CS885 Reinforcement Learning
Lecture 13c: June 13, 2018

Adversarial Search

[RusNor] Sec. 5.1-5.4

Outline

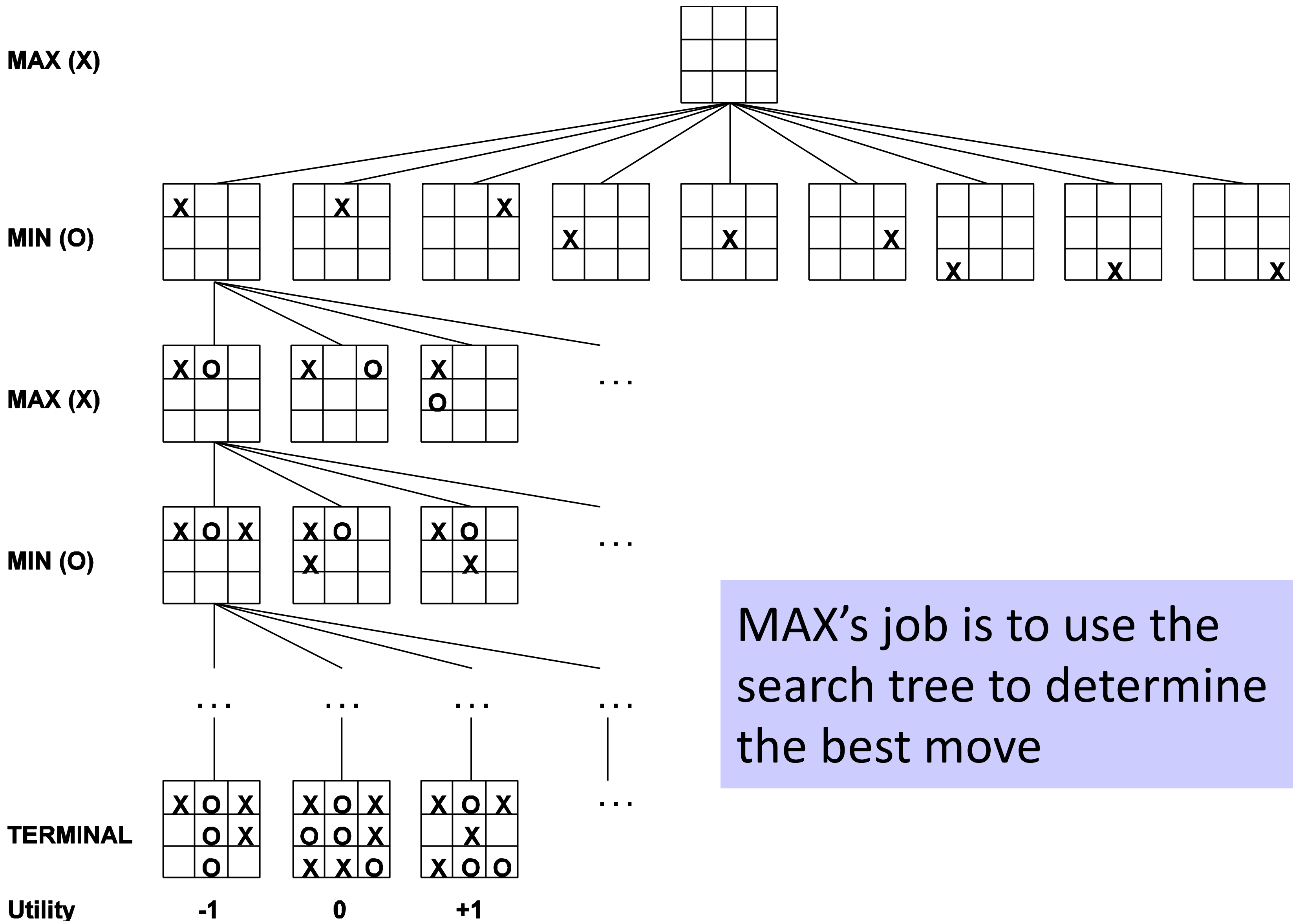
- Minimax search
- Evaluation functions
- Alpha-beta pruning

Game search challenge

- What makes game search challenging?
  - There is an opponent!
  - The opponent is malicious – it wants to win (i.e. it is trying to make you lose)
  - We need to take this into account when choosing moves
    - Simulate the opponent's behaviour in our search

- Notation: One player is called **MAX** (who wants to maximize its utility) and one player is called **MIN** (who wants to minimize its utility)

Example: Tic-Tac-Toe



MAX’s job is to use the search tree to determine the best move

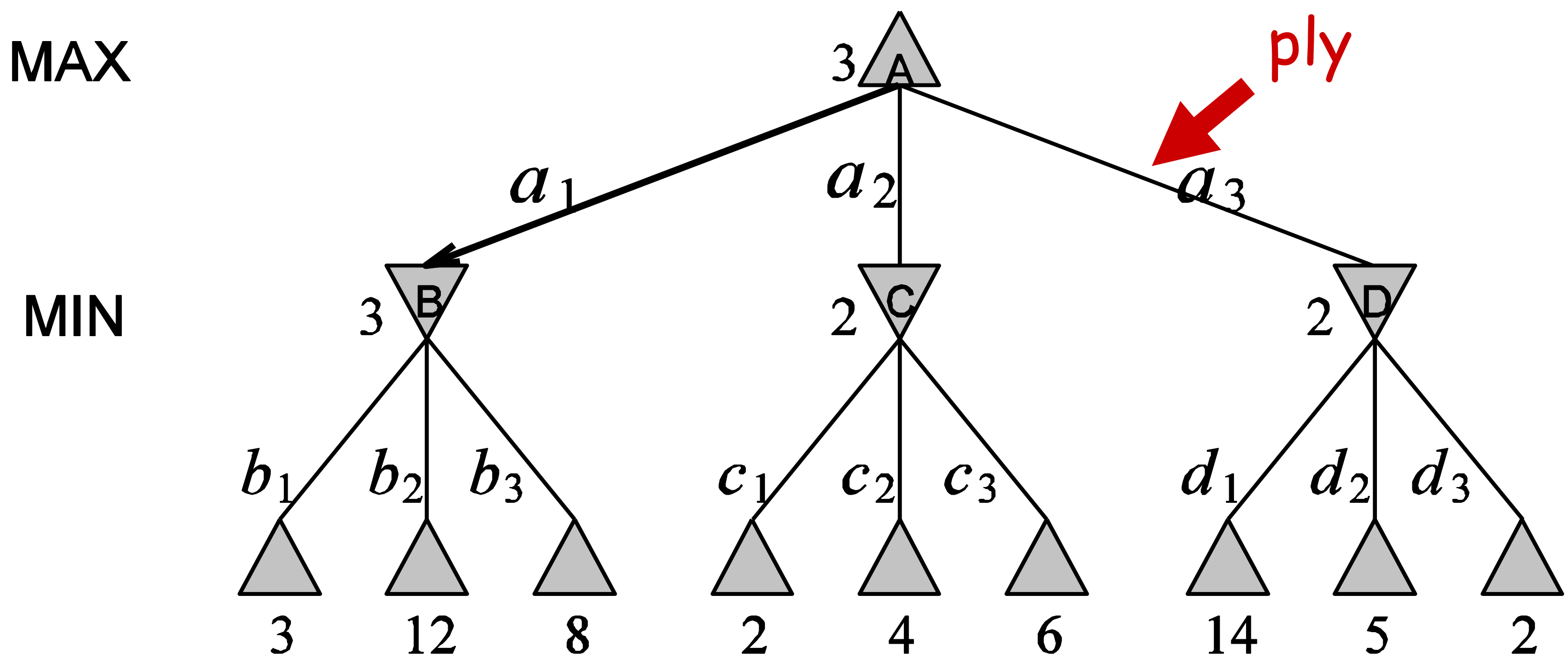
Optimal strategies

- Want to find the optimal strategy
  - One that leads to outcomes at least as good as any other strategy, given that MIN is playing optimally
  - Equilibrium (game theory)
  - Zero-sum game of perfect information

Minimax Value

MINIMAX-VALUE(n) =

- Utility(n) if n is a terminal state
- $\text{Max}_{s \in \text{Succ}(n)}$ MINIMAX-VALUE(s) if n is a MAX node
- $\text{Min}_{s \in \text{Succ}(n)}$ MINIMAX-VALUE(s) if n is a MIN node



Minimax algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*
  $v \leftarrow$ MAX-VALUE(*state*)
  return the *action* in SUCCESSORS(*state*) with value v

Returns action corresponding to best possible move

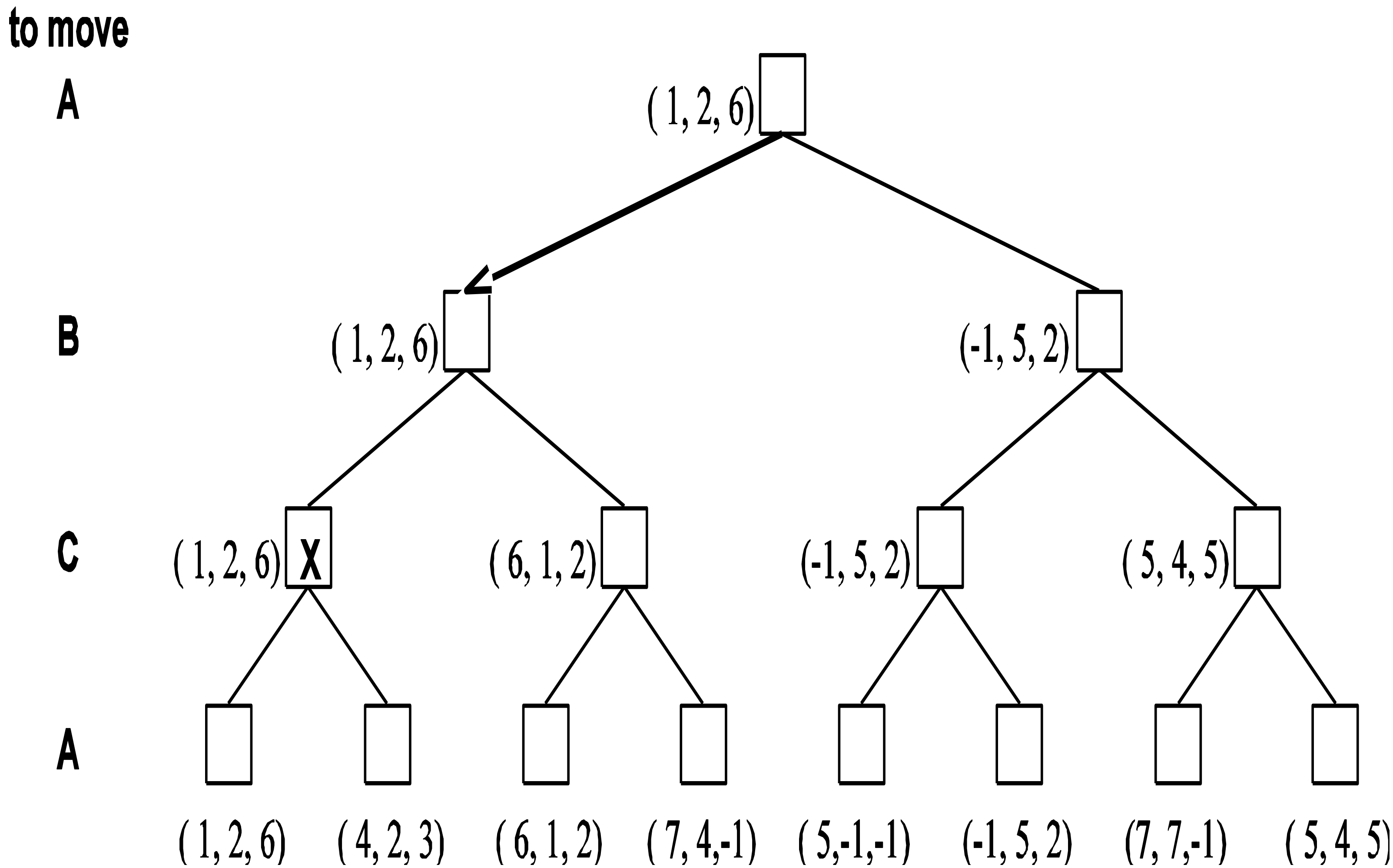
function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(s))
  return v

Properties of Minimax

- Time complexity:
  – $O(b^d)$ Where b is branching factor and d is depth of the tree
- Space complexity:
  – $O(bd)$ just need to keep in memory the current branch with its children

Minimax and multi-player games



Chess

- Can we write a a minimax program that will play chess reasonably well?
  - For chess $b \approx 35$ and $d \approx 100$
  - Do we really need to look at all those nodes?

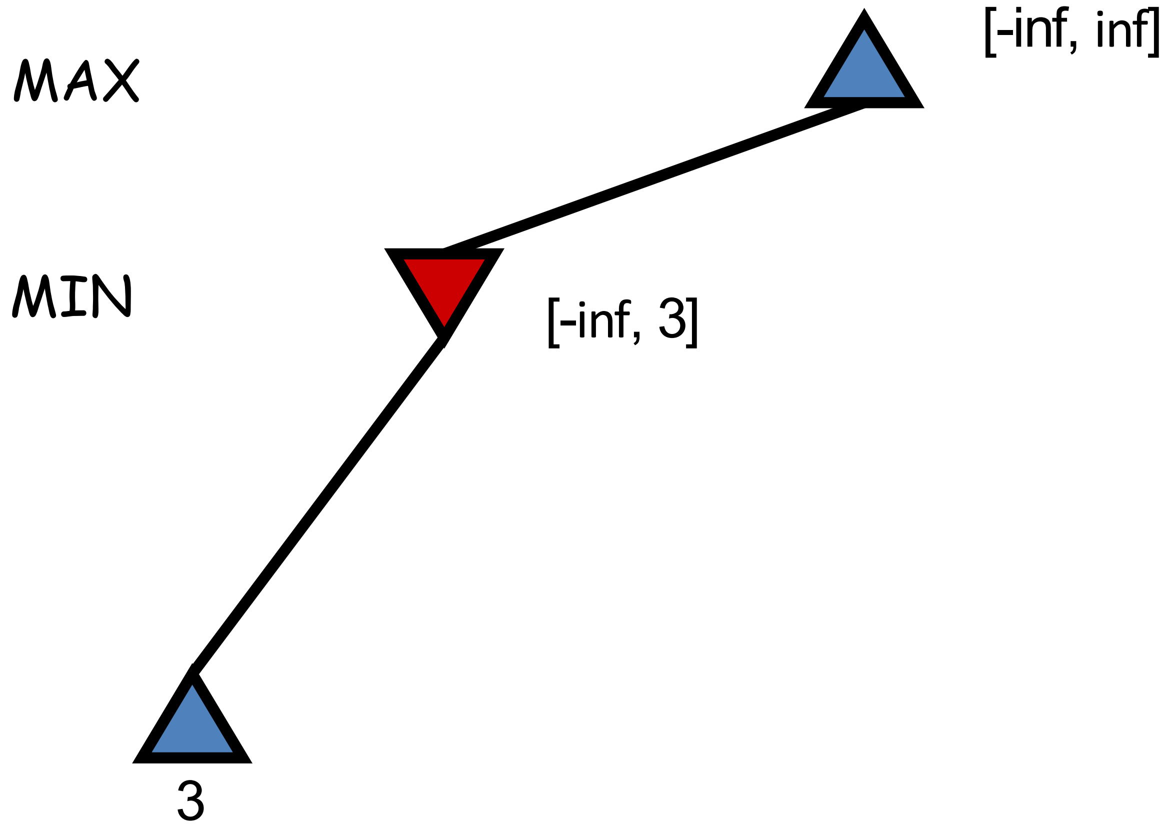
Alpha-Beta Pruning

- No!
  - If we are smart (and careful) we can do **pruning**
    - Eliminate large parts of the tree from consideration
- Alpha-Beta pruning applied to a minimax tree
  - Returns the same decision as minimax
  - Prunes branches that cannot influence final decision

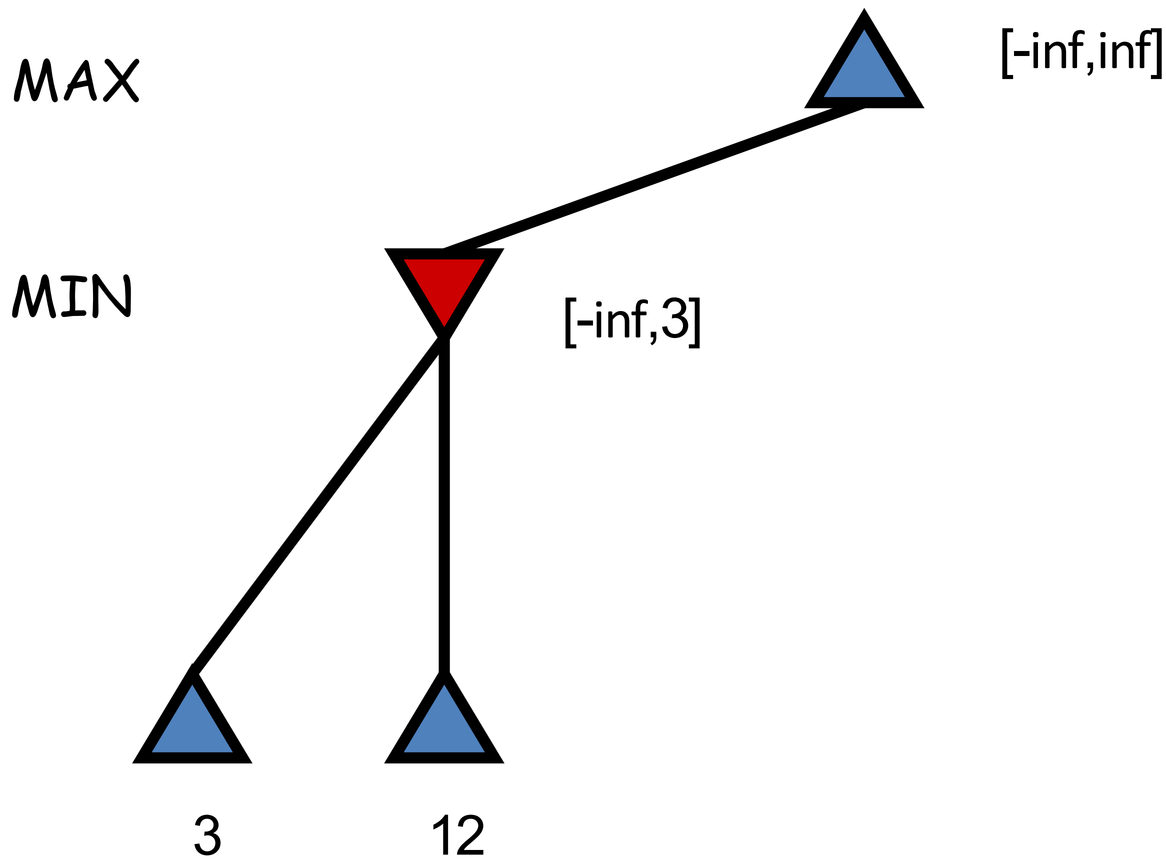
Alpha-Beta Pruning

- Alpha:
  - Value of best (highest value) choice we have found so far on the path for MAX
- Beta:
  - Value of best (lowest value) choice we have found so far on path for MIN
- Update alpha and beta as search continues
- Prune as soon as the value of the current node is known to be worse than current alpha or beta values for MAX or MIN

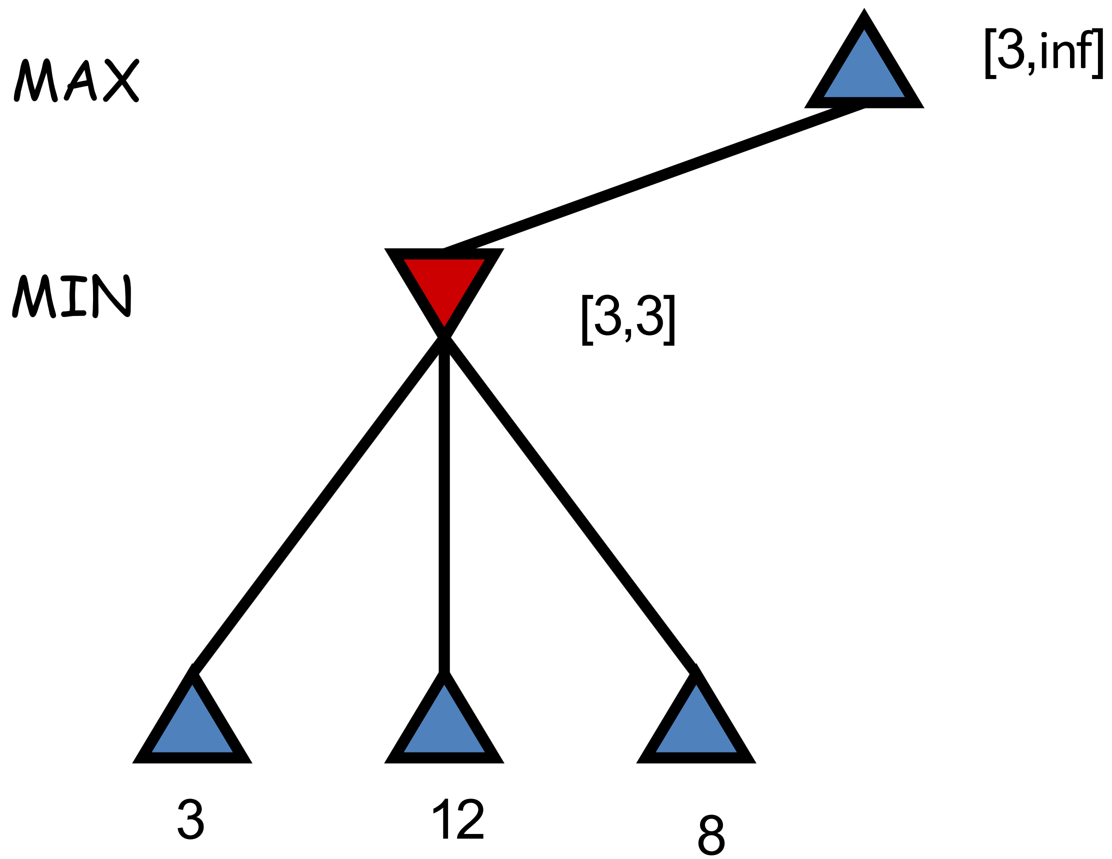
Alpha-Beta example



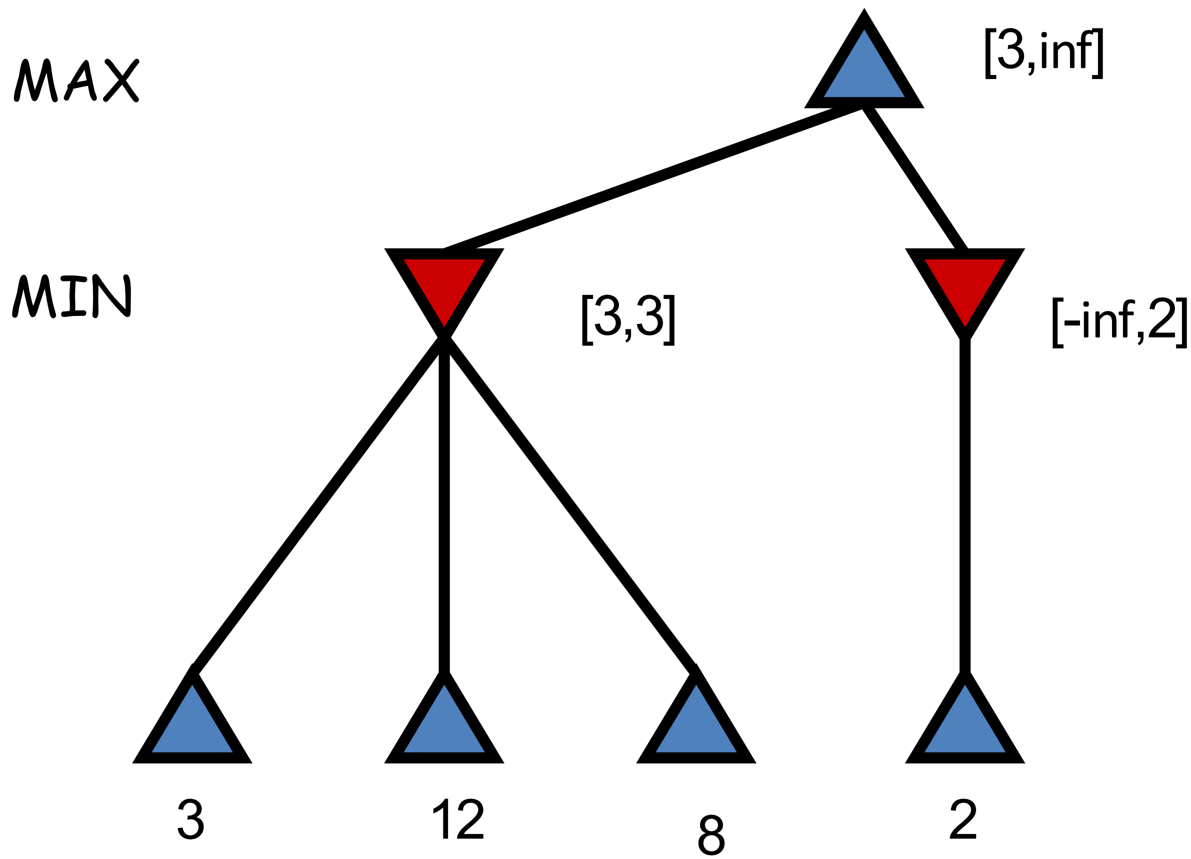
Alpha-Beta example



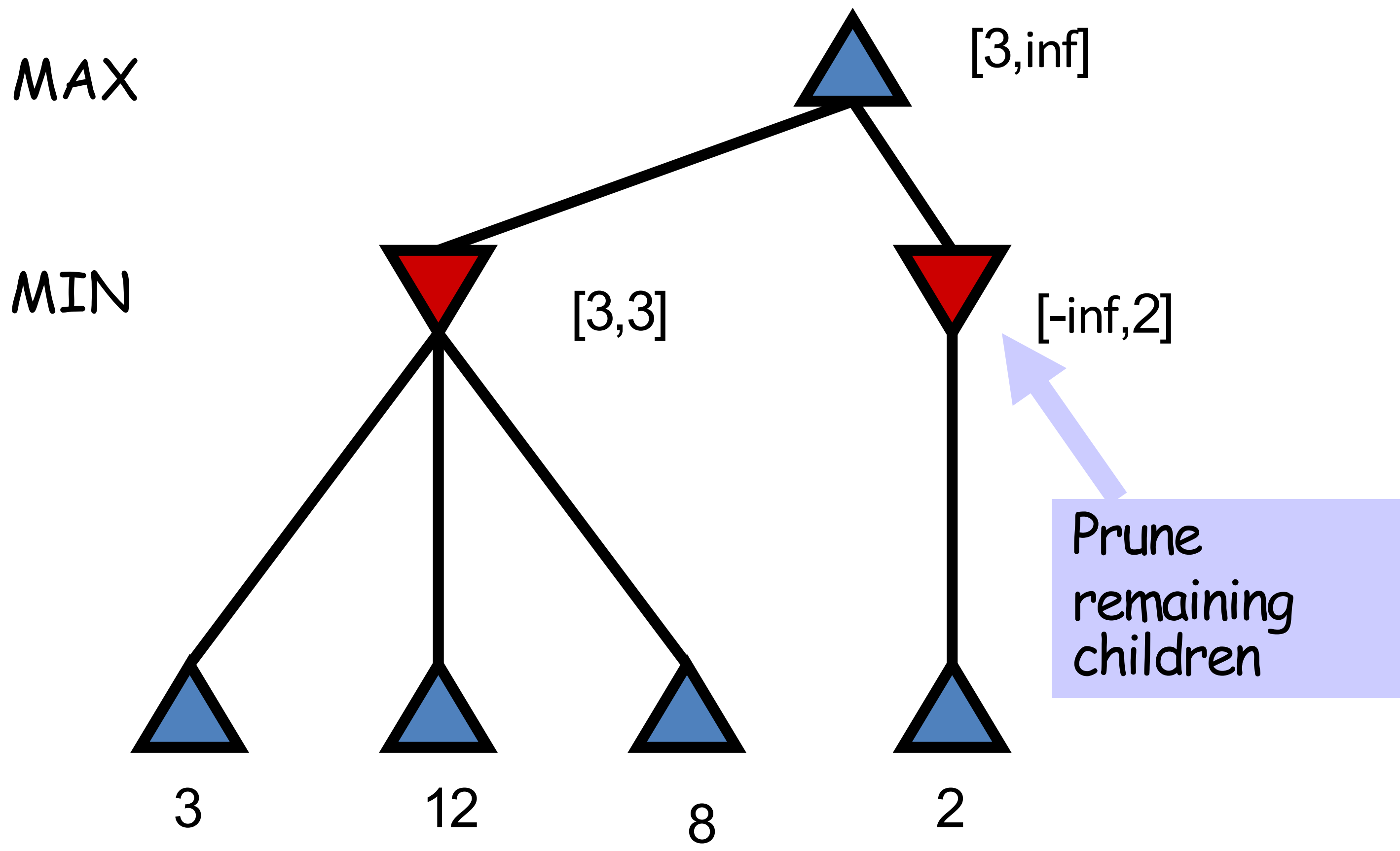
Alpha-Beta example



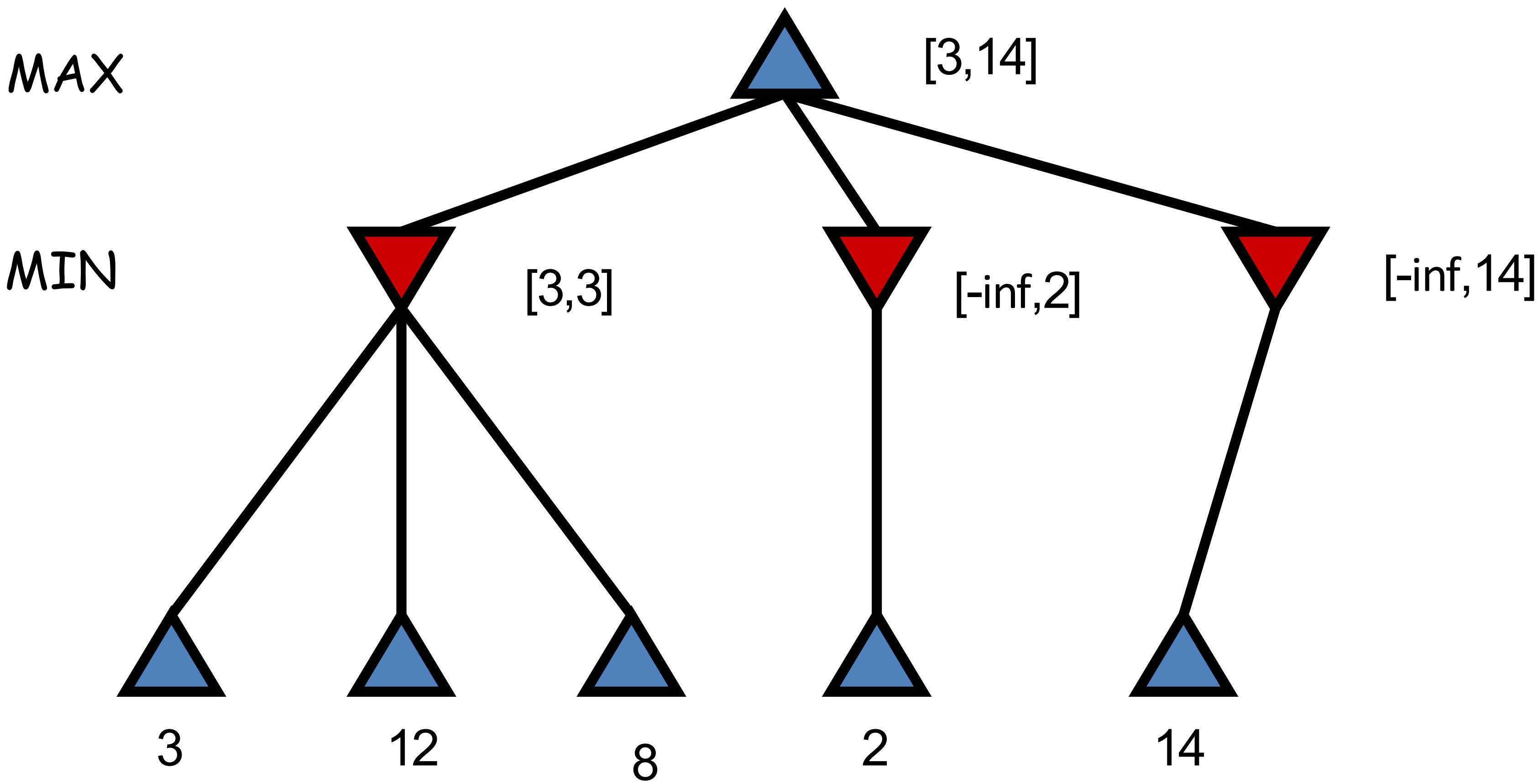
Alpha-Beta example



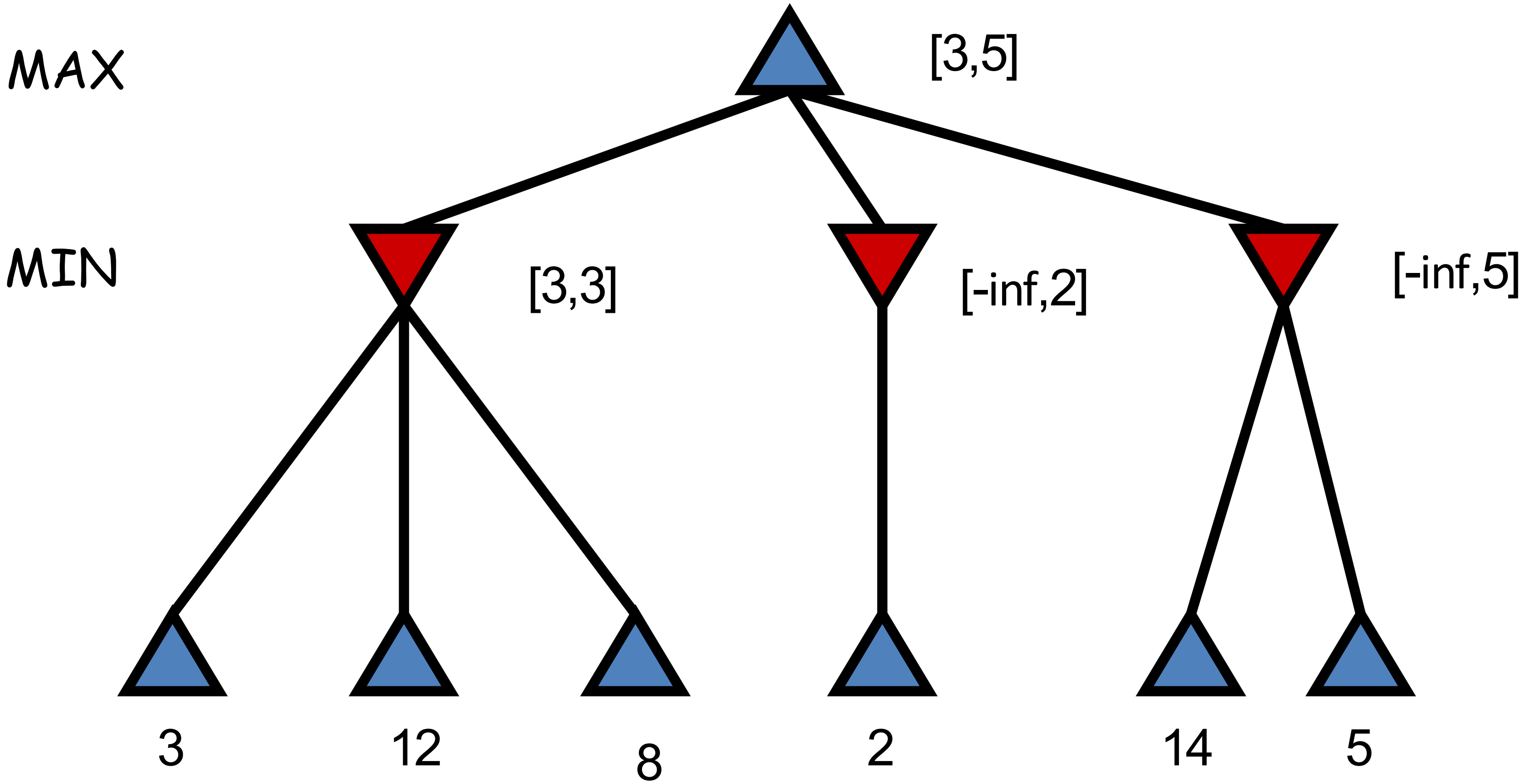
Alpha-Beta example



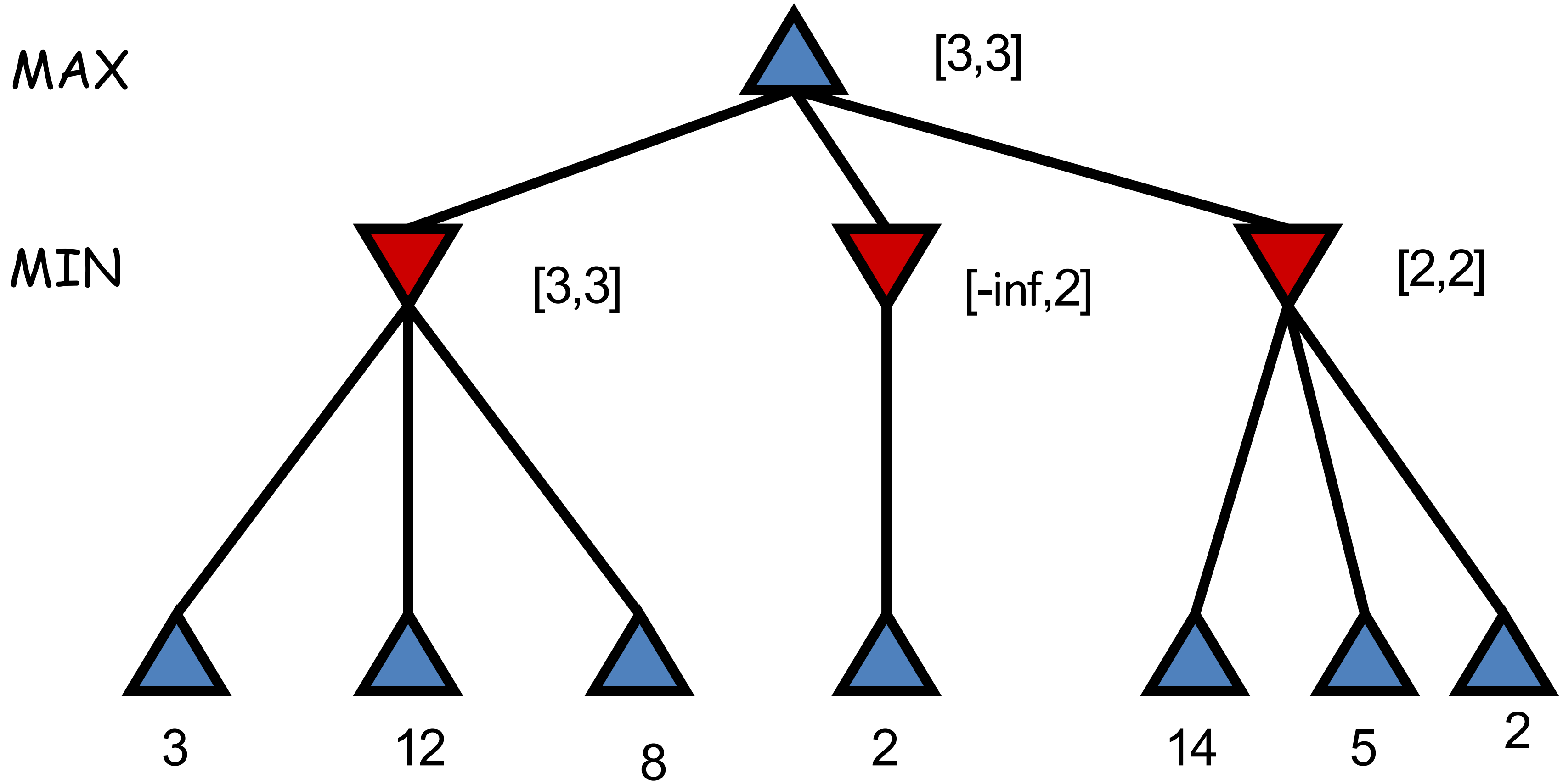
Alpha-Beta example



Alpha-Beta example



Alpha-Beta example



Properties of Alpha-Beta

- Pruning does not affect the final result
  - Prune parts of the tree that would never be reached in actual play
- The order in which moves are evaluated are important
  - A bad move ordering will prune nothing
  - A perfect node ordering can reduce time complexity to $O(b^{d/2})$

Real-time decisions

- Alpha-beta can be a huge improvement over minimax
  - Still not good enough as we need to search all the way to terminal states for at least part of the search space
  - Need to make a decision about a move quickly
- Heuristic evaluation function + cutoff test

Evaluation functions

- Apply an evaluation function to a state
  - If terminal state, function returns actual utility
  - If non-terminal, function returns estimate of the expected utility (i.e. the chance of winning from that state)
  - Function must be fast to compute

Evaluation functions

- Evaluation functions can be given by the designer of the program (using expert knowledge) or learned from experience

- If features can be judged independently, a weighted linear function is good
  - $w_1 f_1(s)+w_2 f_2(s)+\ldots+w_n f_n(s)$ with s as board state
- Neural networks are commonly used today

Cutting off search

- Instead of searching until we find a terminal state, we can cut search sooner and apply the evaluation function
- When?
  - Arbitrarily (but deeper is better)
  - Quiescent states
    - States that are “stable” – not going to change value (by a lot) in the near future
  - Singular extensions
    - Searching deeper when you have a move that is “clearly better” (i.e. moving the king out of check)
    - Can be used to avoid the **horizon effect**

Cutting off search

- How deep do we need to search?
  - Novice chess human player
    - 5-ply (minimax)
  - Master chess human player
    - 10-ply (alpha-beta)
  - Grandmaster chess human player
    - 14-ply + a fantastic evaluation function, opening and endgame databases