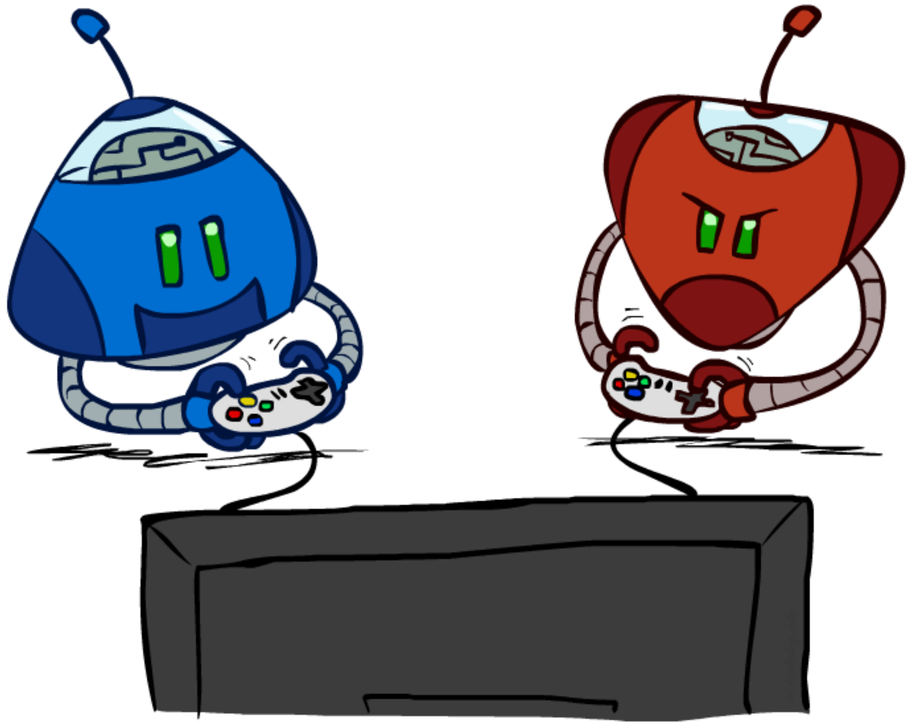
Adversarial Search

MINIMAX Search



MAEs and Games

- **Multi-agent environment**: every agent needs to consider the actions of the other agents, in order to optimize its own welfare
  - **Cooperative**:
    - Agents act collectively to achieve a common goal
  - **Competitive**:
    - Agents compete against each other
    - Their goals are in conflict
    - Gives rise to the concept of **adversarial** search problems – often known as **games**.

Games vs. Search problems

- In typical search problems, we optimize a measure to acquire the goal: there is no opponent

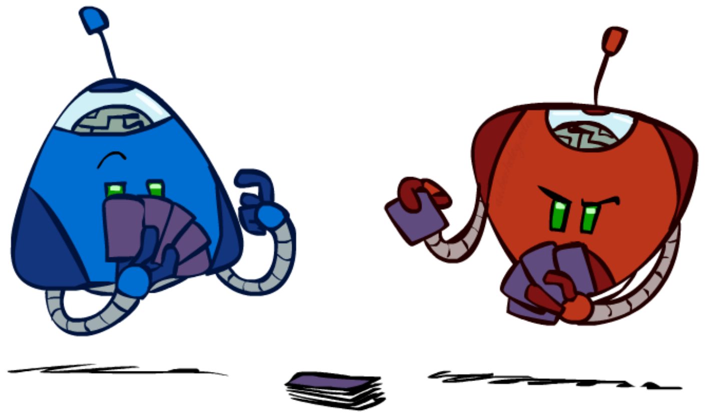
- In games, there is an "Unpredictable" opponent
  - Need to specify a move for every possible opponent reply
  - Strict penalty on an inefficient move
  - Stringent time constraints
  - Unlikely to find goal, must approximate
  - Requires some type of a decision to move the search forward.

Types of Games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

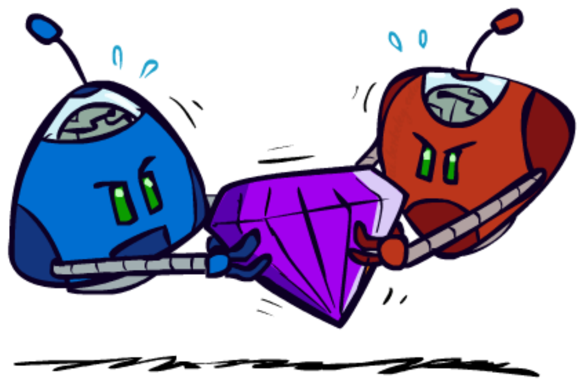
- Many different kinds of games!

- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?



- Want algorithms for calculating a **<u>strategy (policy)</u>** which recommends a move from each state

Zero-Sum Games



- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition

- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
  - More later on non-zero-sum games

Game Formulation

- **States**: S (start at Initial State s_0)
- **Players**: P={1...N} (usually take turns)
- **Actions**: A (may depend on player / state)
- **Transition Function**: SxA $\rightarrow$ S
- **Terminal State Test**: S $\rightarrow$ {t,f}
- **Terminal Utilities**: SxP $\rightarrow$ R
  - defining the usefulness of the terminal states from the point of view of one of the players.

Basic Strategy

- Grow a search tree – and examines enough nodes to allow a player to determine what move to make.
- Only one player can make move at each turn.
- Assume we can assign a payoff (utility value) at final position.
- Assume opponent always makes moves worst for us.
- Pick best move on own turn.

Utility function

- For Tic-Tac-Toe, the function could be as simple as returning:

• +1/ +C (+10);	it means that MAX wins
• -1/ -C (-10);	it means that MIN wins
• 0	otherwise.

- However, **this simple evaluation function may require deeper search**

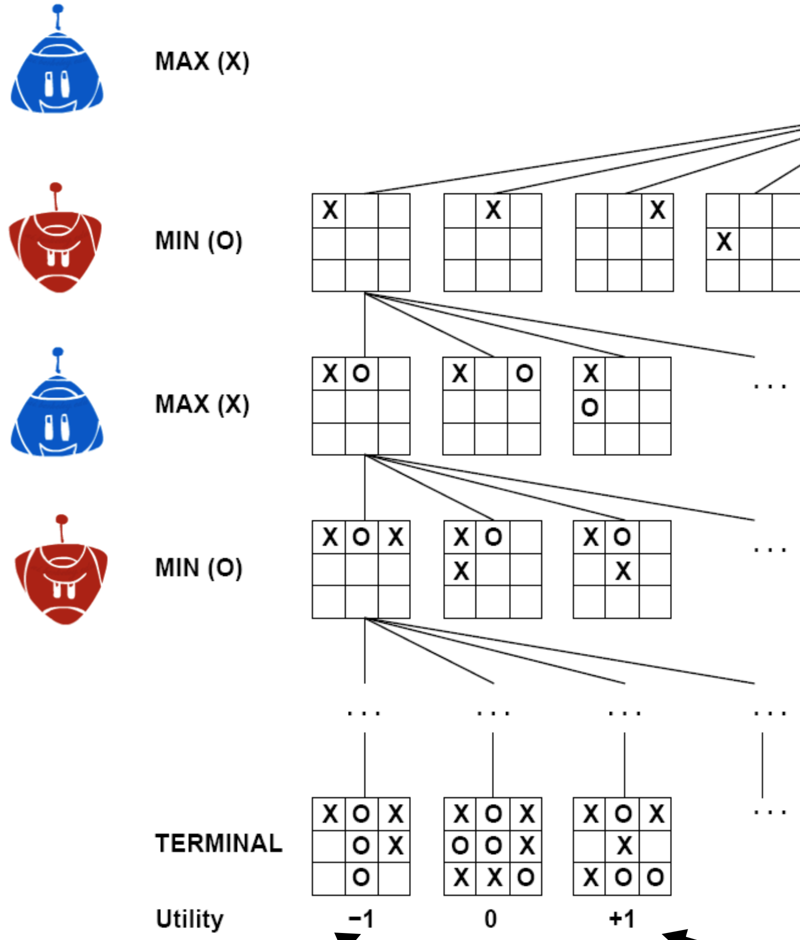
- Complete tree needs to be generated to calculate moves
  - May be slow/infeasible in some cases!

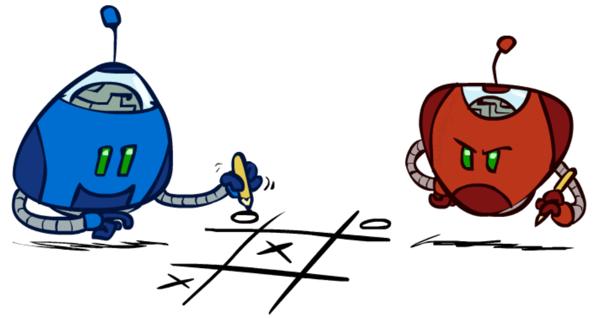
Optimal Strategy For MAX

- This is a Multi Agent Environment
  - In discovering the sequence of optimal moves by MAX, we have to consider that MIN is taking moves as well
  - So, the optimal strategy will tell how should MAX play against MIN (by considering the moves of the latter), in order to win the game

- Let us consider a shortened game tree of TIC-TAC-TOE
  - Numbers are the utilities
  - Ply: a move by MAX followed by a move from MIN.

Tic-Tac-Toe Game Tree





This terminal state is one of the worst for MAX and one of the best for MIN

This terminal state is one of the best for MAX and one of the worst for MIN

Minimax

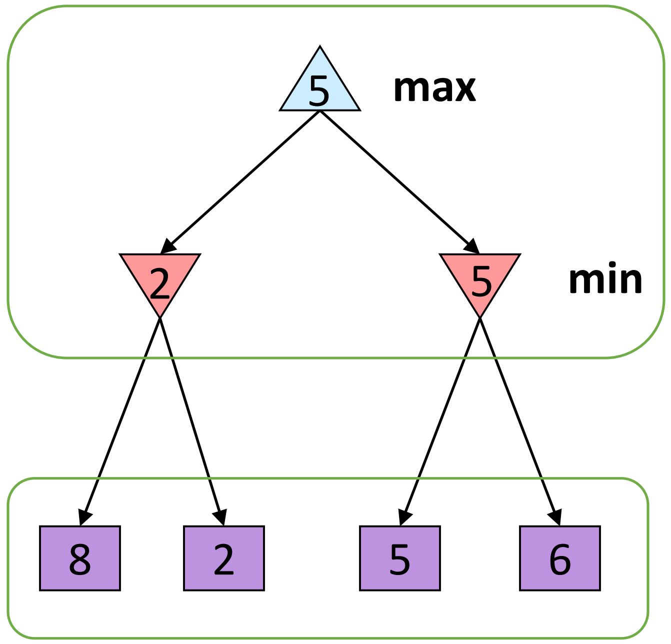
- When it is the turn of MAX, it will always take an action in order to **maximize** its utility, because it’s winning configurations have high utilities
- When it is the turn of MIN, it will always take an action in order to **minimize** its utility, because it’s winning configurations have low utilities
- In order to implement this, we need to define a measure in each state that takes the move of the opponent into account.
- The term “**MINIMAX**” is used because:
  - the opponent is always trying to **minimize** the utility of the player, and
  - the player is always trying to **maximize** this minimized selection of the opponent.

Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

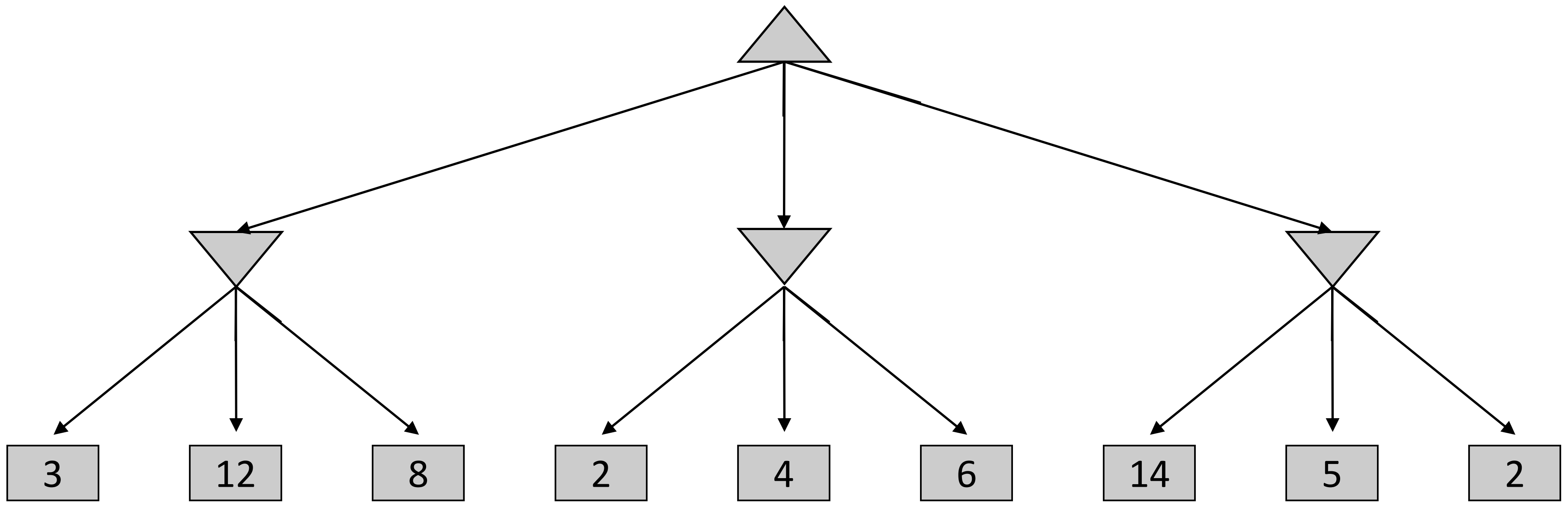
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

Minimax values: computed recursively



Terminal values: part of the game

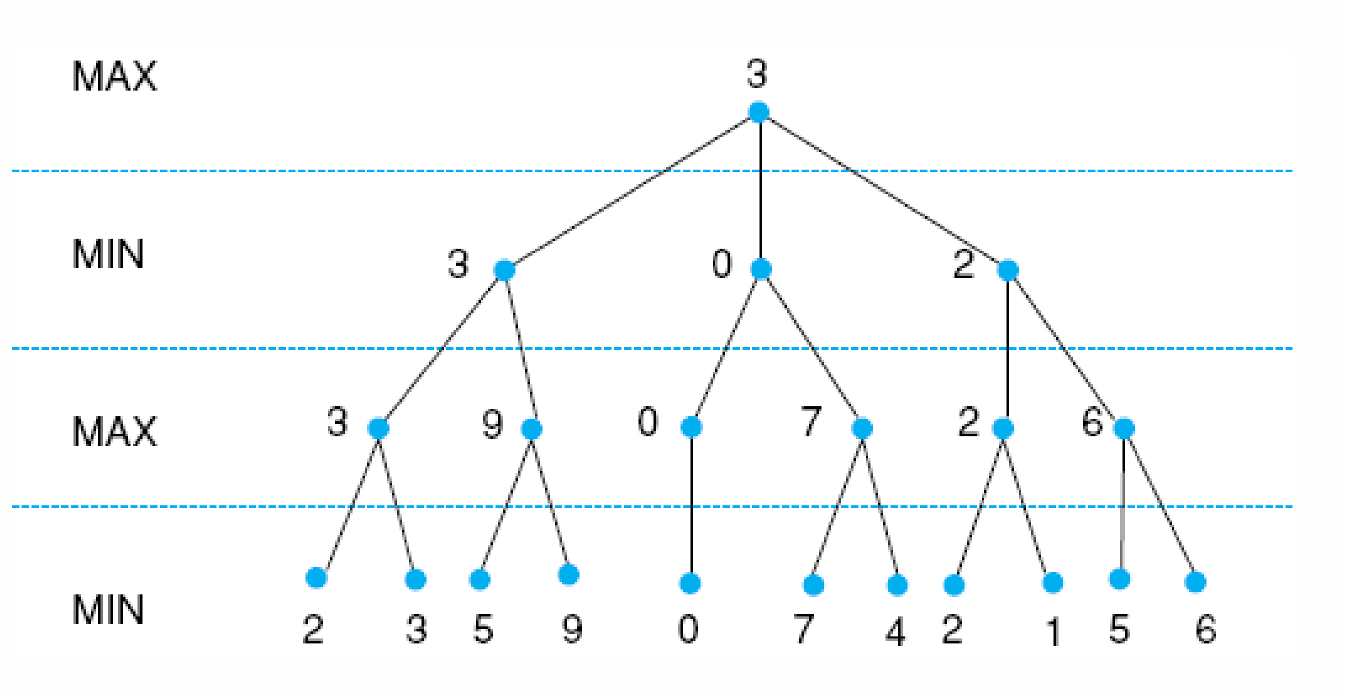
Minimax Example



- At each node, **MAX** will always select the action with **highest minimax value** (it wants to reach states with higher utilities)

- At each node, **MIN** will always select the action with **lowest minimax** value (it wants to reach states with lower utilities).

Minimax to a hypothetical state space. Leaf states show heuristic values; internal states show backed-up values.



$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Minimax Implementation (Dispatch)

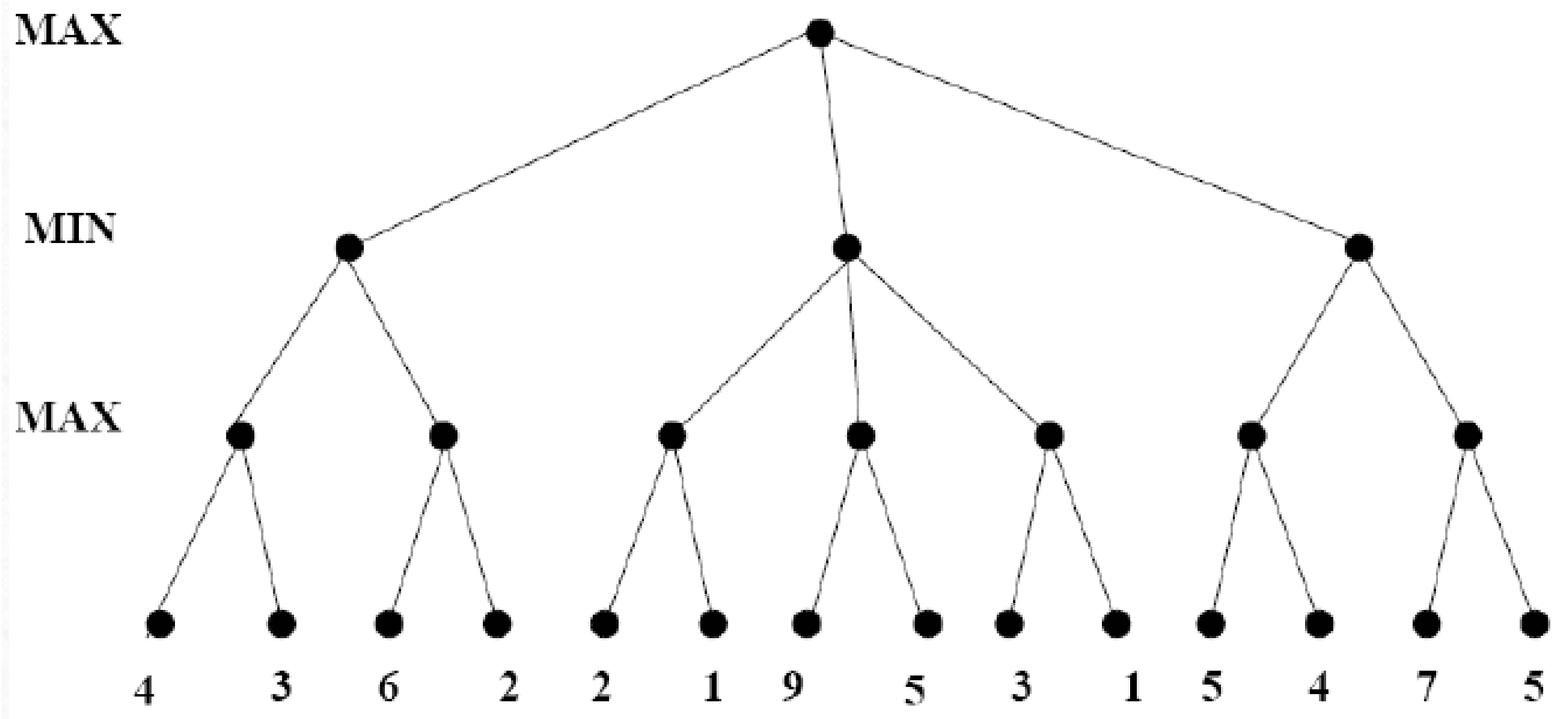
```
def value(state):
    if the state is a terminal state: return the state’s utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

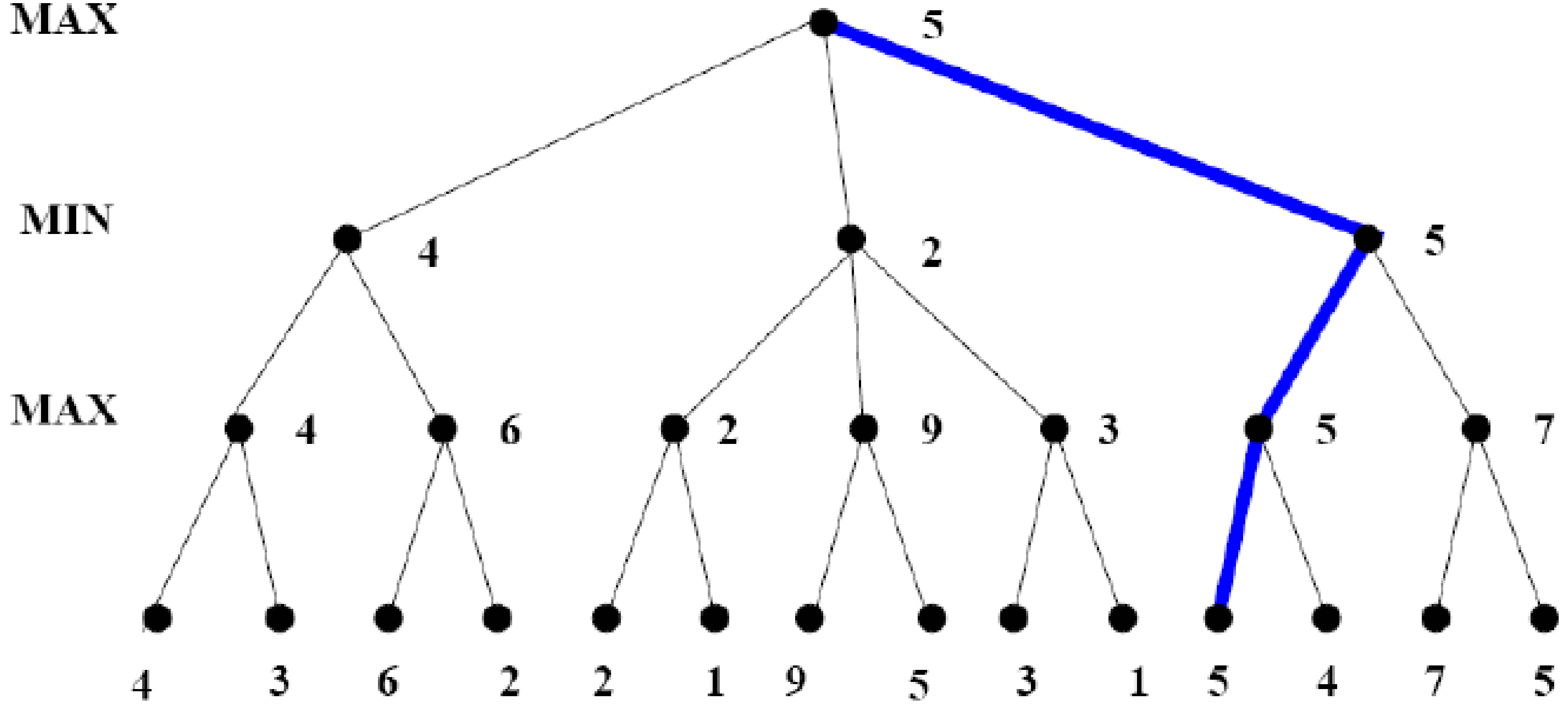
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

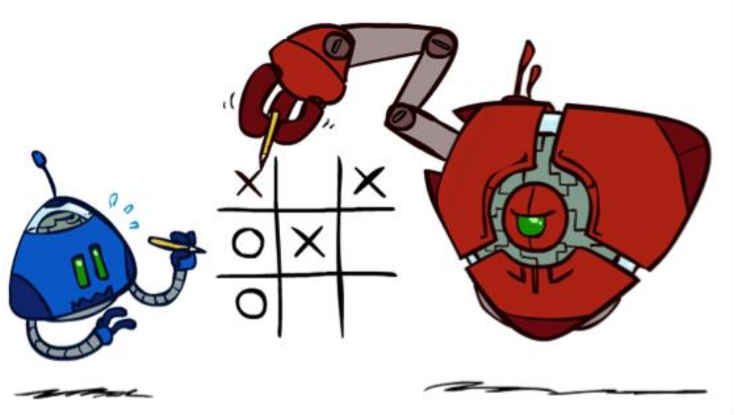
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

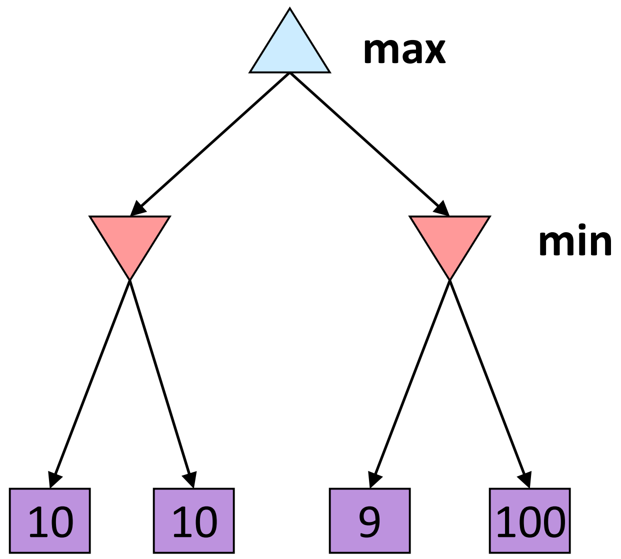
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

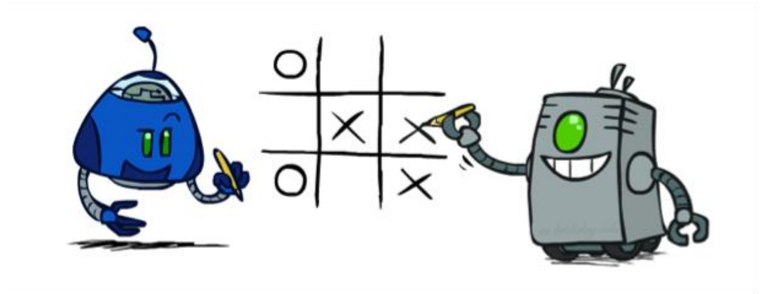

MAX
MIN
MAX
4
3
6
2
2
1
9
5
3
1
5
4
7
5


MAX
MIN
MAX
5
4
2
5
4
6
2
9
3
5
7
4
3
6
2
2
1
9
5
3
1
5
4
7
5

Minimax Properties





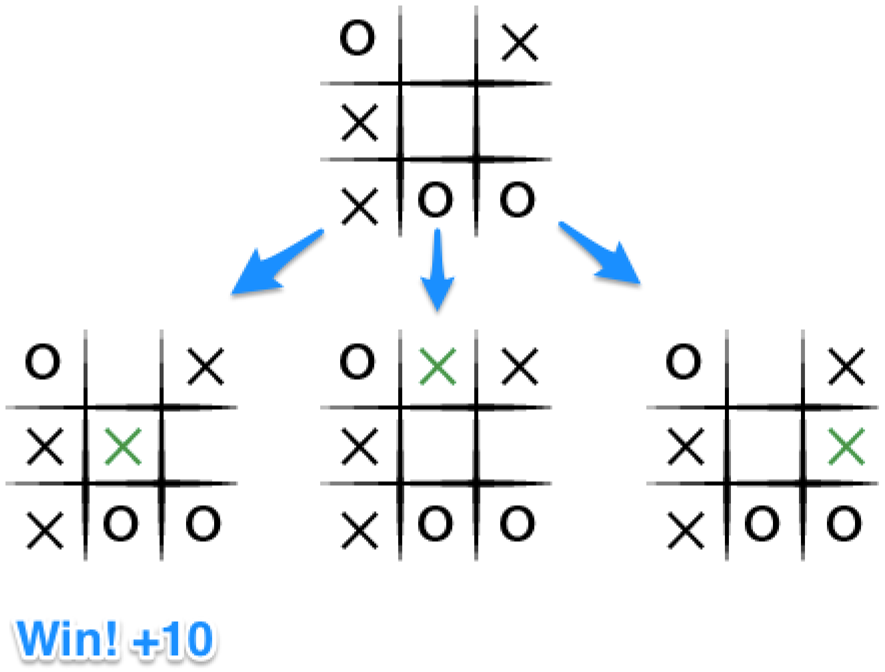


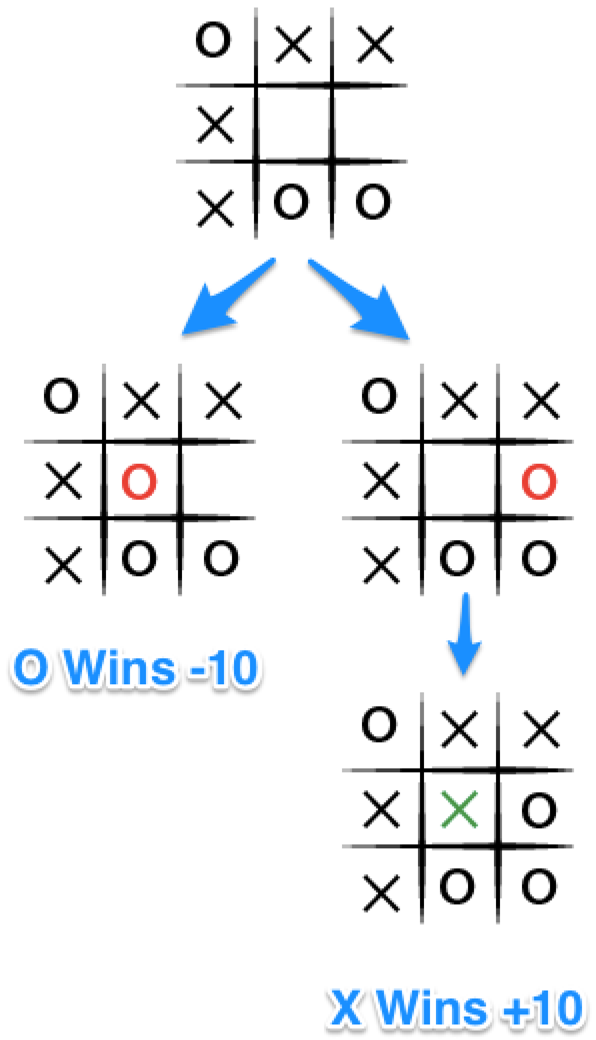
Optimal against a perfect player. Otherwise?

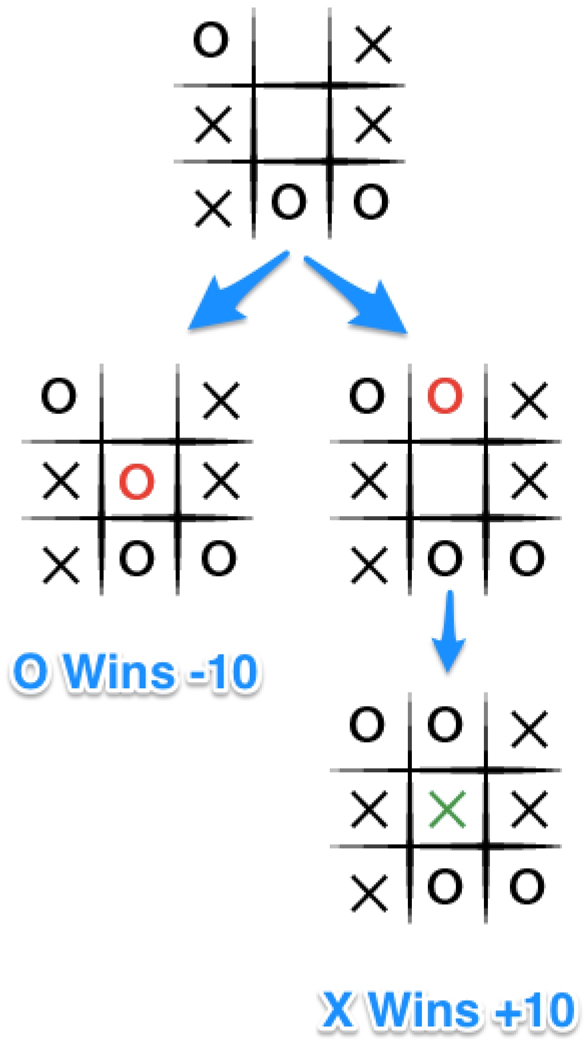
Perhaps a better utility function?

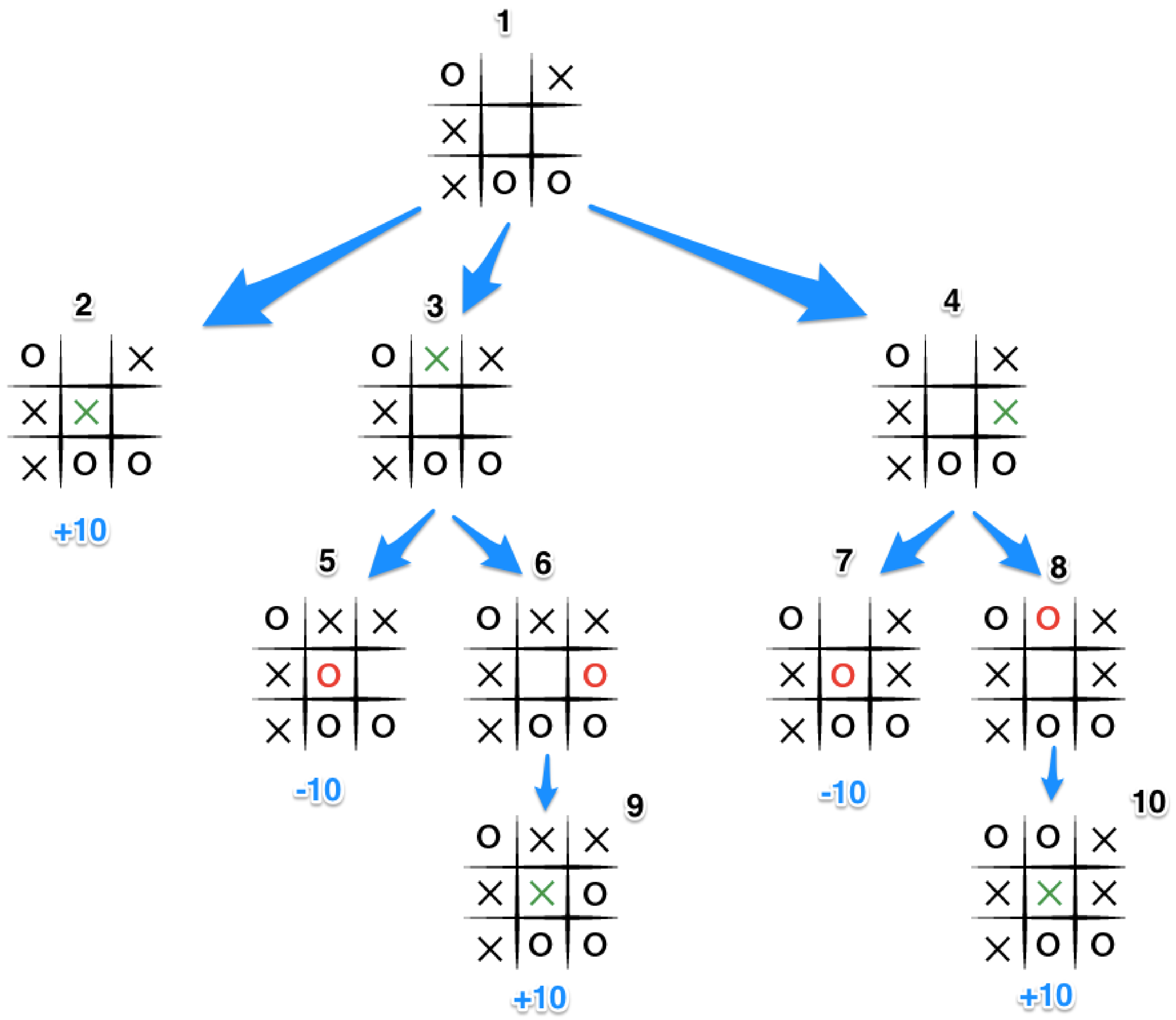
- +100 for EACH 3-in-a-line for computer (Max).
- +10 for EACH 2-in-a-line (with a empty cell) for computer.
- +1 for EACH 1-in-a-line (with two empty cells) for computer.
- Same negative scores for opponent (Min),
    - -100 for EACH 3-in-a-line
    - -10 for EACH 2-in-a-line
    - -1 for EACH 1-in-a-line.
- 0 otherwise (empty lines or lines with both computer's and opponent's seed).
- Compute the scores for each of the 8 lines (3 rows, 3 columns and 2 diagonals) and obtain the sum.

Brief Example




O Wins -10
X Wins +10

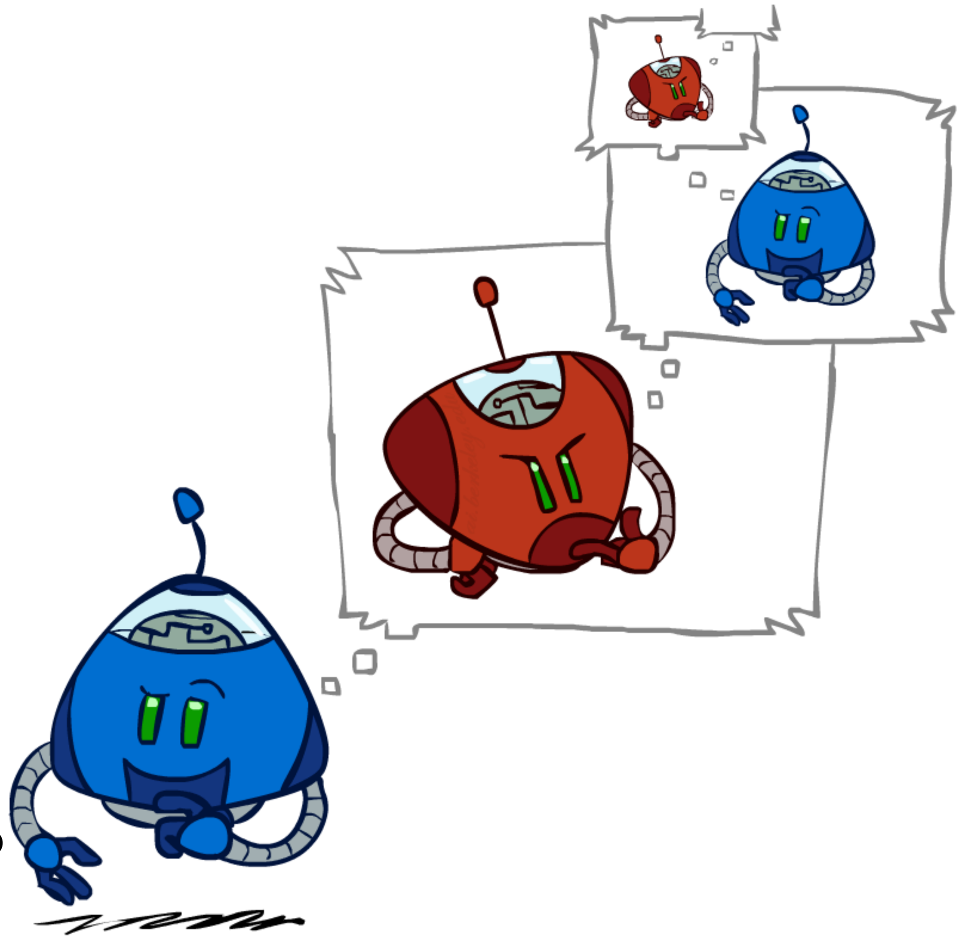

O Wins -10
X Wins +10


1
2
3
4
+10
5
6
7
8
-10
-10
9
10
+10
+10

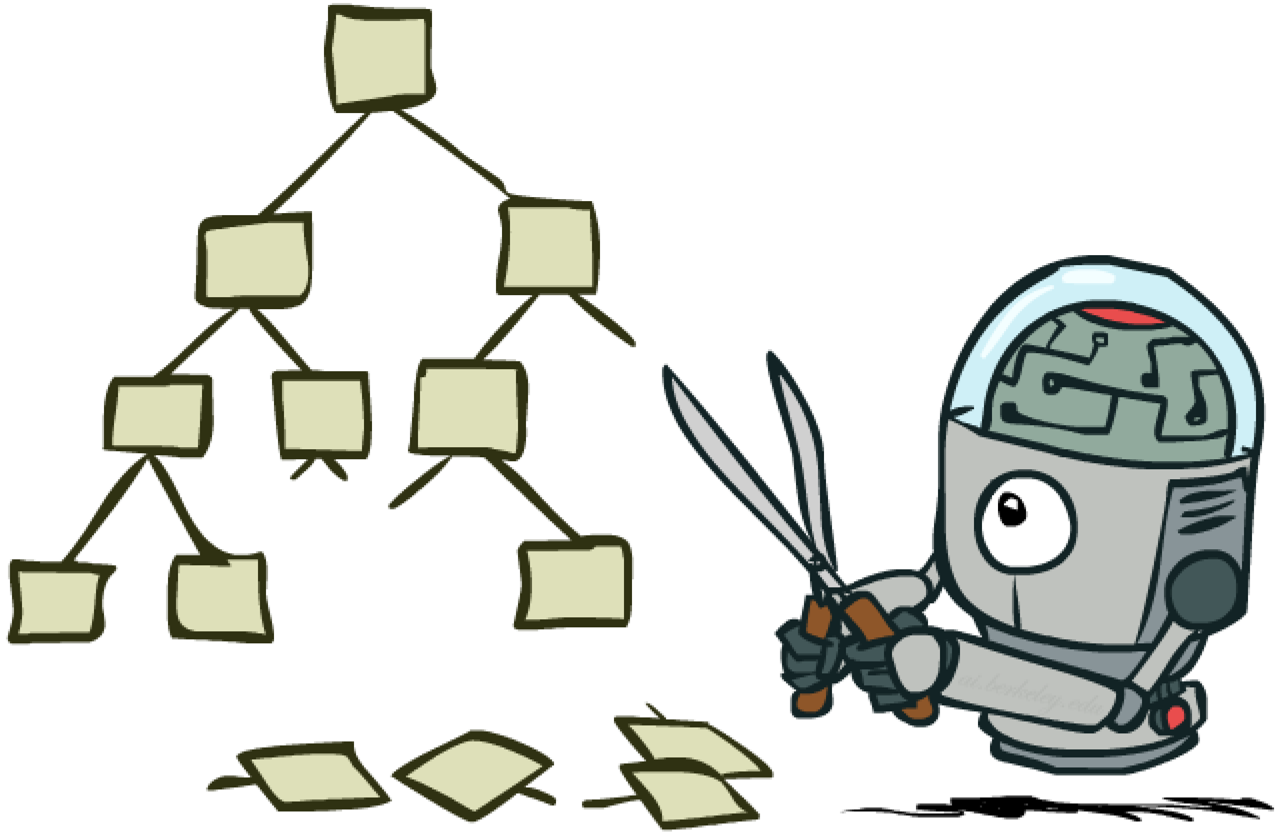
Minimax Efficiency

- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$
  - Complete: **Yes** (if tree is finite)
  - Optimal? **Yes** (against an optimal opponent)

- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?



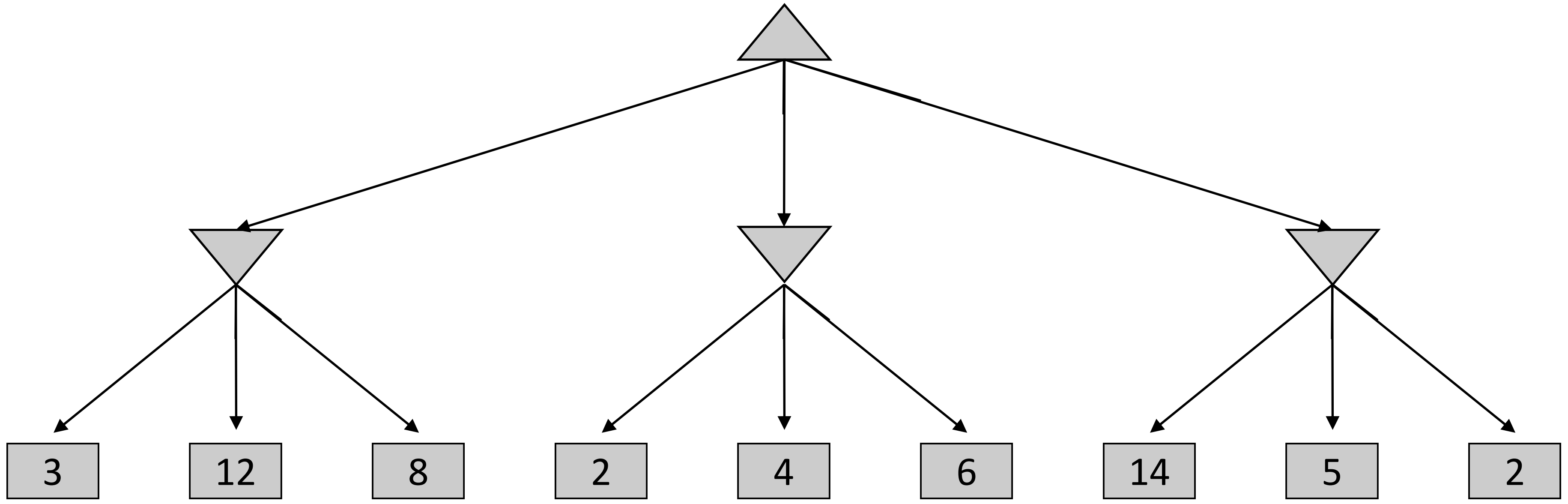
Game Tree Pruning



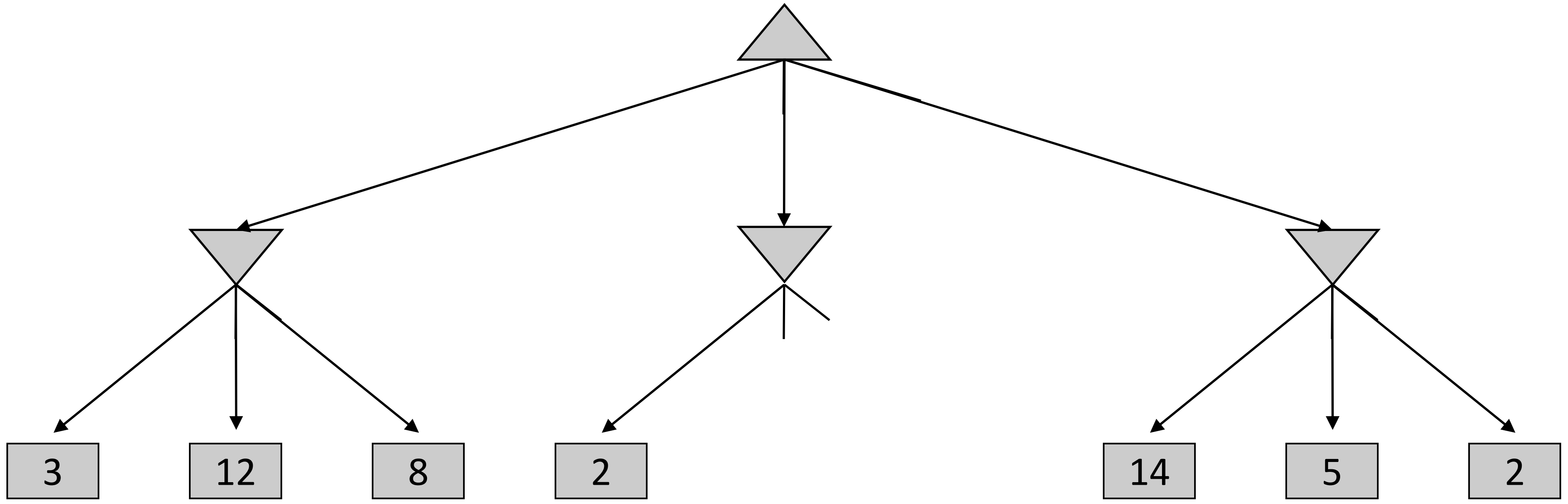
Minimax Search

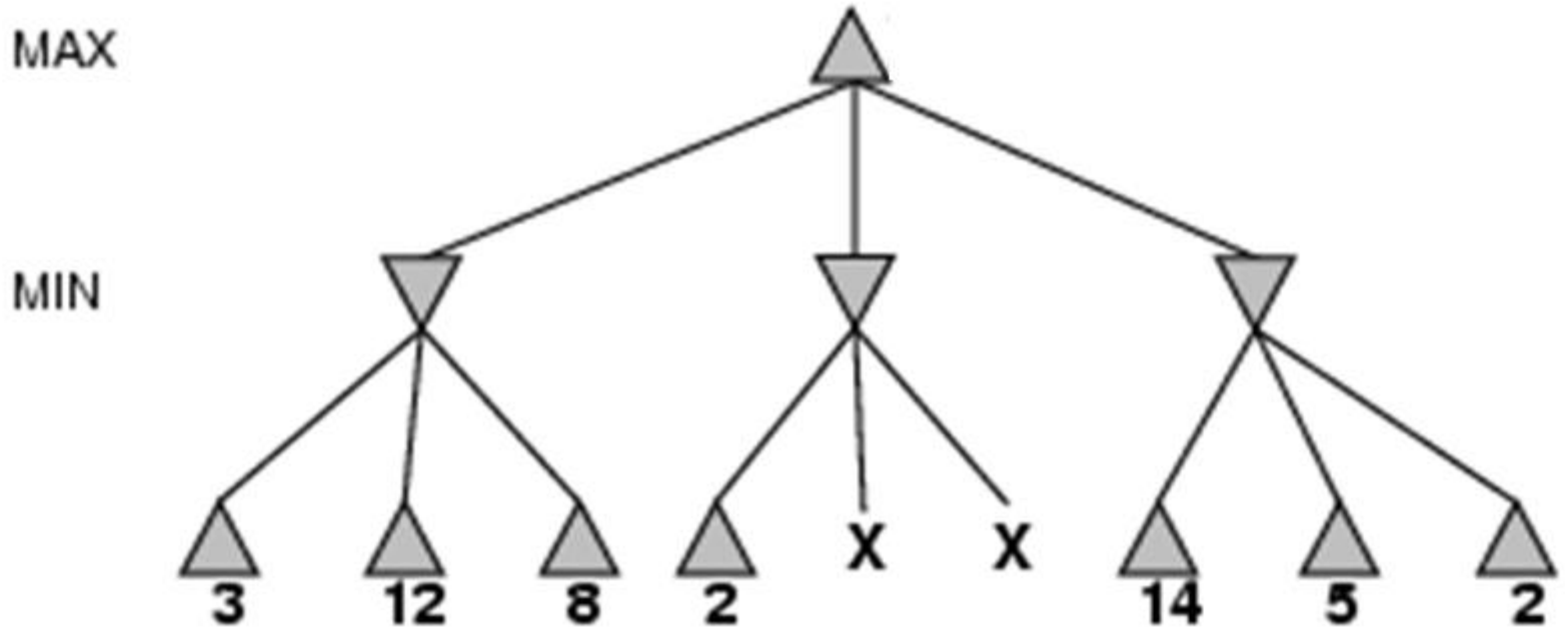
- Straight minimax requires a two-pass analysis of the search space,
  - the first to descend to the level depth and
  - the second to propagate values back up the tree.
- Minimax pursues all branches in the space, including many that could be ignored or pruned by a more intelligent algorithm.
- Recursion: Winds all the way to the terminal nodes, and then unwinds back by backing up the values

Minimax Example



Minimax Pruning





$$\begin{aligned}
\textsc{Minimax}(\mathit{root}) &= \max(\min(3,12,8),\min(2,x,y),\min(14,5,2)) \\
&= \max(3,\min(2,x,y),2) \\
&= \max(3,z,2) \qquad \text{where } z=\min(2,x,y)\leq 2 \\
&= 3.
\end{aligned}$$

Minimax Quiz

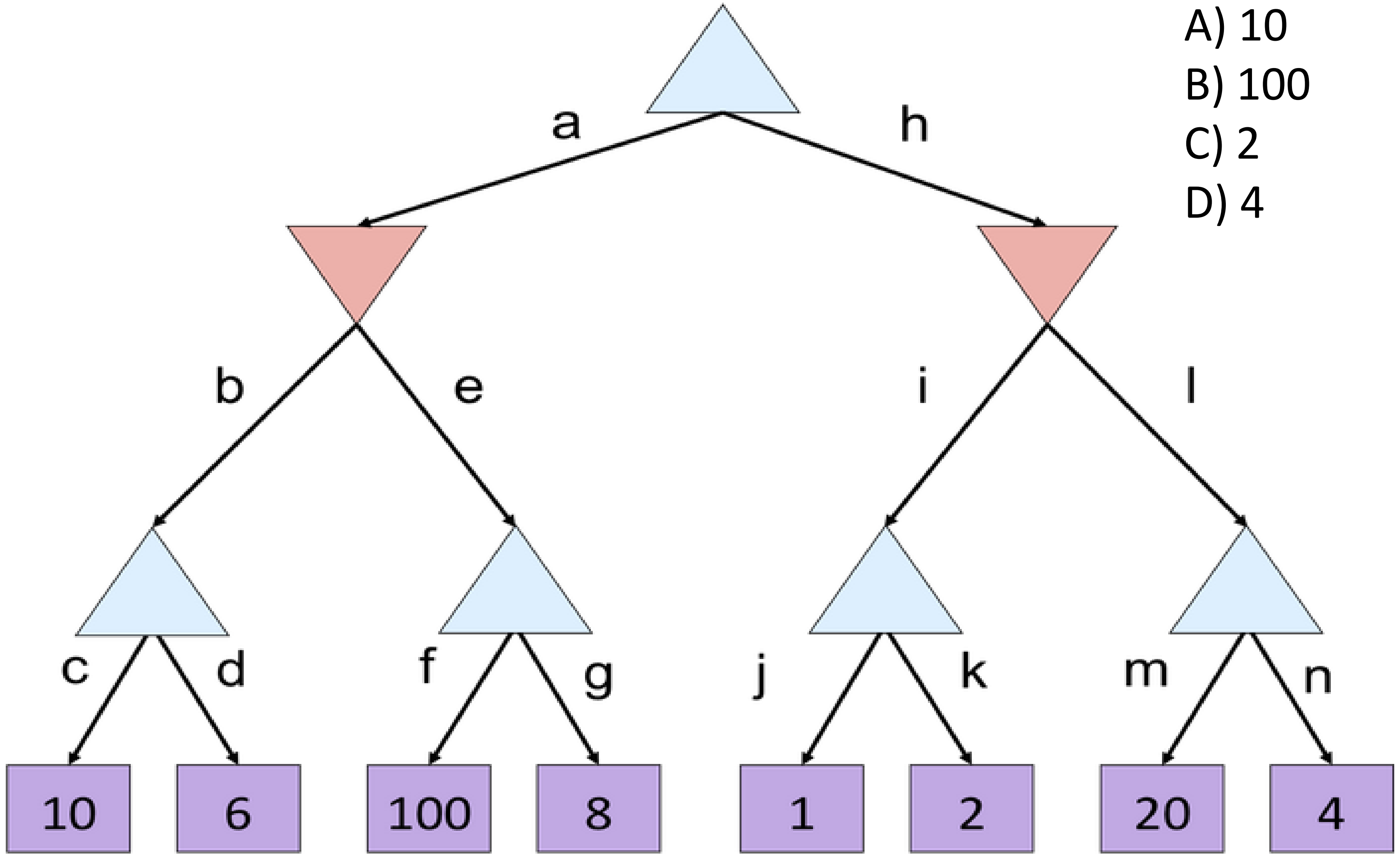
What is the value of the top node?

A) 10

B) 100

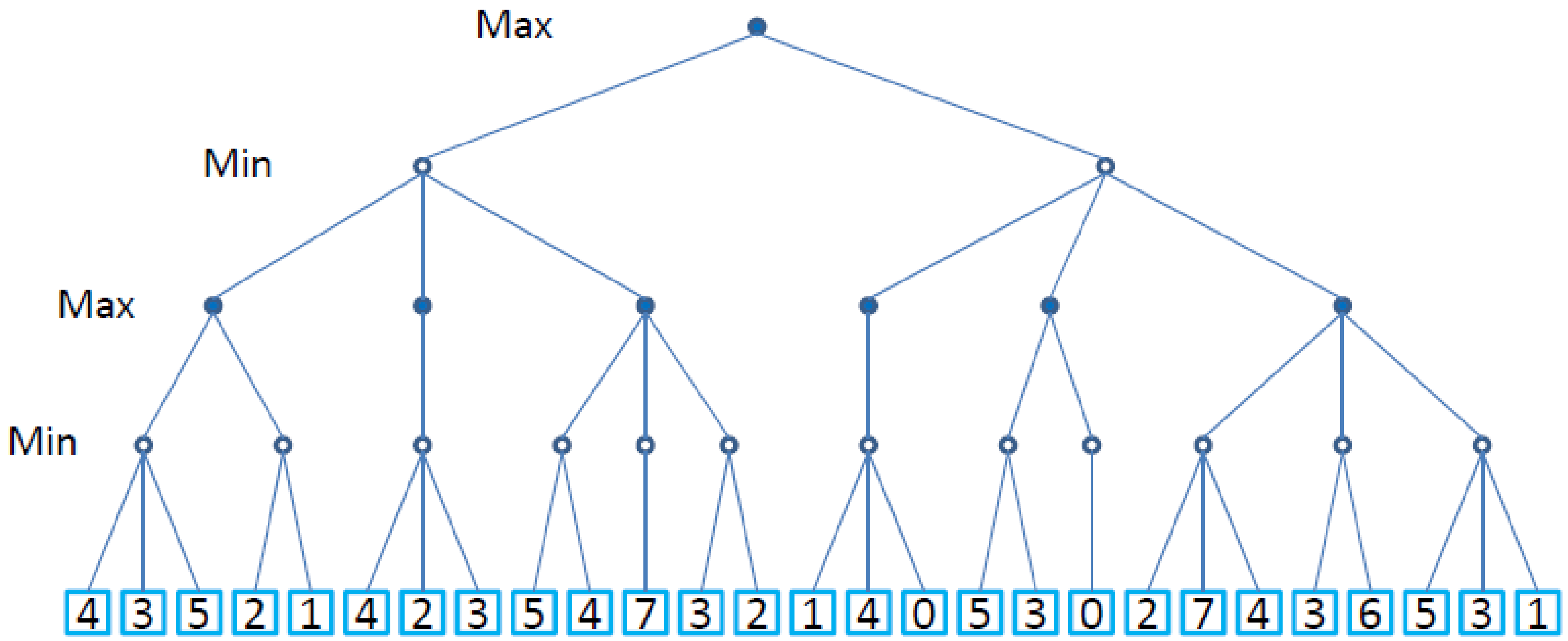
C) 2

D) 4

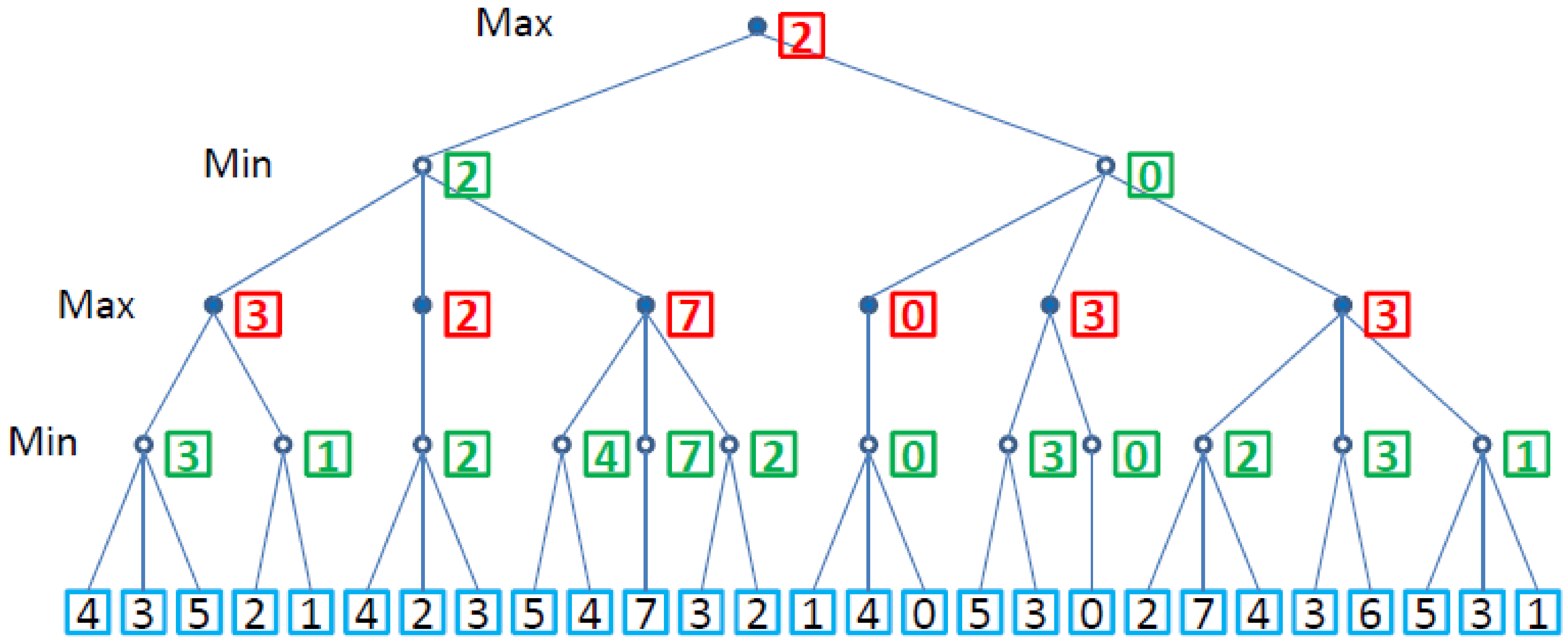


Minimax: Quiz

- Perform the minimax algorithm on the figure below.



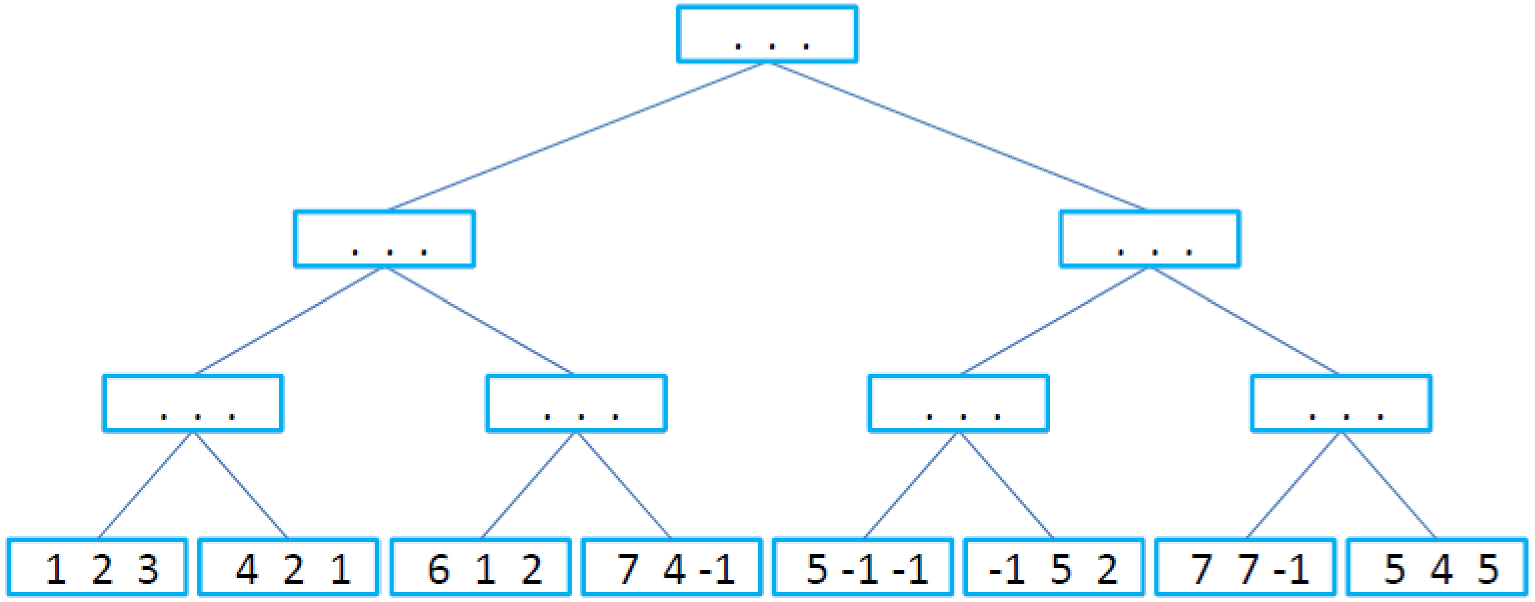
Minimax Search



Minimax for 3 player

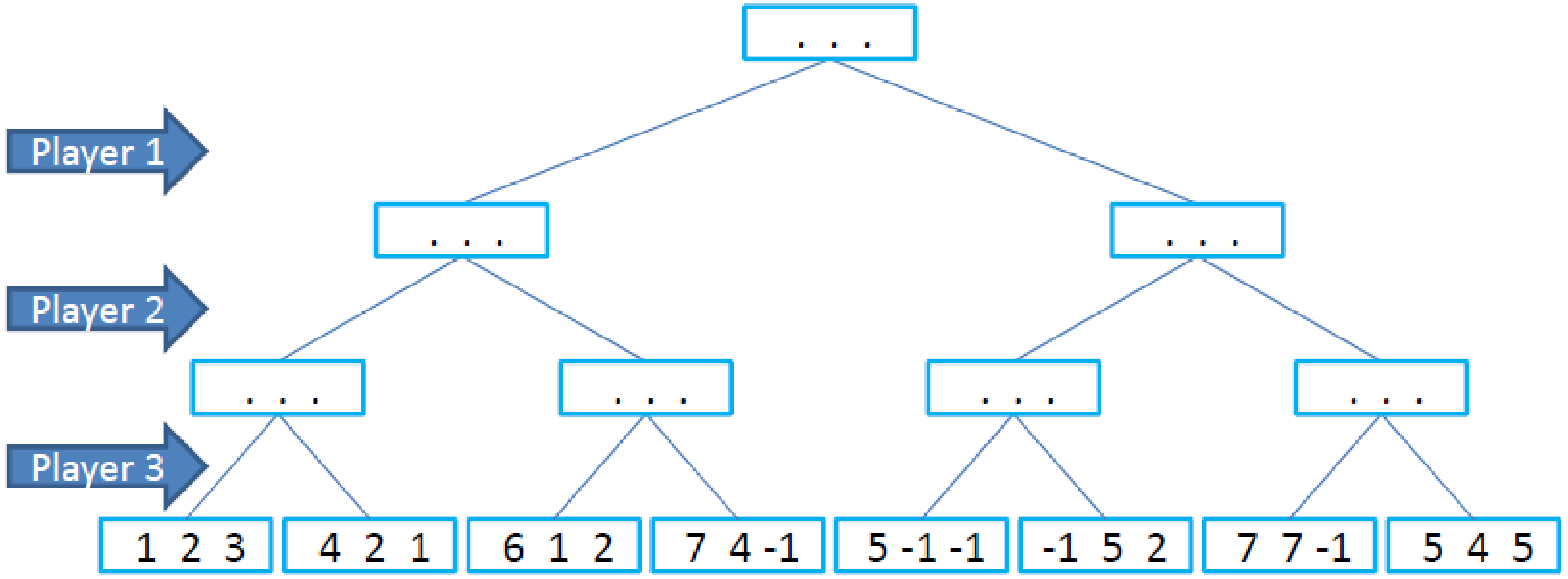
Minimax for 3-Player Game

- All players are MAX.
- Evaluation function given by vectors.



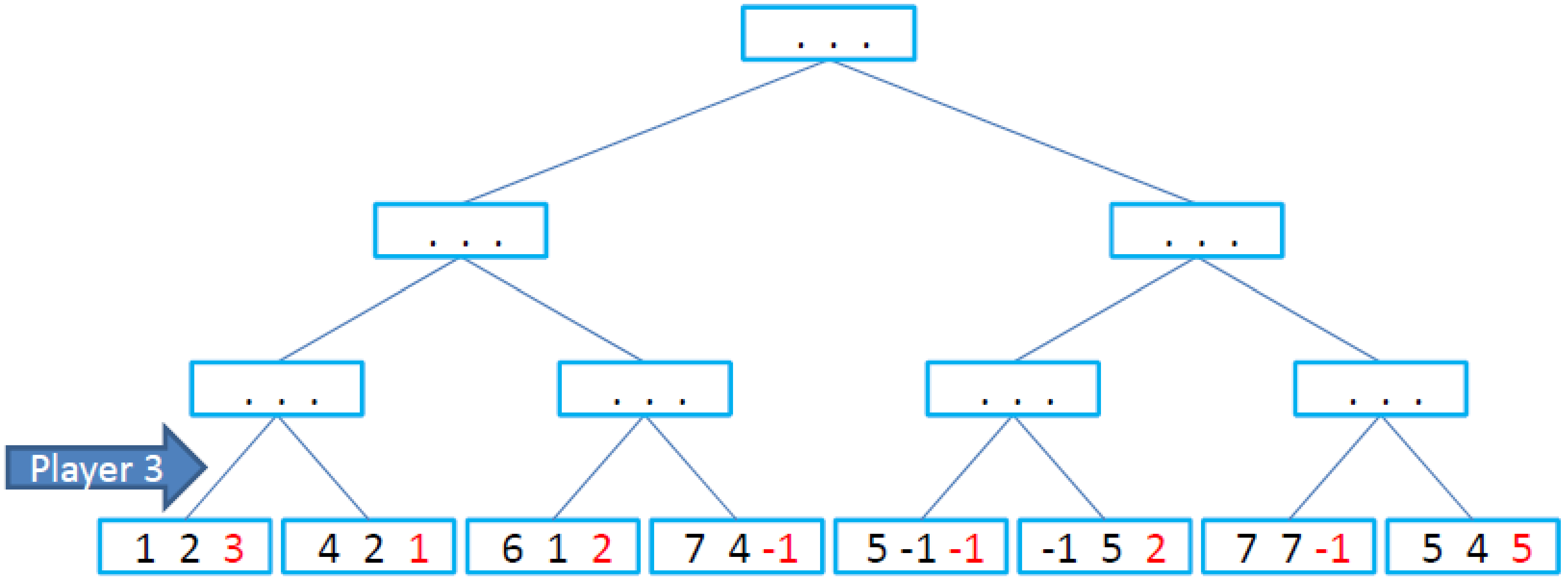
MiniMax For 3 Players

- Each layer assigned to 1 player
- Turn: every 3 layers



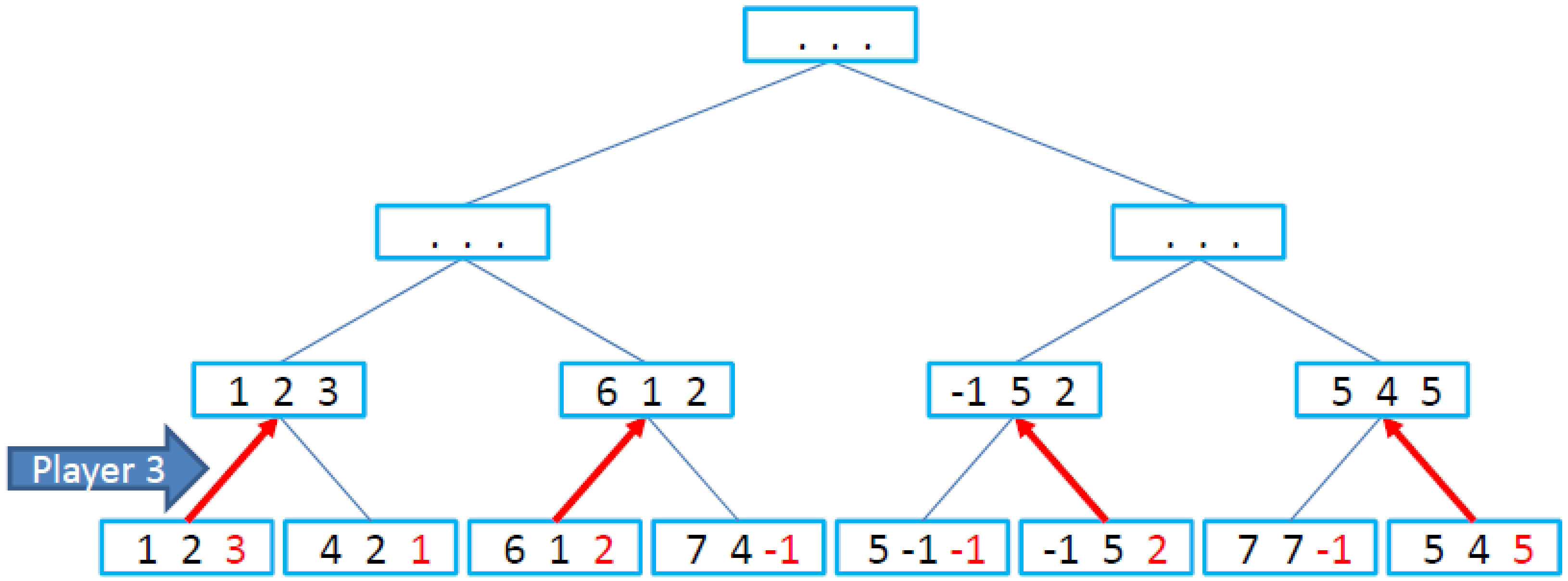
MiniMax For 3 Players

- Max third player: third position of vector



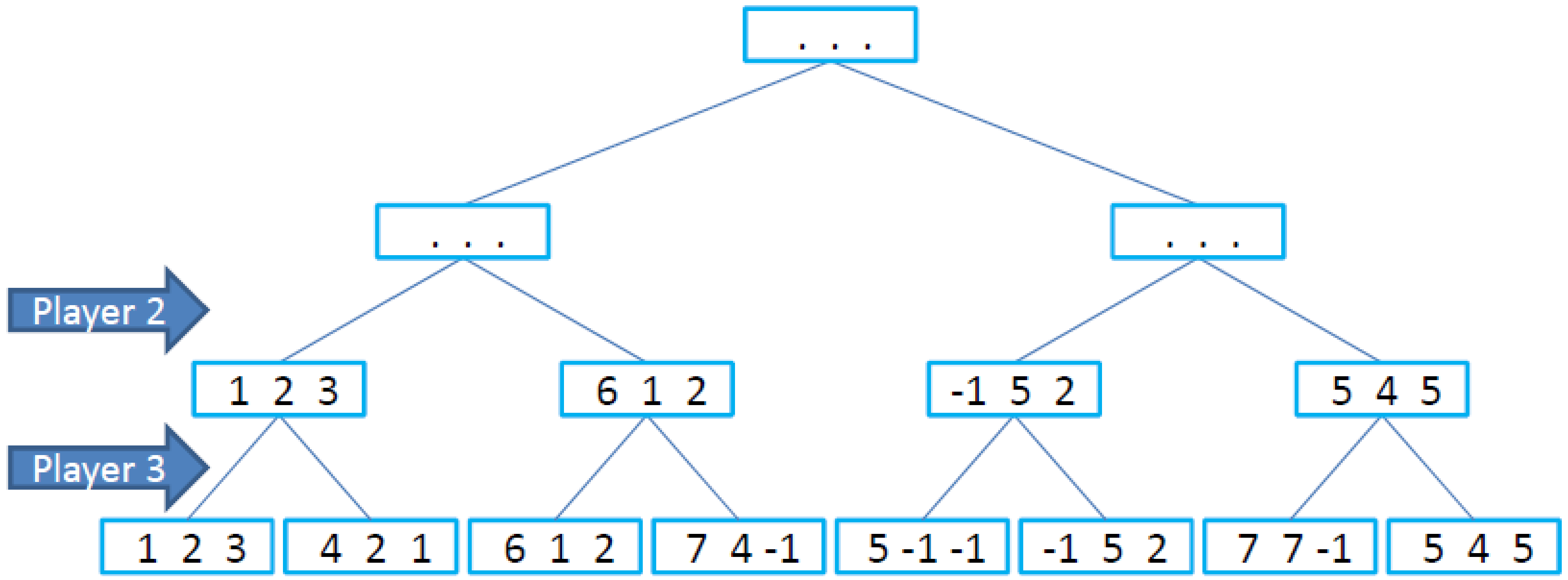
MiniMax For 3 Players

- MaxThirdPlayer([1,2,3],[4,2,1]) = [1,2,3]
- MaxThirdPlayer([6,1,2],[7,4,-1]) = [6,1,2]
- MaxThirdPlayer([5,-1,-1],[-1,5,2]) = [-1,5,2]
- MaxThirdPlayer([7,7,-1],[5,4,5]) = [5,4,5]



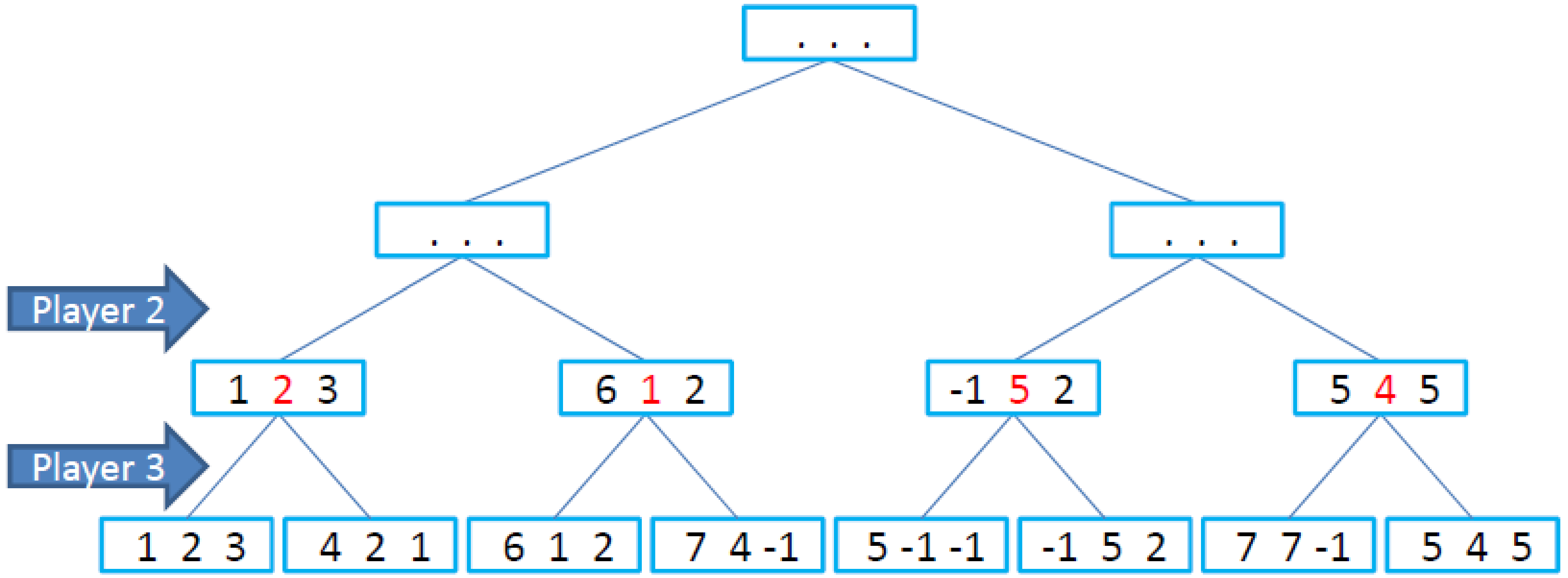
MiniMax For 3 Players

- Second player's move



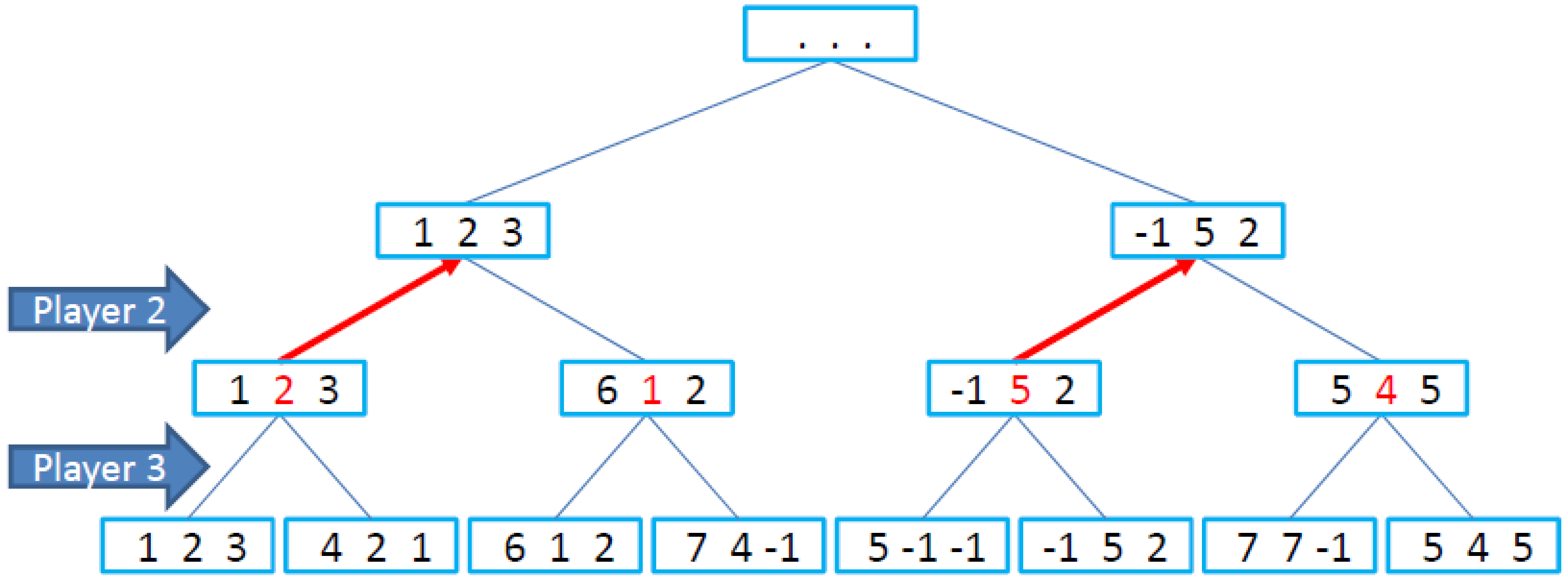
MiniMax For 3 Players

- Max second player: second position of vector



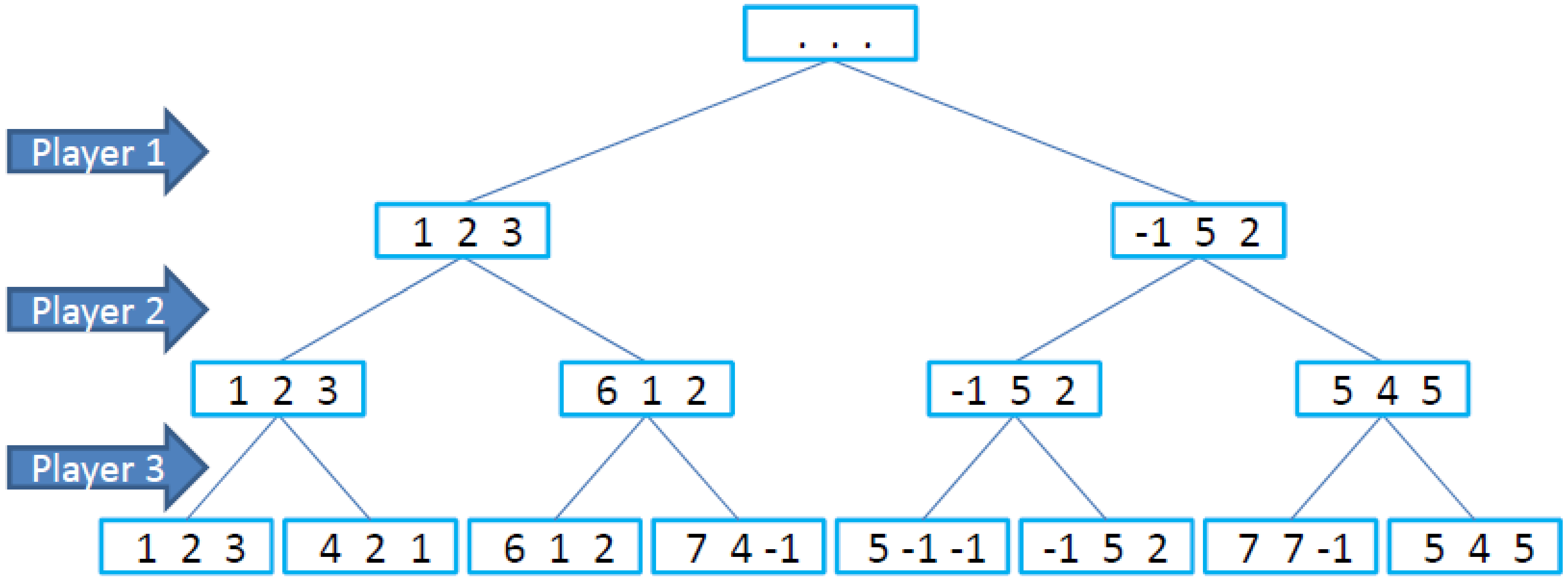
MiniMax For 3 Players

- MaxSecondPlayer([1,2,3],[6,1,2]) = [1,2,3]
- • MaxSecondPlayer([-1,5,2],[5,4,5]) = [-1,5,2]



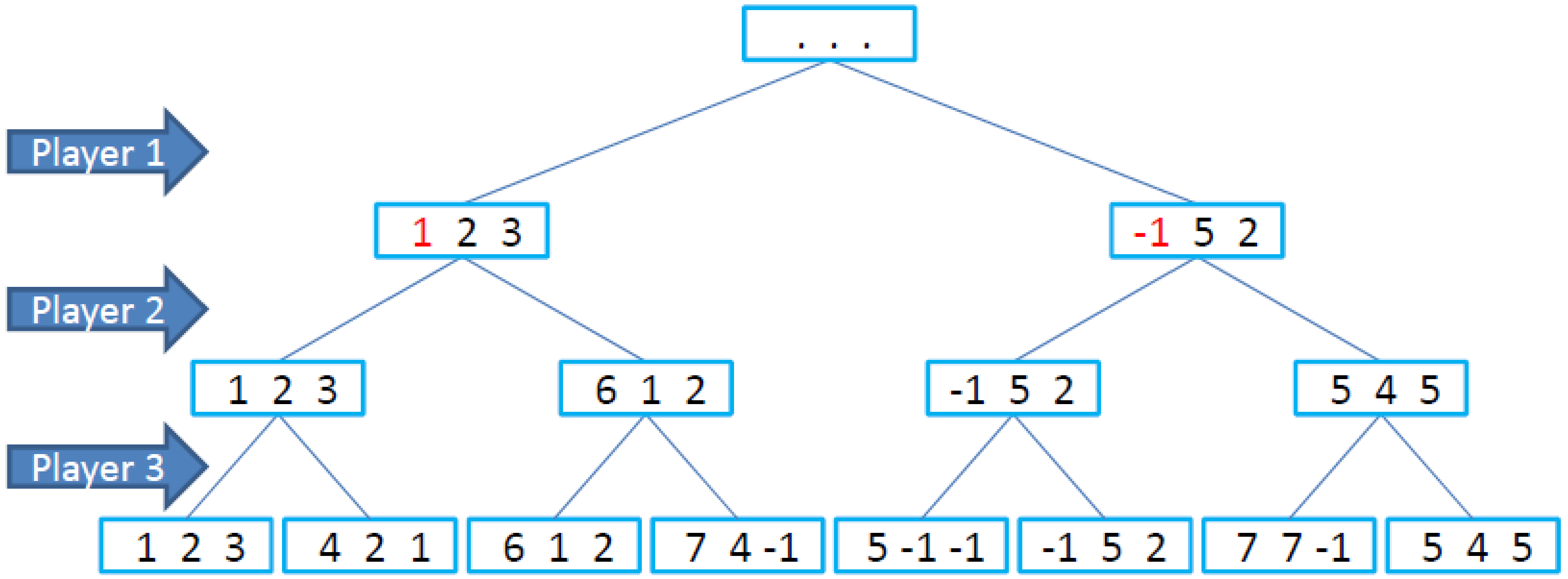
MiniMax For 3 Players

- First player's move



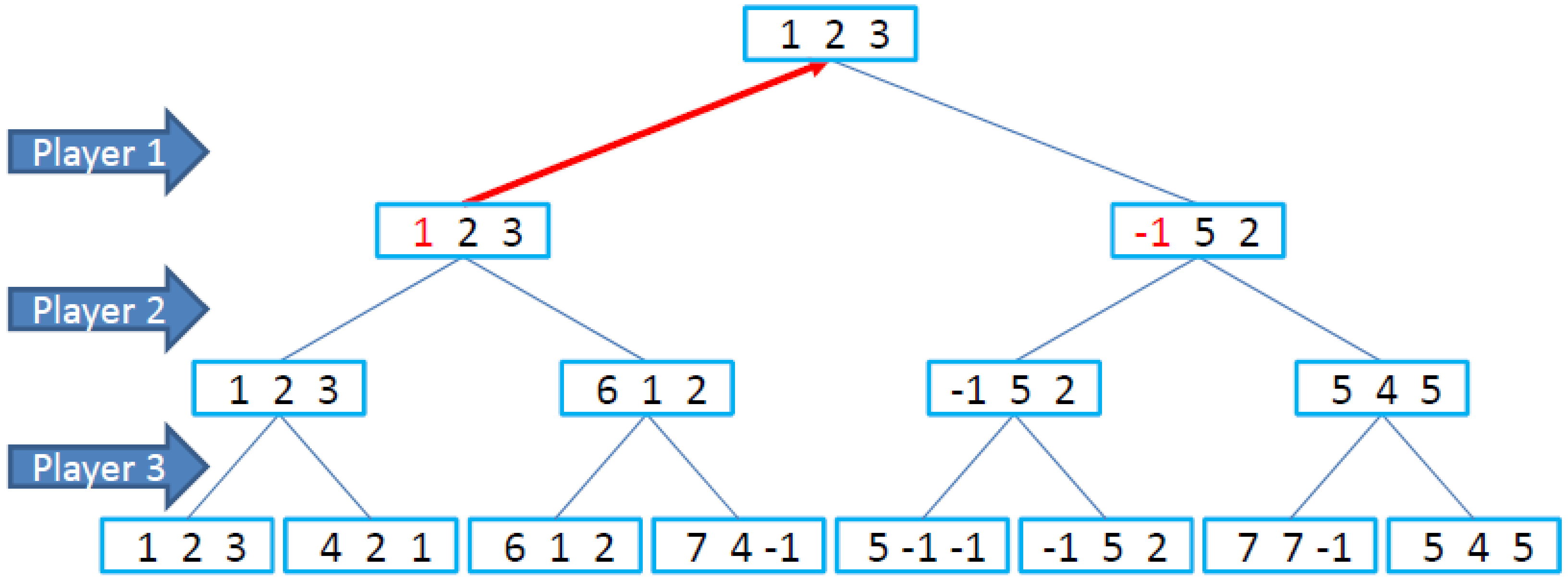
MiniMax For 3 Players

- Max first player: first position of vector



MiniMax For 3 Players

- MaxFirstPlayer([1,2,3],[-1,5,4]) = [1,2,3]


1 2 3
Player 1
1 2 3
-1 5 2
Player 2
1 2 3
6 1 2
-1 5 2
5 4 5
Player 3
1 2 3
4 2 1
6 1 2
7 4 -1
5 -1 -1
-1 5 2
7 7 -1
5 4 5