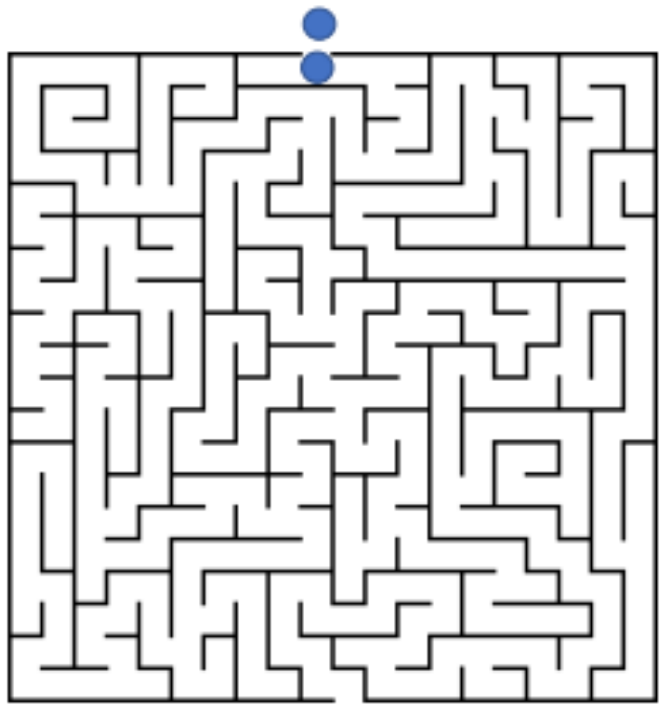
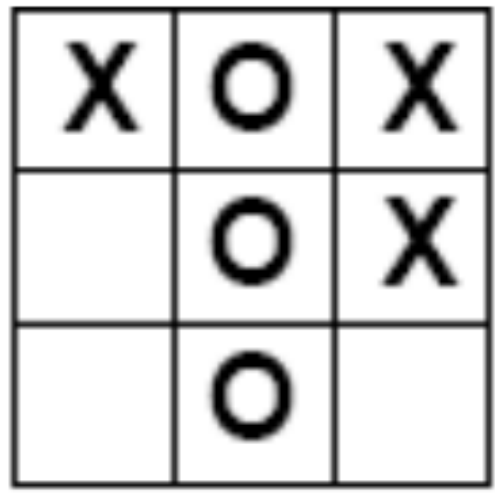
Adversarial Search

Multi-Agent Applications



Collaborative Maze Solving



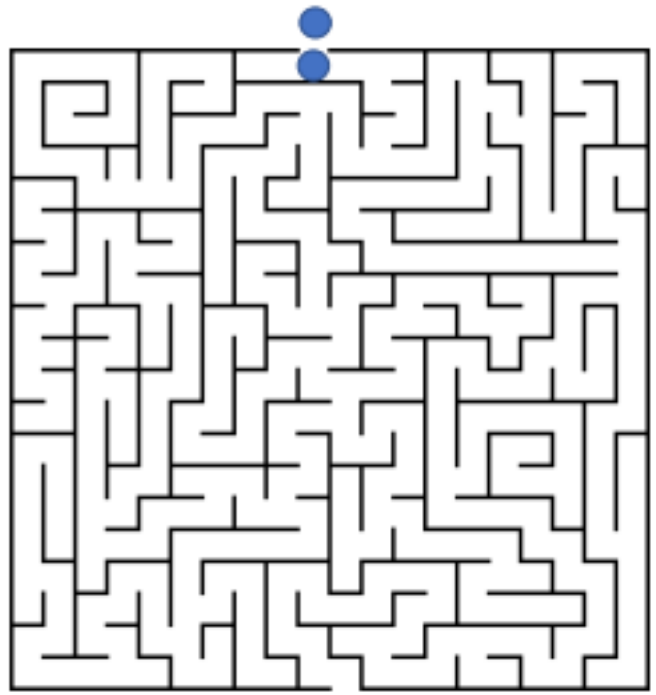
Adversarial

(Football)

Team: Collaborative
Competition: Adversarial

How could we model multi-agent problems?

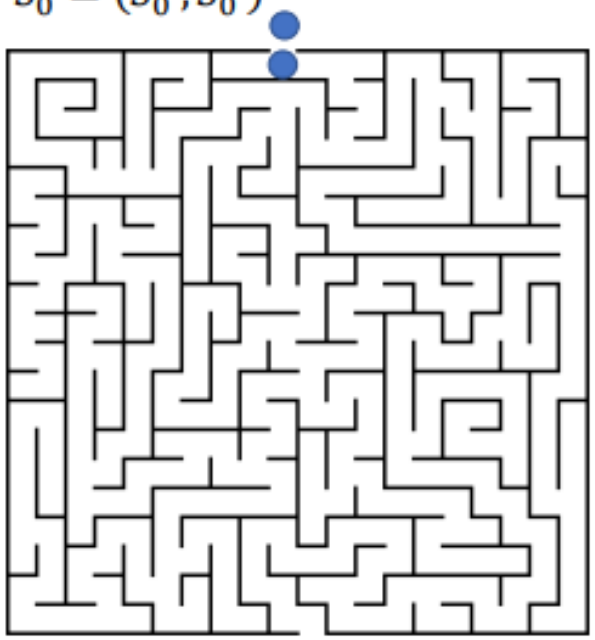
Simplest idea: each agent plans their own actions separately from other



Joint State/Action Spaces

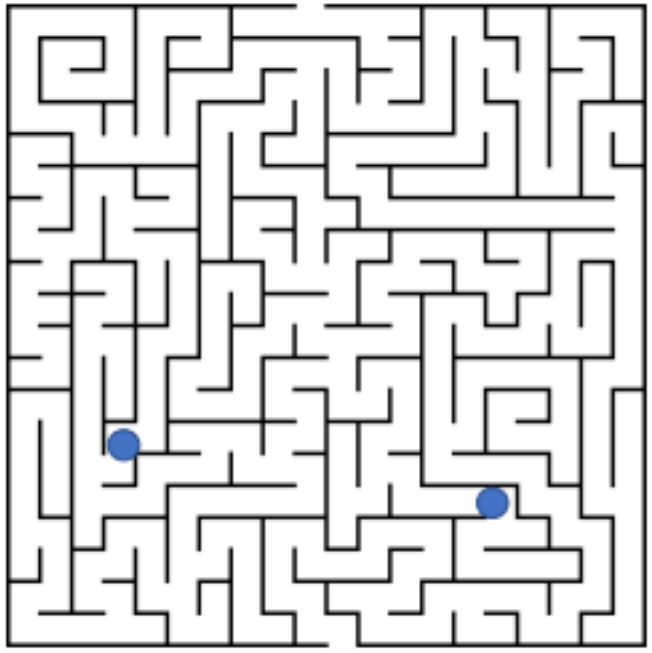
Combine the states and actions of the N agents

$S_0 = (S_0^A, S_0^B)$



- Search looks through all combinations of all agents' states and actions
- Think of one brain controlling many agents

$S_K = (S_K^A, S_K^B)$



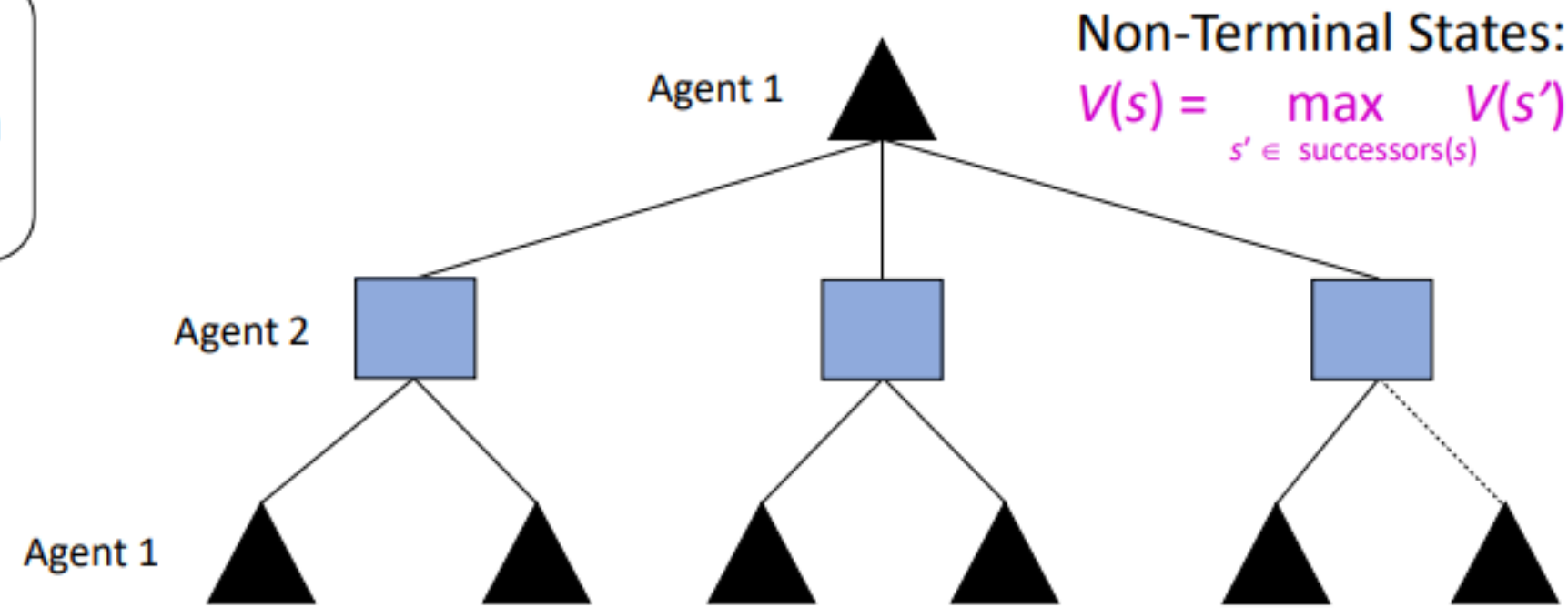
- Each agent proposes their actions and computer confirms the joint plan

Example: Autonomous driving through intersect

Alternate Searching One Agent at a Time

Search one agent's actions from a state, search the next agent's actions from those resulting states , etc…

Choose the best cascading combination of actions



Non-Terminal States:

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

Introduction

- Transition from solving search problems to adversarial scenarios.

- consider scenarios where our agents have one or more adversaries who attempt to keep them from reaching their goal(s) .

- Traditional search algorithms are inadequate for adversarial scenarios due to uncertainty in opponents' actions.
-
- Adversarial search algorithms are essential, as they enable agents to devise strategies in competitive environments by dynamically responding to opponents' moves.

Types of Games

- Many different kinds of games!
  - Axes:
    - Deterministic or stochastic?
    - One, two, or more players?
  - Discrete:
    - Games states or decisions can be mapped on discrete values.
  - Finite:
    - There are only a finite number of states and possible decisions that can be made.
  - Zero sum? if one player wins, the other loses an equal amount;
    e.g. Poker – you win what the other player lose
  - Perfect information all aspects of the state are fully observable

Which of these are: 2-player zero-sum discrete finite deterministic games of perfect information

A

C

- **Two player:** Duh!
- **Zero-sum:** In any outcome of any game, Player A's gains equal player B's losses.
- **Discrete:** All game states and decisions are discrete values.
- **Finite:** Only a finite number of states and decisions.
- **Deterministic:** No chance (no die rolls).
- **Perfect information:** Both players can see the state, and each decision is made sequentially (no simultaneous moves).

D

F

G

B

Chinese checker

ANSWER HERE!
http://etc.ch/ekXi

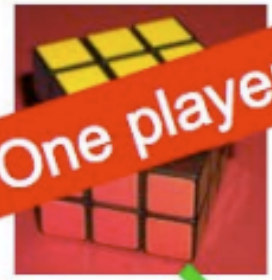
E

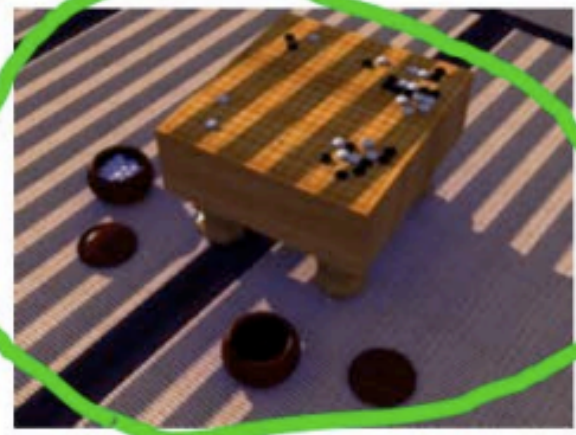
H

Monopoly

Which of these are: 2-player zero-sum discrete finite deterministic games of perfect information

- Two player: Duh!
- Zero-sum: In any outcome of any game, Player A's gains equal player B's losses.
- Discrete: All game states and decisions are discrete values.
- Finite: Only a finite number of states and decisions.
- Deterministic: No chance (no die rolls).
- Perfect information: Both players can see the state, and each decision is made sequentially (no simultaneous moves).





Game Example: Rock, Paper, Scissors

- Scissors cut paper, paper covers rock, rock smashes scissors
- Represented as a matrix: Player I chooses a row, Player II chooses a column
- Payoff to each player in each cell (Pl.I / Pl.II)
- 1: win, 0: tie, -1: loss
  so it's zero-sum

Player I \ Player II	R	P	S
R	0/0	-1/1	1/-1
P	1/-1	0/0	-1/1
S	-1/1	1/-1	0/0

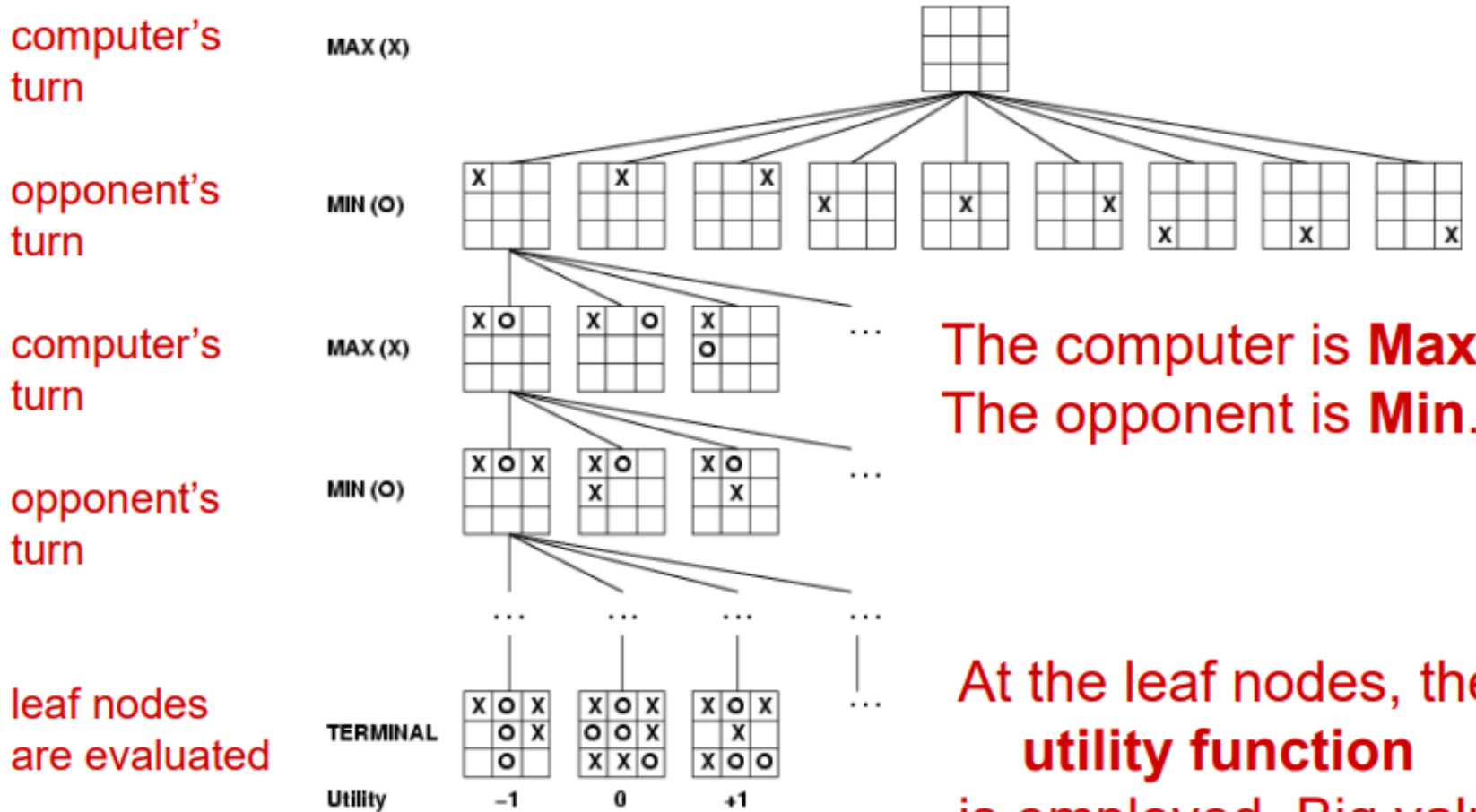
Standard Games

- Standard games are deterministic, observable, two-player, turn-taking, zero-sum
- Game formulation:
  - Initial state: s_0
  - Players: Player(s) indicates whose move it is
  - Actions: Actions(s) for player on move
  - Transition model: Result(s,a)
  - Terminal test: Terminal-Test(s)
  - Terminal values: Utility(s,p) for player p
    - Or just Utility(s) for player making the decision at

- States:
  - board configurations
- Initial state:
  - the board position and which player will move
- Successor function:
  - returns list of (move, state) pairs, each indicating a legal move and the resulting state
- Terminal test:
  - determines when the game is over
- Utility function:
  - gives a numeric value in terminal states

  (e.g., -1, 0, +1 for loss, tie, win)

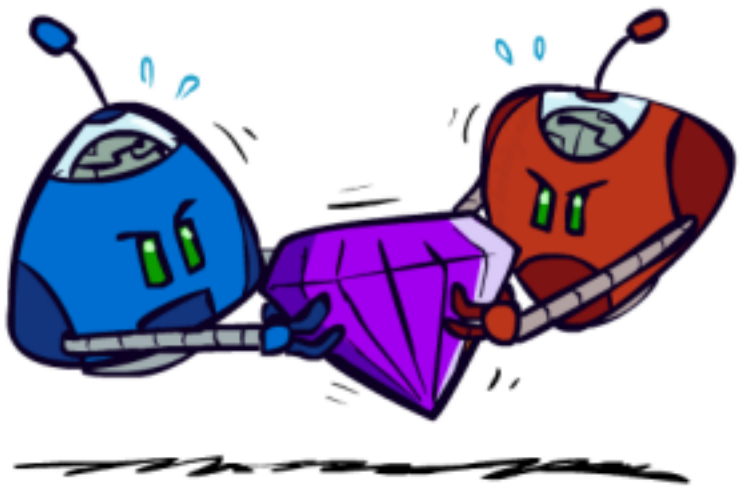
Game Tree (2-player, Deterministic, Turns)



The computer is **Max**.
The opponent is **Min**.

At the leaf nodes, the **utility function** is employed. Big value means good, small is bad.

Zero-Sum Games



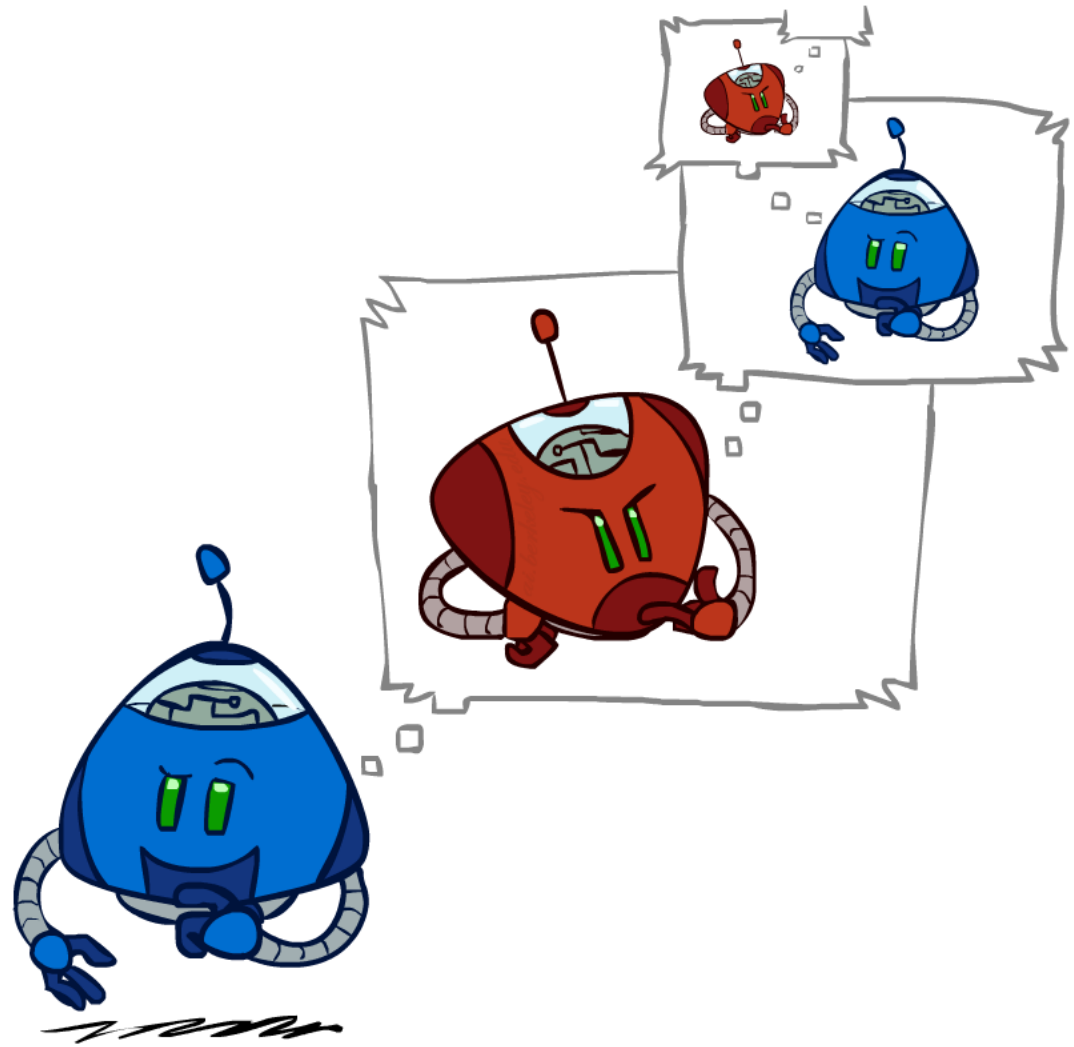
Zero-Sum Games

- Agents have ***opposite*** utilities
- Pure competition:
  - One ***maximizes***, the other ***minimizes***

General Games

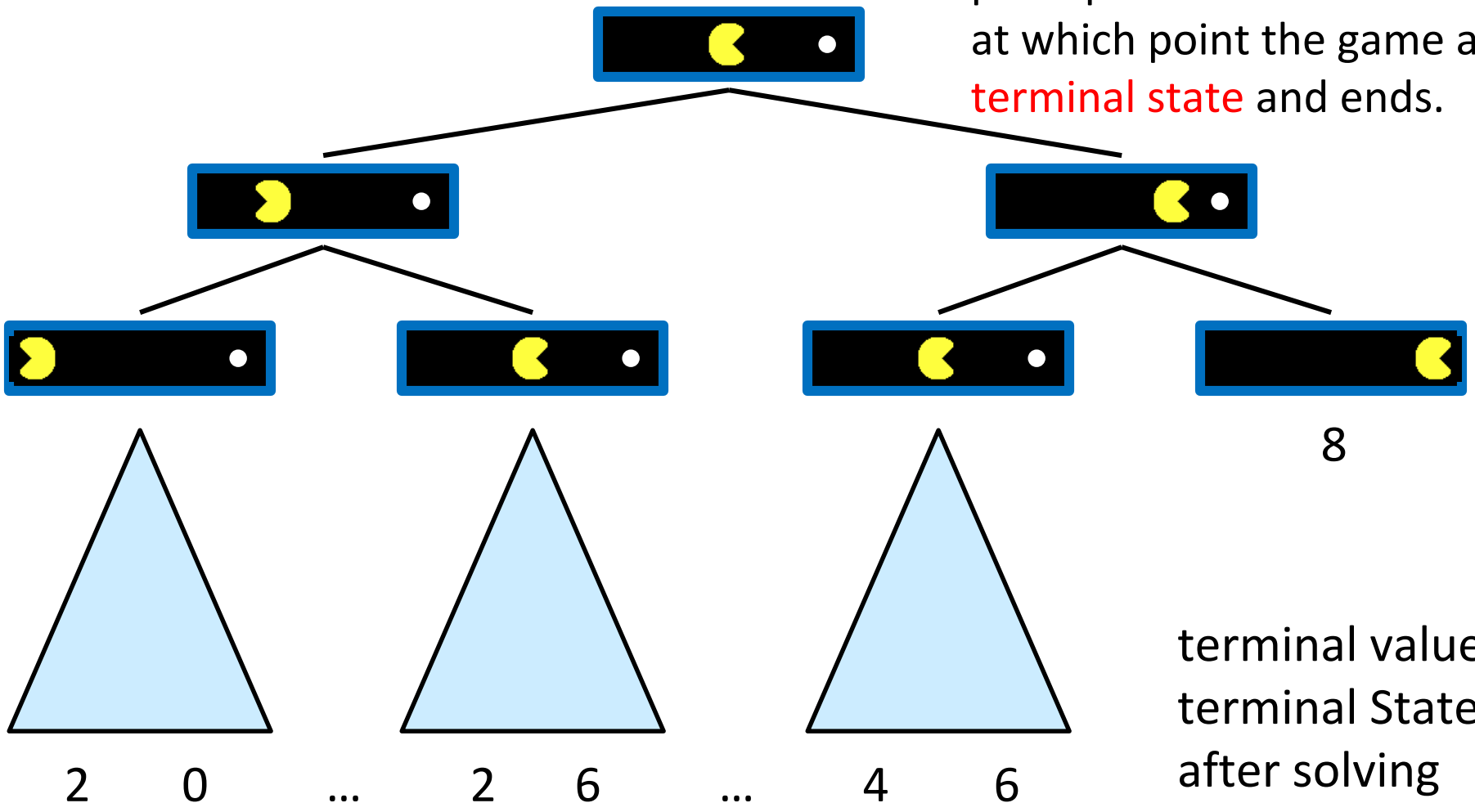
- Agents have ***independent*** utilities
- Cooperation, indifference, competition, shifting alliances, and more are all possible

ADVERSARIAL SEARCH



Single-Agent Trees

Pacman starts with 10 points and loses 1 point per move until he eats the pellet, at which point the game arrives at a terminal state and ends.



terminal values for terminal States after solving subtrees

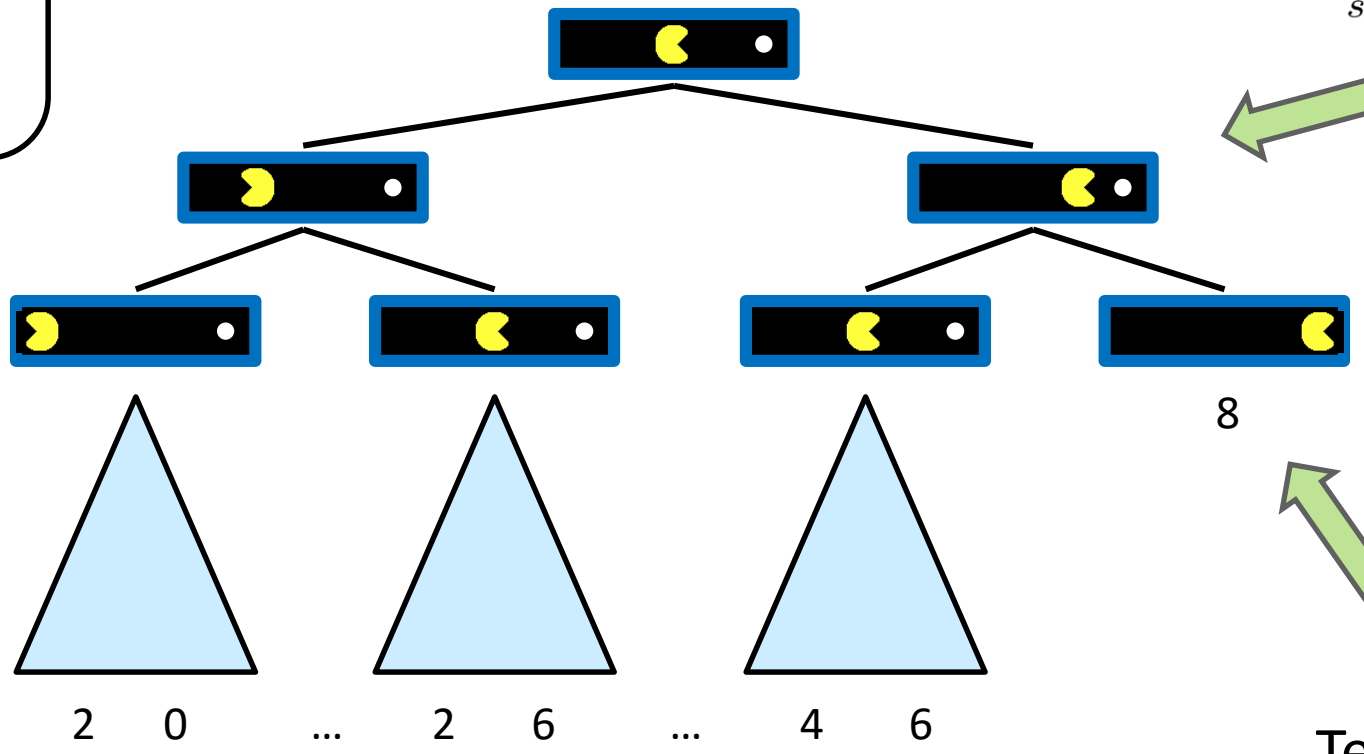
Value of a State

If state is non terminal , value of this state is maximum value among children nodes

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

Non-Terminal States:

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



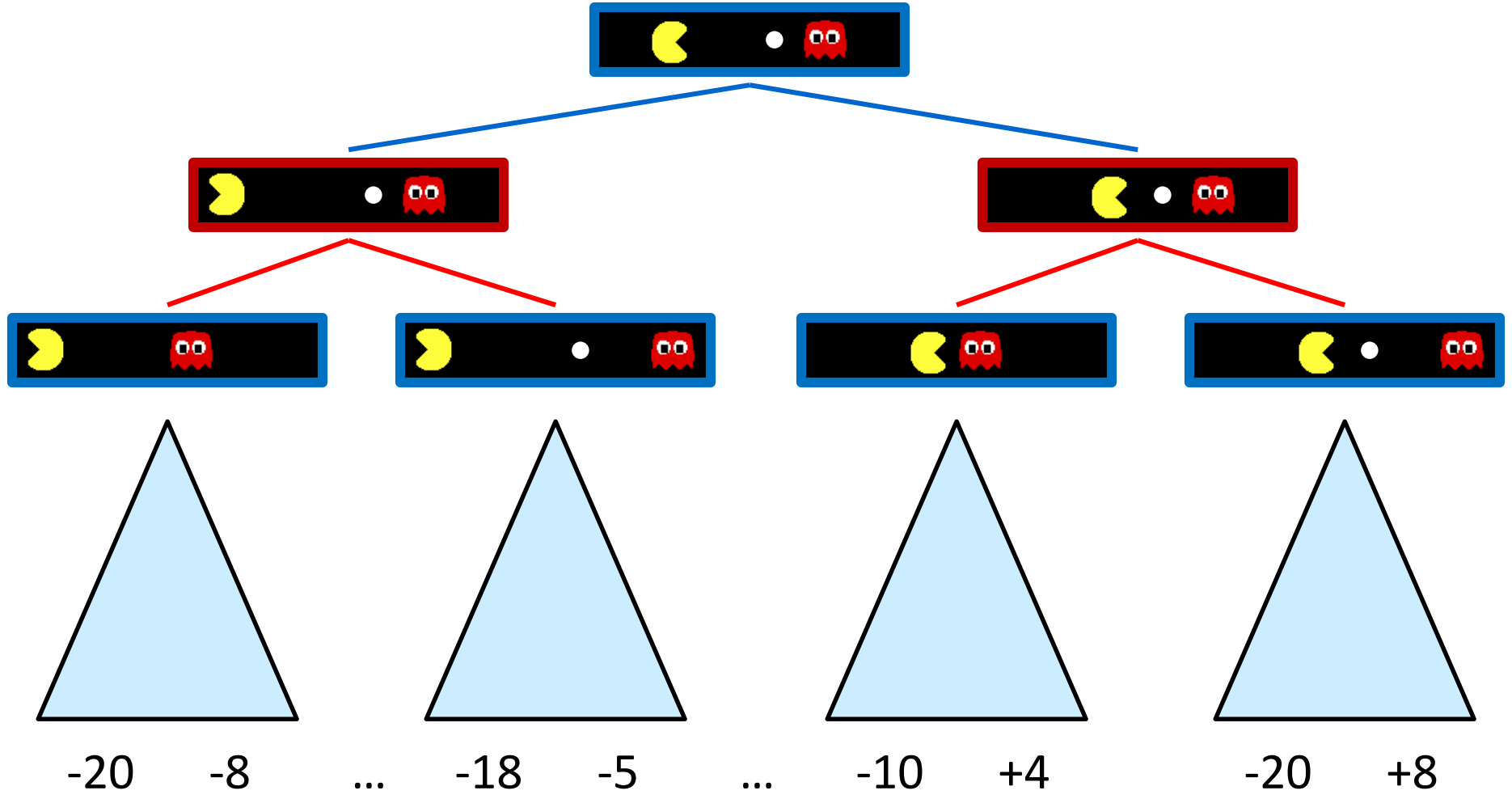
Terminal States:

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

Pacman is optimal agent it will try to maximize its game i.e maximum achievable by the children

Adversarial Game Trees

first turn pacman



utility at terminal state

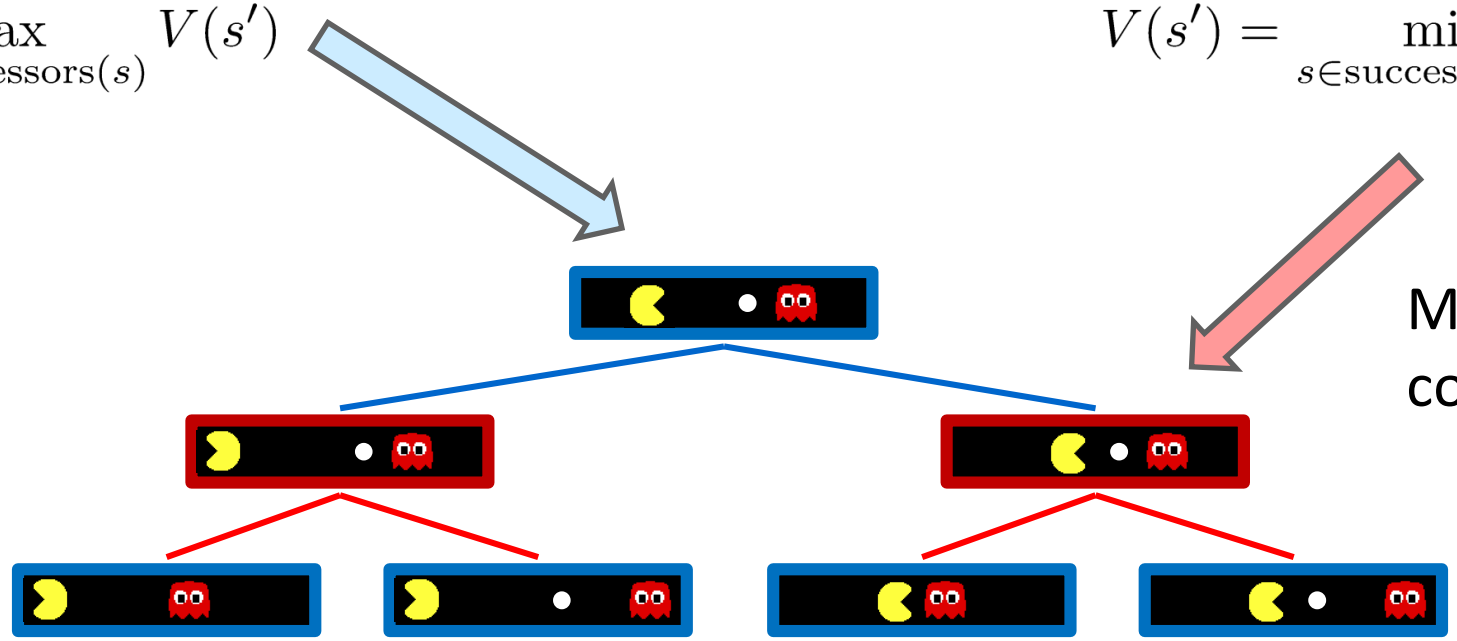
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

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



Min as this state is controlled by Ghost

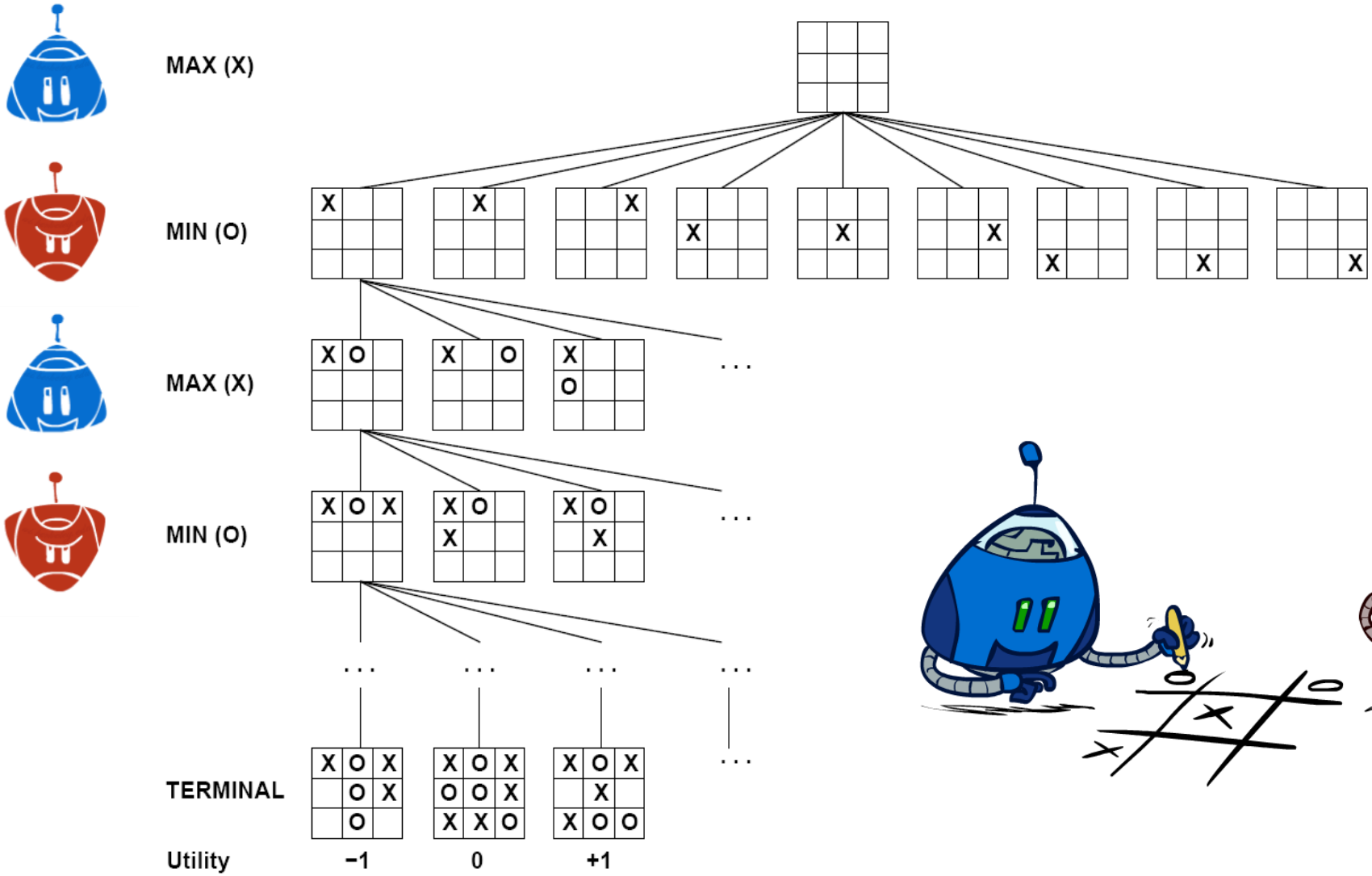
Max as this state is controlled by Pacman

-8 -5 -10 +8

Terminal States:

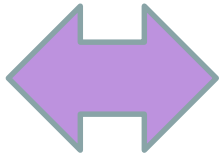
$$V(s) = \text{known}$$

Tic-Tac-Toe Game Tree



Minimax Implementation

```
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
```

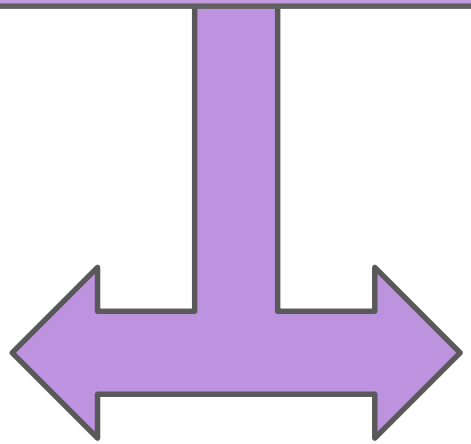
$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

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

Children controlled by opponent

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)
```



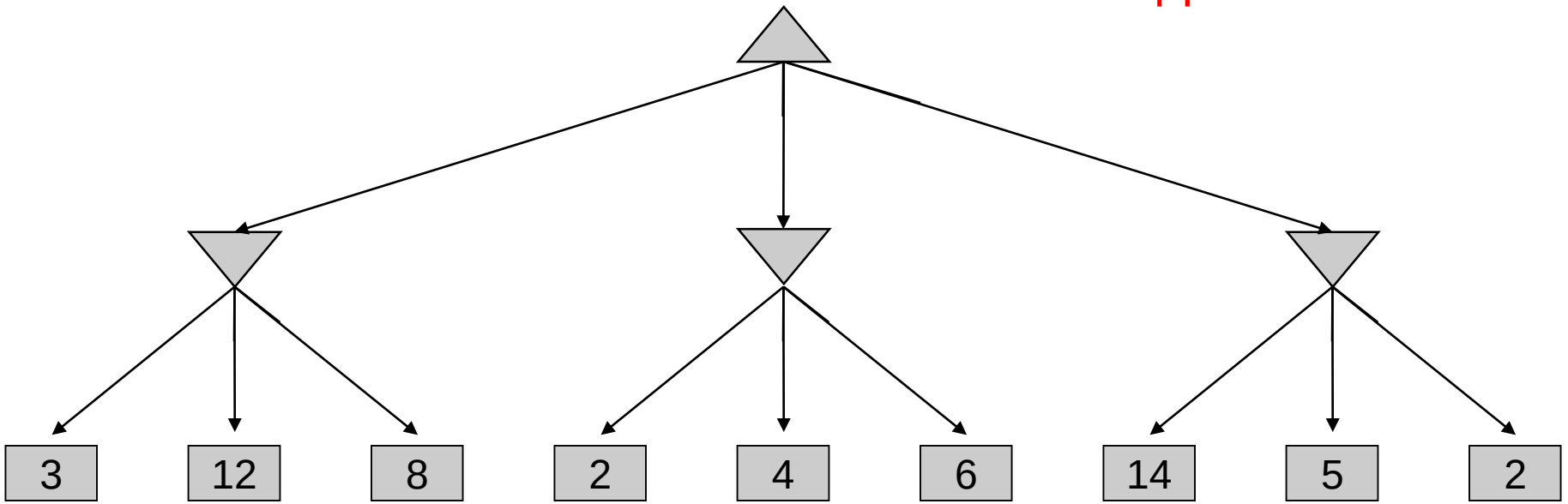
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

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

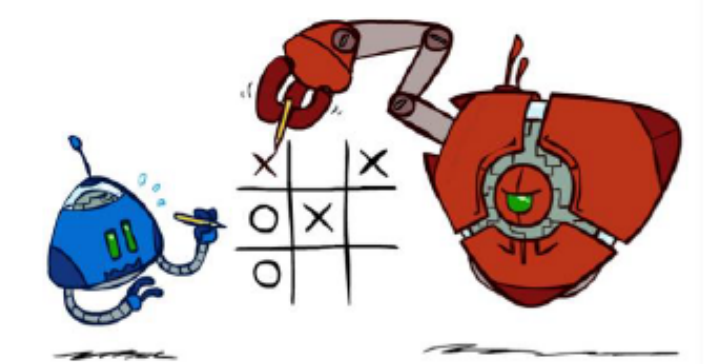
Minimax Example (at Running time)

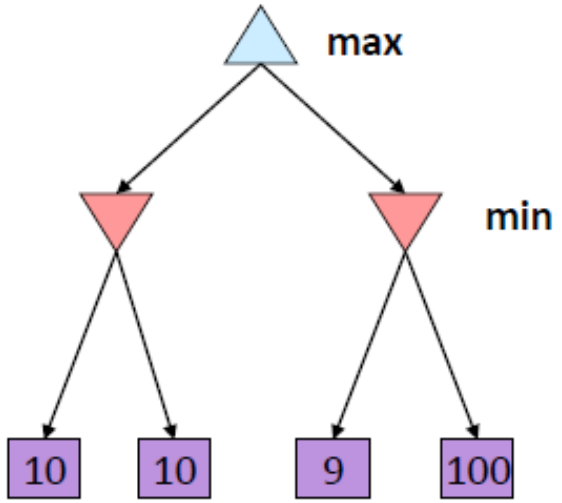
What is the minimax value at the root?

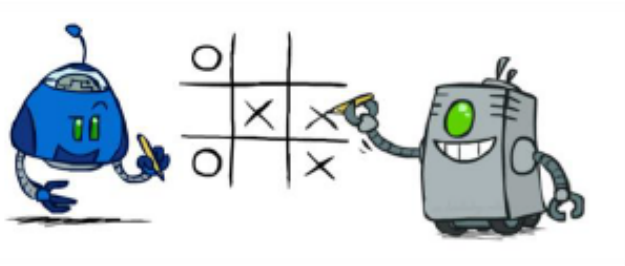
Works as DFS and then Backtrack or bottom up approach



Minimax Properties







Optimal against a perfect player. Otherwise?

Agent might be tempted to get 100. is it possible to get 100 .

Actions of Players ? if opponent not perfect

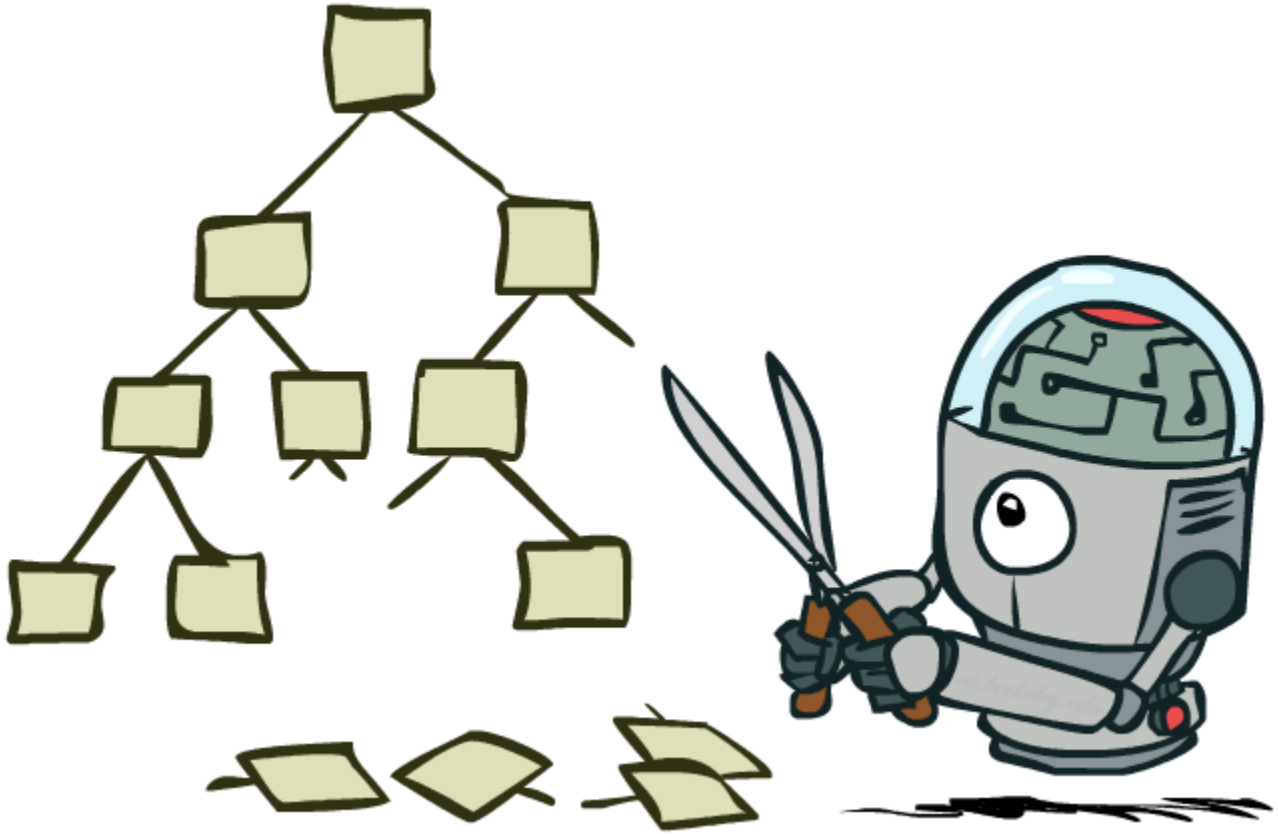
Minimax Efficiency

- How efficient is minimax?
  - Minimax seems just about perfect - it's simple, it's optimal
  - Execution Just like DFS (exhaustive)
  - Time: $O(b^m)$
  - Space: O(bm)
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?

Limitation of the minimax Algorithm:

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide.

To help mitigate this issue, minimax has an optimization - alpha-beta pruning.

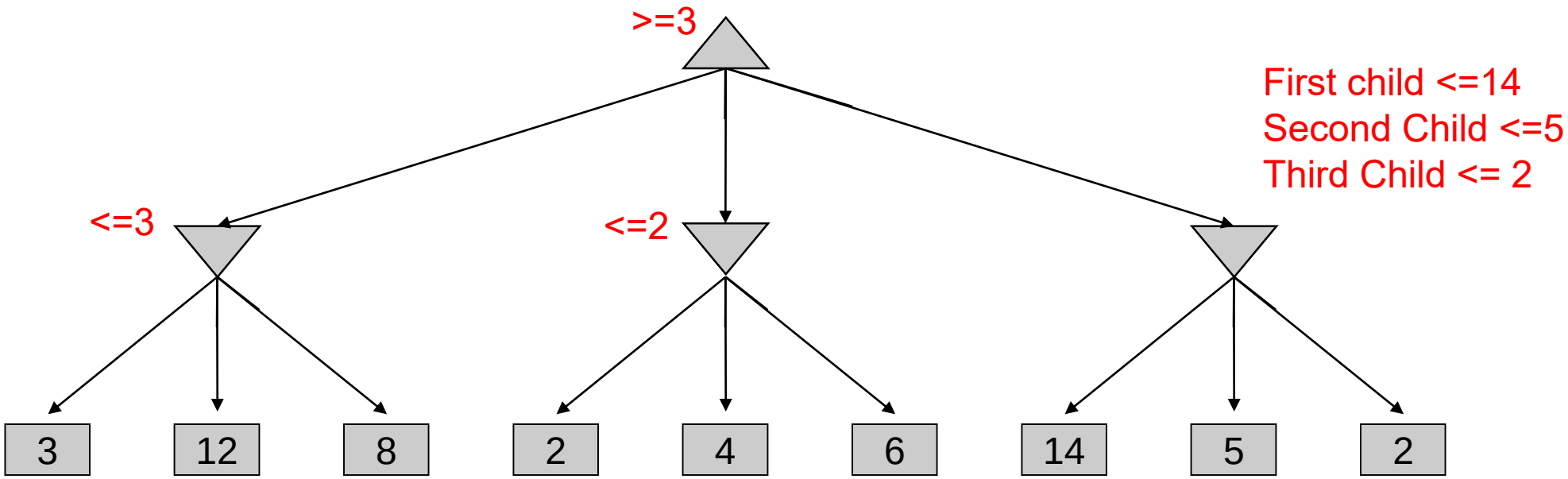


GAME TREE PRUNING

If we are not able to explore the entire tree due to resource limitation . We try to Prune it or cut parts of tree and make algo efficient

Minimax Example

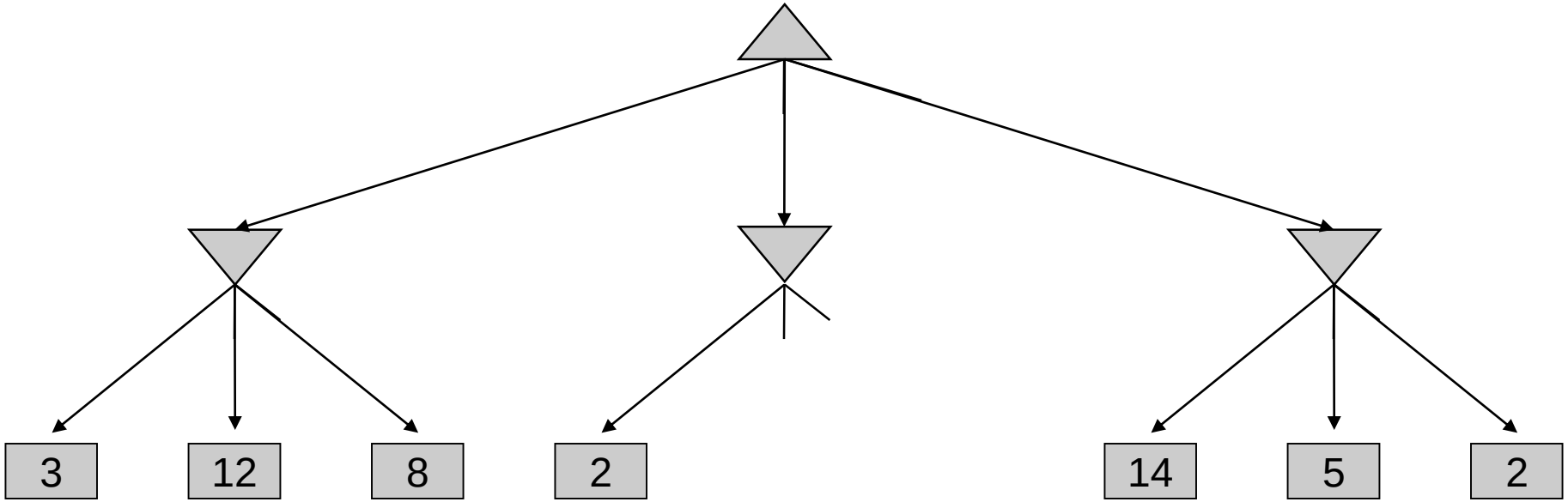
Start With DFS



First child <=14
Second Child <=5
Third Child <= 2

Is three any branches of three that we can ignore and still get same value of tree
or
Any value here on leaves of tree if we Change it will never effect on value of root

Minimax Pruning



Alpha Beta Procedure

- **Idea:**
  - Do depth first search to generate partial game tree,
  - Give static evaluation function to leaves,
  - Compute bound on internal nodes.
- **α, β bounds:**
  - α value for max node means that max real value is at least α.
  - β for min node means that min can guarantee a value no more than β.

Two types of pruning (cuts):

- pruning of max nodes (α-cuts)
- pruning of min nodes (β-cuts)

Alpha-Beta Pruning

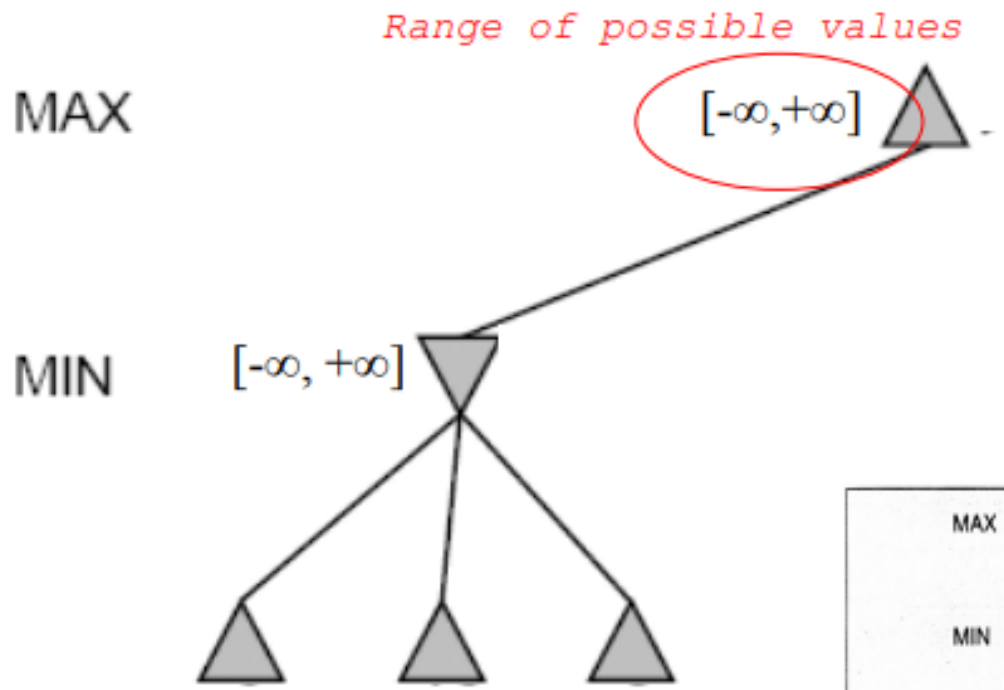
- Each node passes the current value of alpha and beta to each child node evaluated.
- Children nodes update their copy of alpha and beta, but do not pass alpha or beta back up the tree.
- Minimizing nodes return beta as the value of the node.
- Maximizing nodes return alpha as the value of the node.

Note:

1. At the root of the search tree, alpha is set to $-\infty$ and beta is set to $+\infty$.
2. If alpha > beta, stop evaluating children

Alpha-Beta Example

Do DF-search until first leaf



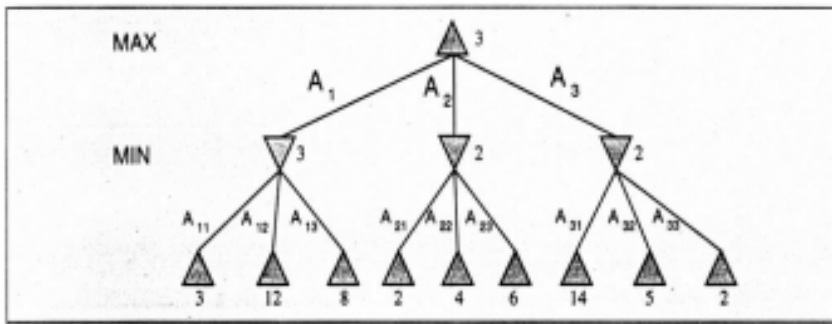
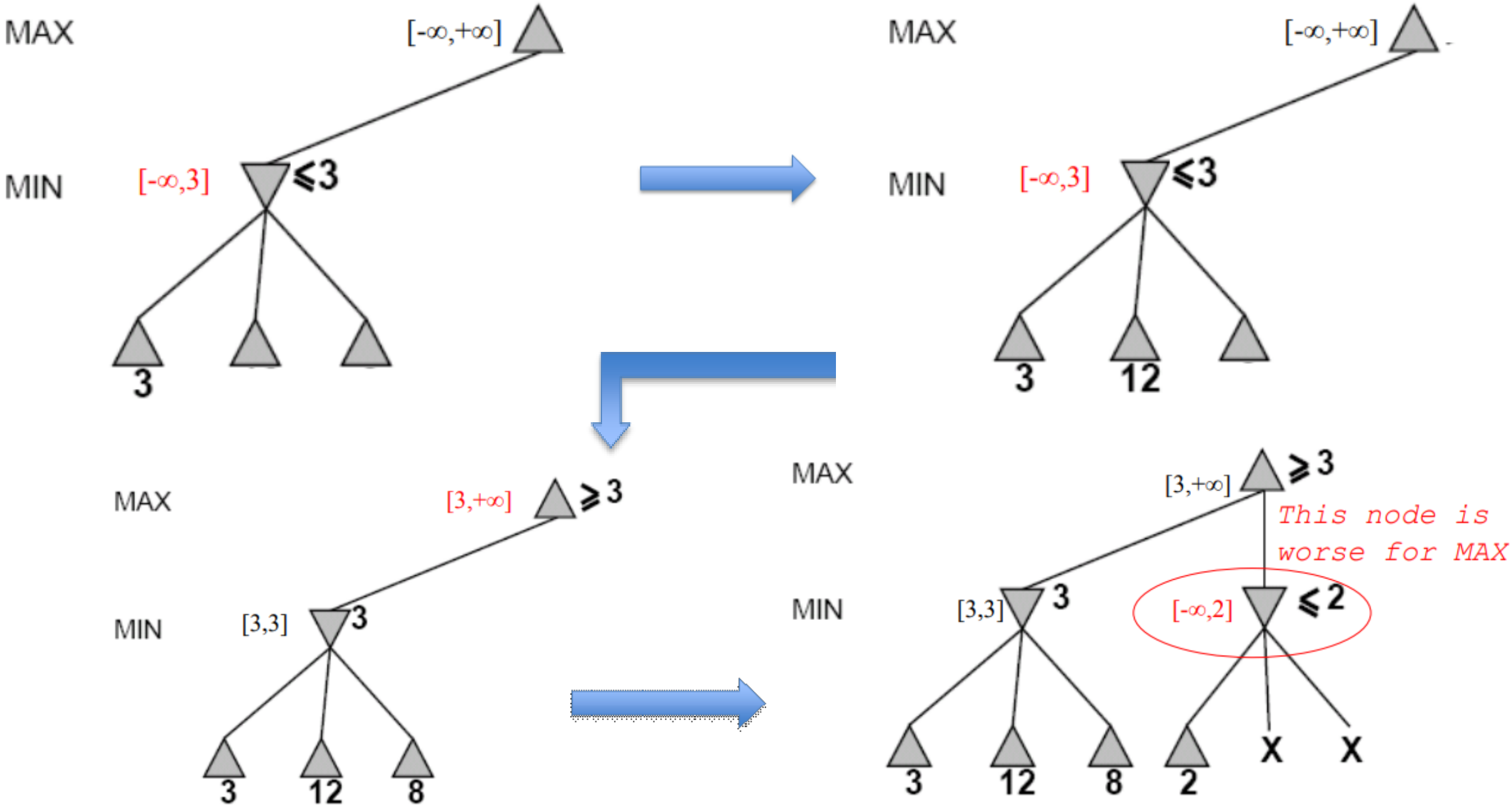
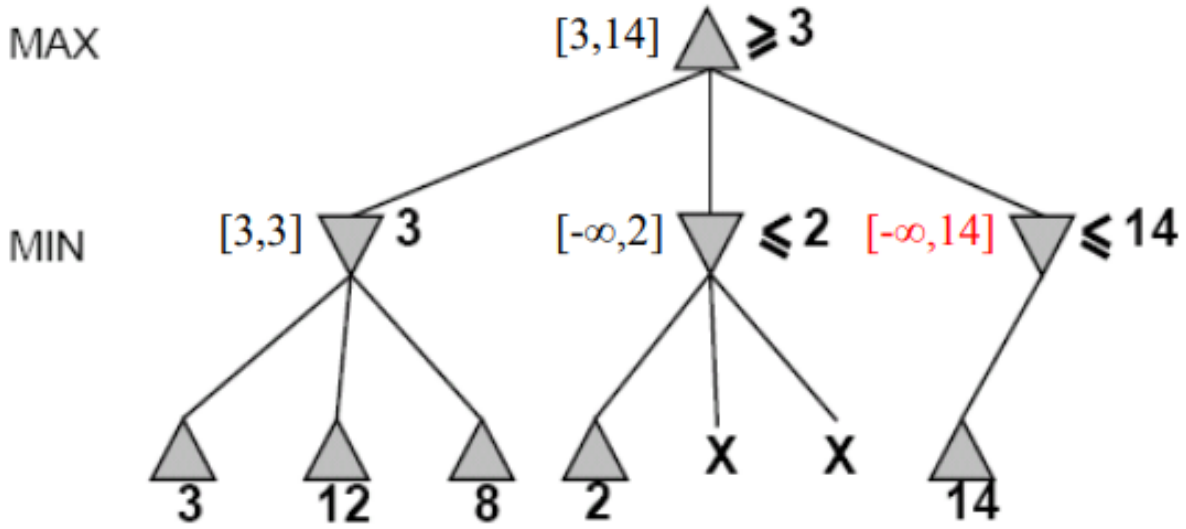


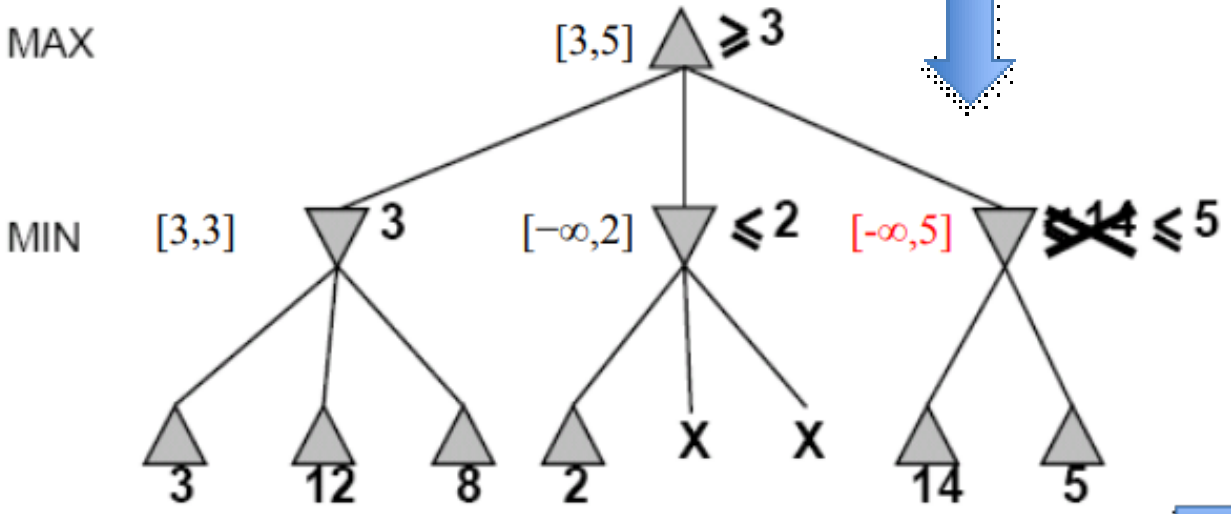
Figure 5.2 A two-ply game tree as generated by the minimax algorithm. The △ nodes are moves by MAX and the ▽ nodes are moves by MIN. The terminal nodes show the utility value for MAX computed by the utility function (i.e., by the rules of the game), whereas the utilities of the other nodes are computed by the minimax algorithm from the utilities of their successors. MAX's best move is A_1, and MIN's best reply is A_{11}.

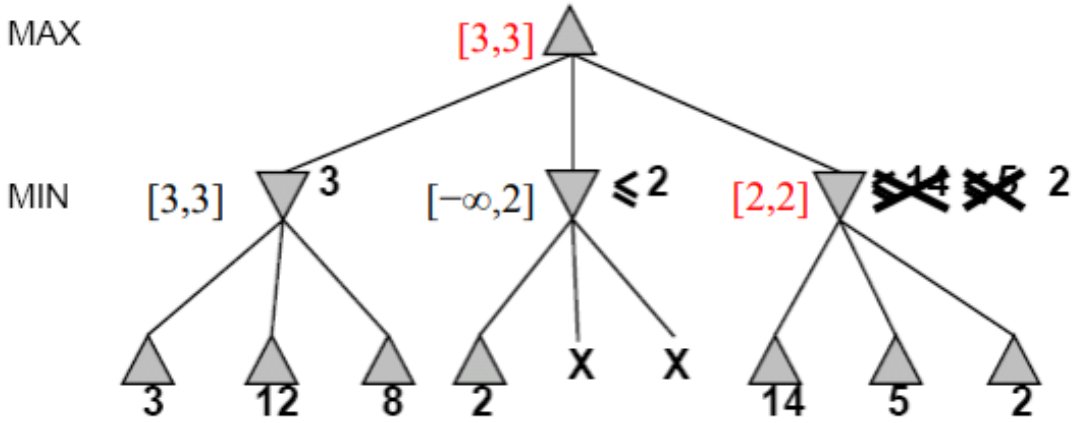
Alpha-Beta Example (continued)



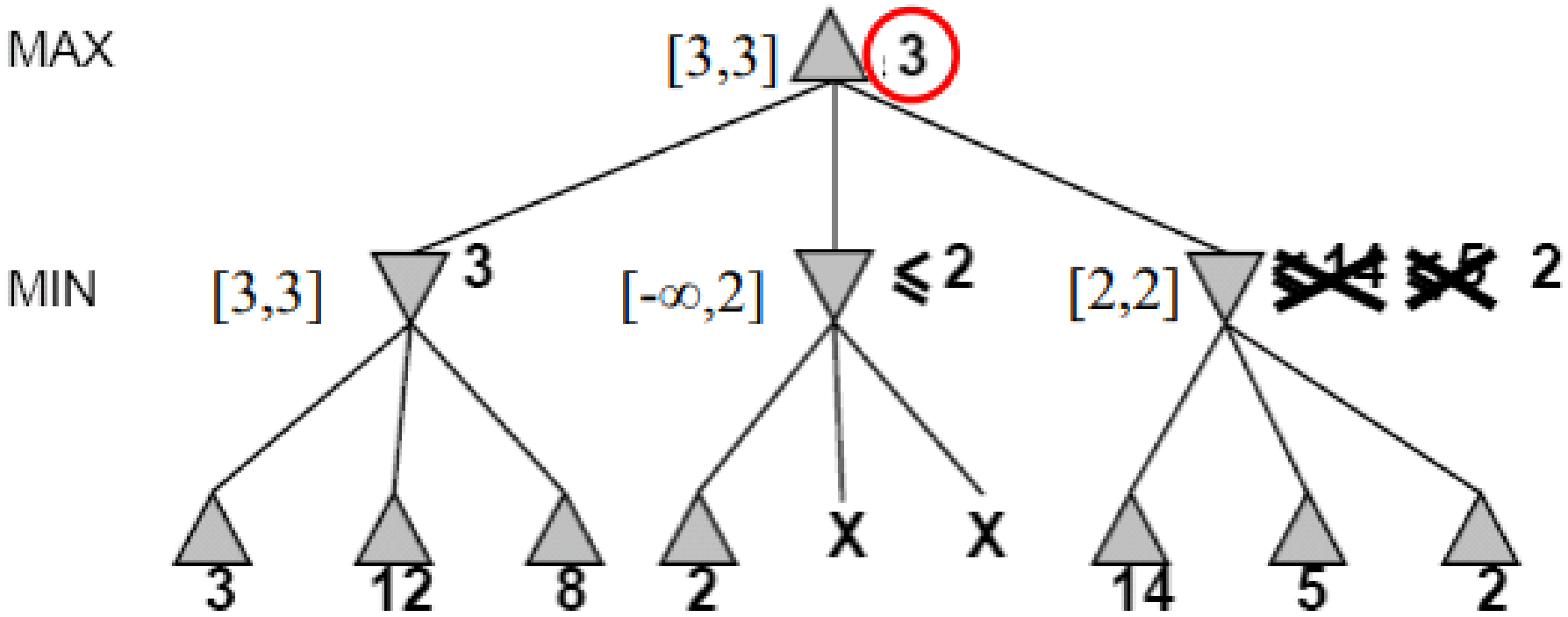
Alpha-Beta Example (continued)



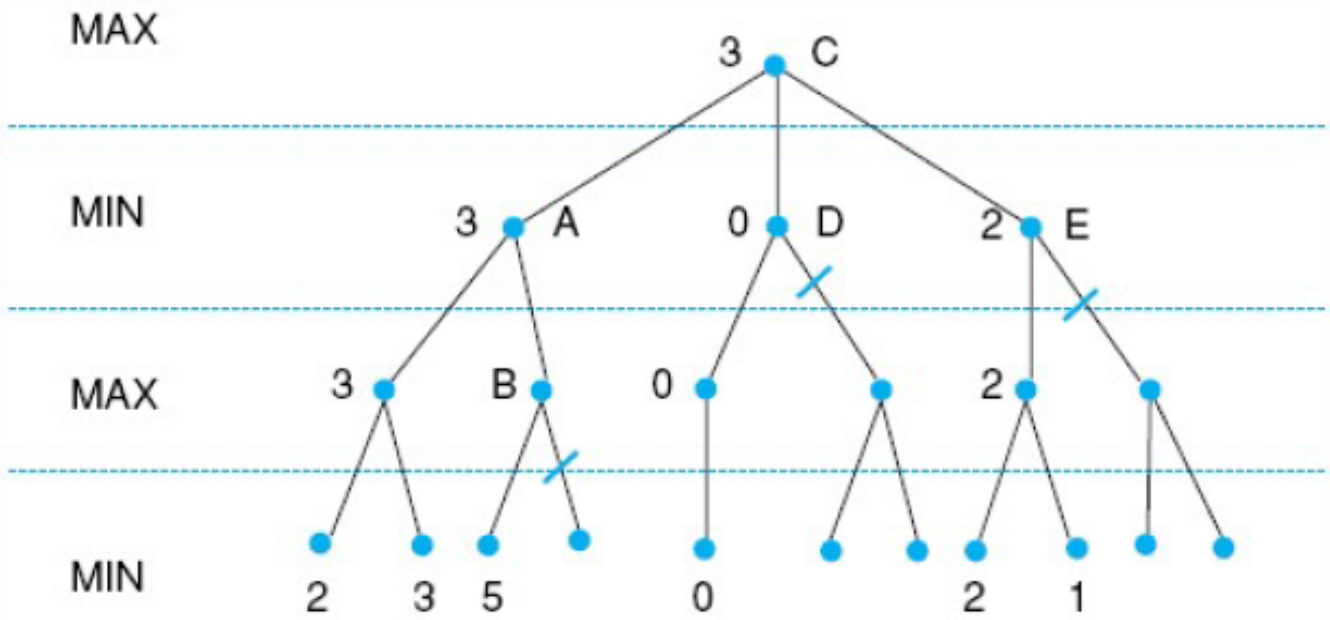




Alpha-Beta Example (continued)



Alpha-Beta Pruning



A has $\beta = 3$ (A will be no larger than 3)

B is β pruned, since $5 > 3$

C has $\alpha = 3$ (C will be no smaller than 3)

D is α pruned, since $0 < 3$

E is α pruned, since $2 < 3$

C is 3

Alpha-Beta Implementation

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

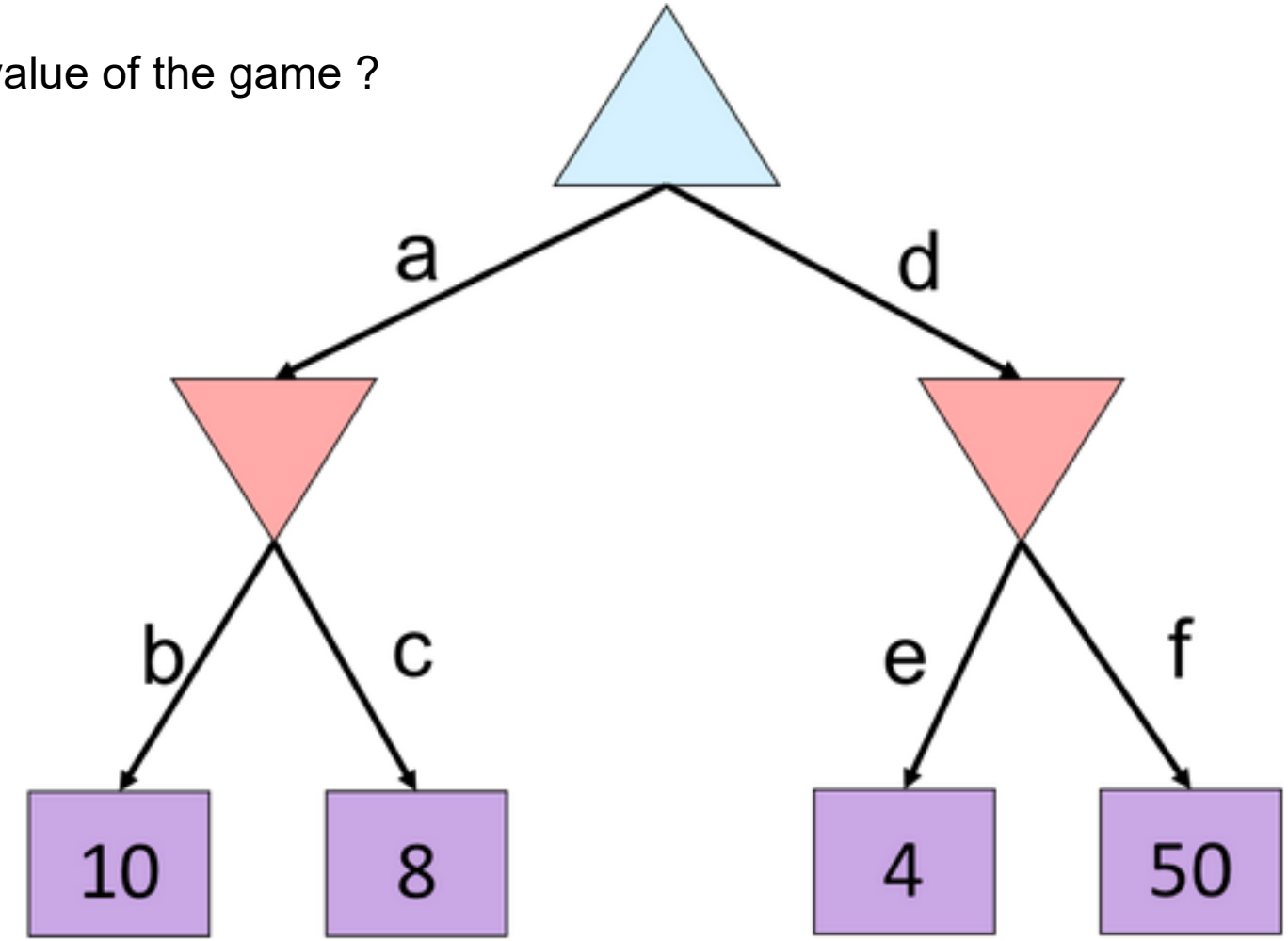
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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
```

Alpha-Beta Quiz

Whats the value ?
Should we prune it?

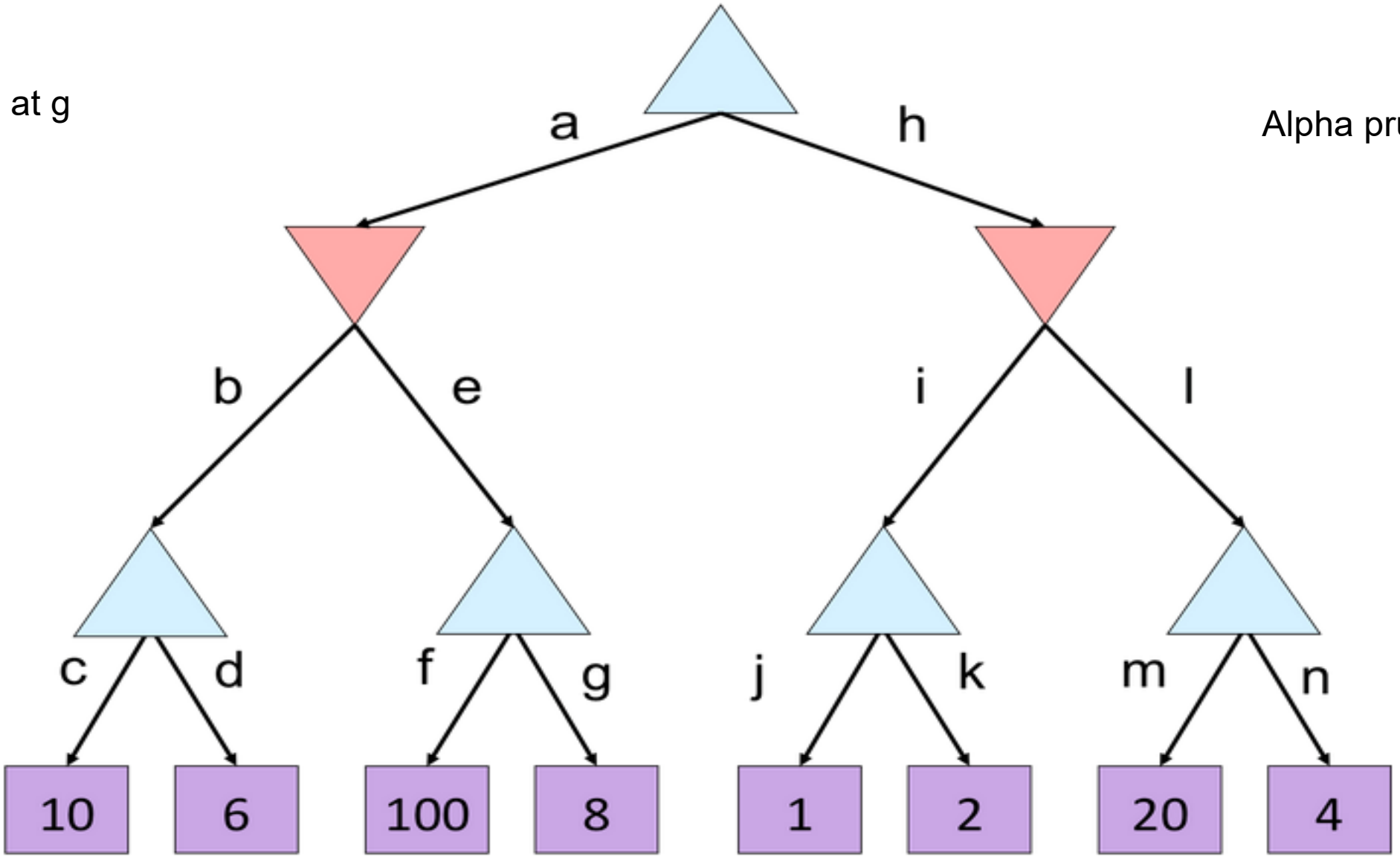
Pruning will effect the value of the game ?



Alpha-Beta Quiz 2

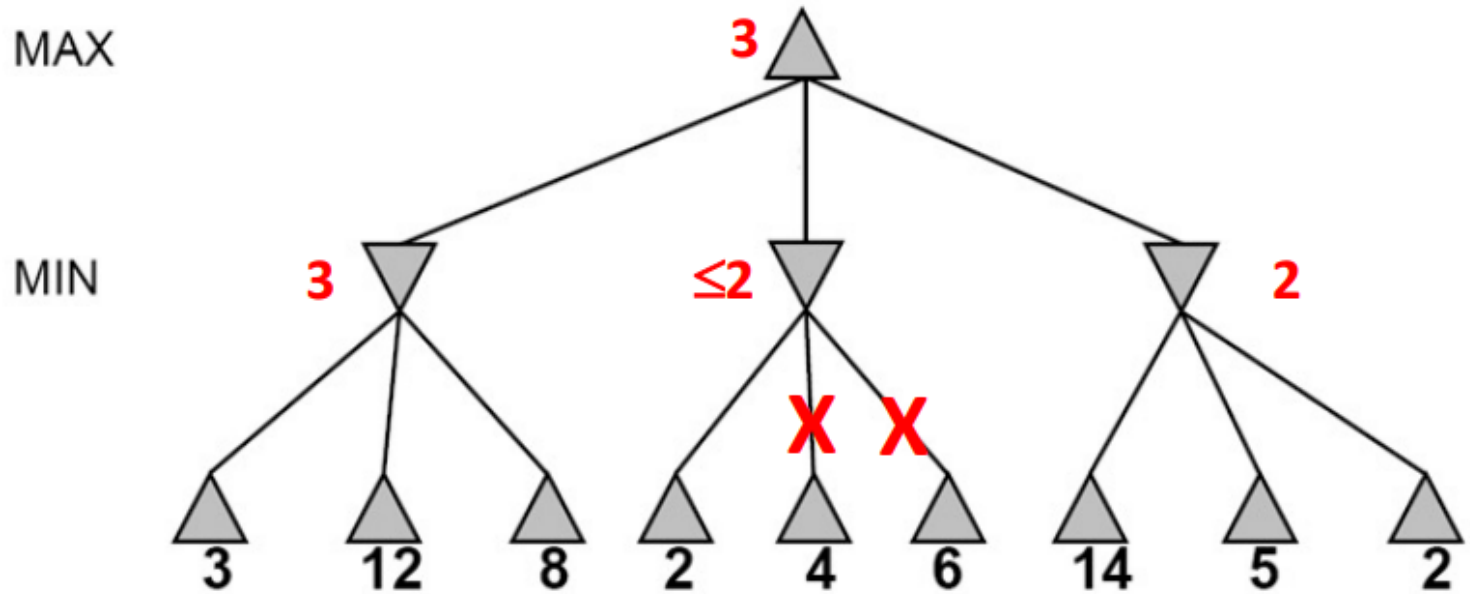
Beta Pruning at g

Alpha pruning at l



Computational complexity of alpha-beta pruning

- The basic idea of alpha-beta pruning is to reduce the complexity of minimax from $O\{n^d\}$ to $O\{n^{d/2}\}$.
- That can be done with no loss of accuracy,
- … but only if the children of any given node are optimally ordered.



Alpha-beta pruning

- Pruning does not affect final result
- Amount of pruning depends on move ordering
  - Should start with the “best” moves (highest-value for MAX or lowest-value for MIN)
  - For chess, can try captures first, then threats, then forward moves, then backward moves
  - Can also try to remember “killer moves” from other branches of the tree
- With perfect ordering, the time to find the best move is reduced to $O(b^{m/2})$ from $O(b^m)$
  - Depth of search is effectively doubled