ARTIFICIAL INTELLIGENCE

LEI/3, LMA/3, MBE/1

University of Beira Interior, Department of Informatics

Hugo Pedro Proença

hugomcp@di.ubi.pt, **2022/23**

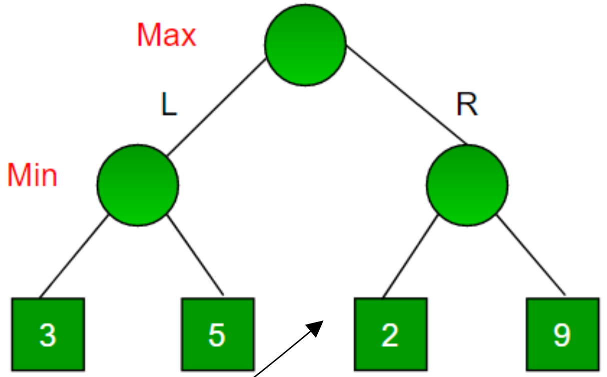
State Space Search - Antagonism

- □ Up to now, we saw different strategies for searching (traversing) a state space tree/graph in order to find the (best) path for a solution.
- □ We assume that the best action depends only on the current state of the environment and of the behavior of the intelligent agent.
- □ However, there are many cases where the environment is populated with agents that have different goals, and – in most cases – antagonist goals.
- □ In such circumstances, I am not only interested in achieving my goals, but also in avoiding that my opponent achieves his own goals.
- □ **Game theory** is a branch of mathematics used to model the interaction between different agents (players), which are equally rational, in a context with predefined rules (of playing or maneuvering) and outcomes.
- □ Every player is selfish and tries to maximize the reward obtained using a particular strategy. Hence, a **game** is defined as a set of players, actions, strategies, and a final playoff for which all the players are competing.
  - □ Also, it is important to minimize the reward that opponents can obtain, to guarantee that they do not assume an advantageous position in the game

Minimax Search

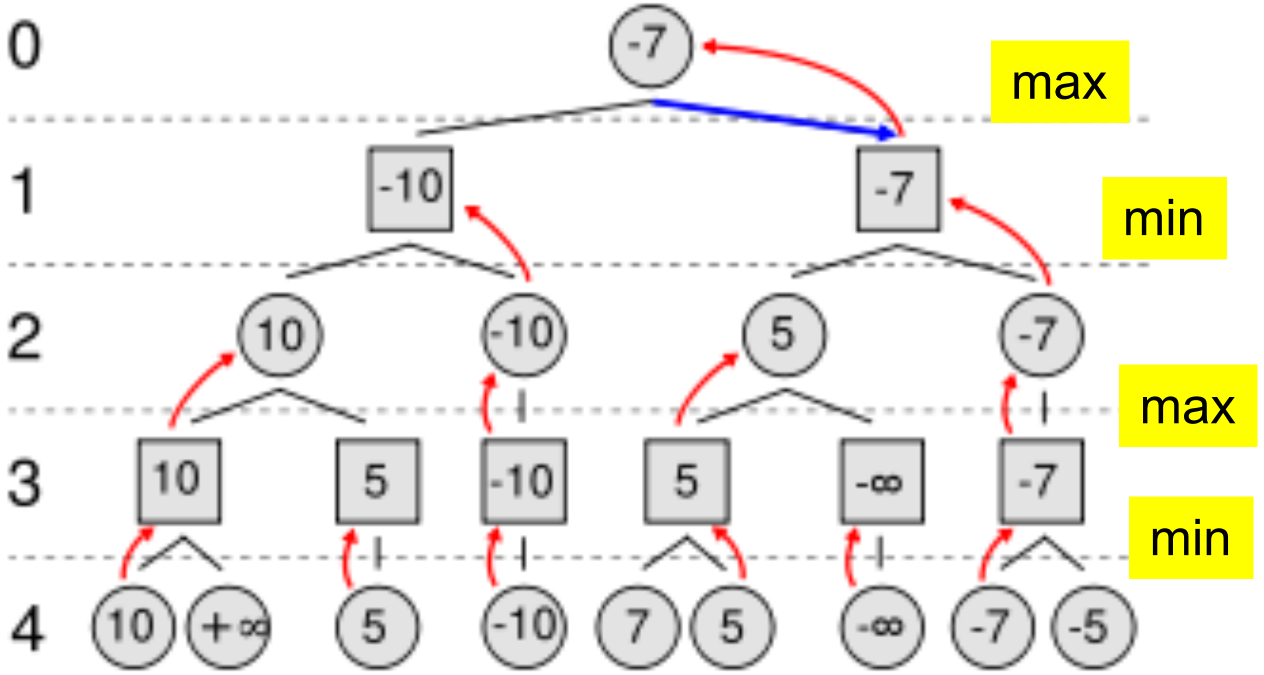
- □ Minimax is an algorithm used in decision making and game theory to find the optimal move for a player, assuming that the opponent also plays optimally.
- □ It is the most classical choice for two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.
- □ In Minimax the two players are called **maximizer** and **minimizer**.
  - □ The maximizer tries to get the highest score possible
  - □ The minimizer tries to do the opposite and get the lowest score possible.
- □ The general idea is to expand the state space **n steps**, and choose the optimal move at each level, depending if it is a maximizer (agent) or minimizer (opponent) level.

In this example, even if there is a potentially better solution at the Right side, the agent will decide to go Left, as he anticipates that – if it chooses R – the opponent will choose a state with a smaller reward (2)



Minimax

- □ 4 Levels Minimax Example
  - □ In the example below, the Agent will decide for a path that expectedly will conduct to a state of the game that is worse than the current state (-7 relative reward).
  - □ Also, there are other branches that will correspond to "win the game", i.e., fully satisfy the problem objectives (∞ reward).
  - □ However, assuming that the opponent will always do the *smart choice*, the best thing to do is evolve to such *worse state*.

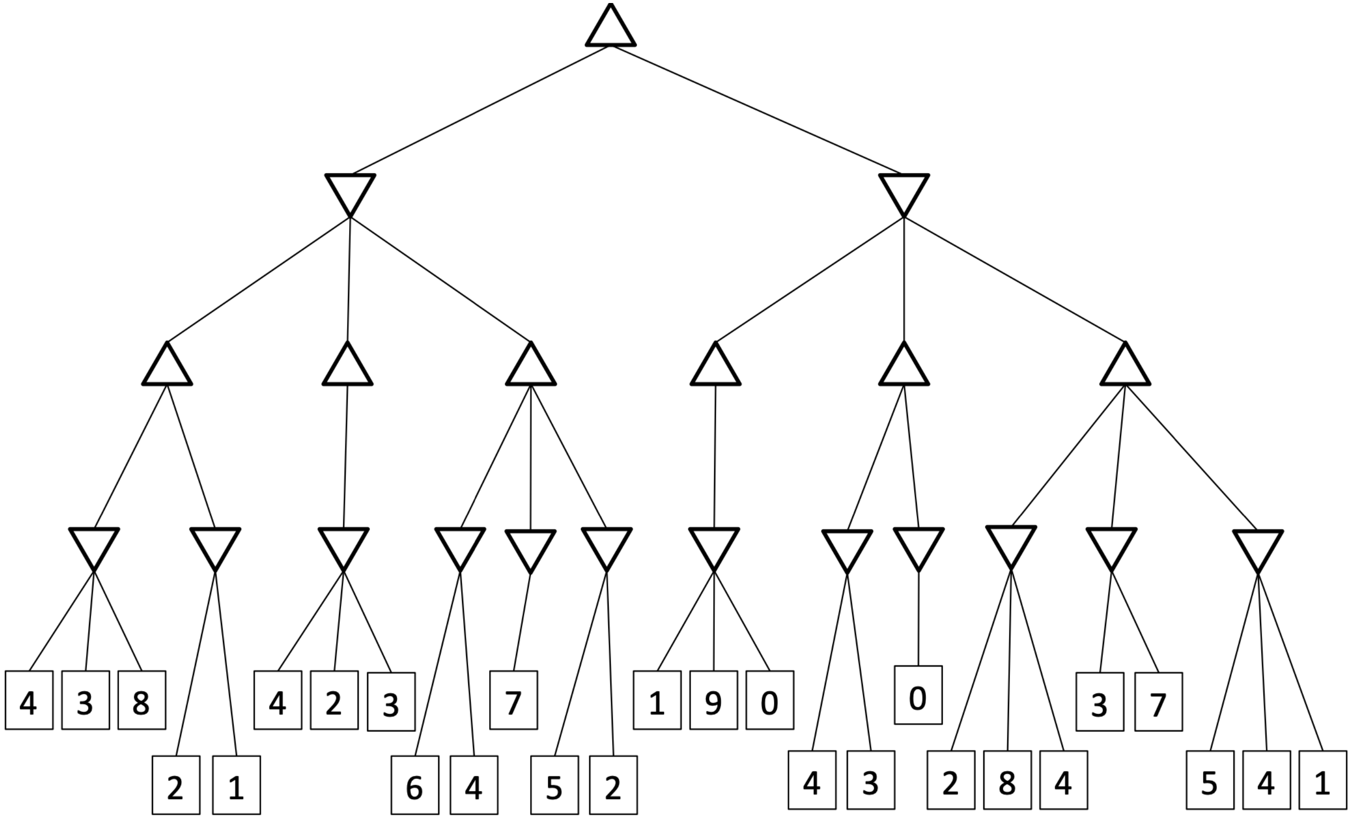


Minimax

```
# initial call: minimax(root, DEPTH, True)
function  minimax( node, depth, maximizing_player):
    if depth == 0 OR node is a terminal node
        return f(node) #heuristic of “node”
    if maximizing_player then
        value = −∞
        for each child of node do
            value = max(value, minimax(child, depth − 1, FALSE))
        return value
    else
        value = +∞
        for each child of node do
            value = min( value, minimax( child, depth − 1, TRUE ) )
        return value
```

Minimax - Exercise

- □ Consider the following tree, using the △/▽ notation for nodes (states), that represents whether the node is of maximizer (△) or minimizer (▽) kind.
  - □ Provide the values obtained at each node, using the minimax algorithm.



Minimax – Alfa-Beta Pruning

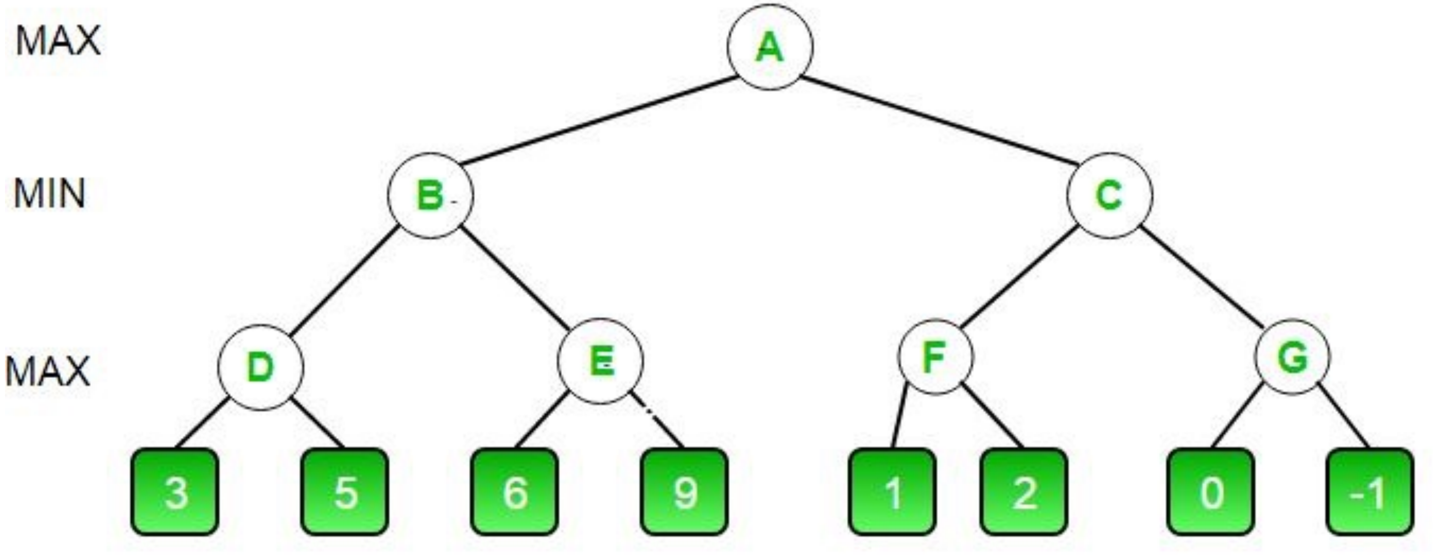
- □ **Alfa-beta Pruning** is an optimization technique for the minimax algorithm.
- □ It reduces the computation time significantly, allowing to search much faster and go into deeper levels in the game tree.
- □ The general idea is to cut off branches in the game tree which need not be searched because there already exists a better move available.
- □ It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely Alpha and Beta.
- □ **Alfa** is the best value that a maximizer node can guarantee at that level or above.
- □ **Beta** is the best value that a minimizer node can guarantee at that level or above.
- □ They are initialized such that $\alpha = -\infty; \beta = \infty$
- □ At every node where $\beta \leq \alpha$, the search stops there, and no deeper levels should be visited.
- □ This is actually the pruning operation

Minimax – Alfa-Beta Pruning

```
# initial call: minimax(root, DEPTH, True, - ∞, ∞)
function  minimax( node, depth, maximizing_player, alpha, beta):
  if depth == 0 OR node is a terminal node
    return f(node) #heuristic of "node"
  if maximizing_player then
    value = −∞
    for each child of node do
         value = max(value, minimax(child, depth − 1, FALSE, alpha, beta))
         alpha = max(alpha, value)
         if beta ≤ alpha
                  break
    return value
  else
    value = +∞
    for each child of node do
         value = min( value, minimax( child, depth − 1, TRUE, alpha, beta) )
         beta = min(beta, value)
         if beta ≤ alpha
                  break
    return value
```

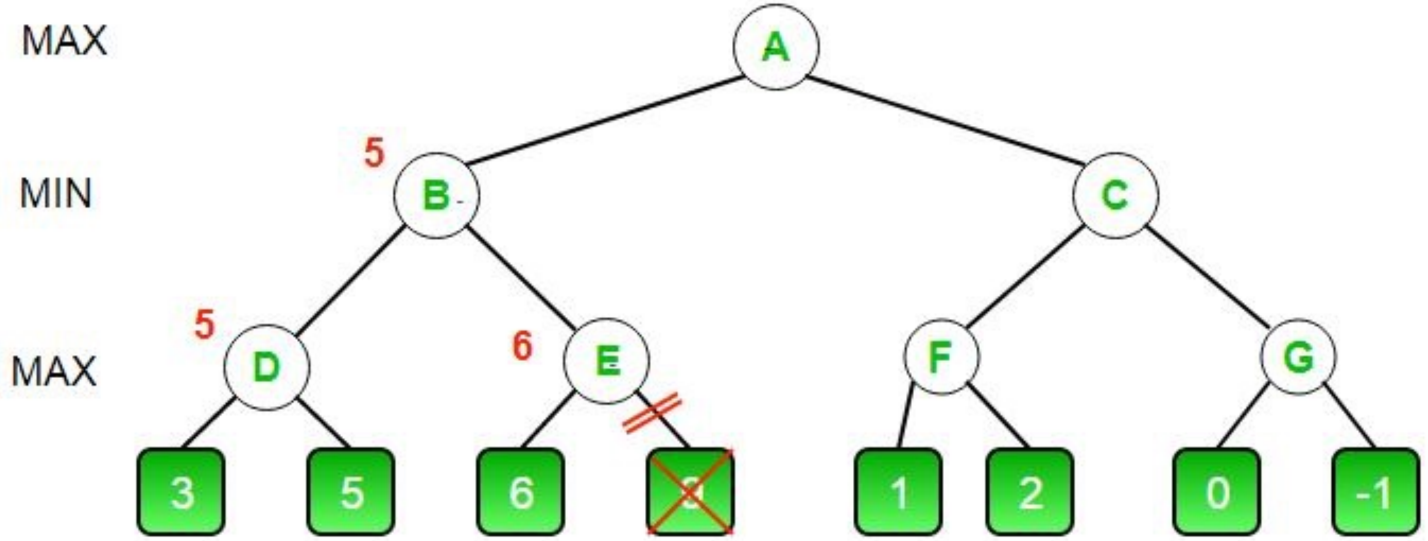
Minimax – Alfa-Beta Pruning

- **Example:** Consider the following tree. Run the Alfa-beta Minimax Pruning Algorithm to find the pruned branches.



Minimax – Alfa-Beta Pruning

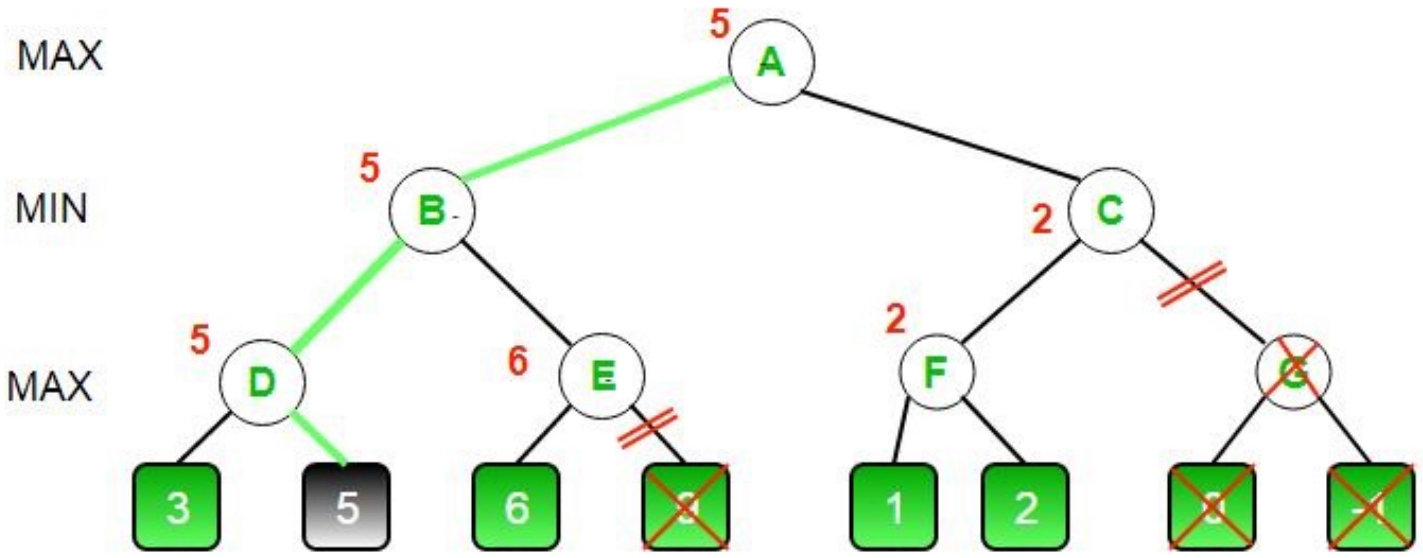
- **Example:** Consider the following tree. Run the Alfa-beta Minimax Pruning Algorithm to find the pruned branches.



B is a Minimizer and one of his successors already returned “5”. Hence, the value returned by B to its predecessor (A) will always be ≤ 5. However, as E is a maximizer and one of his successors returned “6”, it will always return a value ≥ 6. Hence any value coming from the right branch of E (independently of the depth of that branch) will be useless

Minimax – Alfa-Beta Pruning

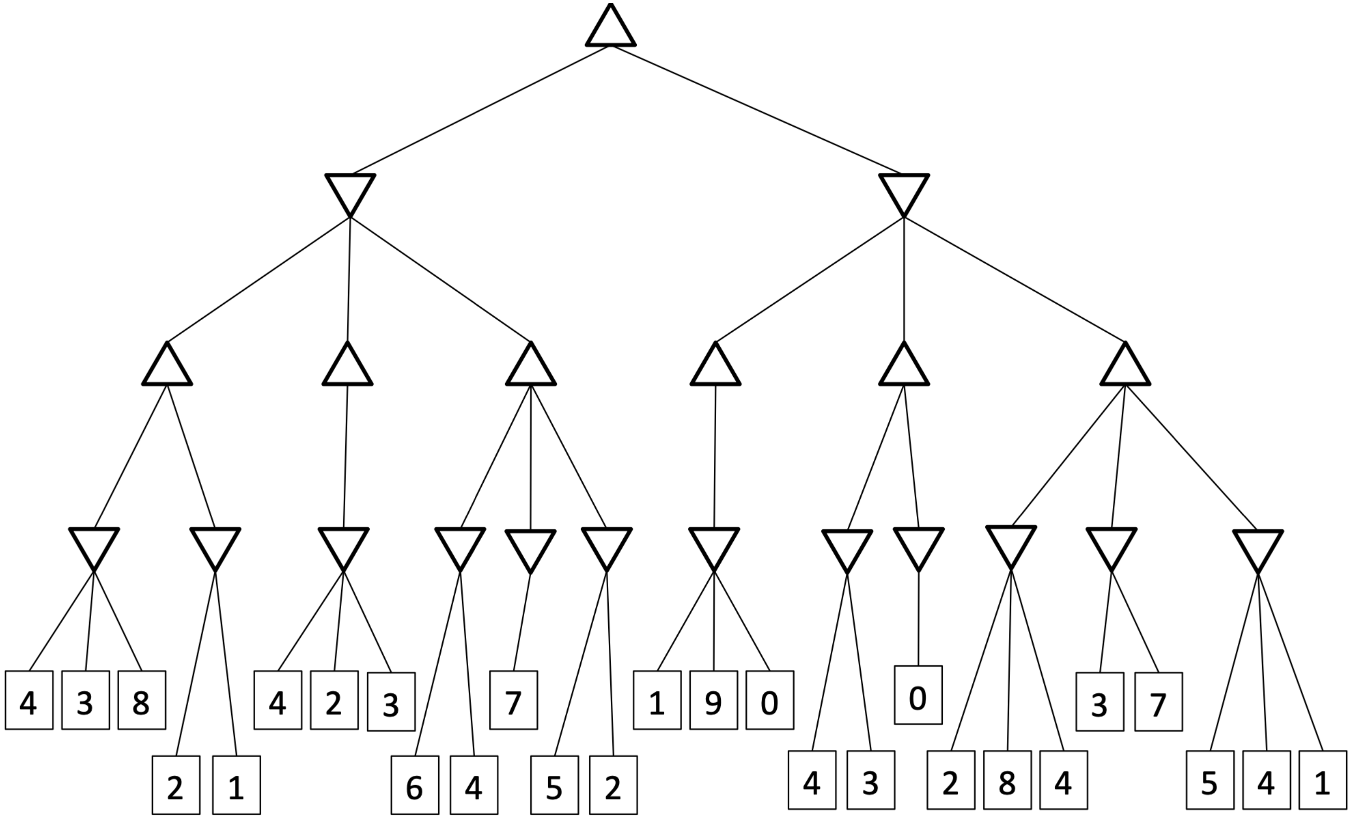
□ **Example:** Consider the following tree. Run the Alfa-beta Minimax Pruning Algorithm to find the pruned branches.



A is a maximizer and has already "5" returned from B. Hence the value returned by A ≥ 5. As C is a minimizer it will return something ≤ 2. Hence. Any value from the right branch of C will be useless (and should be pruned)

Minimax – Exercise: Alfa-Beta Pruning

- □ Exercise: Indicate which branches of the tree can be pruned, according to Alfa-Beta pruning strategy.



Monte Carlo Tree Search

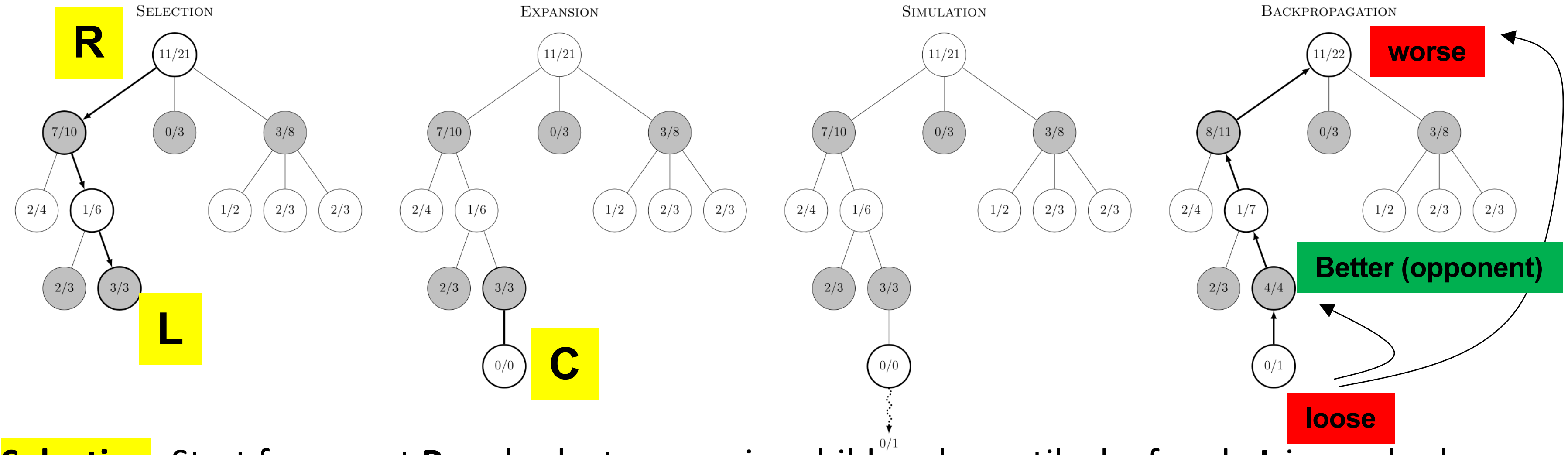
- □ Games like tic-tac-toe, checkers or chess can arguably be solved using the minimax algorithm.
- □ However, things can get a little tricky when there are many potential actions to be taken at each state.
  - □ This is because minimax explores all the nodes available. It becomes difficult to solve a complex problem.
- □ **Monte Carlo Tree Search** (**MCTS**) is a heuristic search algorithm for some kinds of decision processes.
  - □ The terminology comes from the original Monte Carlo method, that uses random sampling for deterministic problems which are difficult or impossible to solve using other approaches, and dates to the 1940s.
  - □ In 1987, Bruce Abramson combined minimax search with an expected-outcome model based on random game playouts to the end, instead of the usual static evaluation function.

Monte Carlo Tree Search

- □ The rationale of MCTS is to analyse/expand first the most promising moves, expanding the search tree based on random sampling of the search space.
- □ The application of Monte Carlo tree search is based on playouts, a.k.a. roll-outs.
  - □ In each playout, the game is played out to the very end by selecting moves at random.
  - □ The final game result of each playout is then used to weight the nodes in the game tree so that better nodes are more likely to be chosen in future playouts.
- □ The process has four main phases:
  - □ Selection
    - □ Selecting good child nodes, starting from the root node R, that represent states leading to better overall outcome (win).
  - □ Expansion
    - □ If L is a not a terminal node, create one or more child nodes and select one of these (C).
  - □ Simulation (rollout)
    - □ Run a simulated playout from C until a result is achieved.
  - □ Backpropagation
    - □ Update the current move sequence with the simulation result.

Monte Carlo Tree Search

Source: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search



- □ **Selection**: Start from root **R** and select successive child nodes until a leaf node **L** is reached. The leaf is any node that has a potential child from which no simulation (playout) has yet been done.
- □ **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.
- □ **Backpropagation**: Use the result of the playout to update information in the nodes on the path from **C** to **R**.
  - □ In case of a two-player game, the states corresponding to moves from the same/opponent players should be updated accordingly.