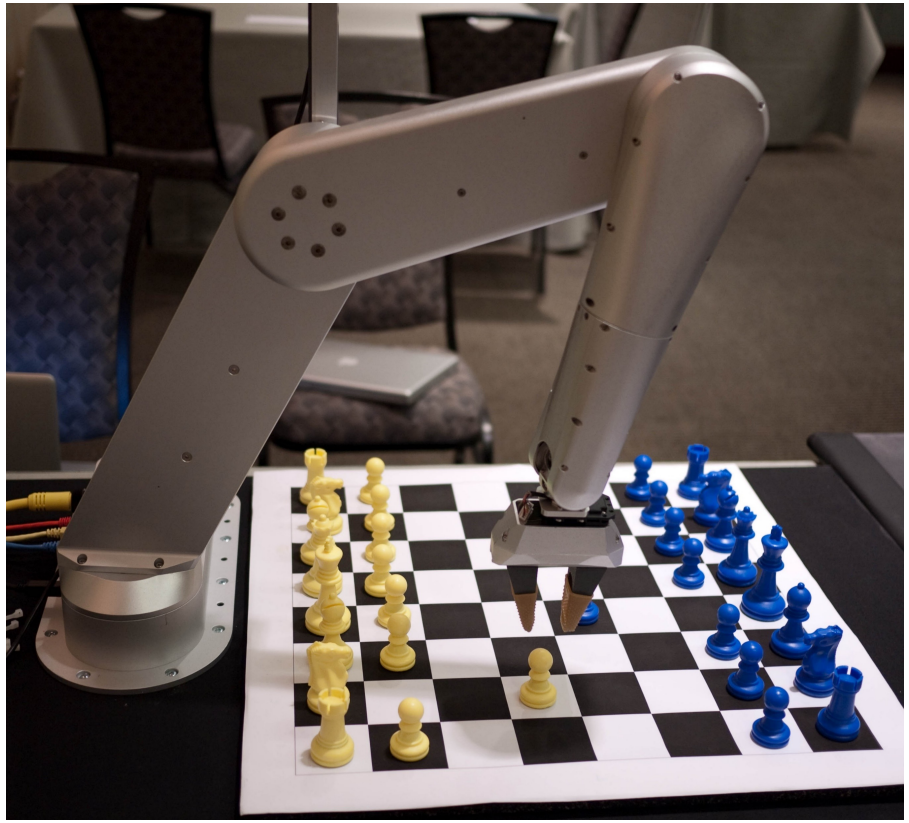
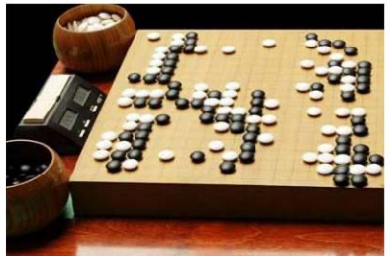
COMP 341 Intro to AI
Adversarial Search



Asst. Prof. Barış Akgün

Koç University

History



- 1770: Mechanical Turk
- 1912: A **paper** that first mentions an algorithm like Minimax
- 1944: Game Theory
- 1950s: First Checkers program, First chess program, Ideas of ML
- 1956: Pruning to allow deeper search with efficiency
- 1992: TD-Gammon – Self-learning backgammon AI
- 1994: Human checkers champion is beaten
- 1997: Deep Blue beats Kasparov with great evaluation functions
  - Now: Grandmaster level play on a ~~desktop~~ smartphone
- 2007: Checkers completely solved
- 2016: AlphaGo receives an honorary 9-dan
  - Monte Carlo Tree search and Machine Learning
- 2019: OpenAI Five beats world champion Dota2 Team
- 2019: DeepMind's Starcraft 2 Agent better than 99.8% of human players

Game Playing vs Search

- "*Unpredictable*" opponent: Solution is a **strategy**
  - Specifying a move for every possible opponent reply
- Time limits – Unlikely to find the *best* move, must **approximate**

- This section of the course:
  - Prune states for deeper search
  - Approximate evaluation of how promising a state is

- Not in this course but if interested see:
  - Machine learning to improve evaluation accuracy (mentioned in the RL part)
  - Sample-based search to deal with high branching factors

Types of Games

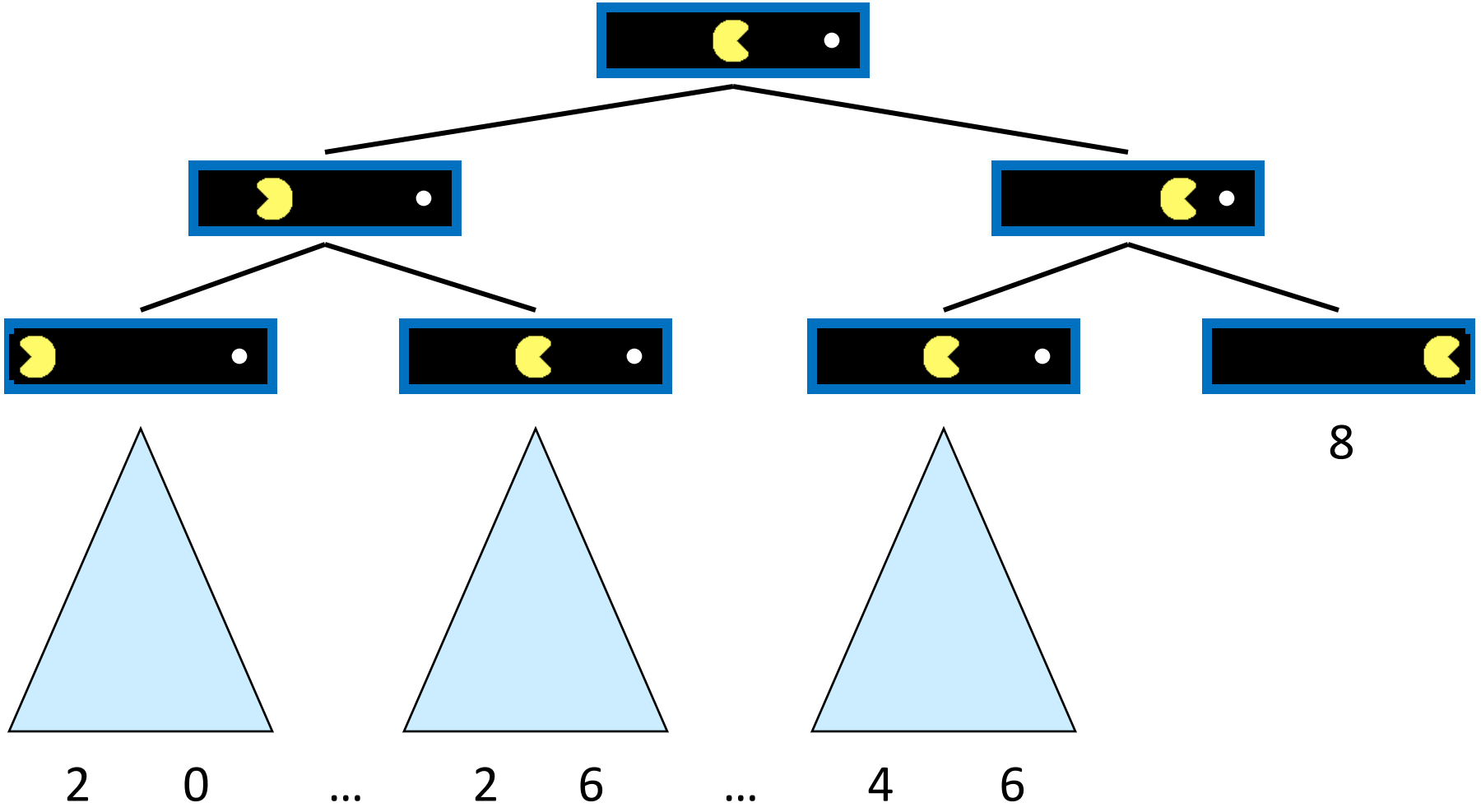
	deterministic	stochastic
perfect information	chess, checkers, go, othello	backgammon, monopoly
imperfect information	battleship, blind tictactoe, kriegspiel	bridge, poker, scrabble, nuclear war

- Turn taking, 2-player, zero-sum games
- Zero-sum: utility of one agent is exact opposite of the other, e.g. I win (+1) you lose (-1) or we draw (both 0)
  - Simplification: A single value that one player maximizes and the other minimizes
- Ideas could be expanded to multiplayer competitive games
- General games
  - Agents have independent utilities
  - Cooperation, indifference, competition, and more are all possible

Problem Formulation (Deterministic Games)

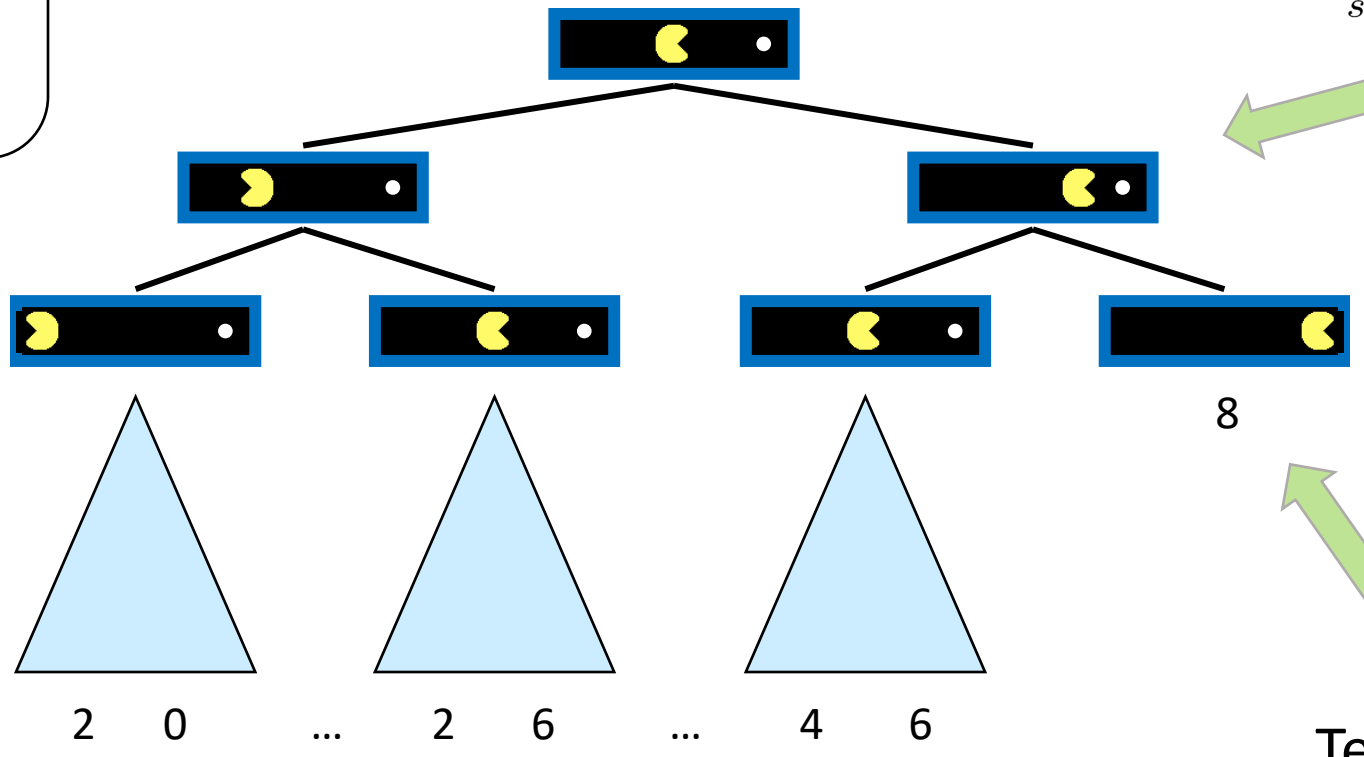
- A state space: S
- Initial state: s_0, where the game starts
- Players: P = {1,…,N}, that take turns (we'll focus on N=2)
- Actions: A, plays/moves players can make, may depend on player and state
- Transition Model/Successor Function: (S x A)→ S
- Terminal Test: g(s) → {true, false}, is the game over?
- Terminal Utilities (scalar): (s x p) → c, value of terminal state s for player p

Single-Agent Trees



Value of a State

Value of a state V(s):
The best achievable outcome (utility) from that state



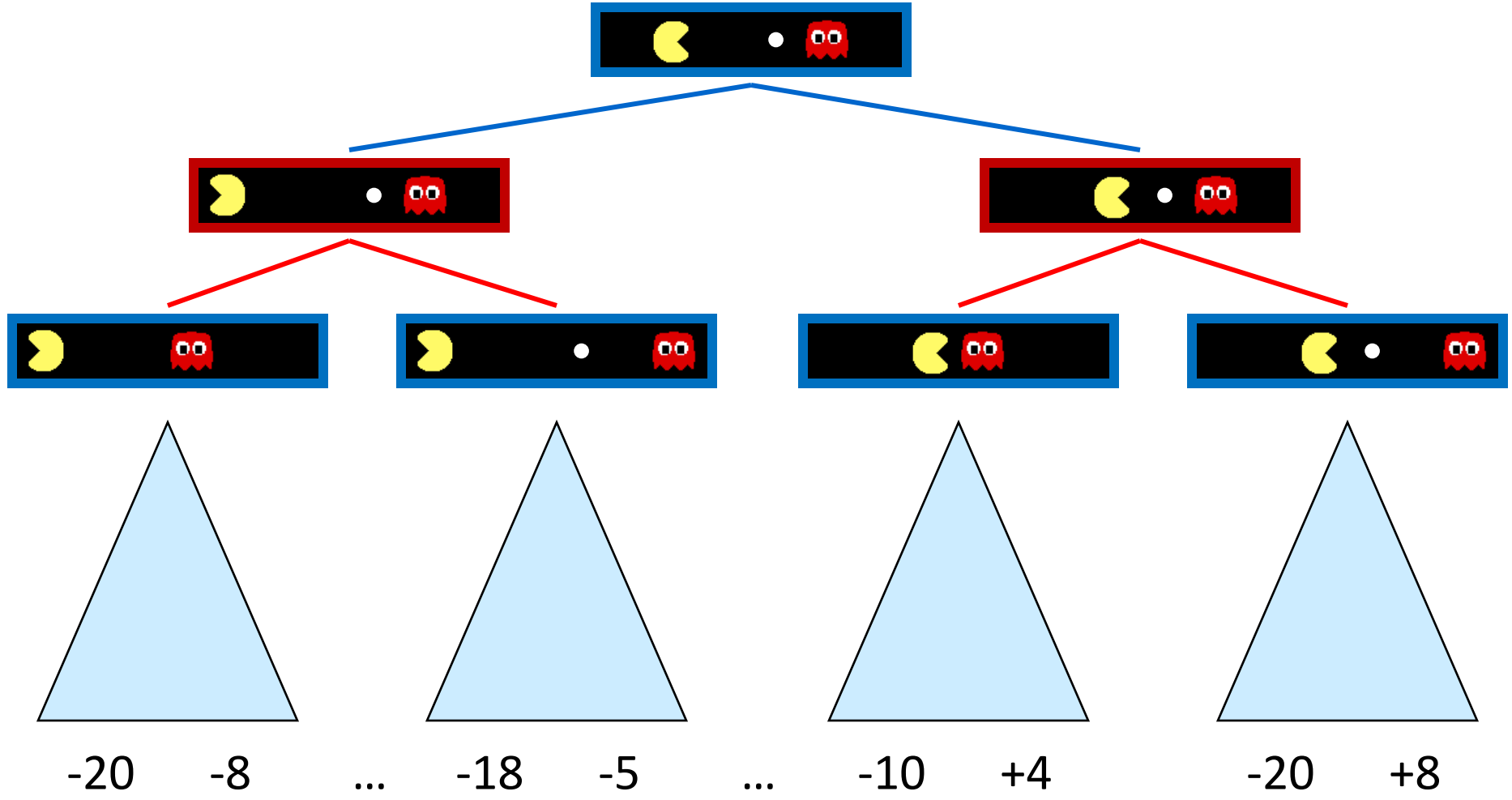
Non-Terminal States:

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

Terminal States:

$$V(s) = \text{known}$$

Adversarial Game Trees



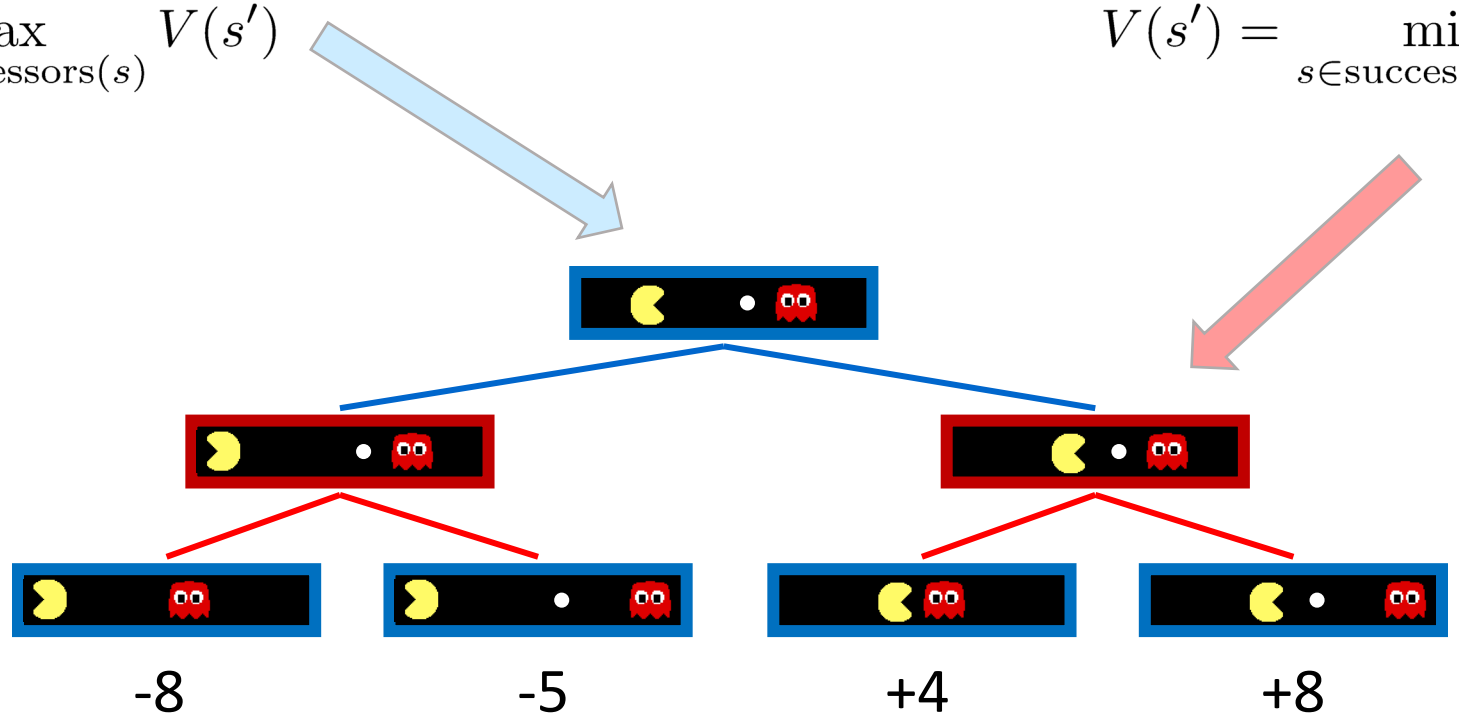
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

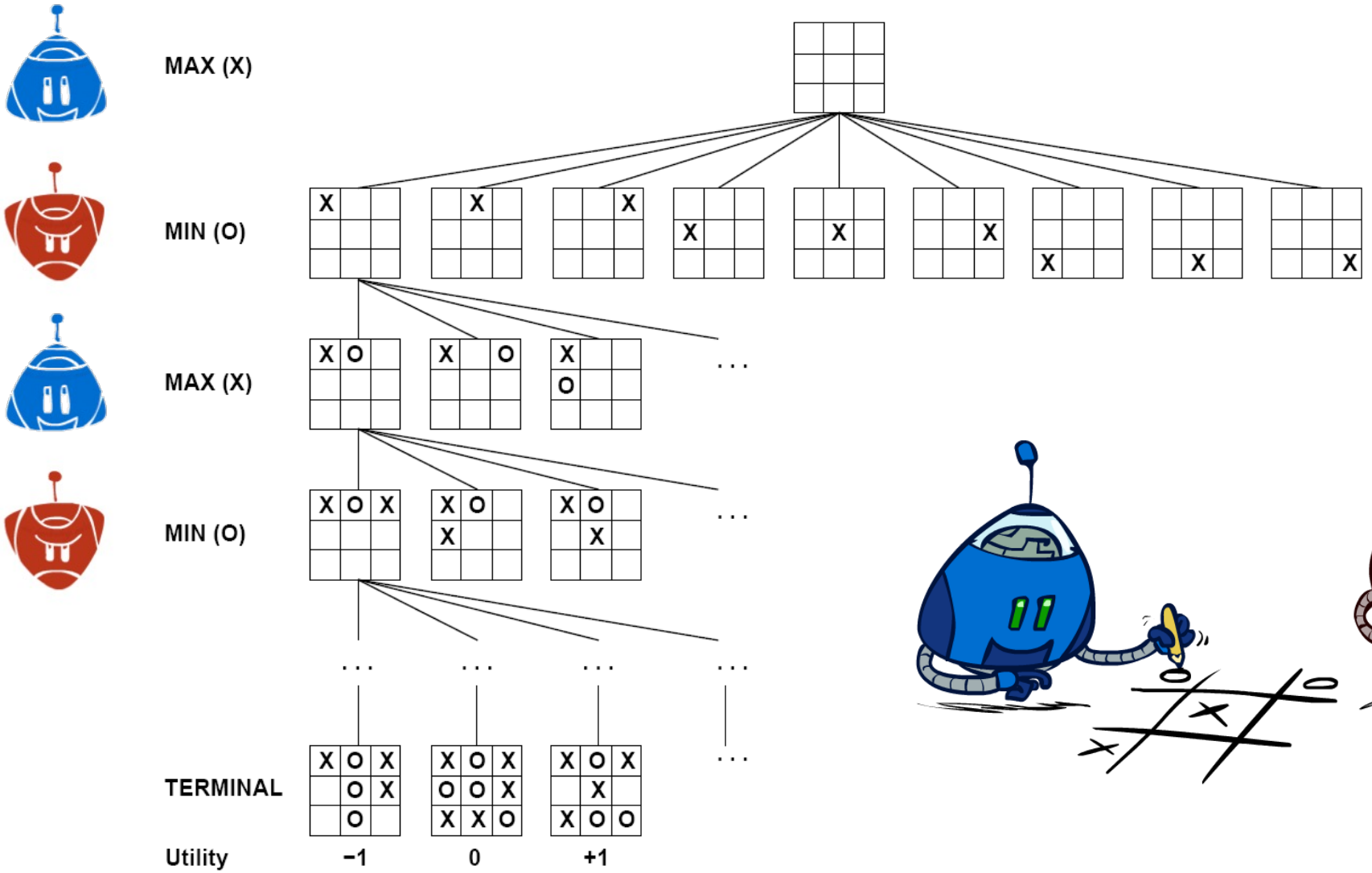
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

$$V(s) = \text{known}$$

Tic-Tac-Toe Game Tree



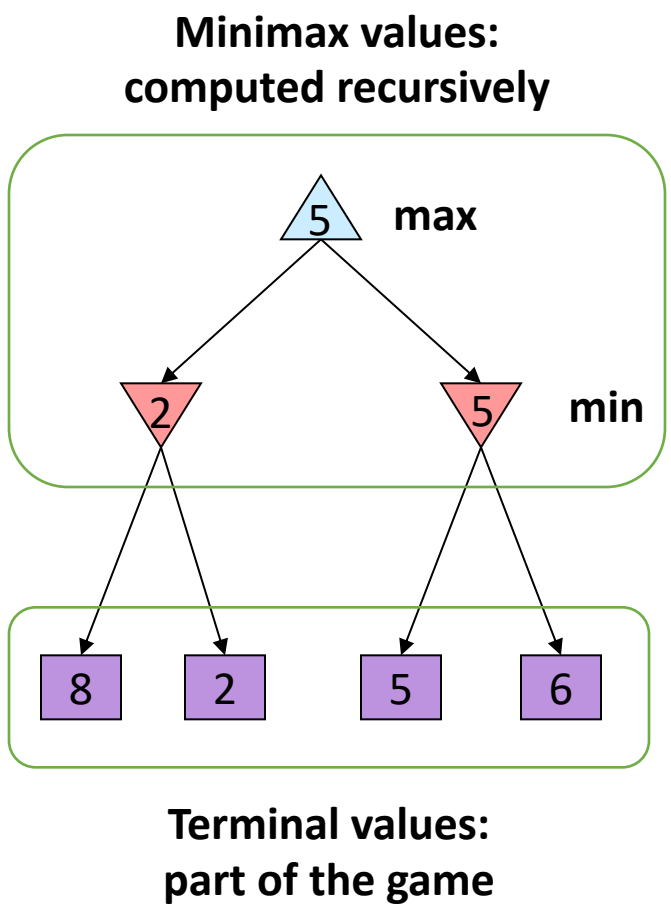
All possible actions that MAX can take
Now it's Player 2's turn

We get the utilities at the end of the game for each player!

Adversarial Search (Minimax)

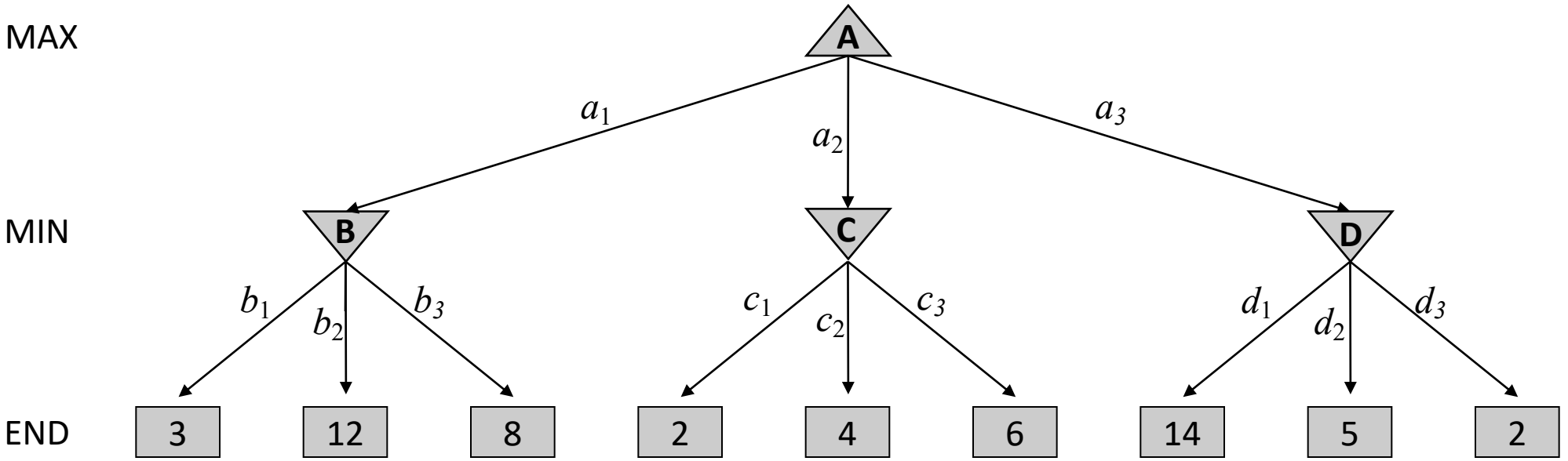
- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- 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



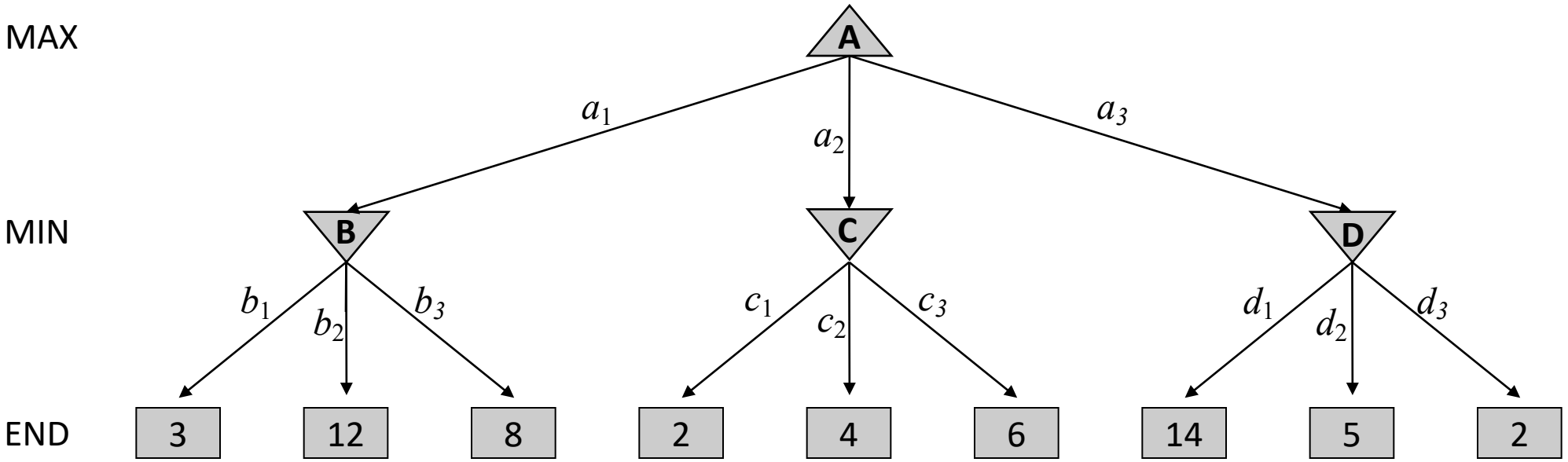
Simple Example

Initial state: **A**
Max has 3 actions
Min has 3 actions at each resulting state
Then the game ends



Simple Example

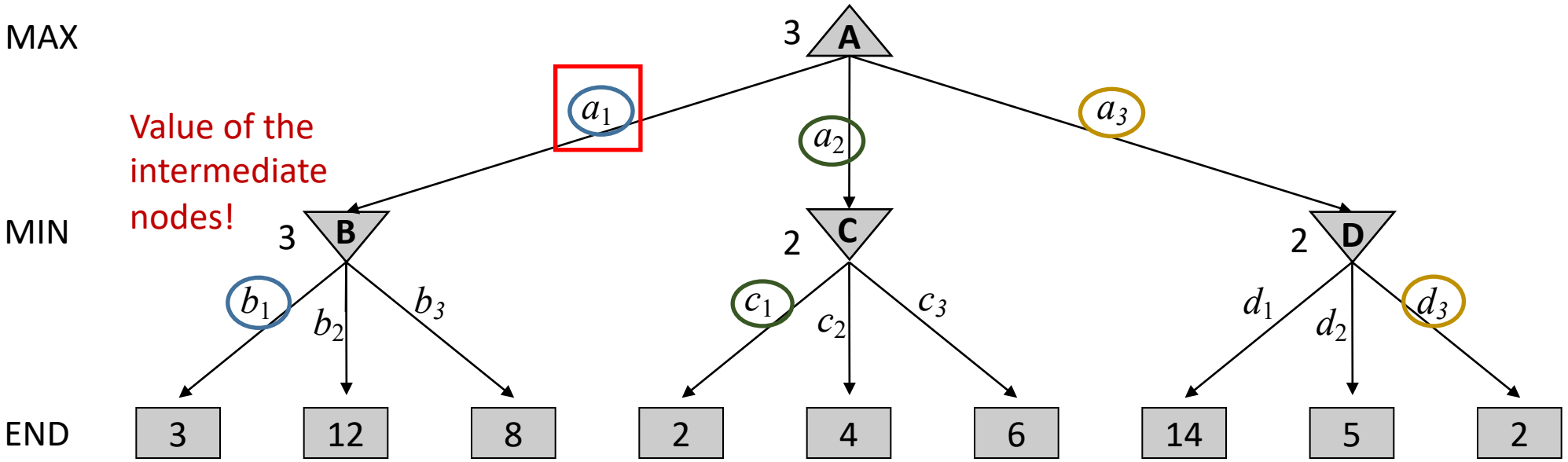
What is the best move for **A**?
How do we chose it, assuming
our opponent will play optimally?



Zero-sum game:
Worst outcome for you → best outcome for your opponent

Simple Example

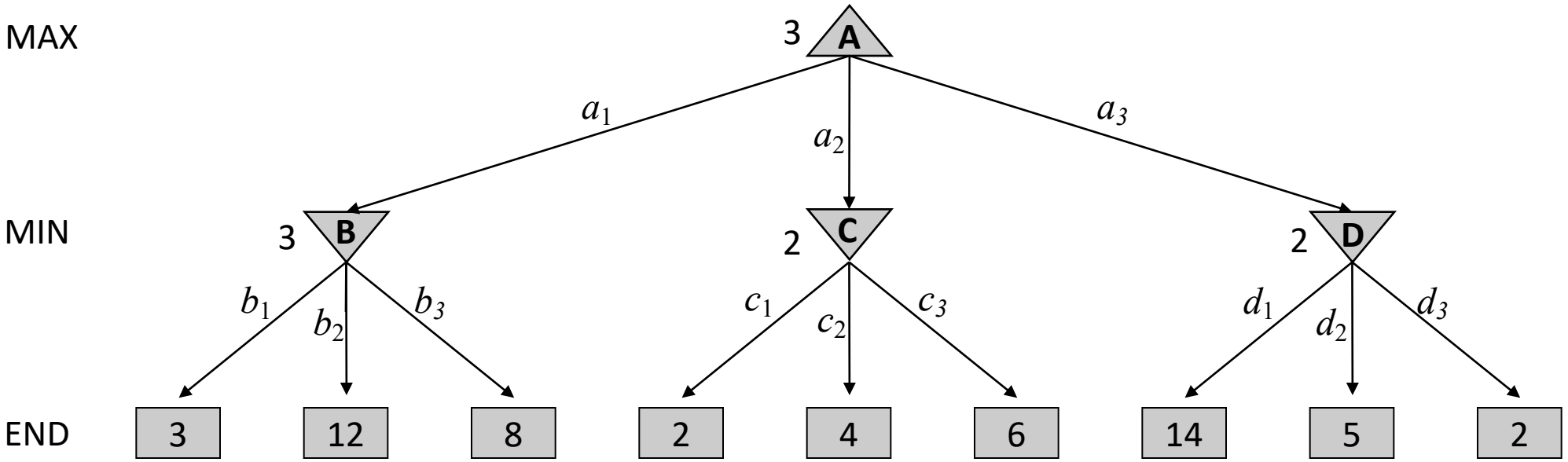
Optimal move for MAX, assuming MIN will play optimally?



These values are called the minimax values

Simple Example

Note that MIN has not actually played, we have just forward simulated what it would be like and chose a_1 based on our search



Minimax Implementation

States Under Agent's Control:

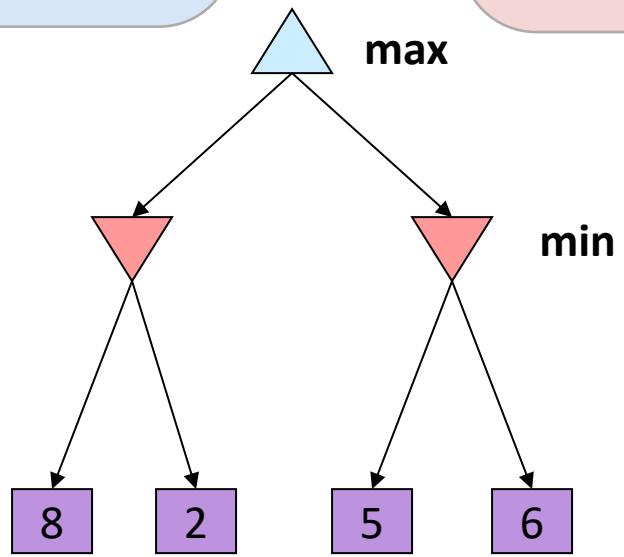
$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

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

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

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



Minimax Implementation

```
def get-best-action(state):
    return  argmax (value(S(state, a))
                    a
```

$a \in Actions(state)$
S$(state, a)$: transition function

```
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)
```

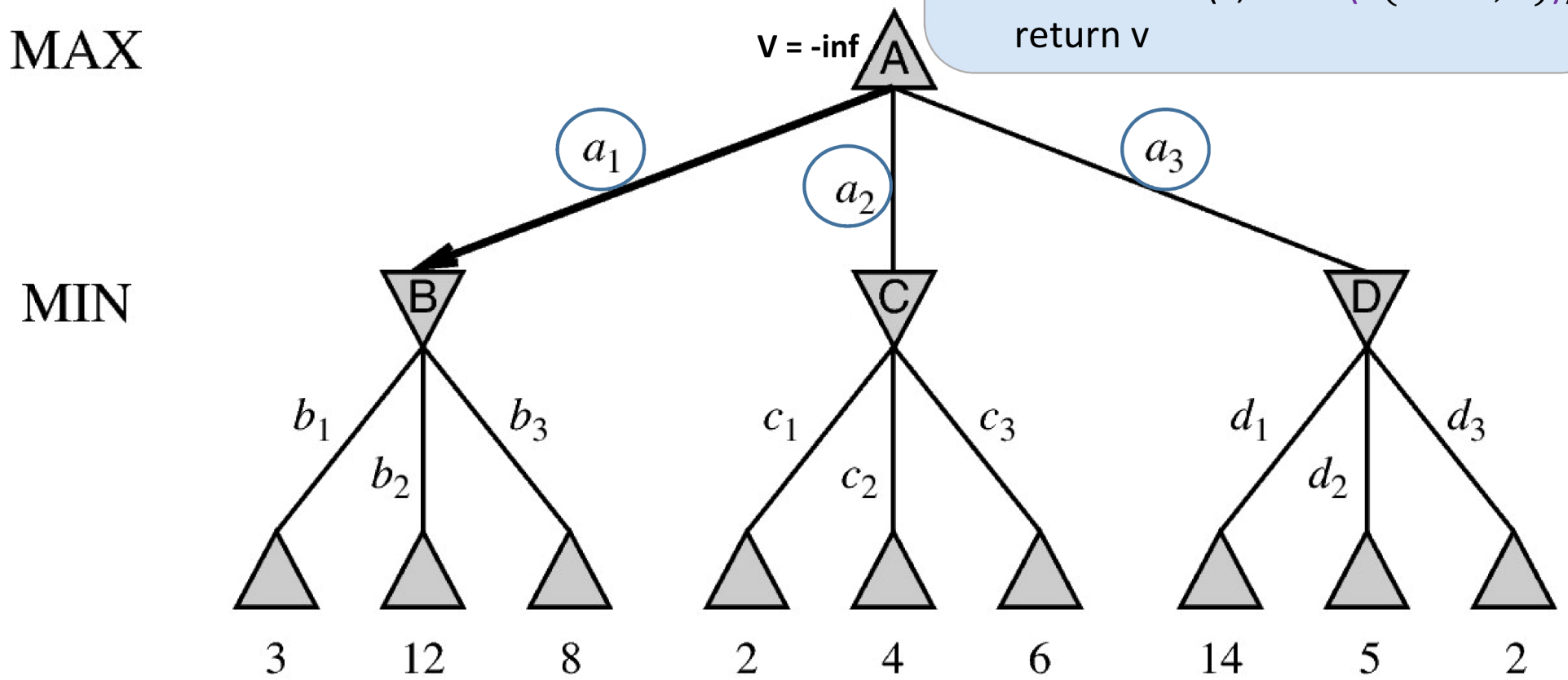
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

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

```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

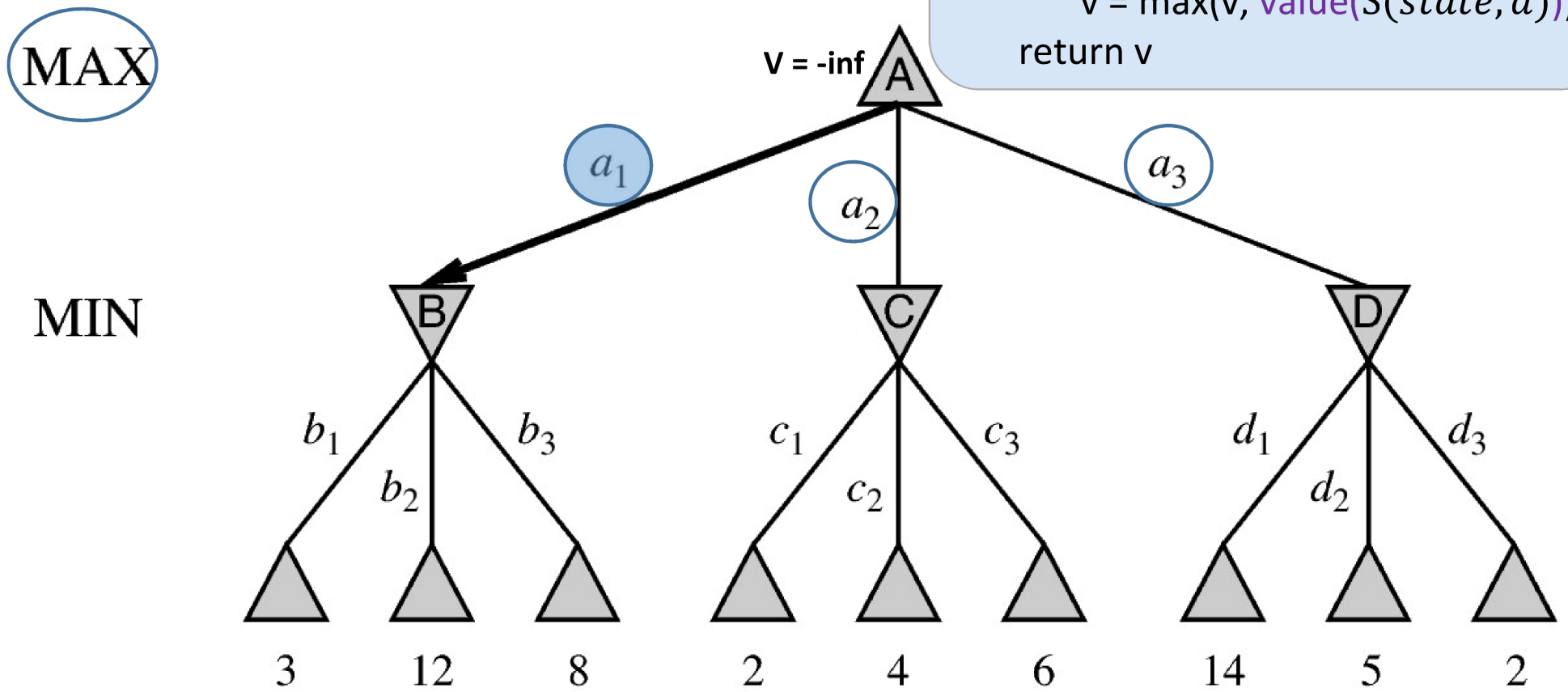
```
def max-value(state):
    initialize v = -∞
    for each action a of state:
        v = max(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

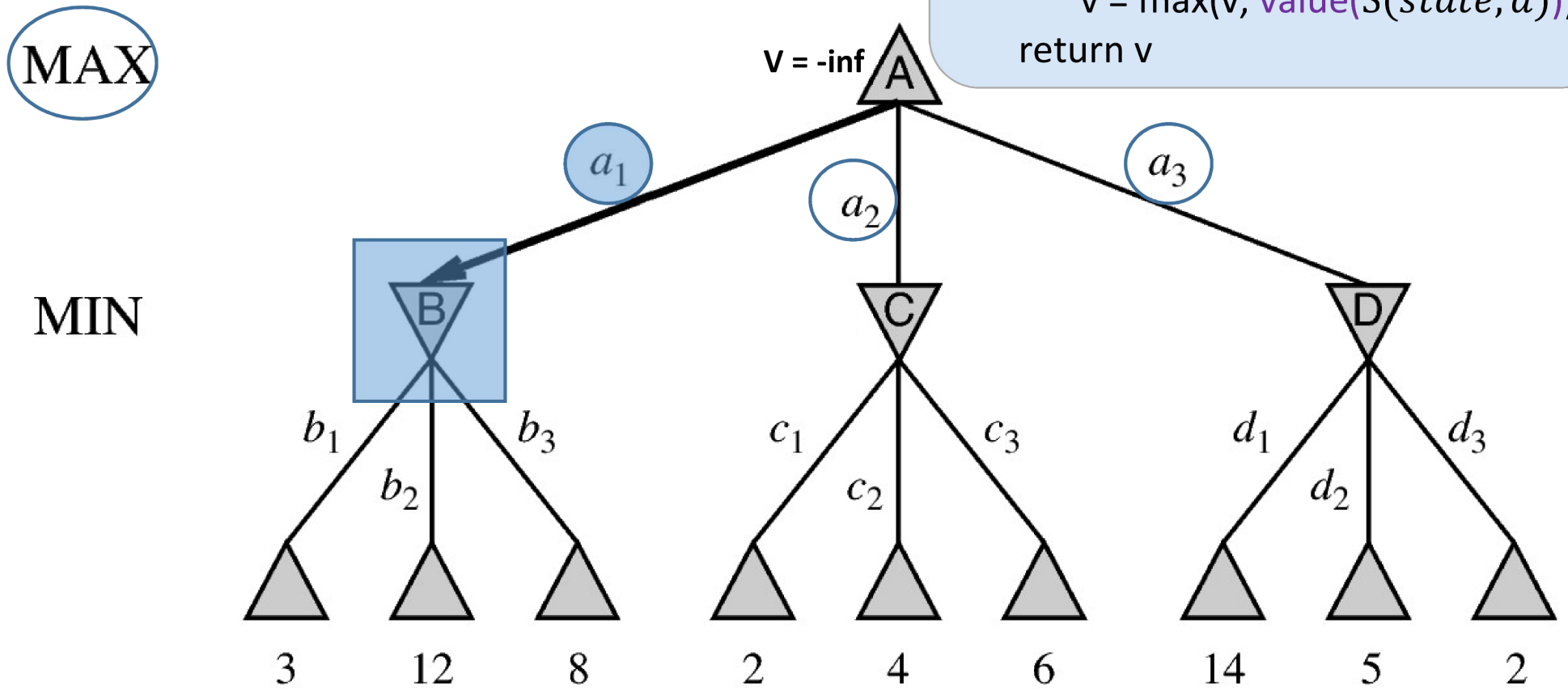
```
def max-value(state):
    initialize v = -∞
    for each action a of state:
        v = max(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

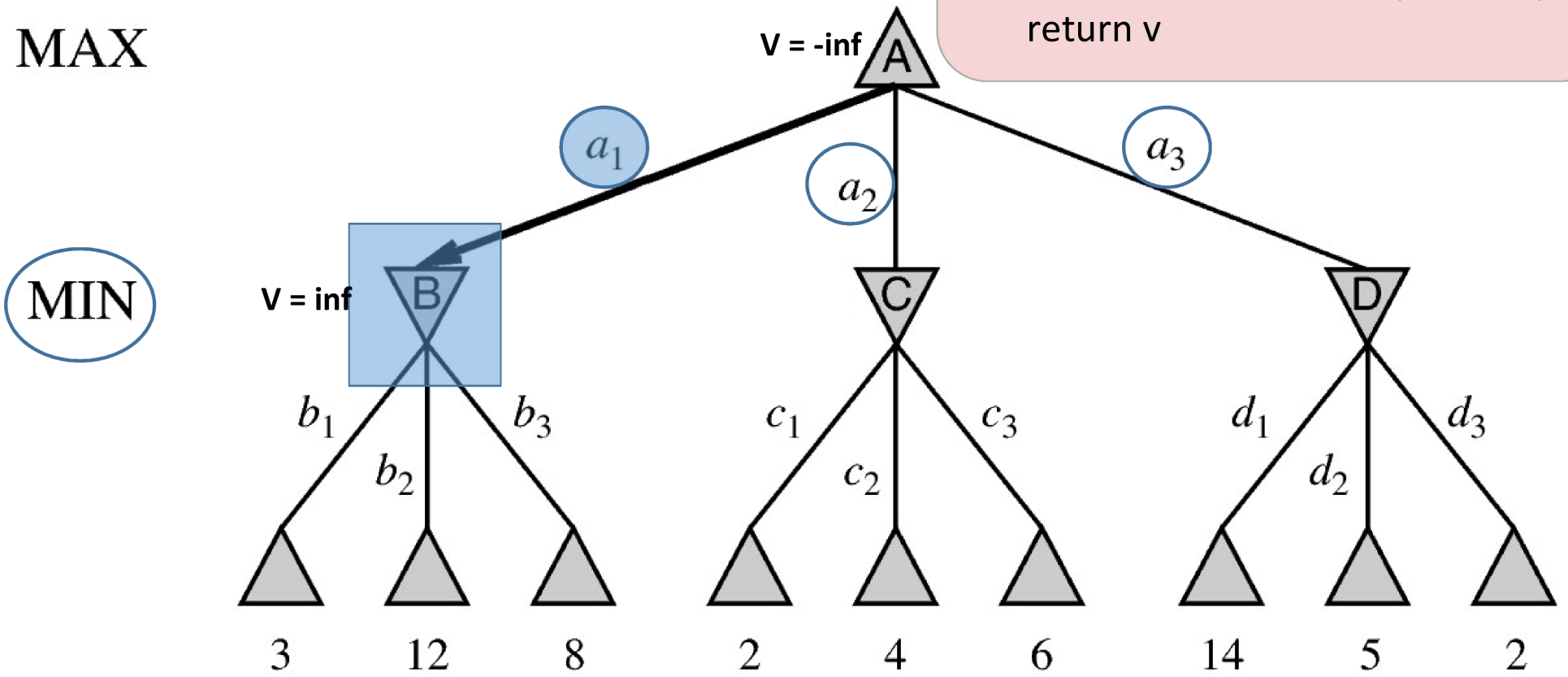
```
def max-value(state):
    initialize v = -∞
    for each action a of state:
        v = max(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

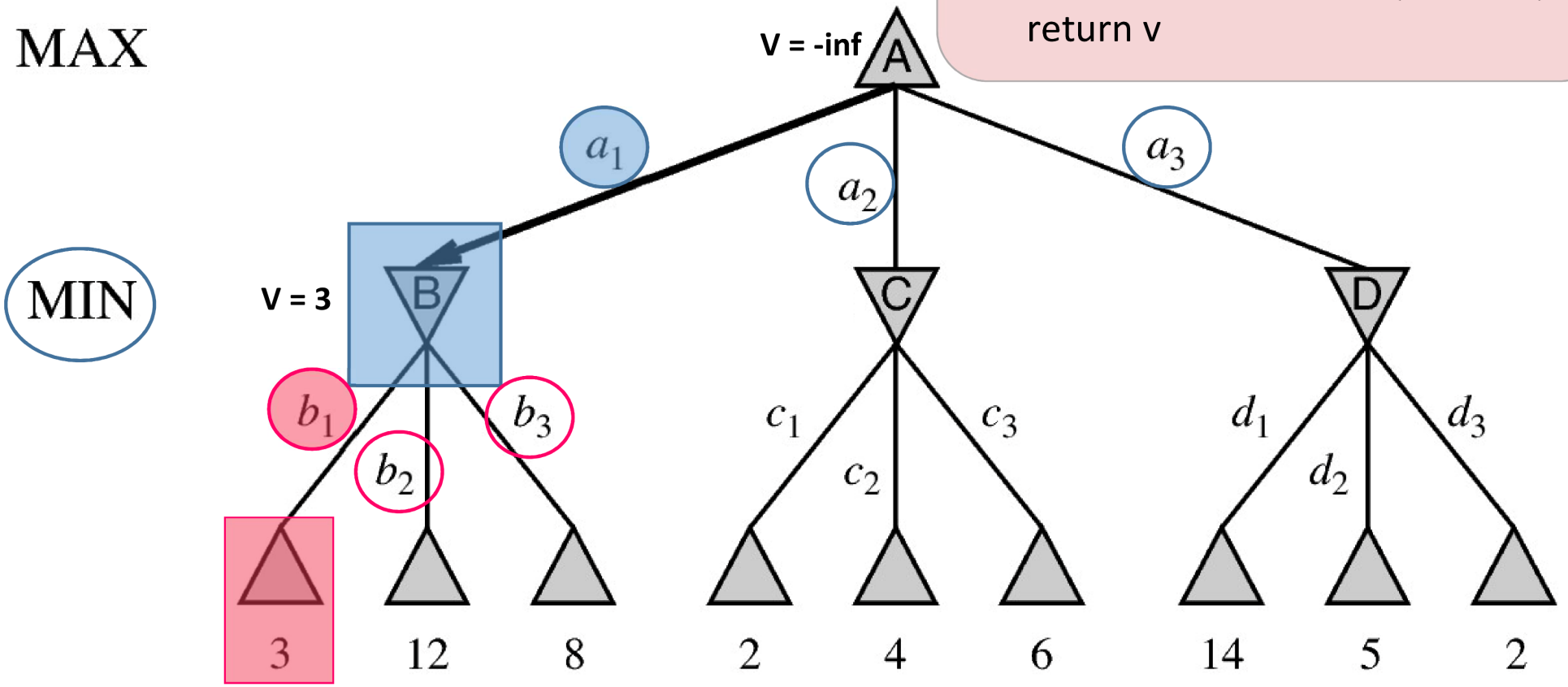
```
def min-value(state):
    initialize v = +∞
    for each action a of state:
        v = min(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

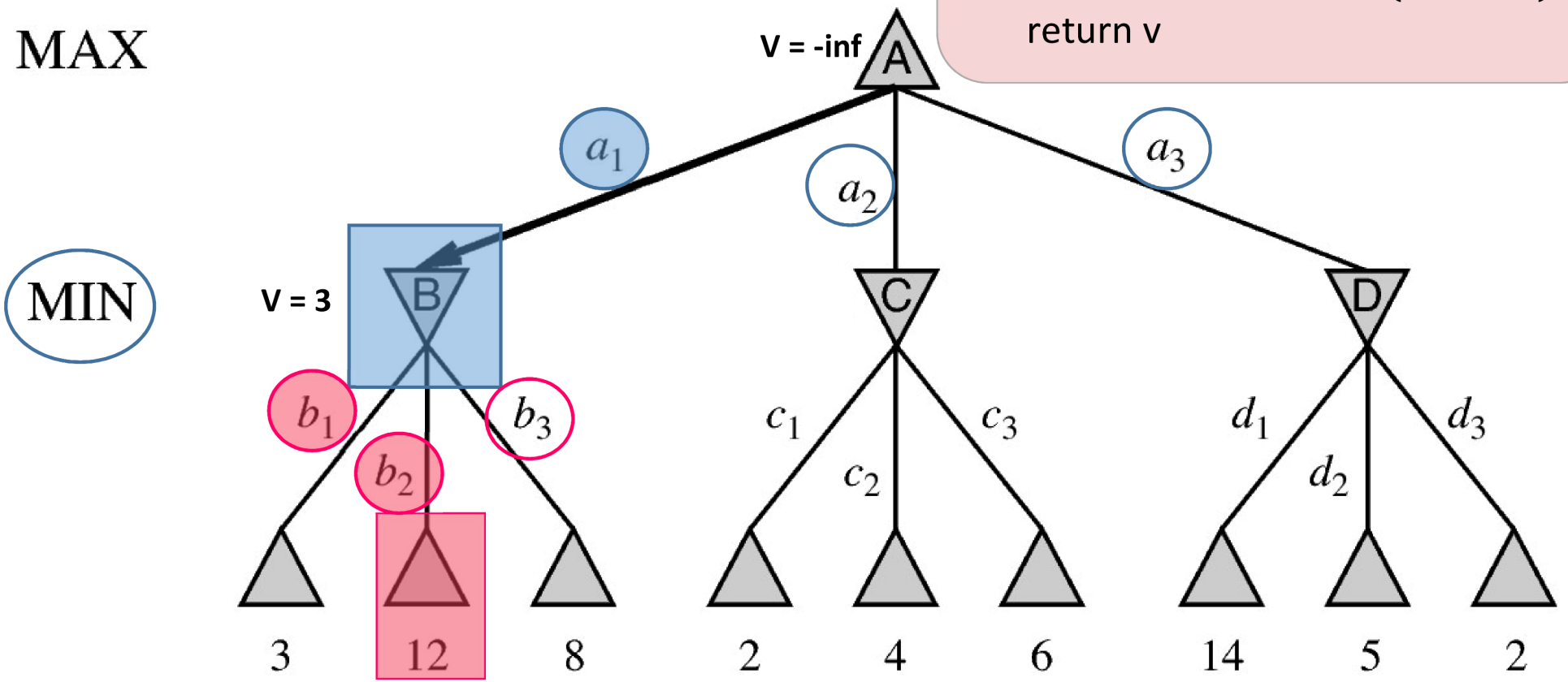
```
def min-value(state):
    initialize v = +∞
    for each action a of state:
        v = min(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

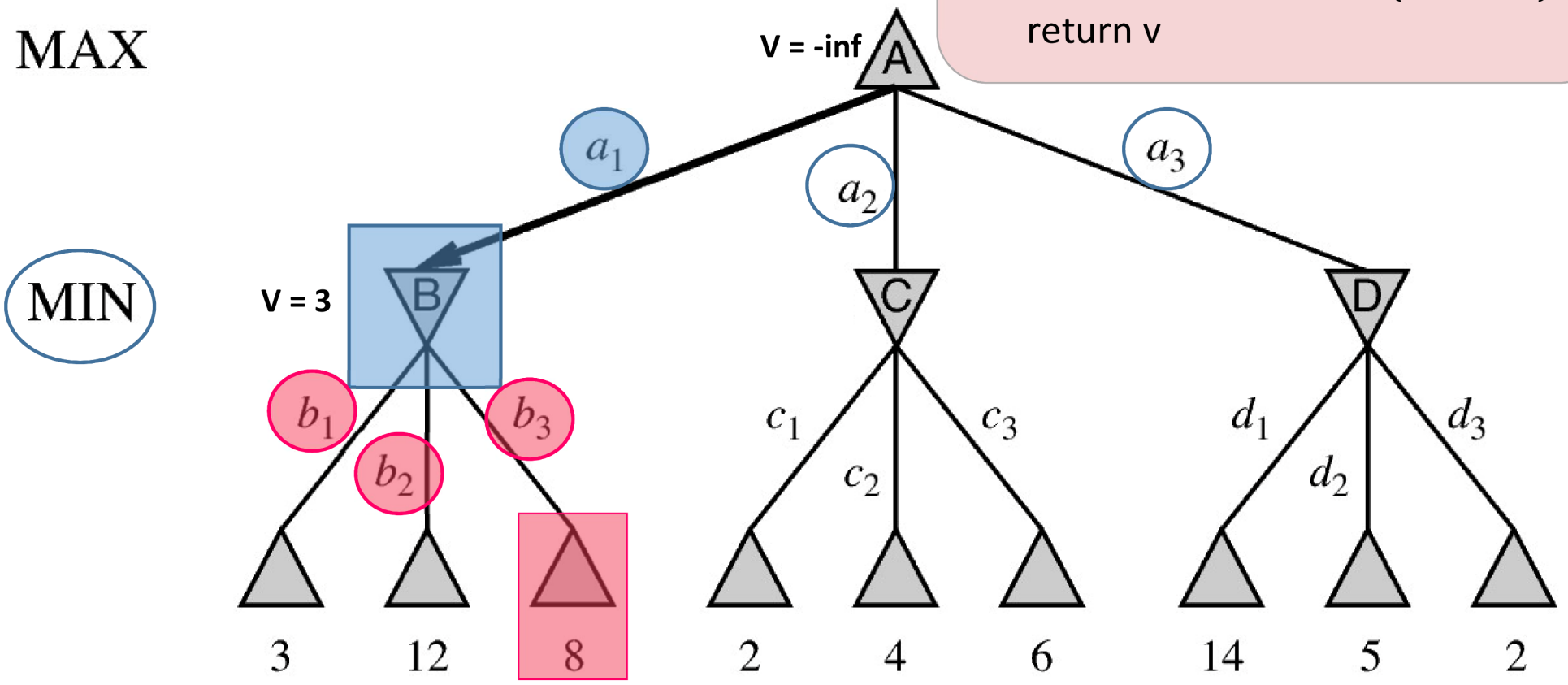
```
def min-value(state):
    initialize v = +∞
    for each action a of state:
        v = min(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

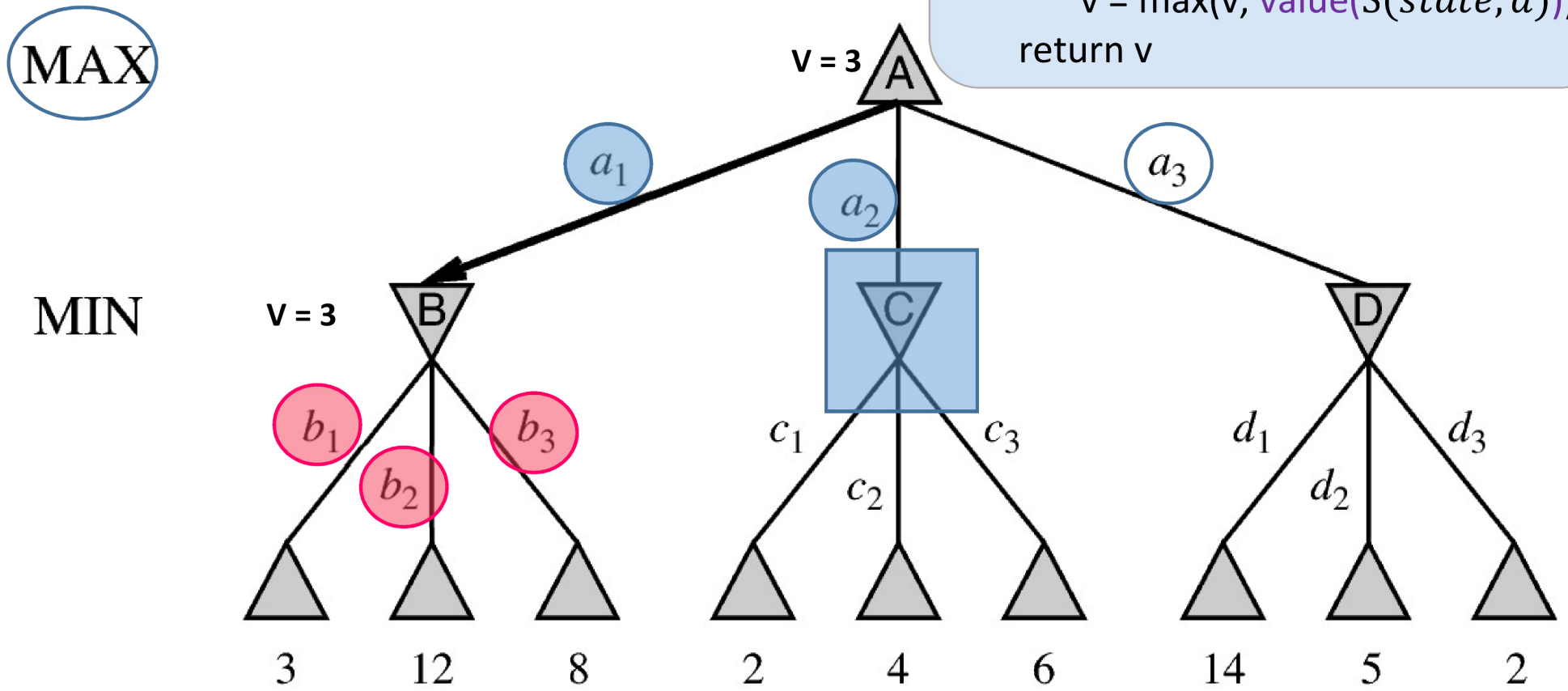
```
def min-value(state):
    initialize v = +∞
    for each action a of state:
        v = min(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

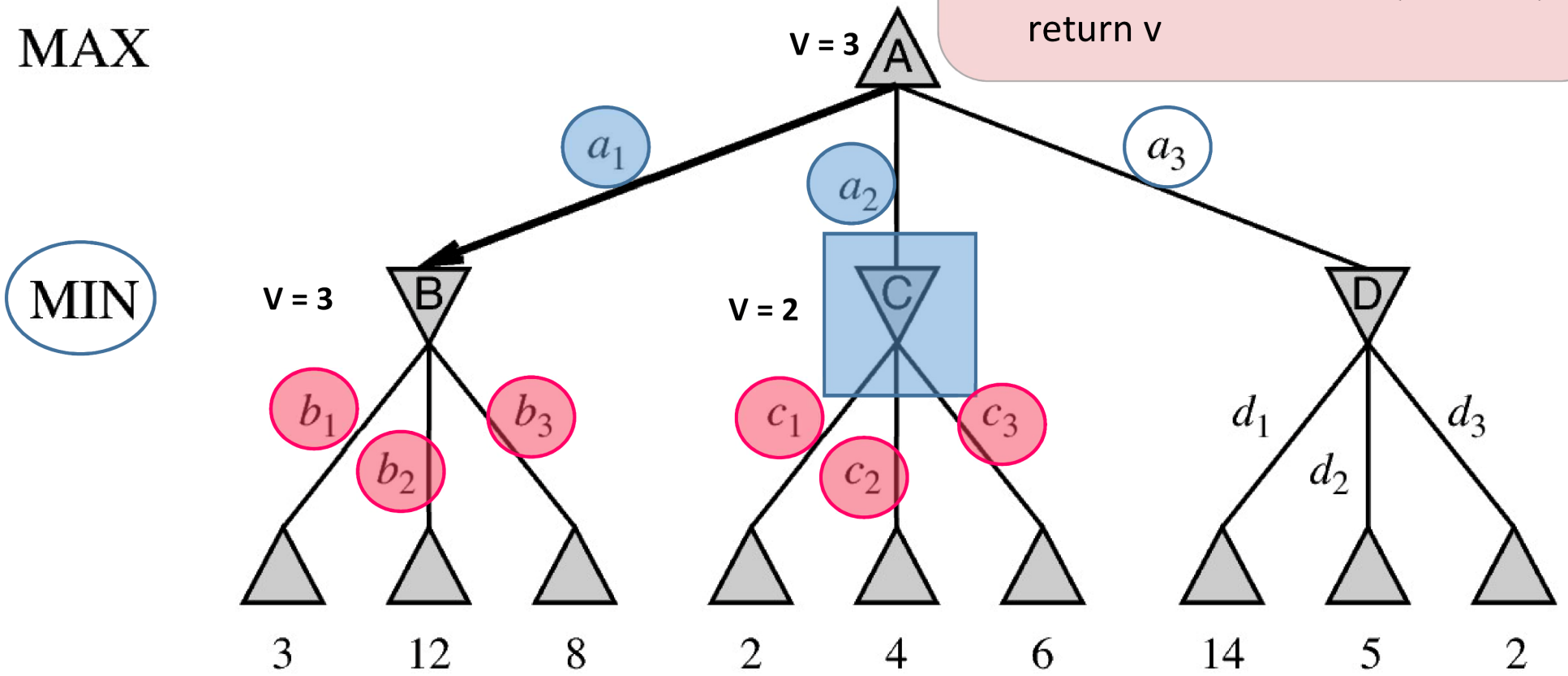
```
def max-value(state):
    initialize v = -∞
    for each action a of state:
        v = max(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

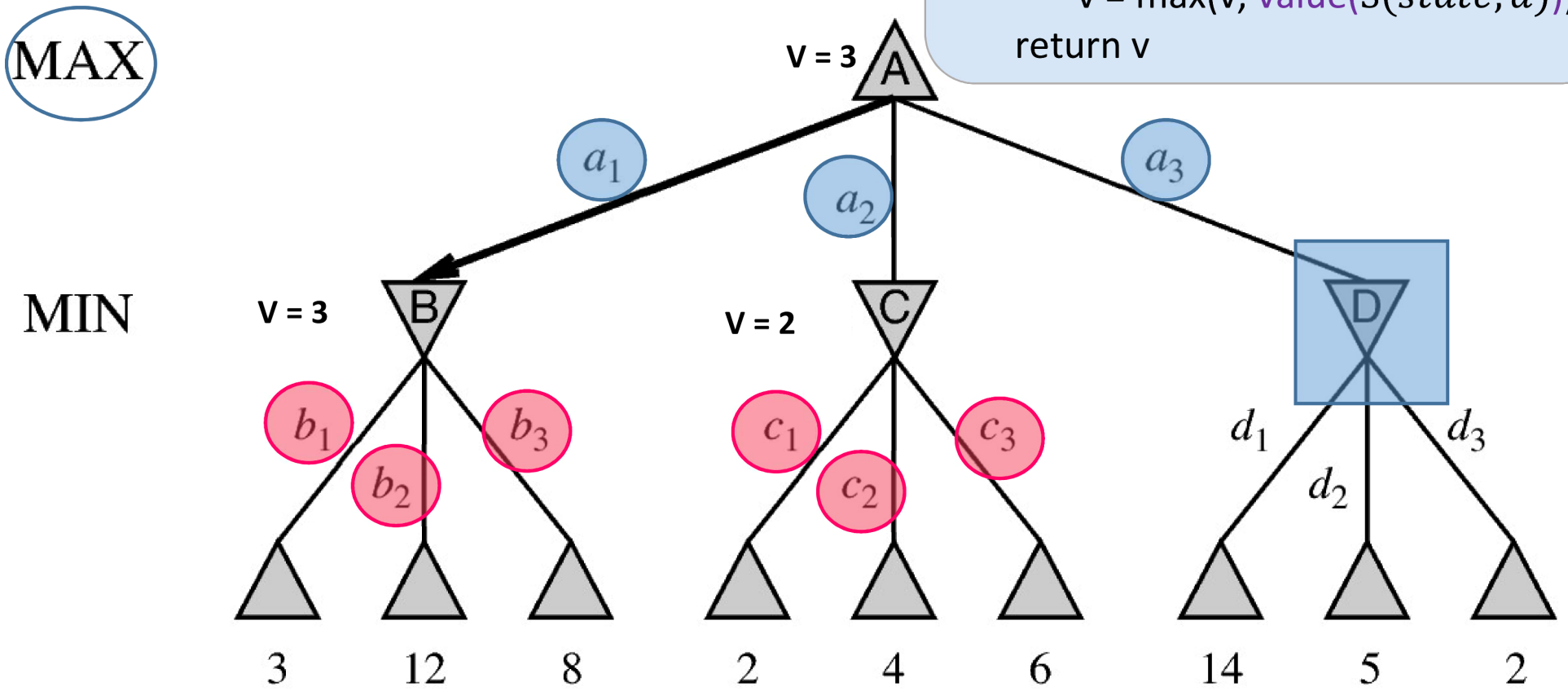
```
def min-value(state):
    initialize v = +∞
    for each action a of state:
        v = min(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

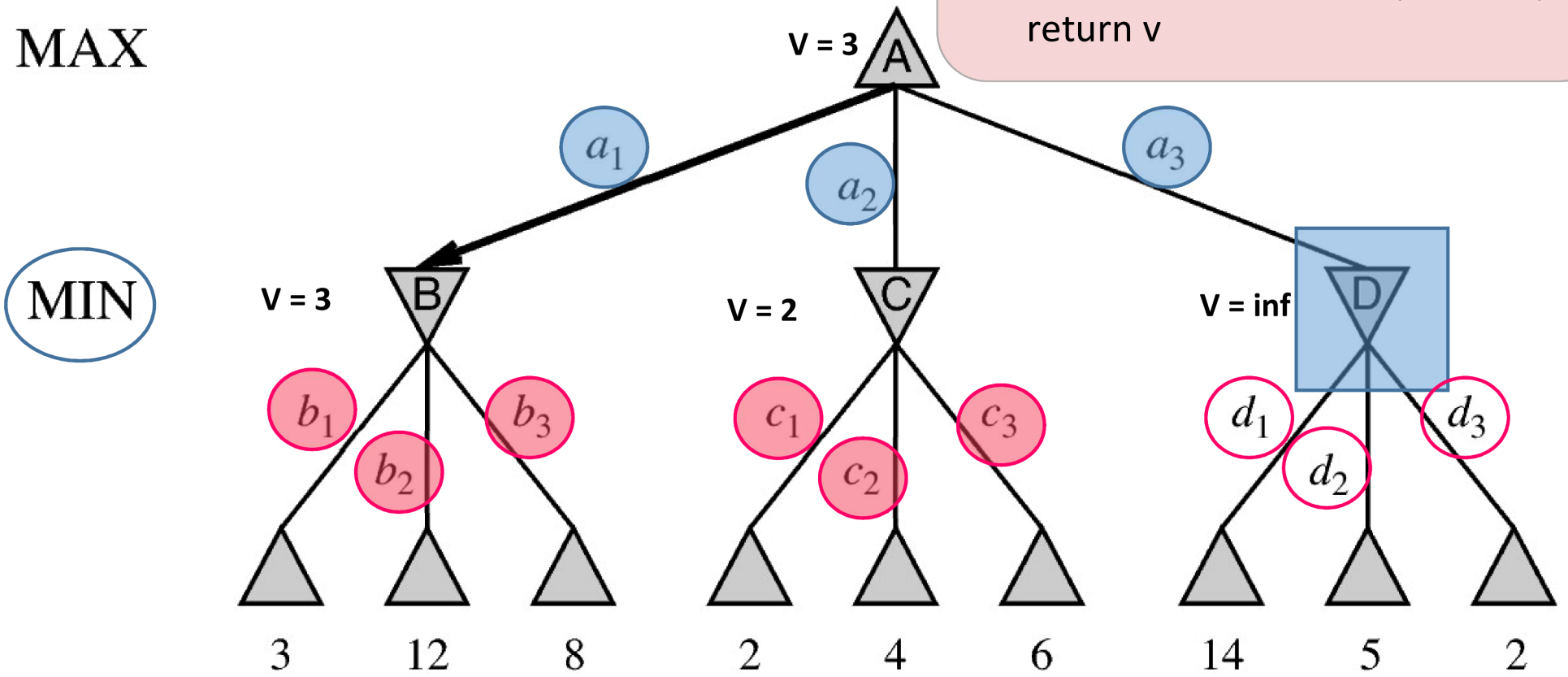
```
def max-value(state):
    initialize v = -∞
    for each action a of state:
        v = max(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

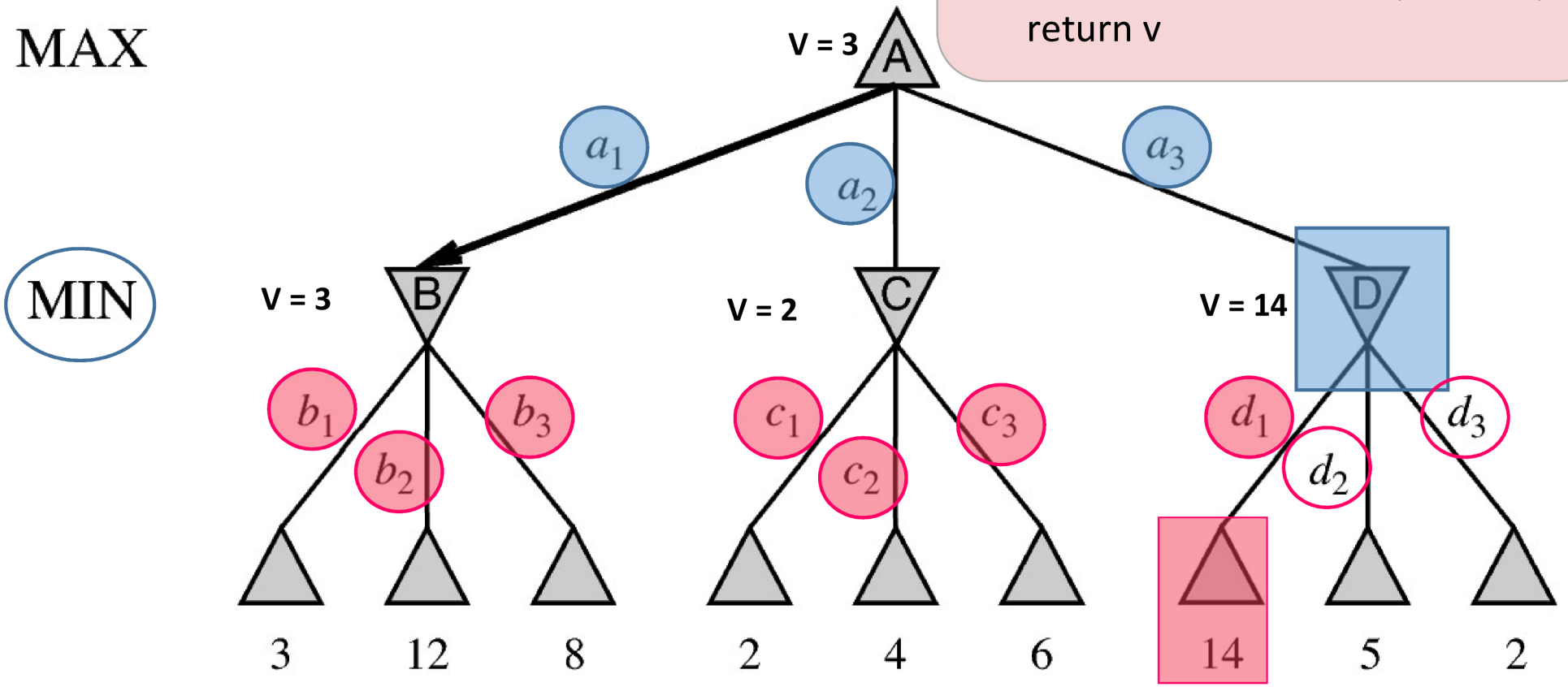
```
def min-value(state):
    initialize v = +∞
    for each action a of state:
        v = min(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

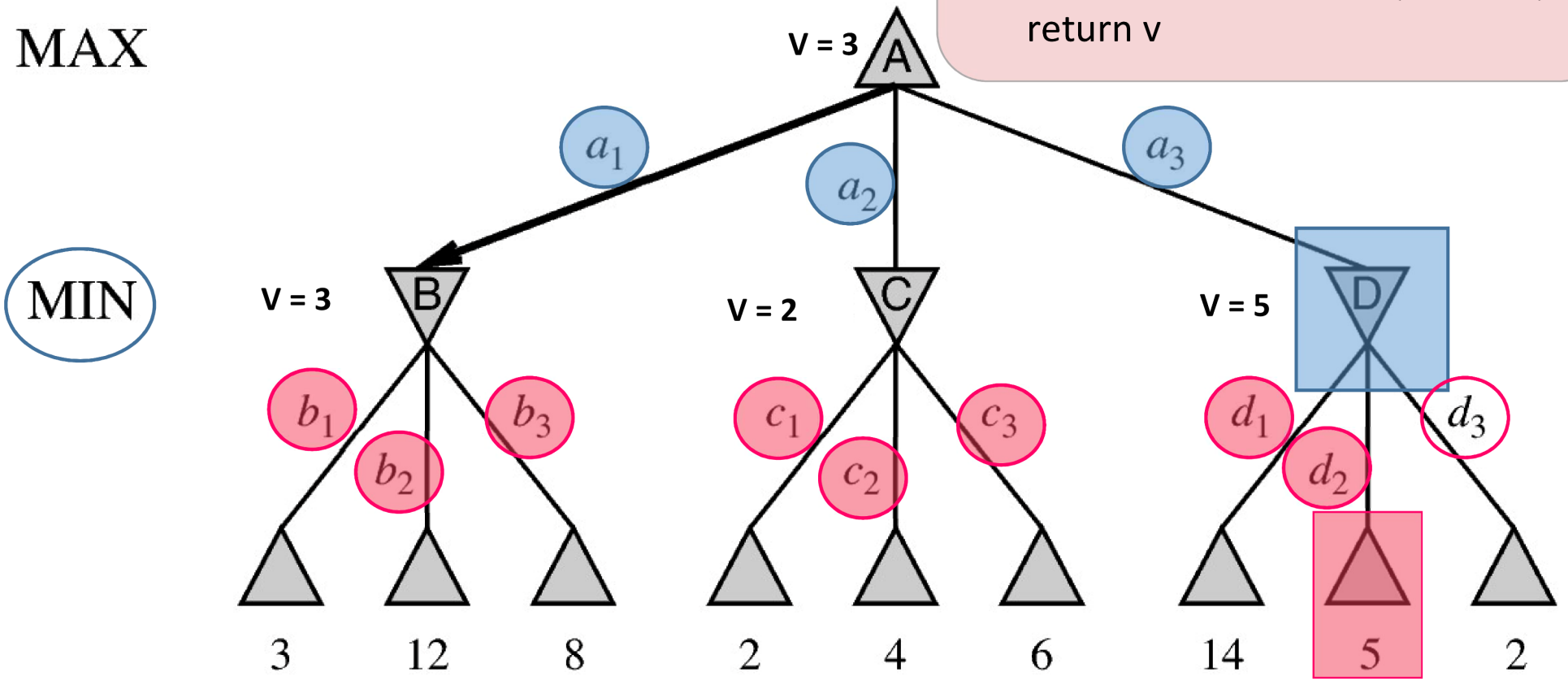
```
def min-value(state):
    initialize v = +∞
    for each action a of state:
        v = min(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

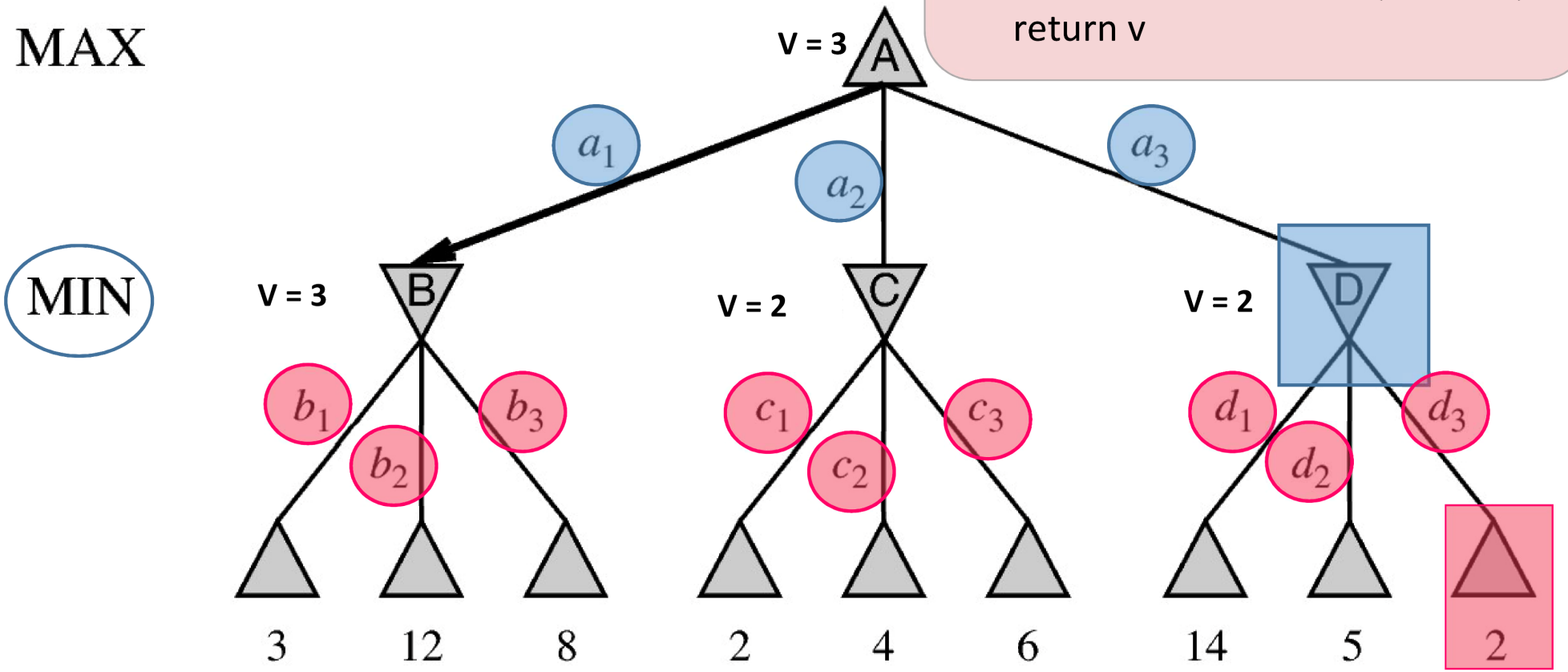
```
def min-value(state):
    initialize v = +∞
    for each action a of state:
        v = min(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

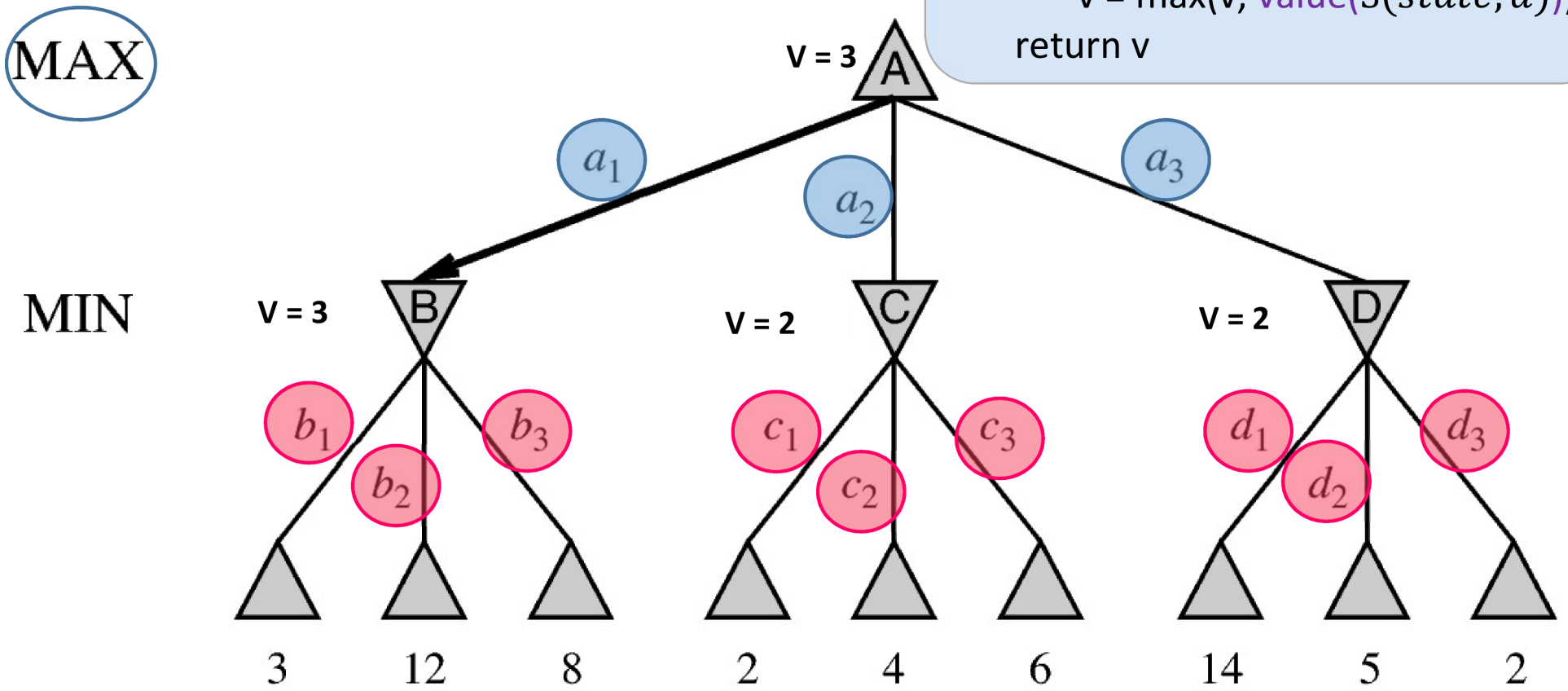
```
def min-value(state):
    initialize v = +∞
    for each action a of state:
        v = min(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

Minimax Example

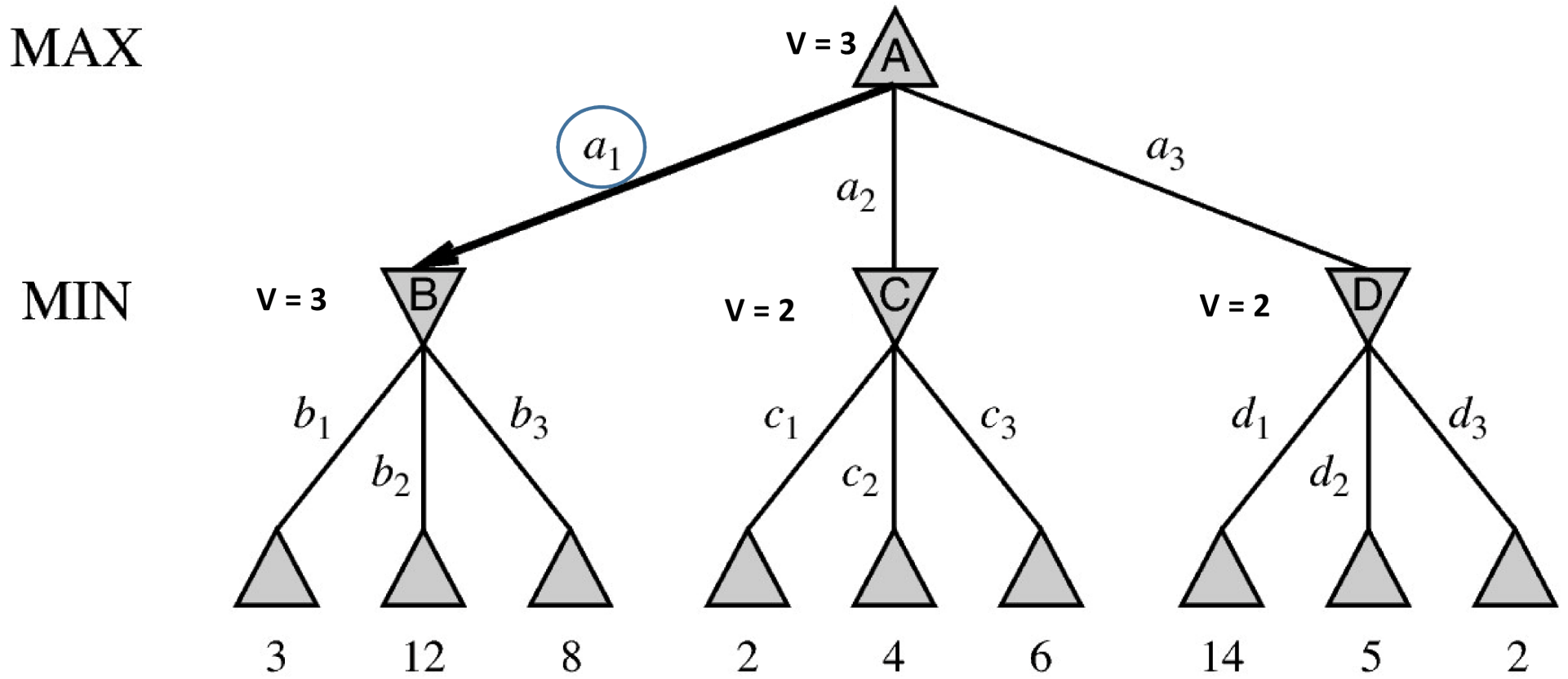
```
def max-value(state):
    initialize v = -∞
    for each action a of state:
        v = max(v, value(S(state, a)))
    return v
```



```
def value(state):
    if terminal: return utility
    if MAX: return max-value(state)
    if MIN: return min-value(state)
```

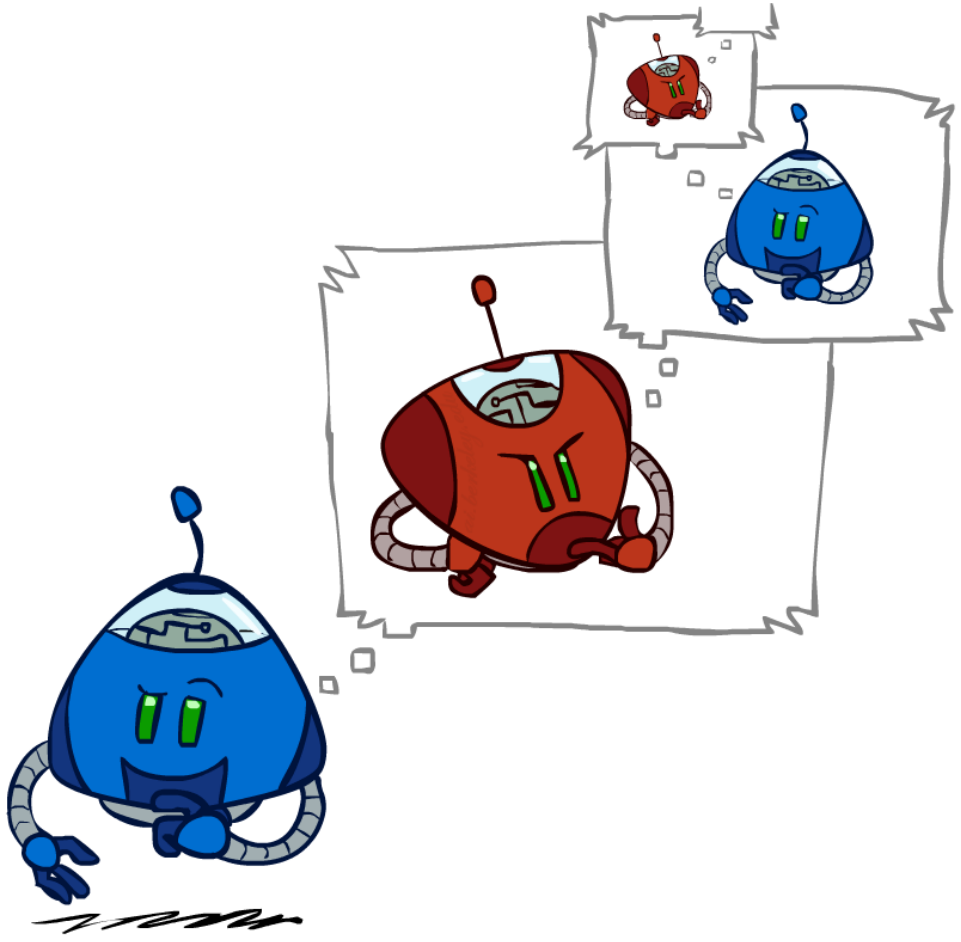
Minimax

```
def get-best-action(state):
    return argmax_{a∈Actions(state)} value(S(state, a))
```

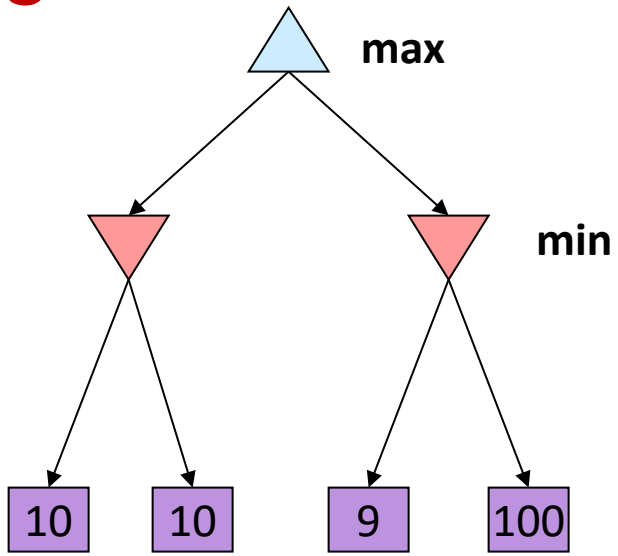


Minimax Properties

- Completeness:
  - Yes, if the game tree is finite
- Optimality:
  - Yes, if the opponent is optimal
  - What if not?



Minimax Properties



Optimal against a perfect player.

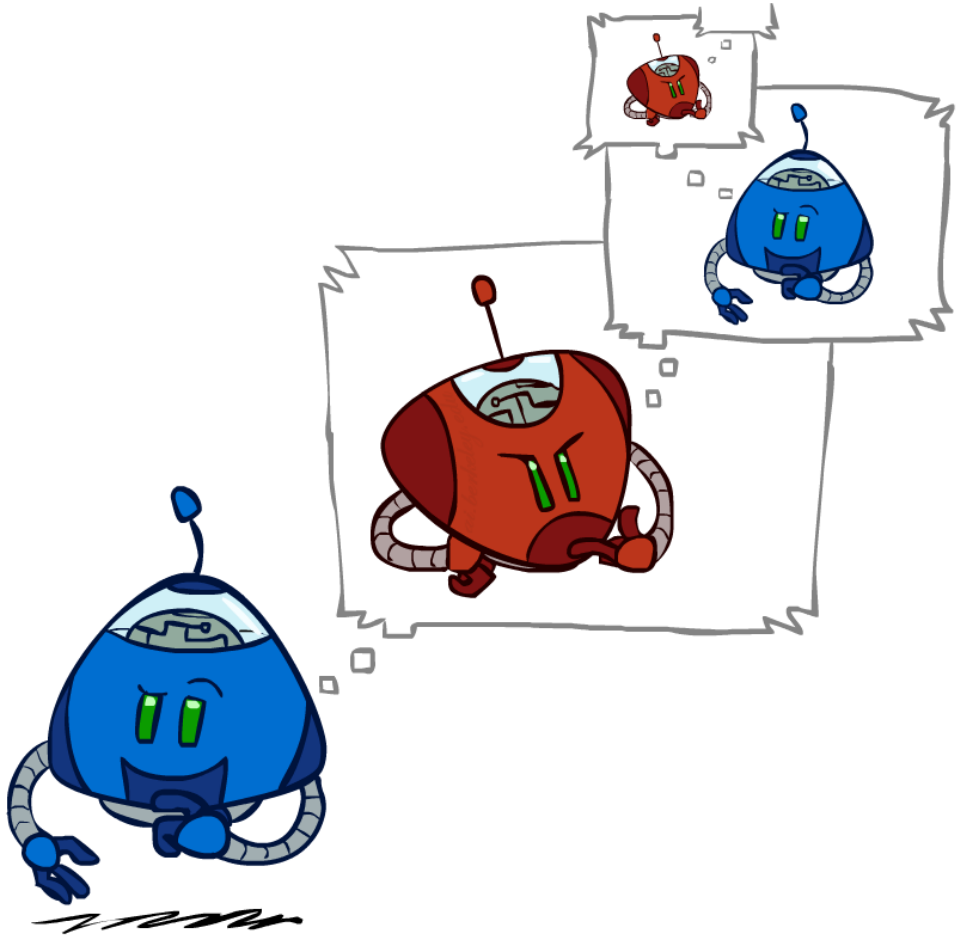
Otherwise not optimal but does even better than if the opponent were optimal

(ie would get at least as much as it would if the opponent were optimal)

Minimax Properties

- Completeness:
  - Yes, if the game tree is finite
- Optimality:
  - Yes, if the opponent is optimal
- Complexity
  - Same as DFS
  - Time: $O(b^m)$
  - Space: O(bm)

- Chess: $b \approx 35$, $m \approx 100$, Do we need to explore all of it? (can we even?)



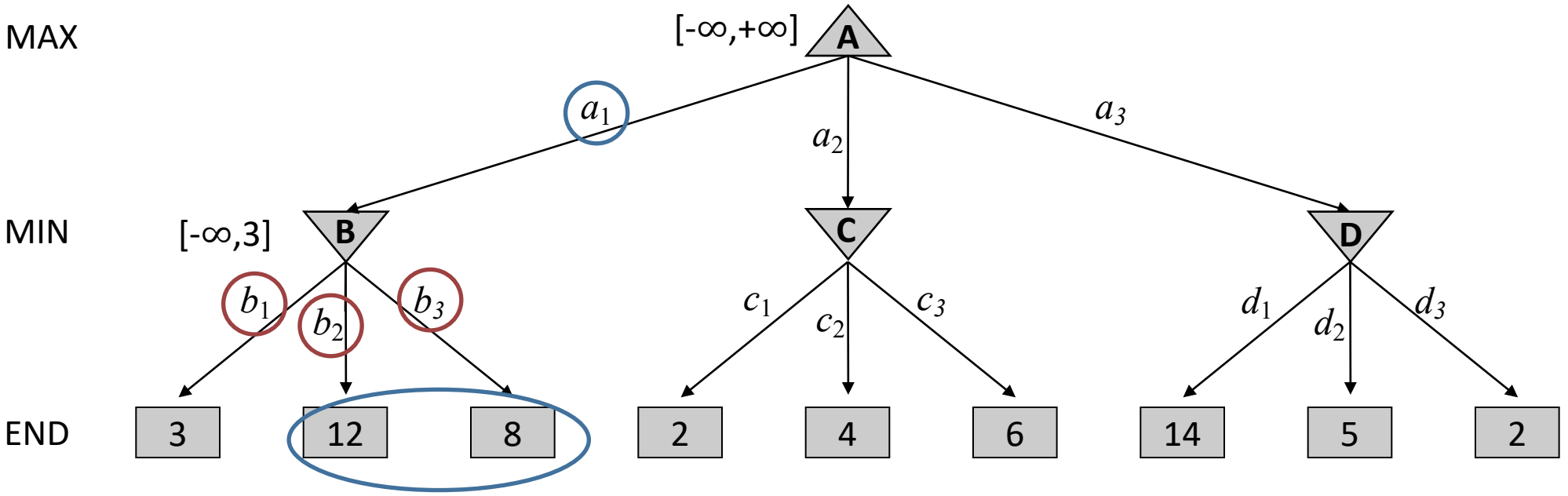
Alpha-Beta Pruning

- Most games are too complex to consider every contingency
- **Prune** paths that don't look relevant/promising

- **Alpha-Beta Pruning**: calculate two values instead of one minimax value
  - Alpha: Best choice MAX can make at at any choice point along this path
  - Beta: Best choice MIN can make at any choice point along this path
- Use these to rule out/prune paths

Alpha-Beta Simple Example

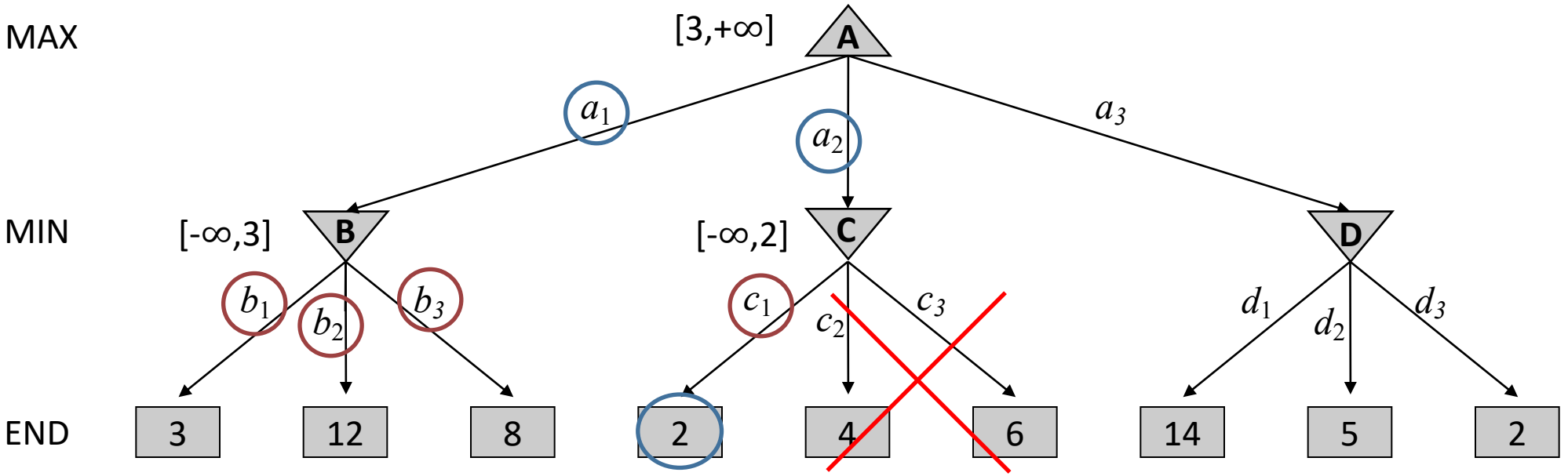
[α, β]
α: best for max along the path, initially -∞
β: best for min along the path, initially +∞



Min would not choose these

Alpha-Beta Simple Example

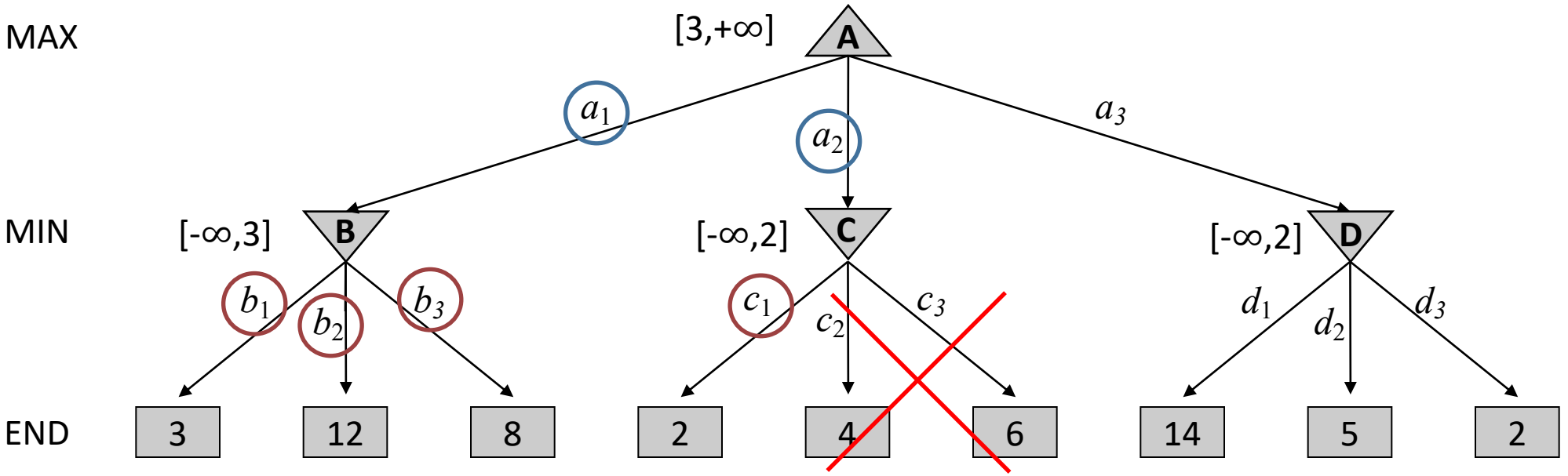
[α, β]
α: best for max along the path, initially -∞
β: best for min along the path, initially +∞



2 is better than 3 for min. So max should never go down this path. No need to search further!

Alpha-Beta Simple Example

[α, β]
α: best for max along the path, initially -∞
β: best for min along the path, initially +∞



In total we have saved ourselves from looking at 2 nodes.
In more complicated games, pruning can help significantly!

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

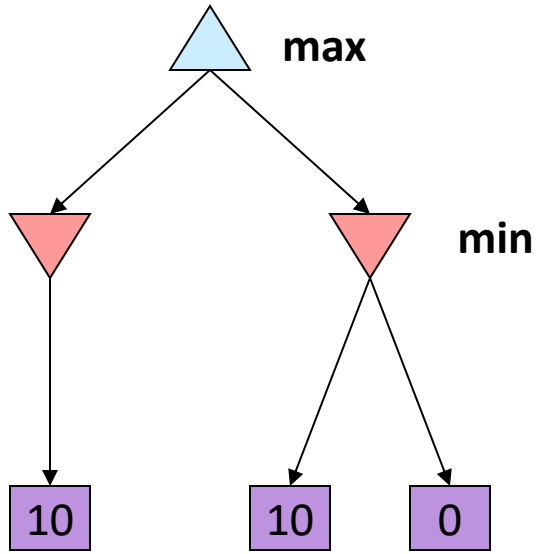
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

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

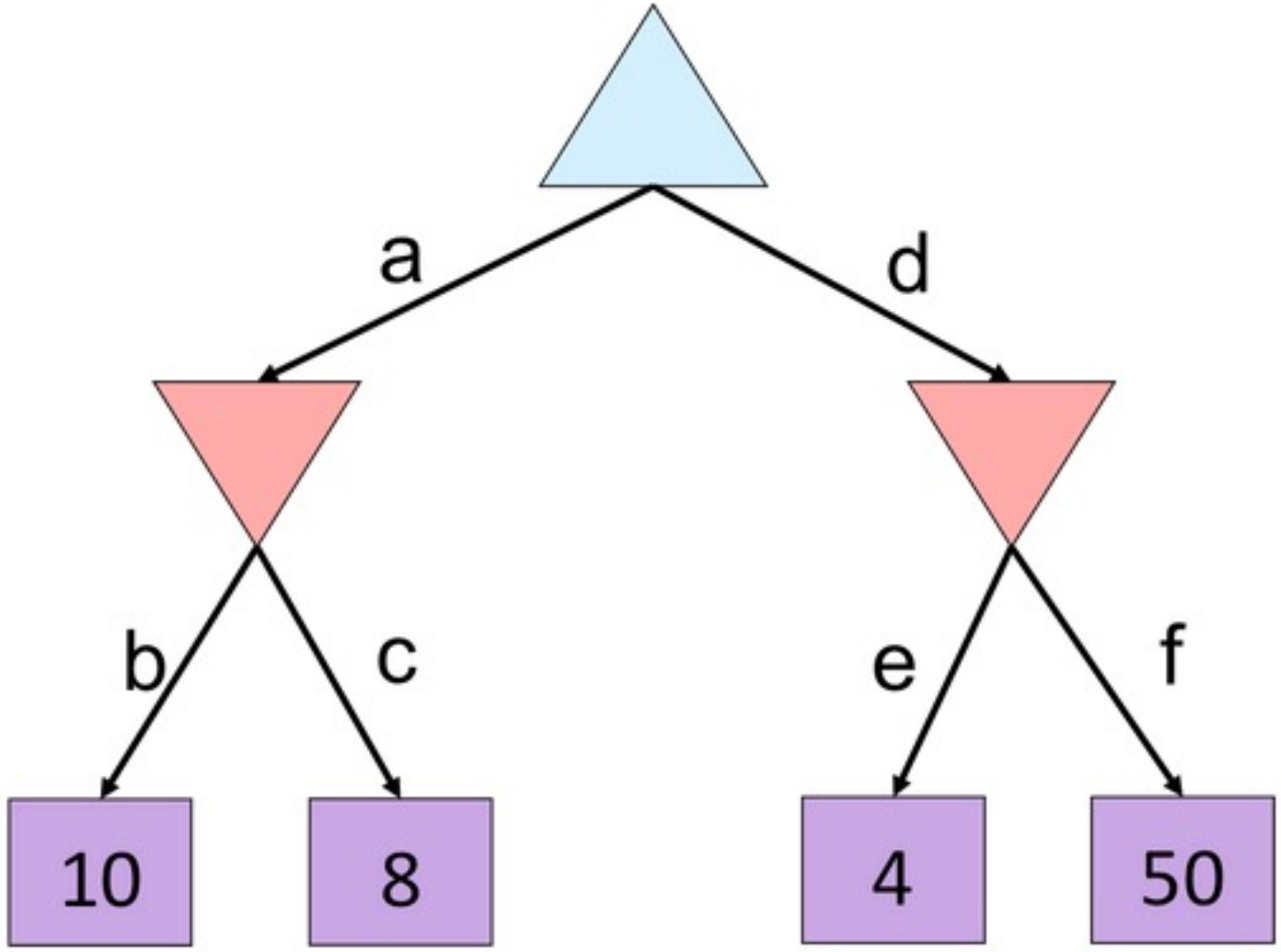
Home Exercise: Trace the alpha-beta pruning algorithm on the previous example

Alpha-Beta Pruning Properties

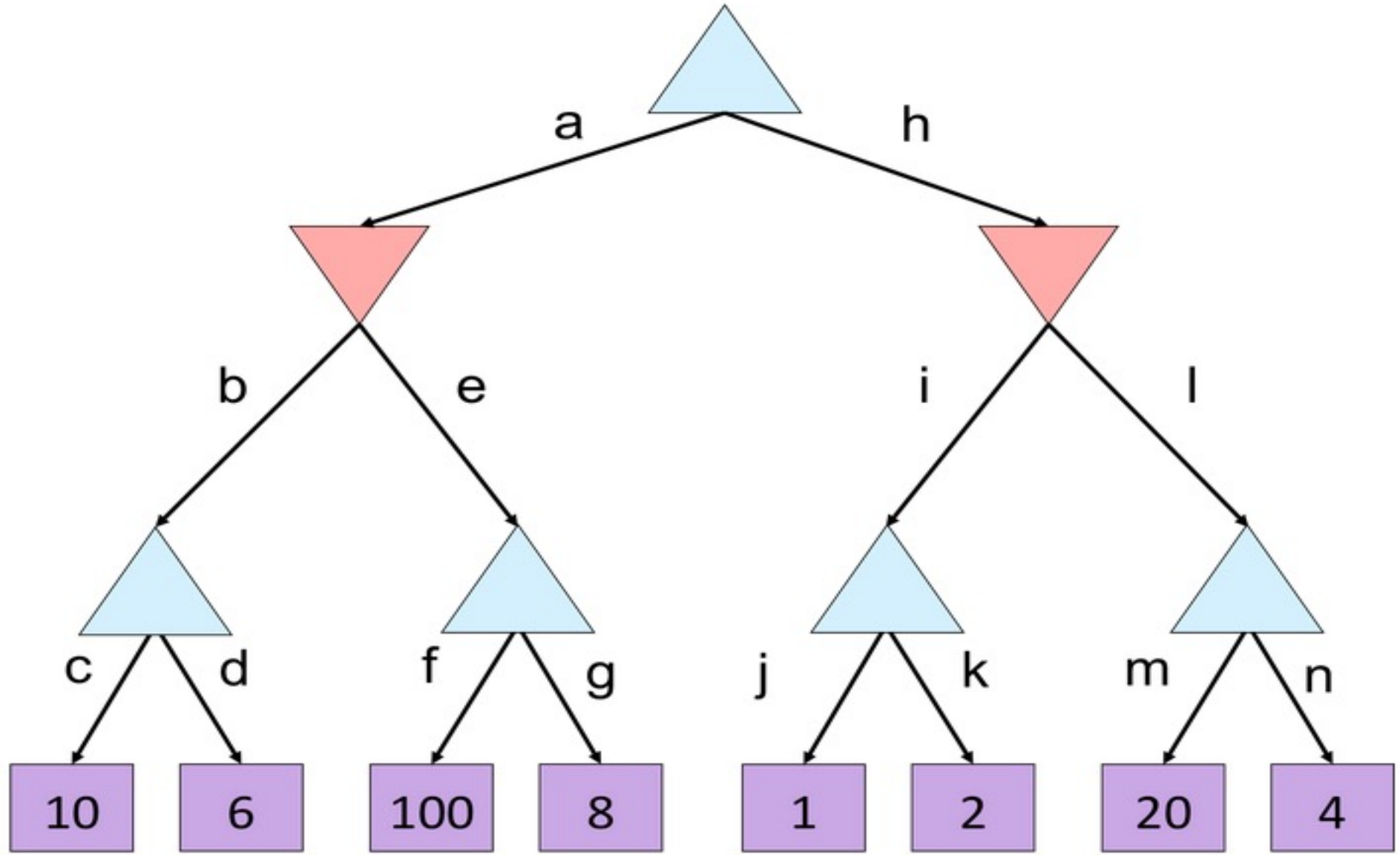
- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of bigger games, e.g. chess, is still hopeless…
- This is a simple example of metareasoning (computing about what to compute)



Alpha-Beta Quiz



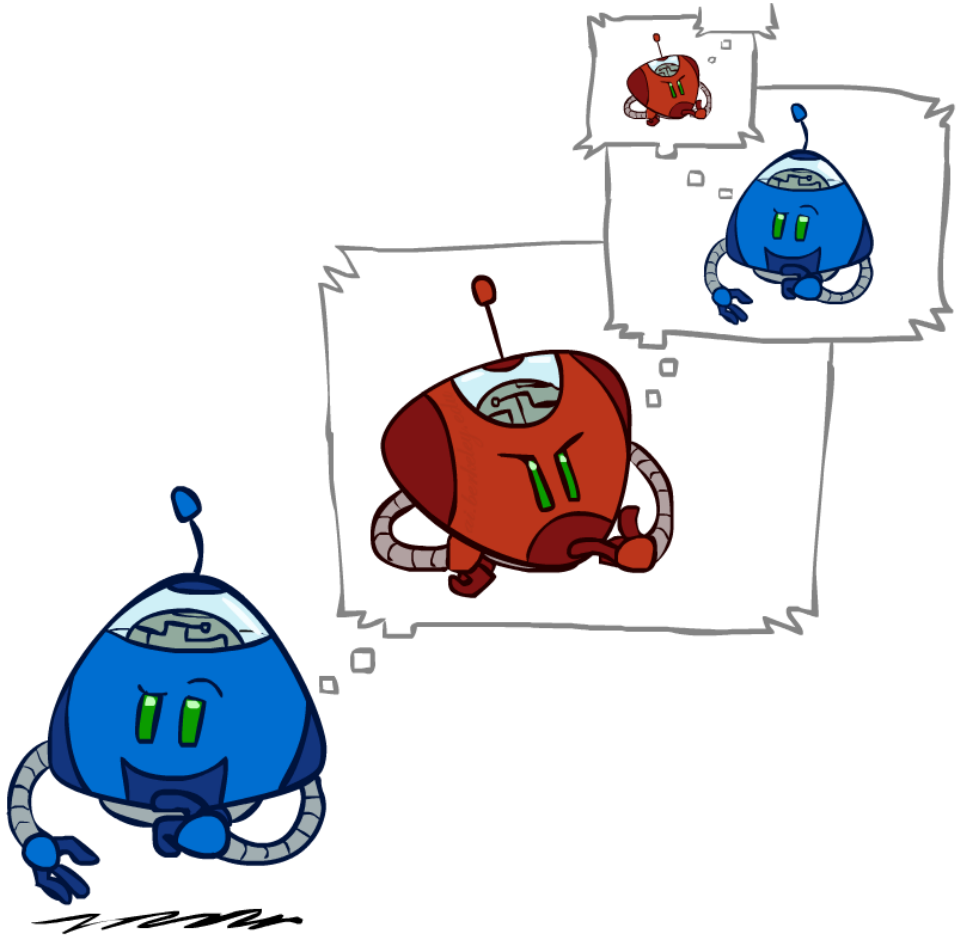
Alpha-Beta Quiz 2



Resource Limits

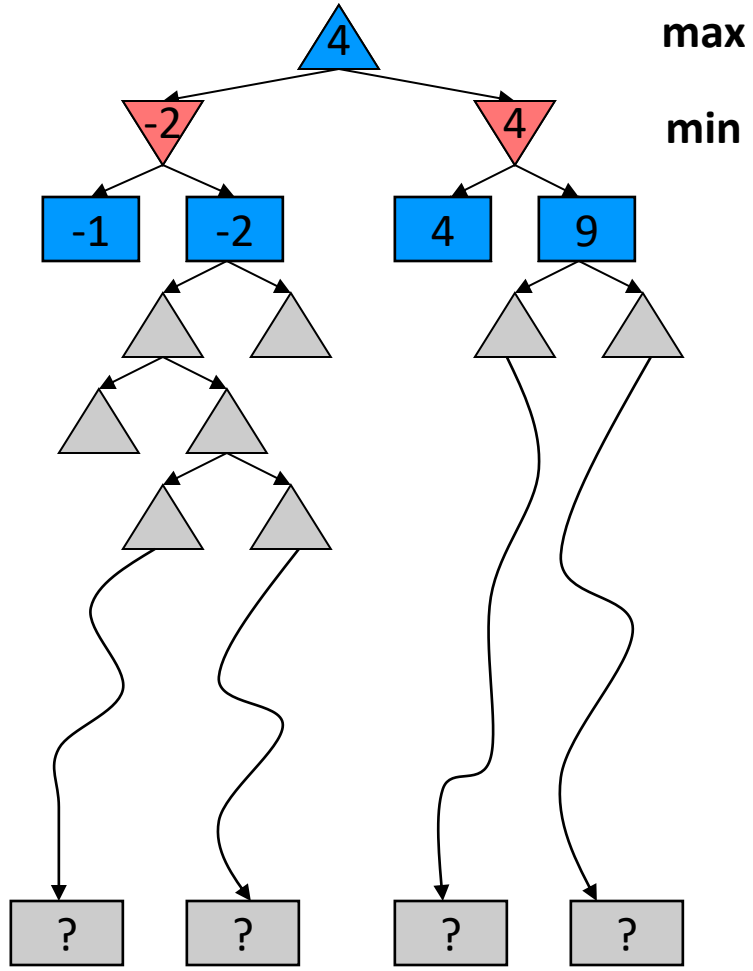
Remember Minimax Properties

- Completeness:
  - Yes, if the game tree is finite
- Optimality:
  - Yes, if the opponent is optimal
- Complexity
  - Same as DFS
  - Time: $O(b^m)$
  - Space: O(bm)
- Chess: $b \approx 35$, $m \approx 100$, Do we need to explore all of it? (can we even?)



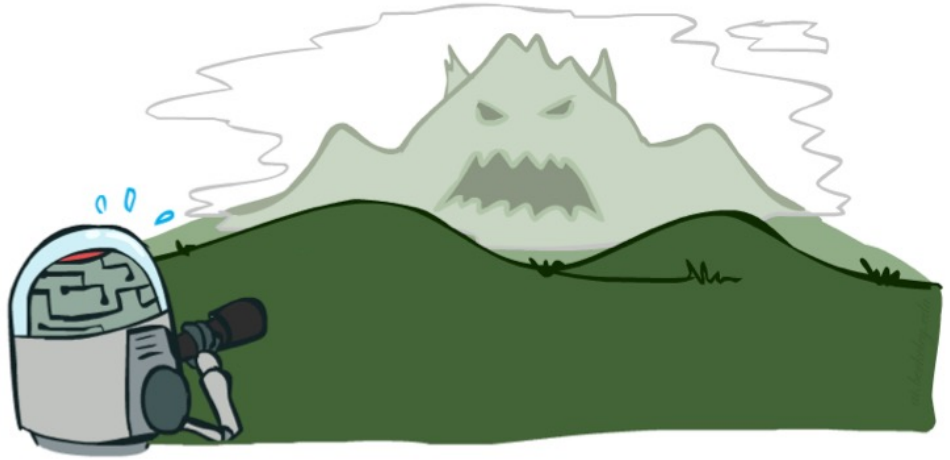
Resource Limits

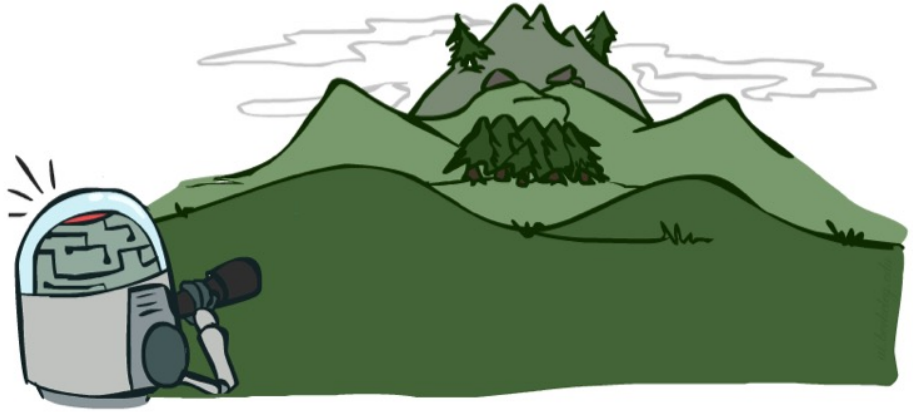
- Problem: In realistic games, cannot search to leaves even with pruning
  - $O(b^m)$ to $O(b^{m/2})$ with perfect ordering, only doubles the solvable depth
- Solution: Depth-limited search
  - How to assign values then?
  - Replace terminal utilities with an **evaluation function** for non-terminal positions
- Optimality is gone BUT we can actually play games ☺
- Example: We have 100 seconds, can explore 10K nodes / sec
  - 10^6 nodes per move ~ $35^{8/2}$
  - α-β reaches about depth 8 – decent chess program
- More depth makes a BIG difference
- Use iterative deepening for an anytime algorithm
  - If you have time, search deeper



Depth Matters

- Evaluation functions are always **imperfect**
- Searching deeper is better!
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation



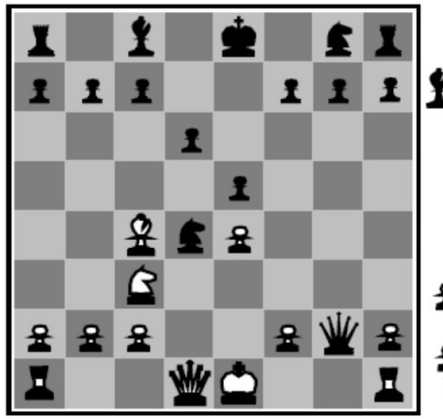


Evaluation Functions

- Score non-terminal states
- Ideally the actual minimax value of the position
- As with heuristics in search, need domain knowledge
- Typically weighted linear combination of features:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_i^n w_i f_i(s)$$

- E.g. features :
  - $f_1(s)$ = (num white queens – num black queens)
  - $f_2(s)$ = (white's sum of piece values – black's sum of piece values)
  - $f_3(s)$ = (white's pawn structure)
  - $f_4(s)$ = (king's safety)



Eval(*s*) = -5.6

White to move

Black winning

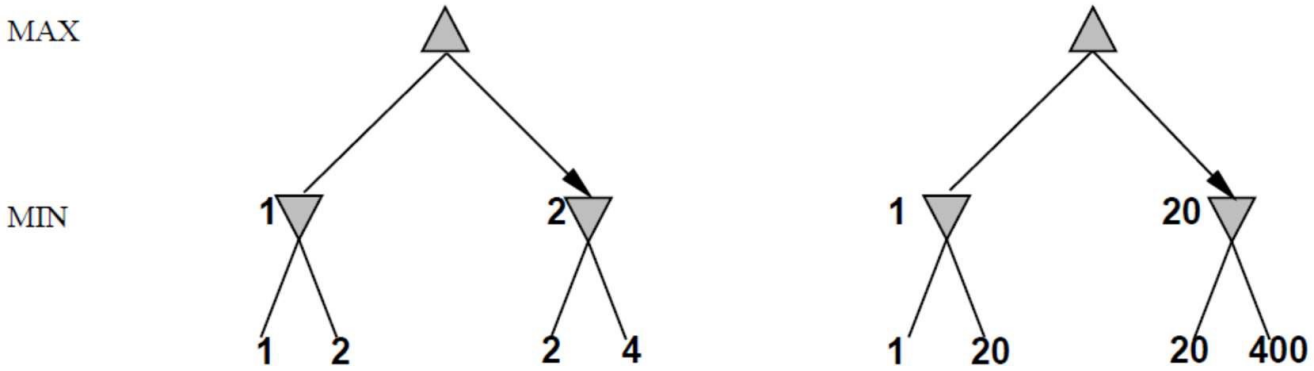
Eval(*s*) = 1.2

Black to move

White slightly better

Evaluation Functions

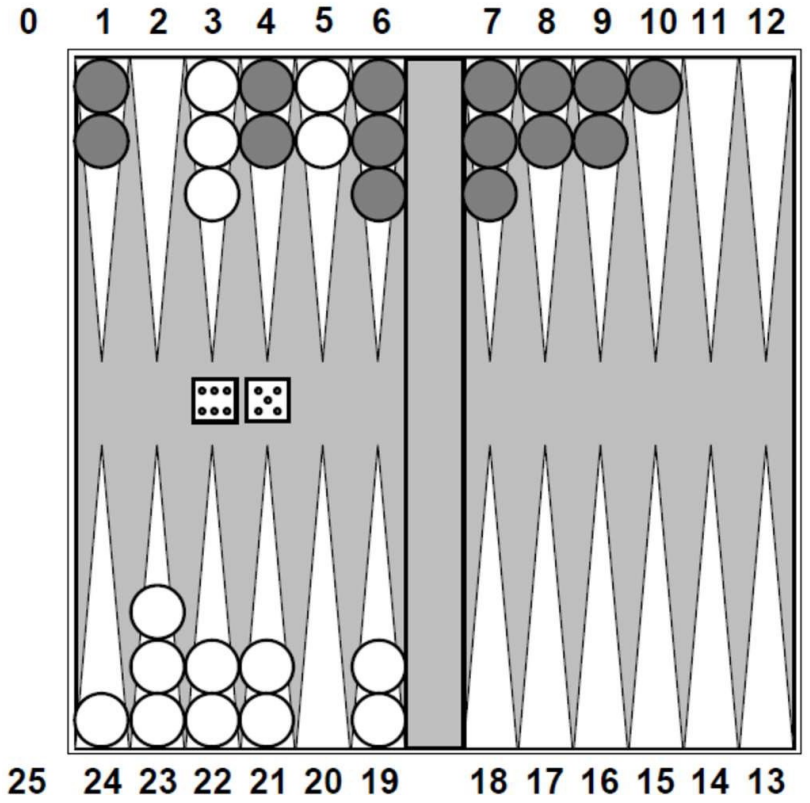
- Do the actual values matter?



- No! Only relative order matters

Stochastic Games

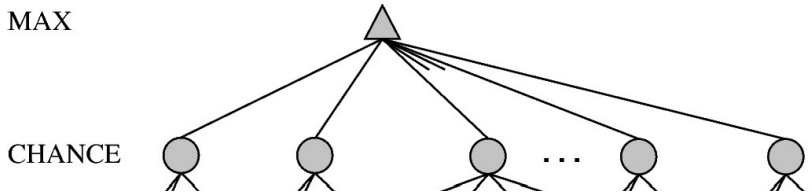
- Uncertain Outcomes
- Where does the stochasticity come from?
  - Explicit randomness: Rolling dice, card shuffle
  - Unpredictable opponents: Random ghosts in pacman
  - Failed actions: Robot wheel slip
- E.g. Backgammon: Allowable actions are determined by dice roll
- Need to include chance nodes in the game tree



Backgammon Game Tree

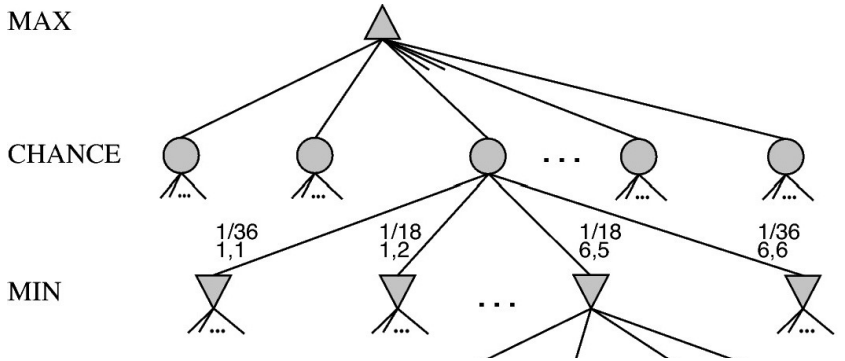
Backgammon Game Tree

MAX takes an action



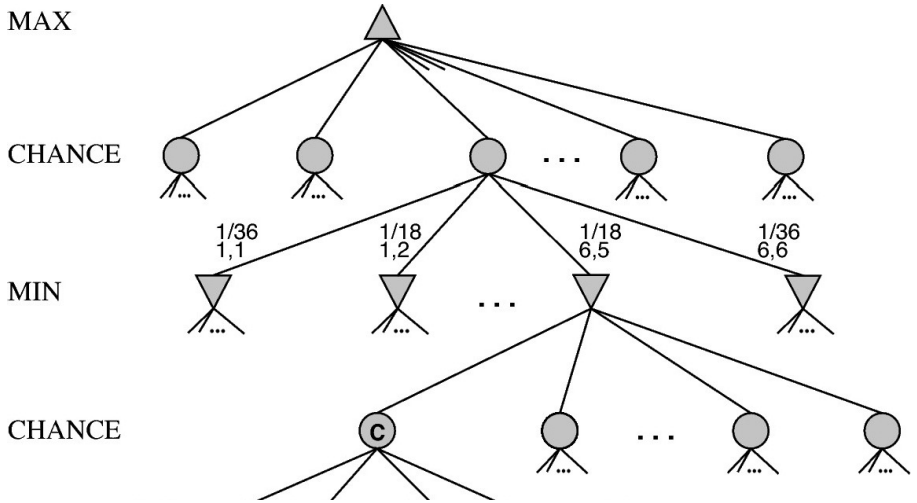
Backgammon Game Tree

MAX takes an action
MIN rolls the dice



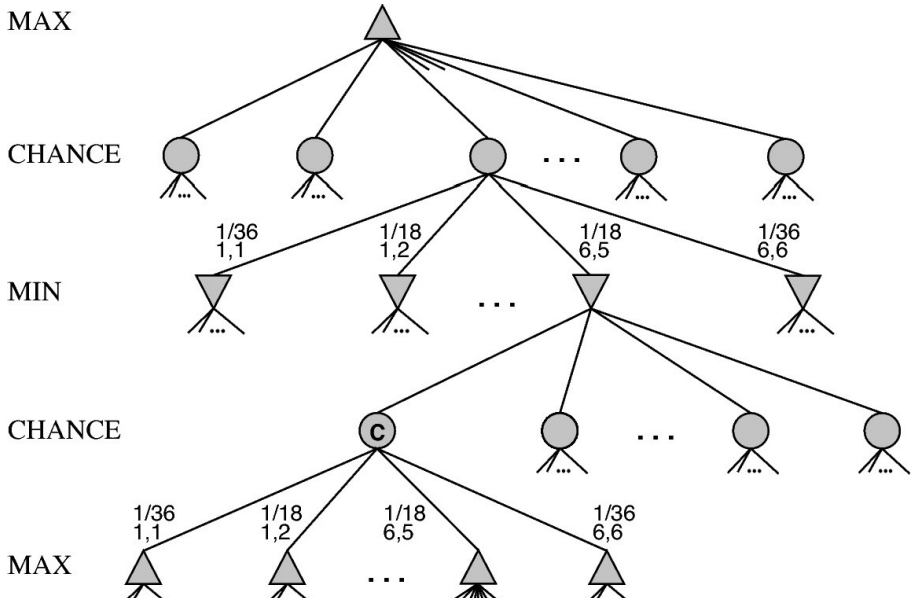
Backgammon Game Tree

MAX takes an action
MIN rolls the dice
MIN takes an action



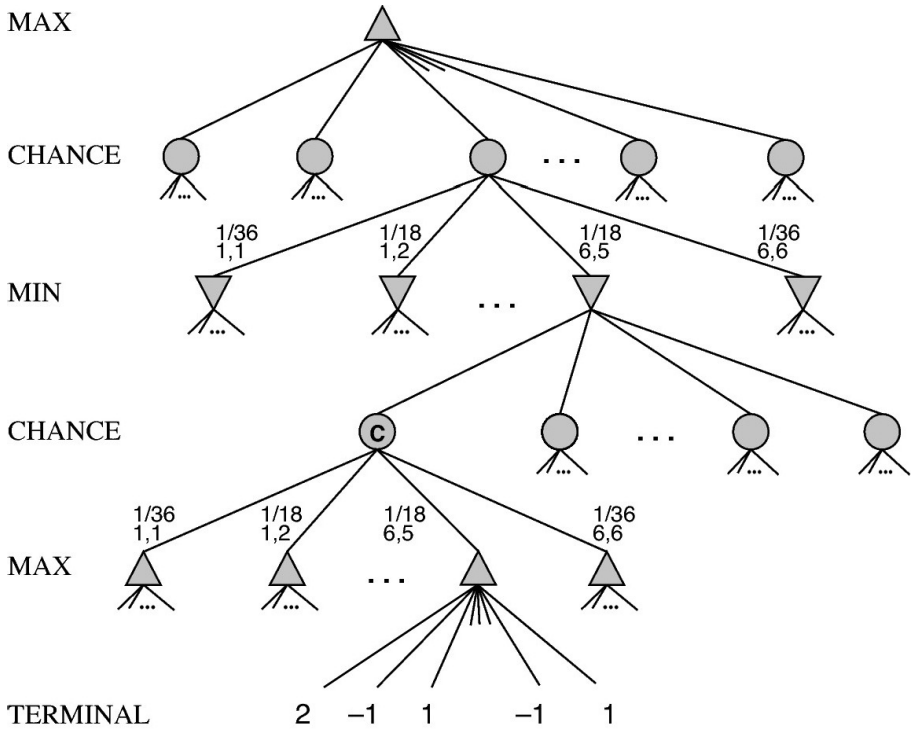
Backgammon Game Tree

MAX takes an action
MIN rolls the dice
MIN takes an action
MAX rolls the dice



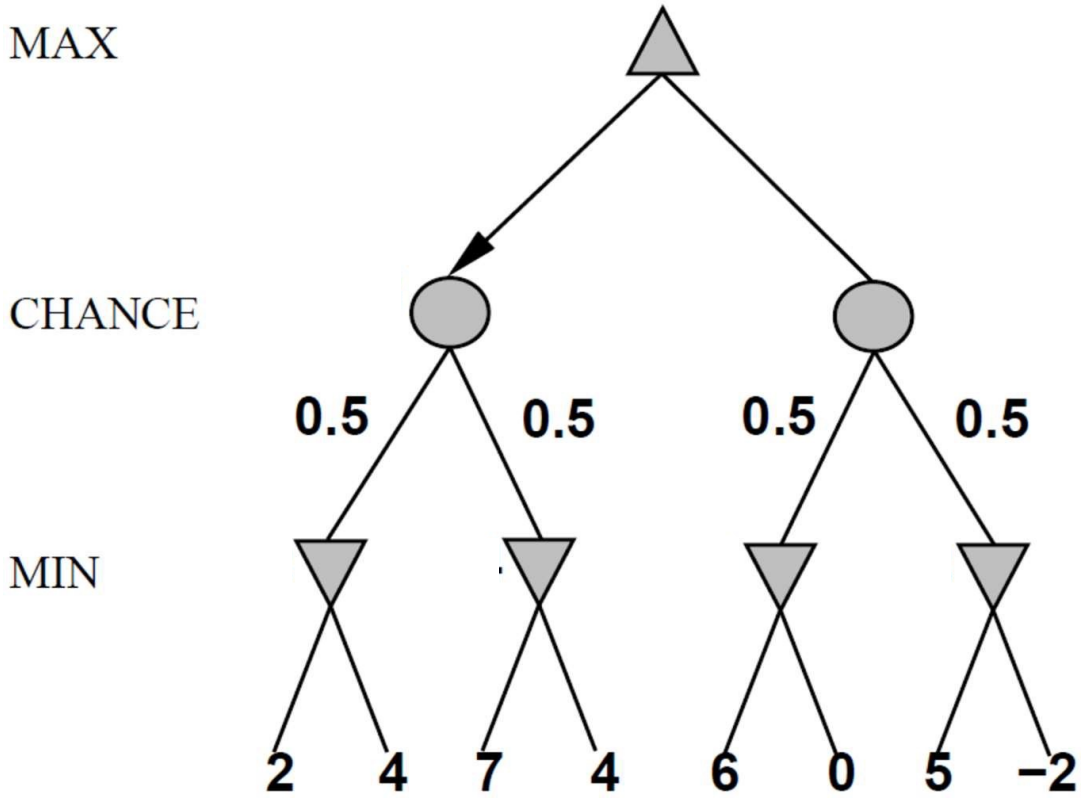
Backgammon Game Tree

MAX takes an action
MIN rolls the dice
MIN takes an action
MAX rolls the dice
MAX takes an action
Game Ends



Expectiminimax Search

- What should the values of nodes be?
  - Best-case, worst-case or average-case?
- Chance nodes take the average value
- MAX and MIN keep their behavior



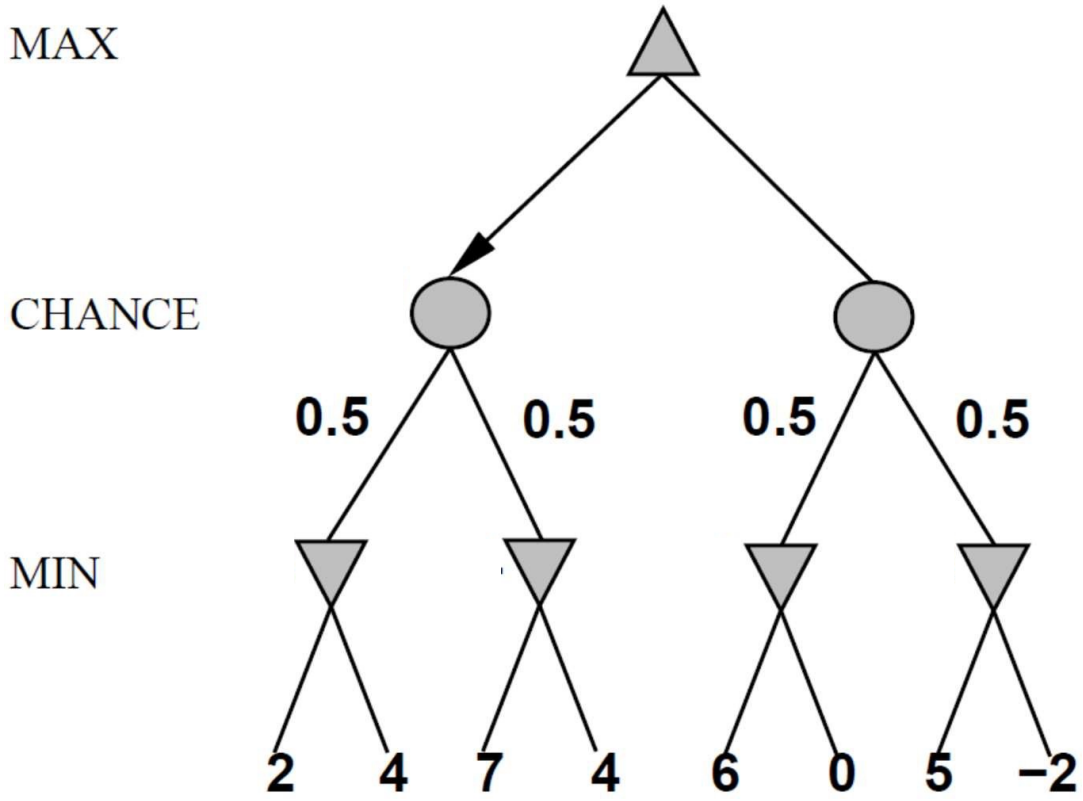
Expectiminimax Pseudocode

```
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)
    if the next agent is CHANCE: return exp-value(state)
```

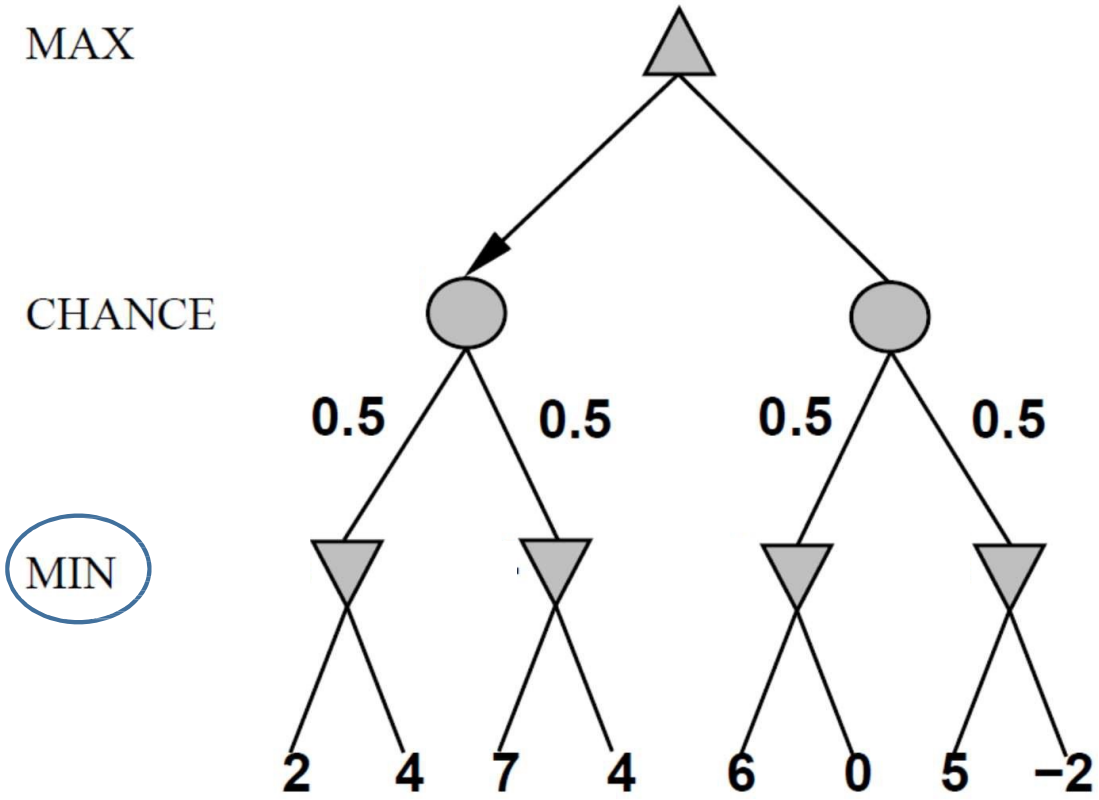
```
def exp-value(state):
    initialize v = 0
    for each action a of state:
        p = probability(S(s,a))
        v += p * value(S(s,a))
    return v
```

S(stat ,a):
Successor of the state given action a

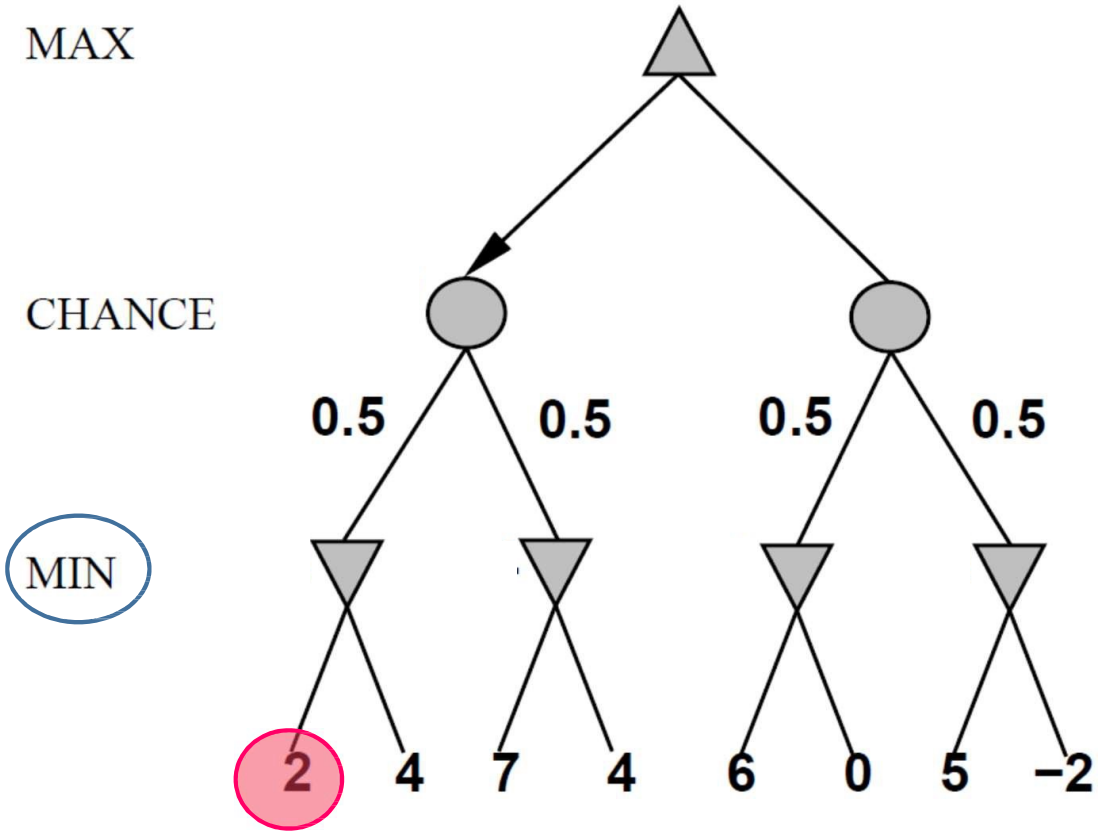
Expectiminimax Search



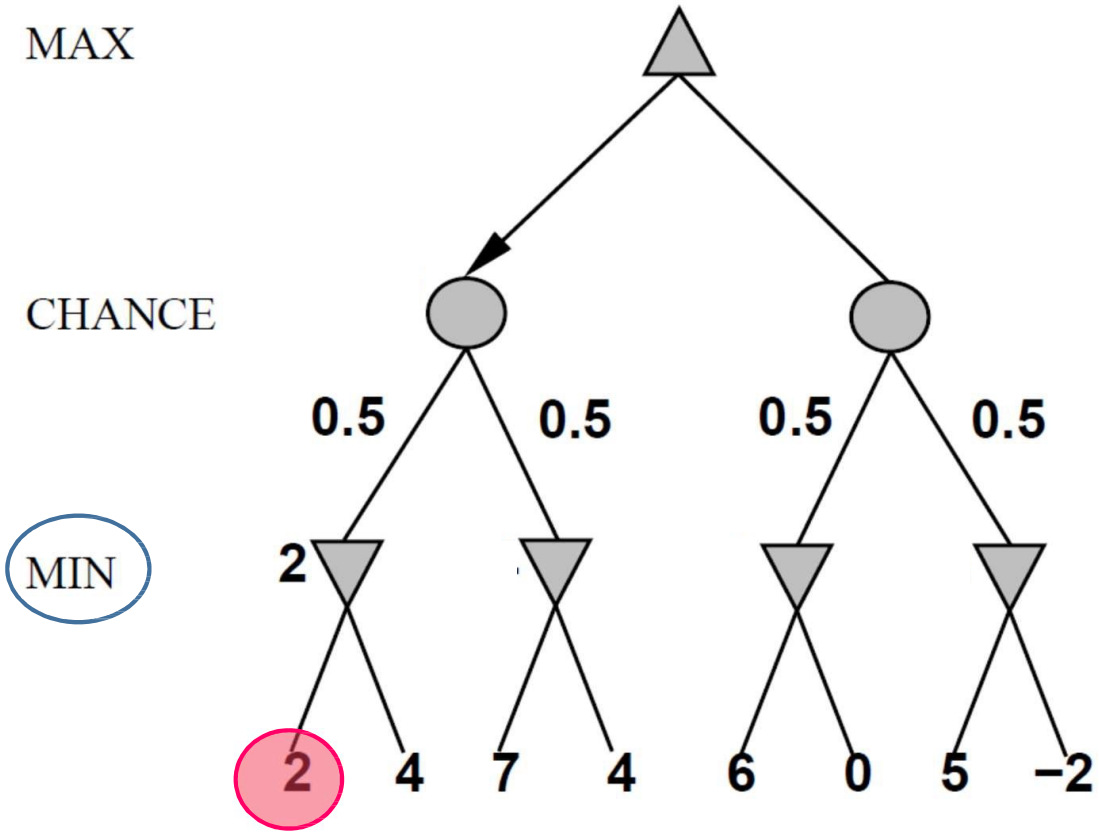
Expectiminimax Search



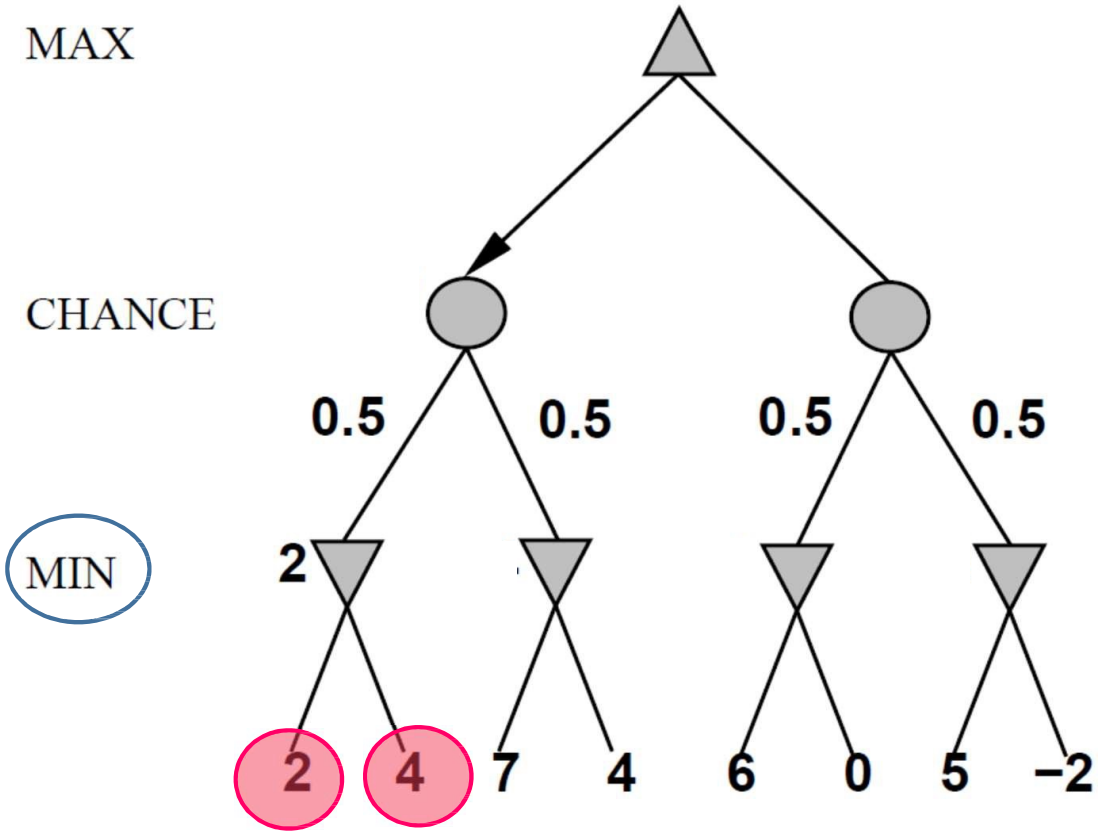
Expectiminimax Search



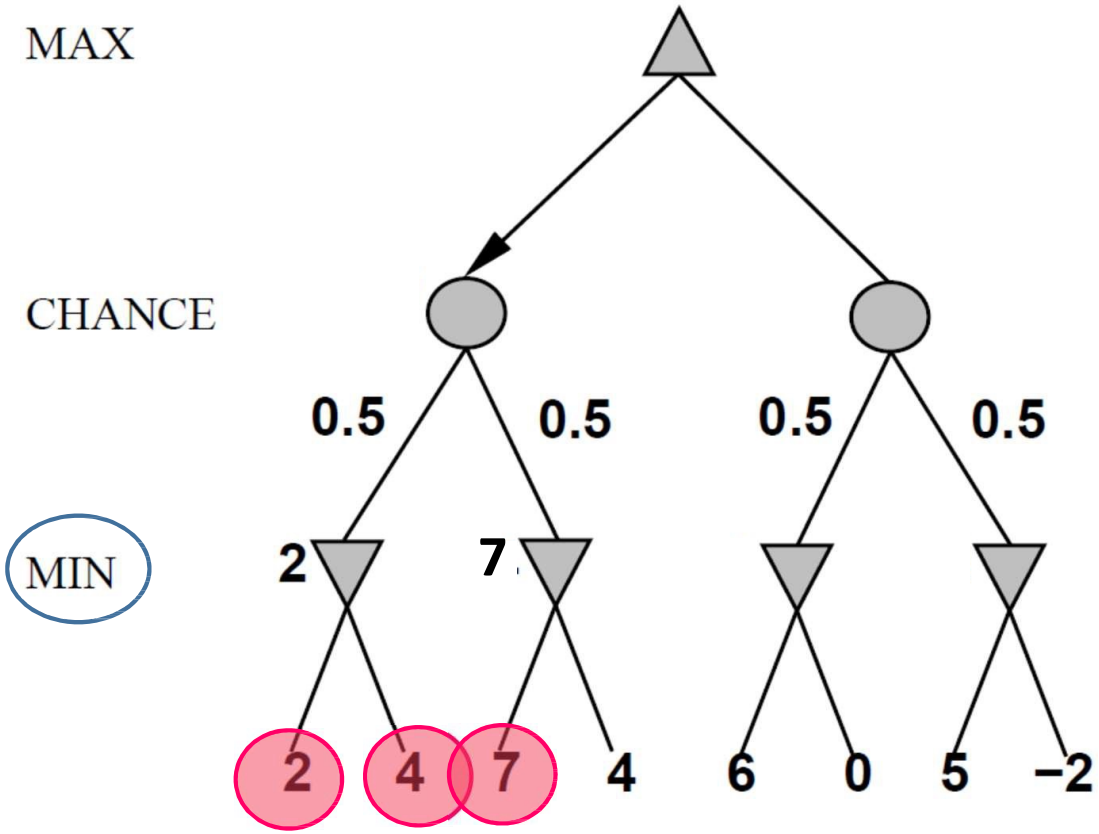
Expectiminimax Search



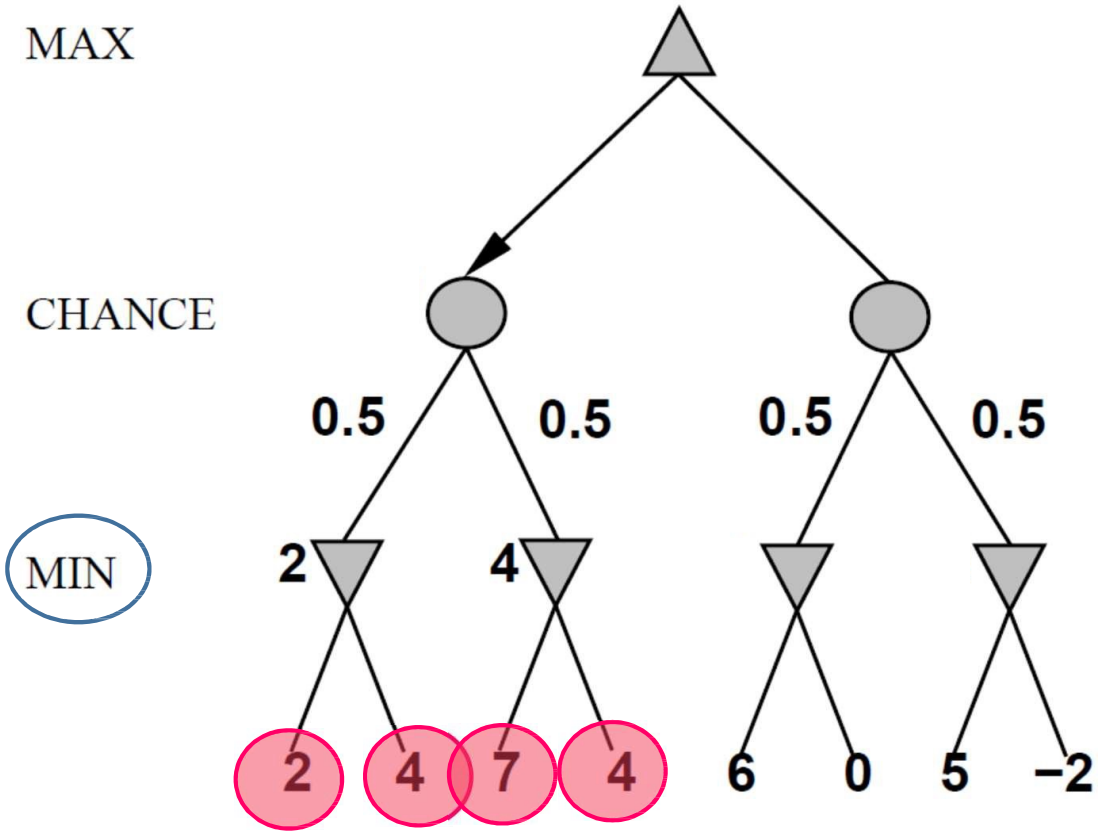
Expectiminimax Search



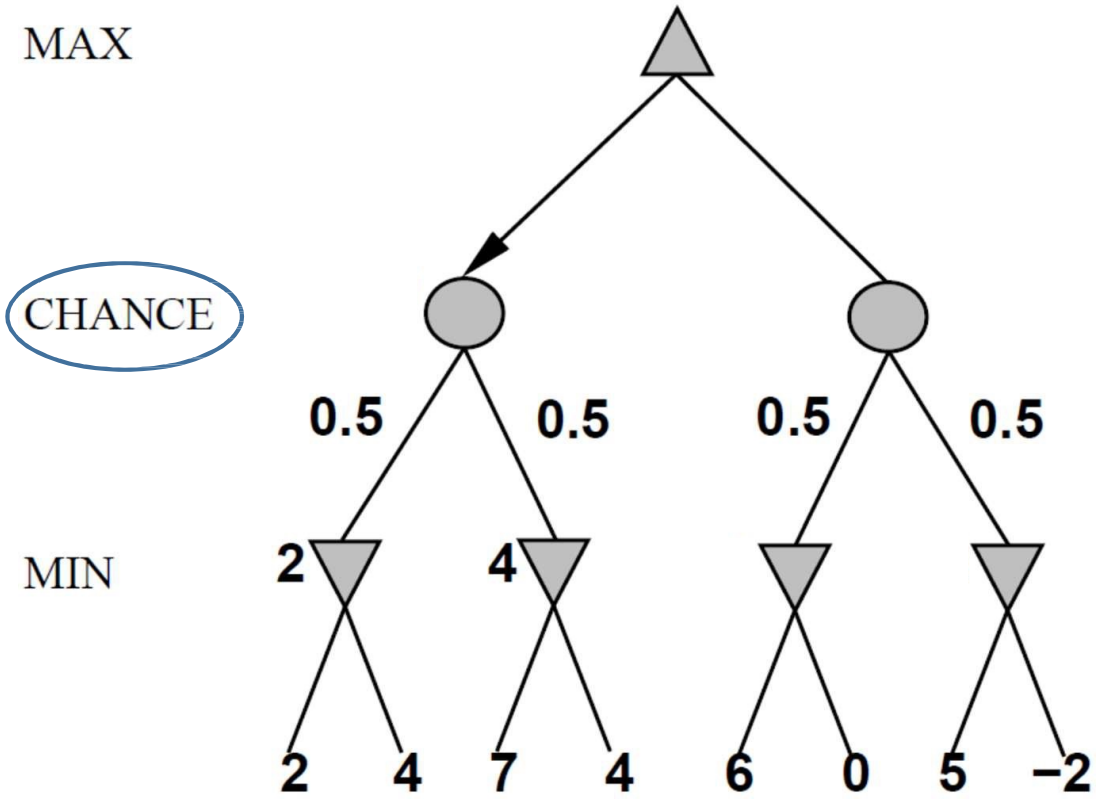
Expectiminimax Search



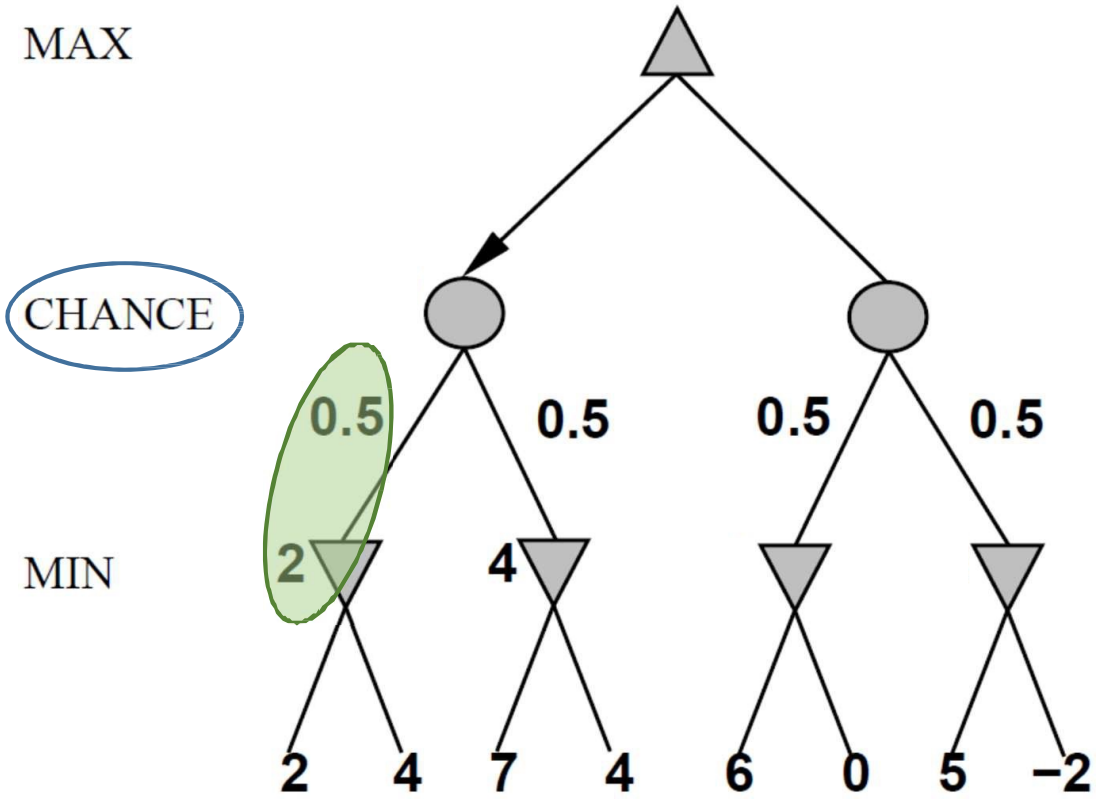
Expectiminimax Search



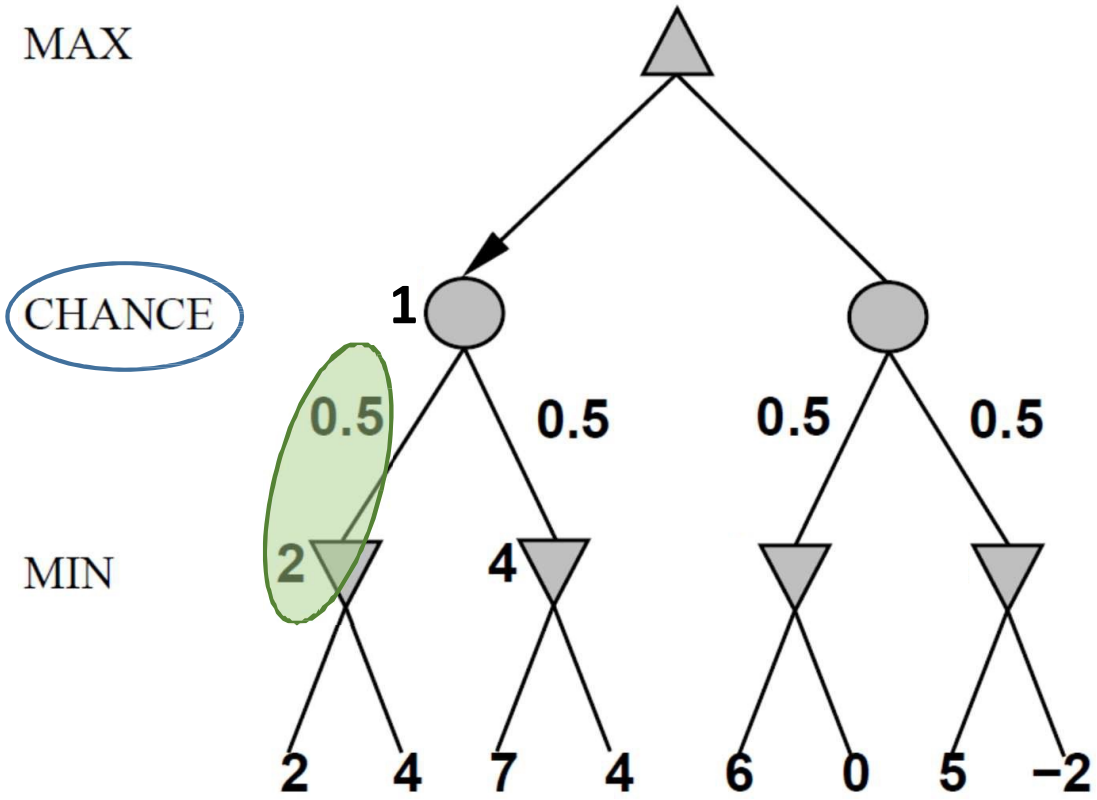
Expectiminimax Search



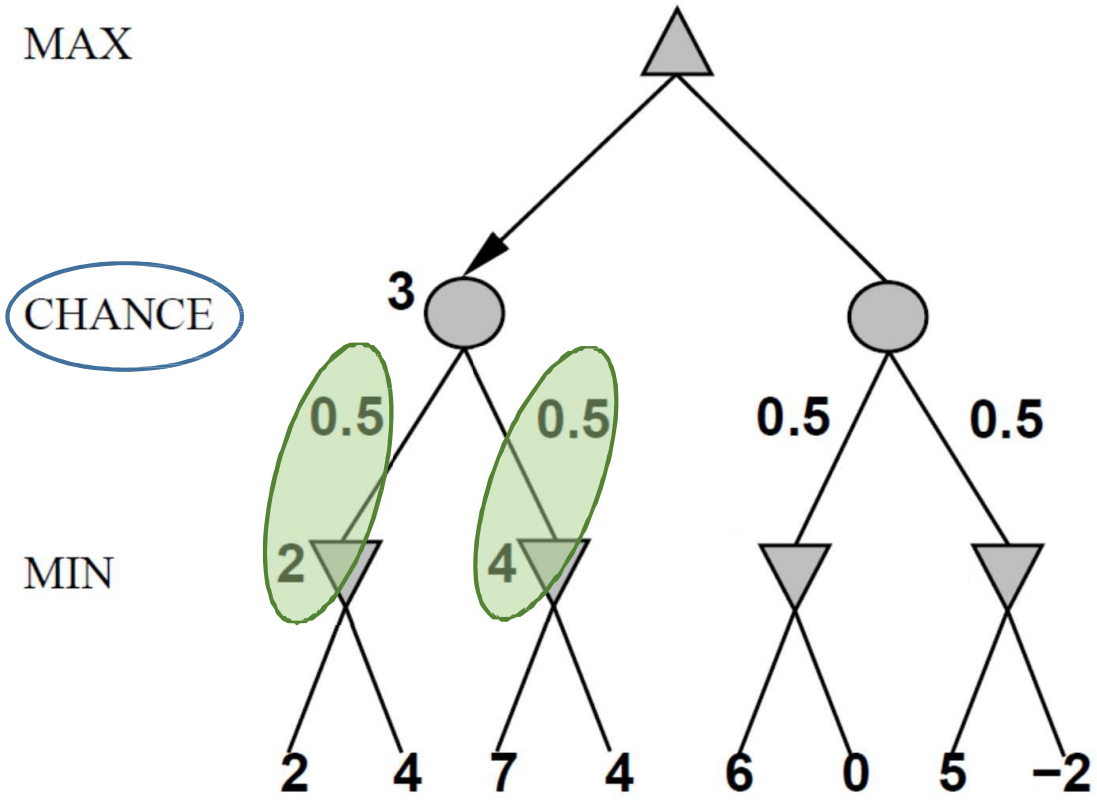
Expectiminimax Search



Expectiminimax Search

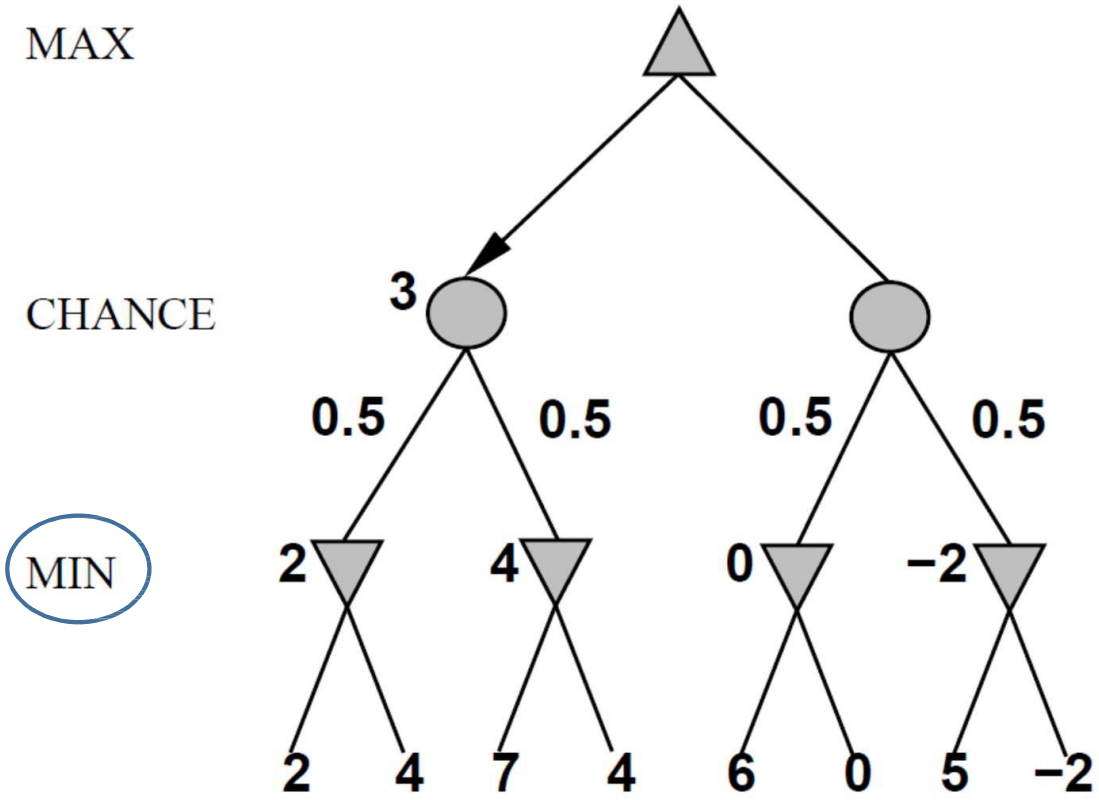


Expectiminimax Search

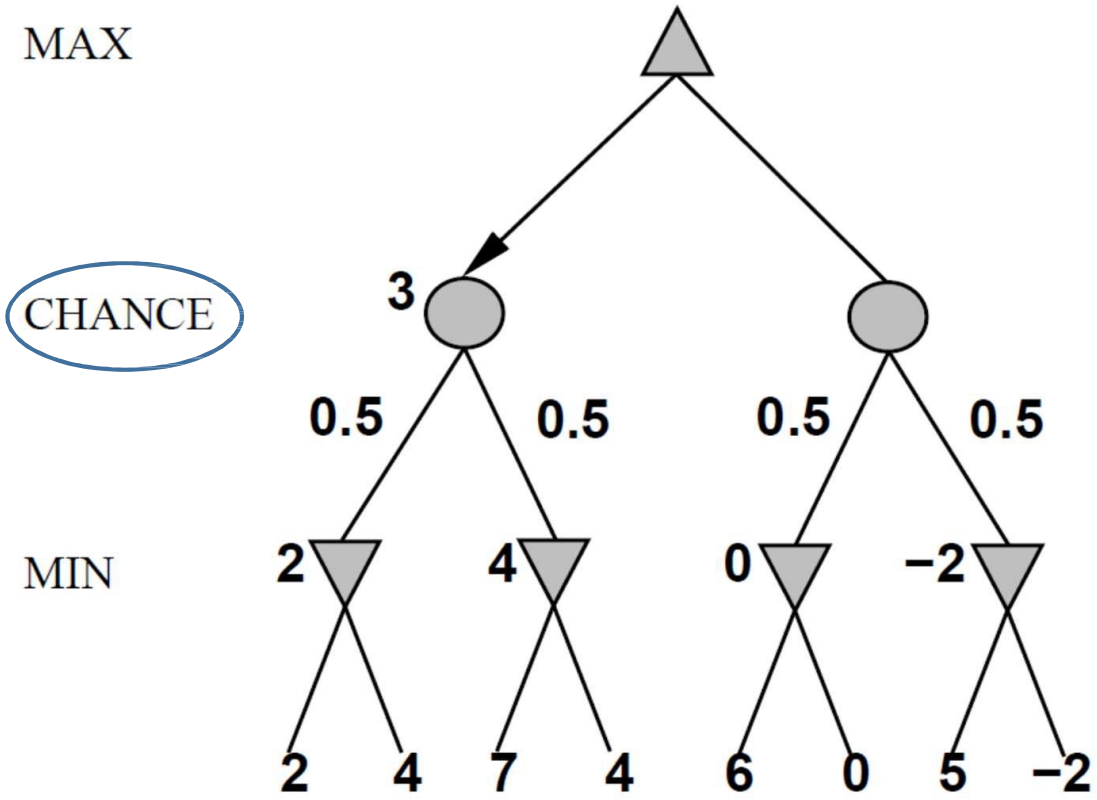


Expectiminimax Search

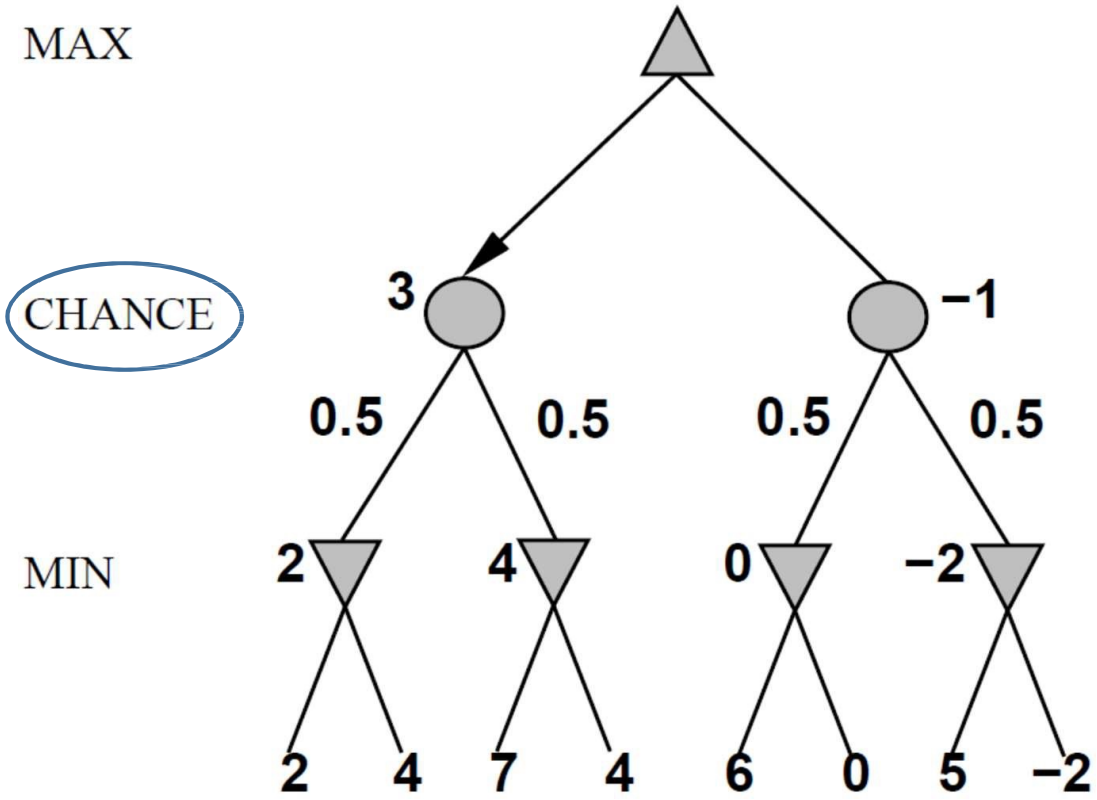
Skipping MIN steps



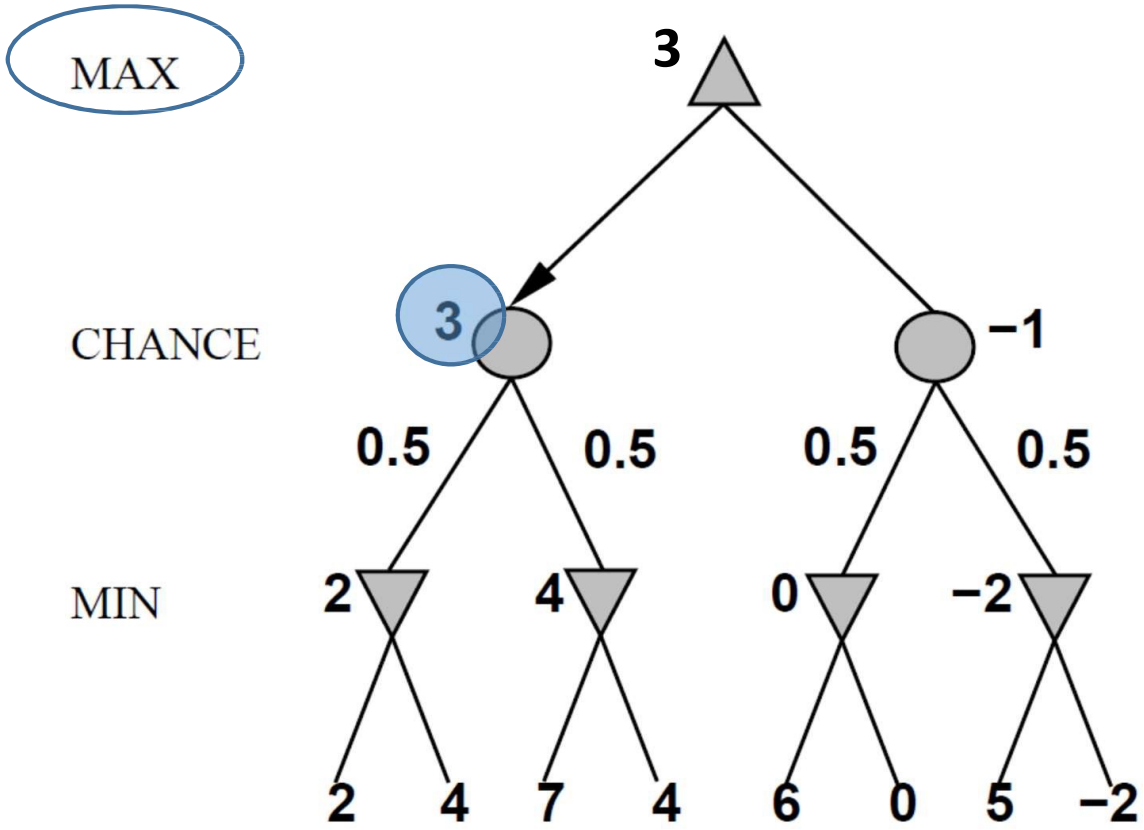
Expectiminimax Search



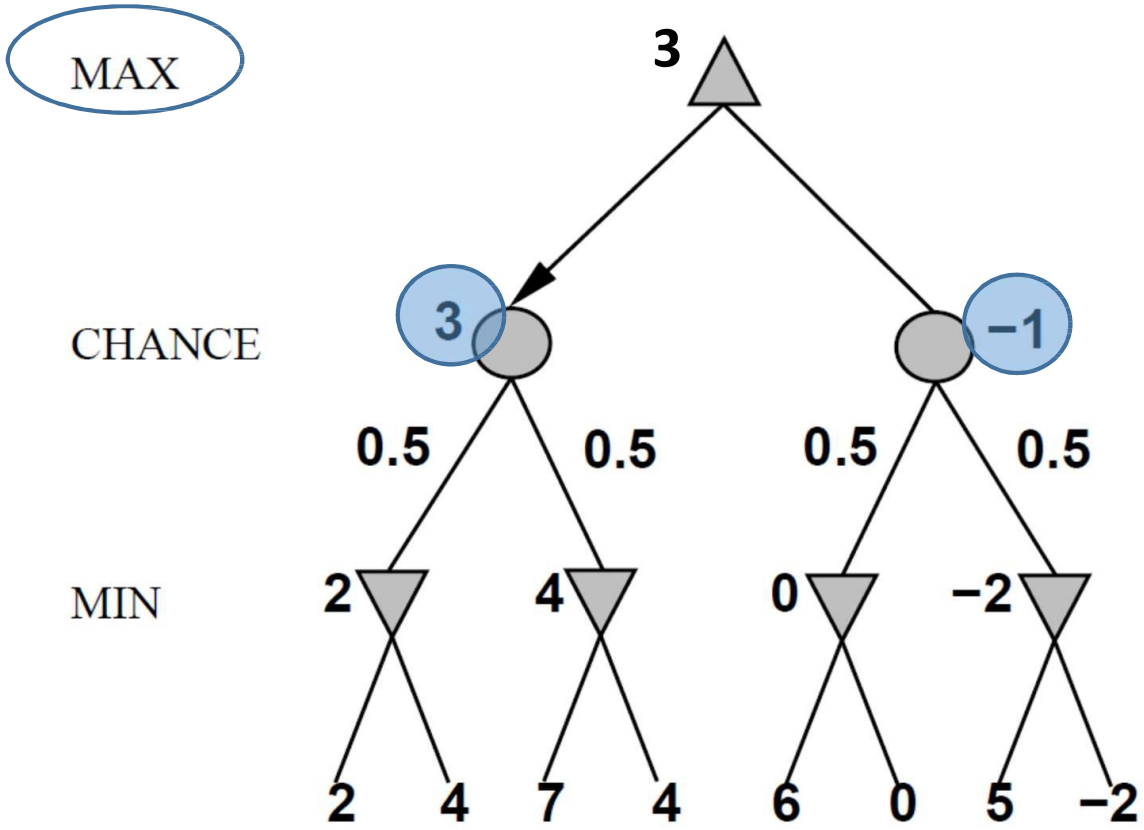
Expectiminimax Search



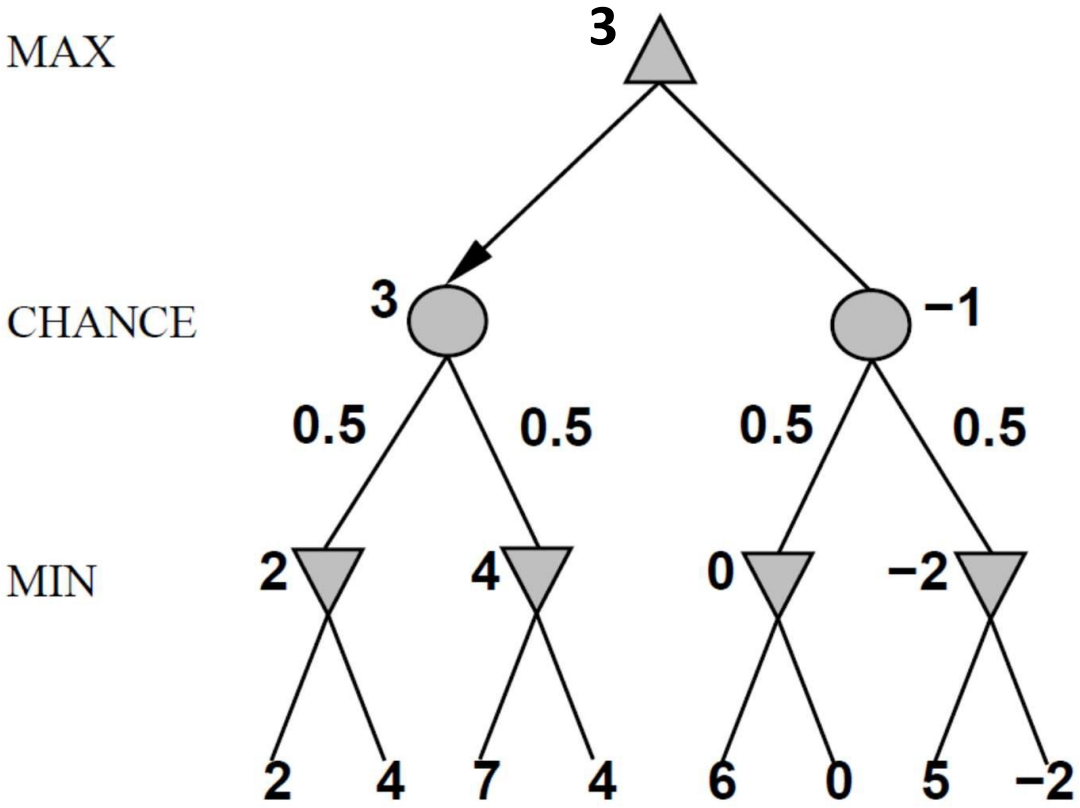
Expectiminimax Search



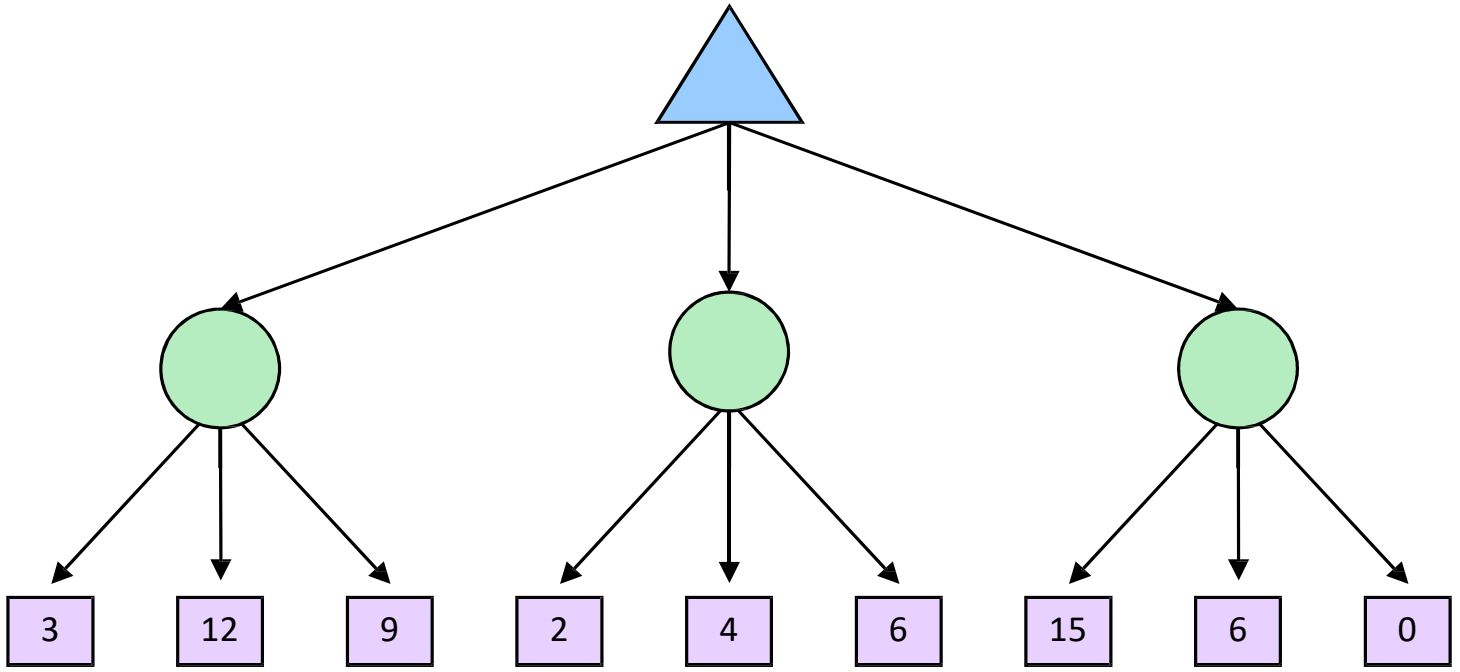
Expectiminimax Search



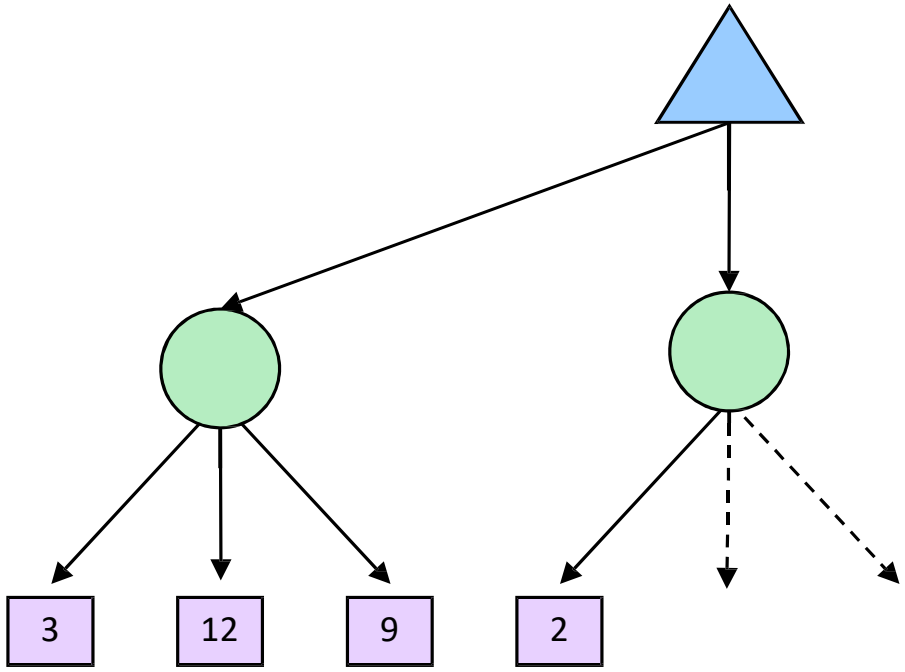
Expectiminimax Search



Expectiminimax Pruning?



Expectiminimax Pruning?

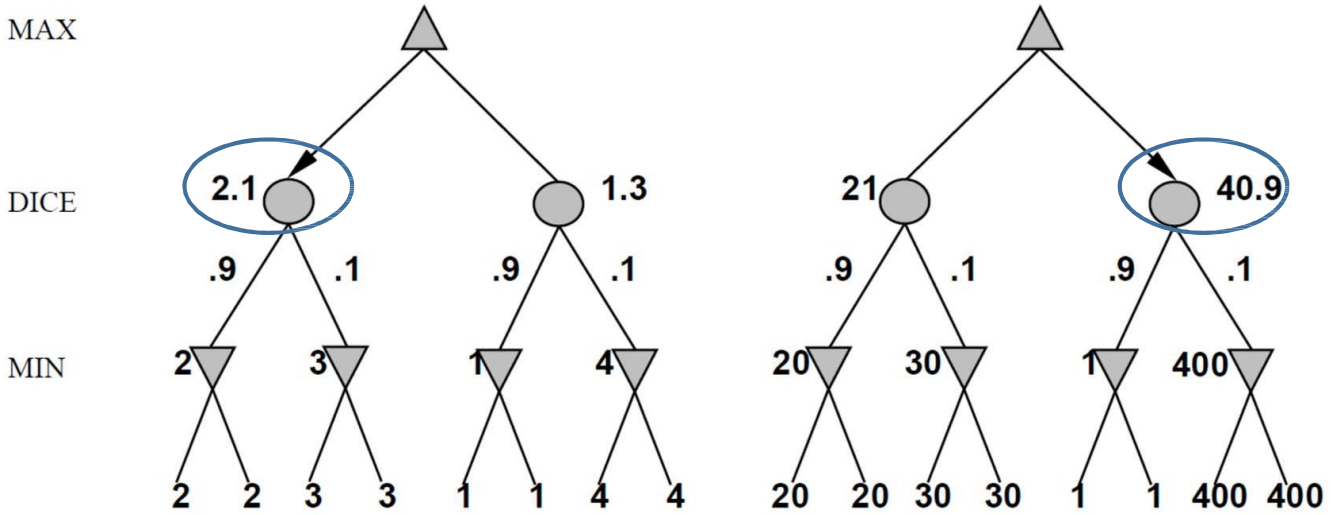


- You cannot prune chance nodes in general*
- Content of unexplored children could change expectimax value remarkably
- Thus, it is slow

- *You can prune if values are bounded (there is a question in a previous exam)

Expectiminimax Evaluation Functions

- Do the actual values matter?



- Yes!
- Evaluation(s) should be proportional to the expected payoff

Stochastic Games in Practice

- Chance nodes increase branching factor enormously
  - Backgammon, 21 possible rolls of 2 dices, ~20 legal moves
  - Depth 2 = $20 \times (21 \times 20)^3 = 1.2 \times 10^9$!!!!!

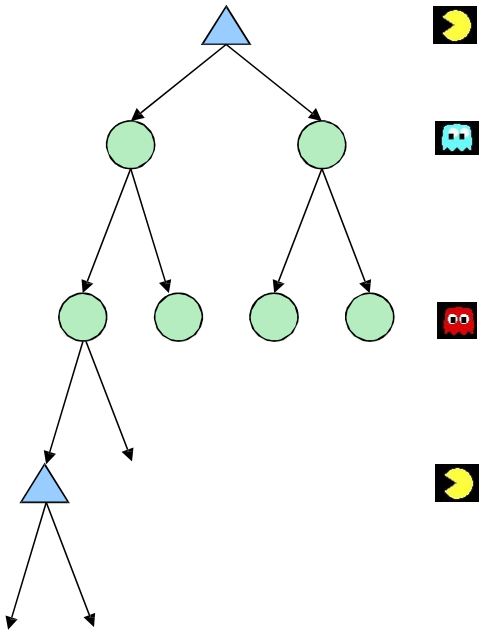
- With so much left up to chance, looking ahead is not that useful
- TDGammon (1995) plays at world class level with search depth=2 + good eval function

1st AI world champion in any game!

What probabilities to use?

- In expectimax search, we have a probabilistic model of how the opponent or the environment will behave in any state
  - Model could be a simple uniform distribution (roll a die)
  - Model could be sophisticated and require a great deal of computation
  - We have a chance node for any outcome out of our control: opponent or environment
  - The model might say that adversarial actions are likely!

- For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes



Having a probabilistic belief about another agent's action does not mean that the agent is flipping any coins!

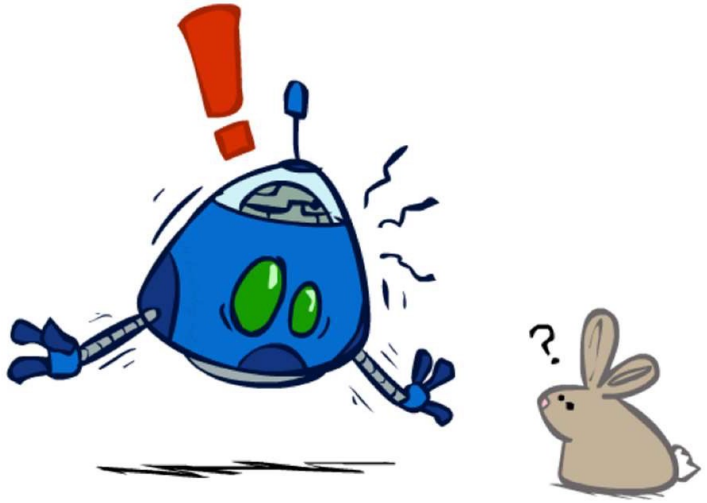
What could go wrong?

Dangerous Optimism

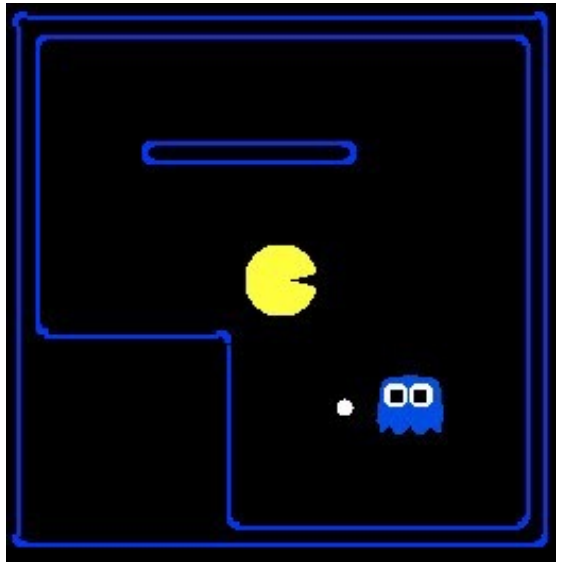
Assuming chance when the world is adversarial

Dangerous Pessimism

Assuming the worst case when it's not likely



Assumptions vs. Reality



	Adversarial Ghost	Random Ghost
Minimax Pacman	Won 5/5 Avg. Score: 483	Won 5/5 Avg. Score: 493
Expectimax Pacman	Won 1/5 Avg. Score: -303	Won 5/5 Avg. Score: 503

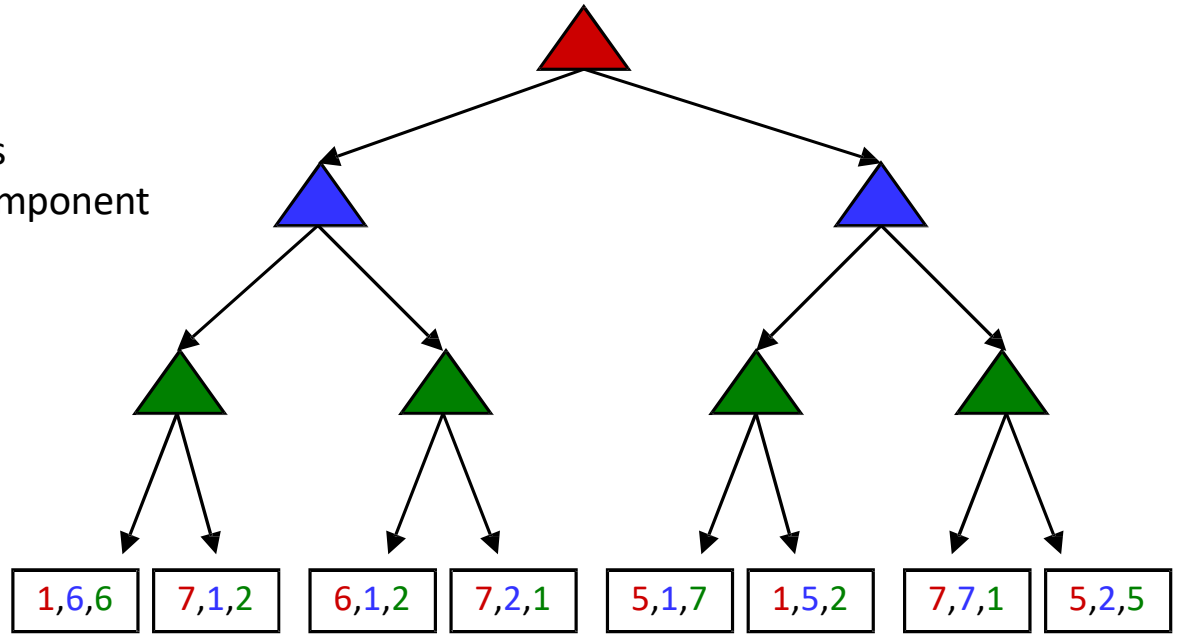
Results from playing 5 games

Pacman used depth 4 search with an eval function that avoids trouble
Ghost used depth 2 search with an eval function that seeks Pacman

Multi-Agent Utilities

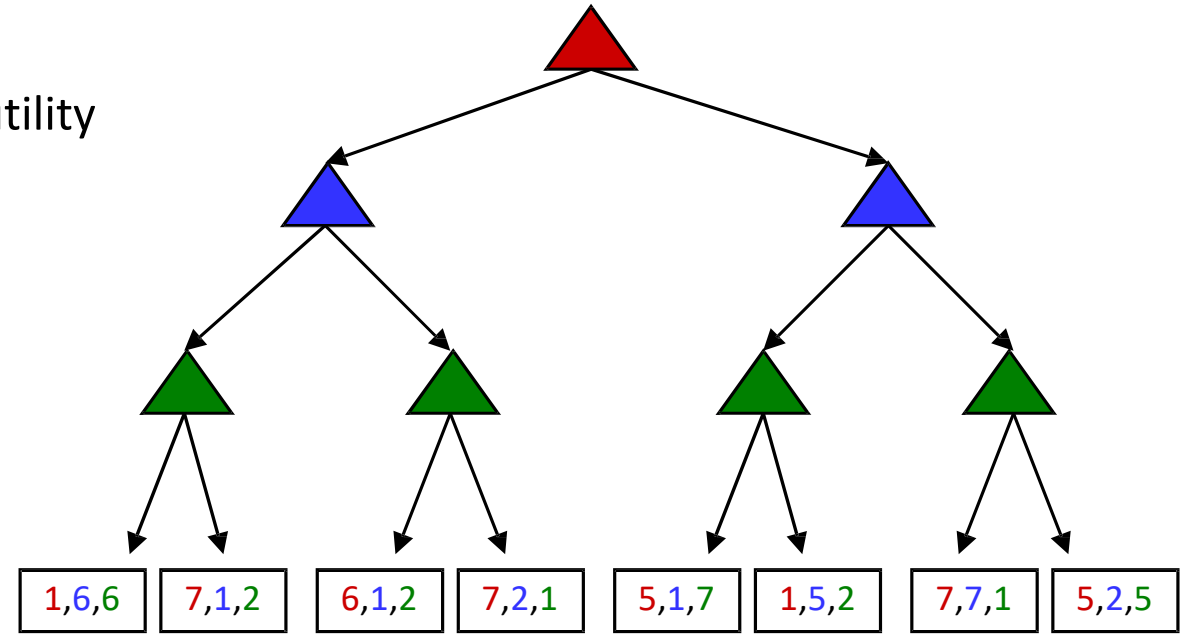
- What if the game is not zero-sum, or has multiple players?

- Generalization of minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically…



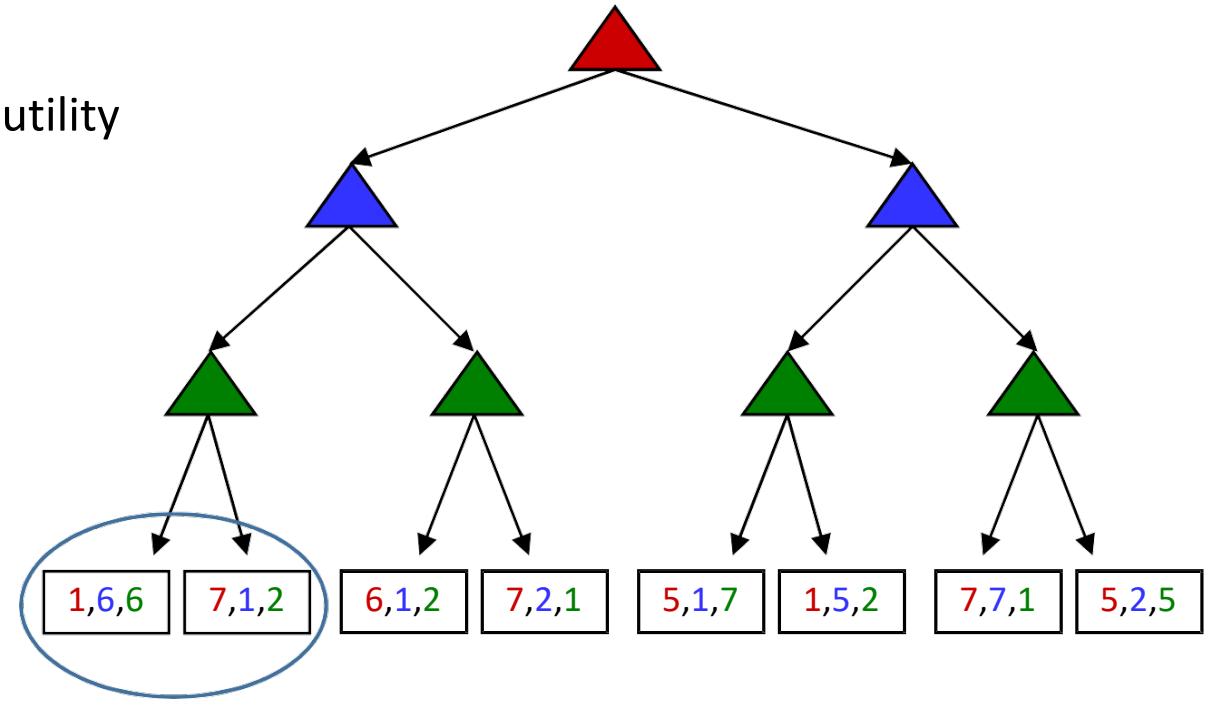
Multi-Agent Utilities

Let everybody
maximize their utility



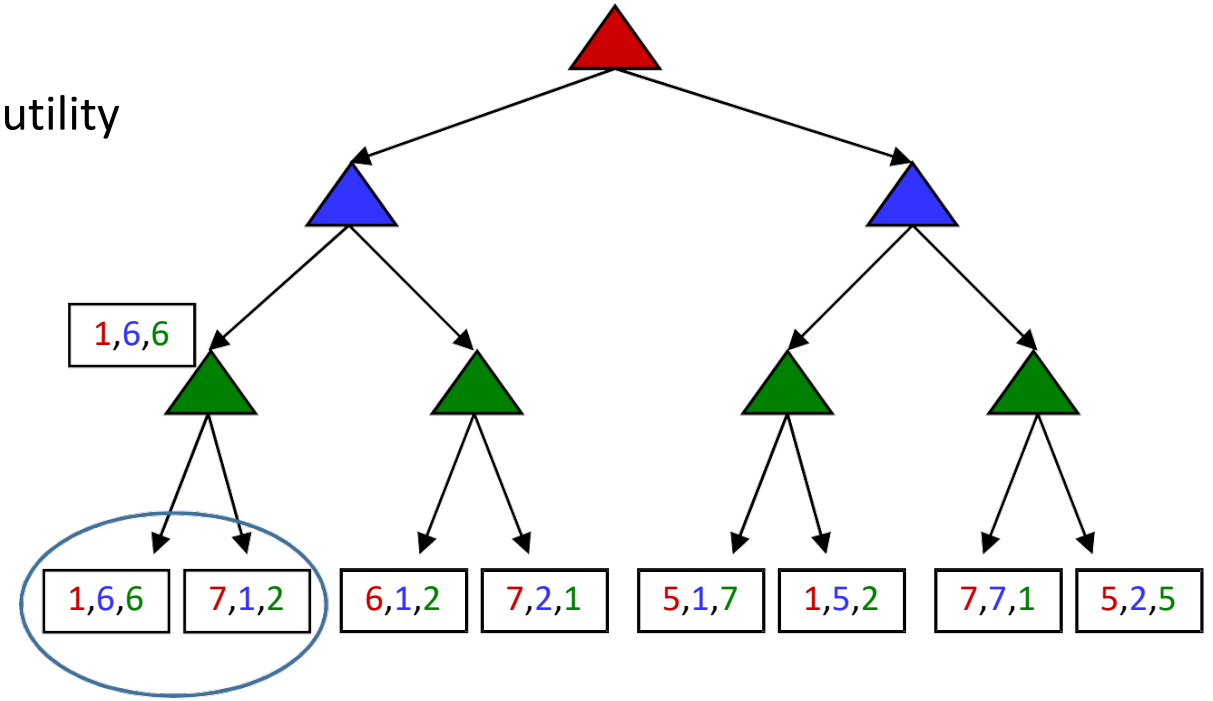
Multi-Agent Utilities

Let everybody
maximize their utility



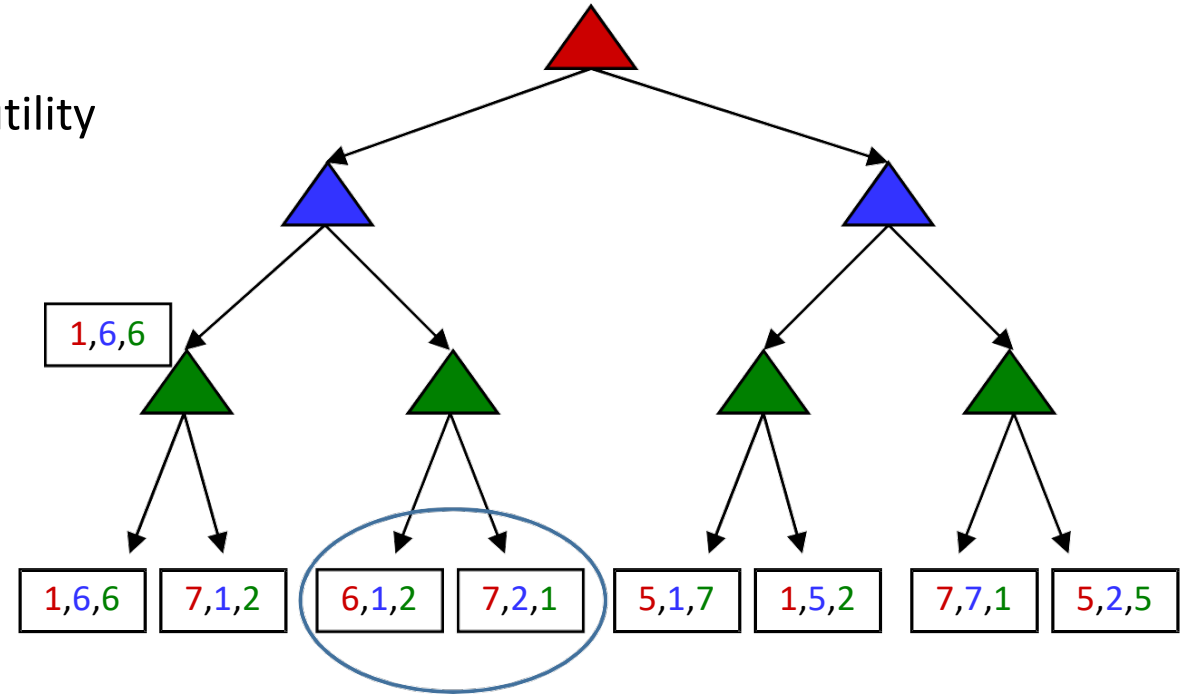
Multi-Agent Utilities

Let everybody
maximize their utility



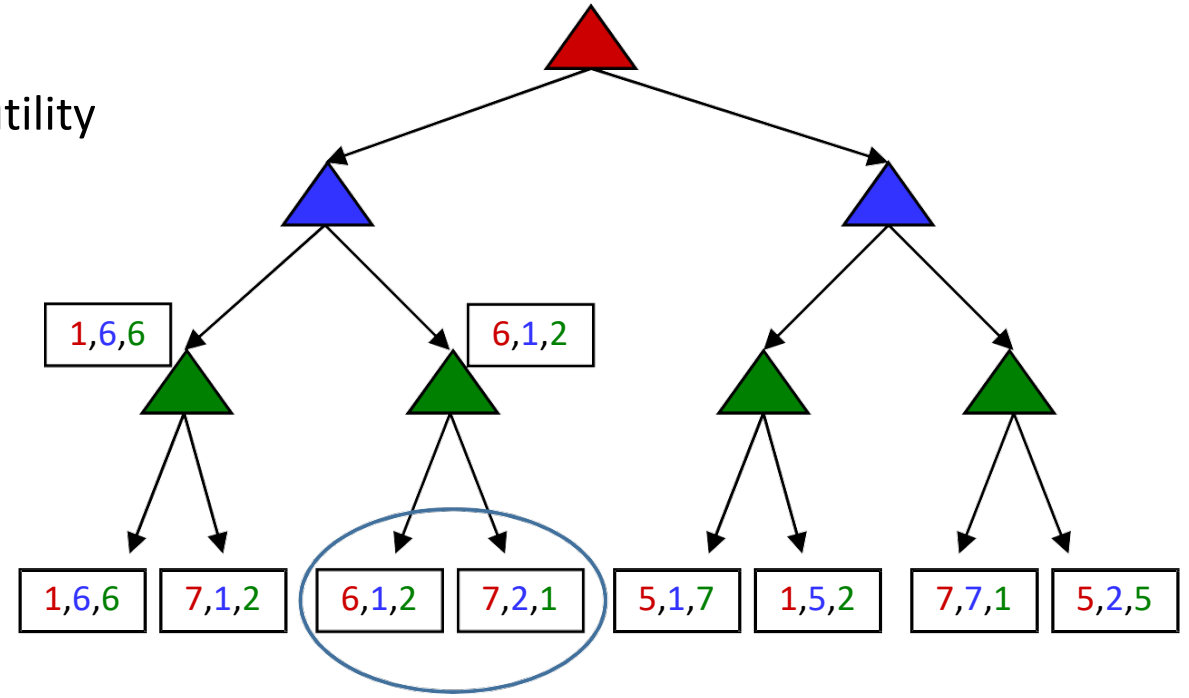
Multi-Agent Utilities

Let everybody
maximize their utility



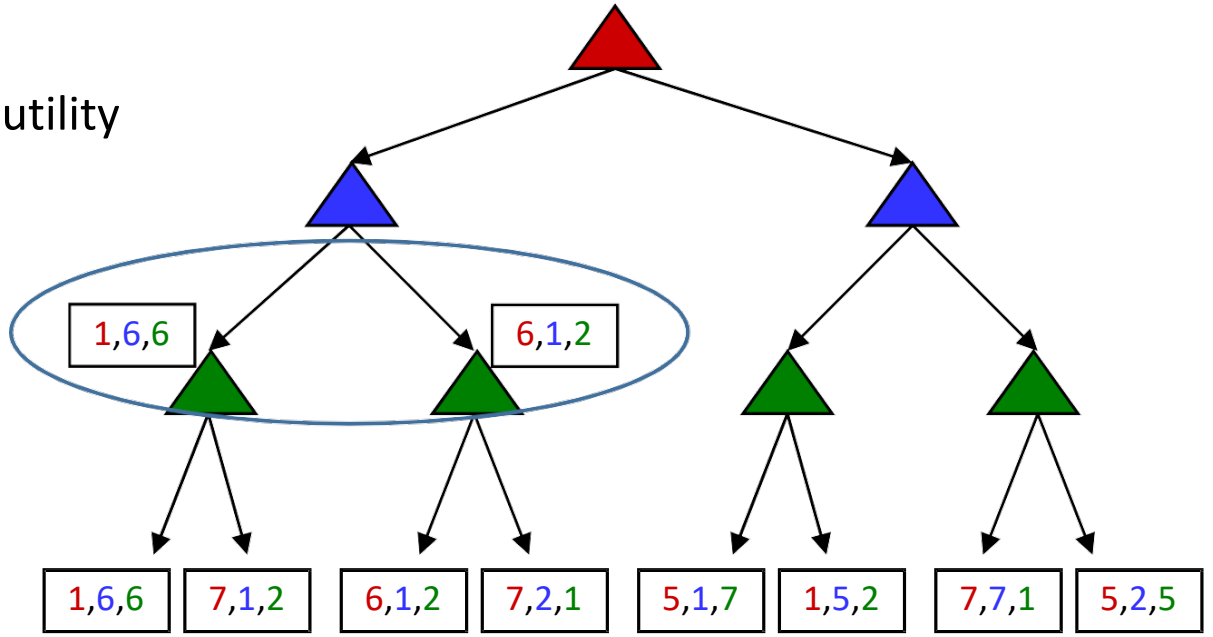
Multi-Agent Utilities

Let everybody
maximize their utility



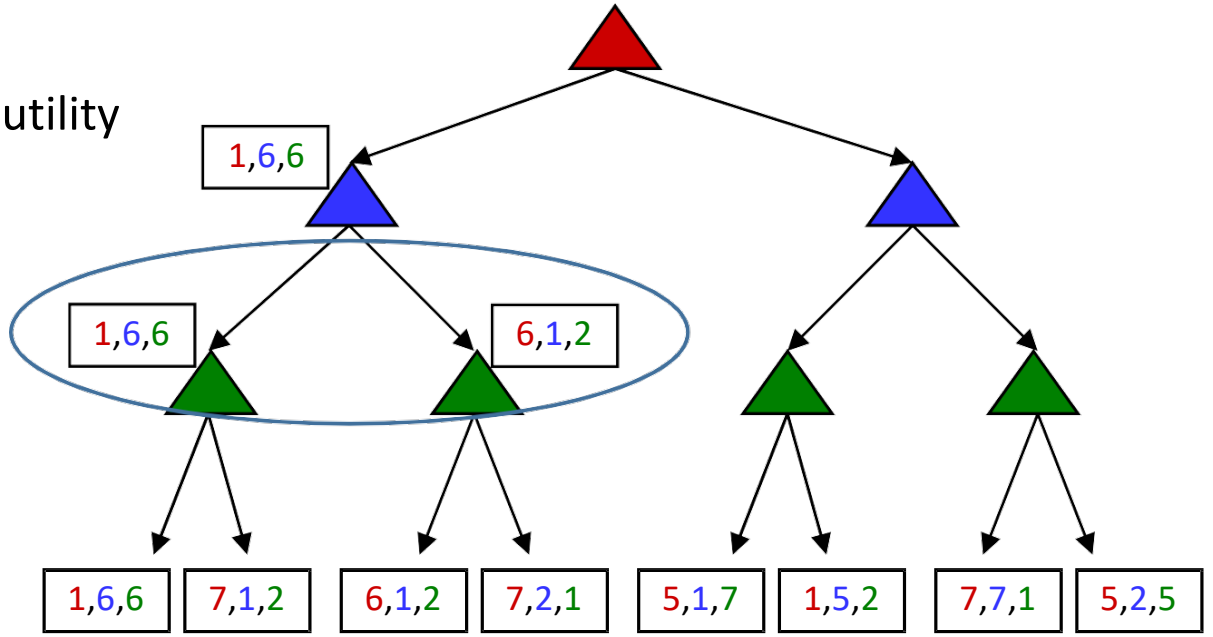
Multi-Agent Utilities

Let everybody
maximize their utility



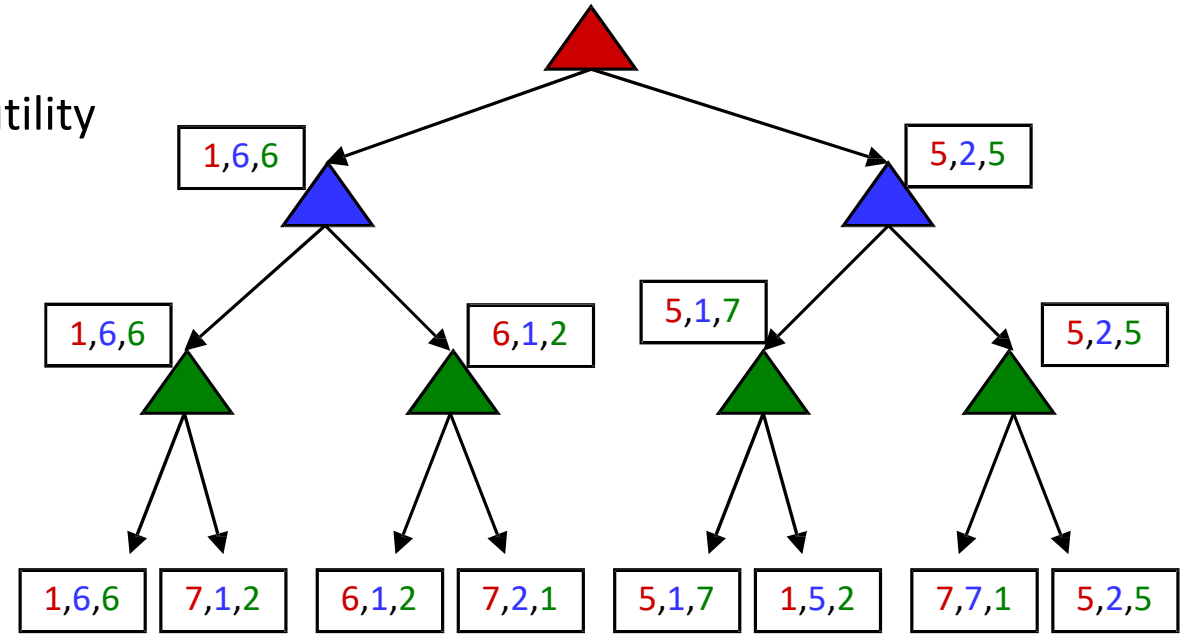
Multi-Agent Utilities

Let everybody
maximize their utility



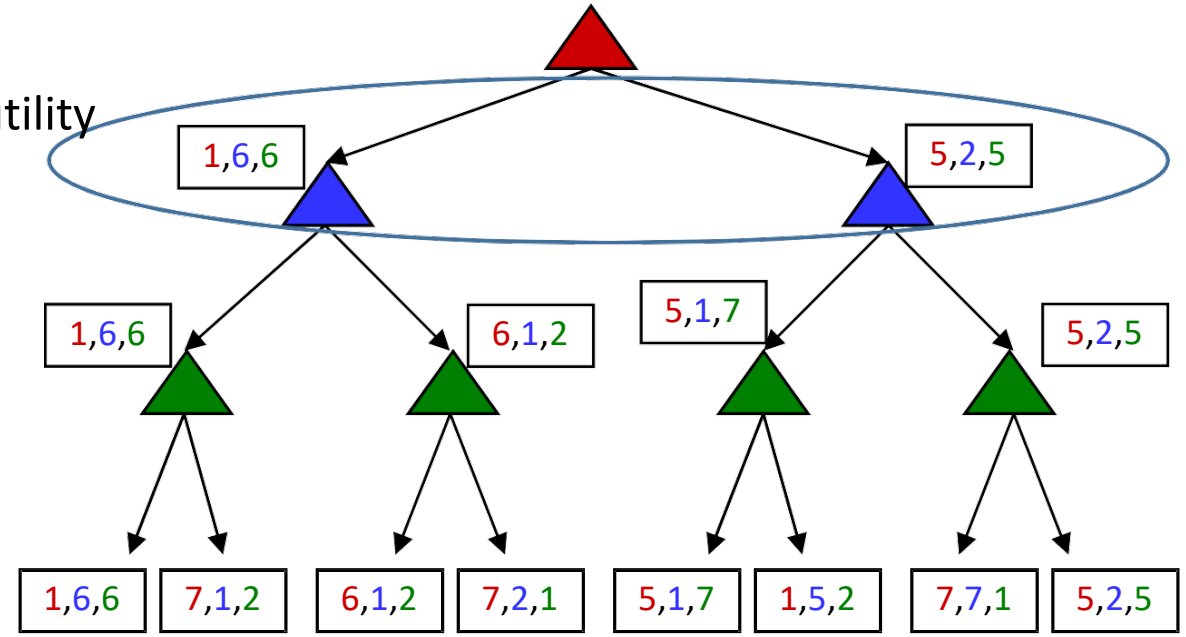
Multi-Agent Utilities

Let everybody
maximize their utility



Multi-Agent Utilities

Let everybody
maximize their utility



Multi-Agent Utilities