Artificial Intelligence

Lecture 2c: Adversarial Search

Luís Paulo Reis

lpreis@fe.up.pt

Director of LIACC – Artificial Intelligence and Computer Science Lab.

Associate Professor at DEI/FEUP – Informatics Engineering Department,

Faculty of Engineering of the University of Porto, Portugal

President of APPIA – Portuguese Association for Artificial Intelligence

Games as Search Problems

- Hostile Agent (opponent) included in the world!
- Unpredictable Opponent => Solution is a Contingency Plan
- Time Limit => Unlikely to find goal! Approximation is necessary
- One of the oldest areas of AI! In 1950 Shannon and Turing created the first Chess programs!
- Chess:
  - Everyone believes that intelligence is needed to play
  - Simple rules but the game is complex
  - World fully accessible to the agent
  - Average branching factor of 35, 50-game play => 35^{100} leaves in the search tree (although there are only 10^{40} legal positions)
- Cutting concepts in the search tree and evaluation function!

Game Types

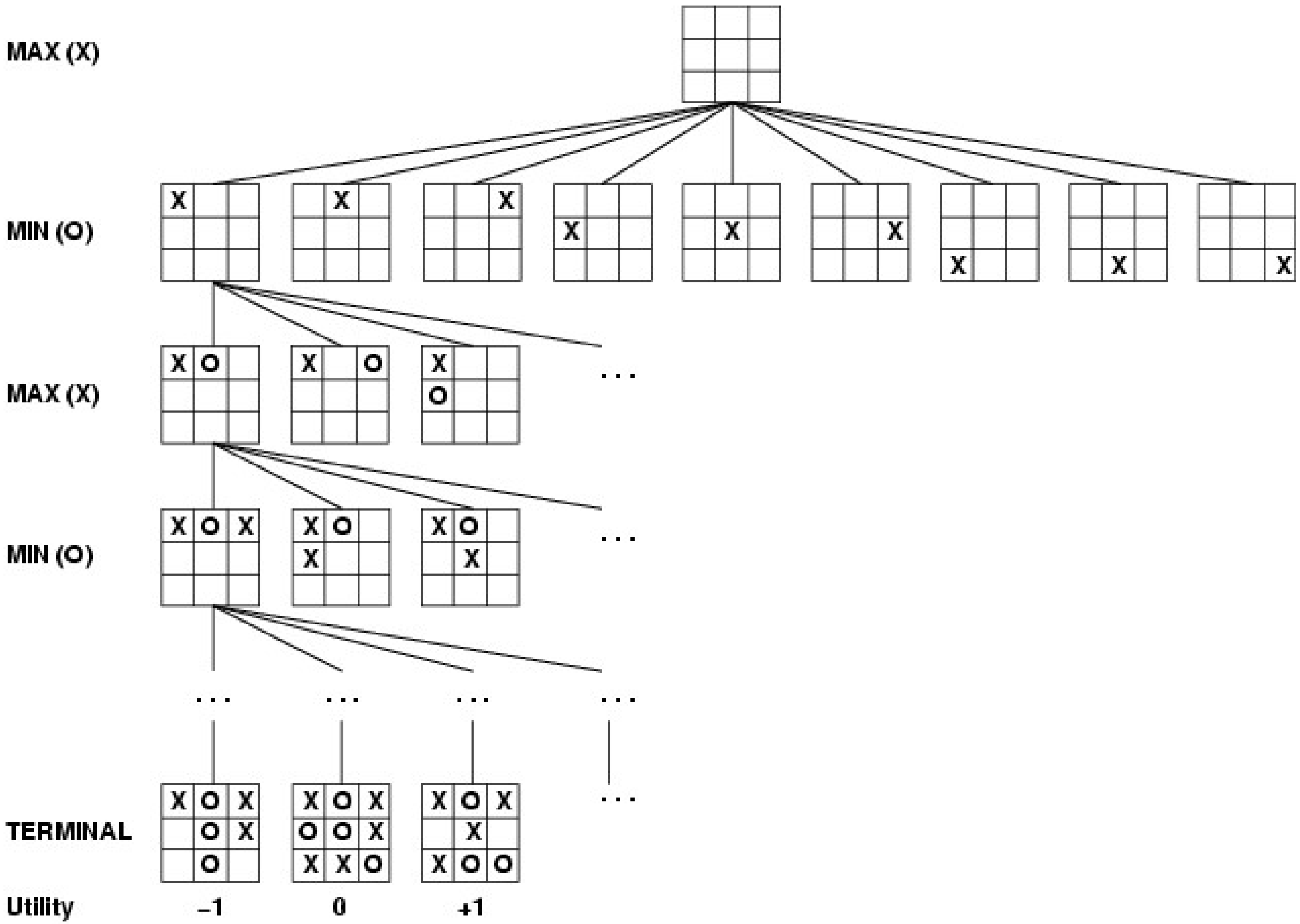
- **Information:**
  - Perfect: Chess, Checkers, Go, Othello, Backgammon, Monopoly
  - Imperfect: Poker, Scrabble, Bridge, King
- **Luck / Deterministic:**
  - Deterministic: Chess, Checkers, Go, Othello
  - Games with luck: Backgammon, Monopoly, Poker, Scrabble, Bridge, King
- **"Attack" plan:**
  - Algorithm for the perfect game
  - Finite horizon, approximate evaluation
  - Tree cuts to reduce costs

Perfect Decisions in Opponent Games - MiniMax Algorithm

- **Game: Search problem with:**
  - **Initial State** (board position and who is next to play)
  - **Set of Operators** (which define the legal movements)
  - **Terminal Test** (which determines whether the game is over or is in a terminal state)
  - **Utility function** (which gives a numerical value to the result of the game, for example 1-win, 0-draw, -1-defeat)
- **Minimax algorithm strategy:**
  - **Generate the complete tree to the end states (limited depth first search)**
  - **Apply the utility function to these states**
  - **Calculate the utility values back to the root of the tree, one layer at a time**
  - **Choose the movement with the highest value!**

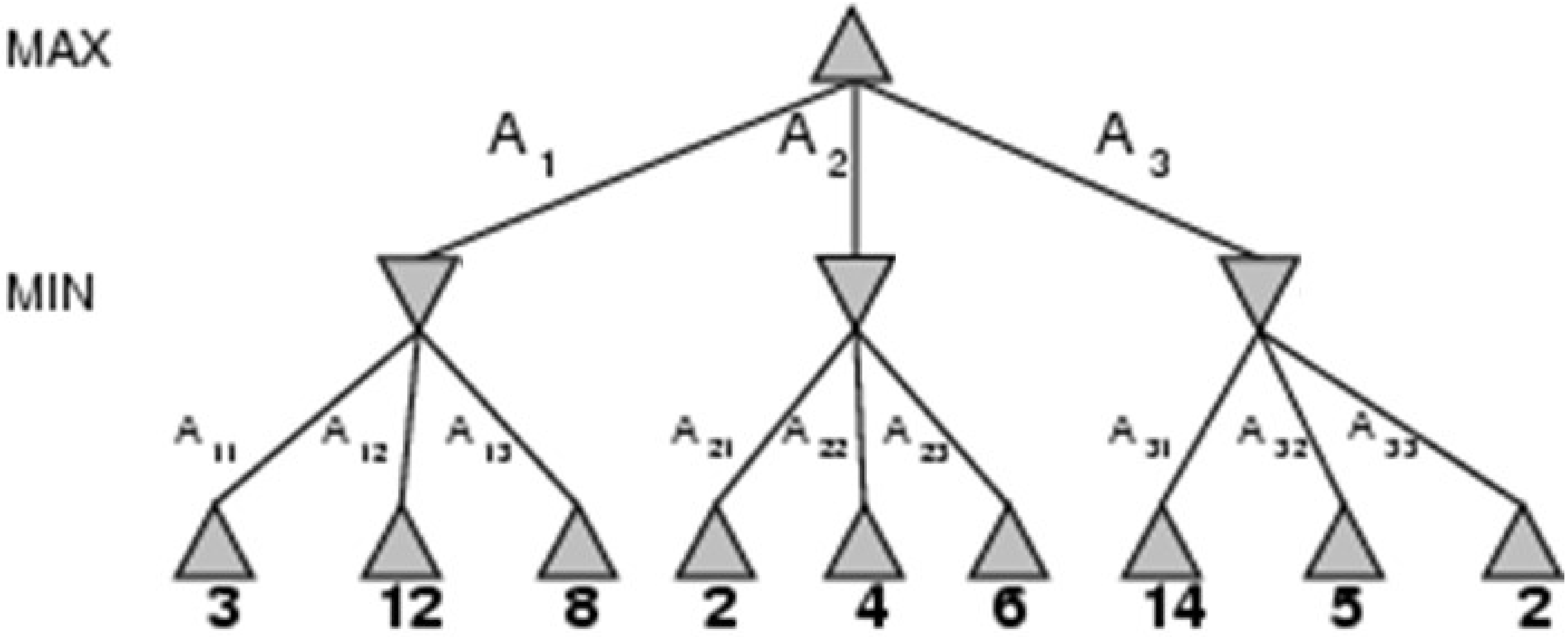
Minimax – Tic-Tac-Toe Game Example

Easy to solve the Tic-Tac-Toe Game using Minimax



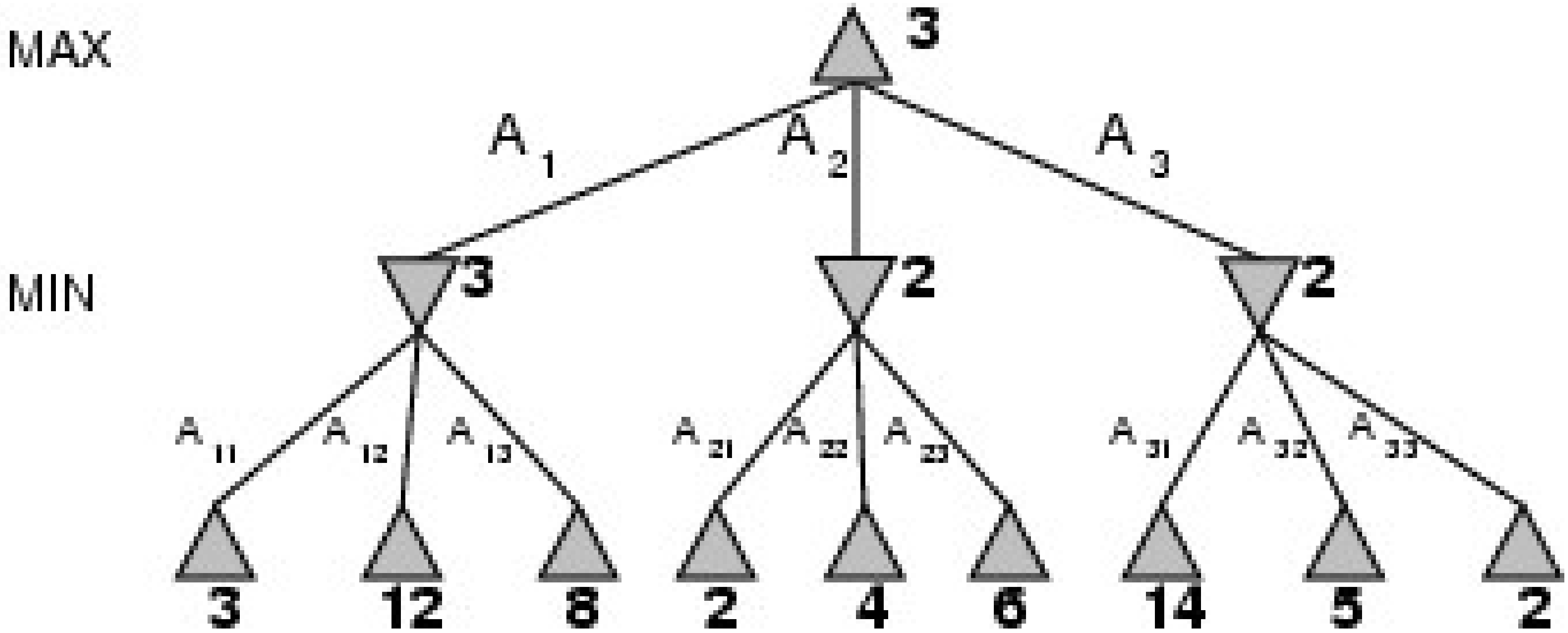
Minimax – General Example

- **Strategy:**
  - Choose the movement that has the highest minimax value = the best that can be achieved against the best responses from the opponent!



Minimax – General Example

- **Strategy:**
  - Choose the movement that has the highest minimax value = the best that can be achieved against the best responses from the opponent!



Minimax Algorithm

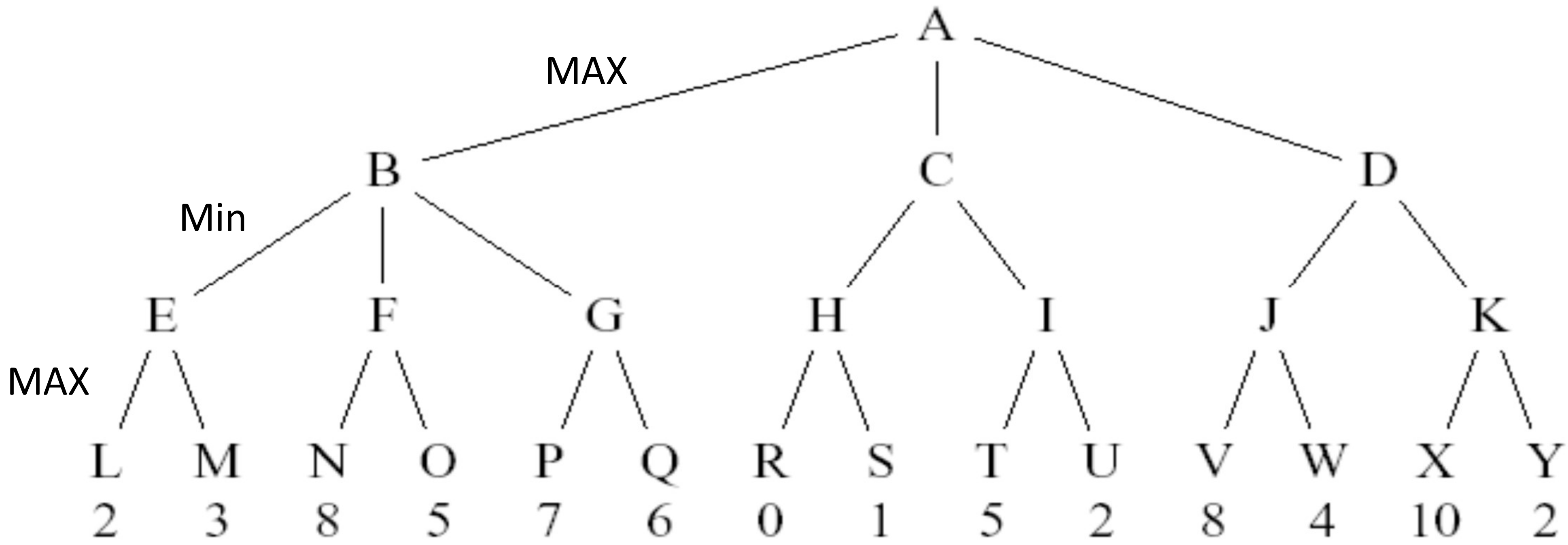
```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v
```

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

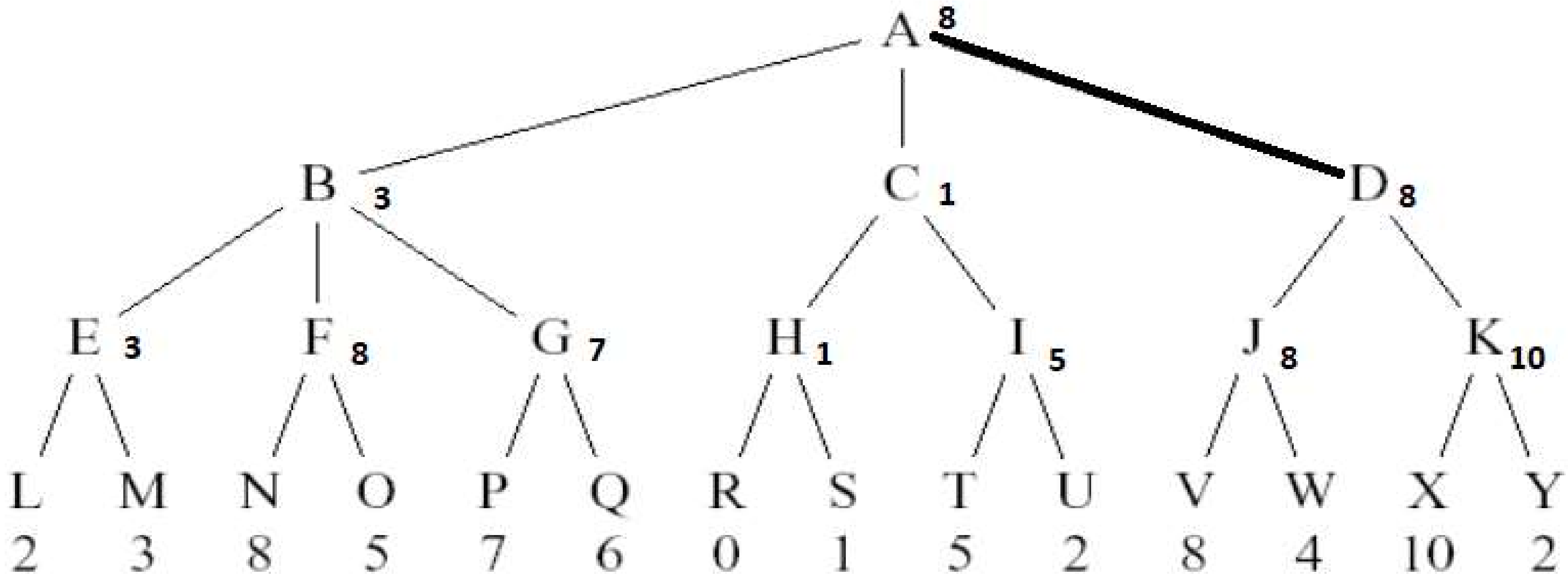
Exercise – Minimax Algorithm

- Assuming that MAX is the first to play, apply the Minimax Algorithm to the following tree, indicating the movement selected by the algorithm and the respective estimated value



Exercise – Minimax Algorithm

- Assuming that MAX is the first to play, apply the Minimax Algorithm to the following tree, indicating the movement selected by the algorithm and the respective estimated value.
- Solution: A -> D (expected value = 8, e.g. oponente will play D->J and we will play J->V)



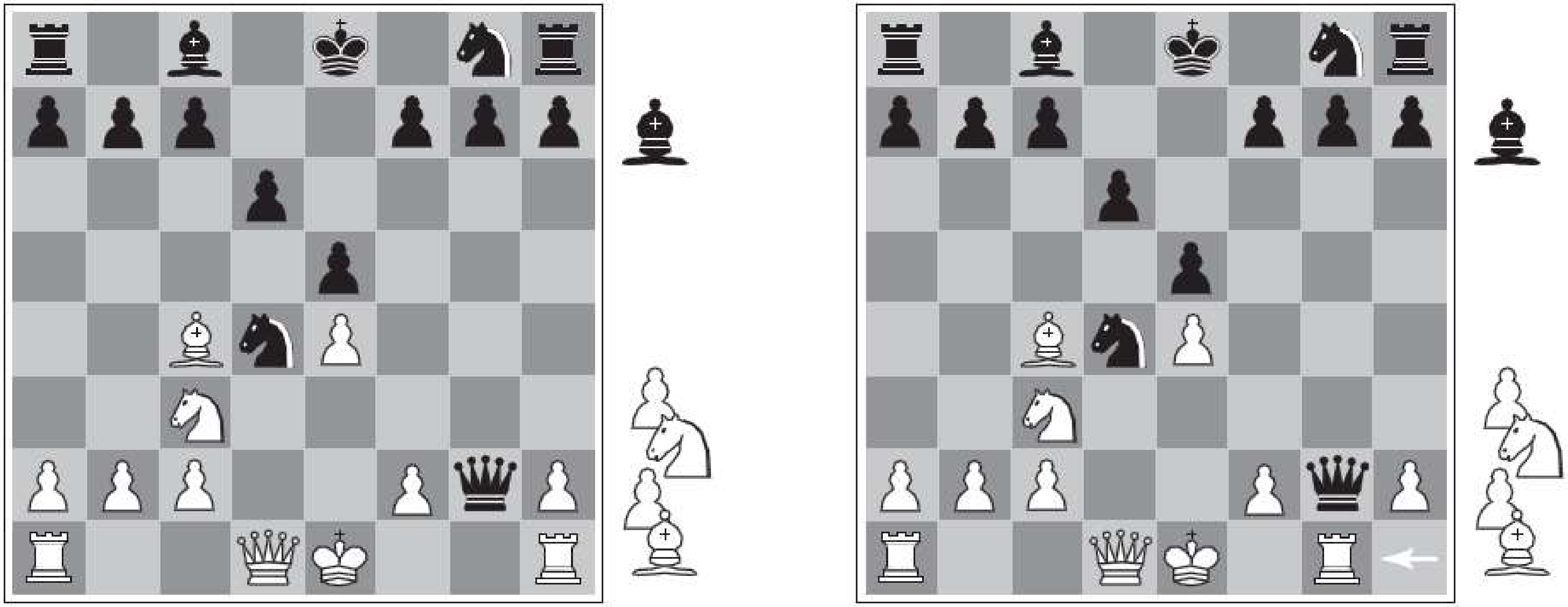
Minimax Proprieties

- **Properties:**
  - Complete? Yes if the tree is finite!
  - Optimal? Yes against an optimal opponent! If no?
  - Time Complexity? O (b^m)
  - Complexity in Space? O (bm) (exploration first in depth)
- **Problem:**
  - Not viable for any minimally complex game
- **Example:**
  - For chess (b = 35, m = 100), $b^m = 35^{100} = 2.5 * 10^{154}$
  - Assuming that 450 million hypotheses are analysed per second => $2 * 10^{138}$ years to arrive at the solution!

Limited Resources

- Assuming we have 100 seconds and explore 10^4 nodes / second, we can explore 10^6 nodes per movement
- Usual approach:
  - Cutting Test: Depth Limit
  - Evaluation Function: Estimated utility (interest) for the position/state
- Example (Chess):
  - Cutting Test: Depth of Analysis n
  - Simple evaluation function = sum of the values of the white pieces in play minus the sum of the values of the black pieces in play!
  - Evaluation function should only be applied to stable positions (in terms of their value). For example, positions with possible captures should be further explored ...
- Another Problem: Horizon problem!

Chess – Evaluation Functions



(a) White to move (b) White to move

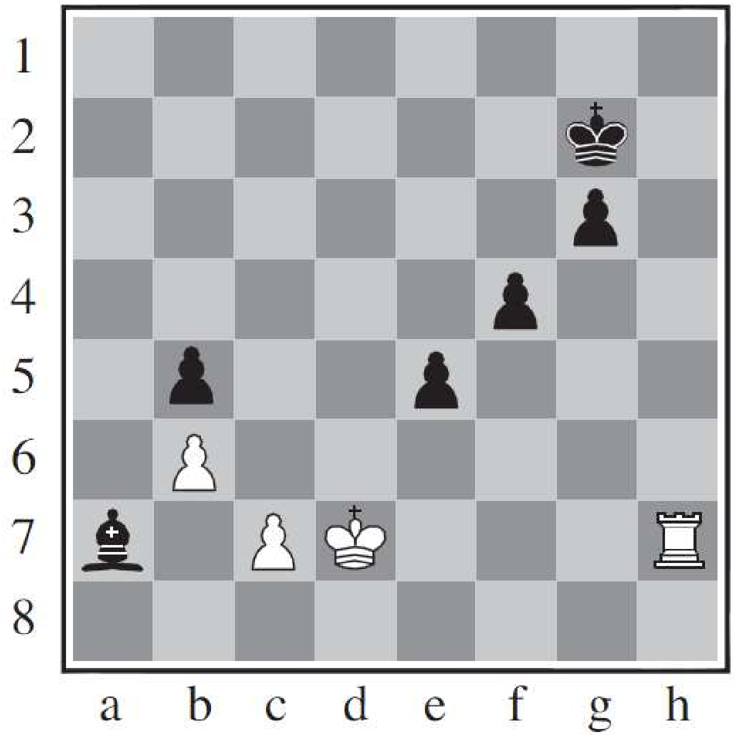
Two chess positions that differ only in the position of the rook at lower right.

In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game.

In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

Chess – Horizon Problem

- Caused by the depth limitation of the search algorithm and manifest when some negative event is inevitable but postponable.
- Because only a partial game tree has been analyzed, it will appear to the system that the event can be avoided when in fact it is not!



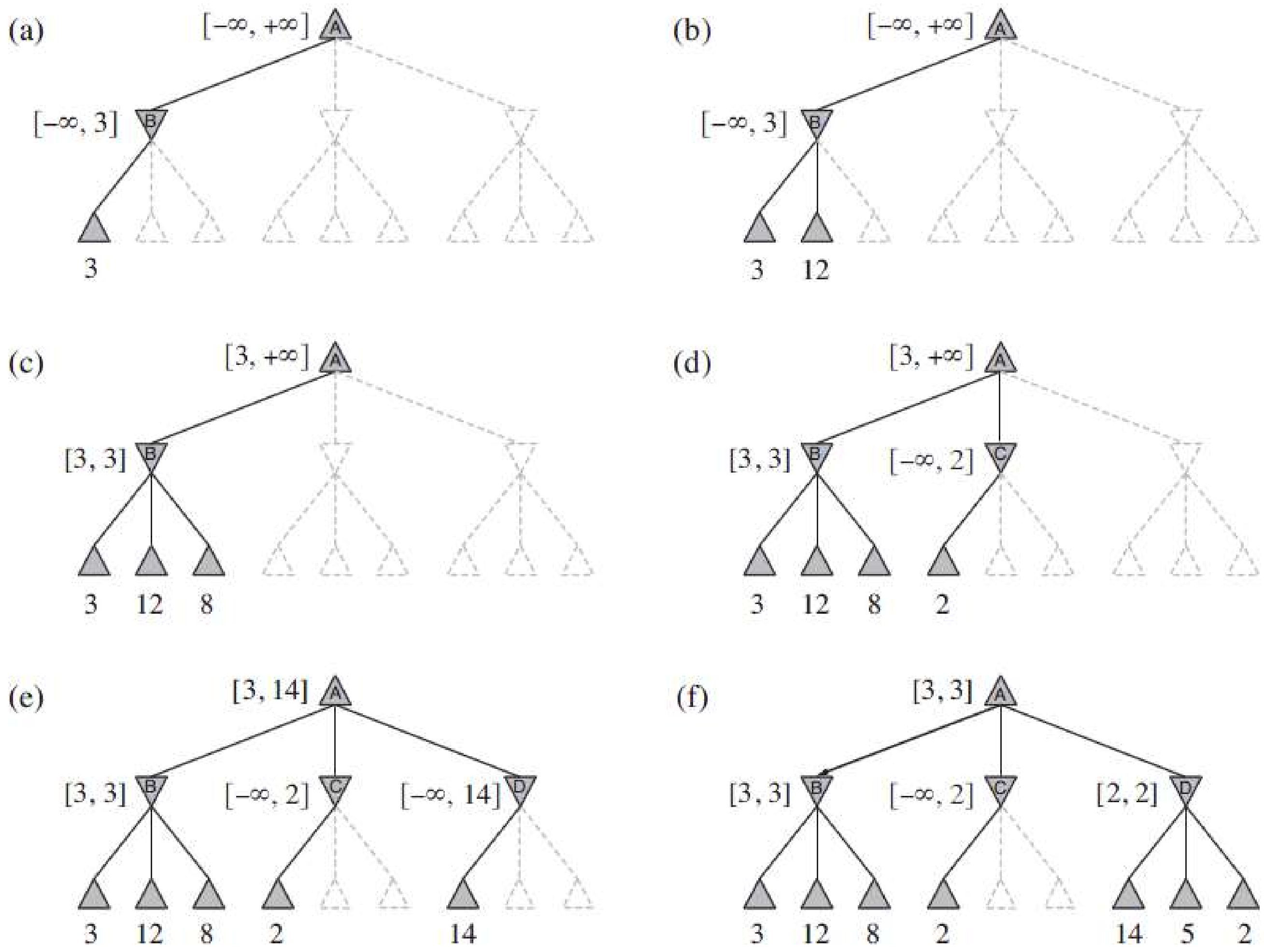
Example: With a limited horizon it seems interesting for Black to check with the pawns and delay the capture of the bishop (beyond the horizon)

Cuts - *MinimaxCutoff*

- ***MinimaxCutoff* is identical to *MinimaxValue* except:**
  - Terminal-Test is replaced by *Cutoff*
  - Utility is replaced by Evaluation (which calculates an evaluation of the reached position)
- **Does it work in practice?**
  - If $b^m = 10^6$ with $b = 35 \Rightarrow m = 4$
- **Playing chess with depth 4 is absolutely miserable!**
  - Depth 4 => Newbie Player
  - Depth 8 => PC, Good Human Player
  - Depth 12 => Deep Blue, Kasparov

Alpha-Beta Cuts (1)

- α is the best value (for Max) found so far on the current path
- If V is worse than α, Max should avoid it => cut off the branch
- β is defined in the same way for Min



Alpha-Beta Cuts (2)

- **Alpha-Beta cuts do not affect the final result**
- **Good ordering improves cutting efficiency**
  - Essential to reason about node ordering
- **Perfectly arranged: Time Complexity = O ($b^{m/2}$)**
  - Doubles search depth
  - Depth 8 => Good chess player
- **Good example of the value of reasoning about which computations are relevant:**
  - This is essential in Intelligent Systems

Minimax with Alpha-Beta Cuts

```
function ALPHA-BETA-SEARCH(state) returns an action
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v

function MIN-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```

Minimax with Alpha-Beta in Python

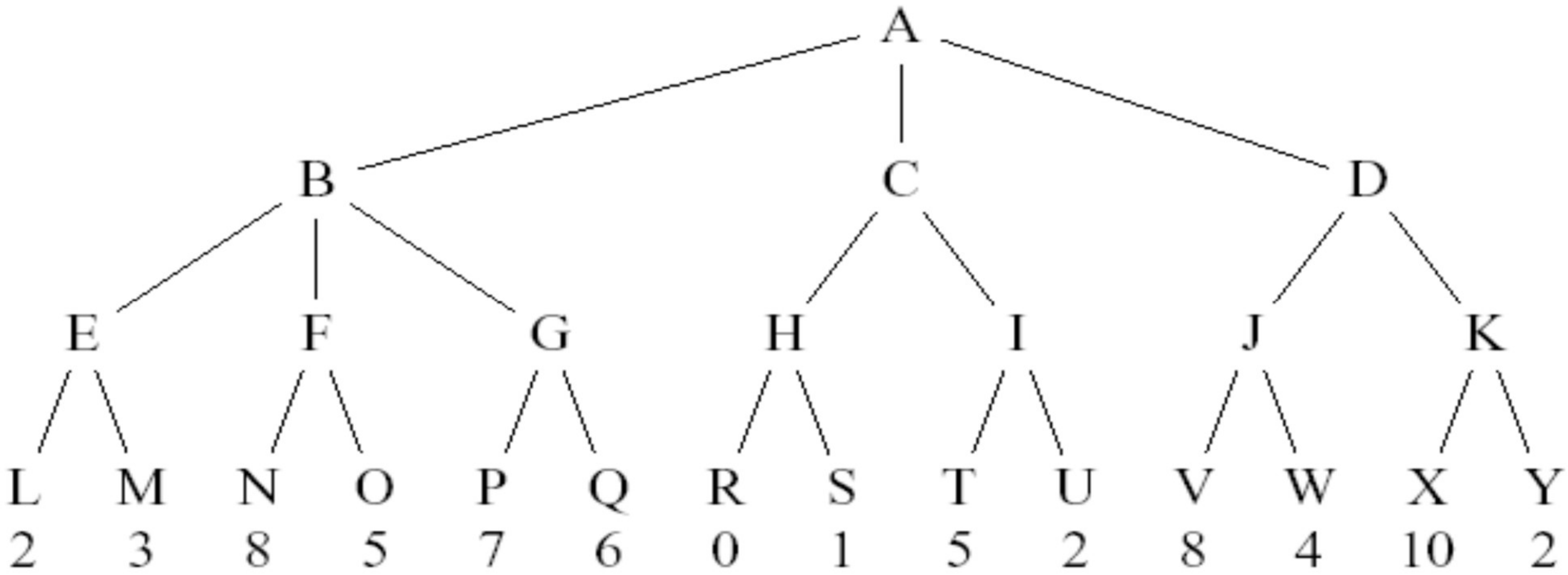
```
def minimax(state, depth, playerMax, alpha, beta):
    if depth == 0 or state.isEndState():    return evalFunction(state)

    if playerMax:
        maxEval = -math.inf
        for move in state.validMoves():
            evaluation = minimax(move,  depth-1, False, alpha , beta)
            maxEval = max(maxEval, evaluation)
            alpha = max(alpha, evaluation)
            if beta <= alpha:    break
        return maxEval

        minEval = math.inf
        for move in state.validMoves():
            evaluation = minimax(move, depth-1, True, alpha, beta)
            minEval = min(minEval, evaluation)
            beta = min(beta, evaluation)
            if beta <= alpha:   break
         return minEval
```

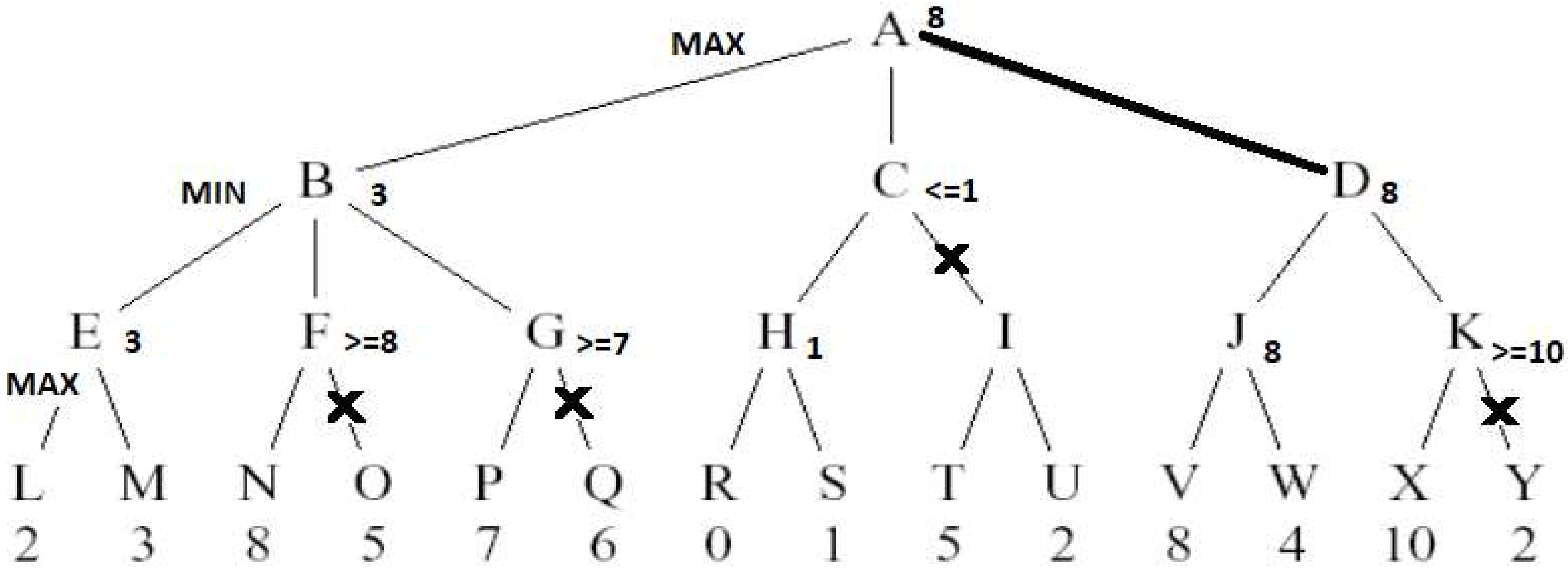
Exercise – MINIMAX with Cuts

- Assuming that MAX is the first to play, apply the Minimax Algorithm with Alpha-Beta cuts to the following tree, indicating the movement selected by the algorithm. Indicate graphically and justify all the cuts made when applying the Minimax algorithm



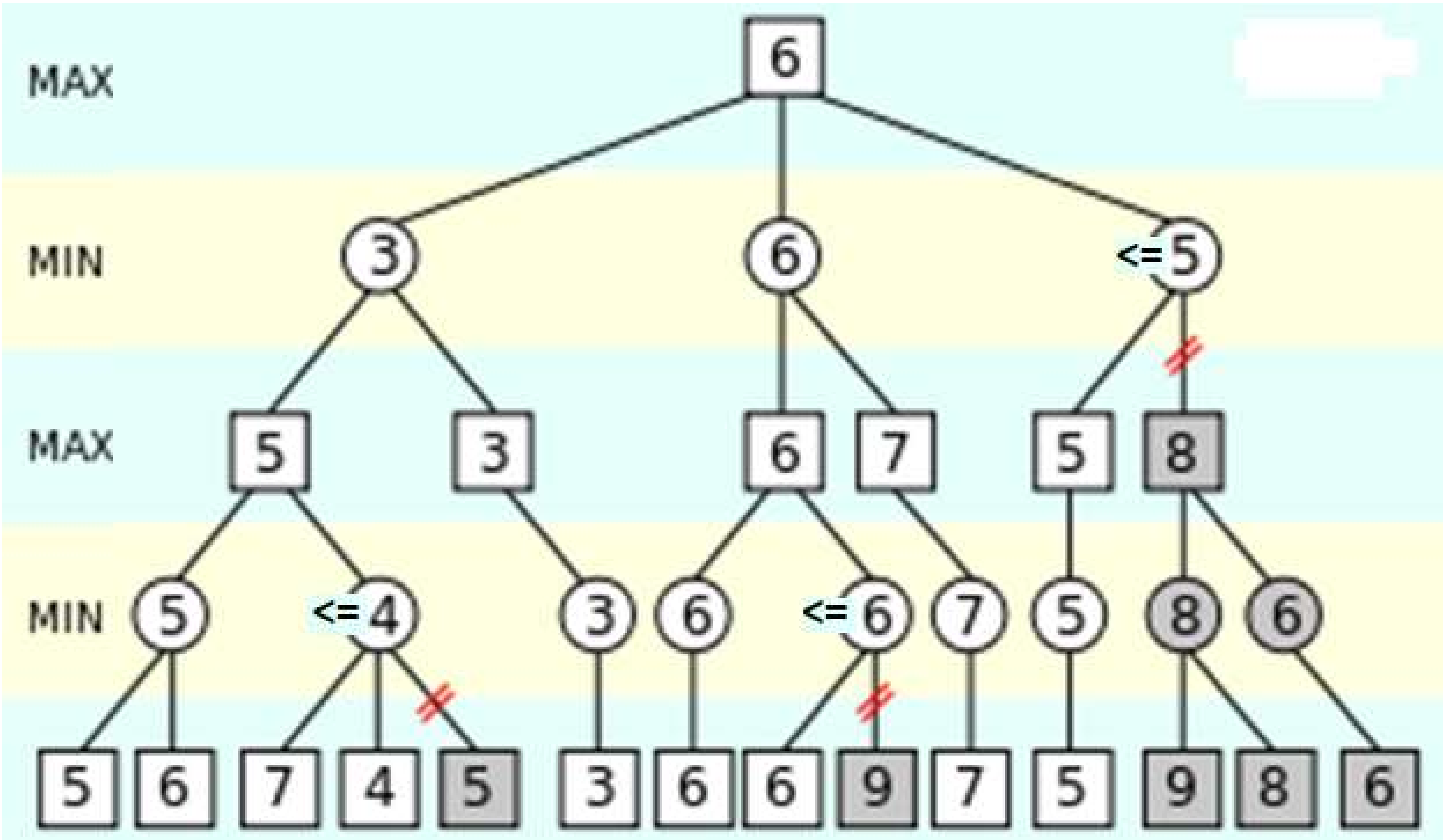
Exercise – MINIMAX with Cuts

- Assuming that MAX is the first to play, apply the Minimax Algorithm with Alpha-Beta cuts to the following tree, indicating the movement selected by the algorithm. Indicate graphically and justify all the cuts made when applying the Minimax algorithm
- Solution:



Alpha-Beta Cuts - Example

The grayed-out subtrees don't need to be explored (when moves are evaluated from left to right), since it is known that the group of subtrees as a whole yields the value of an equivalent subtree or worse, and as such cannot influence the final result. The Max and Min levels represent the turn of the player and the adversary, respectively



Exercise – MINIMAX with Cuts

- Assuming that MAX is the first to play, apply the Minimax Algorithm with Alpha-Beta cuts to a tree with three levels, a branch factor 3 and with the following values of the evaluation function for the final line:

[8 6 3   1 10 3   15 20 6   7 4 6   25 10 4   4 3 5   1 20 4   12 1 10   4 22 10]

- Indicate graphically and justify all the cuts made when applying the Minimax algorithm

Negamax

- Negamax search is a variant form of minimax search that relies on the zero-sum property of a two-player game.
- This algorithm relies on the fact that

  max(a,b)= -min(-a,-b)
- to simplify the implementation of the minimax algorithm.
- More precisely, the value of a position to player A in such a game is the negation of the value to player B. Thus, the player on move looks for a move that maximizes the negation of the value resulting from the move: this successor position must by definition have been valued by the opponent.
- The reasoning of the previous sentence works regardless of whether A or B is on move. This means that a single procedure can be used to value both positions.
- This is a coding simplification over minimax, which requires that A selects the move with the maximum-valued successor while B selects the move with the minimum-valued successor.

Negamax - Pseudocode

```
function negamax(node, depth, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    value := −∞
    for each child of node do
        value := max(value, −negamax(child, depth − 1, −color))
    return value
```

```
(* Initial call for Player A's root node *)
negamax(rootNode, depth, 1)
```

```
(* Initial call for Player B's root node *)
negamax(rootNode, depth, −1)
```

Negamax Alpha-Beta - Pseudocode

```
function negamax(node, depth, α, β, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node

    childNodes := generateMoves(node)
    childNodes := orderMoves(childNodes)
    value := −∞
    foreach child in childNodes do
        value := max(value, −negamax(child, depth − 1, −β, −α, −color))
        α := max(α, value)
        if α ≥ β then
            break (* cut-off *)
    return value
```

```
(* Initial call for Player A's root node *)
negamax(rootNode, depth, −∞, +∞, 1)
```

Deterministic Games

- **Checkers:**
  - **Chinook ended the 40-year reign of human champion Marion Tinsley in 1994**. It used a database for match ends defining the perfect way to win for all positions involving 8 or less pieces (in total 443748401247 positions). Nowadays checkers is a solved problem.
- **Chess:**
  - **Deep Blue defeated human world champion Gary Kasparov** in a 6-matches game **in 1997**. Deep Blue searched 200 million positions per second and used an extremely sophisticated evaluation function and (undisclosed) methods to extend some lines of search beyond depth 40!
- **Othello:**
  - **Human champions refuse to compete with computers** because they have no chance! (b between 5 and 15)

Deterministic Games - Go

- **Go (2015):**
  - Human champions refuse to compete with computers because machines cannot play reasonably (b>300)
- **Go (2017-2020):**
  - AlphaGo (Fan, Lee), AlphaGo Master, AlphaGo Zero and AlphaZero! Machines beat 100-0 human champions and previous machines.

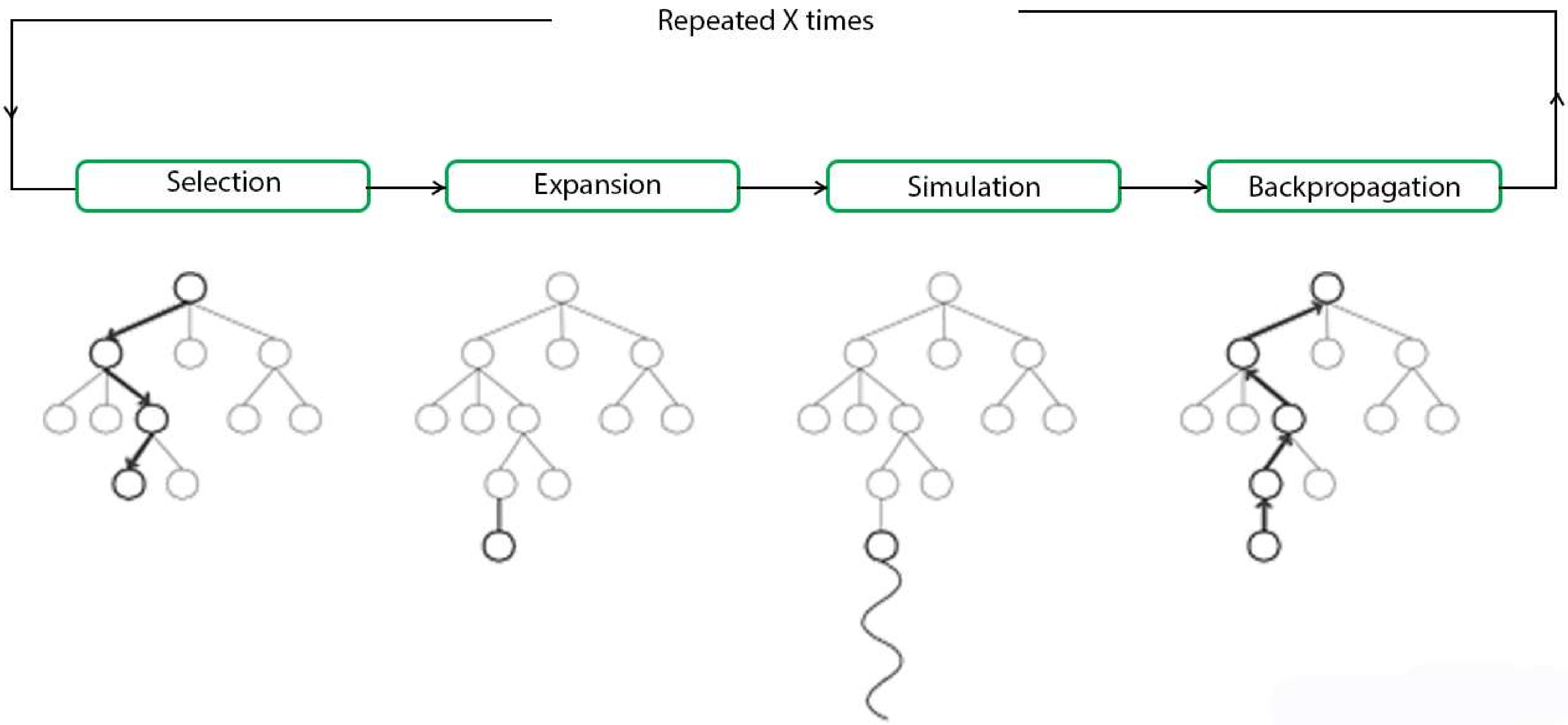
Monte Carlo Tree Search

- **Monte Carlo tree search (MCTS)** is a heuristic search algorithm for some kinds of decision processes, most notably those employed in software that plays board games
- **MCTS was combined with Reinforcement Learning and Deep Neural Networks for computer Go**
- It has been used in other **board games** like **chess** and shogi, **games with incomplete information** such as bridge and poker, as well as in **turn-based-strategy video games**
- **Works well for games with very large branching factor**
- **Good when it is very difficult to define an appropriate evaluation function**
- **Game-Changing Algorithm behind DeepMind's AlphaGo, and AlphaZero**

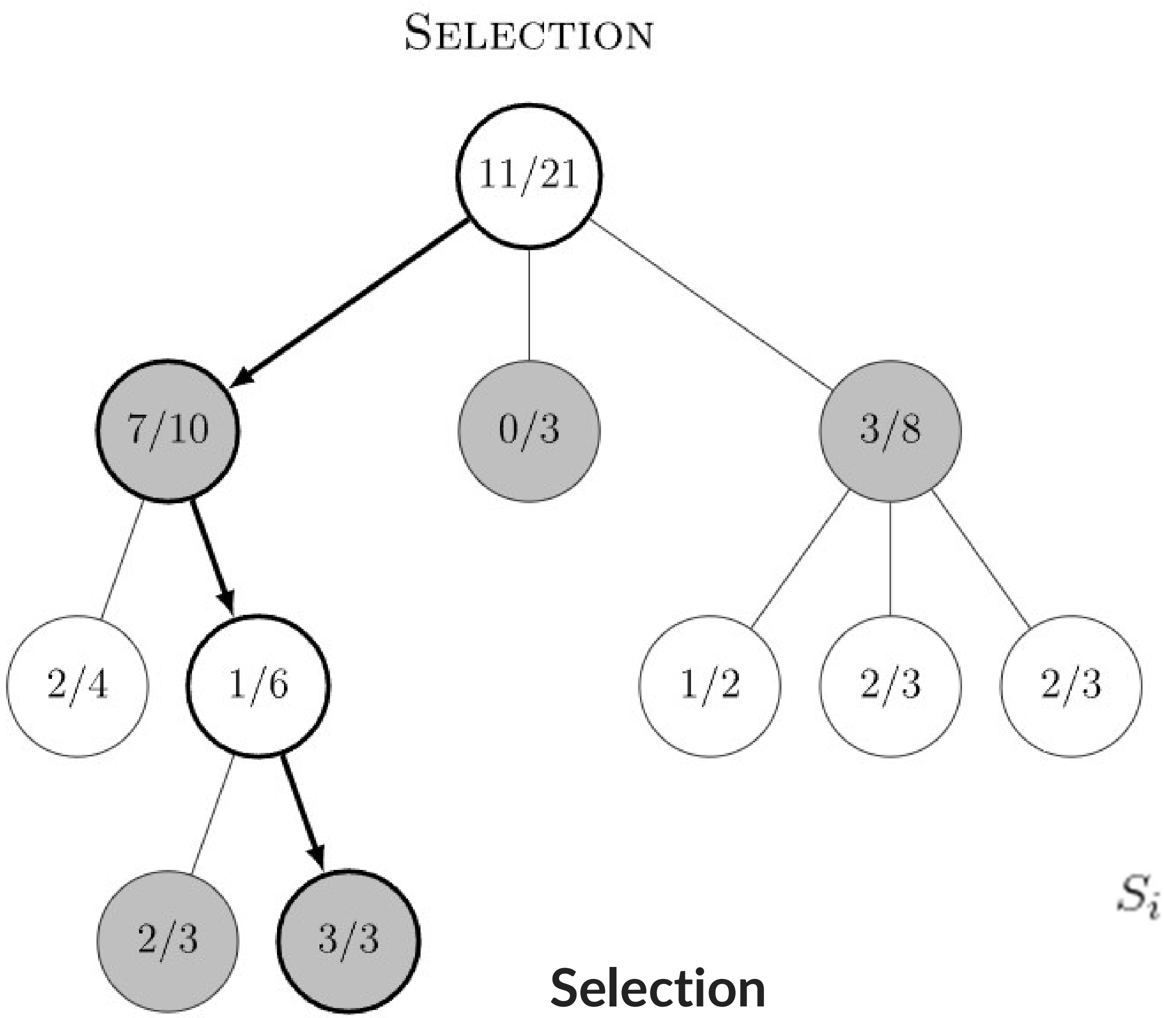
MCTS Algorithm

- **Selection:** Start from root R and select successive child nodes until a leaf node L is reached. The root is the current game state and a leaf is any node that has a potential child from which no simulation (playout) has yet been initiated. There should be a way of biasing choice of child nodes that lets the game tree expand towards the most promising moves, which is the essence of MCTS.
- **Expansion:** Unless L ends the game decisively (e.g. win/loss/draw) for either player, create one (or more) child nodes and choose node C from one of them. Child nodes are any valid moves from the game position defined by L.
- **Simulation:** Complete one random playout from node C. This step is sometimes also called playout or rollout. A playout may be as simple as choosing uniform random moves until the game is decided (for example in chess, the game is won, lost, or drawn).
- **Backpropagation:** Use the result of the playout to update information in the nodes on the path from C to R.

Monte Carlo Tree Search



Monte Carlo Tree Search

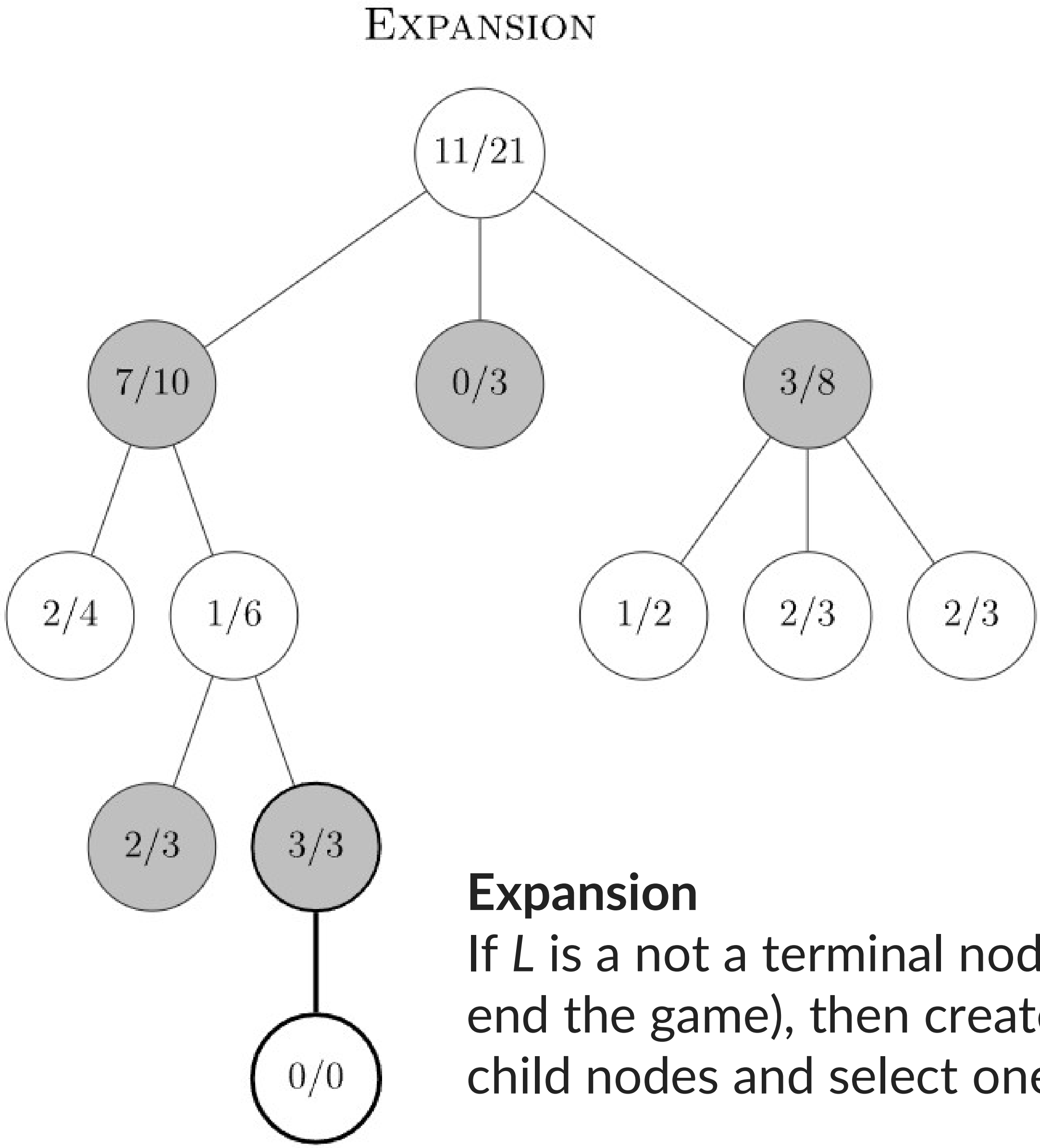


$$S_i = x_i + C\sqrt{\frac{\ln(t)}{n_i}}$$

Selection

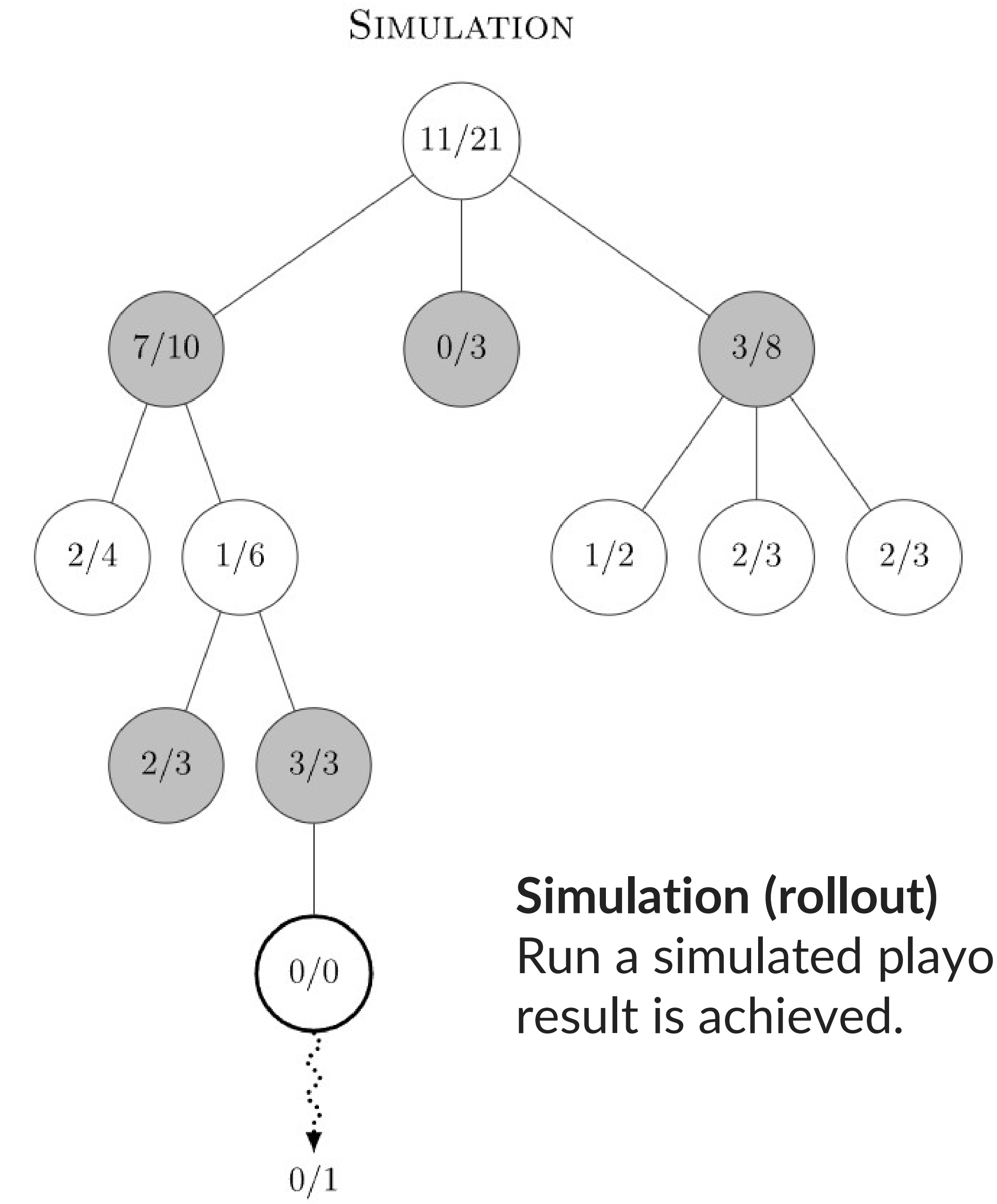
Selecting good child nodes, starting from the root node R, that represent states leading to better overall outcome (win).

Monte Carlo Tree Search



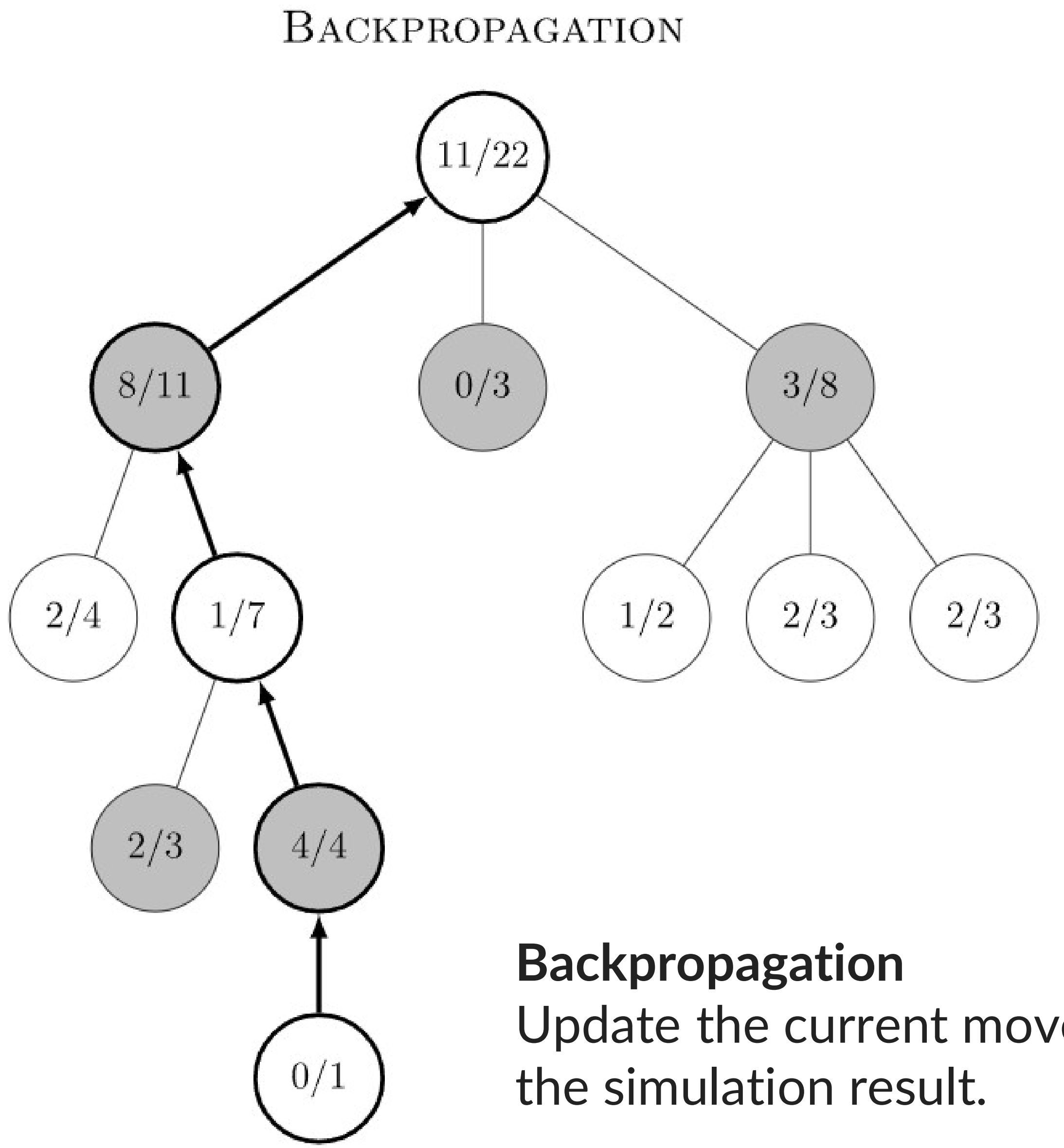
Expansion
If *L* is a not a terminal node (i.e. it does not end the game), then create one or more child nodes and select one (*C*).

Monte Carlo Tree Search



Simulation (rollout)
Run a simulated playout from C until a result is achieved.

Monte Carlo Tree Search



Backpropagation
Update the current move sequence with the simulation result.

Monte Carlo Tree Search

```python
def monte_carlo_tree_search(root):

    while resources_left(time, computational power):
        leaf = traverse(root)
        simulation_result = rollout(leaf)
        backpropagate(leaf, simulation_result)

    return best_child(root)


# function for node traversal
def traverse(node):
    while fully_expanded(node):
        node = best_uct(node)

    # in case no children are present / node is terminal
    return pick_unvisited(node.children) or node
```

Monte Carlo Tree Search

```
# function for the result of the simulation
def rollout(node):
    while non_terminal(node):
        node = rollout_policy(node)
    return result(node)


# function for randomly selecting a child node
def rollout_policy(node):
    return pick_random(node.children)


# function for backpropagation
def backpropagate(node, result):
    if is_root(node) return
    node.stats = update_stats(node, result)
    backpropagate(node.parent)


# function for selecting the best child
# node with highest number of visits
def best_child(node):
    pick child with highest number of visits
```

Monte Carlo Tree Search

Advantages:

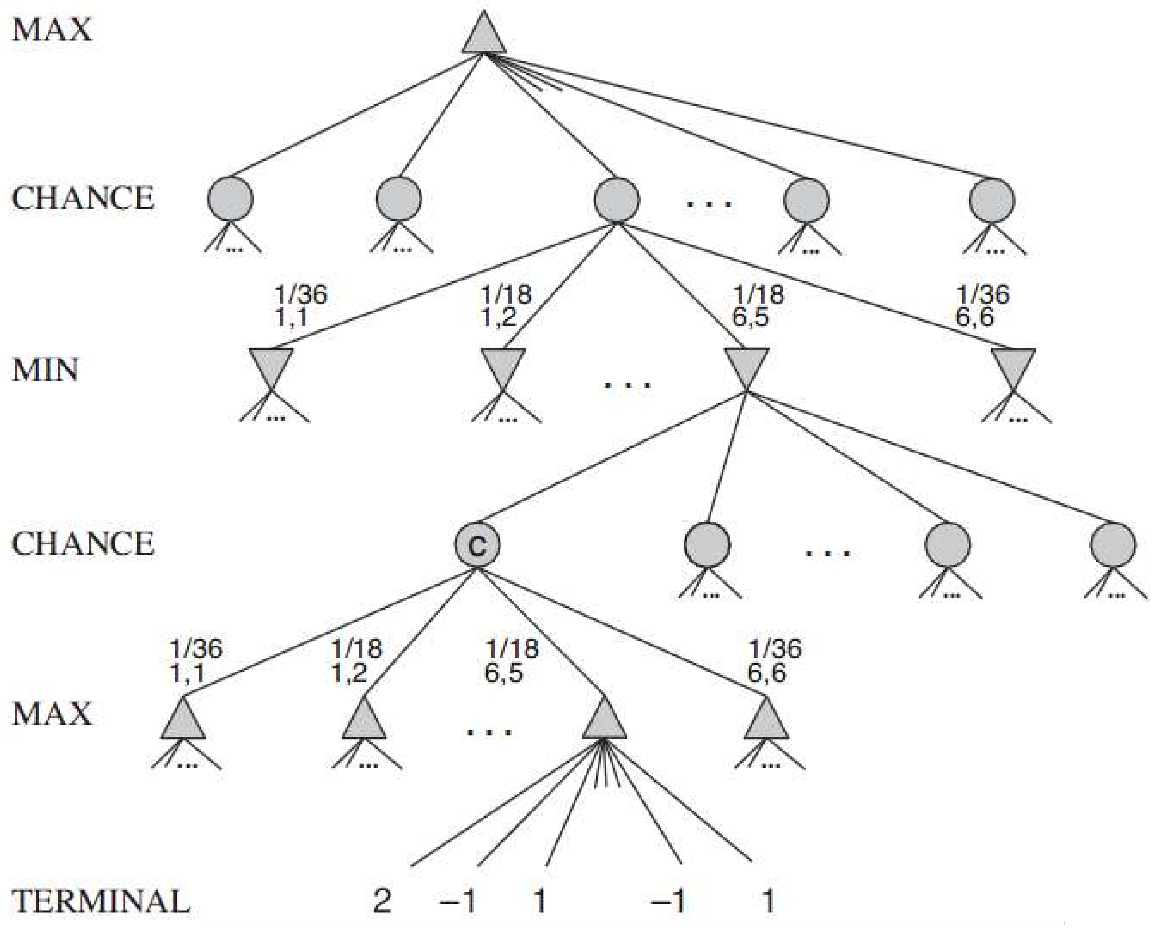
- MCTS is a simple algorithm to implement
- Monte Carlo Tree Search is a heuristic algorithm. MCTS can operate effectively without any knowledge in the particular domain, apart from the rules and end conditions, and can can find its own moves and learn from them by playing random playouts

Disadvantages:

- As the tree growth becomes rapid after a few iterations, it requires a huge amount of memory
- There is a bit of a reliability issue with Monte Carlo Tree Search. In certain scenarios, there might be a single branch or path, that might lead to loss against the opposition when implemented for those turn-based game. This is mainly due to the vast amount of combinations and each of the nodes might not be visited enough number of times to understand its result or outcome in the long run
- MCTS algorithm needs a huge number of iterations to be able to effectively decide the most efficient path. So, there is a bit of a speed issue there

Games of Luck - *ExpectiMiniMax*

- In many games, unlike chess, there are external events that affect the game, such as drawing a card or rolling a dice!
- Examples: Card games, Backgammon, Scrabble, ...
- Search tree must include probability nodes!
- Decision is made based on the expected value!
- *ExpectiMiniMax*



ExpectiMiniMax - Pseudocode

```
function expectiminimax(node, depth)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    if the adversary is to play at node
        // Return value of minimum-valued child node
        let α := +∞
        foreach child of node
            α := min(α, expectiminimax(child, depth-1))
    else if we are to play at node
        // Return value of maximum-valued child node
        let α := -∞
        foreach child of node
            α := max(α, expectiminimax(child, depth-1))
    else if random event at node
        // Return weighted average of all child nodes' values
        let α := 0
        foreach child of node
            α := α + (Probability[child] × expectiminimax(child, depth-1))
    return α
```

Exercises – Adversarial Search/Games

Formulate some of the following 2-player board games as a adversarial search problem (game) and design an intelligent agent capable of playing it, using Minimax with alpha-beta cuts:

1. Chess
2. Shogi
3. Checkers
4. Connect 4
5. Attaxx
6. Chinese Checkers
7. Otelo (Reversi)
8. Abalone
9. Hex
10. Omega Chess
11. Tori Shogi
12. Quoridor (Normal/Kids) (Gigamic)
13. Quarto (Gigamic)
14. Quixo (Gigamic)
15. Quads (Gigamic)
16. Sahara (Gigamic)
17. Pentaminoes (8x8)
18. Stratego
19. Link Five
20. Mancala (normal/4x8)
21. Fanorana
22. Nine Mens Morris
23. Alquerque
24. Tablut
25. Surakarta
26. Terrace
27. Go
28. Dots and Boxes
29. Dots and Hexagons
30. Amazons
31. Scrabble (visible letters and known draws)
32. Tic tac toe (normal/ memory/ movement, 3D)
33. Dominoess (visible pieces and known draws)
34. Backgammon (known draws)
35. Paper & Pencil Racing
36. Arimaa
37. Gipf
38. Lines of Action
39. Connections
40. Fanorama
41. Hexxagon
42. Jungle Game
43. Seega
44. Halma (2 players)
45. Quits (Gigamic)
46. Pylos (Gigamic)
47. Tantrix (2 players)
48. Carcassone (2 players, known draws)
49. Blokus
50. Sputnik (Gigamic)
51. Katamino (Gigamic)
52. Gobblet (Gigamic)

Summary - Games

- A game can be defined by the **initial state** (how the board is set up), the legal **actions** in each state, the **result** of each action, a **terminal test** (which says when the game is over), and a **utility function** that applies to terminal states
- In two-player zero-sum games with perfect information, **minimax algorithm** can select **optimal moves** by a depth-first enumeration of the game tree
- The **alpha–beta search** algorithm computes the same optimal move as minimax, but achieves much greater **efficiency** by eliminating subtrees that are provably irrelevant
- Usually, it is not feasible to consider the whole game tree (even with alpha–beta), so we need to cut the search off at some point and apply a **heuristic evaluation** function that estimates the utility of a state

Summary - Games

- **Working with games is extremely interesting (but also dangerous…)**
  - Easy to test new ideas!
  - Easy to compare agents with other agents
  - Easy to compare agents with humans!
- **Games illustrate several interesting points of AI:**
  - Perfection is unattainable => it is necessary to aproximate!
  - It is a good idea to think about what to think!
  - Uncertainty restricts the allocation of values to states!
- **Games work for AI like Formula 1 for car building!**

U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Artificial Intelligence

Lecture 2c: Adversarial Search

Luís Paulo Reis

lpreis@fe.up.pt

Director of LIACC – Artificial Intelligence and Computer Science Lab.

Associate Professor at DEI/FEUP – Informatics Engineering Department,

Faculty of Engineering of the University of Porto, Portugal

President of APPIA – Portuguese Association for Artificial Intelligence