AI & Robotics

State space and game AI

AI & ROBOTICS LAB

Goals

The **junior-colleague**

- can describe & explain in own words the position game AI as a subfield
- can describe in own words the link between tree search and game AI
- can explain in own words the differences between simple and more complex board games in context of game AI using real world games
- can explain in own words the term “contingency” of a problem
- can describe in own words how a game AI using tree search on an abstract level
- can explain in own words Minimax in context of game AI
- can explain in own words the time and space complexity of Minimax
- can implement Minimax for a given problem
- can explain in own words an improvement of Minimax
- can implement an improvement of Minimax
- can explain in own words Alpha-Beta pruning in context of game AI
- can explain in own words the best case and worst case gain of Alpha-Beta pruning in context of game AI
- can describe in own words the term Heuristic continuation in context of game AI and what problem it solves in context of Alpha-Beta pruning using a real world example
- can implement Alpha-Beta pruning for a given problem
- can implement Heuristic continuation for a given problem

Why?

- One of the oldest subfields of AI
- Abstract and pure form of competition that seems to require intelligence
- Game playing is a special case of a search problem,
  with some new requirements.

How?

- Simple board games
    - Easy to represent the states and actions
    - Very little world knowledge required!
    - “Contingency” problem:

      => We do not know the opponents move!
    - The size of the search space:
        - Chess : +/- 15 moves possible per state, 80 plays => 15^{80} nodes in tree
        - Go : +/- 200 moves per state, 300 plays => 200^{300} nodes in tree
- More complex games
    - State space representation becomes increasingly difficult

      => How to represent a game world?

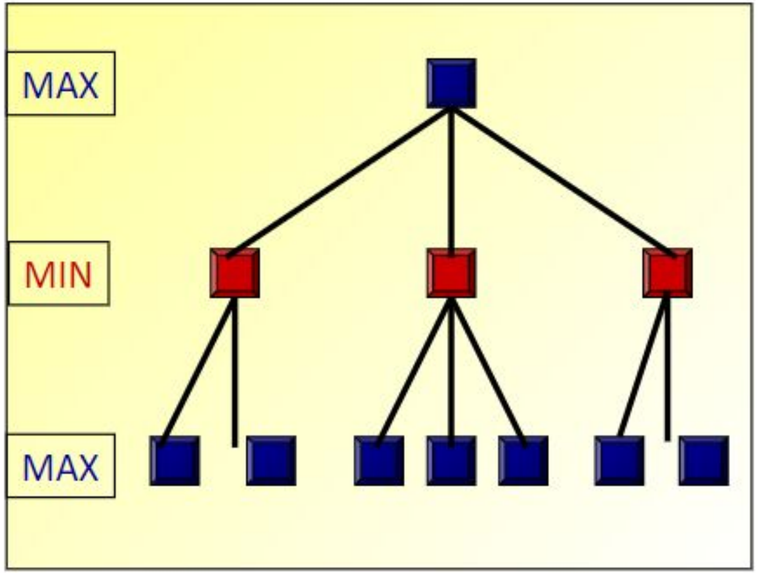
How?

Game playing algorithms:

- Search tree only up to some depth bound
- Use an evaluation function at the depth bound
- Propagate the evaluation upwards in the tree

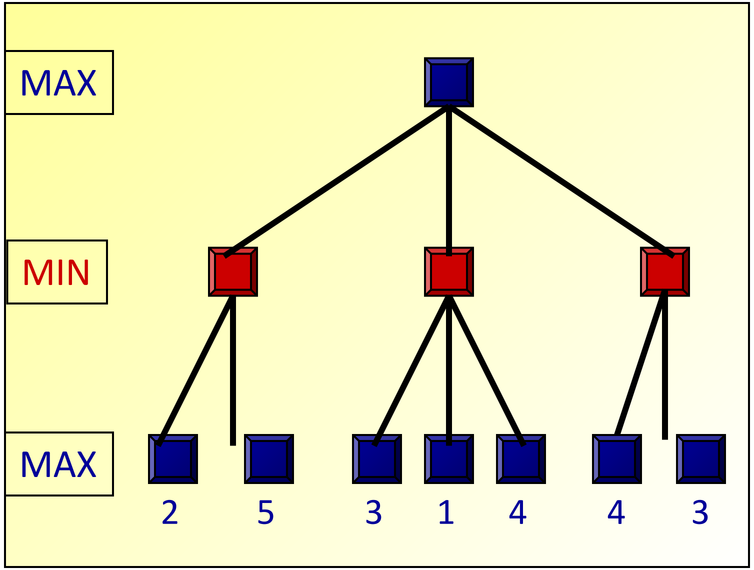
MiniMax

- Consider a board game with 2 players:
  - **MAX** (AI player)
  - **MIN** (opponent)
- Each player alternates between taking a turn
- The game ends when either:
  - One of the 2 players reaches a winning state
  - No more moves are possible
- Assumptions:
  - Deterministic
  - Perfect information



MiniMax

- Select a depth-bound (say: 2) and evaluation function
- Construct the tree up till the depth-bound
- Compute the evaluation function for the leaves

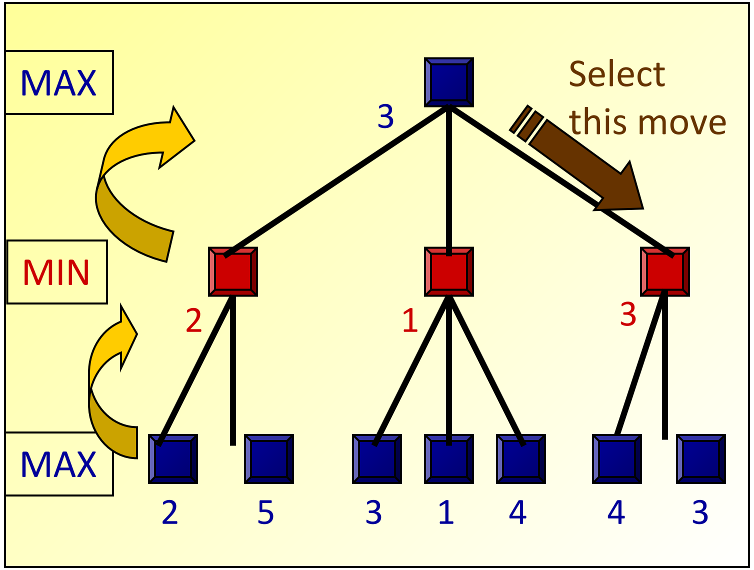


MiniMax

- **MAX**-player wants to maximize **MIN**-player's ultimate score
- **MIN**-player wants to minimize **MAX**-player's ultimate score

=> Propagate the evaluation function Upwards:

  - Take minima in **MIN**
  - Take maxima in **MAX**



MiniMax

```
init depthBound
function miniMax(board, depth):
        if depth == depthBound
             return eval(board)
    else if maximizer(depth)
        for each child c of board
             value = max(value, miniMax(child, depth + 1))
        return value
     else [minimizer]
        for each child c of board
                  value = min(value, miniMax(child, depth + 1))
             return value
```

MiniMax

Analysis

- Time complexity

=> Same as iterative deepening (search bounded by depth m): **$O(b^m)$**

- Space complexity

=> Same as iterative deepening (search bounded by depth m): **O(b*m)**

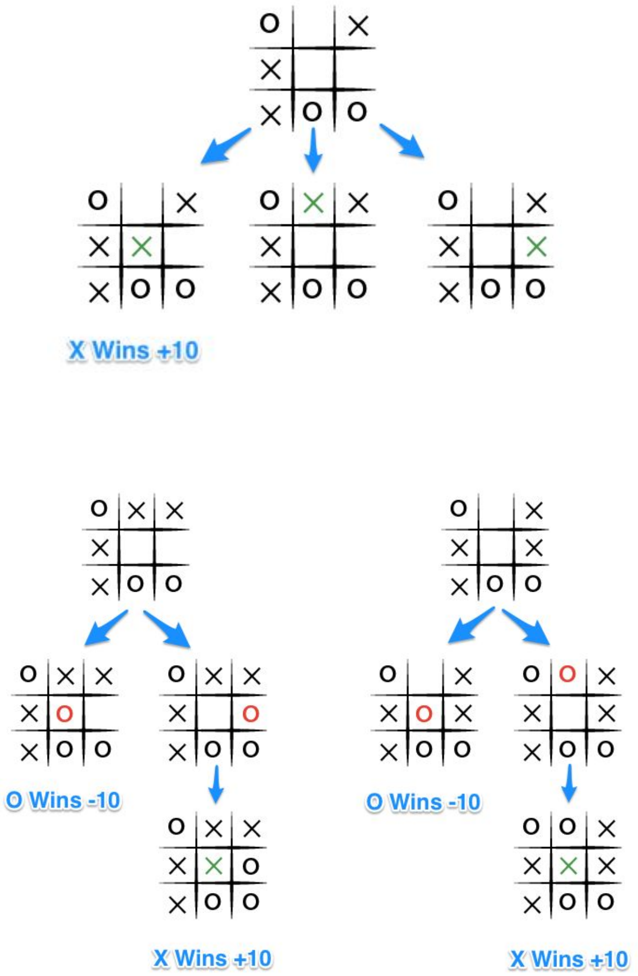
MiniMax: Tic-Tac-Toe

- 2 players: X and O
- State representation of the board: i.e. matrix
- Production rules:
    - X move: place X on a free spot on the board
    - O move: place O on a free spot on the board
- Start state: empty board
- Goal state:
    - 3 X’s in a row
    - 3 O’s in a row
    - Full board

MiniMax: Tic-Tac-Toe

- **MAX**-player (X) wins: +10
- **MIN**-player (O) wins: -10

```
function eval(board, depth):
        if maximizer(depth)
             return 10
        else if
 minimizer(depth)
             return -10
        else
             return 0
```



MiniMax: Tic-Tac-Toe

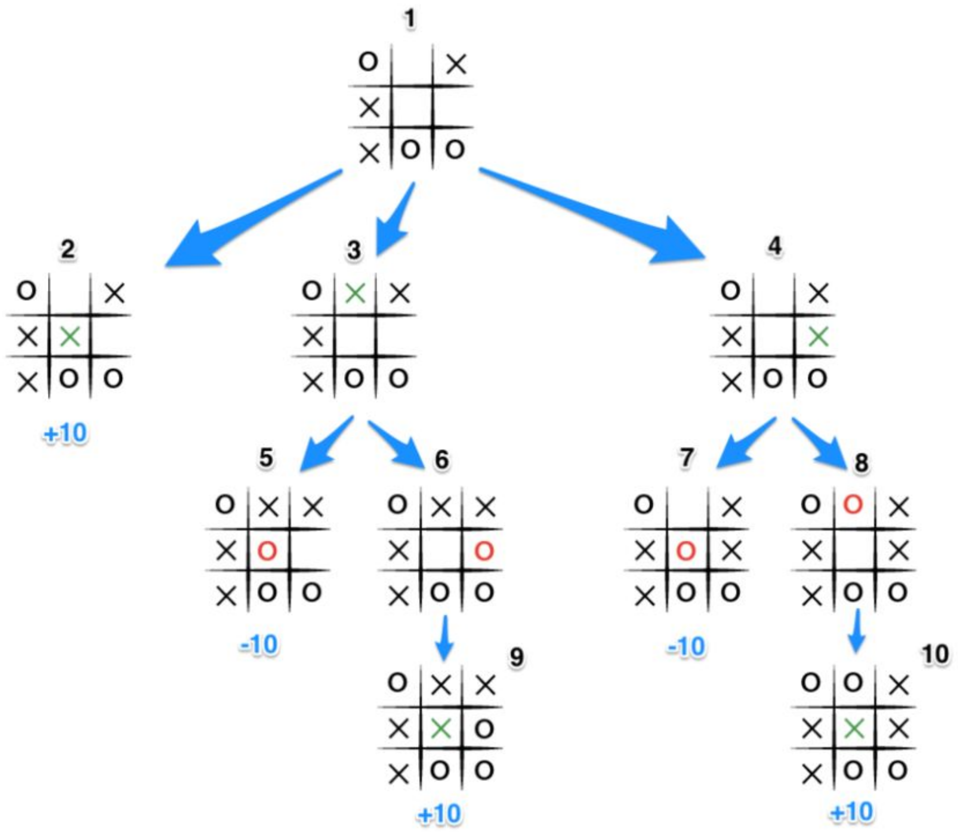
MAX-player X's turn in State 1

- State 1: generates states 2, 3, and 4 and calls minimax on those states
- State 2: goal state

  => X win: return +10 to state 1
- State 3: generates states 5 and 6 and calls minimax on them
- State 4: generates states 7 and 8 and calls minimax on them
- State 5: goal state

  => O win: return -10 to state 3
- State 7: goal state

  => O win: return -10 to state 4



MiniMax: Tic-Tac-Toe

- State 6 and 8: generate states 9 and 10 and call minimax on them
- State 9 and 10: goal states

  => return +10 to states 6 and 8

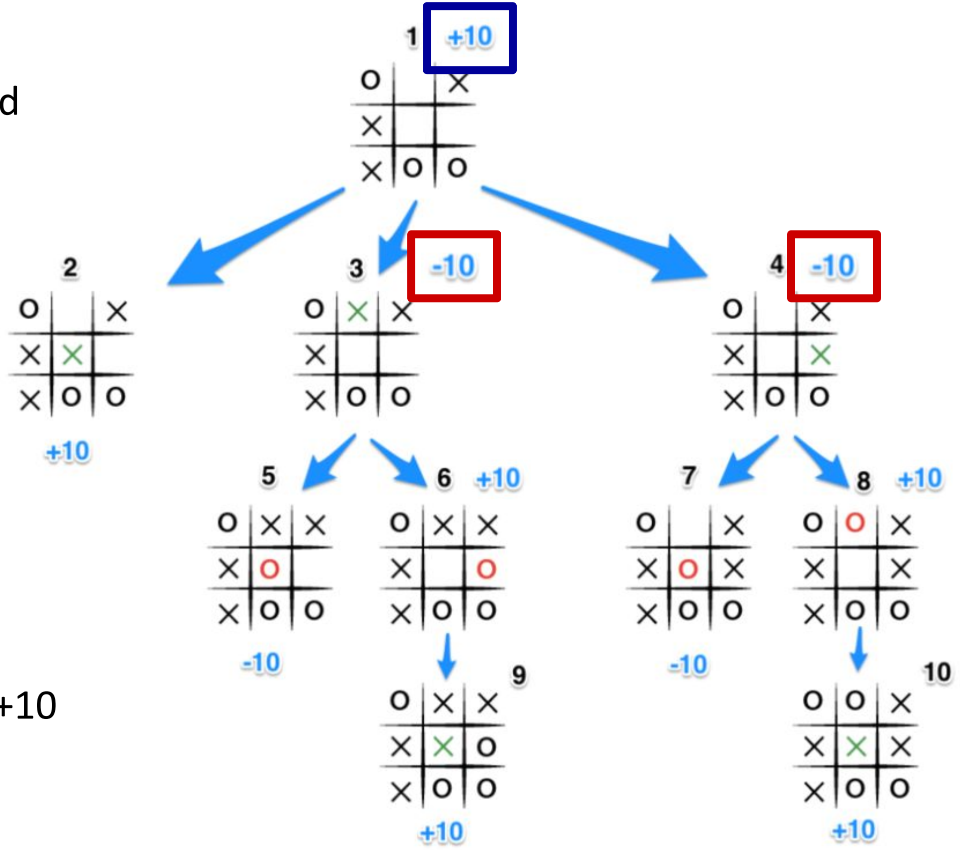
  => return +10 to states 3 and 4
- State 3 and 4: O's turn

  => **MIN**imize score: **MIN**(-10, +10) = -10

  => states 3 and 4 return -10
- State 1: X's turn

  => **MAX**imize score **MAX**(+10, -10, -10) = +10

  => Choose State 2

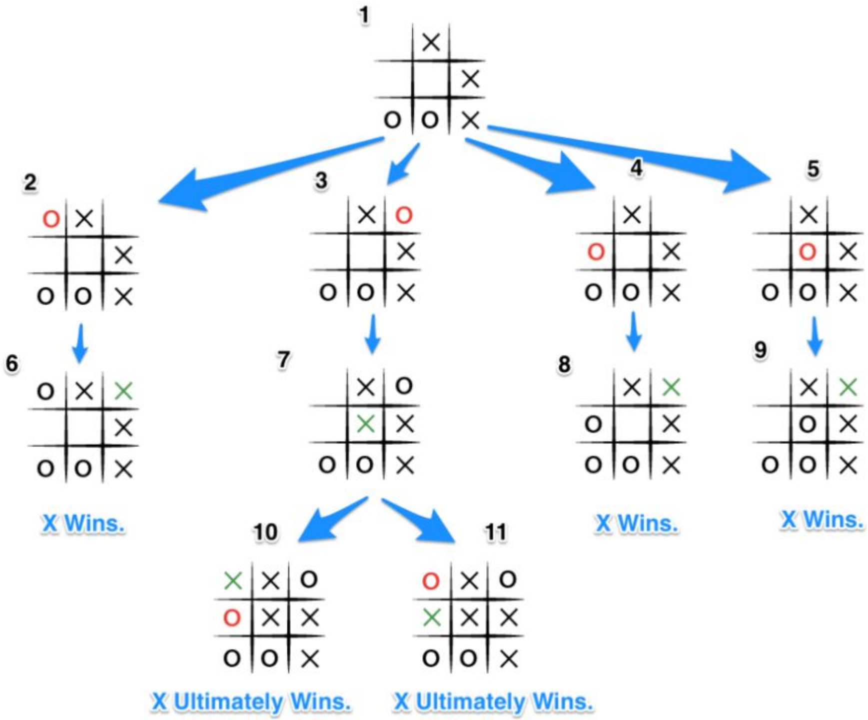


MiniMax: Improvement

Problem

Early demise: algorithm doesn't differentiate between an early and a late defeat

=> O player could choose state 2, 4 or 5 instead of state 3

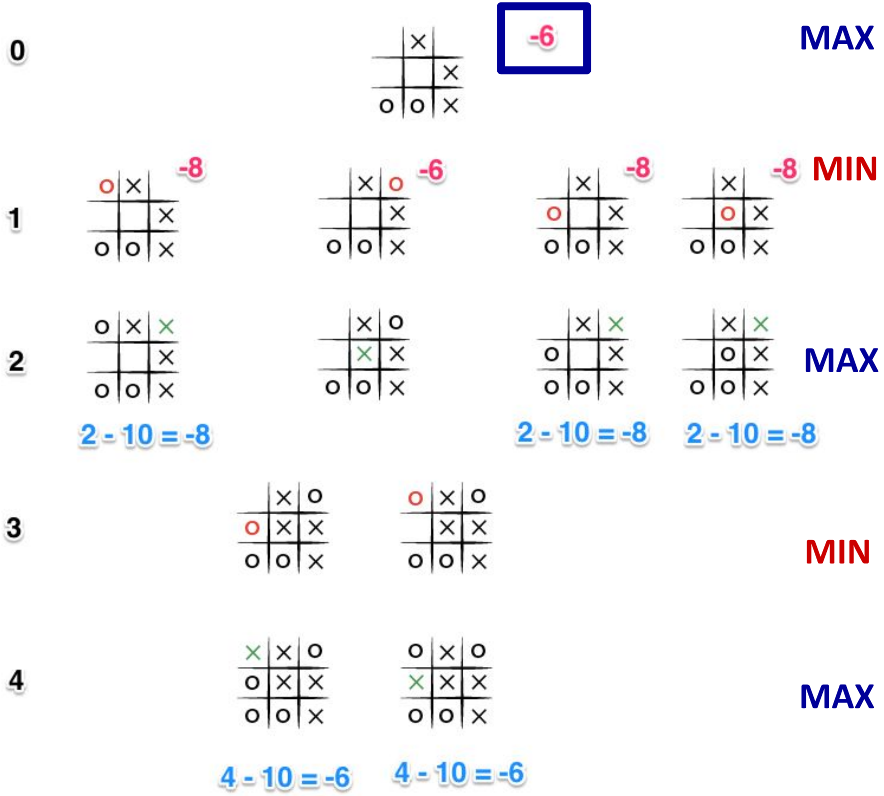


MiniMax: Improvement

Solution

Delay demise: take depth into account for evaluation score

```
function eval(board, depth):
        if maximizer(depth)
             return 10 - depth
        else if minimizer(depth)
             return depth - 10
        else
             return 0
```



Alpha-Beta pruning

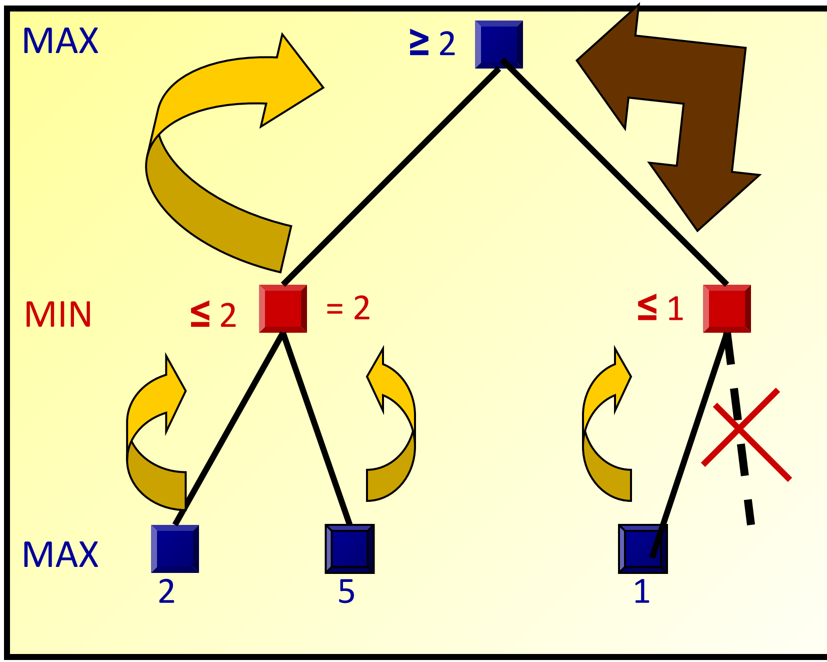
- Optimization for MiniMax
- Instead of:
    - first creating the entire tree (up to depth-level)
    - then doing all propagation
- Interleave the generation of the tree and the propagation of values.

=> some of the obtained values in the tree will provide information that other (non-generated) parts are redundant and do not need to be generated.

Alpha-Beta pruning

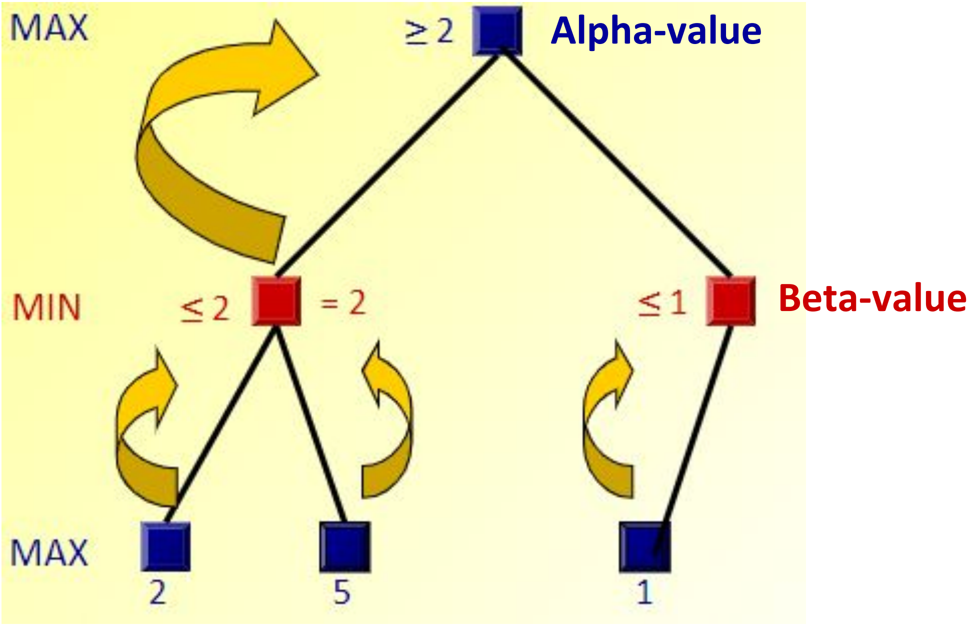
- Generate the tree depth-first, left to right
- Propagate final values of nodes as initial estimates for their parent node

- The MIN-value (1) is already smaller than the MAX-value of the parent (2)
- The MIN-value can only decrease further
- The MAX-value is only allowed to increase
- No point in computing further below this node



Alpha-Beta pruning

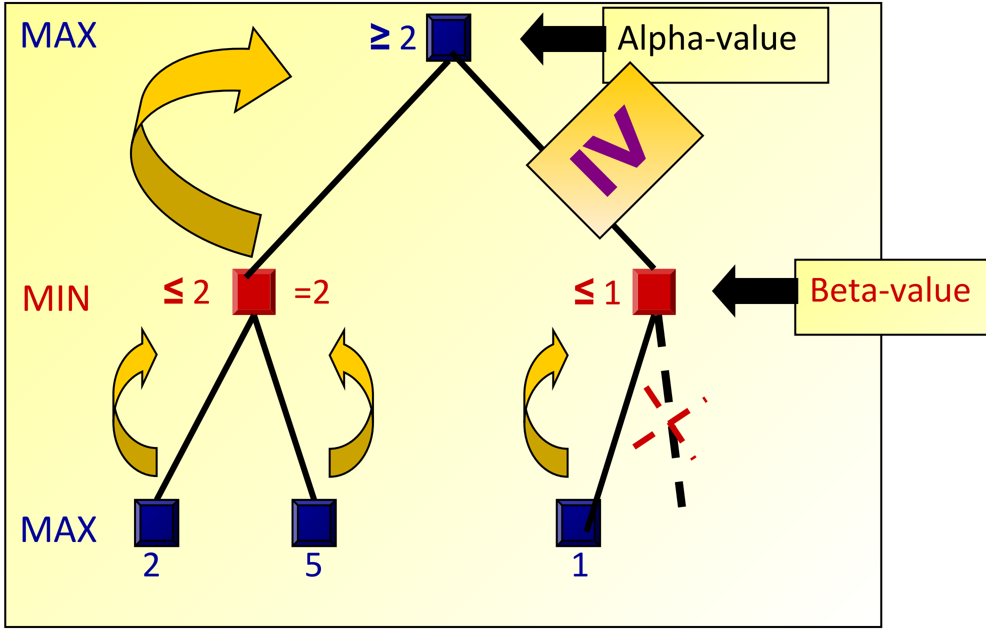
- The values at **MAX** nodes are **Alpha-values**
- The values at **MIN** nodes are **Beta-values**



Alpha-Beta pruning

If an **Alpha-value** is larger or equal to the **Beta-value** of a descendant node:

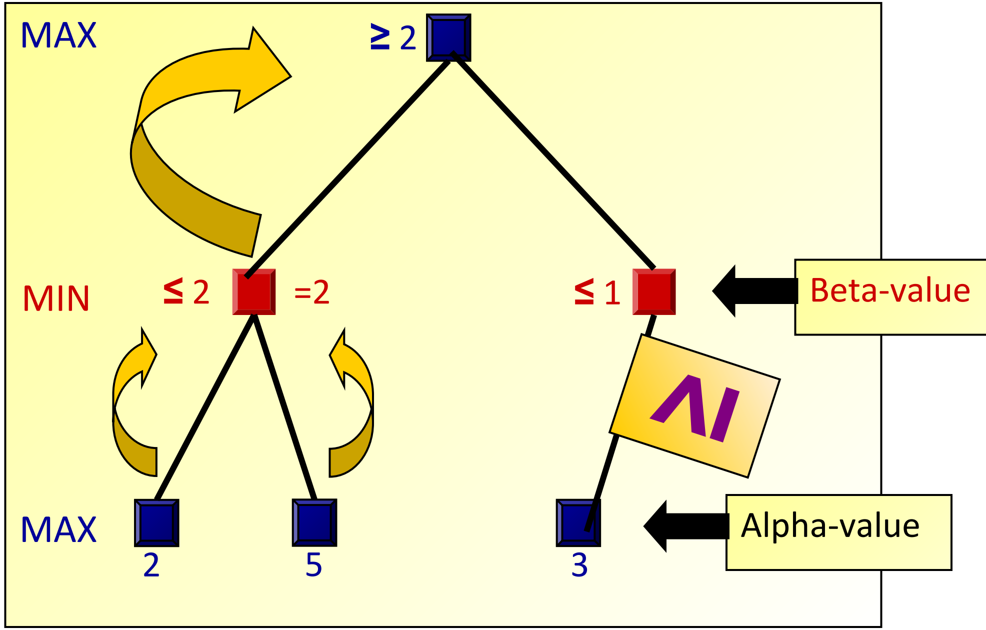
=> Stop generation of the children of the descendant



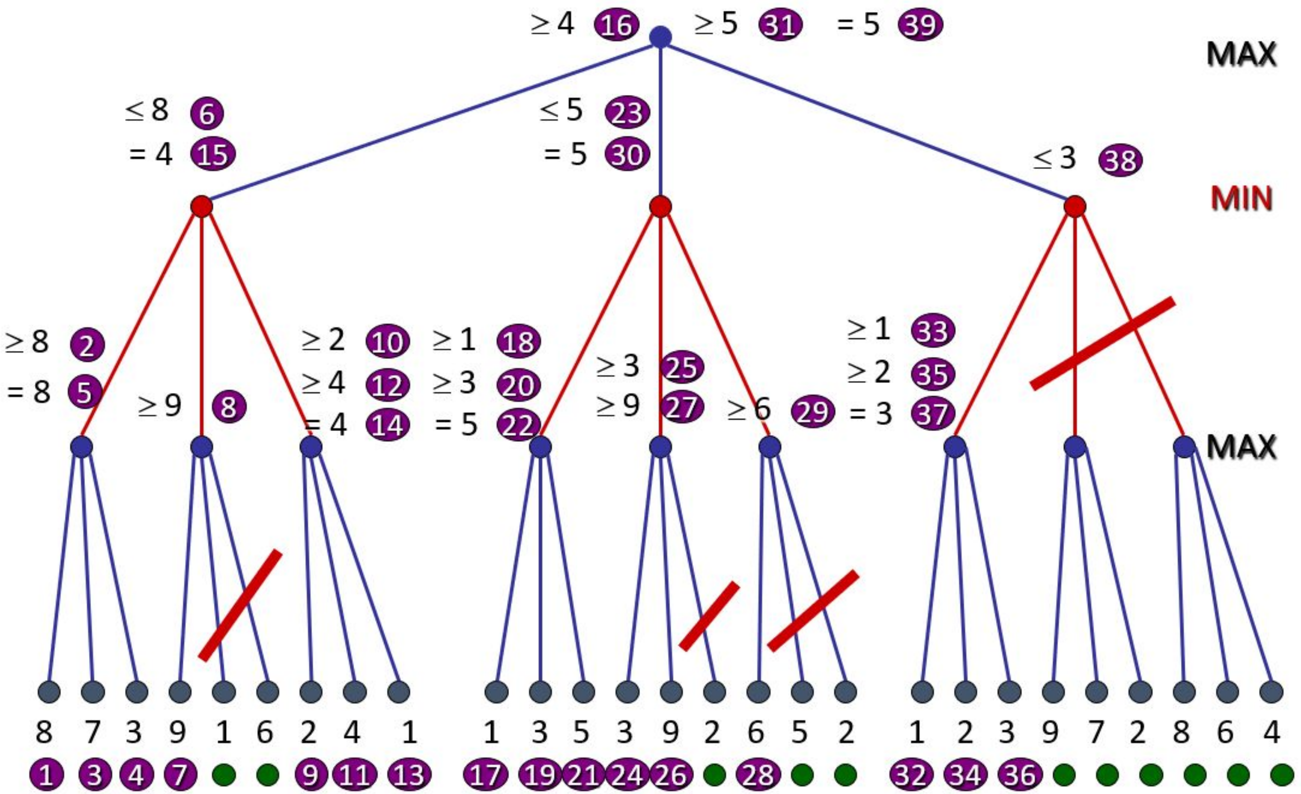
Alpha-Beta pruning

If an **Beta-value** is smaller or equal to the **Alpha-value** of a descendant node:

=> Stop generation of the children of the descendant



Alpha-Beta pruning



Alpha-Beta pruning

Best case gain:

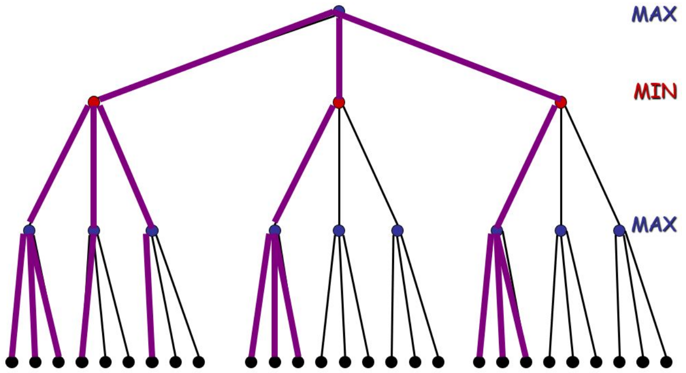
- At every layer, the best node is the **left-most one**:

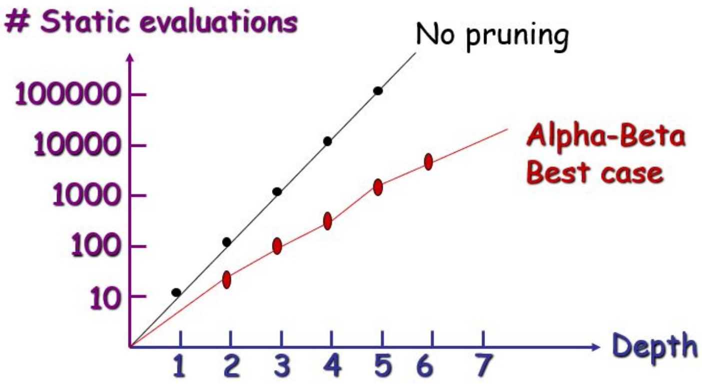
  => only is explored

- Evaluations saved: **O($b^{d/2}$)**

Worst case gain:

- No improvement

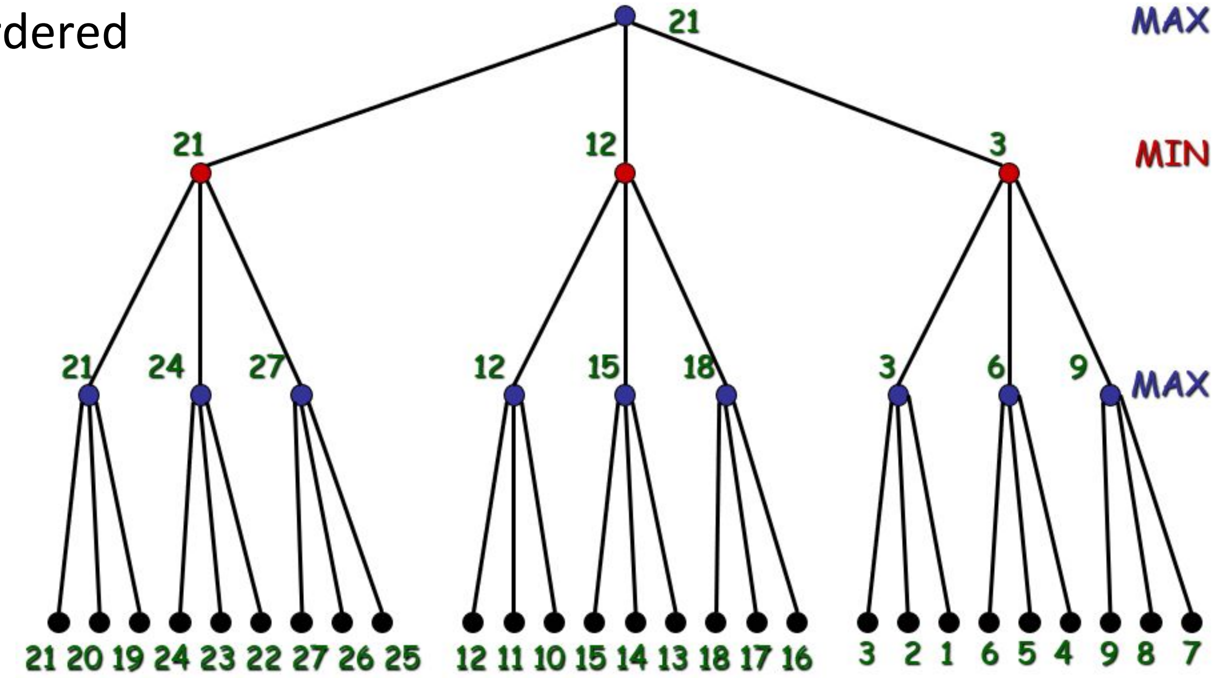




Alpha-Beta pruning

Best case gain:

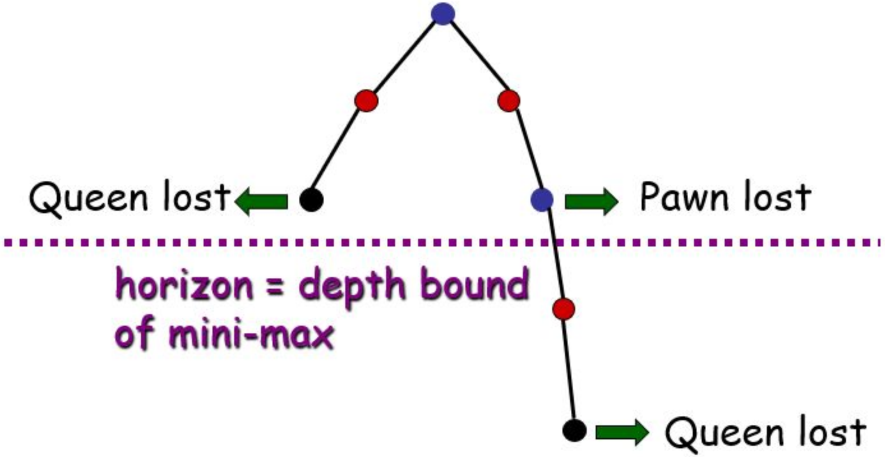
Example of a perfectly ordered tree



Alpha-Beta pruning: Problem

- Depth-bound is limiting factor
- It’s preferable to delay disasters, but they are not prevented

  => possible solution: heuristic continuation



Alpha-Beta pruning: Heuristic continuation

- Change behaviour in certain situations
  - Strategically crucial:
    - e.g. chess: king in danger, pawn can convert to queen, etc

  => extend search beyond the depth bound