# Adversarial Search I

PROF. LIM KWAN HUI

50.021 Artificial Intelligence

# Outline & Objectives

- Understand the differences between standard search problems and adversarial search problems
- Understand the workings behind the Minimax algorithm and α - β pruning
- Able to use Minimax algorithm to solve an adversarial/game search problem
- Able to use α - β pruning to speed up adversarial/game search

# Recap: Environment Types

- For the previous parts on search, we assumed a simple environment, that is
  - Fully observable
  - Deterministic
  - Sequential
  - Static
  - Discrete
  - Single-agent

# Adversarial/Games VS General Search Problems

- Multi-agent vs Single-agent
- “Unpredictable” opponent ⇒ solution is a contingency plan
- Time limits ⇒ unlikely to find goal, must approximate
- Plan of attack
  - Minimax: Algorithm for perfect play
  - α - β Pruning: Finite time, approximate evaluation, pruning
  - Expecti Minimax: Chances in games/search

# Types of Games

| | Deterministic | Chance |
|---|---|---|
| **Perfect Information (Fully Observable)** | Chess, Checker, Go | Backgammon |
| **Imperfect Information (Partially Observable)** | Battleship | Bridge, Poker, Scrabble |

# Recap: Search Formulation

- **State space**, e.g. *At(Arad), At(Bucharest)*
- **Initial state**, e.g. *At(Arad)*
- **Actions**, set of actions given a specific state
  - **Transition model** e.g., *Result(At(Arad),Go(Zerind))* → *At(Zerind)*
  - **Path cost** (additive), e.g., sum of distances, number of actions, etc
- **Goal test**, can be
  - Explicit, e.g. *At(Bucharest)*
  - Implicit, e.g. *checkmate(x)*

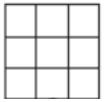
# Representing a Game as a Search Problem

- We can formally define a strategic two-player game by:
  - Initial State
  - Actions
  - Terminal Test (Win / Lose / Draw)
  - Utility Function (numerical reward for the outcome)
    - Chess: +1, 0, -1
    - Poker: Cash won or lose

- In a zero-sum game with two players
  - each player's utility for a state are equal and opposite

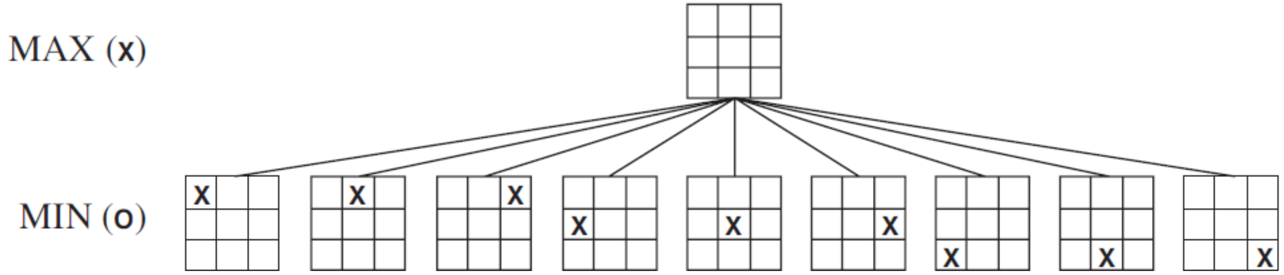
# Game Tree for Tic-tac-toe

- Initial state MAX (**x**)



# Game Tree for Tic-tac-toe

- Initial state
- Actions/Moves



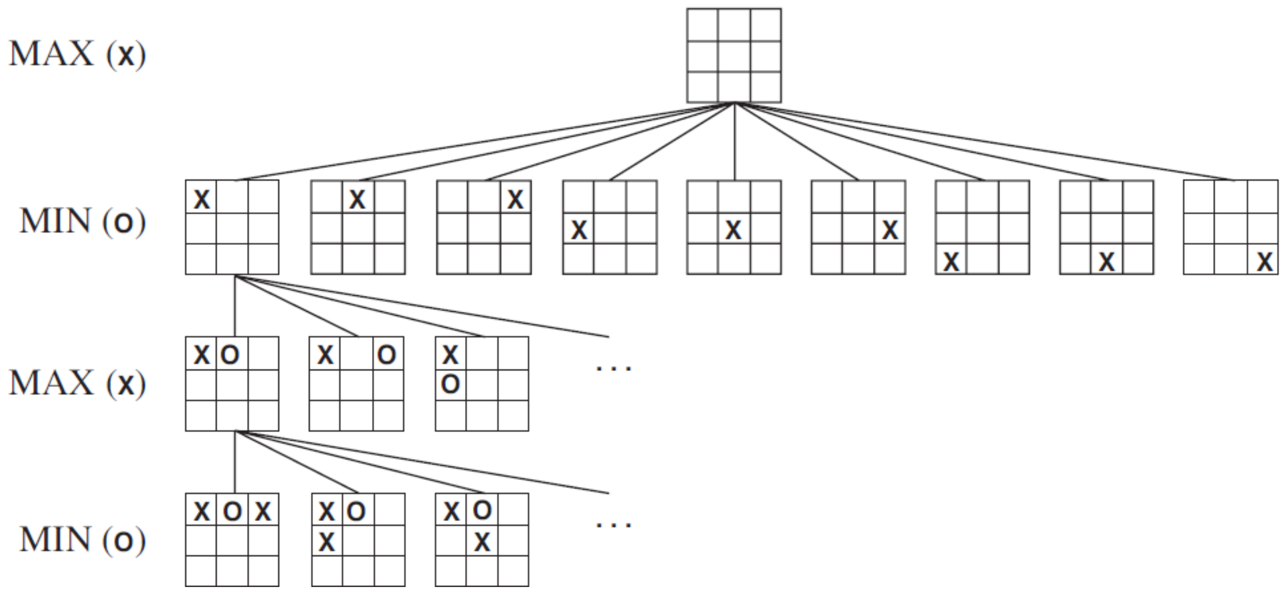
# Game Tree for Tic-tac-toe

- Initial state
- Actions/Moves

⋮

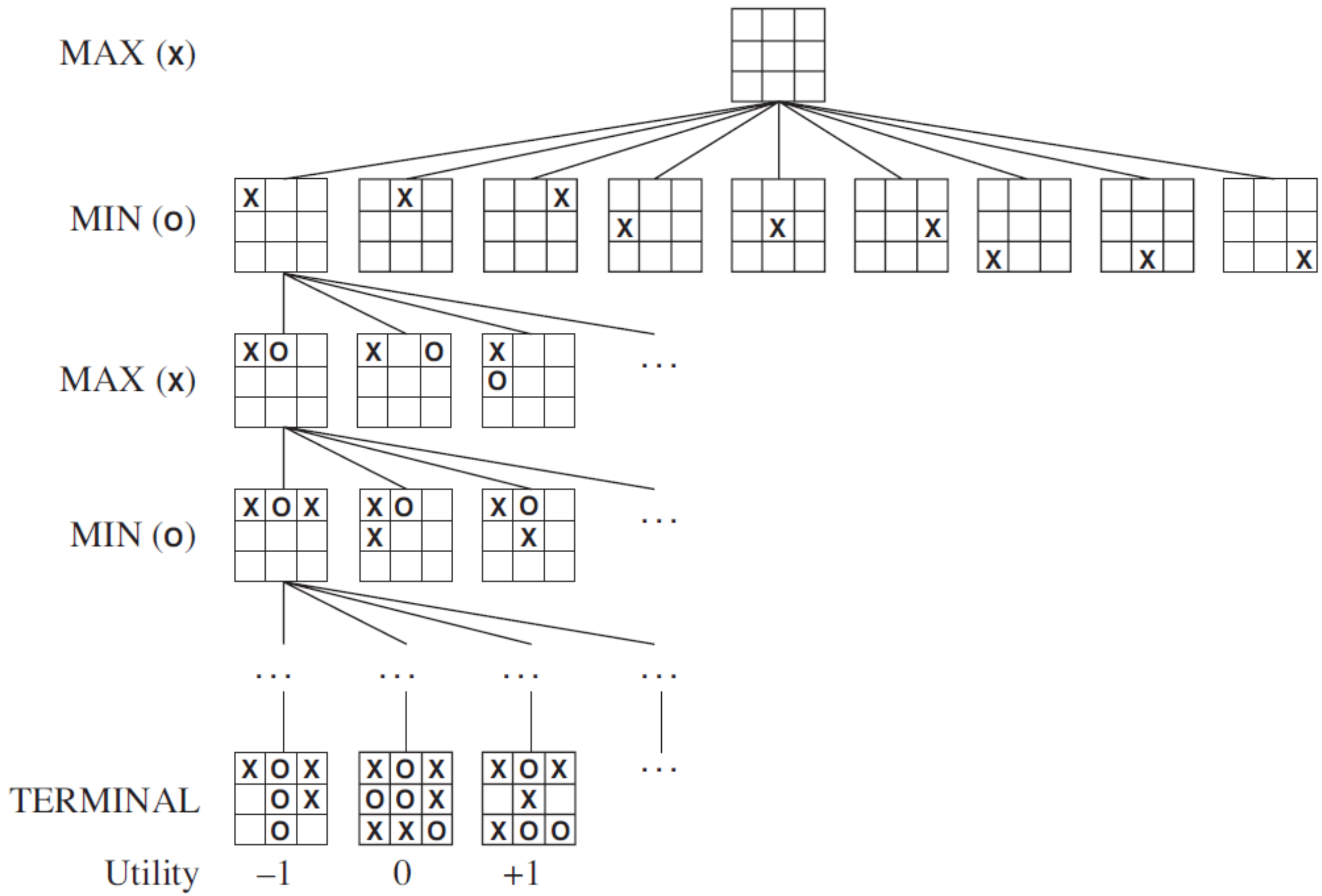
⋮

- More Actions/ Moves



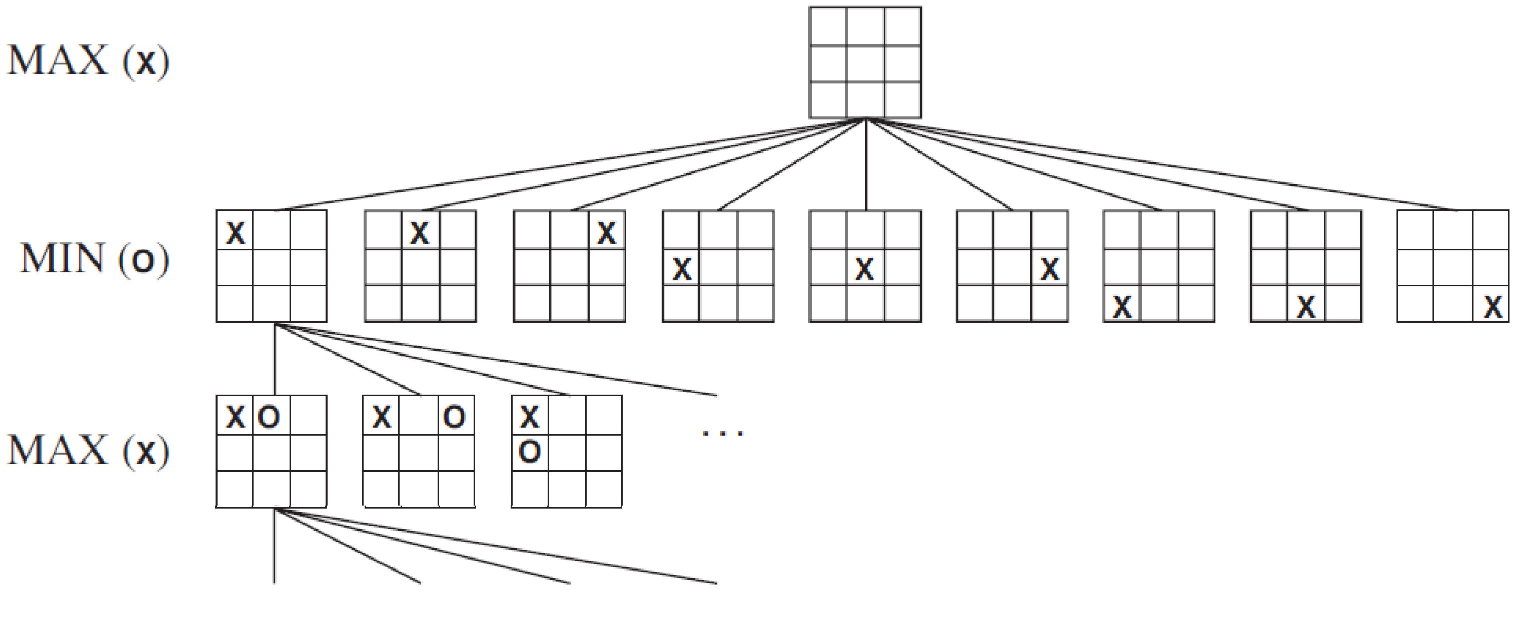
# Game Tree for Tic-tac-toe

- Initial state
- Actions/Moves
- More Actions/ Moves
- Terminal/Goal State + Utility



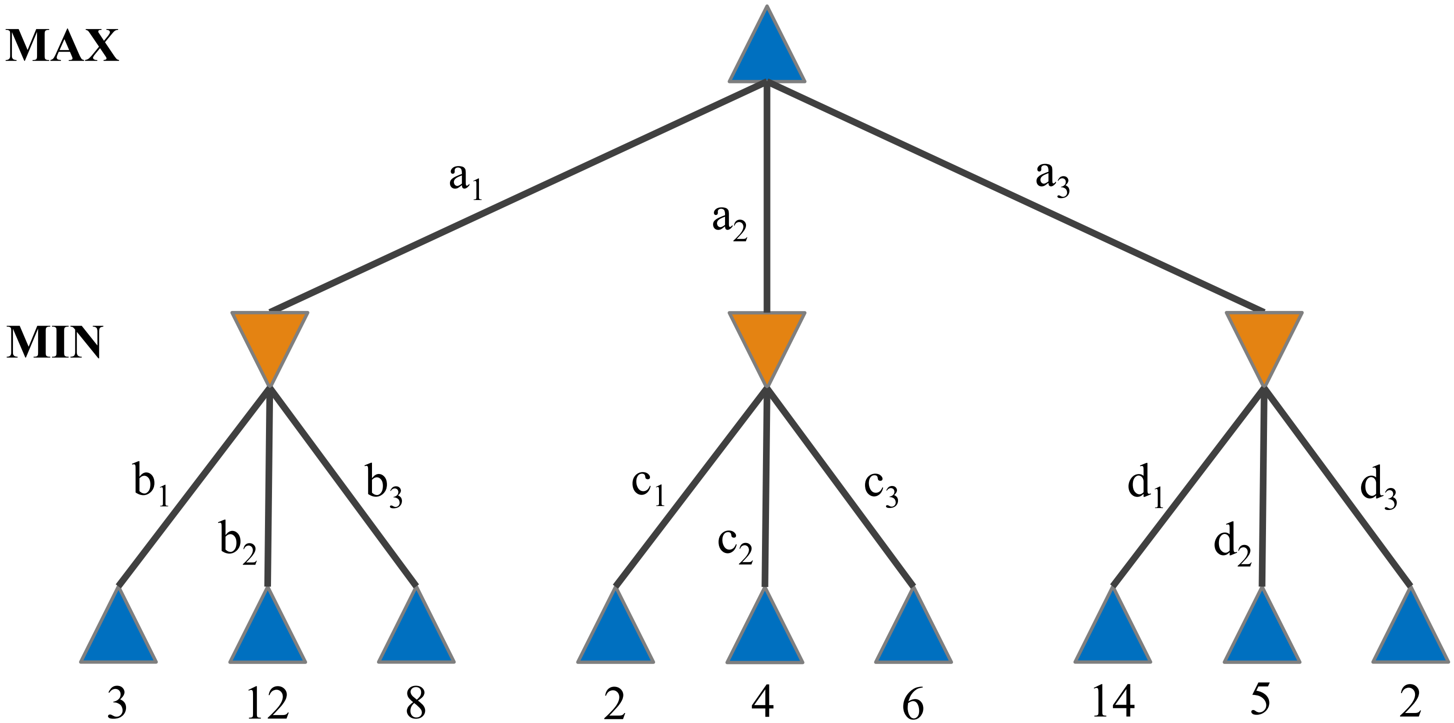
# Minimax

- Perfect play for deterministic, perfect-information (fully observable) games
- Idea: choose move to position with highest minimax value
  - = best achievable payoff against best play

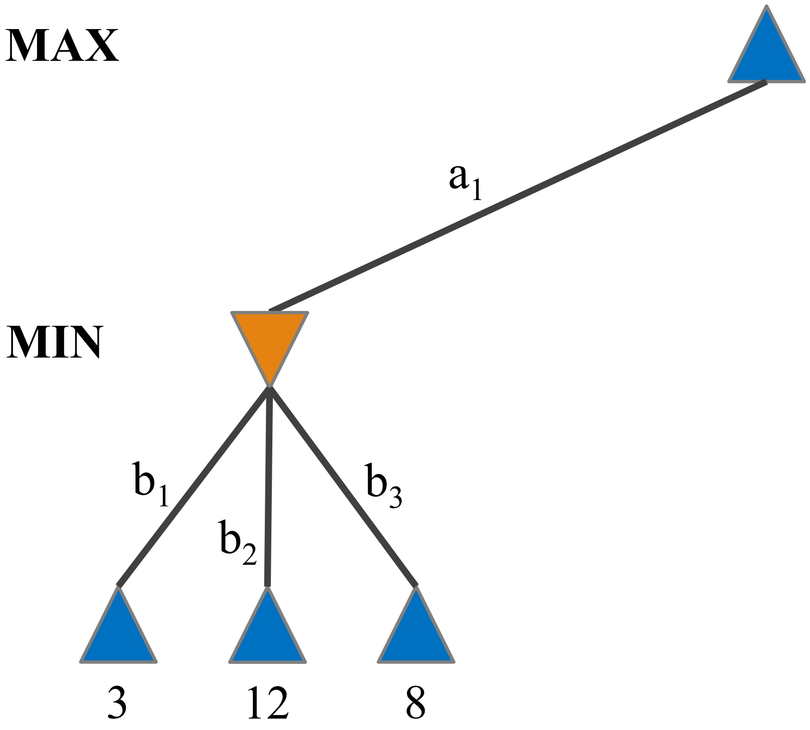


# Minimax Algorithm

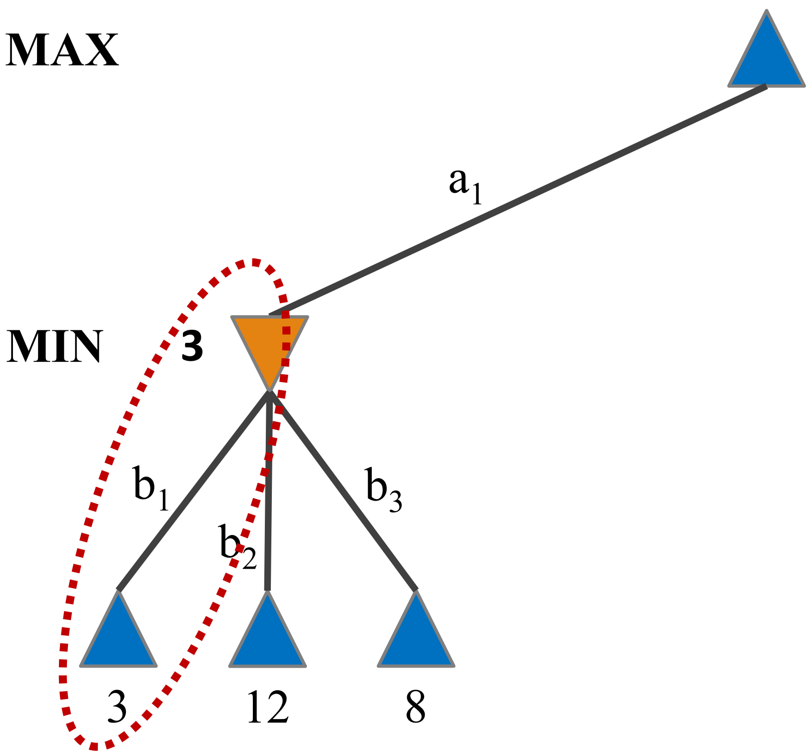
- E.g., 2-ply game



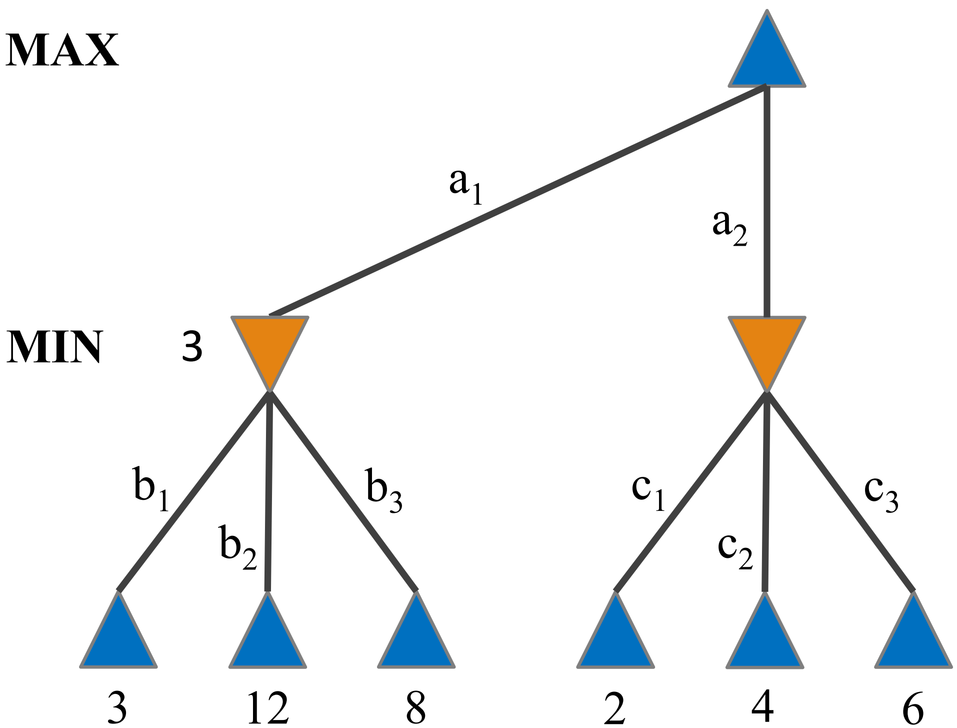
# Minimax Algorithm



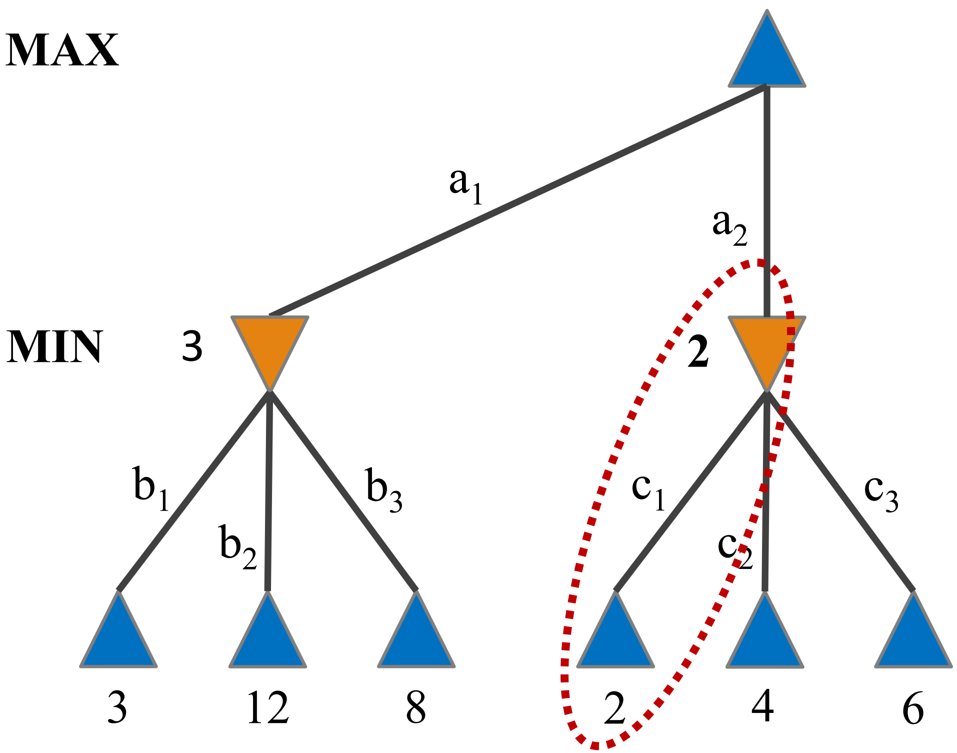
# Minimax Algorithm



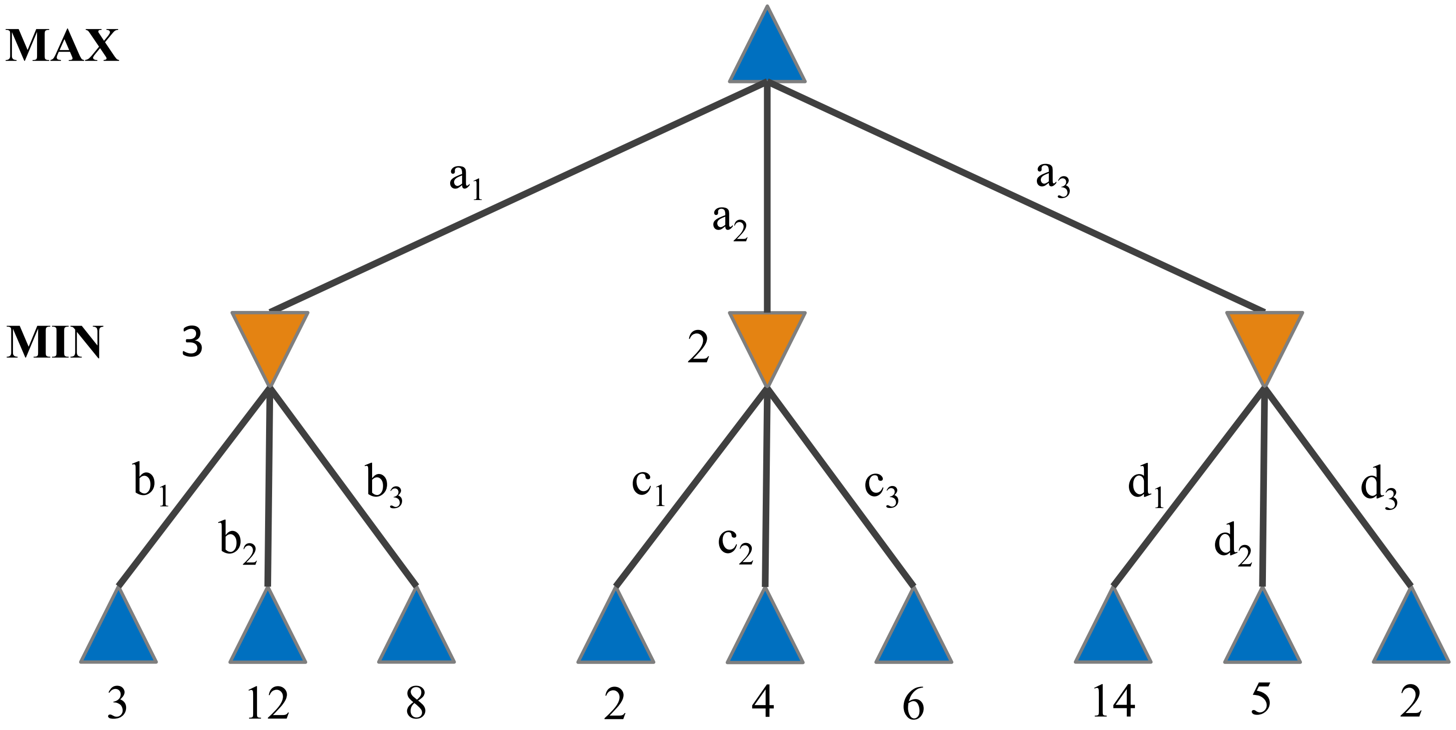
# Minimax Algorithm



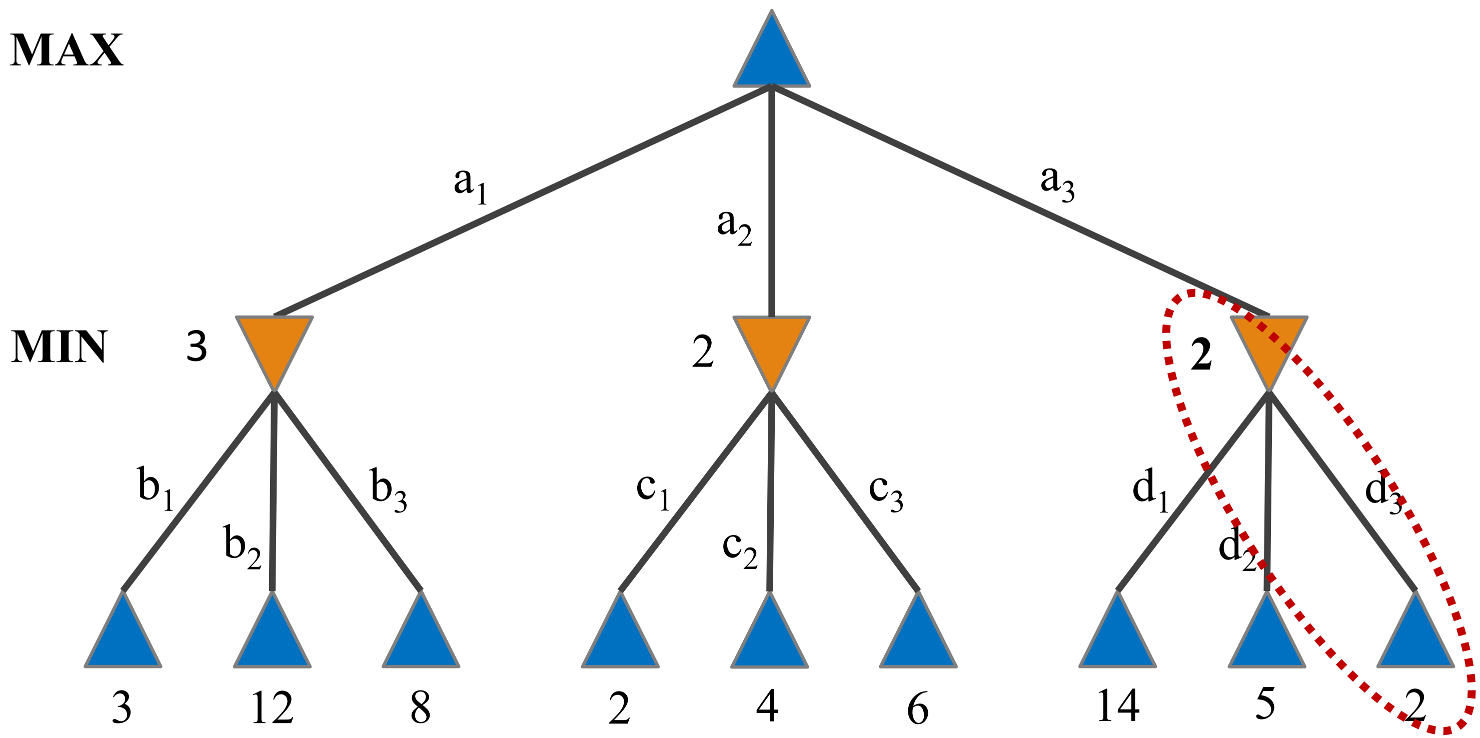
# Minimax Algorithm



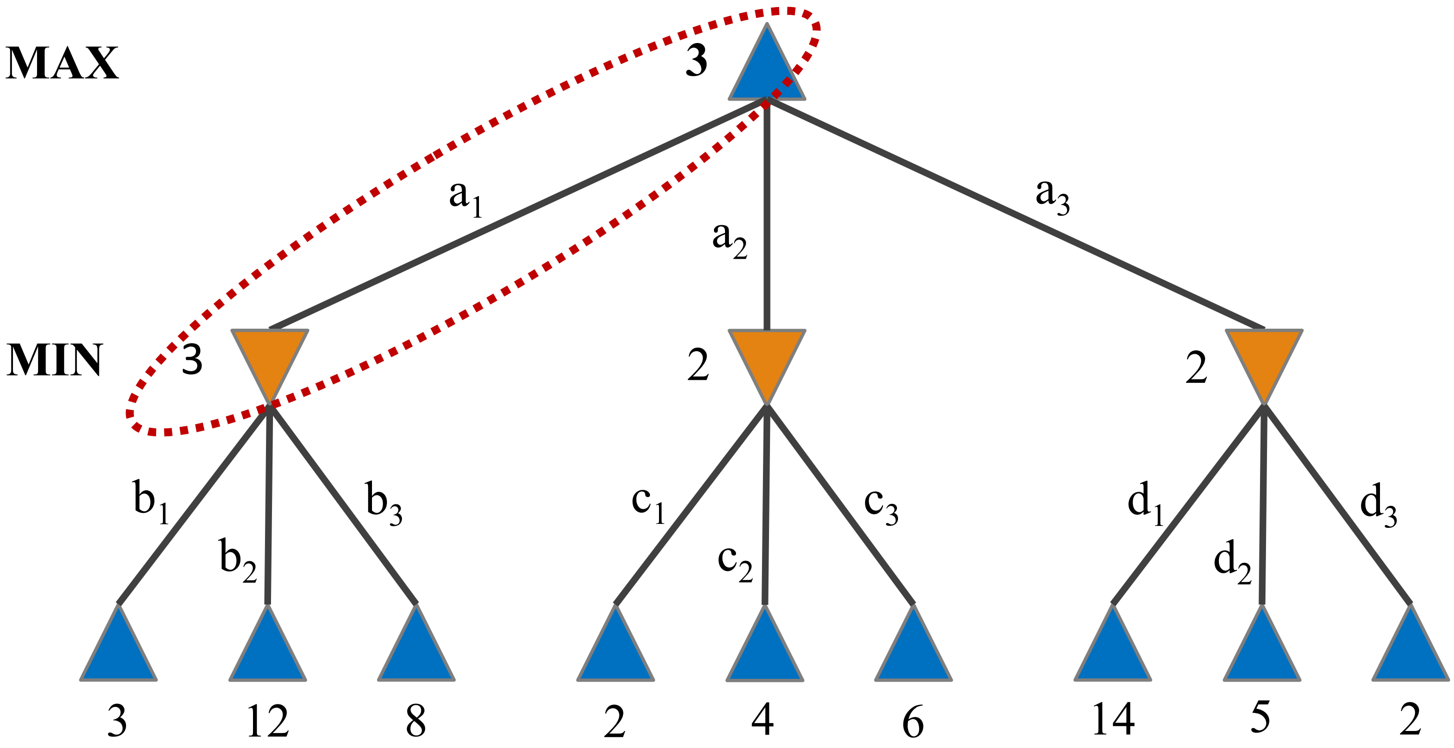
# Minimax Algorithm



# Minimax Algorithm

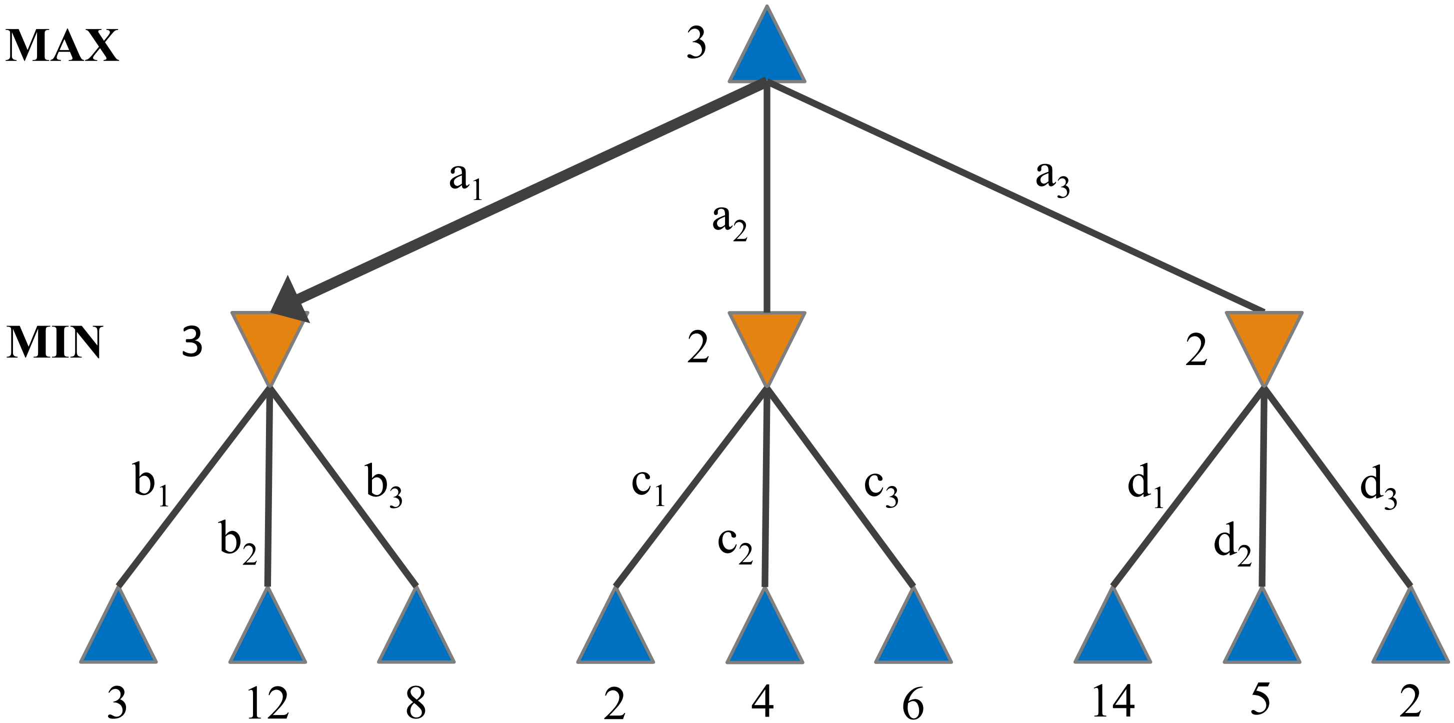


# Minimax Algorithm



# Minimax Algorithm

- E.g., 2-ply game



# Minimax Algorithm

**Operator = Action or Move**

```
function MINIMAX-DECISION(game) returns an operator

   for each op in OPERATORS[game] do
         VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
   end
   return the op with the highest VALUE[op]
```

```
function MINIMAX-VALUE(state, game) returns a utility value

   if TERMINAL-TEST[game](state) then
         return UTILITY[game](state)
   else if  MAX is to move in state then
         return the highest MINIMAX-VALUE of SUCCESSORS(state)
   else
         return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

# Properties of Minimax

- Completeness: Yes, if tree is finite
- Optimality: Yes, against an optimal opponent.
- Time complexity: ?
- Space complexity: ?

# Properties of Minimax

- Completeness: Yes, if tree is finite
- Optimality: Yes, against an optimal opponent.
- Time complexity: $O(b^m)$
- Space complexity: $O(bm)$ (depth-first exploration)