Adversarial Search

MAEs and Games

- Multi-agent environment: every agent needs to consider the actions of the other agents, in order to optimize its own welfare
  - Cooperative: Agents act collectively to achieve a common goal
  - Competitive:
    - Agents compete against each other
    - Their goals are in conflict
    - Gives rise to the concept of adversarial search problems – often known as games.

Game Theory

- MAE as a game provided that the impact of each agent on the other is "significant", i.e., able to affect the actions of the other agent(s)
- In AI, "game" is a specialized concept:
  - Deterministic, fully-observable environments
  - Two agents whose actions must alternate
  - Utility values at the end of the game are always equal and opposite
    - +1 = Chess winner
    - -1 = Chess looser.

AI Games

- Tackled by Konrad Zuse, Claude Shannon, Norbert Wiener, Alan Turing
  - Have seen lot of successes recently, e.g., DeepBlue
- Game states are easy to represent:
- Agents restricted by a limited action rules
- Outcomes defined by precise rules
- Games:
  - Chess: average branching factor of 35
  - If 50 moves by each player, search tree has 35^{100} nodes!

Games vs. Search problems

- In typical search problems, we optimize a measure to acquire the goal: there is no opponent
- In games, there is an "Unpredictable" opponent
  - Need to specify a move for every possible opponent reply
  - Strict penalty on an inefficient move
  - Stringent time constraints
  - Unlikely to find goal, must approximate
  - Requires some type of a decision to move the search forward.

Zero Sum Game

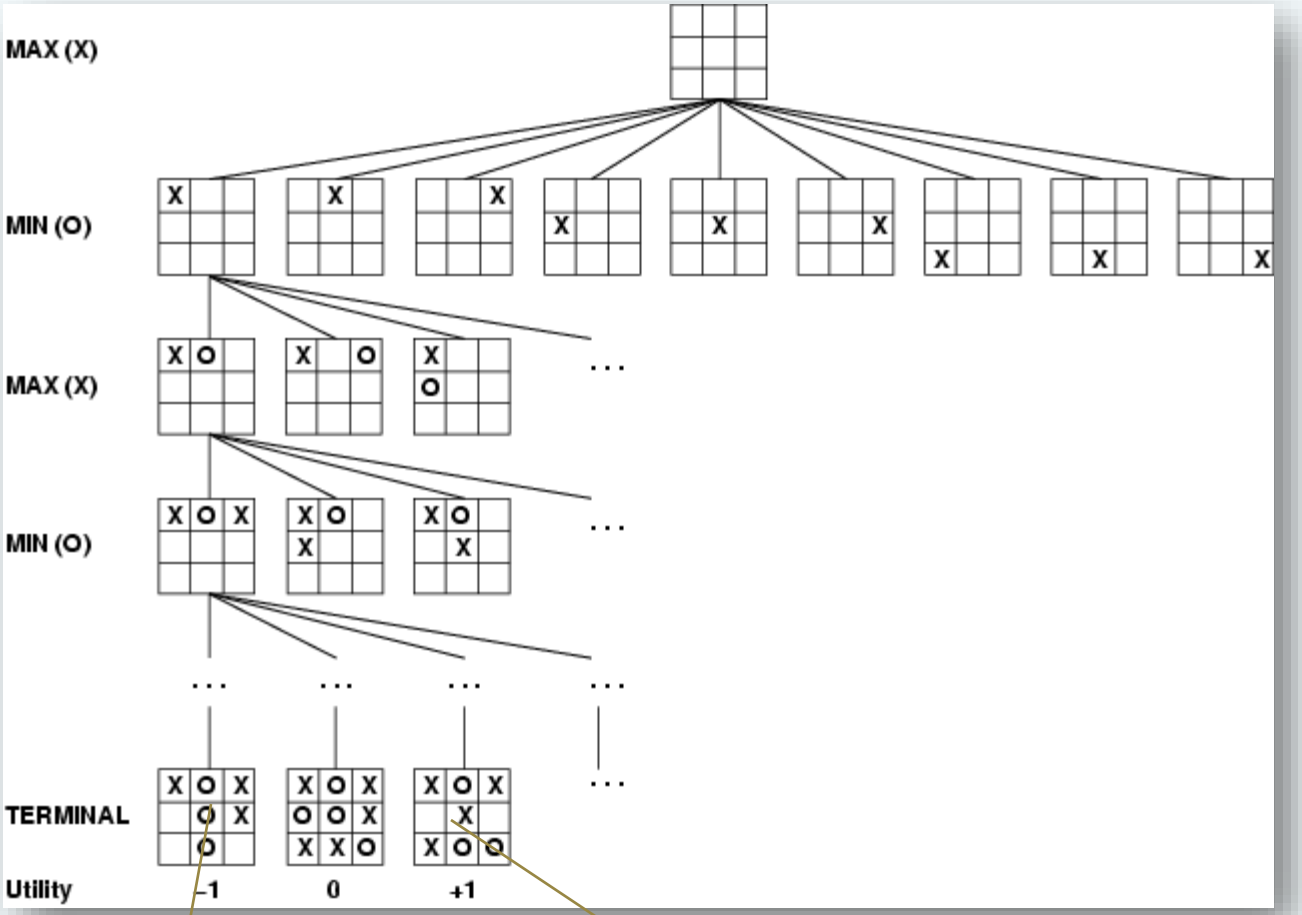
- Imagine 2 players of tic-tac-toe: MAX and MIN
  - MAX moves first: We can generate a game tree
  - The terminal states are at the leaves

- MAX should play in order to maximize its utility, which will minimize the utility for MIN
  - This is called a Zero-Sum Game.

A game defined as a kind of search problem

- S0: The initial state, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The transition model, which defines the result of a move.
- TERMINAL-TEST(s): A terminal test, which is true when the game is over and false, otherwise.
- UTILITY(s, p): A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p.

Game tree (2-player, deterministic, fully observable)



This terminal state is one of the worst for MAX and one of the best for MIN

This terminal state is one of the best for MAX and one of the worst for MIN

Optimal Strategy for MAX

- This is a MAE
  - In discovering the sequence of optimal moves by MAX, we have to consider that MIN is taking moves as well
  - So, the optimal strategy will tell how should MAX play against MIN (by considering the moves of the latter), in order to win the game
- Let us consider a shortened game tree of TIC-TAC-TOE
  - Numbers are the utilities
  - Ply: a move by MAX followed by a move from MIN.

Game Tree

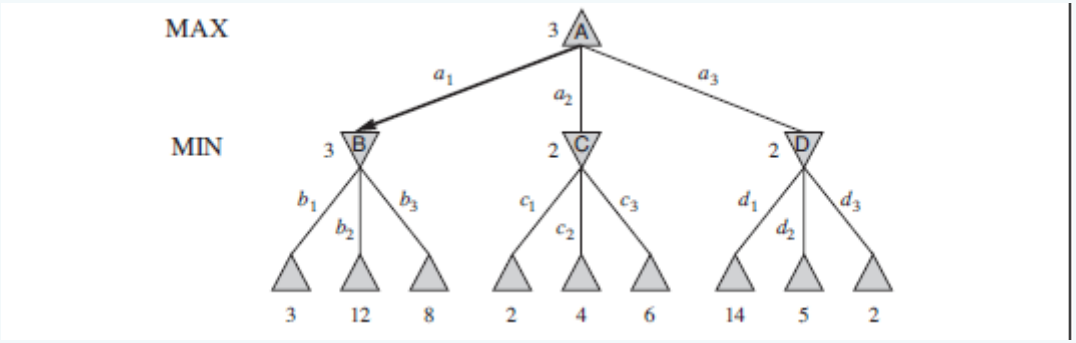
- The initial state, ACTIONS function, and RESULT function define the **game tree** for the game where the nodes are game states and the edges are moves.
- Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states .
- In terminal states one player has three 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 assumed to be good for MAX and bad for MIN

Minimax

- When it is the turn of MAX, it will always take an action in order to maximize its utility, because it's winning configurations have high utilities
- When it is the turn of MIN, it will always take an action in order to minimize its utility, because it's winning configurations have low utilities
- In order to implement this, we need to define a measure in each state that takes the move of the opponent into account:
  - This measure is called Minimax.

Minimax

- Minimax represents the utility of a state, *given that both MAX and MIN will play optimally till the end of the game*
- In any state *s*, one or more actions are possible
- For every possible new state that can be transited into from *s*, we compute the minimax value
- The term "Minimax" is used because:
  - the opponent is always trying to minimize the utility of the player, and
  - the player is always trying to maximize this minimized selection of the opponent.



• The Δ nodes are "MAX nodes," in which it is MAX' turn to move, and the ∇ nodes are "MIN nodes."

• The terminal nodes show the utility value 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.

- The possible moves for MAX at the root node are labeled a_1, a_2, and a_3.
- The possible replies to a_1 for MIN are b_1, b_2, b_3, and so on.
- This particular game ends after one move each by MAX and MIN.
- The utilities of PLY the terminal states in this game range from 2 to 14.
- Given a game tree, the optimal strategy can be determined from the **minimax** value of each node, which we write as MINIMAX(n).

- The **minimax** value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game.

- MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.

- MINIMAX(s) =

  UTILITY(s) if TERMINAL-TEST(s)

  { $\text{Max}_{a \in \text{Actions(s)}}$ MINIMAX(RESULT(s, *a*)) if PLAYER(s) = MAX

  $\text{Min}_{a \in \text{Actions(s)}}$ MINIMAX(RESULT(s, *a*)) if PLAYER(s) = MIN

- The first MIN node, labeled B, has three successor states with values 3, 12, and 8, so its minimax value is 3.

- Similarly, the other two MIN nodes have minimax value 2.

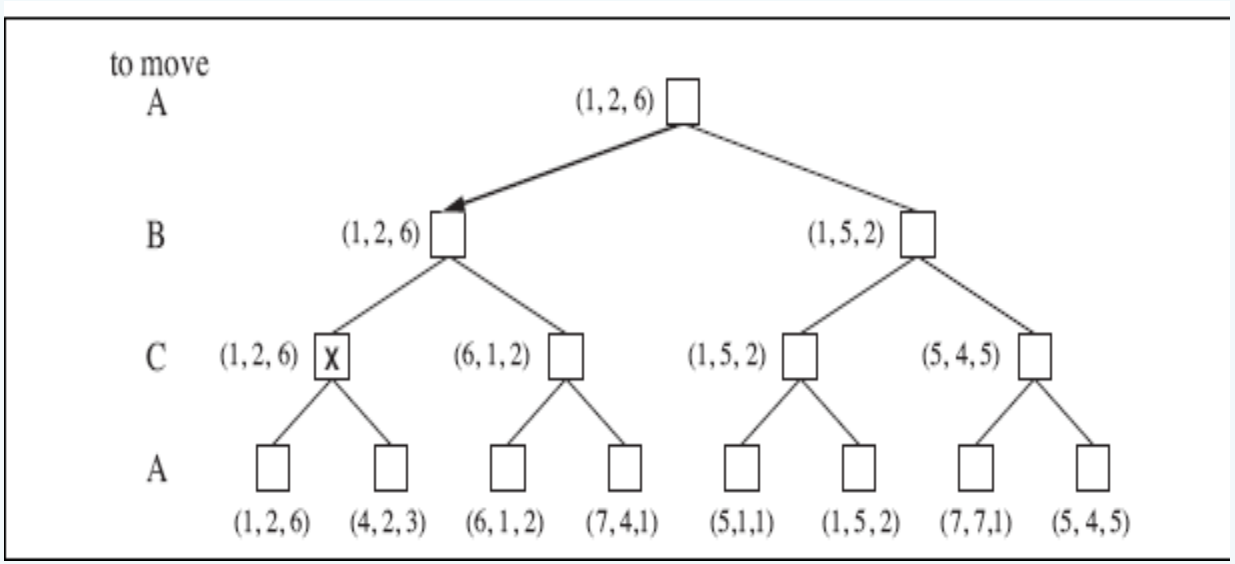
- The root node is a MAX node; its successor states have **minimax** values 3, 2, and 2; so it has a **minimax** value of 3.
- The minimax decision at the root: action a_1 is the optimal choice for MAX because it leads to the state with the highest **minimax** value.
- This definition of optimal play for MAX assumes that MIN also plays optimally—it maximizes the worst-case outcome for MAX.
- If MIN does not play optimally then MAX will do even better

- The **minimax** algorithm performs a complete depth-first exploration of the game tree.
- If the maximum depth of the tree is *m* and there are *b* legal moves at each point, then the time complexity of the **minimax** algorithm is $O(b^m)$.
- The space complexity is $O(b^m)$ for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time.
- For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

Optimal decisions in multiplayer games

- First, we need to replace the single value for each node with a *vector of values.*
- For example, in a three-player game with players A, B, and C, a vector v_A, v_B, v_C is associated with each node.
- For terminal states, this vector gives the utility of the state from each player's viewpoint.
- The simplest way to implement this is to have the UTILITY function return a vector of utilities.

Nonterminal states Consider the node marked X



In corresponding state, player C has two choices lead to terminal states with utility vectors $v_A = 1$, $v_B = 2$, $v_C = 6$ and $v_A = 4$, $v_B = 2$, $v_C = 3$.

• C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities $v_A = 1$, $v_B = 2$, $v_C = 6$. Hence, the backed-up value of X is this vector.

- The backed-up value of a node *n* is always the utility vector of the successor state with the highest value for the player choosing at *n*.
- Multiplayer ALLIANCE games usually involve **alliances, whether** formal or informal, among the players.
- Suppose A and B are in weak positions and C is in a stronger position.
- Then it is often optimal for both A and B to attack C rather than each other, lest C destroy each of them individually.
- As soon as C weakens under the joint onslaught, the alliance loses its value, and either A or B could violate the agreement.

Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Depth-First Exploration

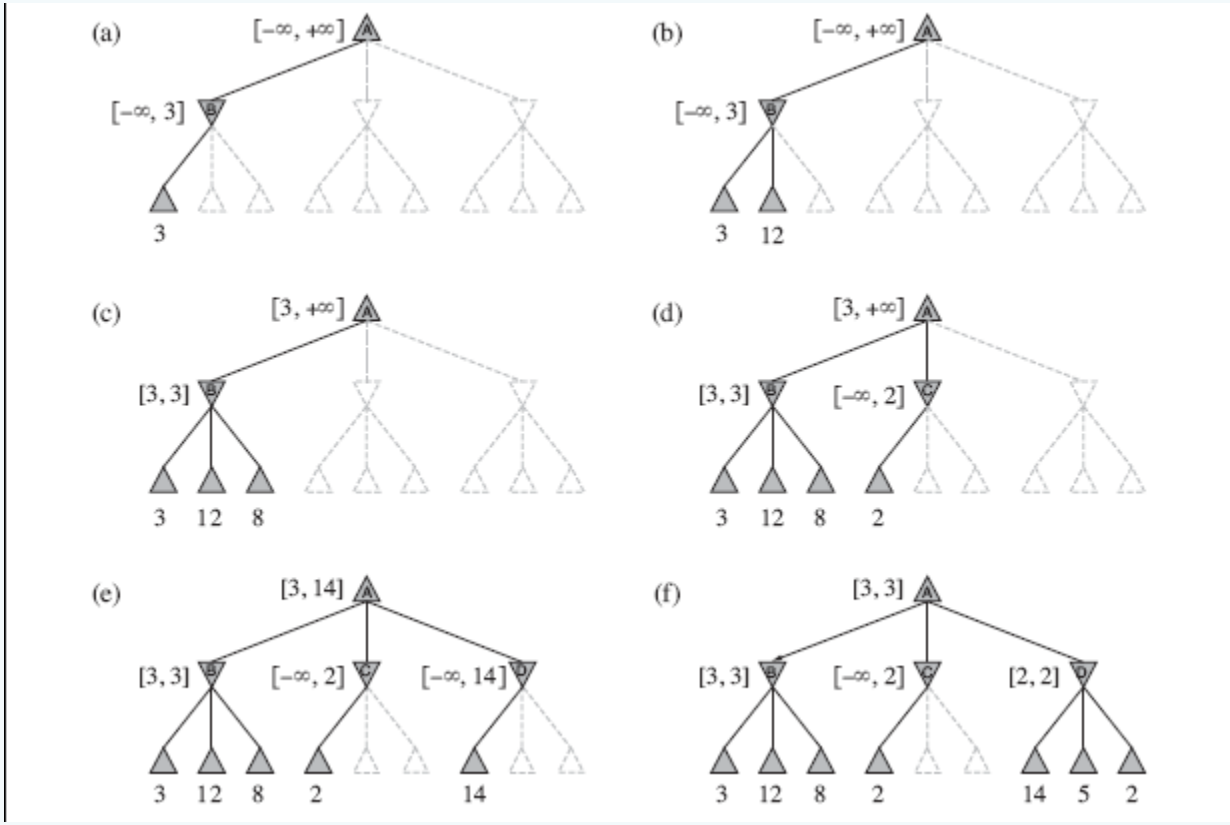
Recursion: Winds all the way to the terminal nodes, and then unwinds back by backing up the values

Min(minimax) for MIN moves
Max(minimax) for MAX moves

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 ≈100 for "reasonable" games → exact solution completely infeasible
  - We need to think of a way to cut down the number of search paths.

- The problem with **minimax** search is that the number of game states it has to examine is exponential in the depth of the tree.
- Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half.
- It is possible to compute the correct **minimax** decision without looking at every node in the game tree. That is, we can borrow the idea of **pruning**.
- **ALPHA–BETA** pruning when applied to a standard **minimax** tree, it returns the same move as **minimax** would, but prunes away branches that cannot possibly influence the final decision.



At each point, we show the range of possible values for each node. (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.

Example of alpha–beta pruning

(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, no point in looking at the other successor states of C.

(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.

- 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.

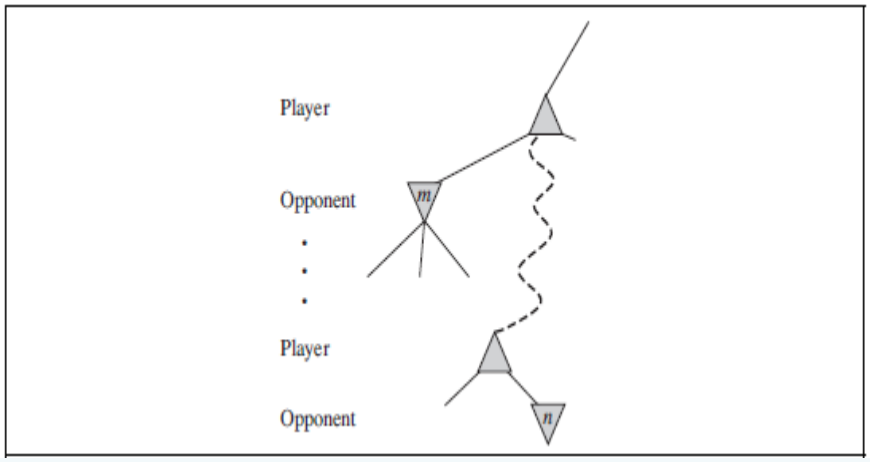
Simplification of the formula for MINIMAX

Let the two unevaluated successors of node C have values x and y.
Then the value of the root node is given by

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

In other words, the value of the root and hence the minimax decision are *independent of the* values of the pruned leaves x and y.

Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves.



The general principle is: consider a node *n* somewhere in the tree such that Player has a choice of moving to that node. If Player has a better choice *m* either at the parent node of *n* or at any choice point further up, then *n will never be reached in actual play.* *So once we have found out enough about n (by* examining some of its descendants) to reach this conclusion, we can prune it.

Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree. Alpha–beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear any where along the path

α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively.

- The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. For example, in Figure 5.5(e) and (f), we could not prune any successors of D at all because the worst successors (from the point of view of MIN) were generated first.

- If the third successor of D had been generated first, we would have been able to prune the other two.

- This suggests that it might be worthwhile to try to examine first the successors that are likely to be best.

- If this can be then it turns out that alpha–beta needs to examine only O(bm/2)nodes to pick the best move, instead of O(bm) for minimax. This means that the effective branching factor becomes √b instead of b—for chess, about 6 instead of 35.

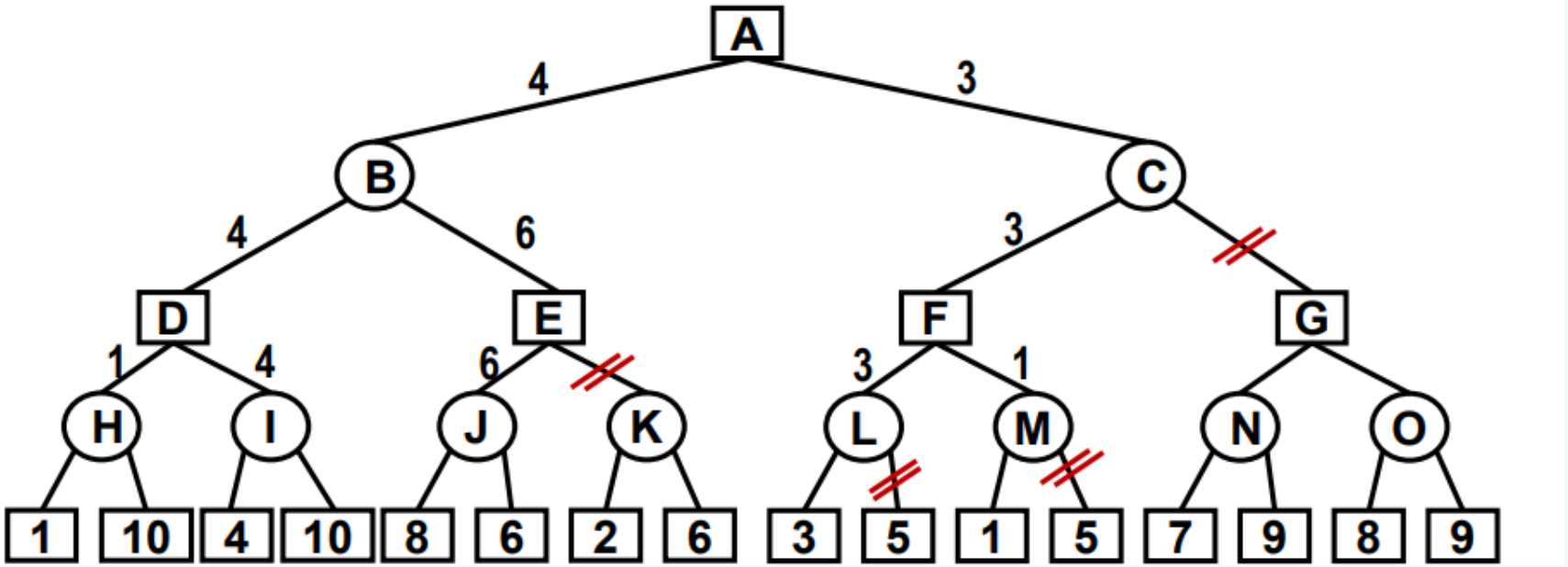
- Put another way, alpha–beta can solve a tree roughly twice as deep as minimax in the same amount of time.

- If successors are examined in random order rather than best-first, the total number of nodes examined will be roughly O(b3m/4) for moderate b.

- For chess, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, and then backward moves) gets you to within about a factor of 2 of the best-case O(bm/2) result.

ALPHA-BETA PRUNING PROCEDURE V(J;α,β)

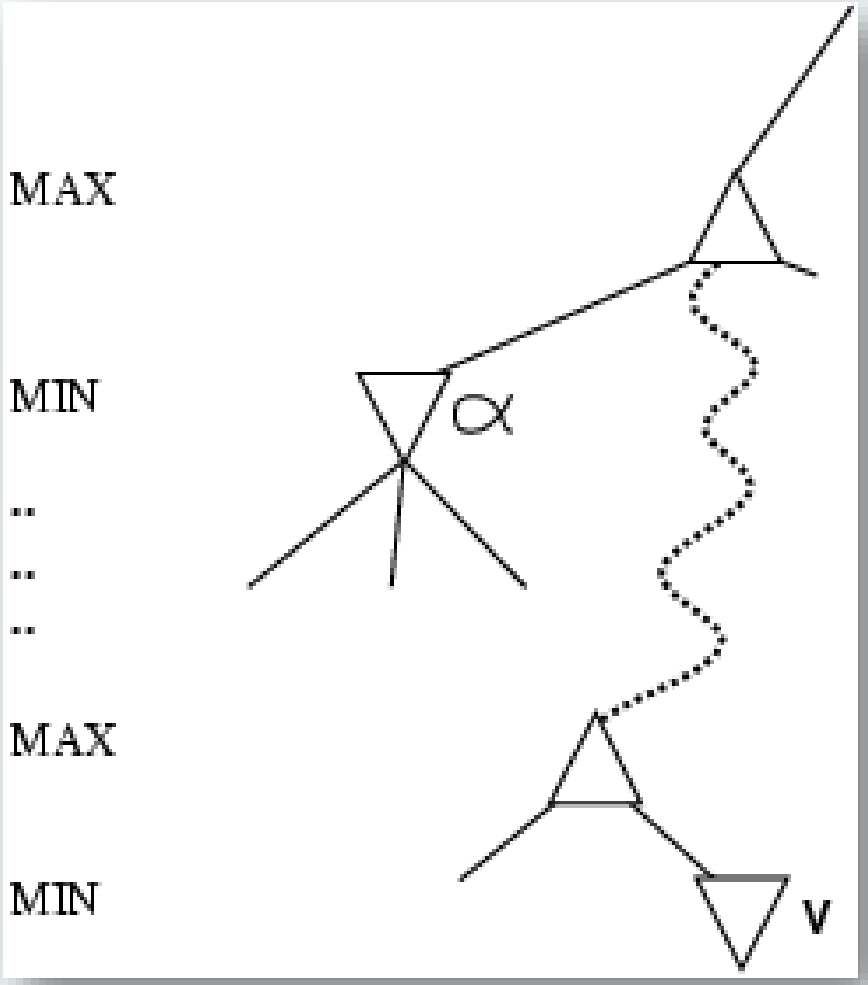
The initial call is with
V(Root ; –∞, +∞)



Why is it called α-β?

- α is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for *MAX*
- If *v* is worse than α, *MAX* will avoid it

  → prune that branch
- Define β similarly for *MIN*.



The α-β algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
    inputs: state, current state in game

    v ← MAX-VALUE(state, −∞, +∞)
    return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s, α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
```

Properties of α-β

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering," time complexity = $O(b^{m/2})$
  → doubles depth of search
- A simple example of the value of reasoning about which computations are relevant (a form of meta-reasoning)