Week 6: Adversarial searches & game playing

Dr. Ammar Masood

Department of Cyber Security,

Air University Islamabad

Table of Contents

- Intro to adversarial searches and game playing
- Minimax algorithm
- Alpha-beta Pruning
- GAN's/ Actor critic Problem

Intro to Adversarial Searches

In many real-world problems, we often encounter environments where multiple agents or players make decisions that influence each other's outcomes. **Adversarial search** is a specialized form of search used in such competitive scenarios, particularly in **two-player, zero-sum games** like chess, tic-tac-toe, and checkers.

What is adversarial search?

Adversarial search is a decision-making process used in **competitive environments** where an agent must account for the actions of an **opponent**. Unlike traditional search problems (such as pathfinding), adversarial search involves **two or more players** who have conflicting goals.

For example, in chess:

- **You try to win the game** by maximizing your position.
- **Your opponent tries to do the same**, minimizing your advantage.

Thus, adversarial search requires strategies that anticipate and counteract the opponent's best possible moves.

Characteristics of adversarial searches

Two or More Agents: Typically involves two players (MAX and MIN) competing.
Zero-Sum Property: A gain for one player is a loss for the other.
Turn-Based or Simultaneous Moves: Players alternate moves or act simultaneously.

Perfect vs. Imperfect Information:

Perfect Information: Players know the entire game state (e.g., chess).
Imperfect Information: Some information is hidden (e.g., poker).

Deterministic vs. Stochastic:

Deterministic: No randomness (e.g., tic-tac-toe, chess).
Stochastic: Random events influence the game (e.g., rolling dice in backgammon).

Games

- Multiagent environment
- Cooperative vs. competitive
  - Competitive environment is where the agents' goals are in conflict
  - Adversarial Search
- Game Theory
  - A branch of economics
  - Views the impact of agents on others as significant rather than competitive (or cooperative).

Why Games?

- Small defined set of rules
- Well defined knowledge set
- Easy to evaluate performance
- Large search spaces
  - Too large for exhaustive search
- Fame and Fortune
  - e.g. Chess and Deep Blue

Games as Search Problems

- Games have a state space search
  - Each potential board or game position is a state
  - Each possible move is an operation to another state
  - The state space can be HUGE!!!!!!!
    - Large branching factor (about 35 for chess)
    - Terminal state could be deep (about 50 for chess)

Games vs. Search Problems

- Unpredictable opponent
- Solution is a strategy
  - Specifying a move for every possible opponent reply
- Time limits
  - Unlikely to find the goal…agent must approximate

Types of Games

	Deterministic	**Chance**
Perfect Information	*Chess, checkers, go, othello*	*Backgammon, monopoly*
Imperfect Information		*Bridge, poker, scrabble*

AI Defeating Humans in Games Over Time

- **1997 – Deep Blue (Chess)**
  - IBM's Deep Blue defeats World Chess Champion Garry Kasparov.
- **2011 – IBM Watson (Jeopardy!)**
  - Watson beats Jeopardy! champions Ken Jennings and Brad Rutter.
- **2016 – AlphaGo (Go)**
  - DeepMind's AlphaGo defeats Go grandmaster Lee Sedol.
- **2017 – AlphaZero (Chess, Shogi, Go)**
  - AlphaZero trains itself and beats top chess engines.
- **2019 – OpenAI Five (Dota 2)**
  - OpenAI's team defeats professional Dota 2 players.
- **2022 – CICERO (Diplomacy)**
  - Meta's CICERO masters human negotiation in Diplomacy.

Two player zero sum games

- The games most commonly studied within AI (such as chess and Go) are what game theorists call deterministic, two-player, turn-taking, perfect information, zero-sum games.
- **"Perfect information"** is a synonym for **"fully observable,"** and **"zero-sum"** means that what is good for one player is just as bad for the other: there is no **"win-win"** outcome.
- For games we often use the term move as a synonym for **"action"** and **position** as a synonym for **"state."**

Optimal Decisions in Games

- Consider games with two players (MAX, MIN)
- Initial State
  - Board position and identifies the player to move
- Successor Function
  - Returns a list of (move, state) pairs; each a legal move and resulting state
- Terminal Test
  - Determines if the game is over (at terminal states)
- Utility Function
  - Objective function, payoff function, a numeric value for the terminal states (+1, -1) or (+192, -192)

Game Structure

A game can be formally defined with the **following elements:**

S0: The initial state, which specifies how the game is set up at the start.

TO-MOVE (s): The player whose turn it is to move in state .

ACTIONS (s): The set of legal moves in state .

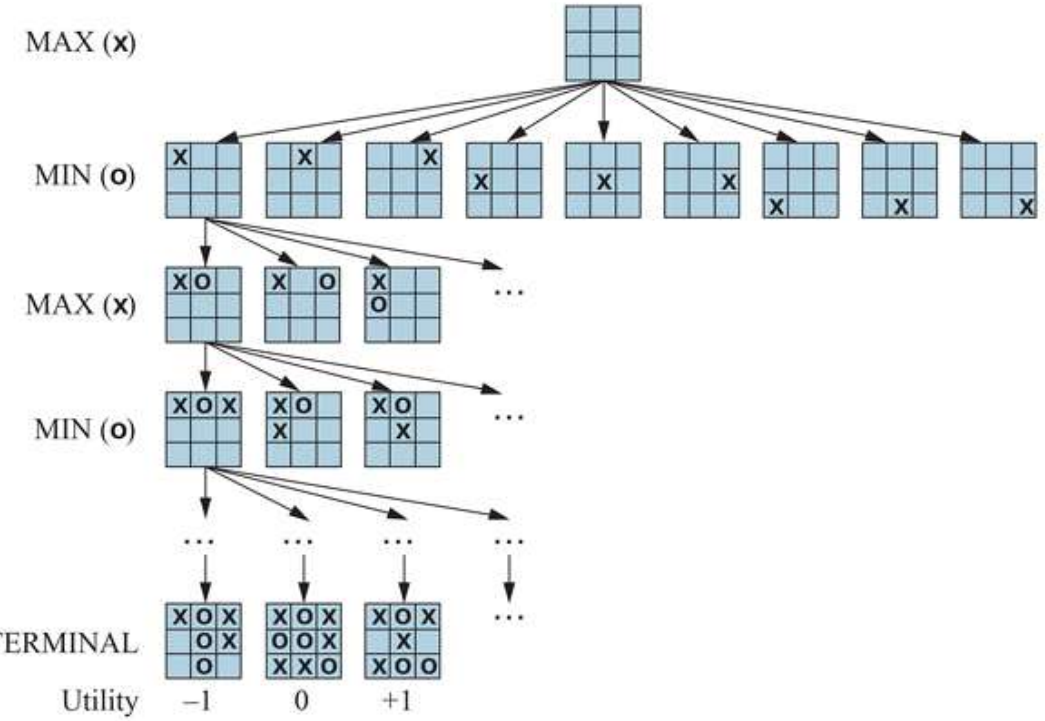
RESULT (s, a): The transition model, which defines the state resulting from taking action a in state s

IS-TERMINAL (s): A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.

UTILITY (s, p): A utility function (also called an objective function or payoff function), which defines the final numeric value to player when the game ends in terminal state p. In chess, the outcome is a win, loss, or draw, with values , 0, or 1 s ½.

Game Trees

- From the initial state, MAX has nine possible moves.
- Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three squares in a row or all the squares are filled.
- The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are good for MAX and bad for MIN (which is how the players get their names)



Game Trees

- The root of the tree is the initial state
  - Next level is all of MAX's moves
  - Next level is all of MIN's moves
  - …
- Example: Tic-Tac-Toe
  - Root has 9 blank squares (MAX)
  - Level 1 has 8 blank squares (MIN)
  - Level 2 has 7 blank squares (MAX)
  - …
- Utility function:
  - win for X is +1
  - win for O is -1

Algorithms in adversarial search

Minimax Algorithm:

- A decision rule for minimizing the possible loss for a worst-case scenario.
- Assumes both players play optimally.
- Used in chess engines, tic-tac-toe solvers, etc.

Alpha-Beta Pruning:

- An optimized version of Minimax that **prunes** unnecessary branches in the search tree.
- Makes the search process **faster** without changing the result.

Minimax Search

- MAX wants to find a sequence of actions leading to a win, but MIN has something to say about it. This means that MAX's strategy must be a conditional plan—a contingent strategy specifying a response to each of MIN's possible moves.
- For games with multiple outcome scores, the general algorithm is called **minimax search.**

Minimax Algorithm

- The possible moves for MAX at the root node are labeled a1, a2 , a3.
- The possible replies to a1 for MIN are b1 b2 b3 and so on.
- This particular game ends after one move each by MAX and MIN.
- The utilities of the terminal states in this game range from 2 to 14.

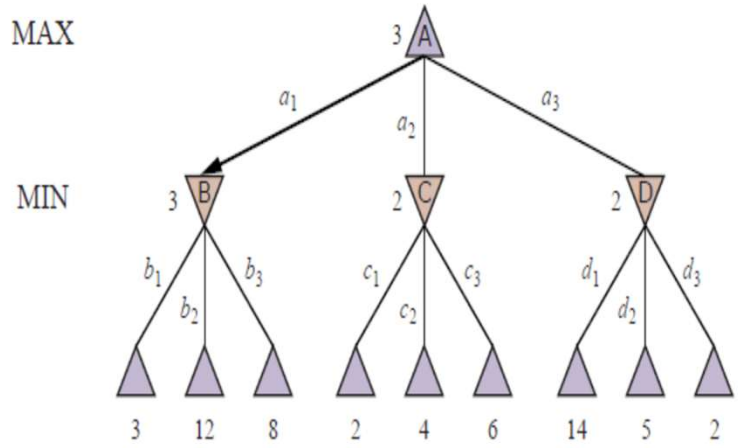


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Minimax Strategy

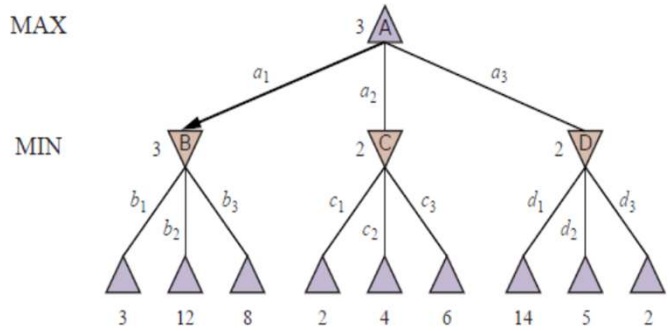
- Basic Idea:
  - Choose the move with the highest minimax value
    - best achievable payoff against best play
  - Choose moves for Max that will lead to a win, even though Min is trying to block

- Max's goal: get to 1 (for zero sum games)
- Min's goal: get to -1 "

- Minimax value of a node (backed up value):
  - If N is **terminal**, use the utility value
  - If N is a Max move, take max of successors
  - If N is a Min move, take min of successors

Cont..

- Given a game tree, the optimal strategy can be determined by working out the minimax value of each state in the tree, which we write as MINIMAX(s).
- The minimax value is the utility (for MAX) of being in that state, assuming that both players play optimally from there to the end of the game.
- The minimax value of a terminal state is just its utility.
- In a non-terminal state, MAX prefers to move to a state of maximum value when it is MAX's turn to move, and MIN prefers a state of minimum value (that is, minimum value for MAX and thus maximum value for MIN).

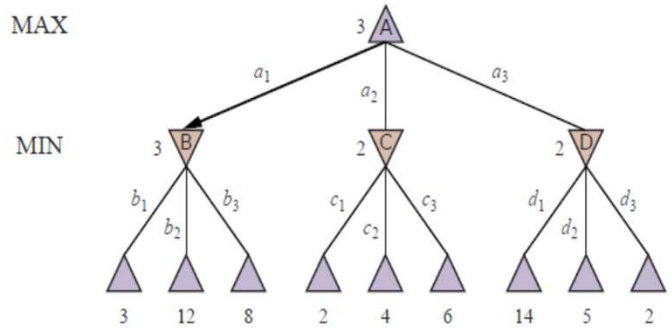


$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s, \textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s, a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s, a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \end{cases}$$

The minimax search algorithm

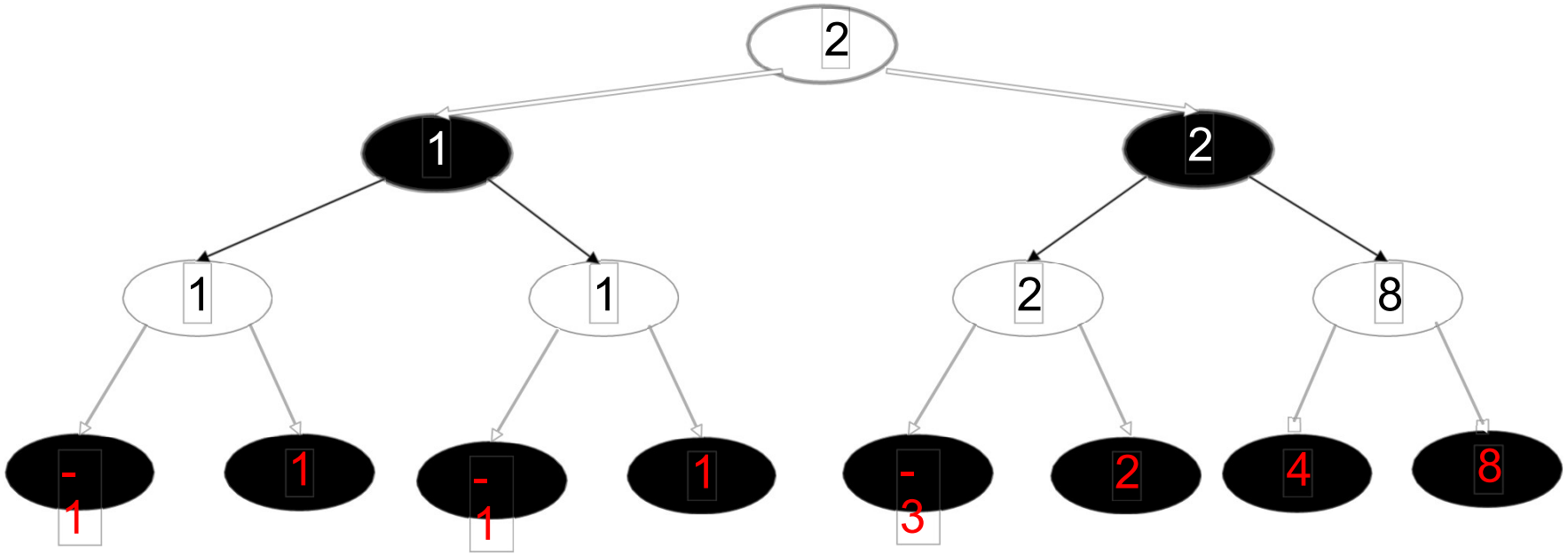
- Now that we can compute MINIMAX(s), we can turn that into a search algorithm that finds the best move for MAX by trying all actions and choosing the one whose resulting state has the highest MINIMAX value. It is a recursive algorithm that proceeds all the way down to the leaves of the tree and then backs up the minimax values through the tree as the recursion unwinds.

```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move

function MAX-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
        if v2 > v then
            v, move ← v2, a
    return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
        if v2 < v then
            v, move ← v2, a
    return v, move
```

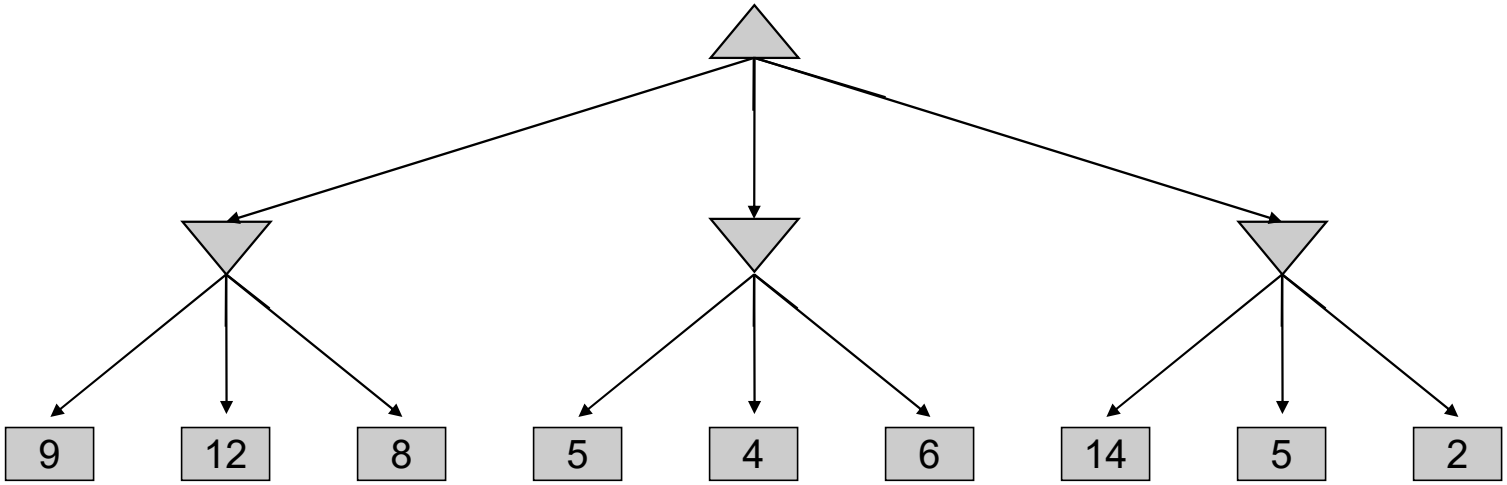


Minimax Algorithm

White – Maximize
Black - Minimize



Minimax Example



Properties of minimax

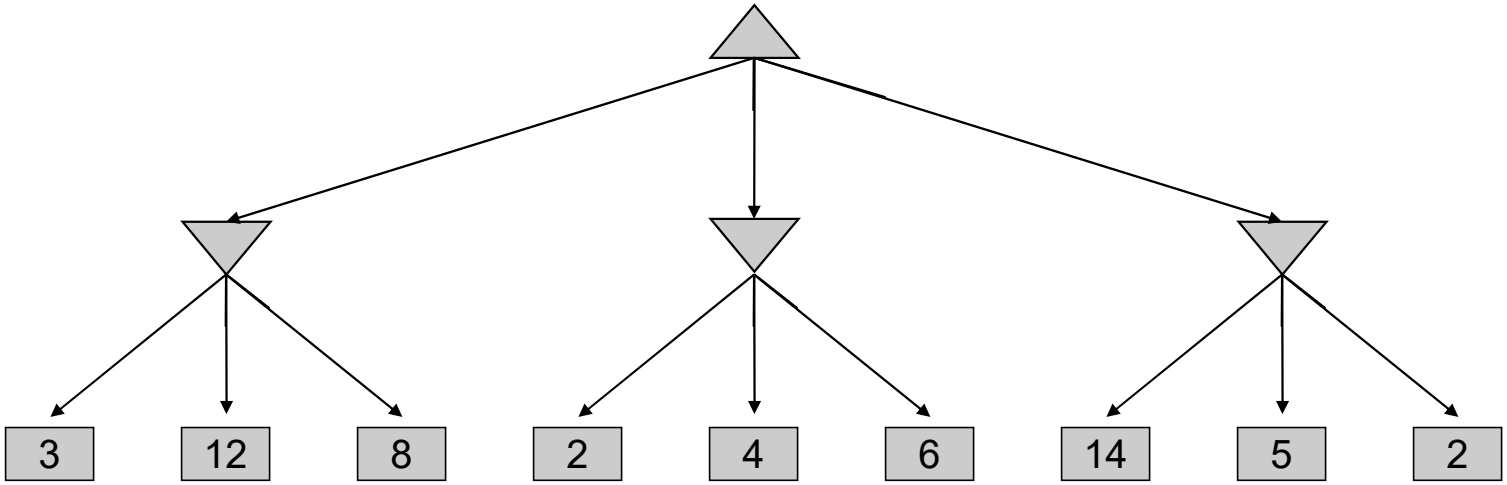
- **Complete?** Yes (if tree is finite)
- **Optimal?** Yes (against an optimal opponent)
- **Time complexity?** $O(b^m)$
- **Space complexity?** O(bm) (depth-first exploration)

- For chess, b ≈ 35, m ≈80 for "reasonable" games
  → exact solution completely infeasible, $35^{80} = 10^{123}$ states
- Do we really need to explore every path???

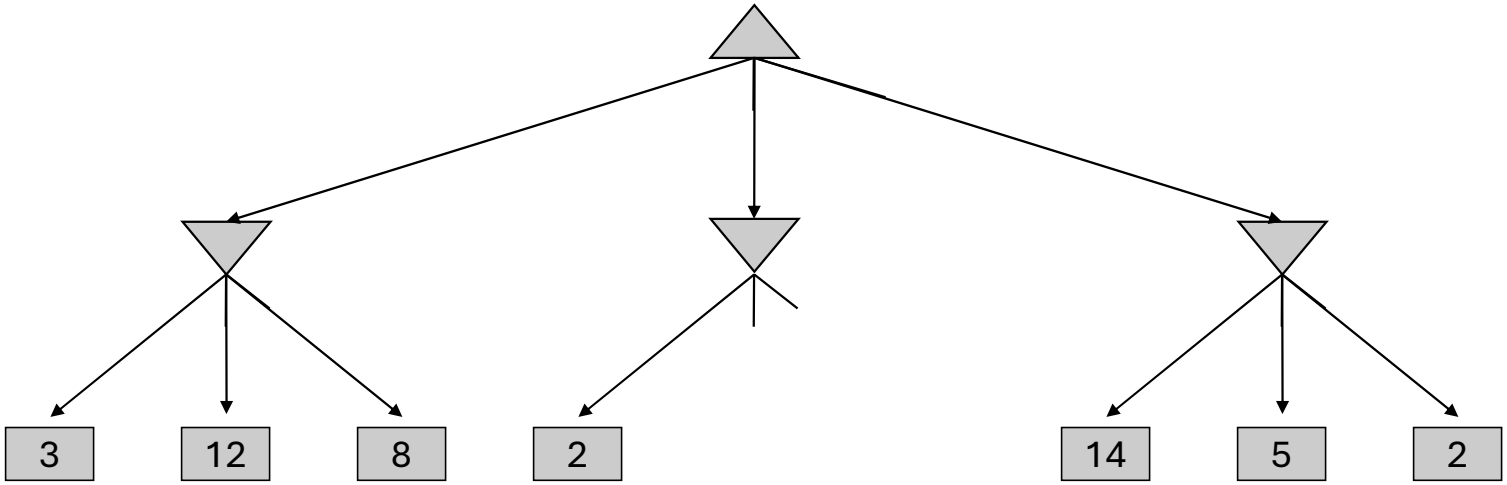
Solutions to the Complexity Problem

- The problem with minimax search is that the number of game states is has to examine is exponential in the number of moves
- Dynamic pruning of redundant branches of the search tree
  - Some branches will never be played by rational players since they include sub-optimal decisions (for either player)
    - Identify a provably suboptimal branch of the search tree before it is fully explored
    - Eliminate the suboptimal branch
  - Procedure: Alpha-Beta Pruning
- Early cutoff of the search tree
  - Use imperfect minimax value estimate of non-terminal states

Minimax Pruning



Minimax Pruning



Alpha-Beta Pruning

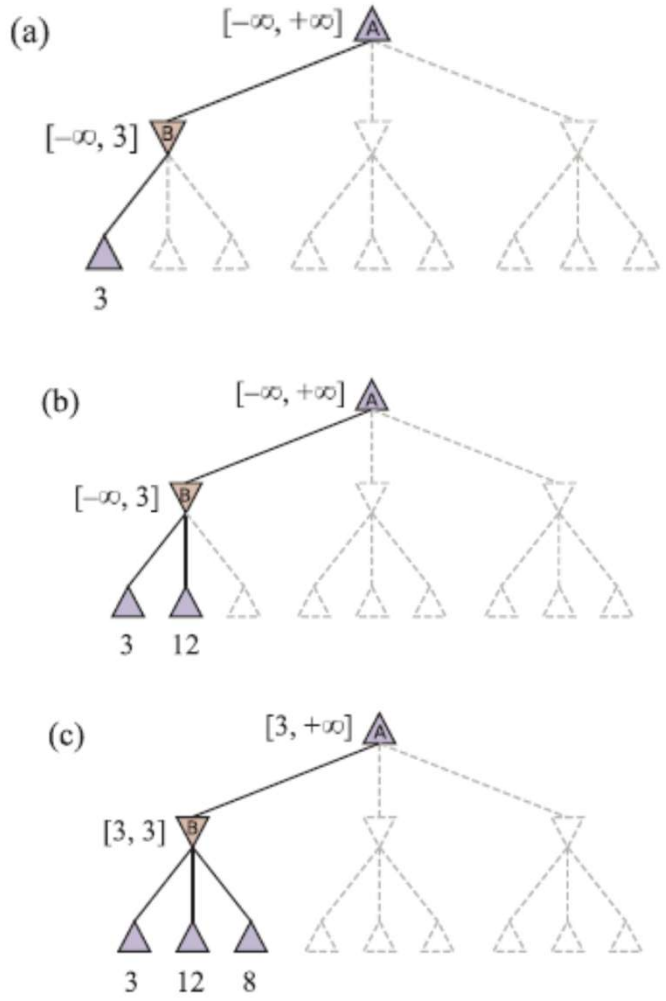
- Recognize when a position can never be chosen in minimax *no matter what its children are*
  - Max (3, Min(2,x,y) …) is always ≥ 3
  - Min (2, Max(3,x,y) …) is always ≤ 2
  - We know this without knowing x and y!

Alpha-Beta Pruning

- Alpha–beta pruning gets its name from the two extra parameters in MAX-VALUE(state,α,β) that describe bounds on the backed-up values that appear anywhere along the path
- Alpha = the value of the best choice we've found so far for MAX (highest)
- Beta = the value of the best choice we've found so far for MIN (lowest)
- When maximizing, cut off values lower than Alpha
- When minimizing, cut off values greater than Beta

Alpha-Beta Example

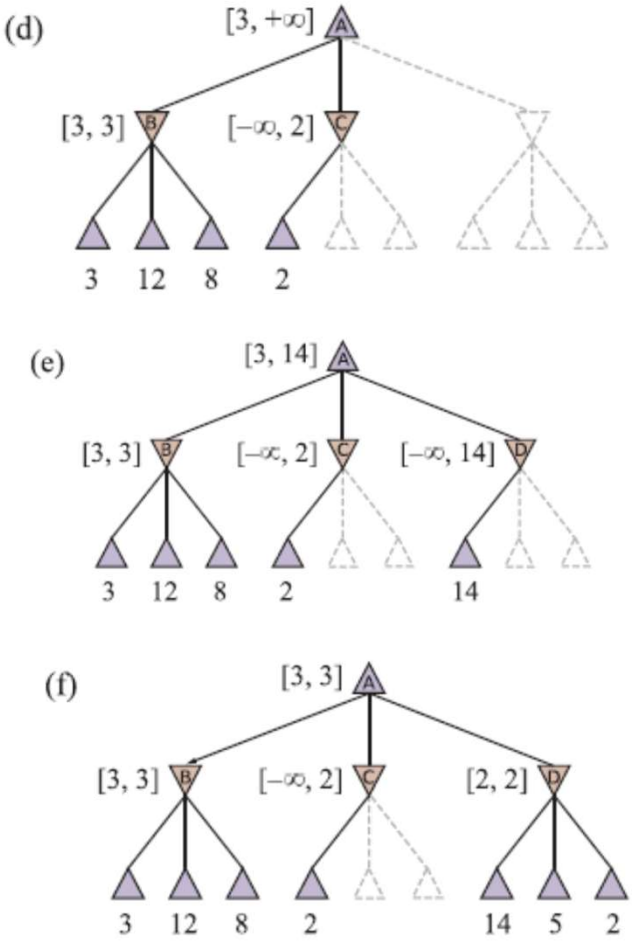
- (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of at most 3.
- (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3.
- (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now we can infer that the value of the root is at least 3, because MAX has a choice worth 3 at the root.



Alpha-Beta Example

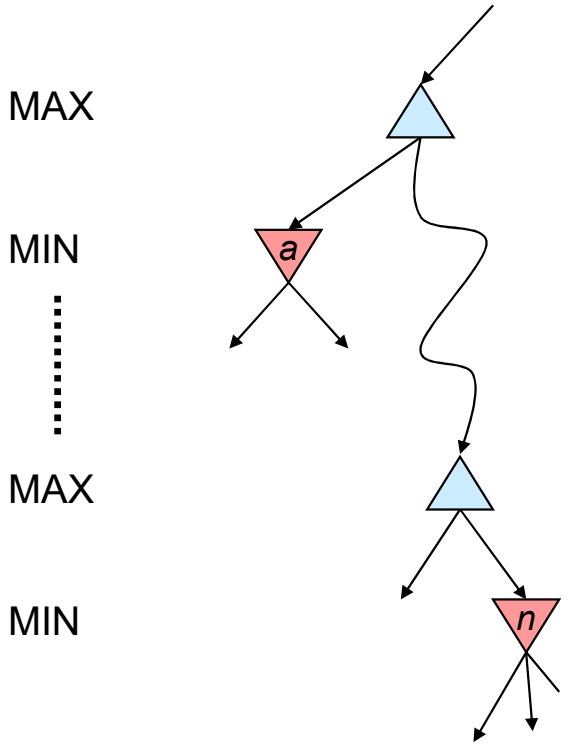
- (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of at most 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of **alpha–beta pruning.**
- (e) The first leaf below D has the value 14, so D is worth at most 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14.
- (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

When maximizing, cut off values lower than Alpha
When minimizing, cut off values greater than Beta



Alpha-Beta Pruning

- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node *n*
  - We're looping over *n*'s children
  - *n*'s estimate of the childrens' min is dropping
  - Who cares about *n*'s value? MAX
  - Let *a* be the best value that MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than *a*, MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)



- MAX version is symmetric

  When maximizing, cut off values lower than Alpha
  When minimizing, cut off values greater than Beta

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

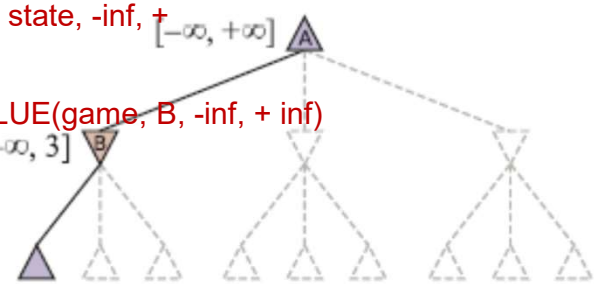
cut off values greater than Beta

cut off values lower than Alpha

1 Value,move=MAX-VALUE(game, state, -inf, + inf)

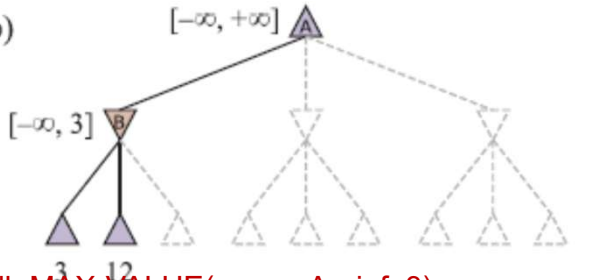
2 V2,a2=MIN-VALUE(game, B, -inf, + inf)
V2=3<+inf
Beta= MIN(+inf,3)
V=3> -inf

3 3,null=MAX-VALUE(game, A, -inf, + inf)



V2=12 >3
V=3> -inf

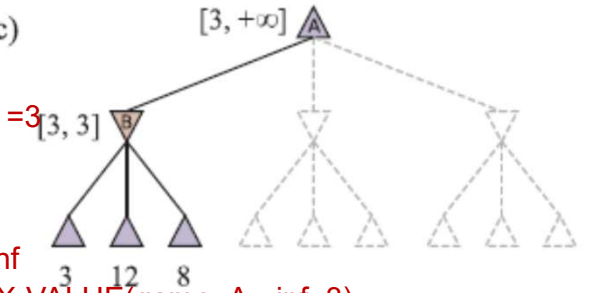
4 12,null=MAX-VALUE(game, A, -inf, 3)



V= -inf
Return v2=3
V2=3>-inf
V=3, Alpha= Max(-inf, 3) =3

V2=8>3
V=3 > -inf

5 8,null=MAX-VALUE(game, A, -inf, 3)



https://pascscha.ch/info2/abTreePractice/

Notes on Alpha-Beta Pruning

- Pruning *does not* affect the final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering", time complexity $O(b^{m/2})$
  - This means that the effective branching factor becomes $\sqrt{b}$ instead of b—for chess, about 6 instead of 35.
  - Put another way, alpha– beta with perfect move ordering can solve a tree roughly twice as deep as minimax in the same amount of time.
  - doubles the depth of search, can easily reach depth of 80 and play good chess (branching factor of 6 instead of 35)

Optimizing Minimax Search

- Use alpha-beta cutoffs
  - Evaluate most promising moves first
- Redundant paths to repeated states can cause an exponential increase in search cost.
  - Keeping a table of previously reached states can address this problem.
  - In game tree search, repeated states can occur because of transpositions—different permutations of the move sequence that end up in the same position, and the problem can be addressed with a transposition table that caches the heuristic value of states.
- But, we still can't search a game like chess to the end!

Cutting Off Search : Killer Moves

- Replace terminal test (end of game) by cutoff test (don't search deeper)
- Replace utility function (win/lose/draw) by heuristic evaluation function that estimates results on the best path below this board
  - Like A* search, good evaluation functions mean good results (and vice versa)
- Replace move generator by plausible move generator (don't consider "dumb" moves)

Comparison

Feature	Minimax Algorithm	Alpha-Beta Pruning
Purpose	Determines the optimal move in adversarial games	Optimizes Minimax by reducing explored nodes
Efficiency	Explores all possible game states	Prunes unnecessary branches, reducing computations
Time Complexity	O(bd) where b is the branching factor and d is depth	Best case: O(bd/2) (reduces depth effectively)
Pruning	No pruning; evaluates all nodes	Prunes branches that do not affect final decision
Space Complexity	O(bd)	O(bd) (same as Minimax but faster execution)
Performance	Slower, especially in deep trees	Faster due to pruning, works efficiently in deep trees
Usage	Used in small search spaces	Preferred for complex games like Chess, Go
Optimal Result	Always finds the best move	Finds the best move but more efficiently