Adversarial Search

Games vs. search problems

- "Unpredictable" opponent → specifying a move for every possible opponent reply
-
- Time limits → unlikely to find goal, must approximate
-

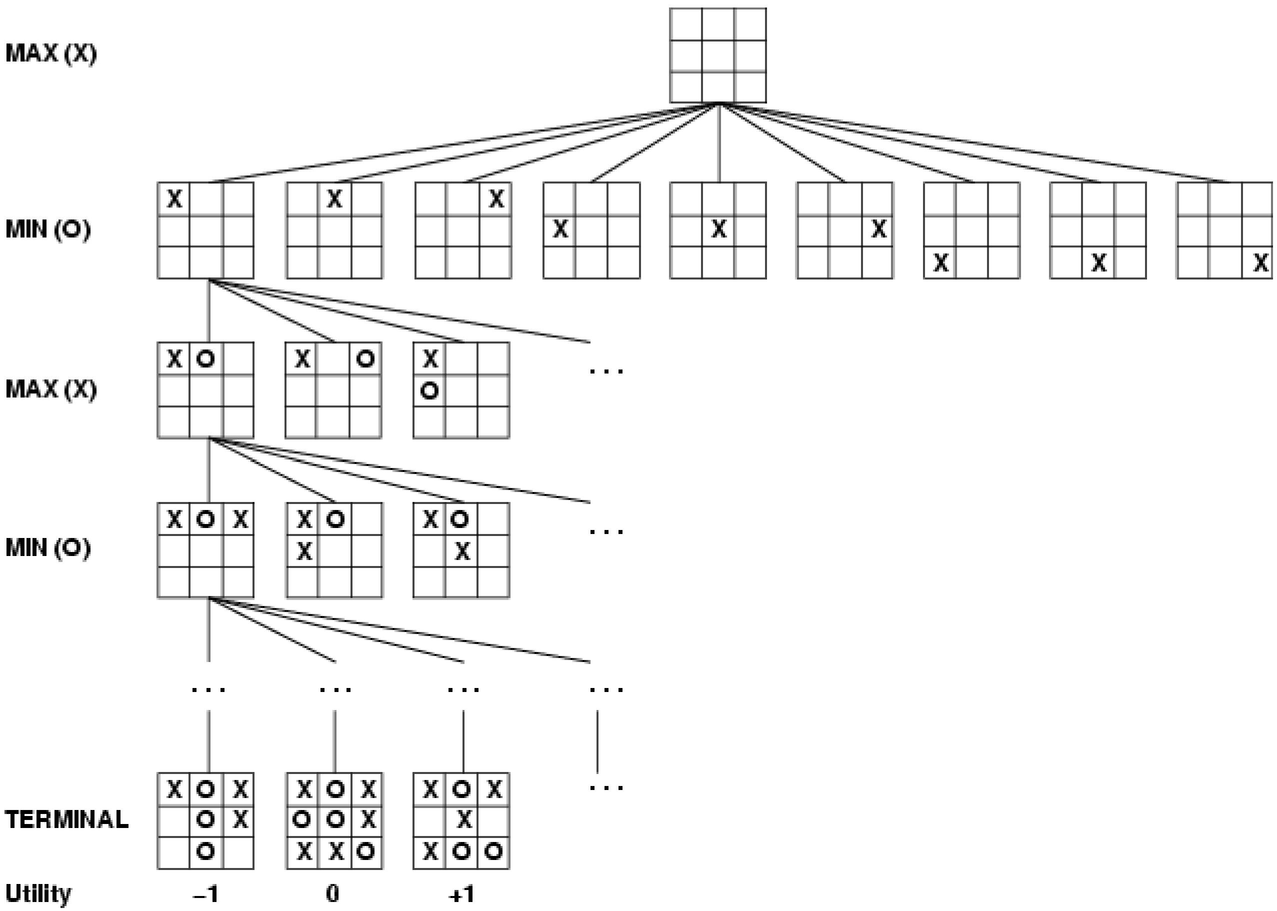
Game search

- Game-playing programs developed by AI researchers since the beginning of the modern AI era
  - Programs playing chess, checkers, etc (1950s)
- Specifics:
  - Sequences of player's decisions we control
  - Decisions of other player(s) we do not control
- Contingency problem: many possible opponent's moves must be "covered" by the solution
- Opponent's behavior introduces uncertainty
- Rational opponent – maximizes its own utility (payoff) function

Game Search Problem

- Problem formulation
  - Initial state: initial board position + whose move it is
  - Operators: legal moves a player can make
  - Goal (terminal test): game over?
  - Utility (payoff) function: measures the outcome of the game and its desirability
- Search objective:
  - Find the sequence of player's decisions (moves) maximizing its utility (payoff)
  - Consider the opponent's moves and their utility

Game tree (2-player, deterministic, turns)

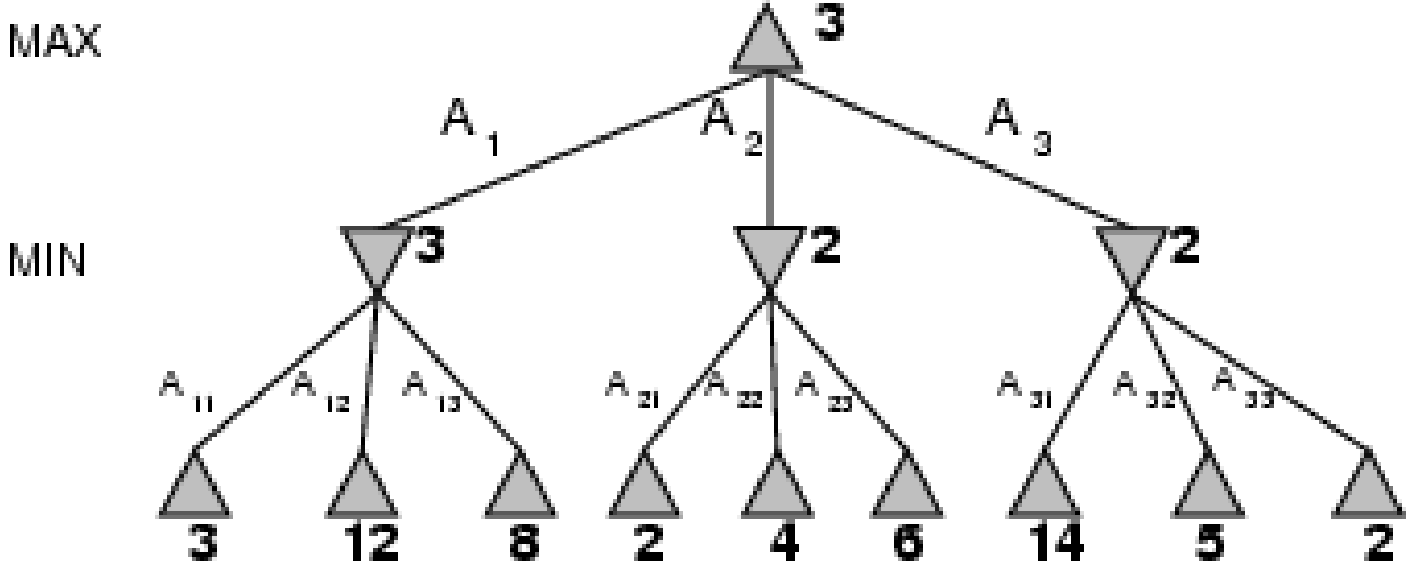


Minimax Algorithm

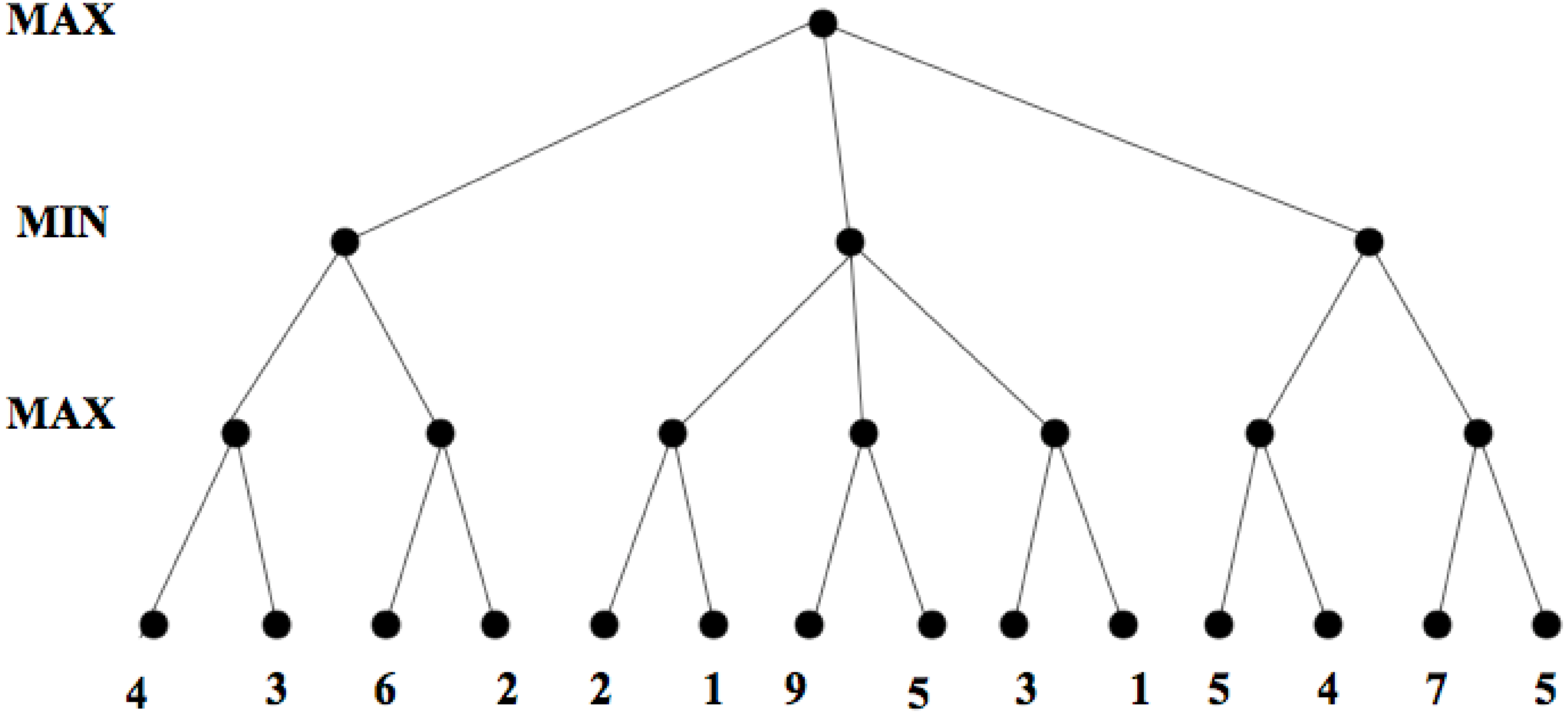
- How to deal with the contingency problem?
  - Assuming the opponent is always rational and always optimizes its behavior (opposite to us), we consider the best opponent's response
  - Then the minimax algorithm determines the best move

Minimax

- Perfect play for deterministic games
-
- Idea: choose move to position with highest minimax value
  = best achievable payoff against best play
-
- E.g., 2-ply game: [will go through another eg in lecture]
-



Minimax. Example

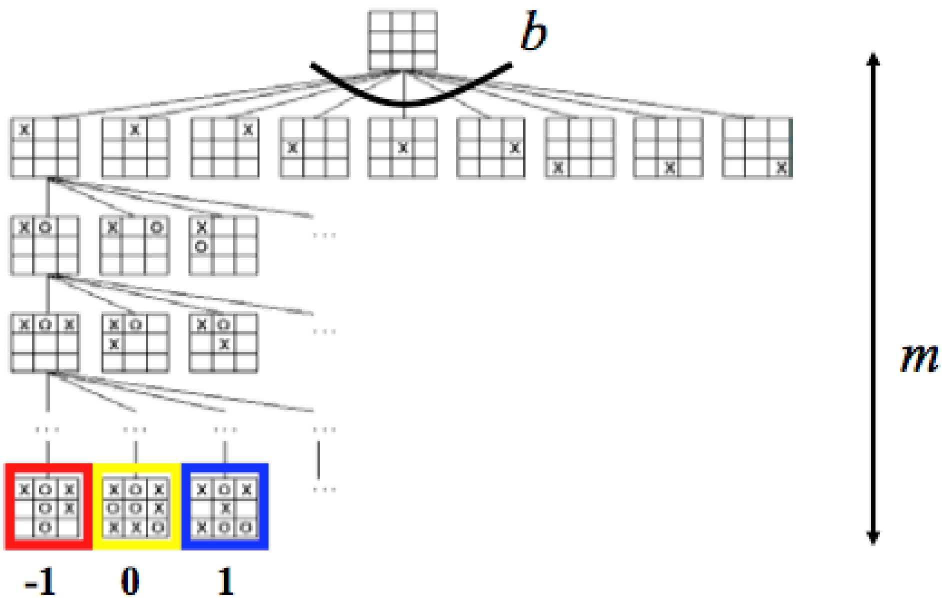


```
function MINIMAX-DECISION(game) returns an operator

    for each op in OPERATORS[game] do
        VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
    end
    return the op with the highest VALUE[op]
```

```
function MINIMAX-VALUE(state, game) returns a utility value

    if TERMINAL-TEST[game](state) then
        return UTILITY[game](state)
    else if MAX is to move in state then
        return the highest MINIMAX-VALUE of SUCCESSORS(state)
    else
        return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

Complexity of the minimax algorithm

- We need to explore the complete game tree before making the decision



Complexity:

$O(b^m)$

- Impossible for large games
  - Chess: 35 operators, game can have 50 or more moves

Solution to the complexity problem

Two solutions:

1. **Dynamic pruning of redundant branches** of the search tree
   – identify a provably suboptimal branch of the search tree before it is fully explored
   – Eliminate the suboptimal branch

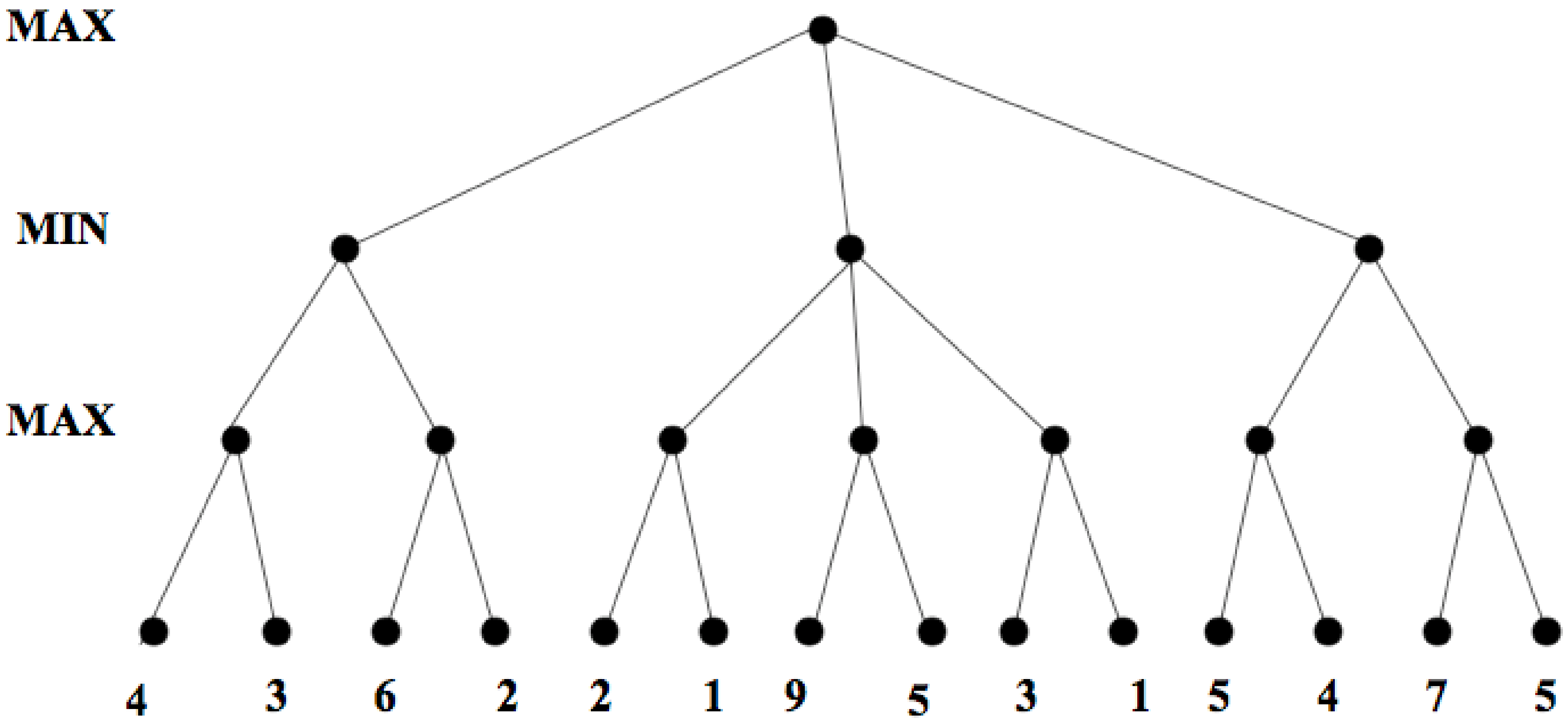
   Procedure: Alpha-Beta pruning

2. **Early cutoff of the search tree**
   – uses imperfect minimax value estimate of non-terminal states (positions)

Alpha Beta Pruning

- Some branches will never be played by rational players since they include sub-optimal decisions for either player
- First, we will see the idea of Alpha Beta Pruning
- Then, we'll introduce the algorithm for minimax with alpha beta pruning, and go through the example again, showing the book-keeping it does as it goes along

Alpha beta pruning. Example

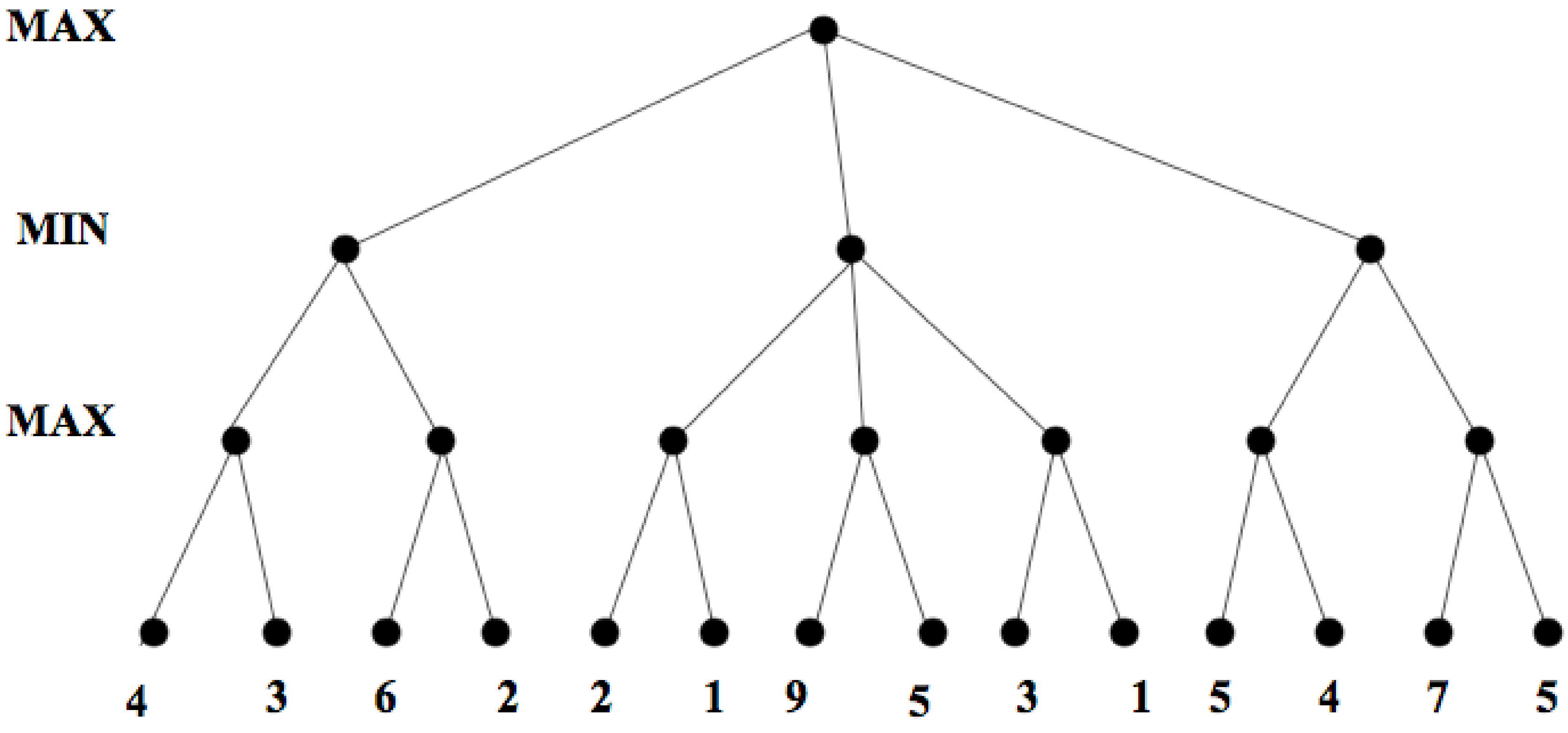


Minimax with alpha-beta pruning: The algorithm

- Maxv: function called for max nodes
  - Might update alpha, the best max can do far
- Minv: function called for min nodes
  - Might update beta, the best min can do so far

- Each tests for the appropriate pruning case
- We'll go through the algorithm on the course website

Algorithm example: alphas/betas shown



Properties of α-β

- Pruning does not affect final result
-

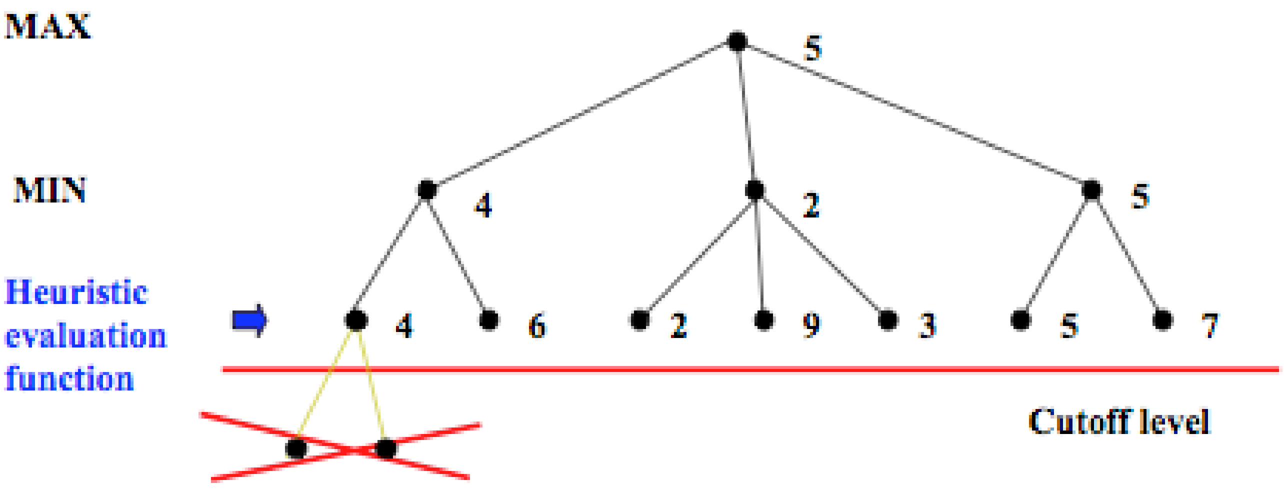
- Good move ordering improves effectiveness of pruning
-

- 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 metareasoning)
-

Using minimax value estimates

- **Idea:**
  - Cutoff the search tree before the terminal state is reached
  - Use imperfect estimate of the minimax value at the leaves
    - Evaluation function



Design of evaluation functions

- **Heuristic estimate** of the value for a sub-tree
- **Examples of a heuristic functions:**
  - **Material advantage in chess, checkers**
    - Gives a value to every piece on the board, its position and combines them
  - More general **feature-based evaluation function**
    - Typically a linear evaluation function:

$$f(s) = f_1(s)w_1 + f_2(s)w_2 + \dots f_k(s)w_k$$

$f_i(s)$ - a feature of a state s

w_i - feature weight

Further extensions to real games

- Restricted set of moves to be considered under **the cutoff level** to reduce branching and improve the evaluation function
  - E.g., consider only the capture moves in chess