CS 1571 Introduction to AI
Lecture 10

Finding optimal configurations
Adversarial search

Milos Hauskrecht
milos@cs.pitt.edu
5329 Sennott Square

Parametric optimization

Optimal configuration search:

- Configurations are described in terms of variables and their values
- Each configuration has a quality measure **f**
- Goal: find the configuration with the best value of **f**

When the state space we search is finite, the search problem is called a **combinatorial optimization problem**

When parameters we want to find are real-valued
- **parametric optimization problem**

Parametric optimization

Parametric optimization:

- Configurations are described by a vector of variables (free parameters) **w** with real-valued values
- **Goal:** find the set of parameters **w** that optimize the quality measure $f(\mathbf{w})$

Parametric optimization techniques

- **Special cases (with efficient solutions):**
  - **Linear programming**
  - **Quadratic programming**
- **First-order methods:**
  - **Gradient-ascent (descent)**
  - **Conjugate gradient**
- **Second-order methods:**
  - **Newton-Rhapson methods**
  - **Levenberg-Marquardt**
- **Constrained optimization:**
  - **Lagrange multipliers**

Linear programming

- **A special case and when:**
  - The objective function **f** is a linear combination of variable values **w**
  - Values variables **w** can take are constrained by a set of linear constraints
- **Assume variables:** $\mathbf{w_1, w_{2, \dots} w_k}$

Minimize $\mathbf{f(w_1, w_{2,\dots} w_k) = a_1 w_1 + a_2 w_2 + \dots + a_k w_k}$ (over $w_1, w_{2,\dots} w_k$)

Subject to constraints:

$$\mathbf{b_{1,1} w_1 + b_{1,2} w_2 + \dots + b_{1,k} w_k + b_{1,0} \leq 0}$$
$$\mathbf{b_{2,1} w_1 + b_{2,2} w_2 + \dots + b_{2,k} w_k + b_{2,0} \leq 0}$$

…

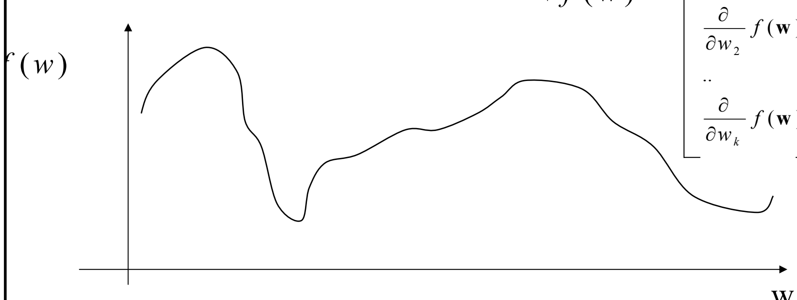
$$\mathbf{b_{m,1} w_1 + b_{m,2} w_2 + \dots + b_{m,k} w_k + b_{m,0} \leq 0}$$

Gradient ascent method

- A method for finding parameters $w_1, w_{2, \dots} w_k$ optimizing an arbitrary differentiable function $f(w_1, w_{2, \dots} w_k)$
- **Example:**

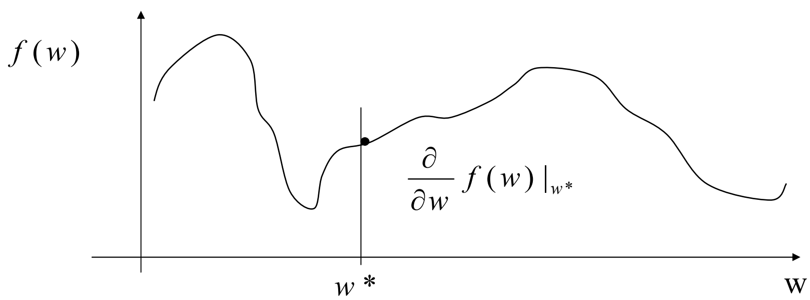
$$f(\mathbf{w})$$

$$\nabla f(\mathbf{w}) = \begin{bmatrix} \frac{\partial}{\partial w_1} f(\mathbf{w}) \\ \frac{\partial}{\partial w_2} f(\mathbf{w}) \\ .. \\ \frac{\partial}{\partial w_k} f(\mathbf{w}) \end{bmatrix}$$



Gradient ascent method

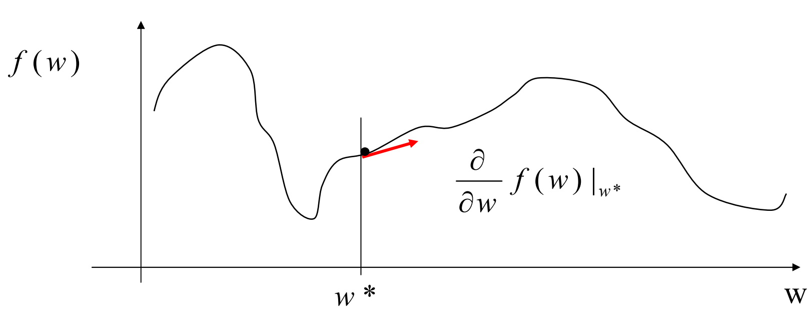
- **Gradient ascent:** the same as hill-climbing, but in the continuous parametric space **w**



- What is the derivative of an increasing function?

Gradient ascent method

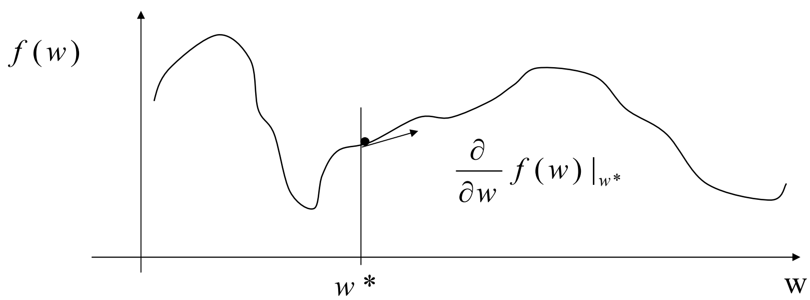
- **Gradient ascent:** the same as hill-climbing, but in the continuous parametric space **w**



- What is the derivative of an increasing function?
  - positive

Gradient ascent method

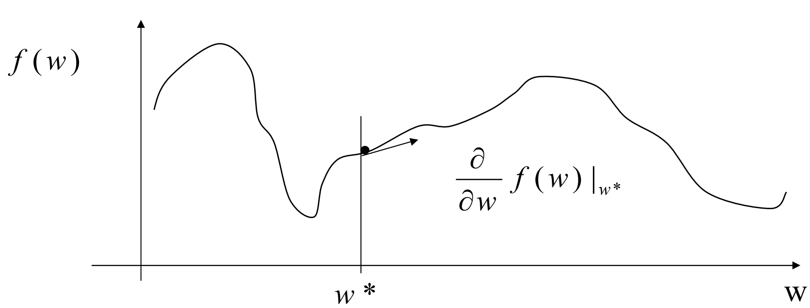
- **Gradient ascent:** the same as hill-climbing, but in the continuous parametric space **w**



- Change the parameter value of w according to the gradient

$$w \leftarrow w^* + \alpha \frac{\partial}{\partial w} f(w)|_{w^*}$$

Gradient ascent method



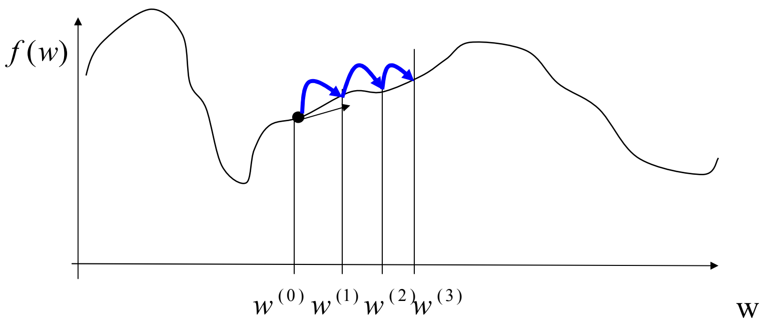
- New value of the parameter

$$w \leftarrow w^* + \alpha \frac{\partial}{\partial w} f(w)|_{w^*}$$

$\alpha > 0$ - a learning rate (scales the gradient changes)

Gradient ascent method

- To get to the function minimum repeat (iterate) the gradient based update few times



- **Problems:** local optima, saddle points, slow convergence
- More complex optimization techniques use additional information (e.g. second derivatives)

Adversarial search

Game search

- Game-playing programs developed by AI researchers since the beginning of the modern AI era
  - Programs playing chess, checkers, etc (1950s)

- **Specifics of the game search:**
  - 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 an uncertainty in to the game
  - We do not know exactly what the response is going to be
- **Rational opponent** – maximizes it own **utility (payoff) function**

Types of game problems

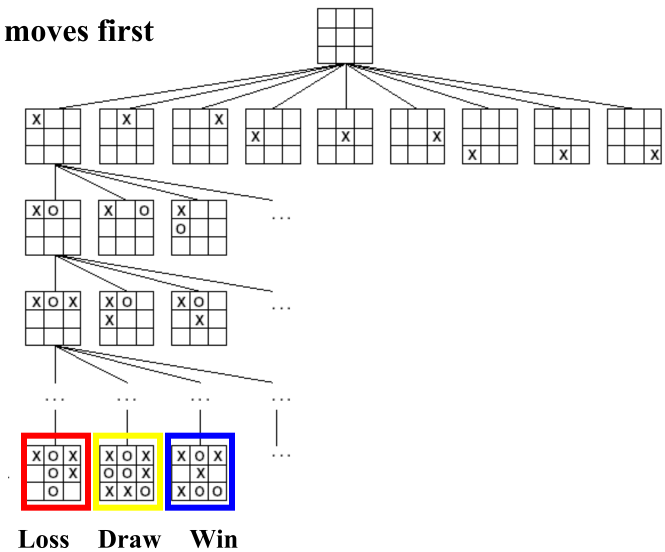
- **Types of game problems:**
  - **Adversarial games:**
    - win of one player is a loss of the other
  - **Cooperative games:**
    - players have common interests and utility function
  - **A spectrum of game problems in between the two:**

Adversarial games **Fully cooperative games**

we focus on adversarial games only!!

Example of an adversarial 2 person game: Tic-tac-toe

- **Player 1 (x) moves first**



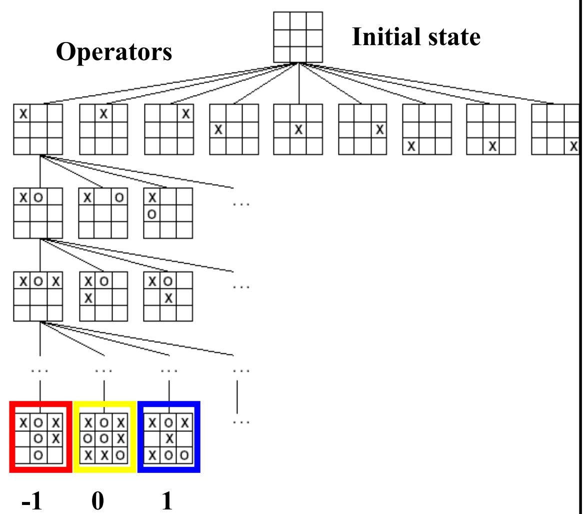
Game search problem

- **Game problem formulation:**
  - **Initial state:** initial board position + info whose move it is
  - **Operators:** legal moves a player can make
  - **Goal (terminal test):** determines when the game is 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 problem formulation (Tic-tac-toe)

Objectives:
- **Player 1**: maximize outcome
- **Player 2**: minimize outcome



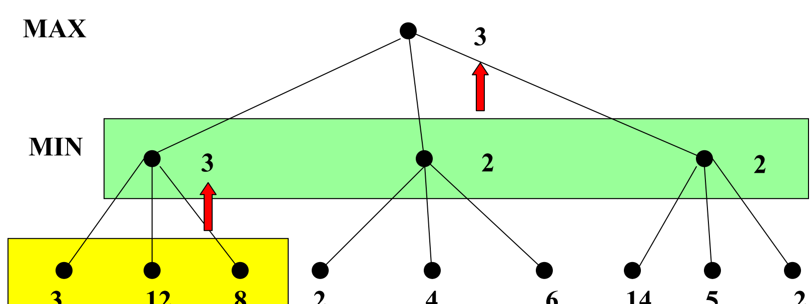
Terminal (goal) states

Utility:

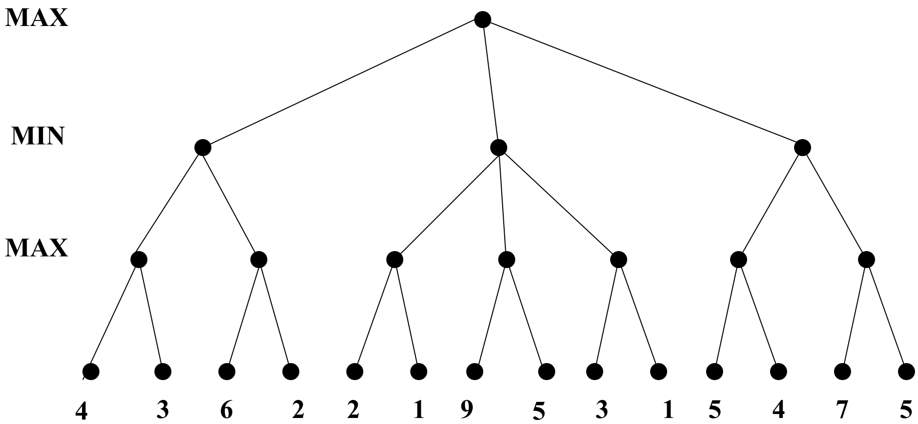
Minimax algorithm

How to deal with the contingency problem?

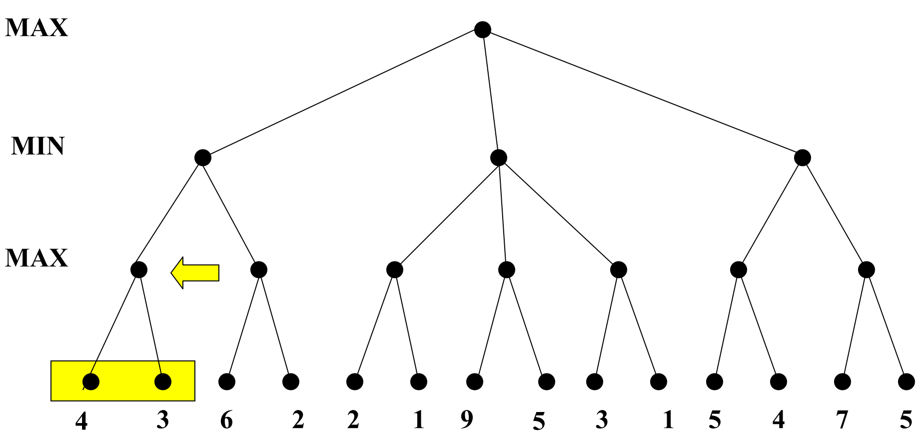
- Assuming that the opponent is rational and always optimizes its behavior (opposite to us) we consider **the best opponent's response**
- Then the **minimax algorithm** determines the best move



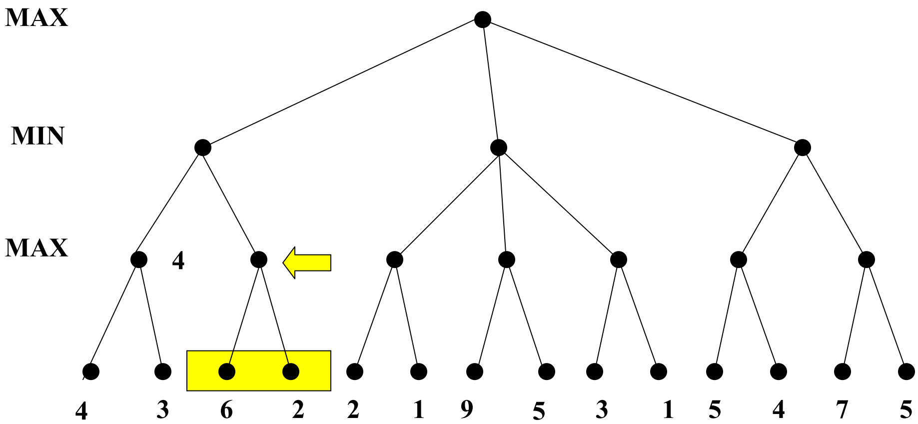
Minimax algorithm. Example



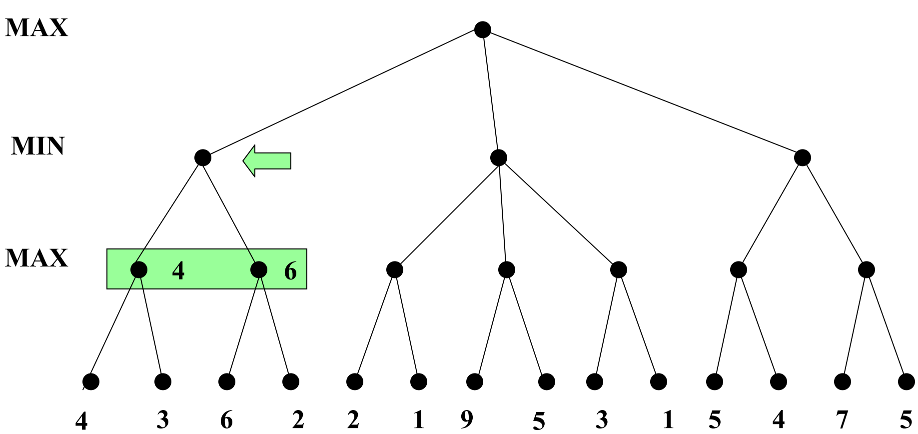
Minimax algorithm. Example



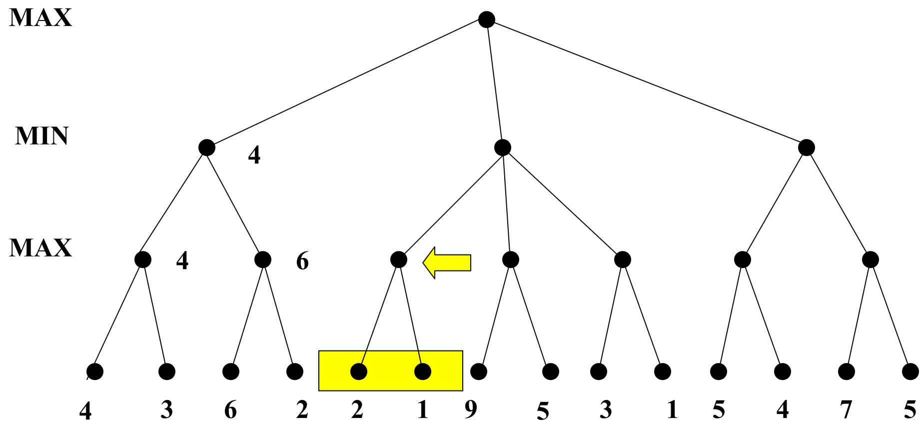
Minimax algorithm. Example



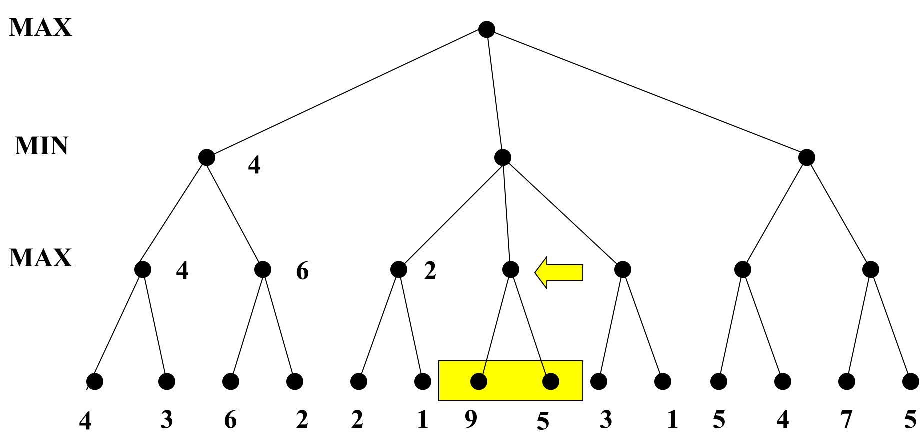
Minimax algorithm. Example



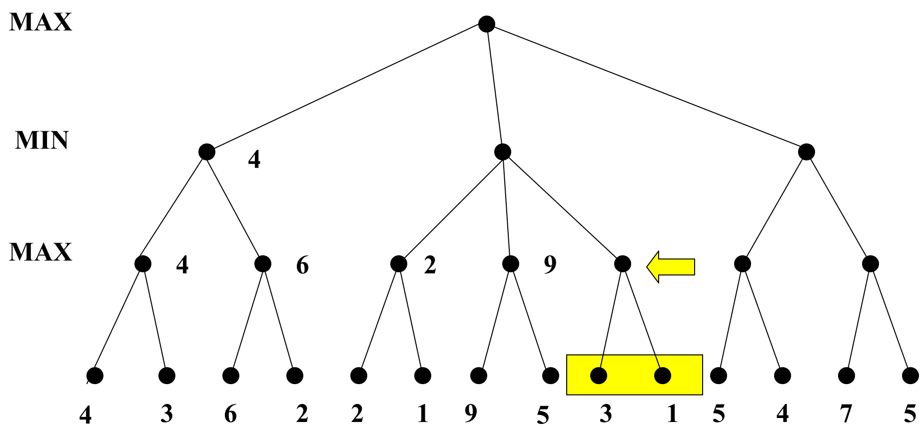
Minimax algorithm. Example



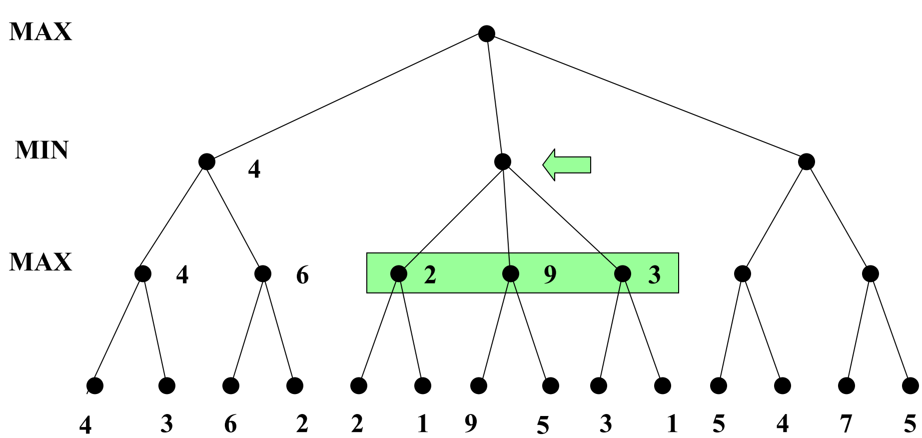
Minimax algorithm. Example



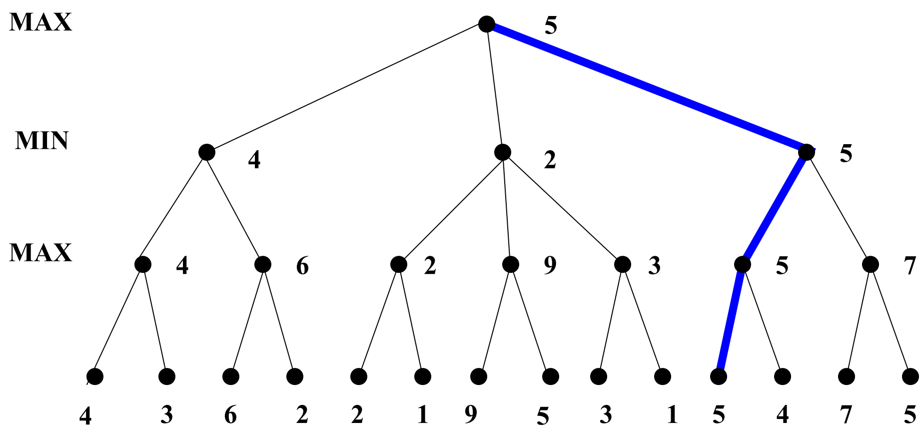
Minimax algorithm. Example



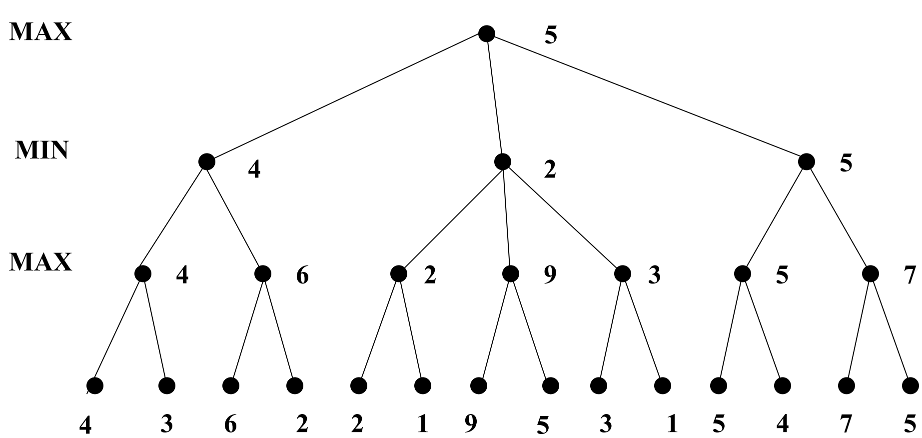
Minimax algorithm. Example



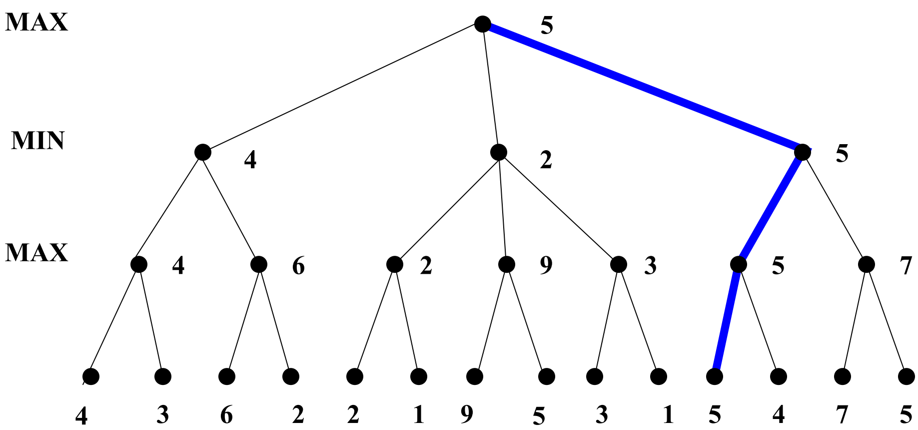
Minimax algorithm. Example



Minimax algorithm. Example



Minimax algorithm. Example



Minimax algorithm

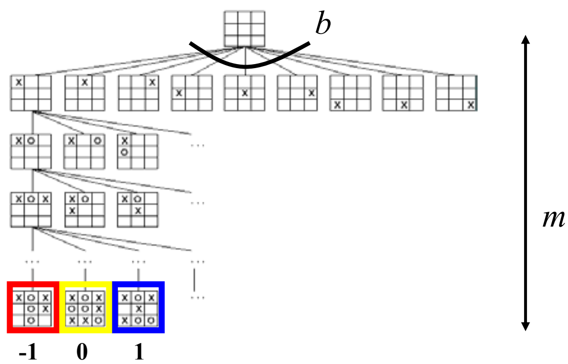
```
function MINIMAX-DECISION(game) returns an operator

  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

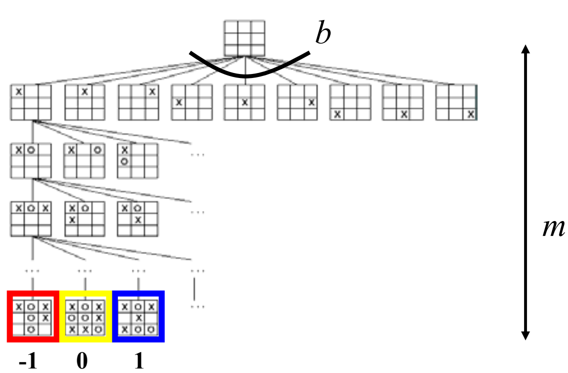


Complexity:

?

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

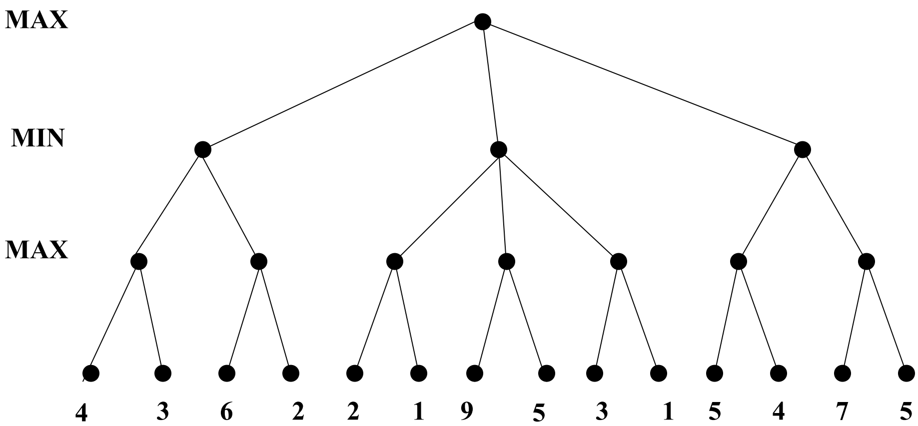
   Procedure: Alpha-Beta pruning

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

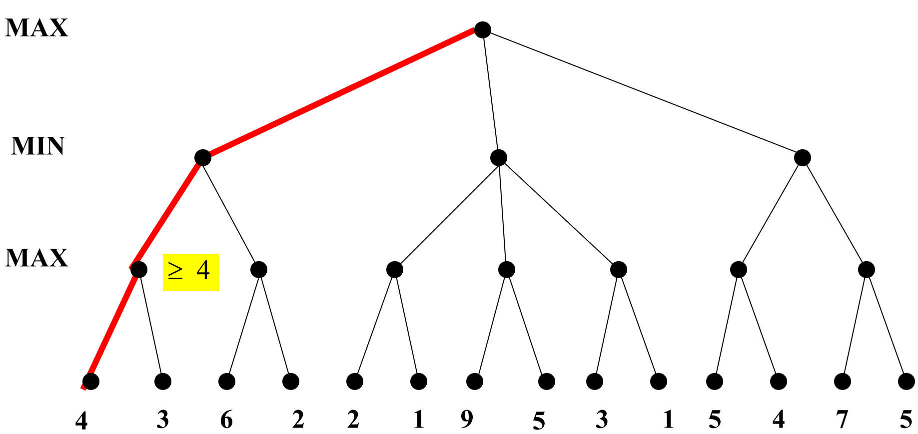
Alpha beta pruning

- Some branches will never be played by rational players since they include sub-optimal decisions (for either player)

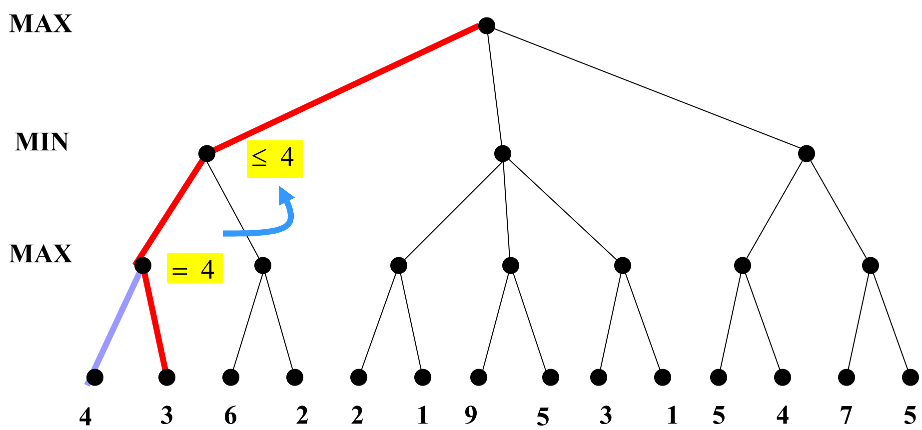
Alpha beta pruning. Example



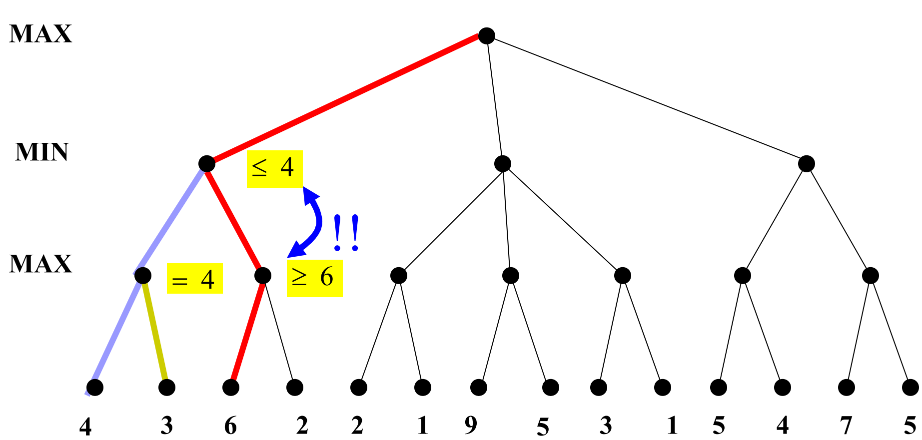
Alpha beta pruning. Example



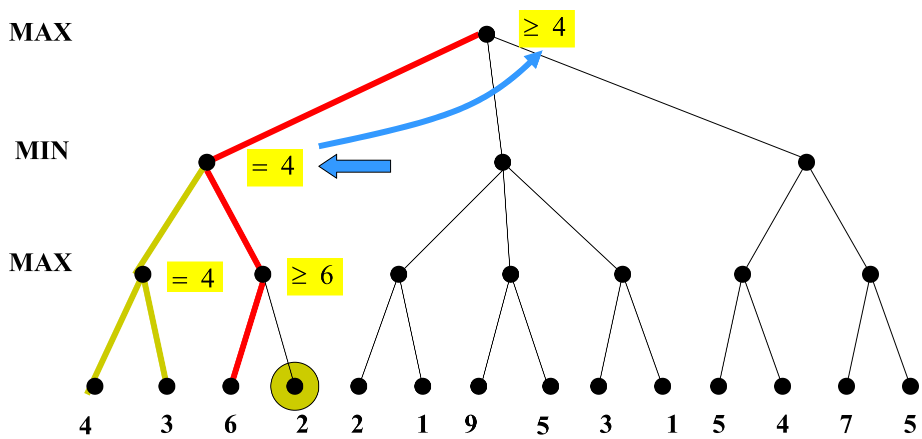
Alpha beta pruning. Example



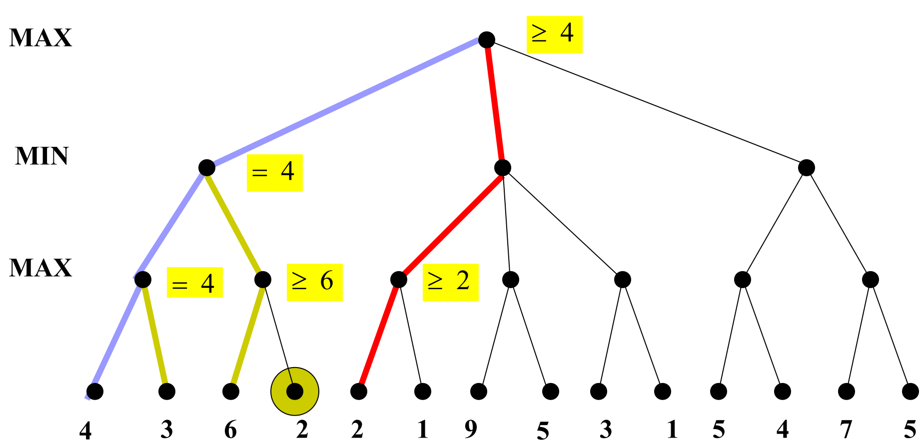
Alpha beta pruning. Example



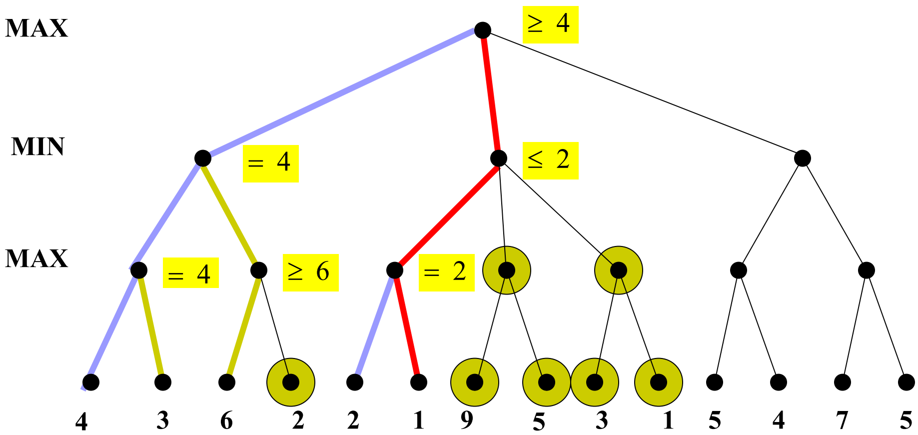
Alpha beta pruning. Example



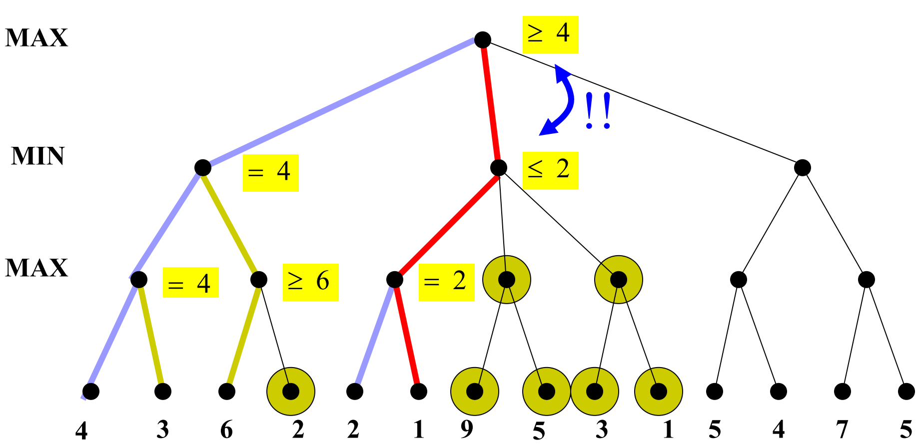
Alpha beta pruning. Example



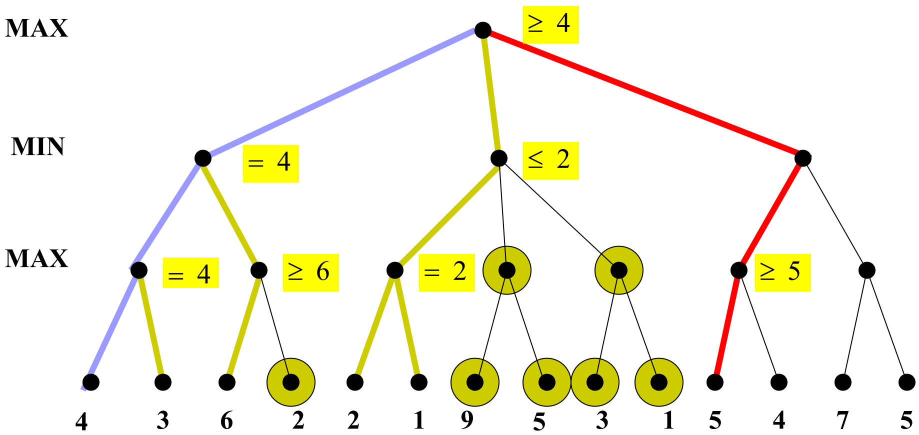
Alpha beta pruning. Example



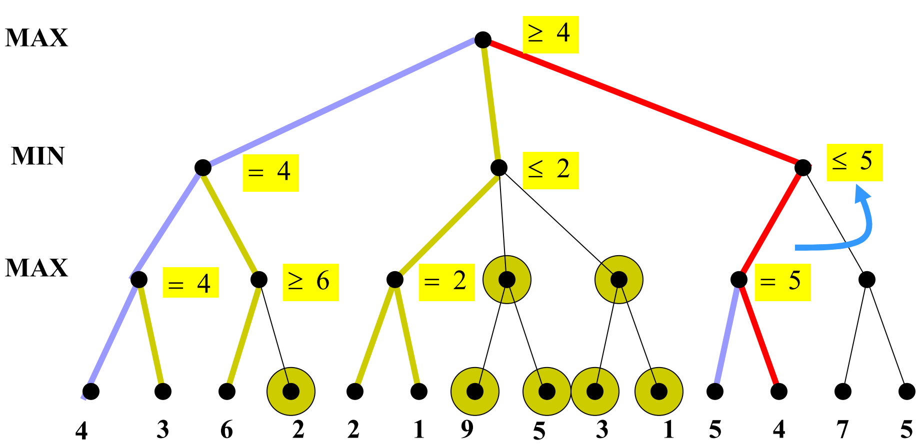
Alpha beta pruning. Example



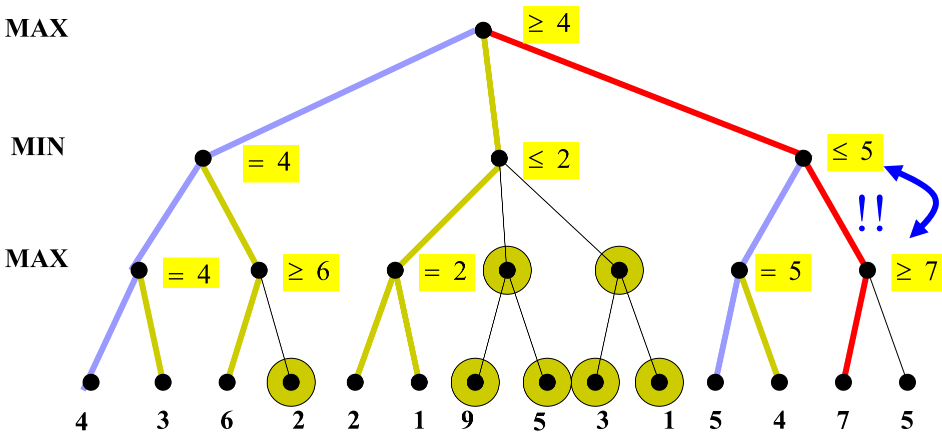
Alpha beta pruning. Example



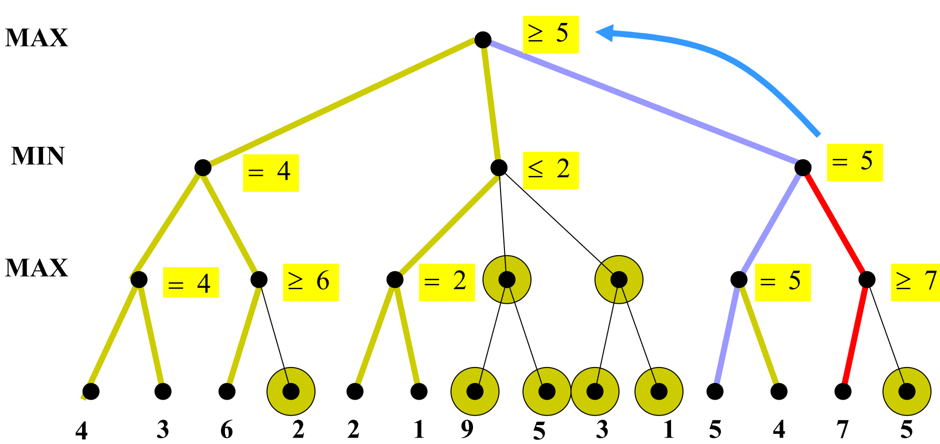
Alpha beta pruning. Example



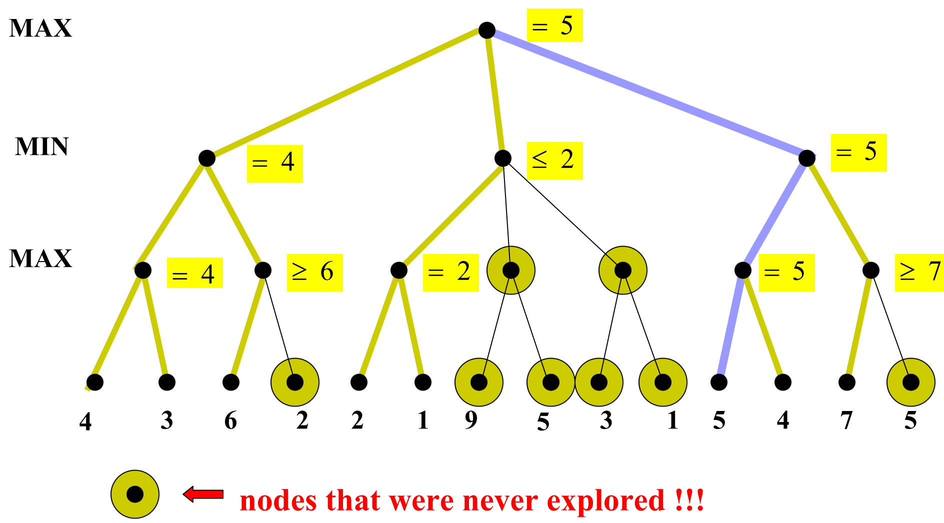
Alpha beta pruning. Example



Alpha beta pruning. Example



Alpha beta pruning. Example



Alpha-Beta pruning

function MAX-VALUE(*state, game*, α, β) **returns** the minimax value of *state*
  inputs: *state*, current state in game
    game, game description
    α, the best score for MAX along the path to *state*
    β, the best score for MIN along the path to *state*

  if GOAL-TEST(*state*) **then return** EVAL(*state*)
  for each *s* **in** SUCCESSORS(*state*) **do**
    $\alpha \leftarrow$ MAX(α, MIN-VALUE(*s, game*, α, β))
    if $\alpha \geq \beta$ **then return** β
  end
  return α

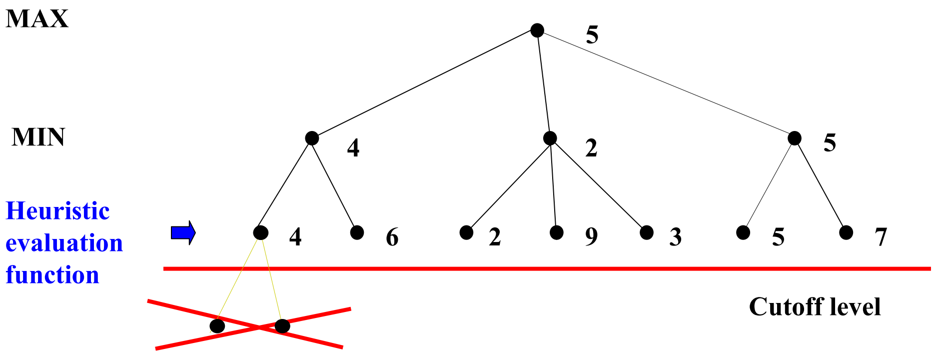
---

function MIN-VALUE(*state, game*, α, β) **returns** the minimax value of *state*

  if GOAL-TEST(*state*) **then return** EVAL(*state*)
  for each *s* **in** SUCCESSORS(*state*) **do**
    $\beta \leftarrow$ MIN(β, MAX-VALUE(*s, game*, α, β))
    if $\beta \leq \alpha$ **then return** α
  end
  return β

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:

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

$f_i(s)$ - 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