CS 1571 Introduction to AI
Lecture 12

Adversarial search

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

Announcements

- **Homework assignment 4 is out**
  – Programming and experiments
  – Simulated annealing + Genetic algorithm
  – Competition

Course web page:

http://www.cs.pitt.edu/~milos/courses/cs1571/

Search review

Search
- Path search
- Configuration search

Optimality
- Finding a path versus finding the optimal path
- Finding a configuration satisfying constraints versus finding the best configuration

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

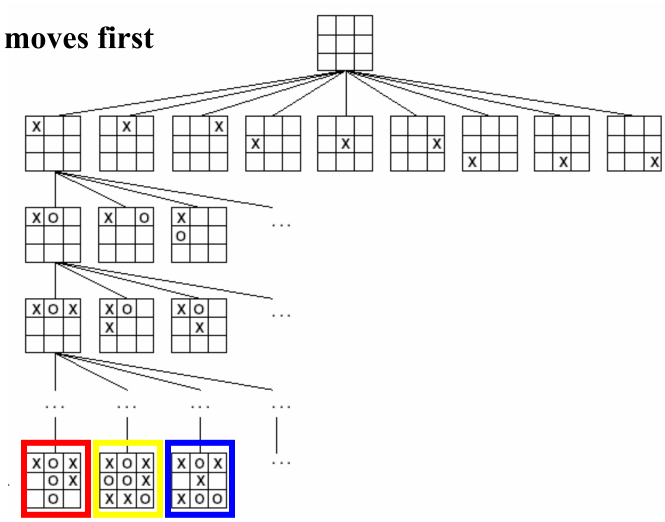
- **Types of game problems:**
  - **Adversarial games:**
    - win of one player is a loss of the other
  - **Cooperative games:**
    - players have common interests and utility function
  - **A spectrum of game problems in between the two:**

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**



Loss Draw Win

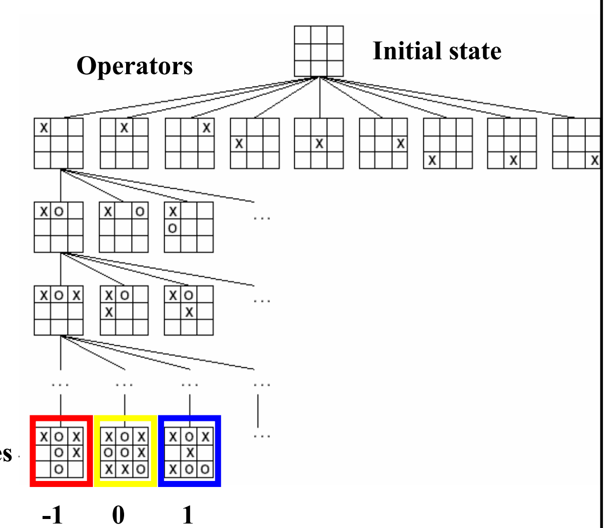
Game search problem

- **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



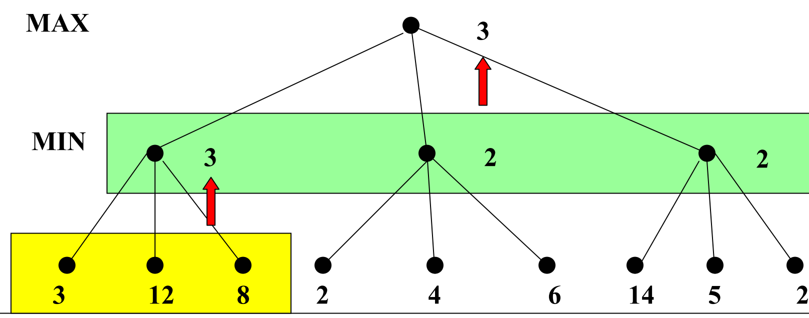
Terminal (goal) states

Utility:

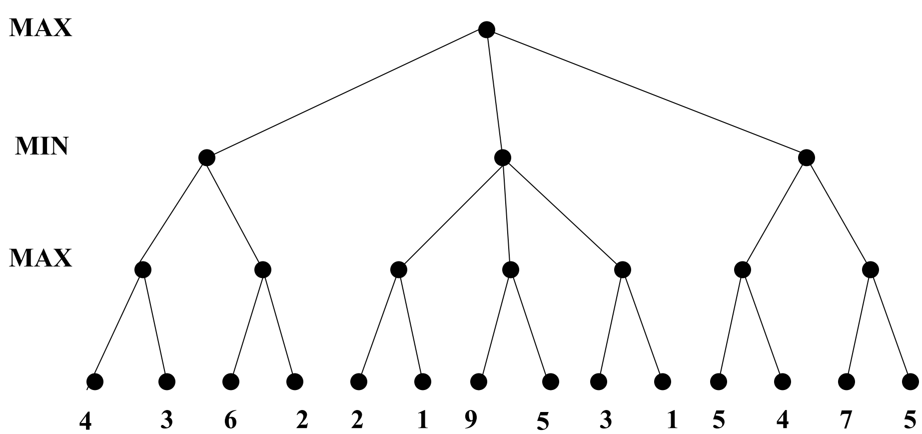
Minimax algorithm

How to deal with the contingency problem?

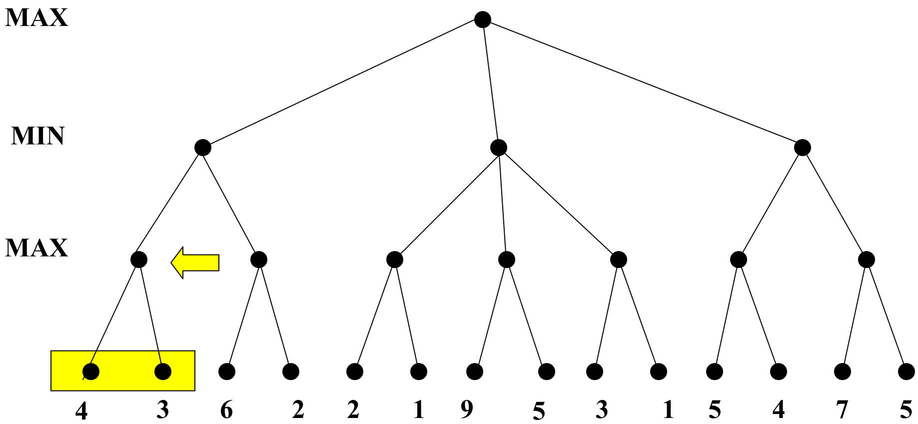
- Assuming that the opponent is rational and always optimizes its behavior (opposite to us) we consider **the best opponent's response**
- Then the **minimax algorithm** determines the best move



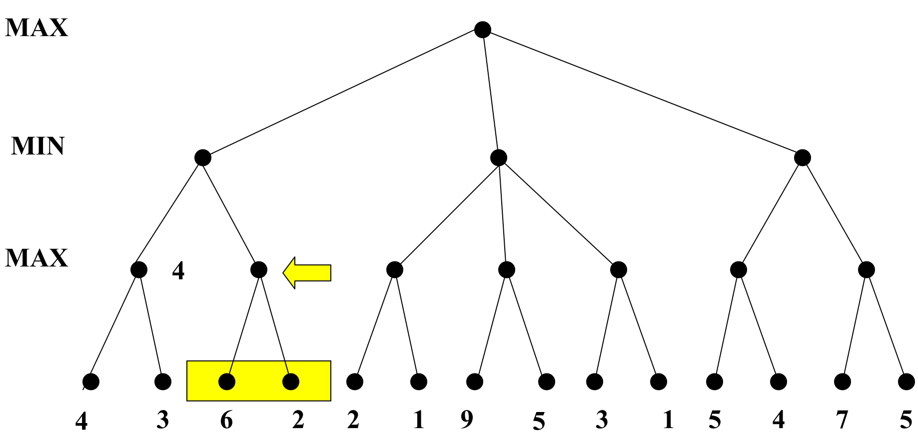
Minimax algorithm. Example



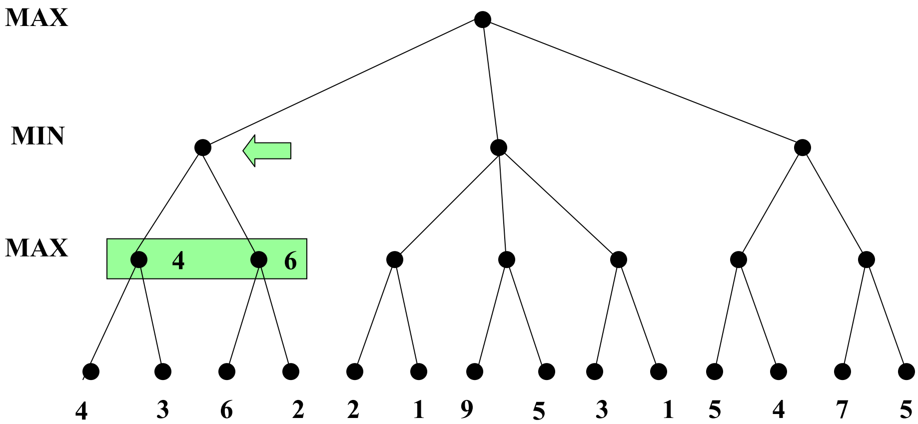
Minimax algorithm. Example



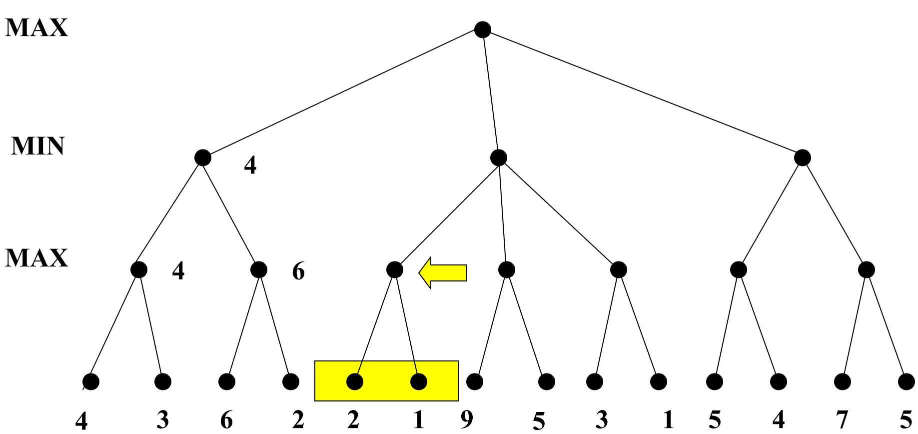
Minimax algorithm. Example



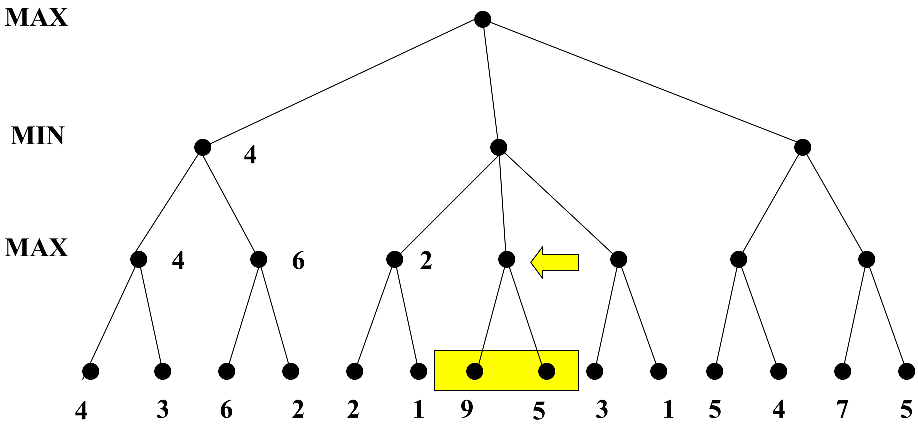
Minimax algorithm. Example



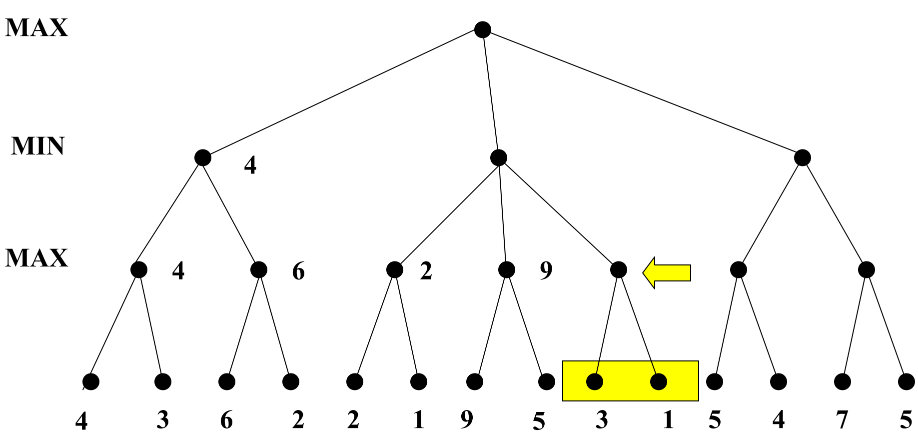
Minimax algorithm. Example



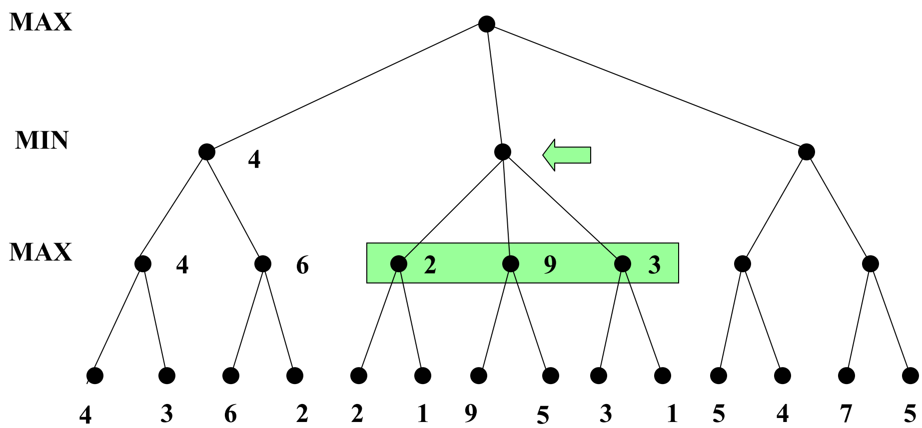
Minimax algorithm. Example



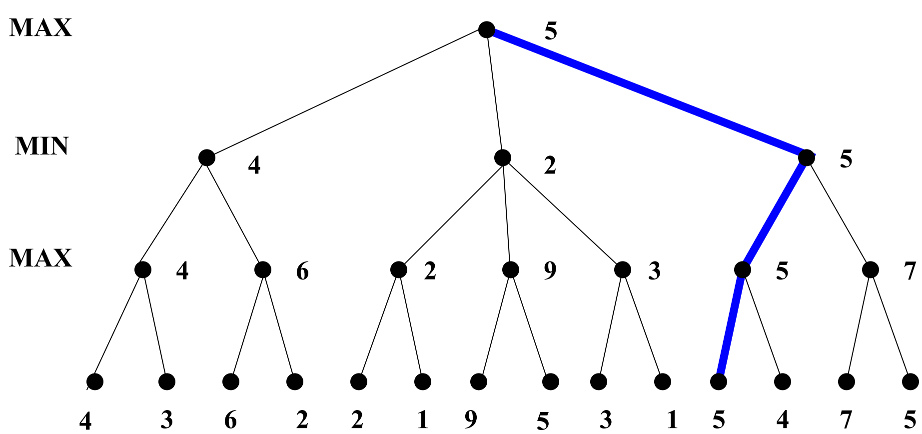
Minimax algorithm. Example



Minimax algorithm. Example



Minimax algorithm. Example



Minimax algorithm

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*]

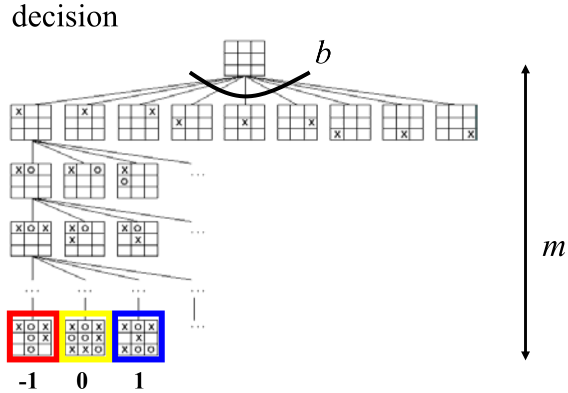
---

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

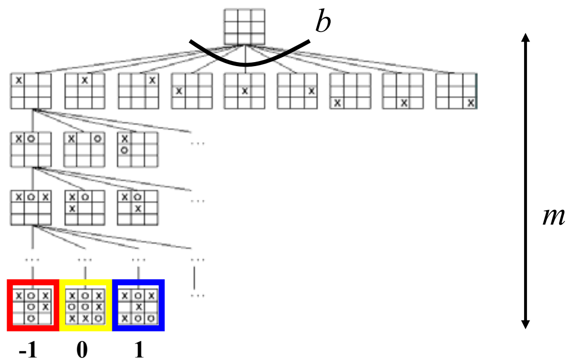


Complexity:

?

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

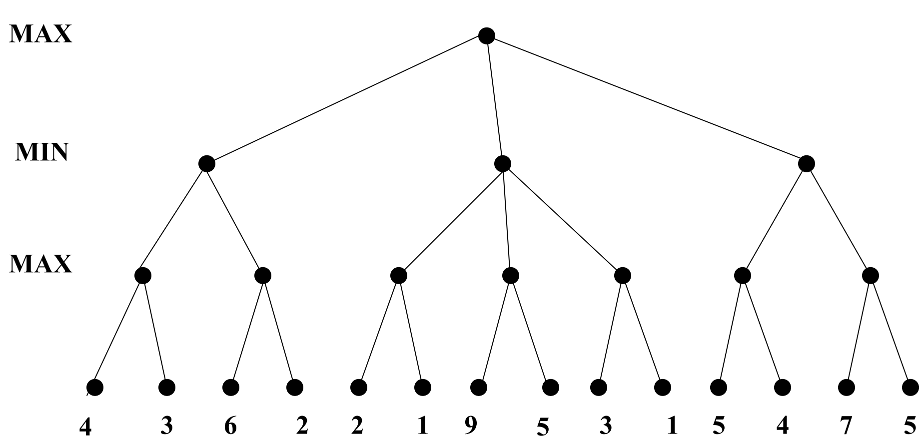
   Procedure: Alpha-Beta pruning

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

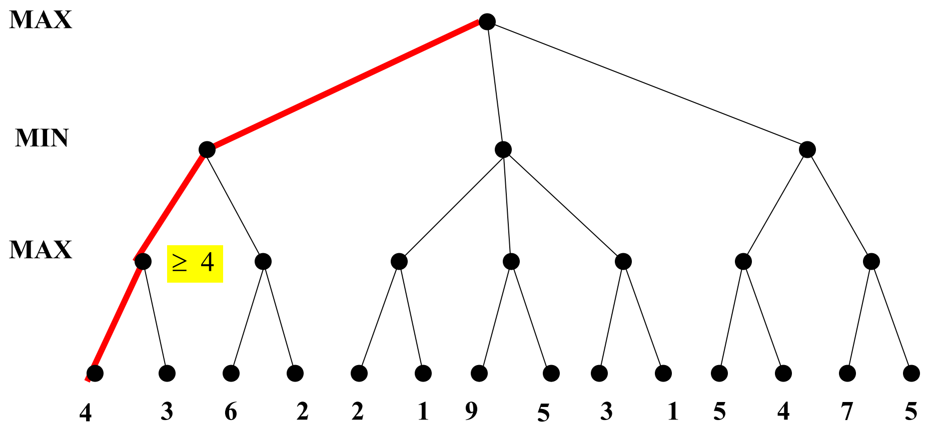
Alpha beta pruning

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

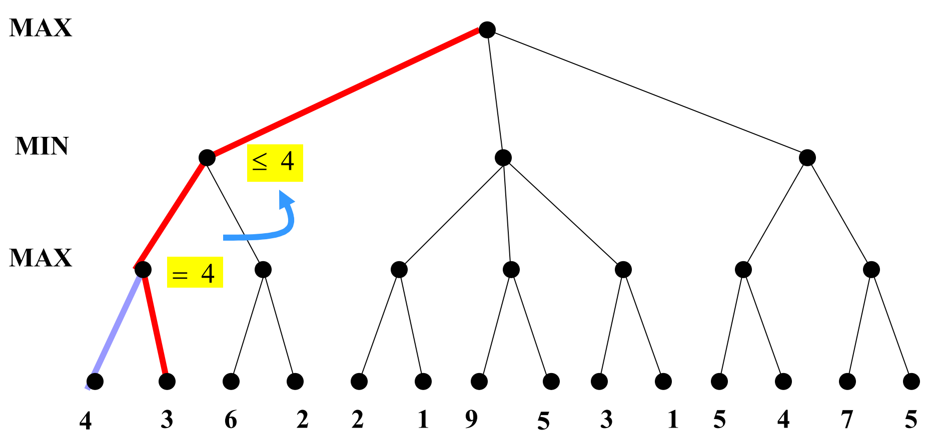
Alpha beta pruning. Example



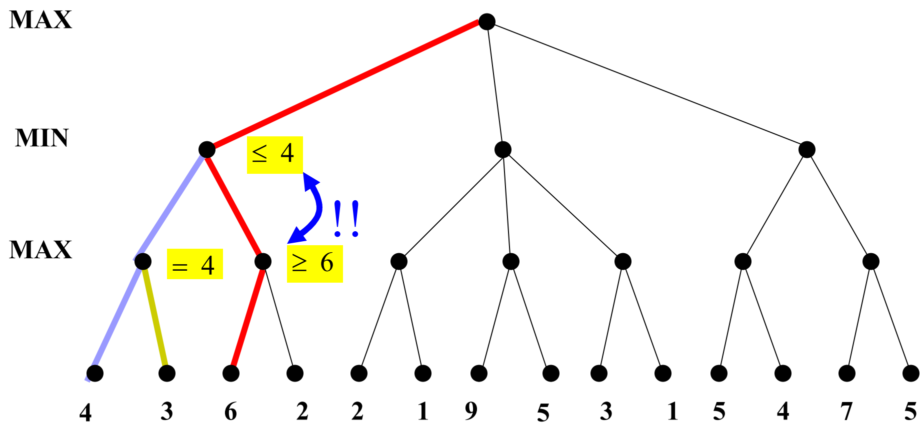
Alpha beta pruning. Example



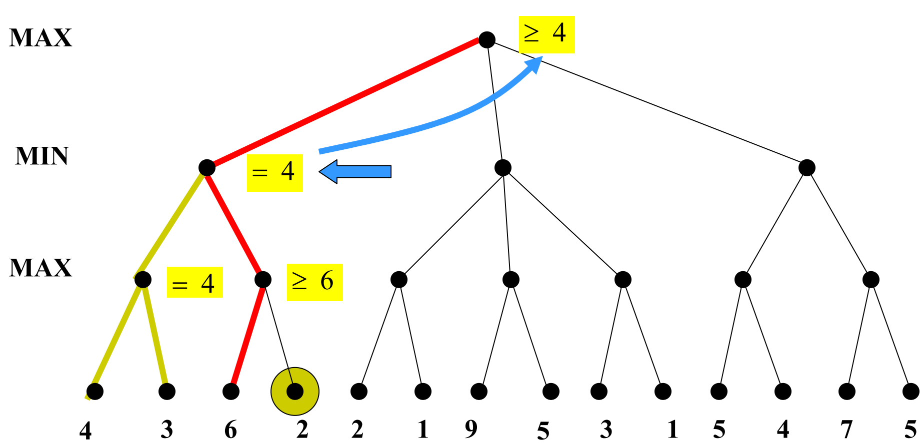
Alpha beta pruning. Example



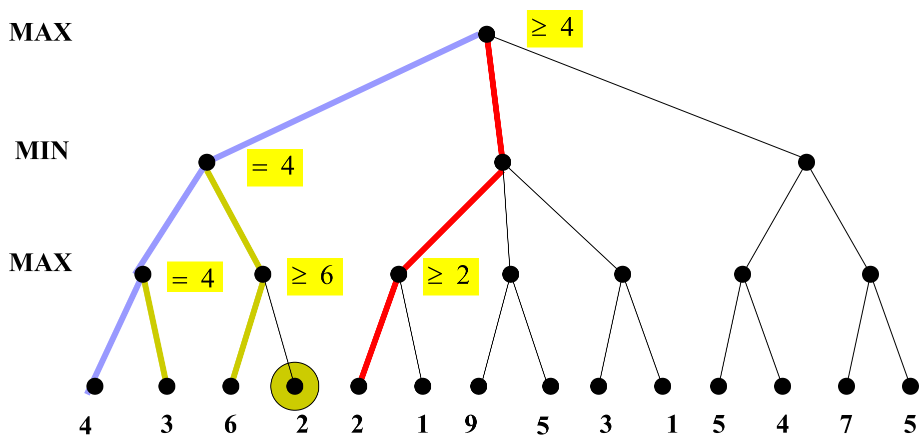
Alpha beta pruning. Example



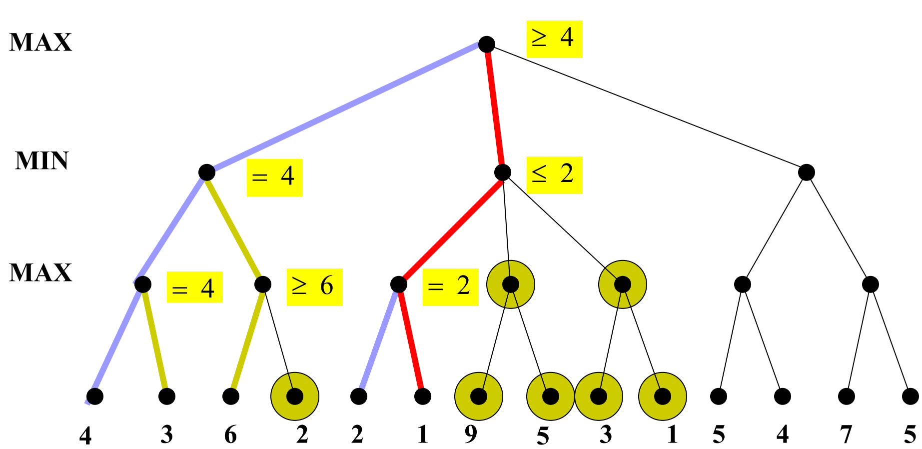
Alpha beta pruning. Example



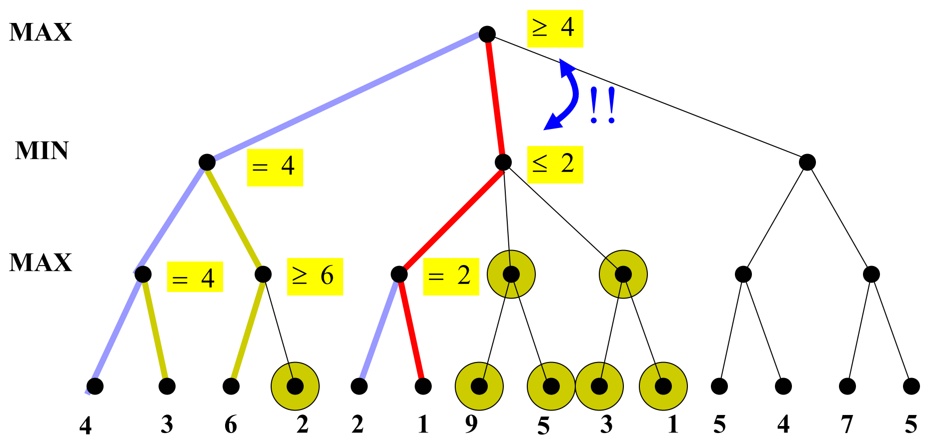
Alpha beta pruning. Example



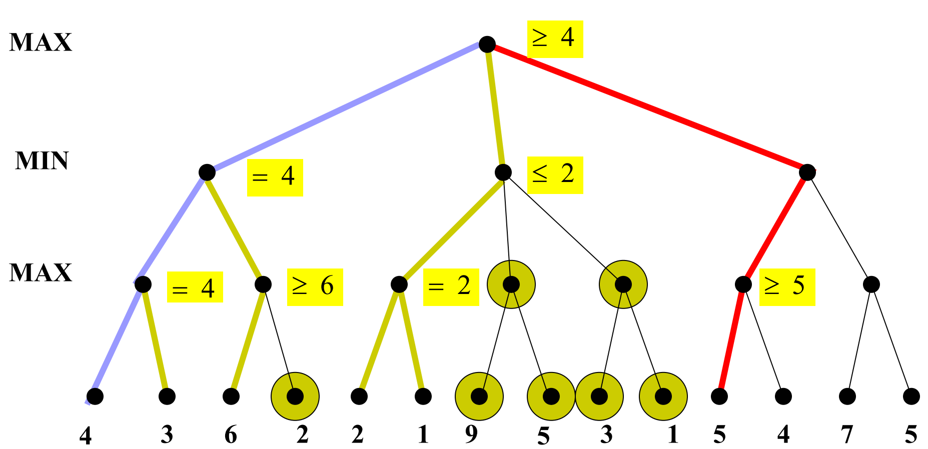
Alpha beta pruning. Example



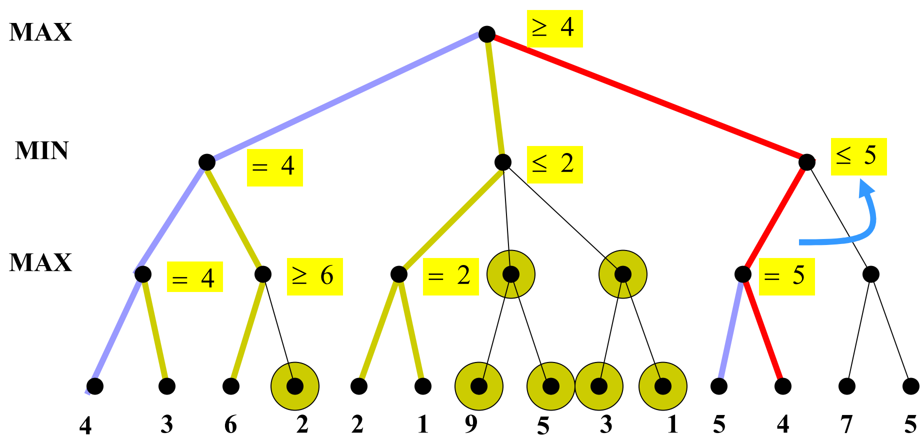
Alpha beta pruning. Example



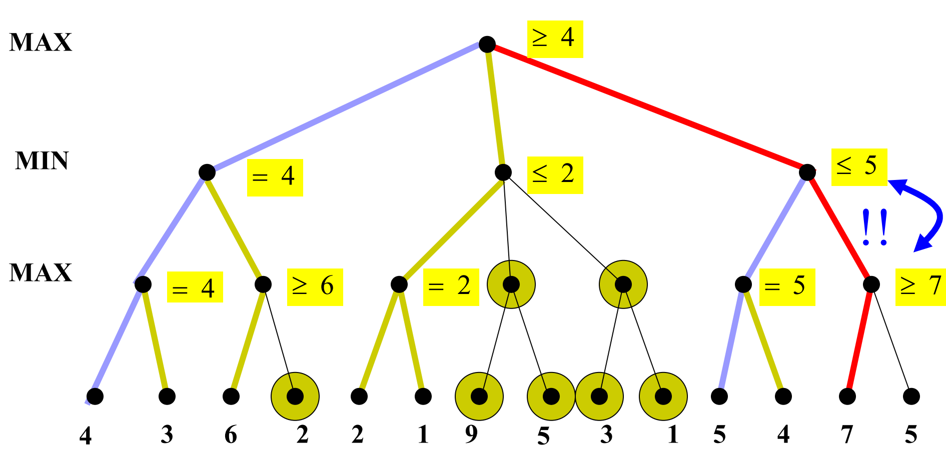
Alpha beta pruning. Example



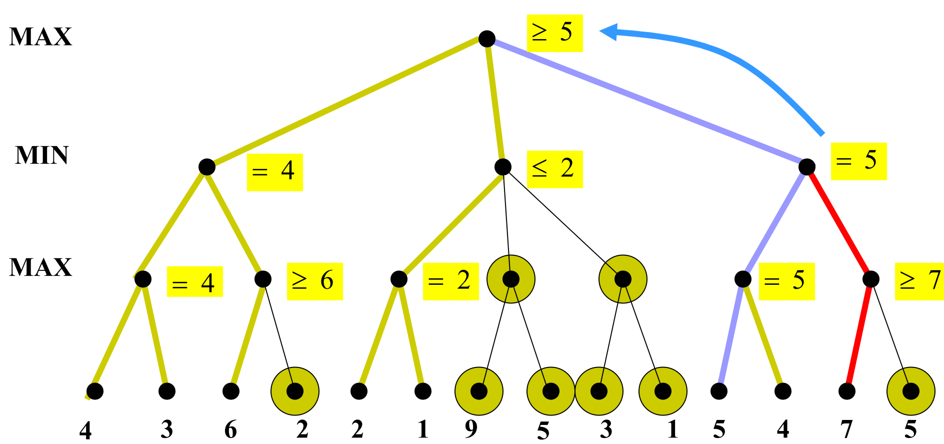
Alpha beta pruning. Example



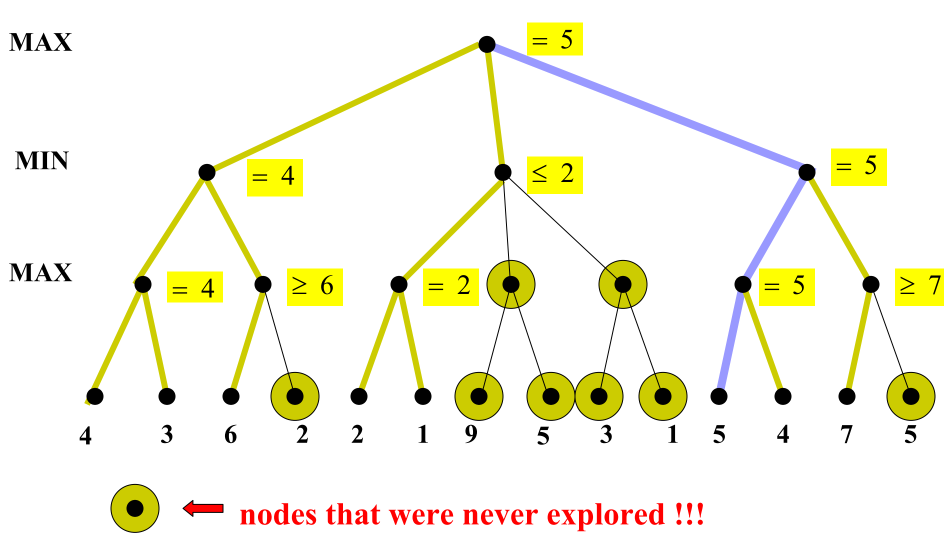
Alpha beta pruning. Example



Alpha beta pruning. Example



Alpha beta pruning. Example



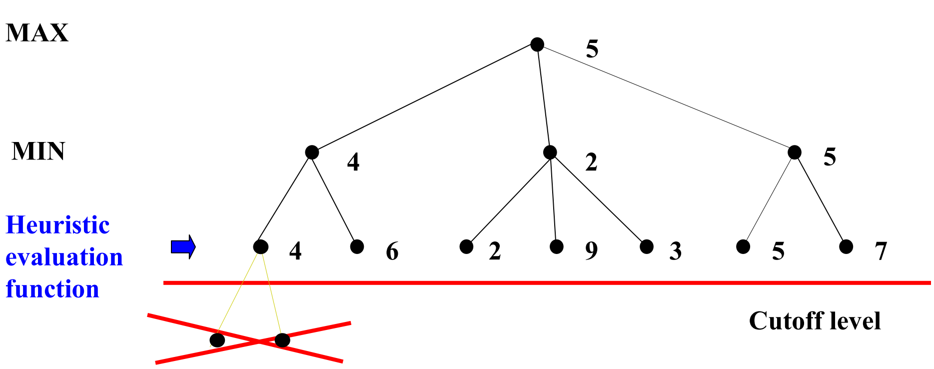
Alpha-Beta pruning

```
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
      α ← MAX(α, MIN-VALUE(s, game, α, β))
      if α ≥ β 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
      β ← MIN(β, MAX-VALUE(s, game, α, β))
      if β ≤ α 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:

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

$f_i(s)$ - 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