Games and adversarial search

Why study games?

- Traditional hallmark of intelligence
- Easy to formalize
- Good model of real-world competitive activities
  - Military confrontations, negotiation, auctions, etc.

Types of game environments

	Deterministic	**Stochastic**
Perfect information (fully observable)	Chess, checkers, go	Backgammon, monopoly
Imperfect information (partially observable)	Battleships	Scrabble, bridge

Alternating two-player zero-sum games

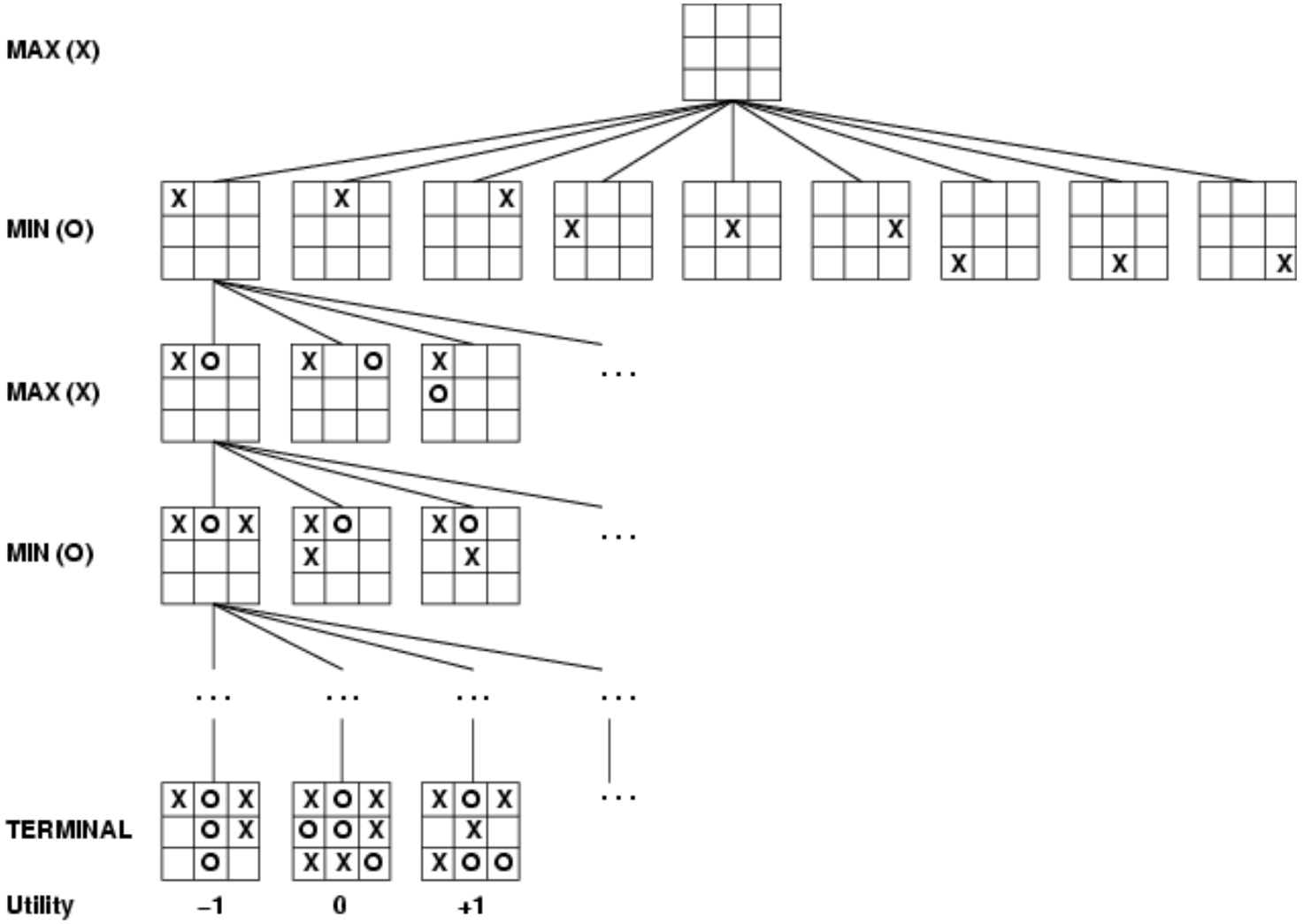
- Players take turns
- **Utility** for each player (e.g., 1 for win, 0 for loss) for each game outcome or **terminal state**
- Constant sum of both players' utilities

Games vs. single-agent search

- We don't know how the opponent will act
  - Solution: ***strategy*** or ***policy*** (a mapping from state to best move in that state)
- Efficiency is critical to playing well
  - Limited time to make a move
  - Huge branching factor, search depth, and number of terminal configurations
    - In chess, branching factor ≈ 35 and depth ≈ 100, giving a search tree of 10^{154} nodes
  - Searching all the way to the end of the game – ruled out

Game tree

- A game of tic-tac-toe between two players, “max” and “min”

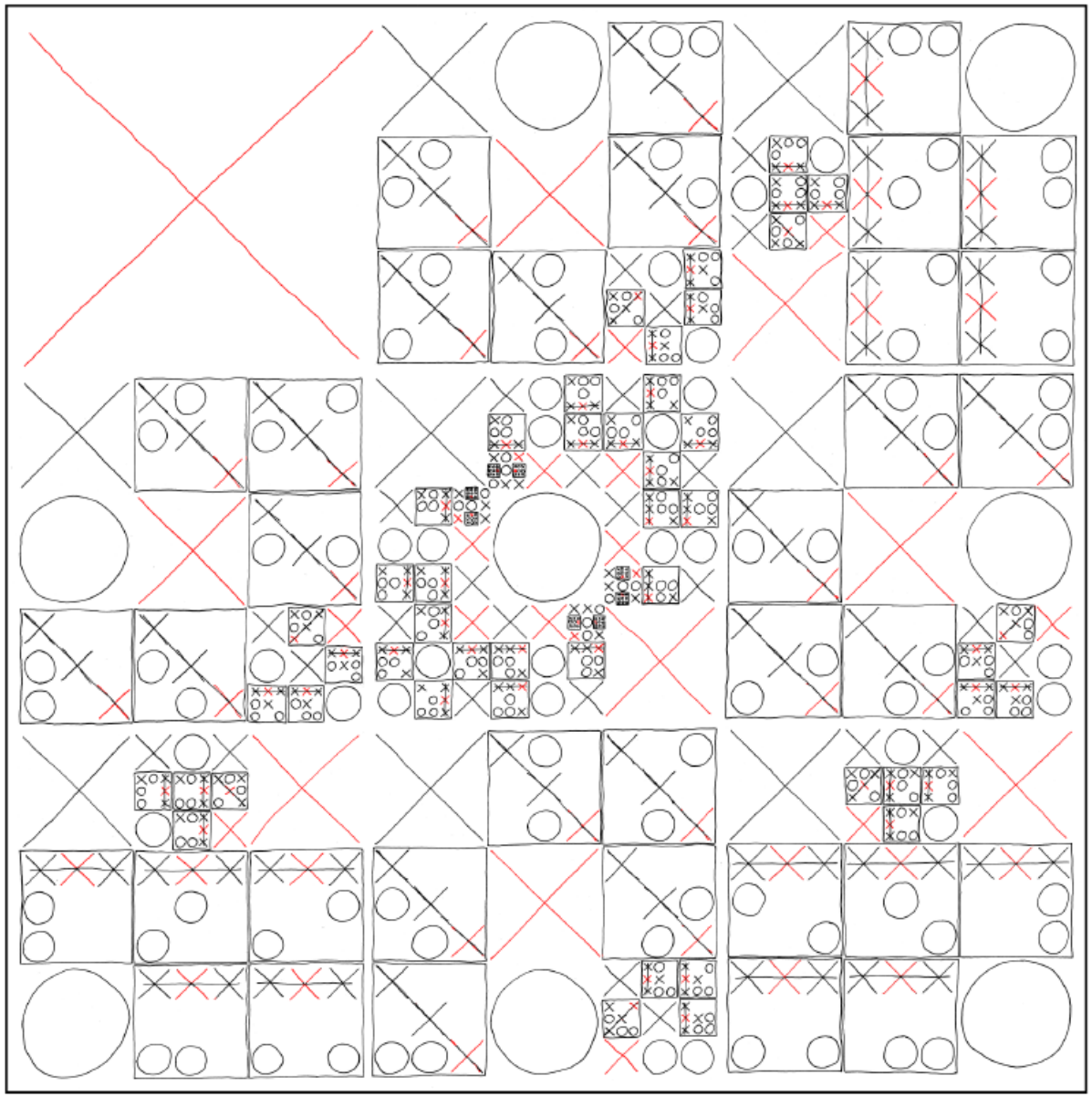


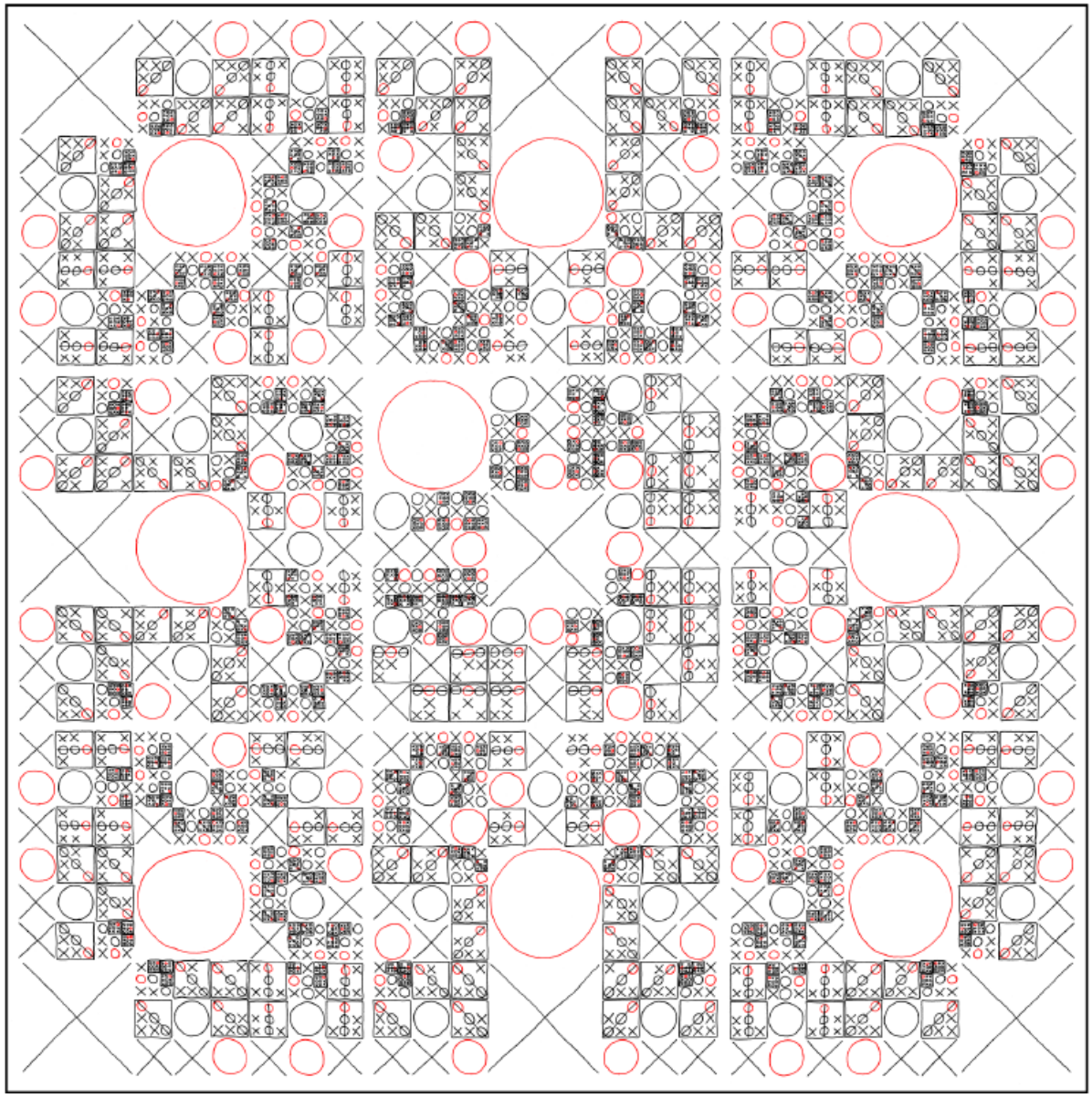
COMPLETE MAP OF OPTIMAL TIC-TAC-TOE MOVES

YOUR MOVE IS GIVEN BY THE POSITION OF THE LARGEST RED SYMBOL ON THE GRID. WHEN YOUR OPPONENT PICKS A MOVE, ZOOM IN ON THE REGION OF THE GRID WHERE THEY WENT. REPEAT.

http://xkcd.com/832/

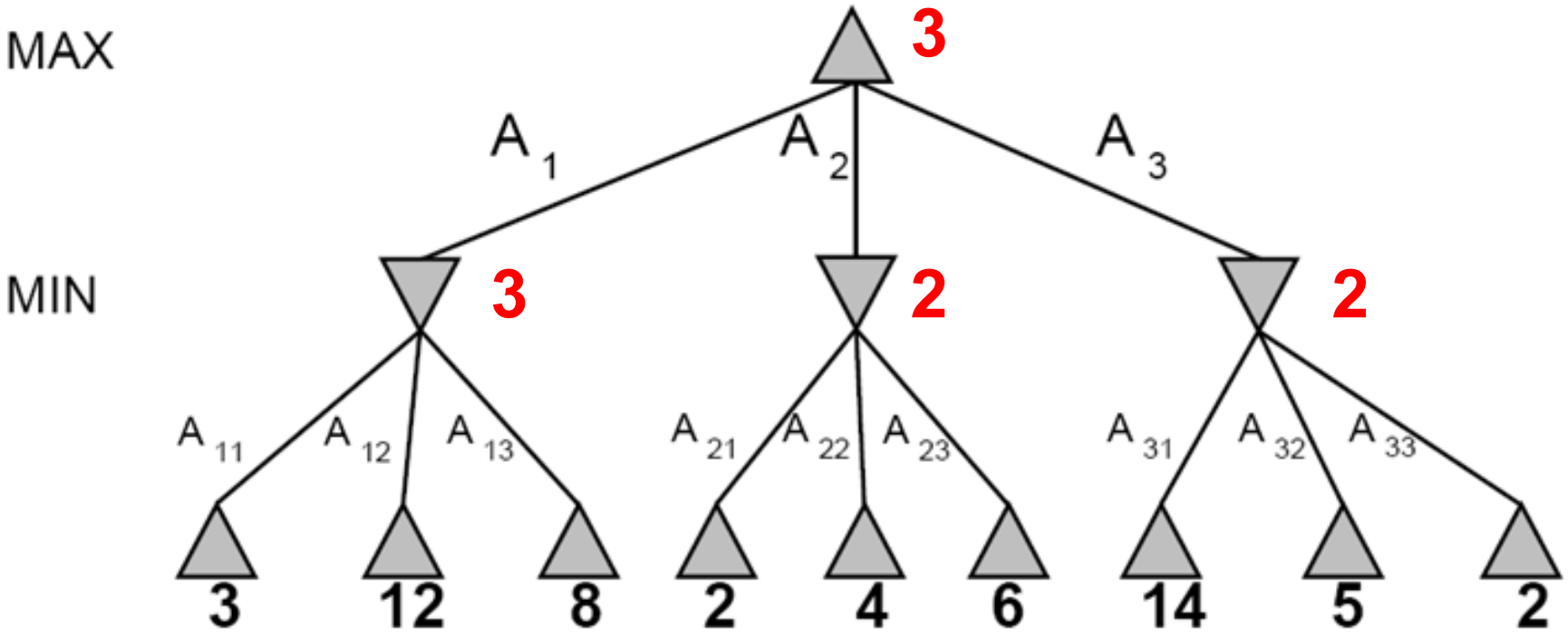
MAP FOR X:





http://xkcd.com/832/

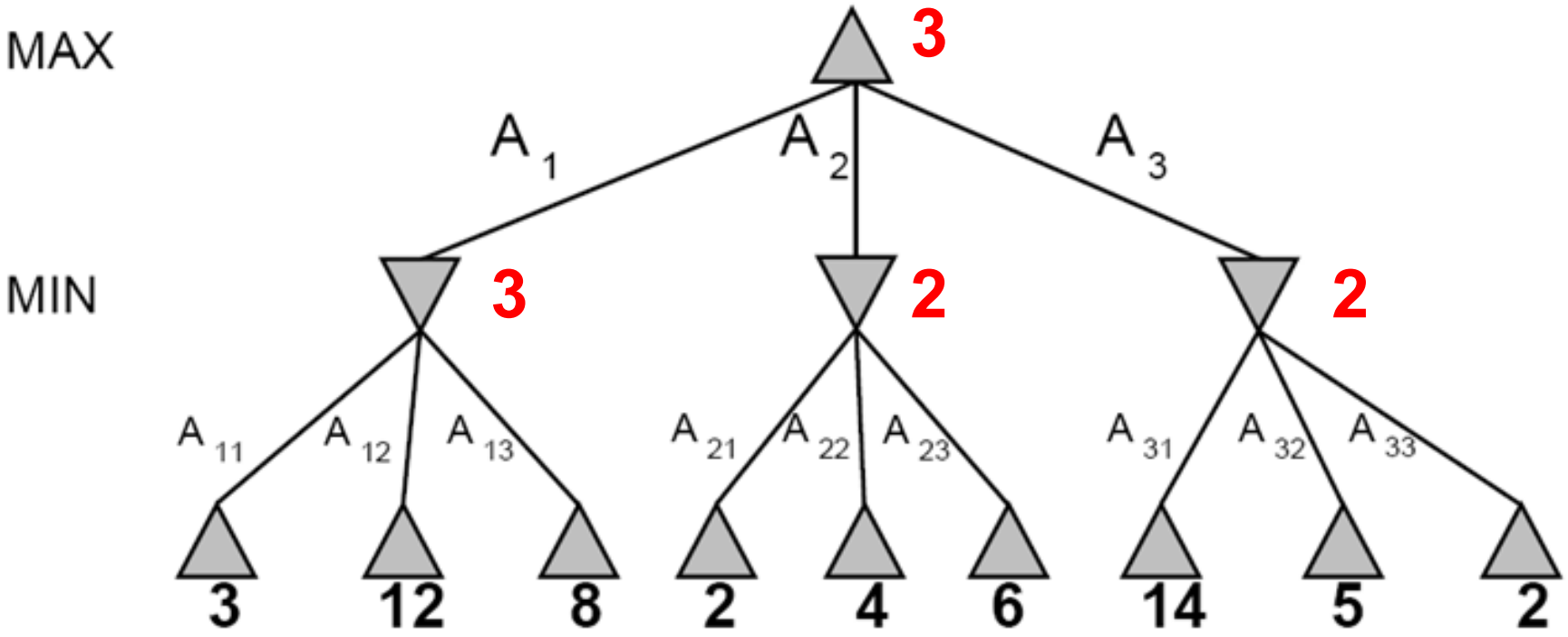
A more abstract game tree



Terminal utilities (for MAX)

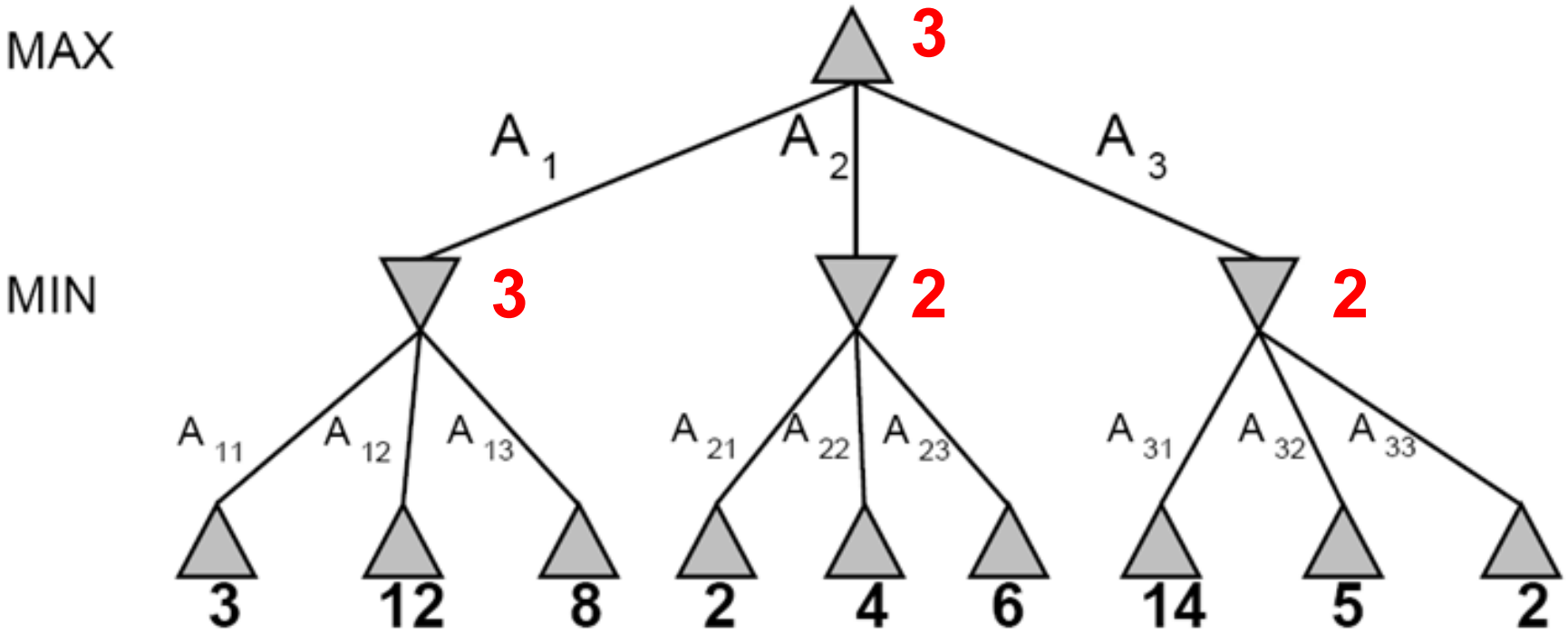
A *two-player* game

A more abstract game tree



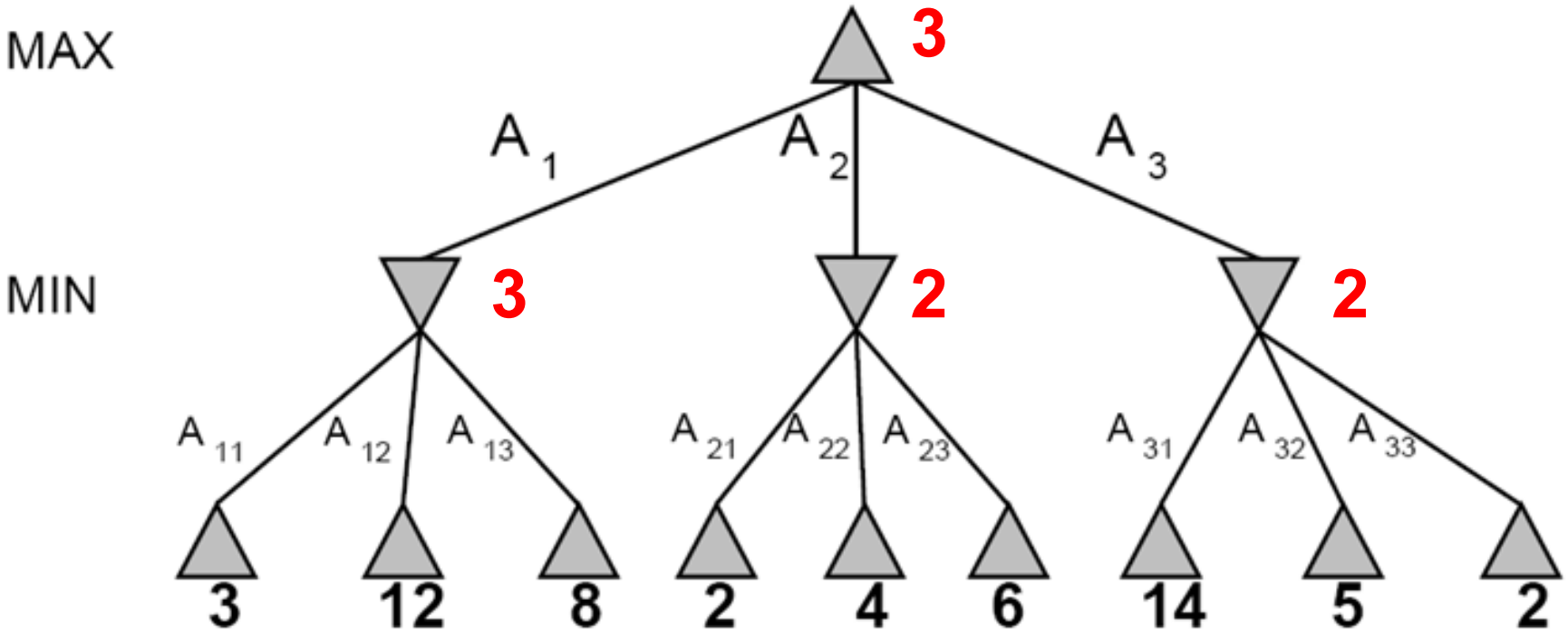
- **Minimax value of a node**: the utility (for MAX) of being in the corresponding state, assuming perfect play on both sides
- **Minimax strategy:** Choose the move that gives the best worst-case payoff

Computing the minimax value of a state



- **Minimax**(*state*) =
  - Utility(*state*) if *state* is terminal
  - max **Minimax**(successors(*state*)) if *player* = MAX
  - min **Minimax**(successors(*state*)) if *player* = MIN

Computing the minimax value of a state



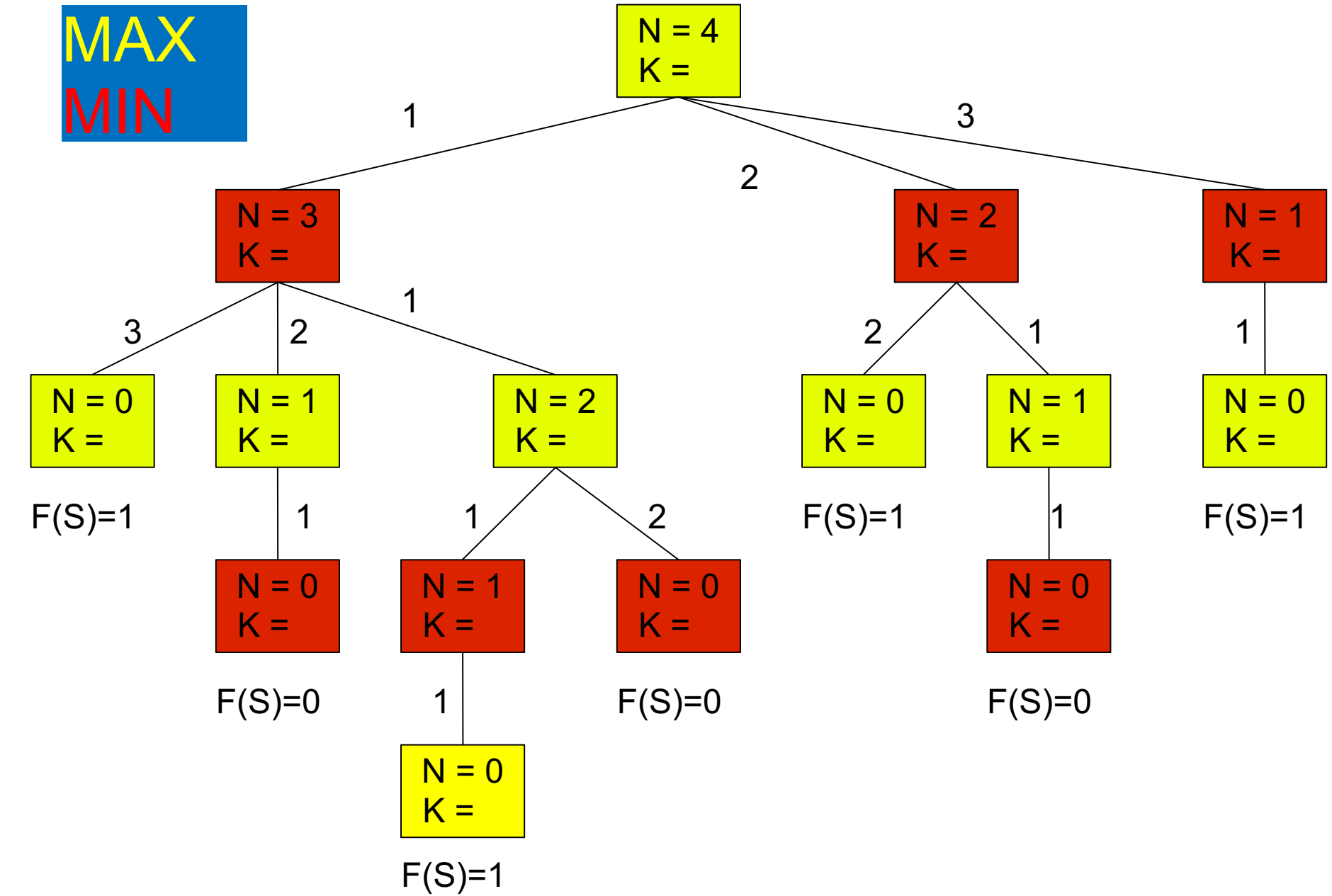
- The minimax strategy is optimal against an optimal opponent
  - If the opponent is sub-optimal, the utility can only be higher
  - A different strategy may work better for a sub-optimal opponent, but it will necessarily be worse against an optimal opponent

Example

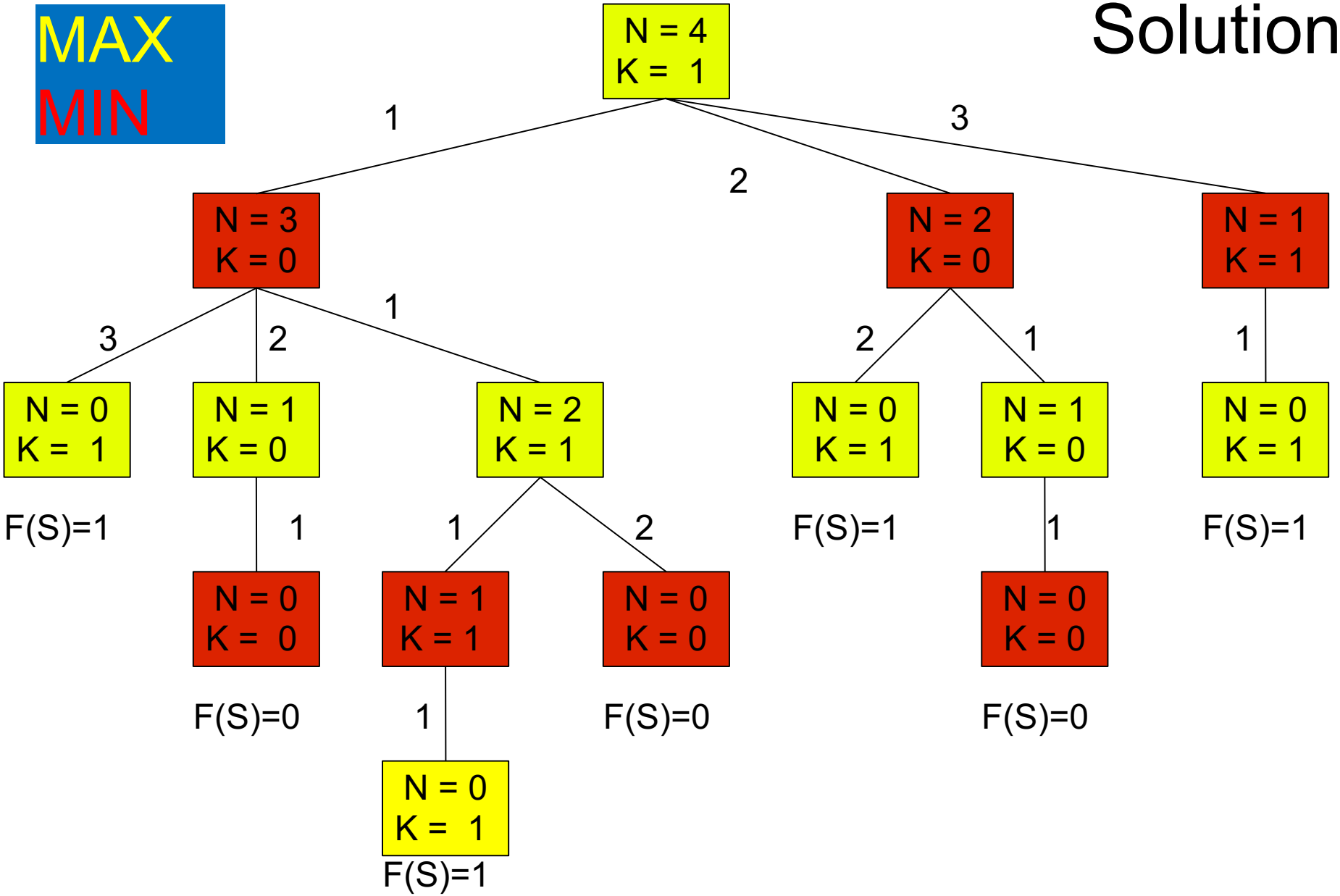
- Coins game
  - There is a stack of N coins
  - In turn, players take 1, 2, or 3 coins from the stack
  - The player who takes the last coin loses

Coins Game: Formal Definition

- Initial State: The number of coins in the stack
- Operators:
  1. Remove one coin
  2. Remove two coins
  3. Remove three coins
- Terminal Test: There are no coins left on the stack
- Utility Function: F(S)
  - F(S) = 1 if MAX wins, 0 if MIN wins


MAX
MIN
N = 4
K =
1
2
3
N = 3
K =
N = 2
K =
N = 1
K =
3
2
1
2
1
1
N = 0
K =
N = 1
K =
N = 2
K =
N = 0
K =
N = 1
K =
N = 0
K =
F(S)=1
1
1
2
F(S)=1
1
F(S)=1
N = 0
K =
N = 1
K =
N = 0
K =
N = 0
K =
F(S)=0
1
F(S)=0
F(S)=0
N = 0
K =
F(S)=1

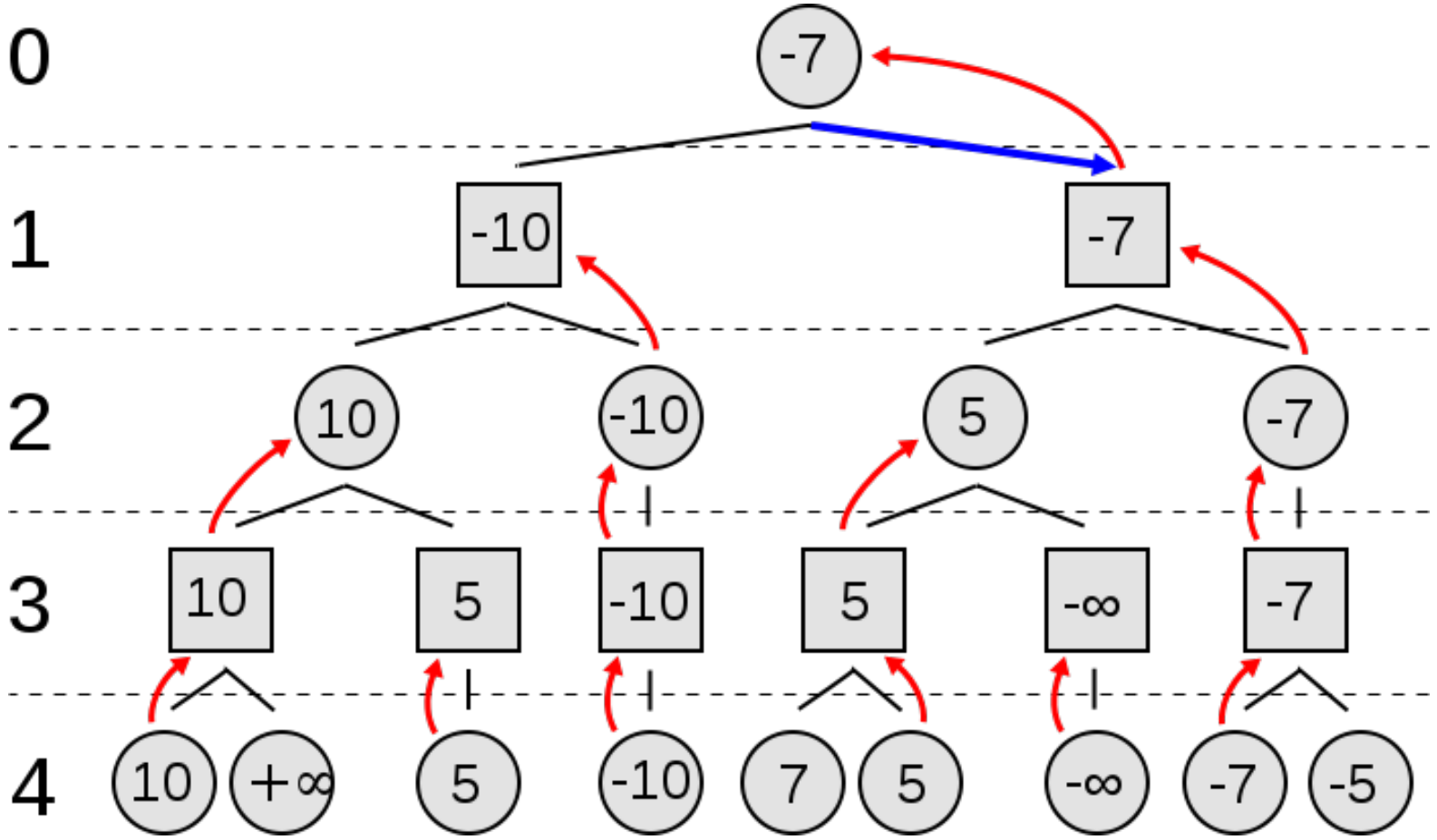
Solution


MAX
MIN
N = 4
K = 1
1
2
3
N = 3
K = 0
N = 2
K = 0
N = 1
K = 1
3
2
1
2
1
1
N = 0
K = 1
N = 1
K = 0
N = 2
K = 1
N = 0
K = 1
N = 1
K = 0
N = 0
K = 1
F(S)=1
1
1
2
F(S)=1
1
F(S)=1
N = 0
K = 0
N = 1
K = 1
N = 0
K = 0
N = 0
K = 0
F(S)=0
1
F(S)=0
F(S)=0
N = 0
K = 1
F(S)=1

Analysis

- Max Depth: 5
- Branch factor: 3
- Number of nodes: 15
- Even with this trivial example, you can see that these trees can get very big
  - Generally, there are $O(b^d)$ nodes to search for
    - Branch factor b: maximum number of moves from each node
    - Depth d: maximum depth of the tree
  - Exponential time to run the algorithm!
  - How can we make it faster?

Another Example



Alpha-Beta Pruning

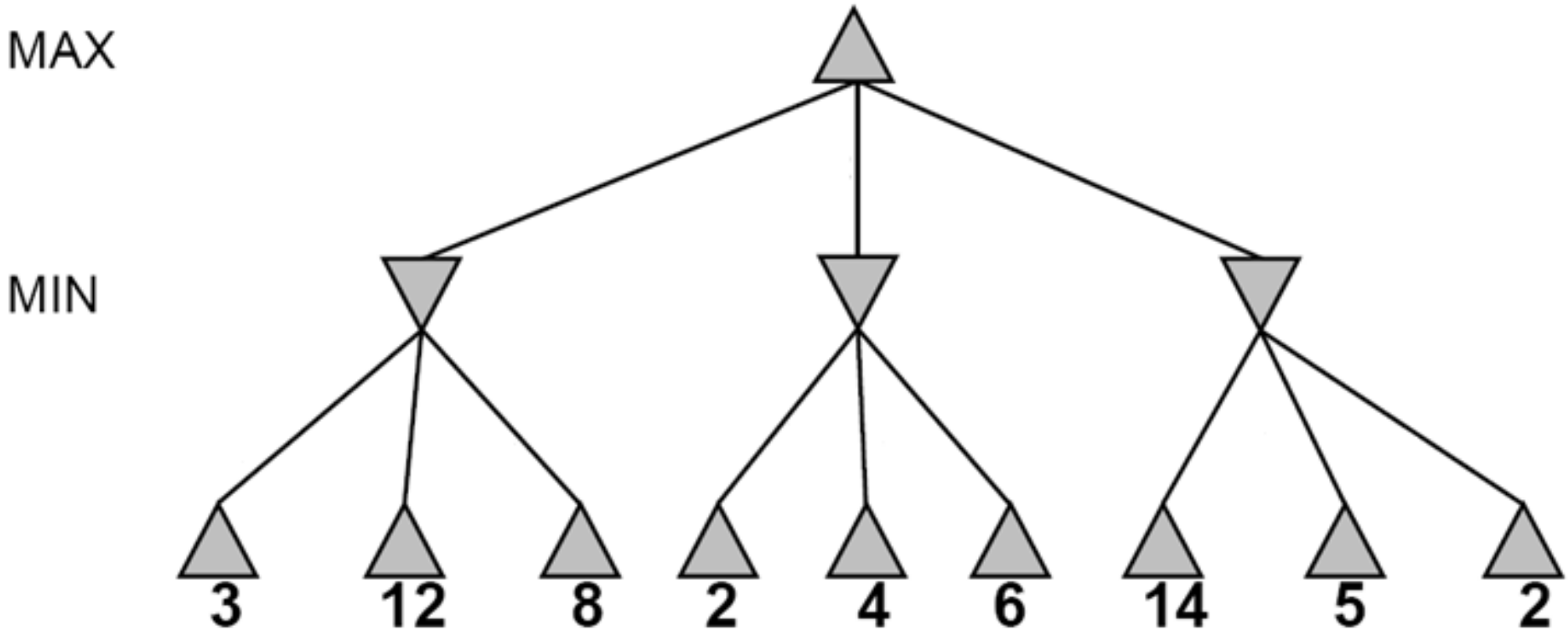
- Main idea: Avoid processing subtrees that have no effect on the result
- Two new parameters
  - α: The best value for MAX seen so far
  - β: The best value for MIN seen so far
- α is used in MIN nodes, and is assigned in MAX nodes
- β is used in MAX nodes, and is assigned in MIN nodes

Alpha-Beta Pruning

- MAX (Not at level 0)
  - If a subtree is found with a value k greater than the value of β, then we do not need to continue searching subtrees
    - MAX can do at least as good as k in this node, so MIN would never choose to go here!
- MIN
  - If a subtree is found with a value k less than the value of α, then we do not need to continue searching subtrees
    - MIN can do at least as good as k in this node, so MAX would never choose to go here!

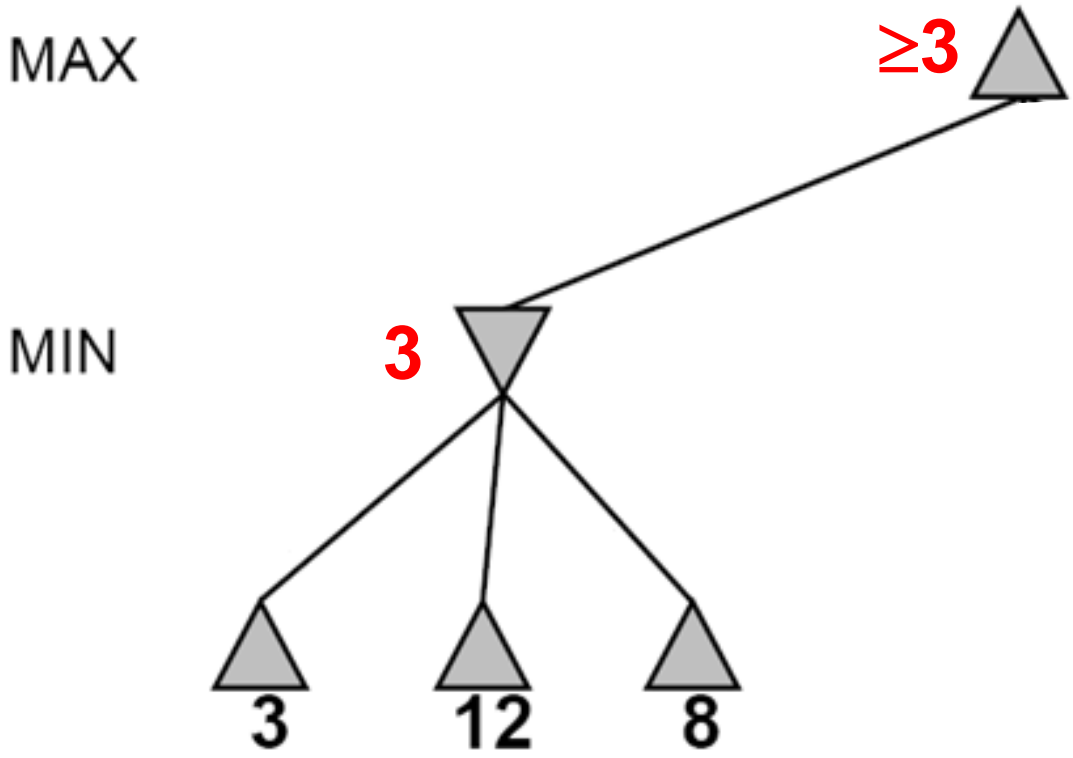
Alpha-beta pruning

- It is possible to compute the exact minimax decision without expanding every node in the game tree



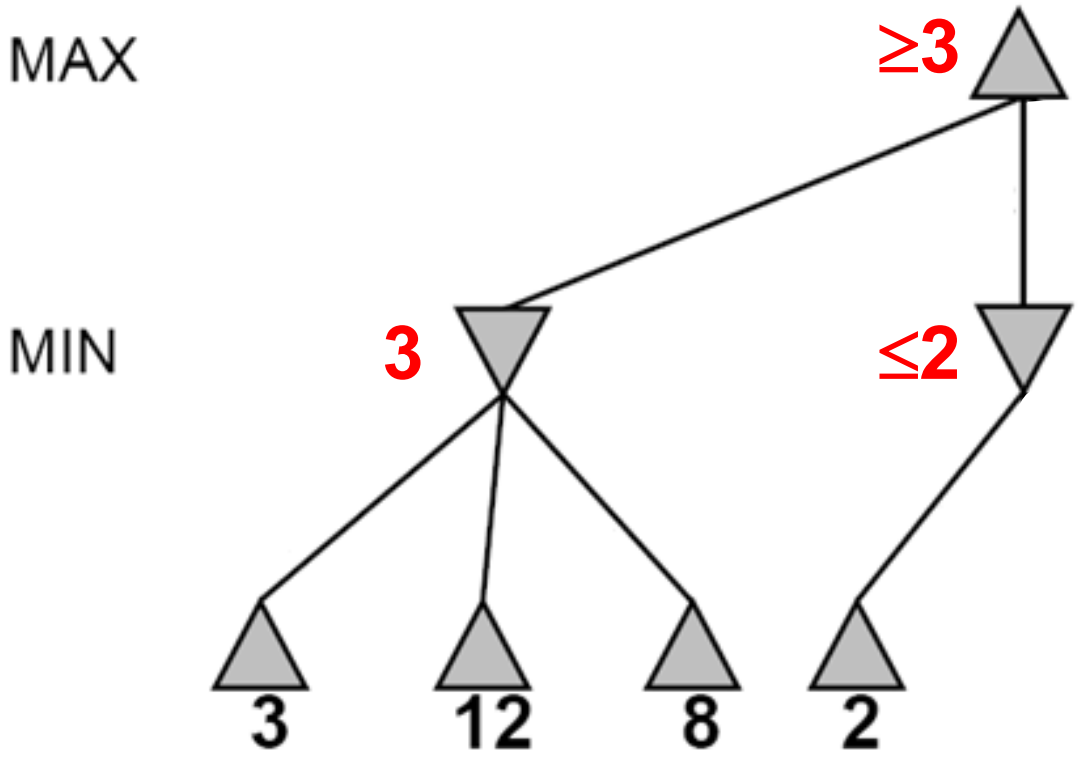
Alpha-beta pruning

- It is possible to compute the exact minimax decision without expanding every node in the game tree



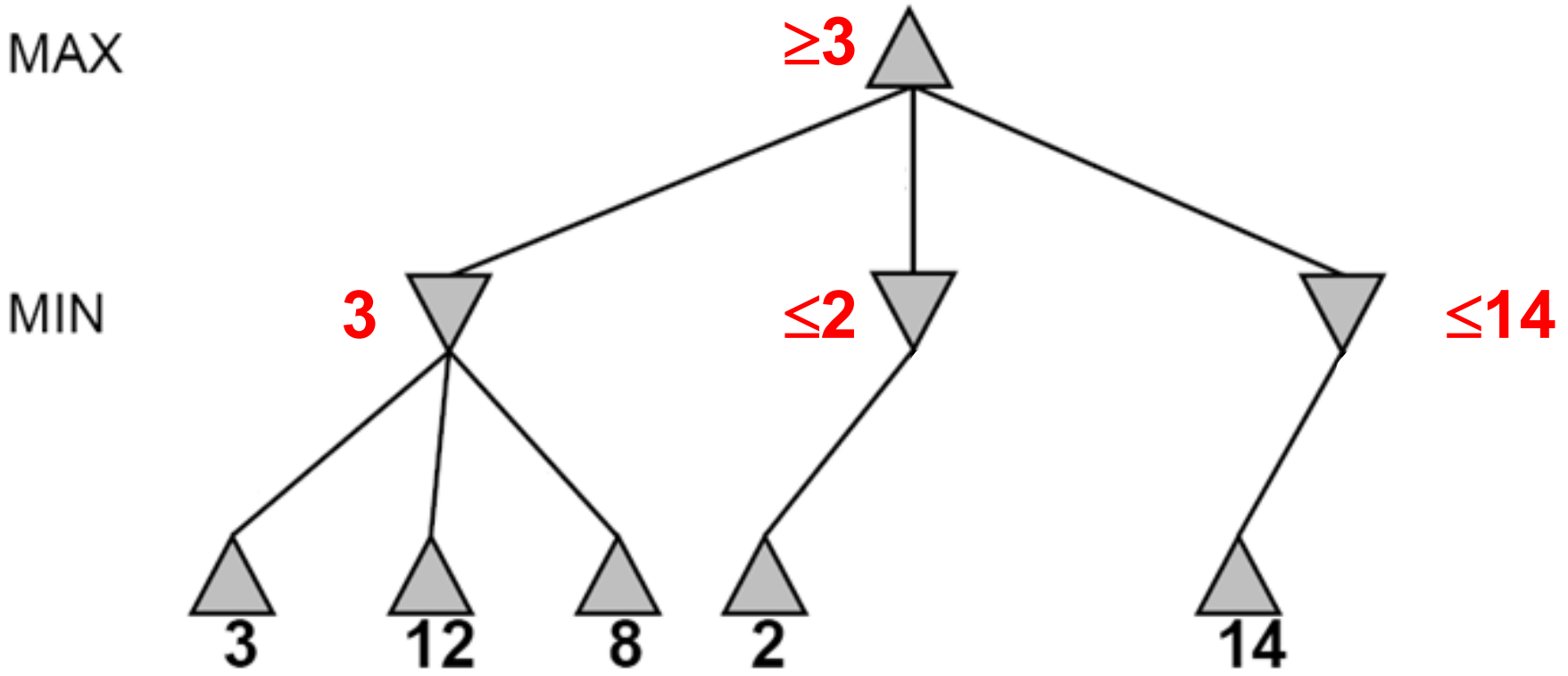
Alpha-beta pruning

- It is possible to compute the exact minimax decision without expanding every node in the game tree



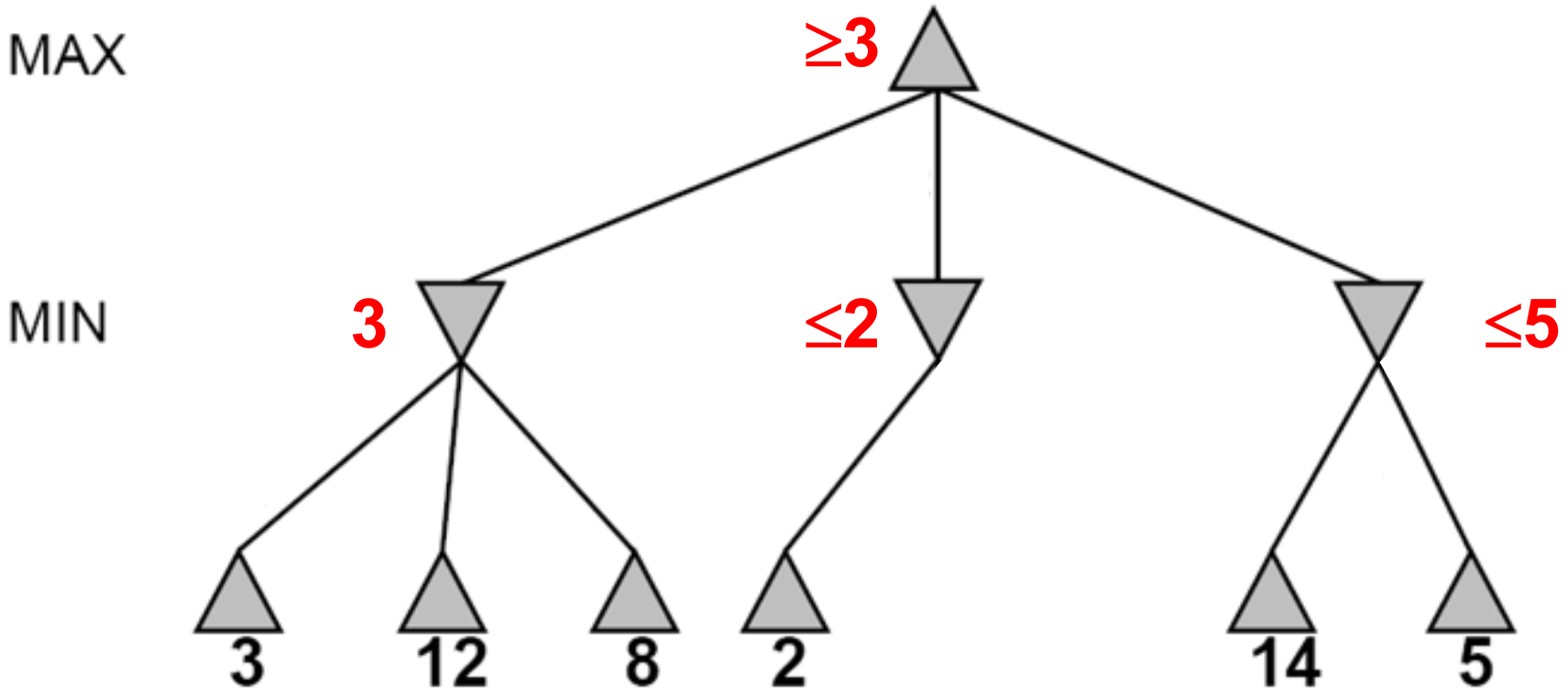
Alpha-beta pruning

- It is possible to compute the exact minimax decision without expanding every node in the game tree



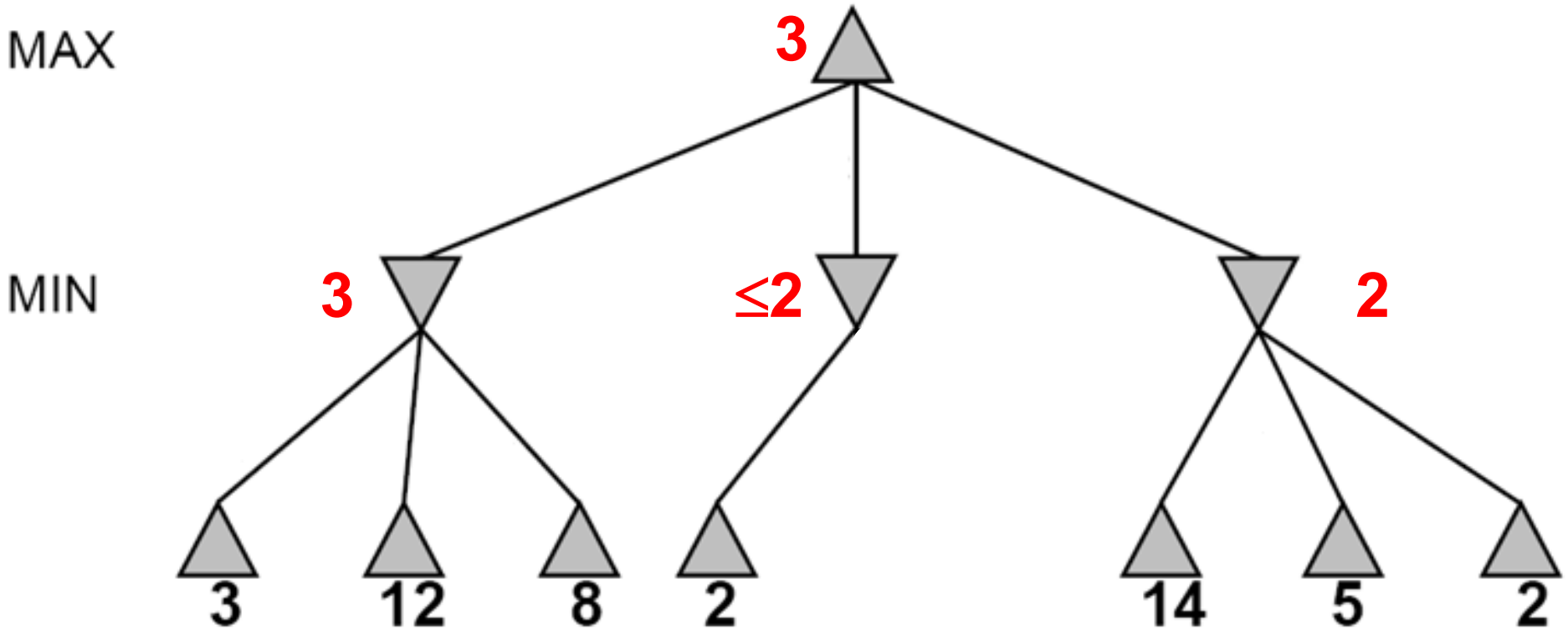
Alpha-beta pruning

- It is possible to compute the exact minimax decision without expanding every node in the game tree

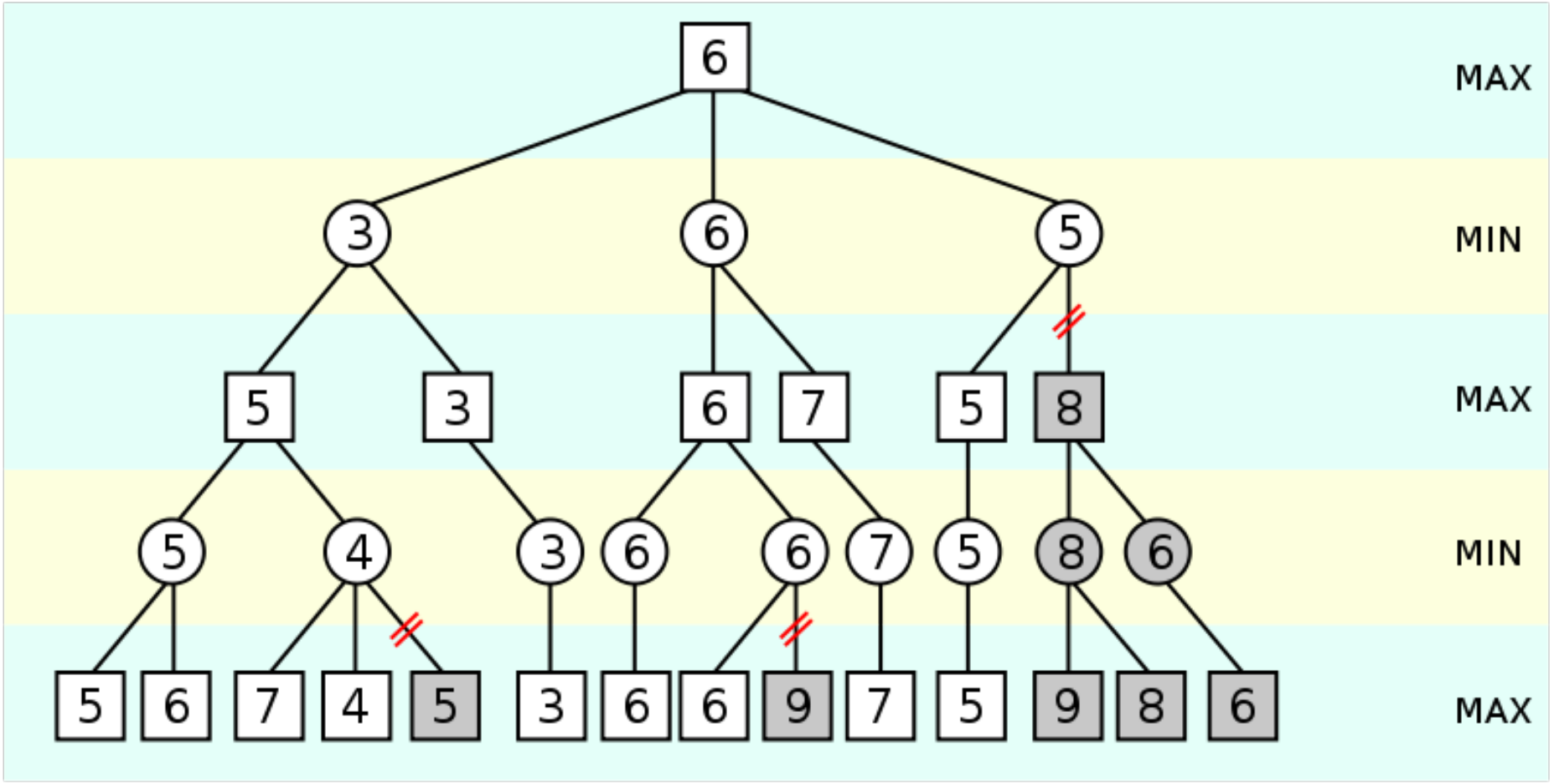


Alpha-beta pruning

- It is possible to compute the exact minimax decision without expanding every node in the game tree



Another Example



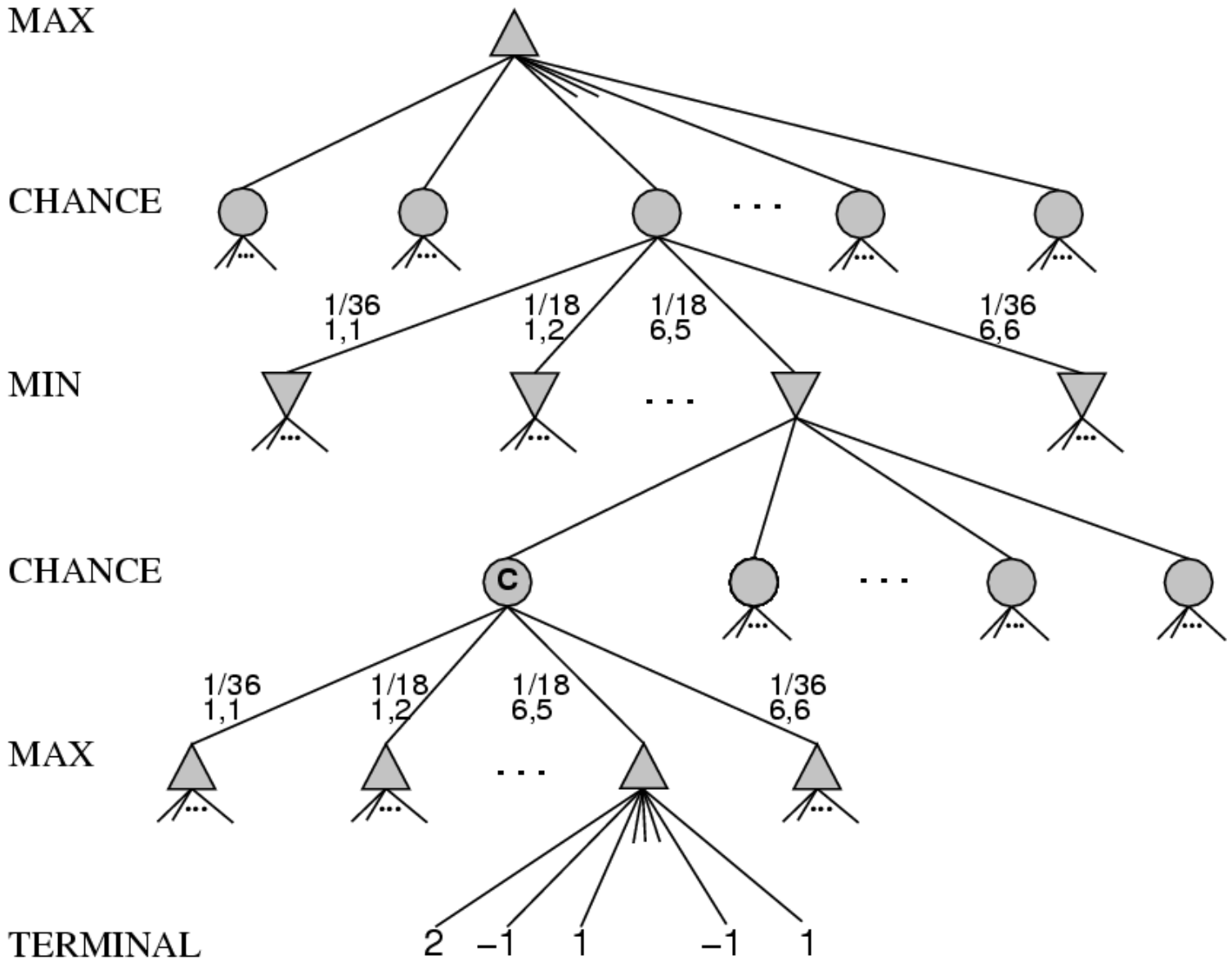
Additional techniques

- **Transposition table** to store previously expanded states and conclusions
- **Forward pruning** to avoid considering all possible moves
- **Lookup tables** for opening moves and endgames

Games of chance

- How to incorporate dice throwing into the game tree?

Games of chance



Games of chance

- **Expectiminimax:** for chance nodes, average values weighted by the probability of each outcome
  - Nasty branching factor, defining evaluation functions and pruning algorithms more difficult
- **Monte Carlo simulation:** when you get to a chance node, simulate a large number of games with random dice rolls and use win percentage as evaluation function
  - Can work well for games like Backgammon

Game playing algorithms today

- Computers are better than humans
  - **Checkers:** solved in 2007
  - **Chess:** IBM Deep Blue defeated Kasparov in 1997
- Computers are competitive with top human players
  - **Backgammon:** TD-Gammon system used reinforcement learning to learn a good evaluation function
  - **Bridge:** top systems use Monte Carlo simulation and alpha-beta search
- Computers were not competitive, but now they are ☺
  - **Go:** branching factor 361. Existing systems use Monte Carlo simulation and pattern databases. **AlphaGo's** algorithm uses a Monte Carlo tree search to find its moves based on knowledge previously "learned" by machine learning, specifically by an artificial neural network (a deep learning kind) by extensive training, both from human and computer play https://www.youtube.com/watch?v=g-dKXOlsf98
  - Latest: Google AI beats top human players at strategy game StarCraft II, link

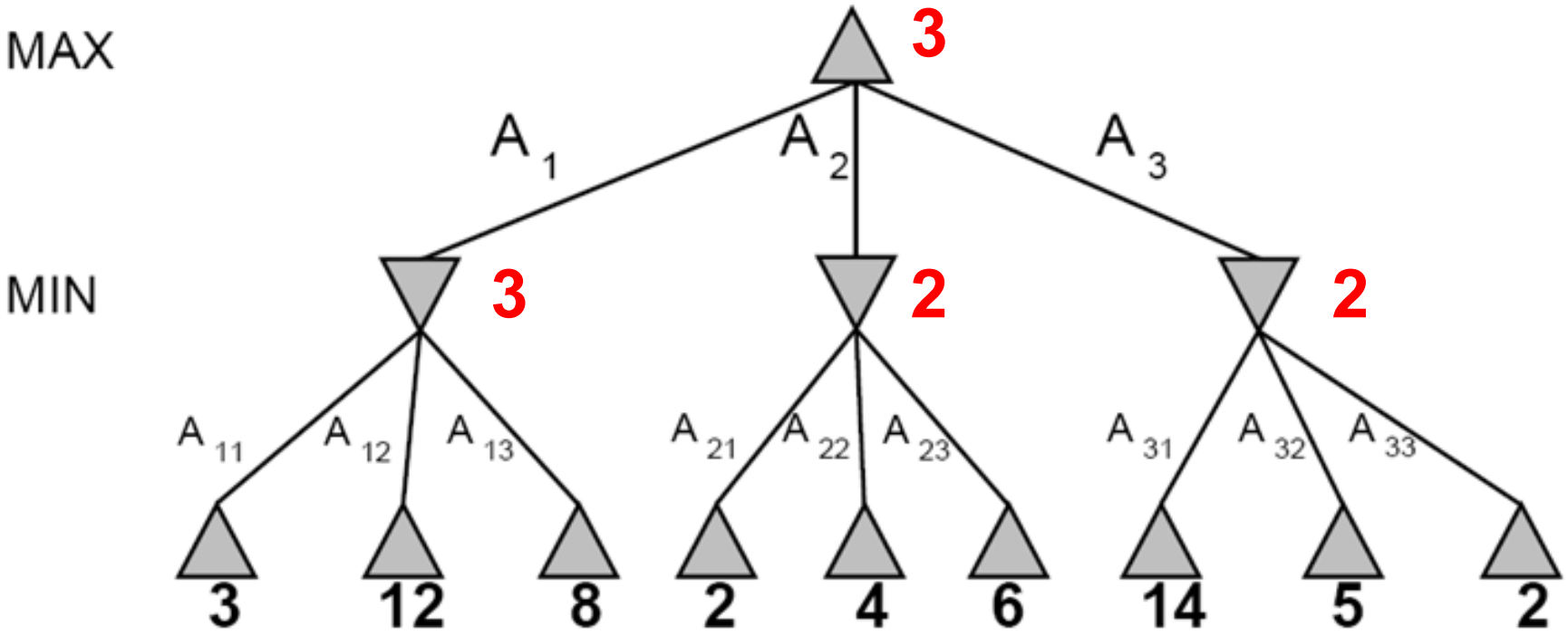
Origins of game playing algorithms

- Ernst Zermelo (1912): Minimax algorithm
- Claude Shannon (1949): chess playing with evaluation function
- John McCarthy (1956): Alpha-beta search
- Arthur Samuel (1956): checkers program that learns its own evaluation function by playing against itself

Review: Games

- What is a zero-sum game?
- What's the optimal strategy for a player in a zero-sum game?
- How do you compute this strategy?

Review: Minimax



- **Minimax**(*state*) =
  - Utility(*state*) if *state* is terminal
  - max **Minimax**(successors(*state*)) if *player* = MAX
  - min **Minimax**(successors(*state*)) if *player* = MIN

Review: Games

- Efficiency improvement: alpha-beta pruning
- Stochastic games