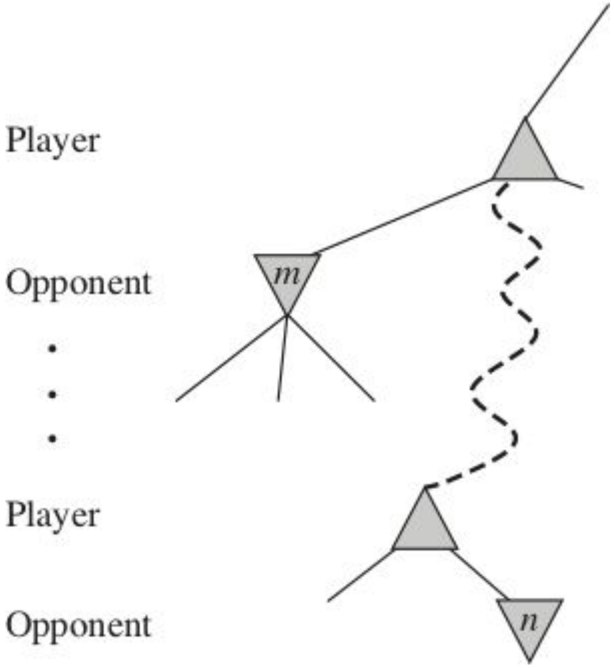
Unreachable nodes

If we can move to *m*, which has a strictly better `Minimax` value than *n*, we don't need to explore the path containing *n*.

Define the following helper variables

α = the value of the **highest value** choice we have found so far at any choice point along the path for `Max`

β = the value of the **lowest value** choice we have found so far at any choice point along the path for `Min`



The Alpha-Beta search algorithm

```
def Alpha_Beta_Search(state):
     best_action,best_val = None,-math.inf
     for a in actions(state):
          s_prime = result(a,state)
          a_val = Min_Value(s_prime,-math.inf,math.inf)
          if a_val>best_val:
               best_val = a_val
               best_action = a
     return best_action
```

```
def Max_Value(state,alpha,beta):
     if isTerminal(state):
          return utility(state)
     v = -math.inf
     for a,s in successors(state):
          v = max(v,Min_Value(s,alpha,beta))
          if v >= beta:
               return v
          alpha = max(alpha,v)
     return v
```

```
def Min_Value(state,alpha,beta):
     if isTerminal(state):
          return utility(state)
     v = math.inf
     for a,s in successors(state):
          v = min(v,Max_Value(s,alpha,beta))
          if v<= alpha:
               return v
          beta = min(beta,v)
     return v
```

Alpha-Beta notes

Pruning does not affect the optimality of the final result

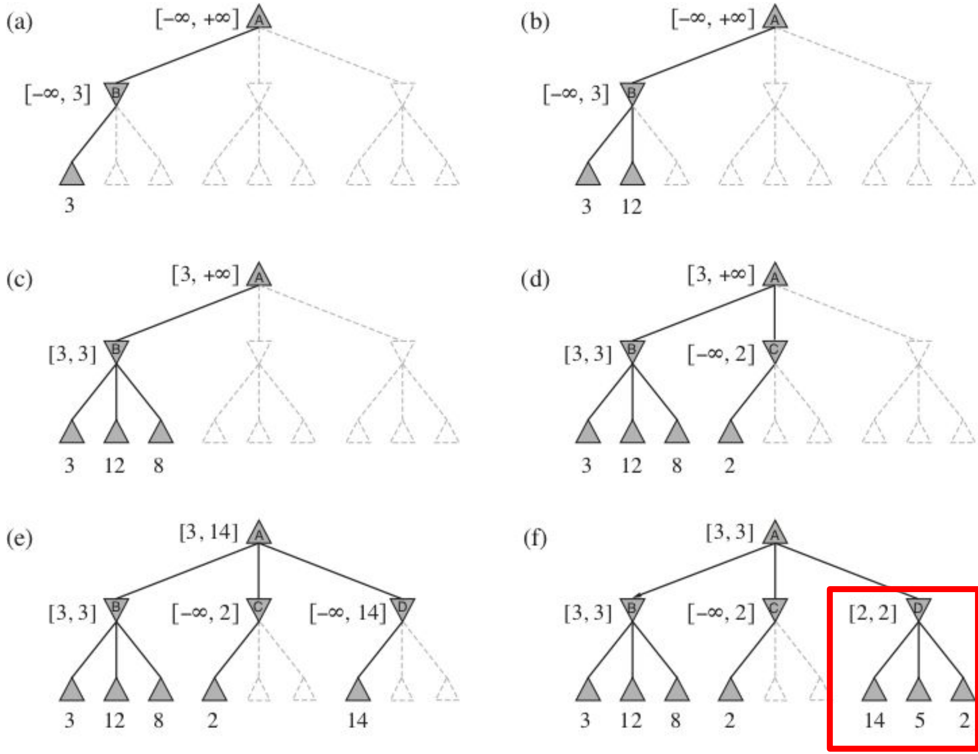
Which paths we can prune highly depends on the expansion order (just like DFS)

Worst case, same complexity as `Minimax`

With the best ordering, time complexity: $\mathbf{O(b^{m/2})}$ (effective branching factor $\sqrt{b}$)

For chess, $b \cong 6$ instead of 35

6^{100} is still pretty big



Alternatives to finding the optimal solution

- Use `Cutoff-Test` and `Eval` instead of `Terminal` and `Utility`
  - `Cutoff-Test` checks to see if terminal, or if we’ve already expanded past some depth limit
  - `Eval` provides an **estimate** of the true utility of this node without searching all the way down to the leaves. Similar to the **heuristic** from previous lectures
- Use **Iterative Deepening Search** and return the move selected by the deepest completed search (alternatively, use IDS to help with move ordering)
- Use a lookup table of previously visited states (**transposition table**, **memoization**)
  - Kind of like the closed list from previous lectures
  - Need to have a cache-like structure to keep memory usage bounded

Summary and preview

Wrapping up

- **Games** with multiple agents can be solved with similar search methods, when certain assumptions are made that let us reason about the objectives of the other agents
- `Minimax` uses an alternating, DFS-like search to pick the best **action** for the current state
- We can effectively **double** the depth of the trees we can search in a fixed time by using Alpha-Beta **pruning**

Next time

- Introducing stochasticity