Adversarial search (game playing)

Chapter 5

Outline

- ♢ Games
- ♢ Perfect play
  - – minimax decisions
  - – α–β pruning
- ♢ Imperfect decisions in real time

Adversarial search problems (games)

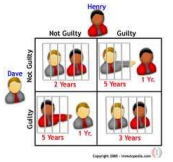
Arise in **competitive multi-agent** environments

In Game Theory, a multi-agent environment is called a game

no opp. of changing results

In AI, a game is often a deterministic, turn-taking, two-player, zero-sum game of perfect information:

- deterministic
- two agents
- whose action alternate
- utility values are opposite e.g. (+1,-1)
- fully observable



We will write algorithms that play such games against an opponent.

Games Definition

A Game consists of:

- ♢ sets of players P, states S (board and player to play), and moves M
- ♢ an initial state $s_0 \in S$ which specifies how the game is set up
- ♢ $\textsc{Player}(s) \in P$: defines the player to move in state s
- ♢ $\textsc{Moves}(s) \in 2^M$: defines the set of legal moves in state s
- ♢ $\textsc{Result}(s,m) \in S$: defines the result of perfoming move m in state s
- ♢ $\textsc{Terminal}(s) \in \mathbb{B}$: the terminal test says whether the game is over
- ♢ $\textsc{Utility}(s,p) \in \mathbb{R}$: the utility function gives a numeric value to terminal states from the point of view of a given player, e.g. $\{+1, -1, 0\}$ for chess or $\{-192, \ldots, 192\}$ for backgammon

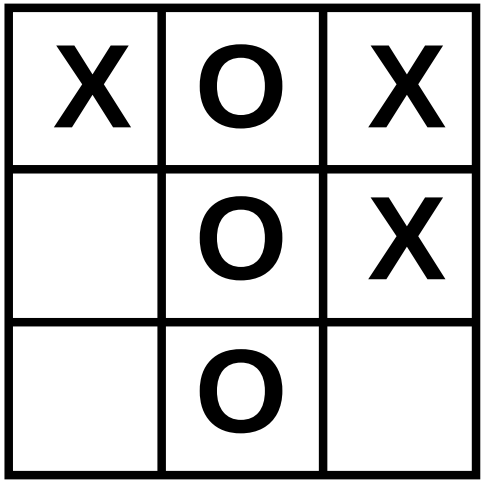
Strategy

♢ "Unpredictable" opponent
⇒ solution for a player is not a sequence of actions but a strategy

♢ A strategy for a player: a function mapping the player's states to (legal) moves.

♢ A winning strategy always lead the player to a win from s_0

Example: tic-tac-toe



states??: content of each cell {X,O,empty}, player to play

moves??: an empty cell

result??: content of chosen cell is X or O depending on the player playing; next player

terminal test??: are 3 O or 3 X aligned or is the board full?

utility function??: for a given player gives +1 if player has aligned 3 tokens, -1 if his opponent has, and 0 otherwise. There is a draw strategy.

Example: nim

states??: number of rows R, number of matches $n(r)$ in each row, player to play

moves??: a non-empty row $r \in R$ and a number of matches $0 < k \leq n(r)$ to remove

result??: $n(r) \leftarrow n(r) - k$, next player

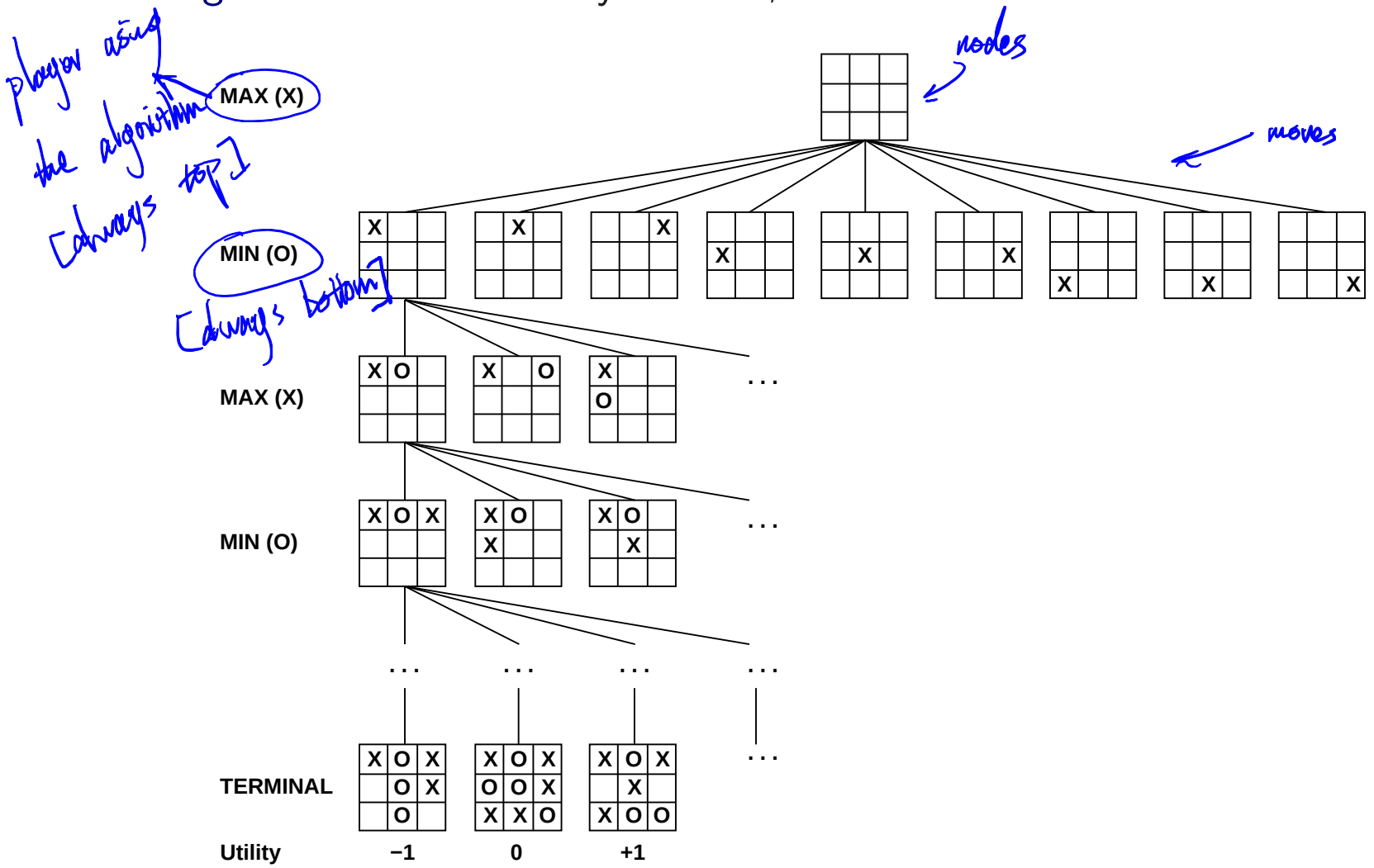
terminal test??: $n(r) = 0$ for all $r \in R$

utility function??: for a given player, $+1$ if it is the player to play, -1 if his opponent is. There is a winning strategy for one of the players.

↑ depending on the number of matches.

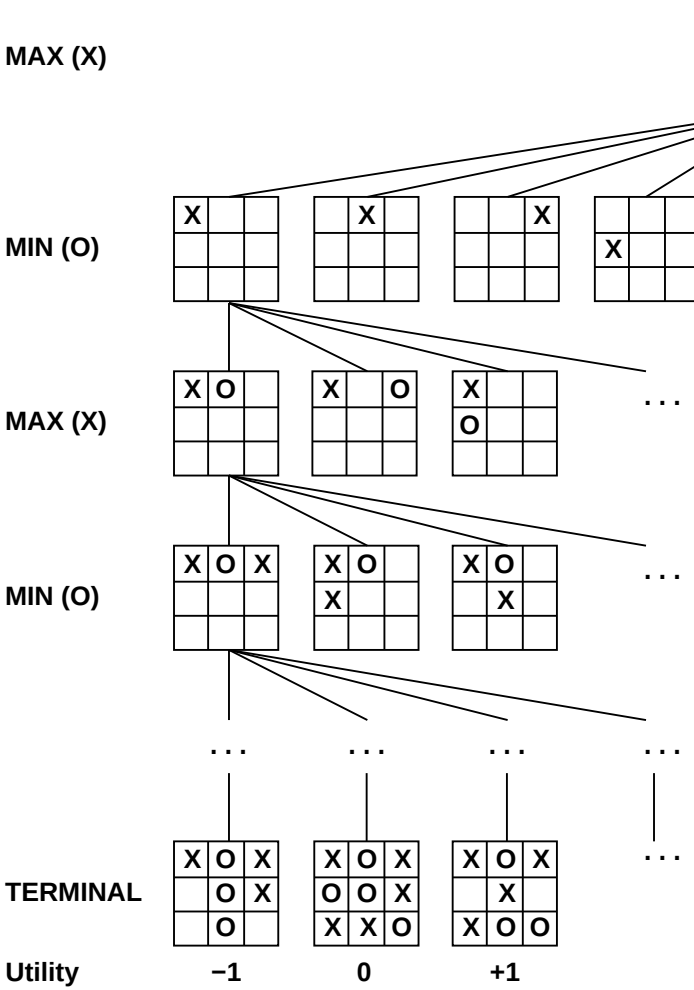
Game tree (2-player, deterministic, turns)

The game tree is defined by MOVES, RESULT and TERMINAL



Game tree (2-player, deterministic, turns)

The game tree is defined by MOVES, RESULT and TERMINAL

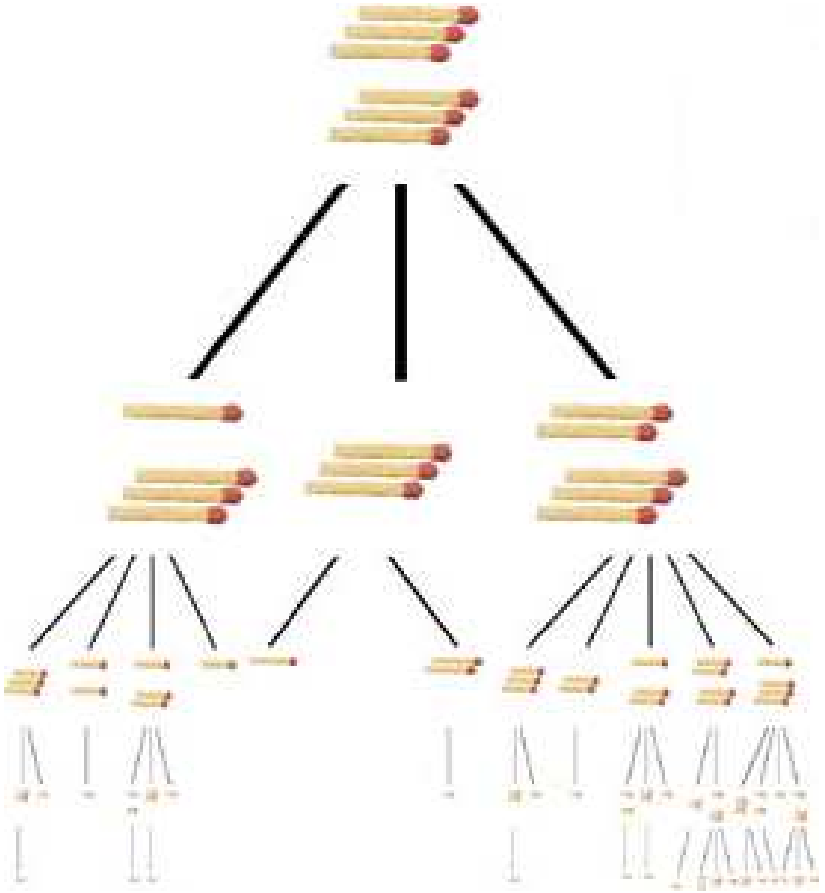


Max is the player using the algorithm to compute his strategy

Max is always at the root of the tree

Game tree (2-player, deterministic, turns)

The game tree is defined by MOVES, RESULT and TERMINAL



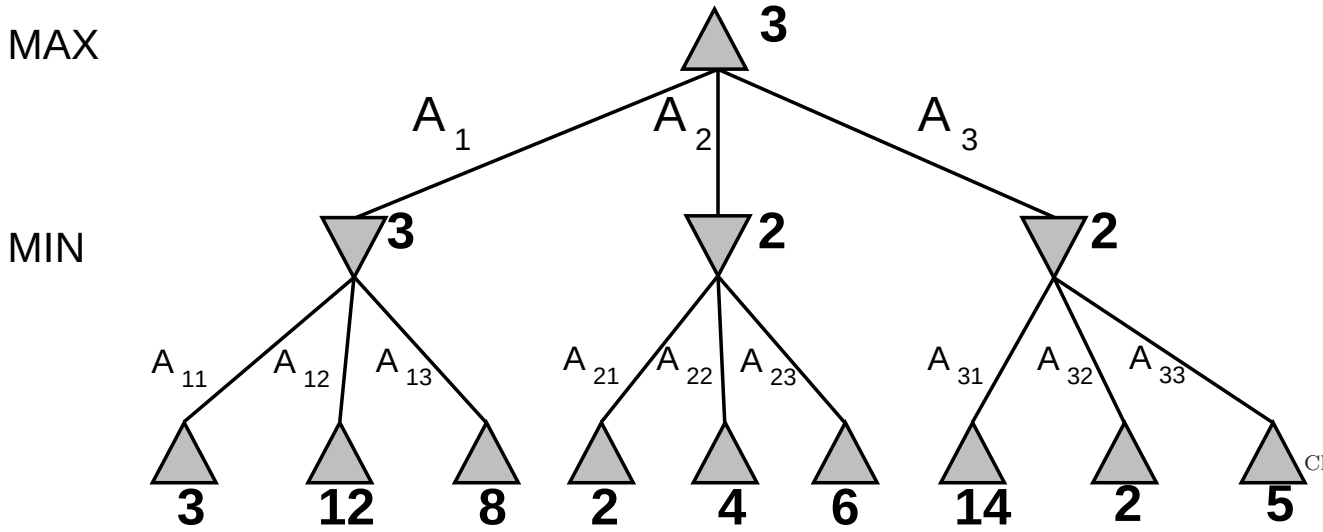
Minimax

Perfect play for deterministic, two-player, zero-sum, perfect-information games

Idea: choose move to position with highest minimax value
= best achievable utility against best possible opponent

$$\textsc{Minimax-Value}(s) = \begin{cases} \textsc{Utility}(s, \textsc{max}) & \text{if } \textsc{Terminal}(s) \\ \max_{m \in \textsc{Moves}(s)} \textsc{Minimax-Value}(\textsc{Result}(s,m)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{m \in \textsc{Moves}(s)} \textsc{Minimax-Value}(\textsc{Result}(s,m)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

E.g., 2-ply game:



Minimax

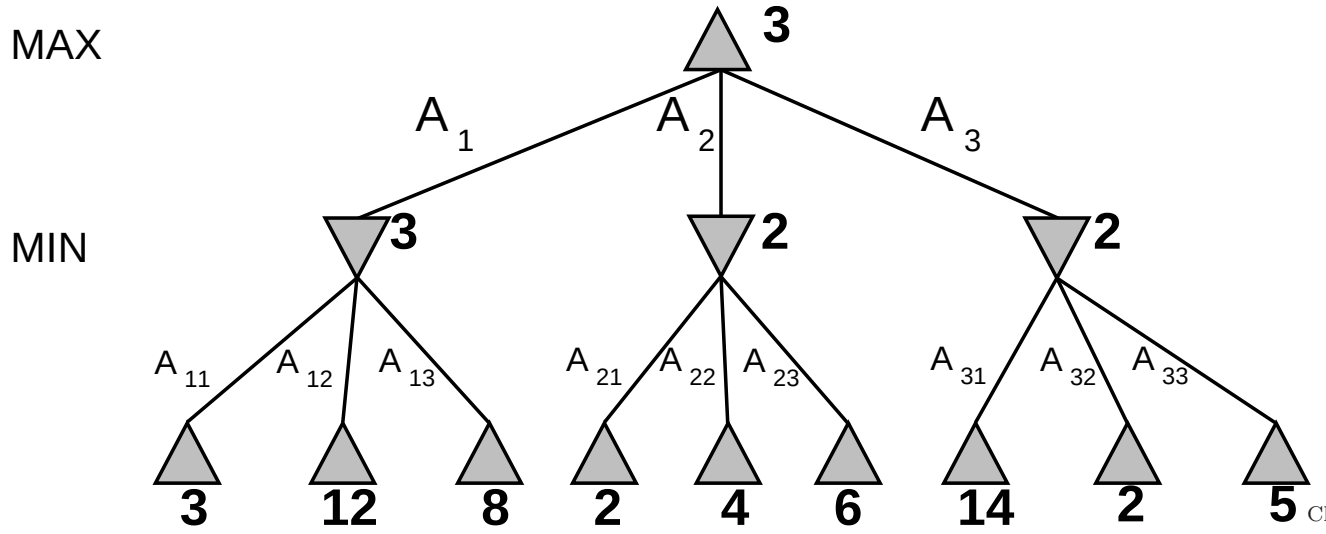
Computing the Minimax value:

1. Apply utility function to each leaf of the game tree
2. Back-up values from the leaves through inner nodes up to the root:
   (a) MIN node: compute the min of its children values
   (b) MAX node: compute the max of its children values
3. At the root: choose the move leading to the child of highest value

too memory consuming

"dumb way"

Better method: use a depth-first like approach to save space



Minimax algorithm

```
function MINIMAX-DECISION(state) returns a move
    inputs: state, current state in game

    v ← MAX-VALUE(state)
    return the move m in MOVES(state) with value v

---

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

---

function MIN-VALUE(state) returns a utility value
    if TERMINAL(state) then return UTILITY(state,MAX)
    v ← +∞
    for m in MOVES(state) do
        v ← MIN(v, MAX-VALUE(RESULT(state,m)))
    return v
```

Properties of minimax

Complete?? Yes, if tree is finite

Optimal?? Yes, against an optimal opponent. Otherwise??

Time complexity?? $O(b^m)$

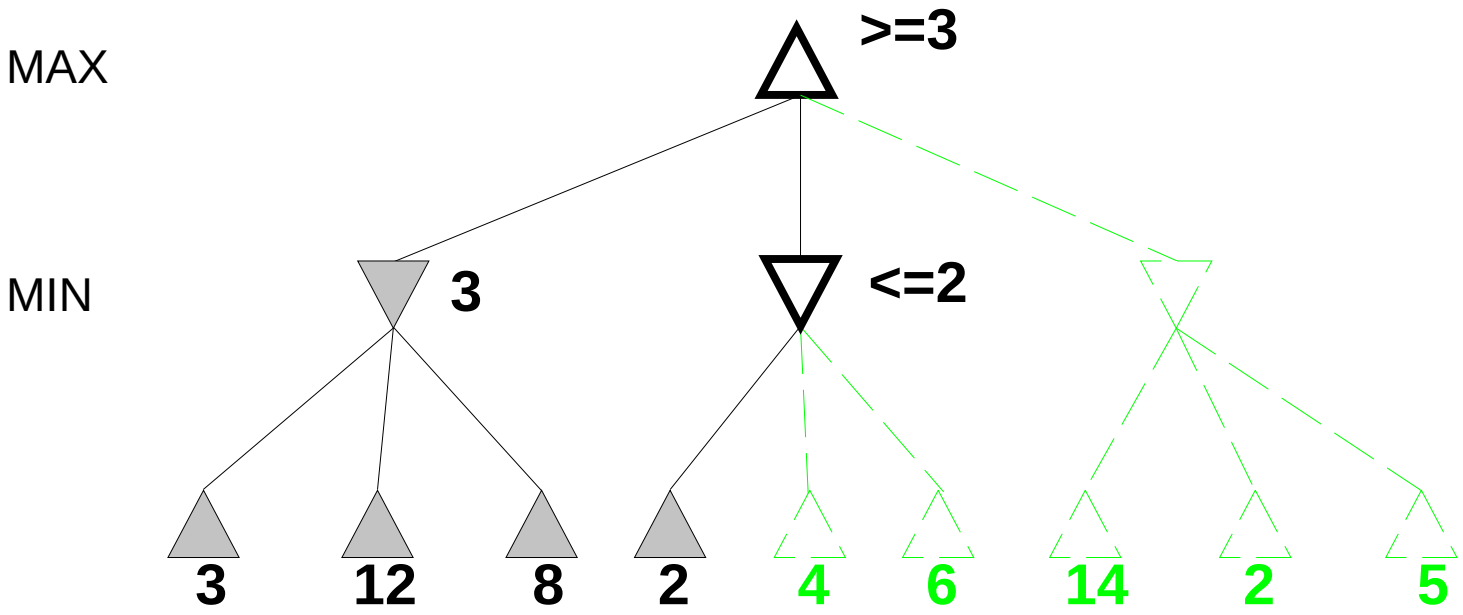
Space complexity?? $O(bm)$ (depth-first exploration)

For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games

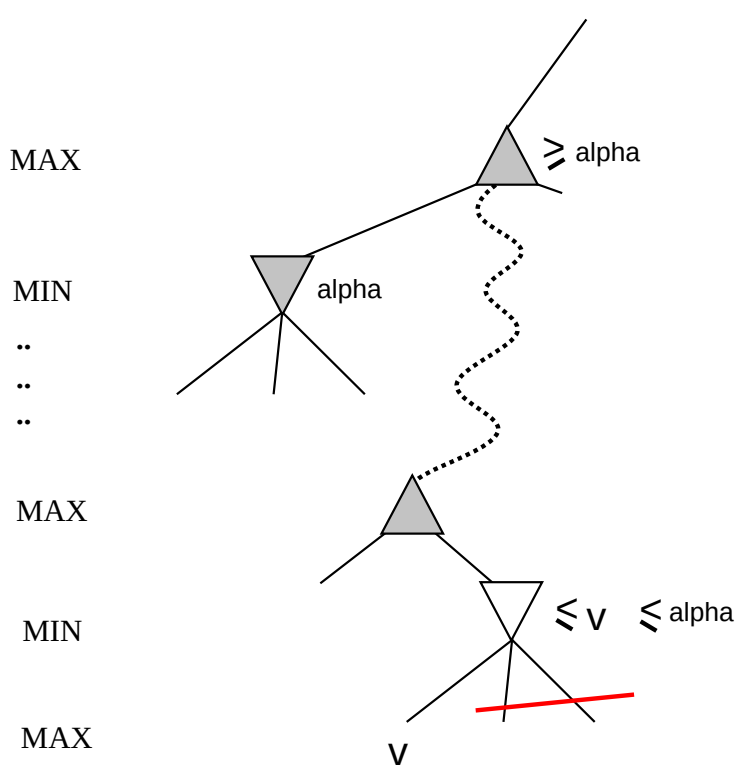
$\Rightarrow$ exact solution completely infeasible

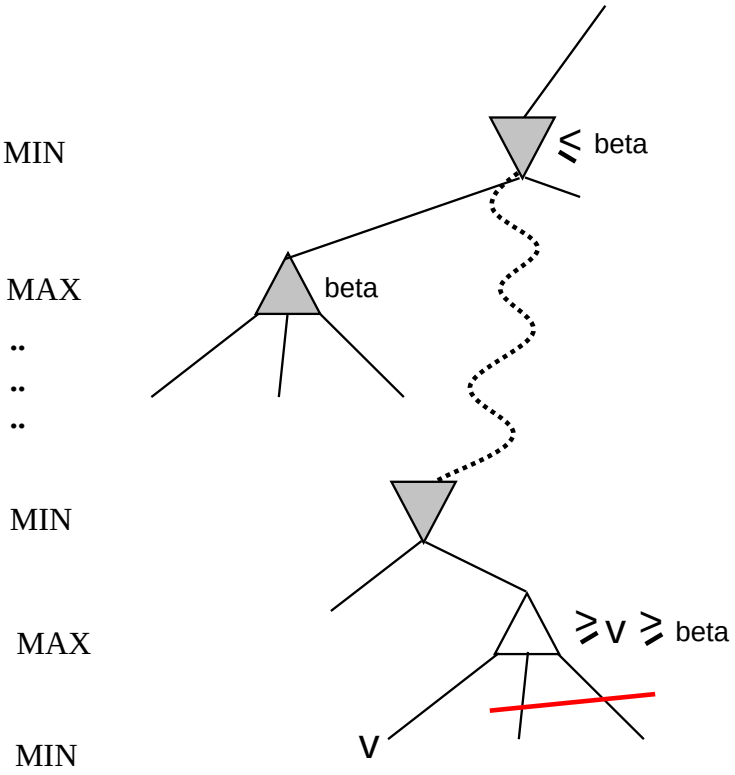
But do we need to explore every path?

Do we need to explore every path?



α–β pruning





α is the best value (to MAX, i.e. highest) found so far

If V is not better (greater) than α, MAX will avoid it $\Rightarrow$ prune that branch

Define β similarly for MIN

best - worst

The α–β algorithm

function ALPHA-BETA-DECISION($state$) **returns** a move
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the move m in MOVES($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  inputs: $state$, current state in game
    α, the value of the best choice for MAX so far
    β, the value of the best choice for MIN so far
  if TERMINAL($state$) **then return** UTILITY($state$,MAX)
  $v \leftarrow -\infty$
  for m in MOVES($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(m,s), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

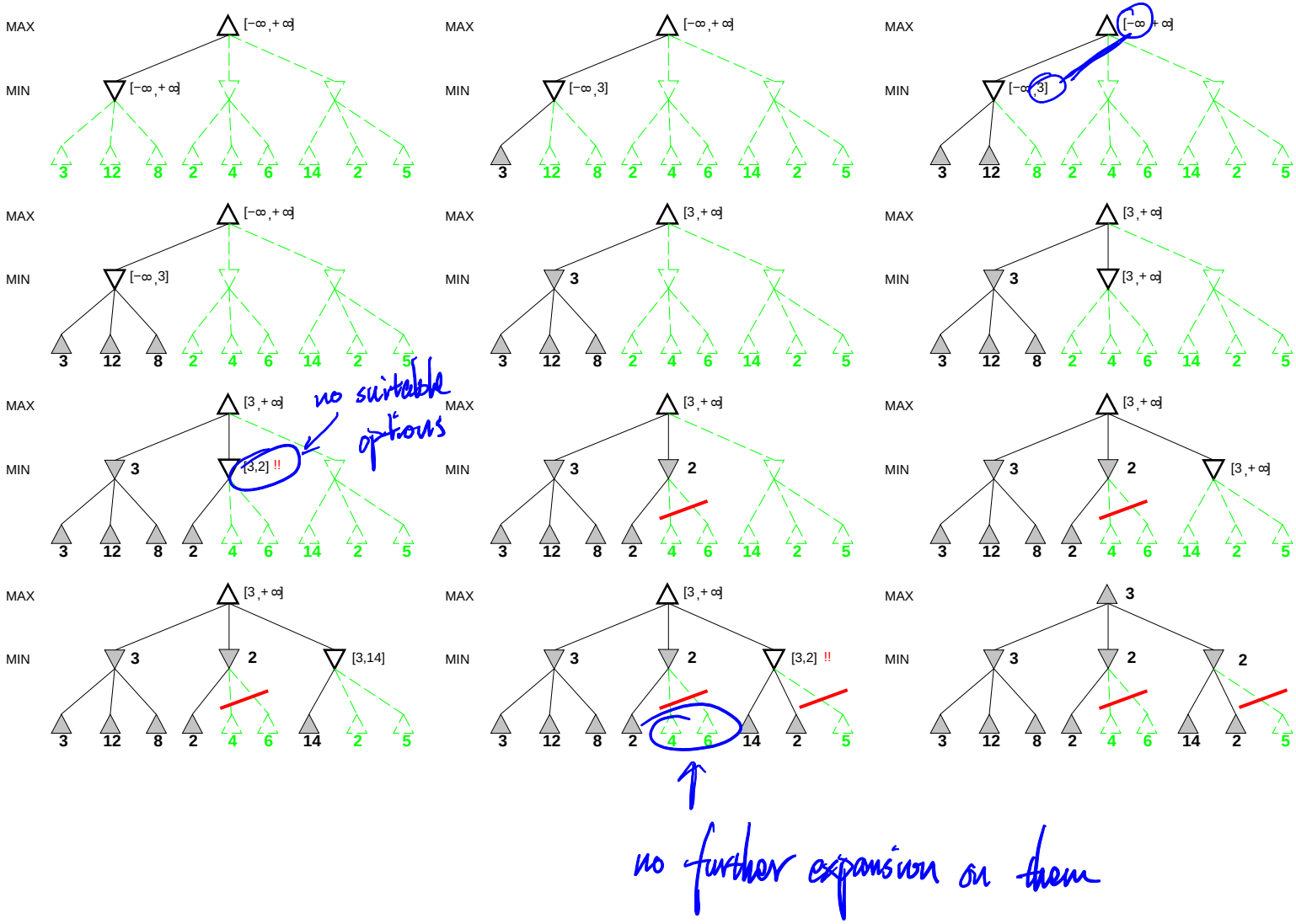
---

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  same as MAX-VALUE but with roles of α, β reversed

IDEA:
→ α : the best MAX value on path to current node
→ β : the best MIN value on path to current node
– Some values outside the interval $[\alpha, \beta]$ can be pruned

ALGORITHM:
→ Node passes its current values for α and β to its children in turn
→ Child passes back up its value to node
→ Node updates its current value v (max or min) with child's value
→ Node checks whether $v \leq \alpha$ (MIN) or $v \geq \beta$ (MAX)
• if so, child's siblings can be pruned & v returned to Parent
• otherwise β (MIN) or α (MAX) is updated

α–β pruning example



Remarks on and Properties of α–β

Minimax with Alpha-Beta pruning is used with depth-first exploration of the game tree (not complete generation!).

A parent node passes its current values for α and β to its children in turn. A child passes back up its value v to the parent. The parent compares v to α (MIN) or β (MAX) to decide whether to prune the child's sibling and if so return v to the parent. Otherwise, it updates its current values for α (MAX) or β (MIN) using v and go on.

Pruning **does not** affect final result.

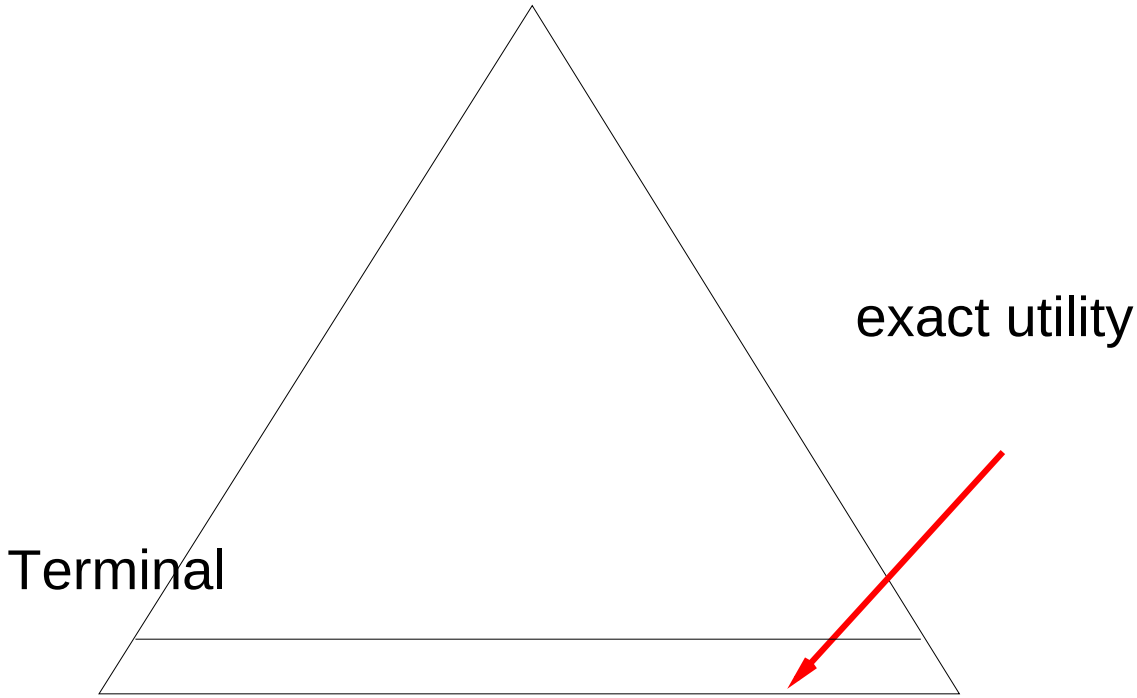
Good move ordering improves effectiveness of pruning. Perfect ordering (unachievable): increasing order for MAX and decreasing order for MIN.

With perfect ordering, the time complexity is asymptotically $O(b^{m/2}) \Rightarrow$ **doubles** solvable depth (random ordering yields $O(b^{3m/4})$).

Unfortunately, 35^{50} is still impossible!

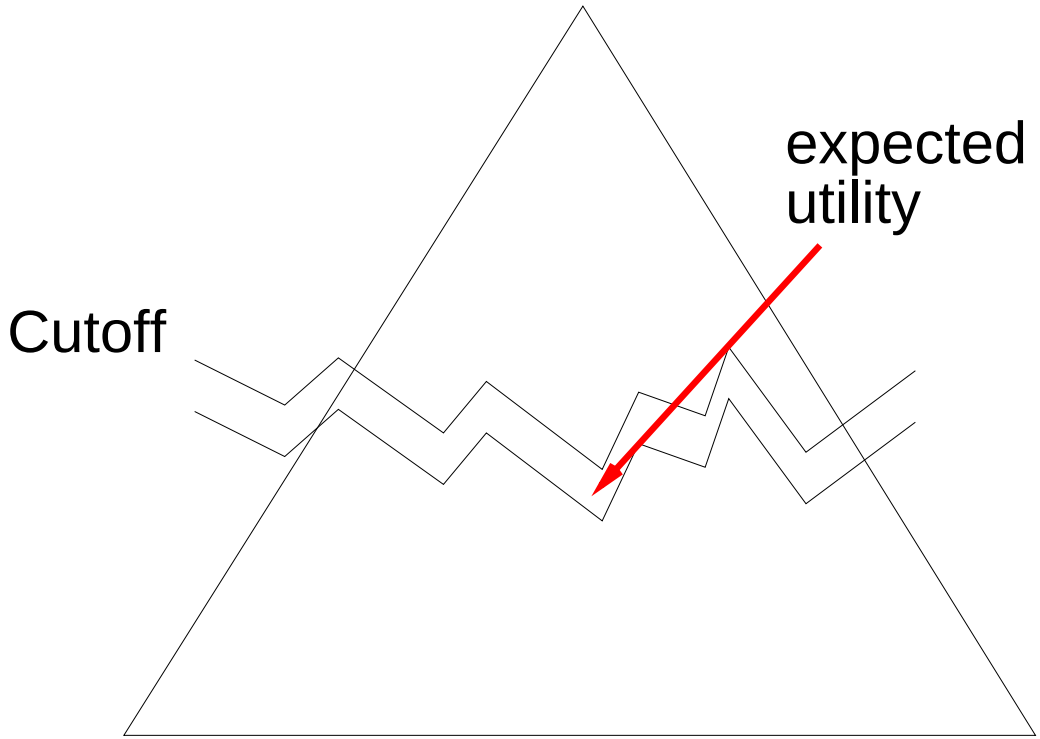
Imperfect decisions in real-time

Approach: **limit search depth and estimate expected utility**



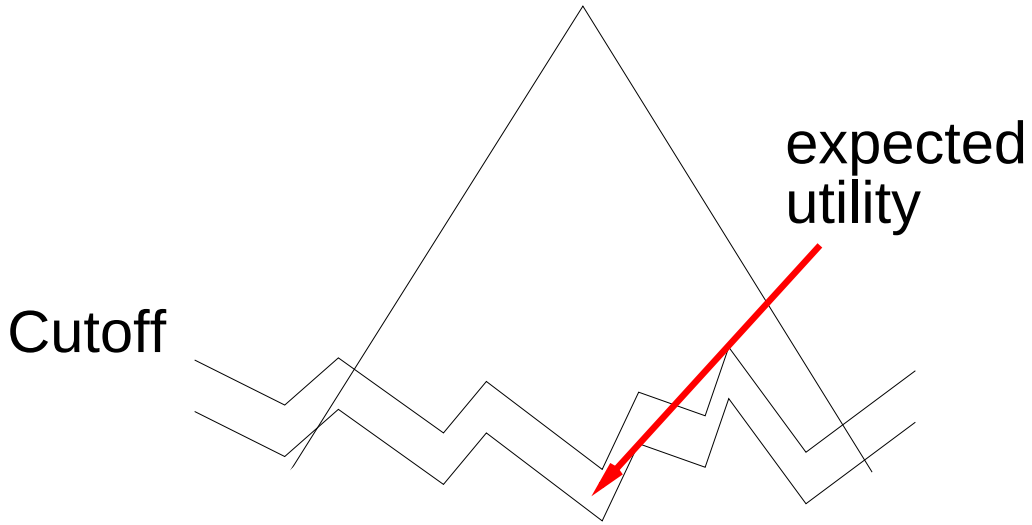
Imperfect decisions in real-time

Approach: **limit search depth and estimate expected utility**



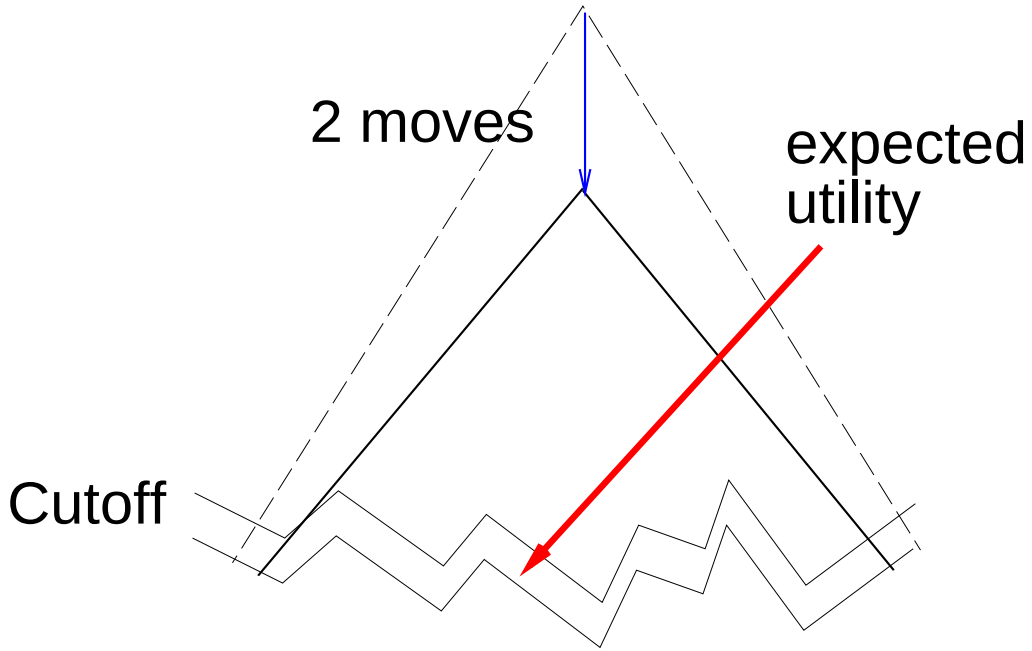
Imperfect decisions in real-time

Approach: **limit search depth and estimate expected utility**



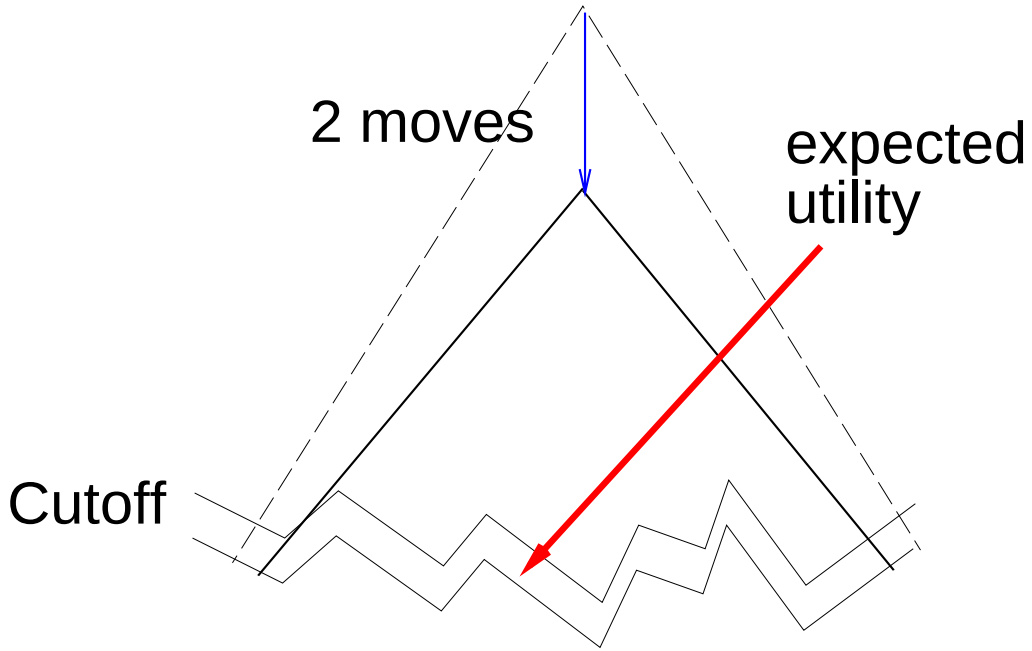
Imperfect decisions in real-time

Approach: **limit search depth and estimate expected utility**



Imperfect decisions in real-time

Approach: **limit search depth and estimate expected utility**



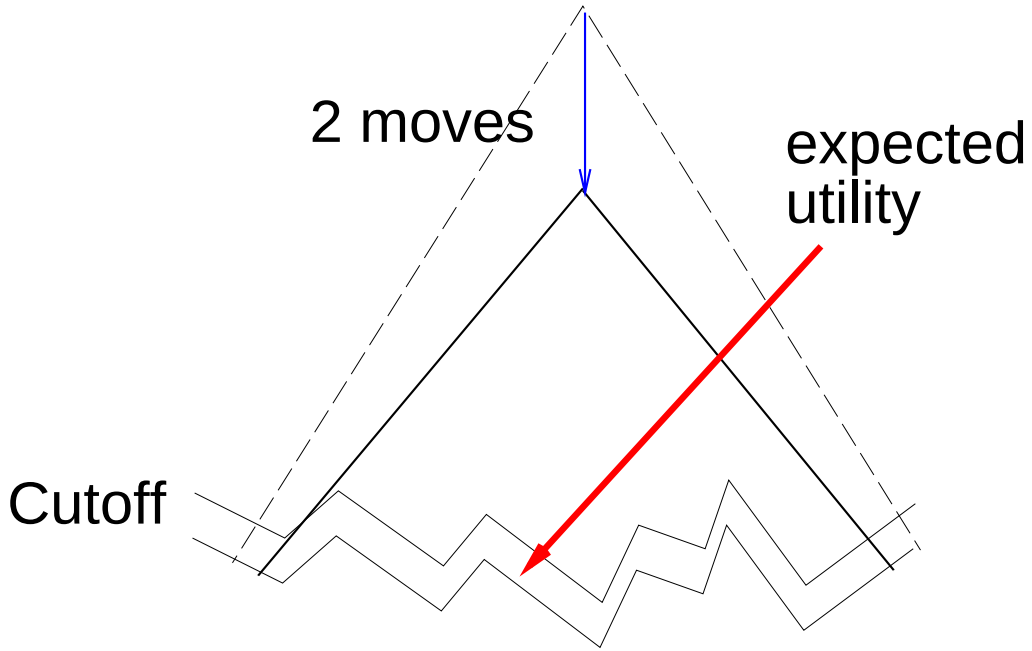
Suppose we have 100 seconds, explore 10^4 nodes/second

$\Rightarrow 10^6$ nodes per move $\approx 35^{8/2}$

$\Rightarrow \alpha$–β reaches depth 8 $\Rightarrow$ pretty good chess program

Imperfect decisions in real-time

Approach: **limit search depth and estimate expected utility**



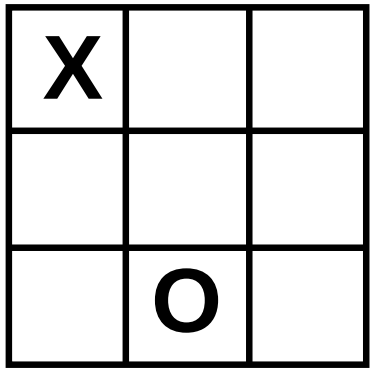
Changes to minimax: replace TERMINAL-TEST(s) with CUTOFF(s,d) and UTILITY(s,p) with EVAL(s,p) to estimate expected utility.

Changes to Minimax

- **Use CUTOFF test instead of TERMINAL test**
  - CUTOFF(s, d): true iff the state s encountered at depth d in the tree must be considered as a leaf (or s is terminal).
  - e.g., depth limit, estimated number of nodes expanded
  - perhaps add quiescence search
- **Use EVAL instead of UTILITY**
  - EVAL(s,p) i.e., evaluation function that estimates the expected utility of cutoff state s wrt player p, and correlates with chances of winning
  - should order the *terminal* states in the same way as UTILITY
  - should not take too long

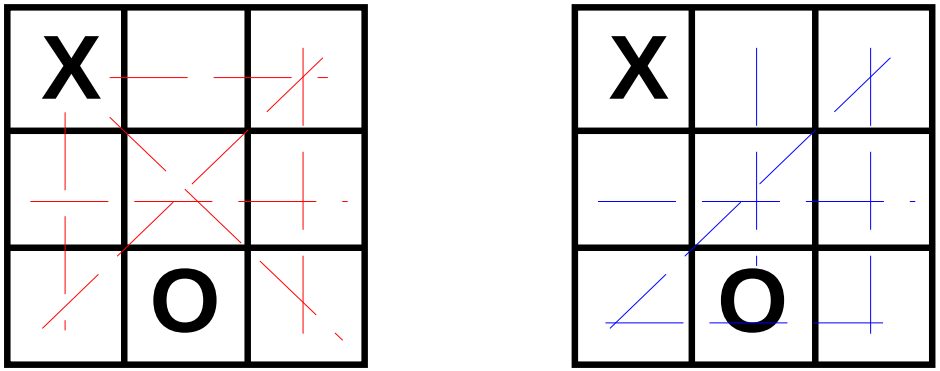
$$\text{D-MINIMAX-VALUE}(s, d) = \begin{cases} \text{EVAL}(s, \text{MAX}) & \text{if CUTOFF}(s, d) \\ \max_{m \in \text{MOVES}(s)} \text{D-MINIMAX-VALUE}(\text{RESULT}(s, m), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{m \in \text{MOVES}(s)} \text{D-MINIMAX-VALUE}(\text{RESULT}(s, m), d+1) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Evaluation functions



What would be a good evaluation function for tic tac toe??

Evaluation functions

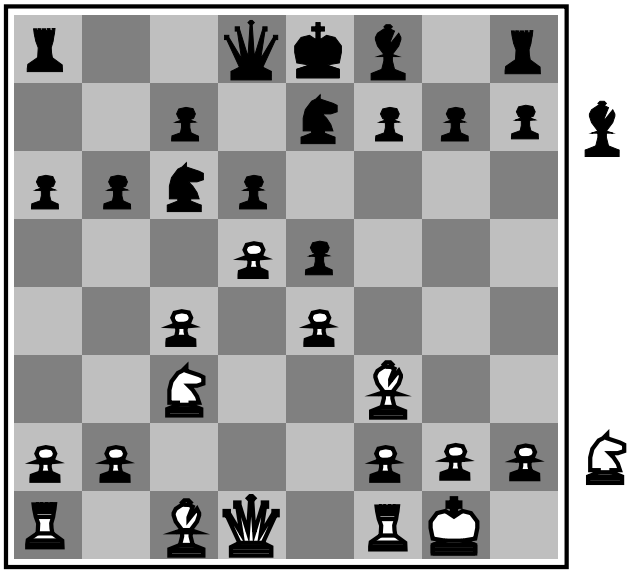


What would be a good evaluation function for tic tac toe??

EVAL(s,p) = winning-patterns(s,p) - winning-patterns(OPPONENT(s,p))

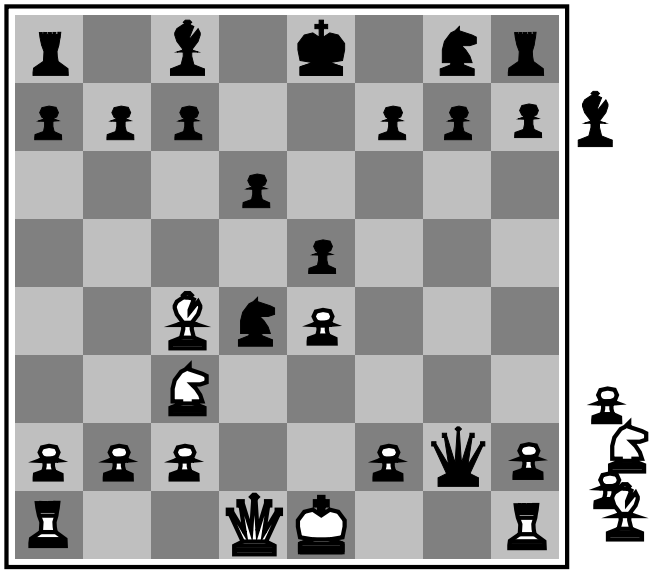
EVAL(s,X) = 6 - 5 = 1

Evaluation functions



Black to move

White slightly better



White to move

Black winning

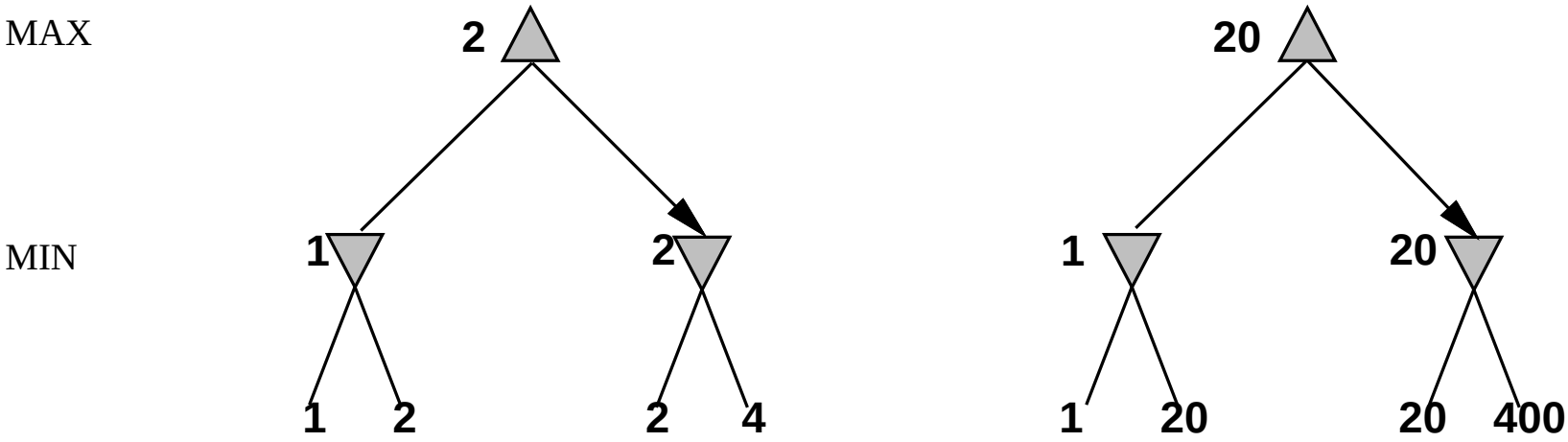
For chess, typically linear weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

// usually use ML to learn the weights

e.g., $w_2 = 5$ with
$f_2(s)$ = (number of white castles) – (number of black castles), etc.

Observation: Exact values don't matter



Behaviour is preserved under any **monotonic** transformation of EVAL

Only the order matters:

payoff in deterministic games acts as an ordinal utility function

Other techniques to tame complexity

- ♢ Singular extensions
- ♢ Iterative deepening
- ♢ Symmetry prunning
- ♢ Pattern databases
- ♢ Monte carlo sampling

- Deep Learning
- Monte Carlo Tree Search (MCTS)

// Random Path

P : move prob.
N : times moves taken
Q : move value

AlphaGO

[MCTS + Deep Learning]

Replace depth-first search by MCTS exploration of game tree

→ in state s, select move a maximizing $Q(a,s) + P(a,s)/(1+N(a,s))$
→ Leaf states are evaluated using $f(l) = \lambda e(l) + (1-\lambda) v(t)$
→ terminal state t obtained by sampling from l with Proba $P'(a,s)$
→ after each simulation, $N(a,s)$ & $Q(a,s)$ are updated along path

Deterministic games in practice

Checkers: Chinook ended 40-year-reign of world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board (500 billion states). In 2007, checkers became the largest game to be completely solved (with 500×10^{20} states!).

Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per sec, i.e. 300 billion per move (depth 14), uses very sophisticated evaluation (8000 features), and singular extensions to extend some search lines up to 40 ply.

Go: branching factor $b > 300$ made this more challenging. Monte Carlo Tree Search (MCTS) is the method of choice. Zen defeated a 9 Dan in 2013. In 2016 AlphaGo made a surprising 4-1 win against Lee Sedol, using deep learning to learn a value function and policy to guide MCTS.

Poker: the next big thing!

Summary

A **Game** is defined by an initial state, a successor function, a terminal test, and a utility function

The **minimax** algorithm select optimal actions for two-player zero-sum games of perfect information by a depth first exploration of the game-tree

Alpha-beta pruning does not compromise optimality but increases efficiency by eliminating provably irrelevant subtrees

It is not feasible to consider the whole game tree (even with alpha-beta), so we need to **cut the search off** at some point and apply an **evaluation function** that gives an estimate of the expected utility of a state