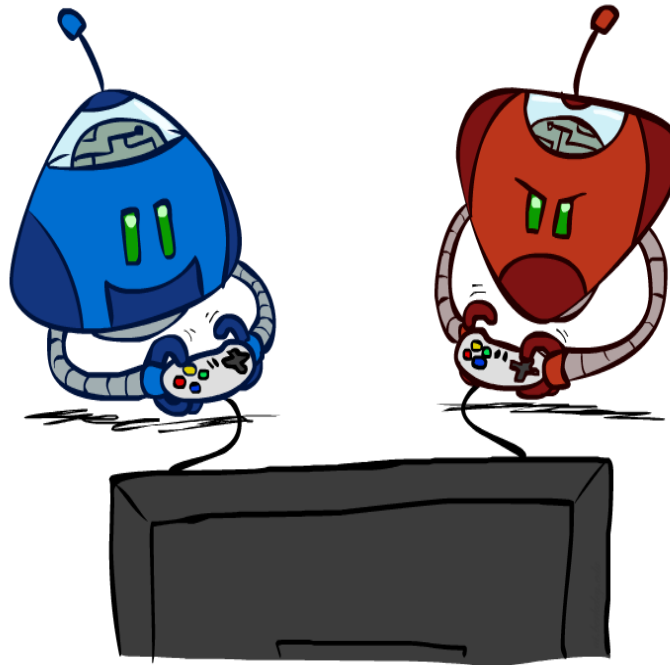


Artificial Intelligence

Adversarial Search

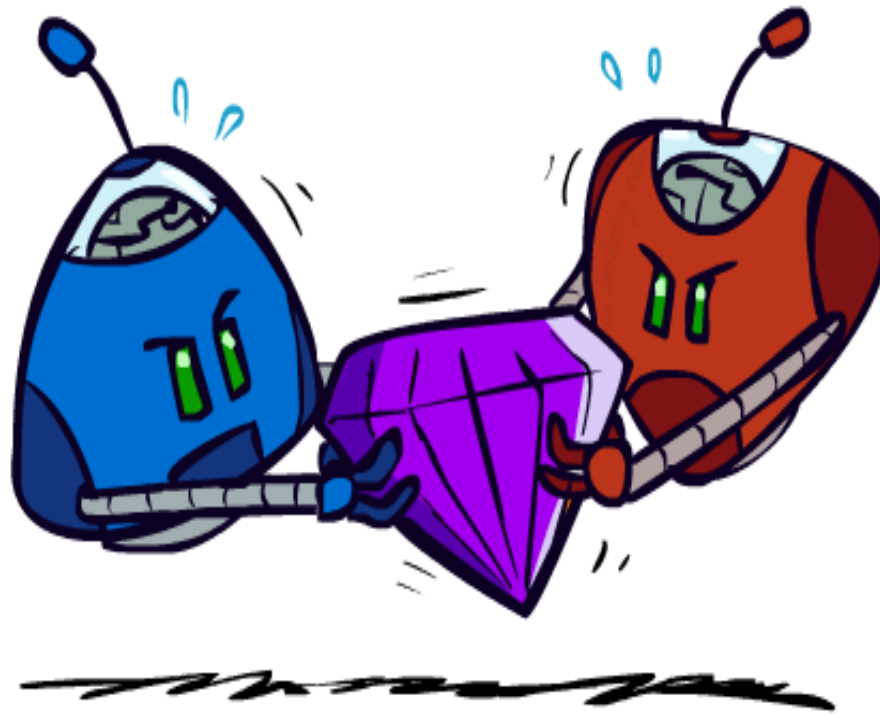


Instructors: David Suter and Qince Li

Course Delivered for HIT

[Many slides adapted from those created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. Some others from colleagues at Adelaide University.]

Adversarial Games

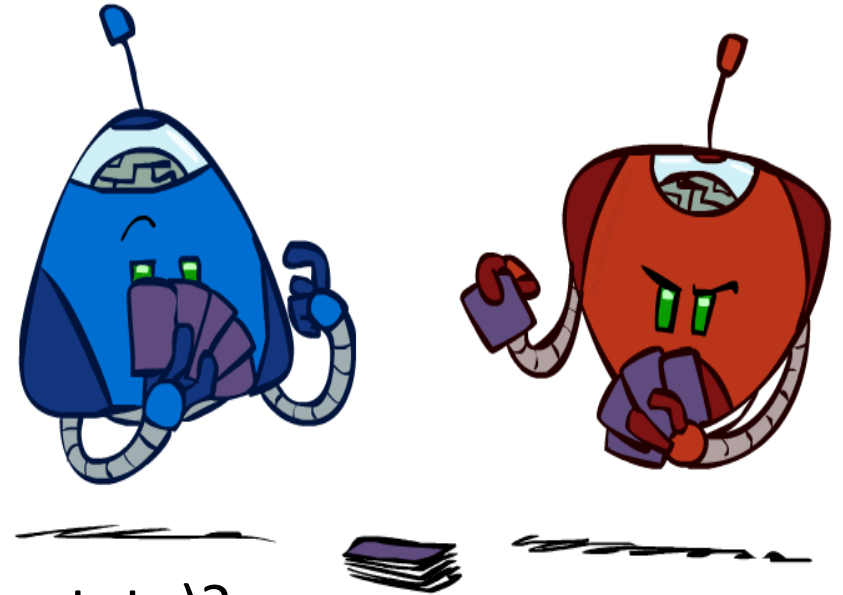


Types of Games

- Many different kinds of games!

- Axes:

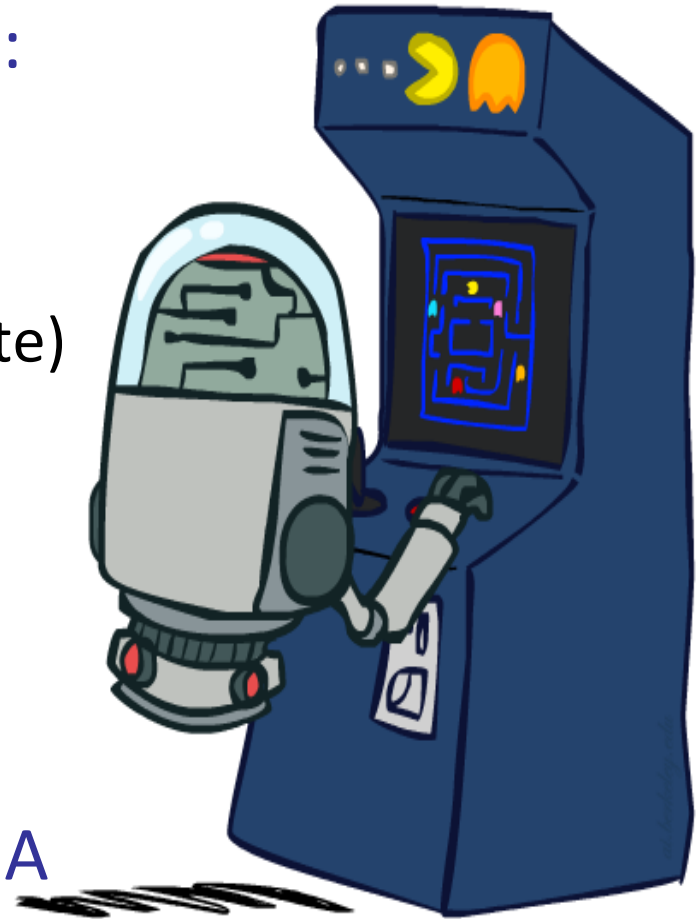
- Deterministic or stochastic?
- One, two, or more players?
- Zero sum?
- Perfect information (can you see the state)?



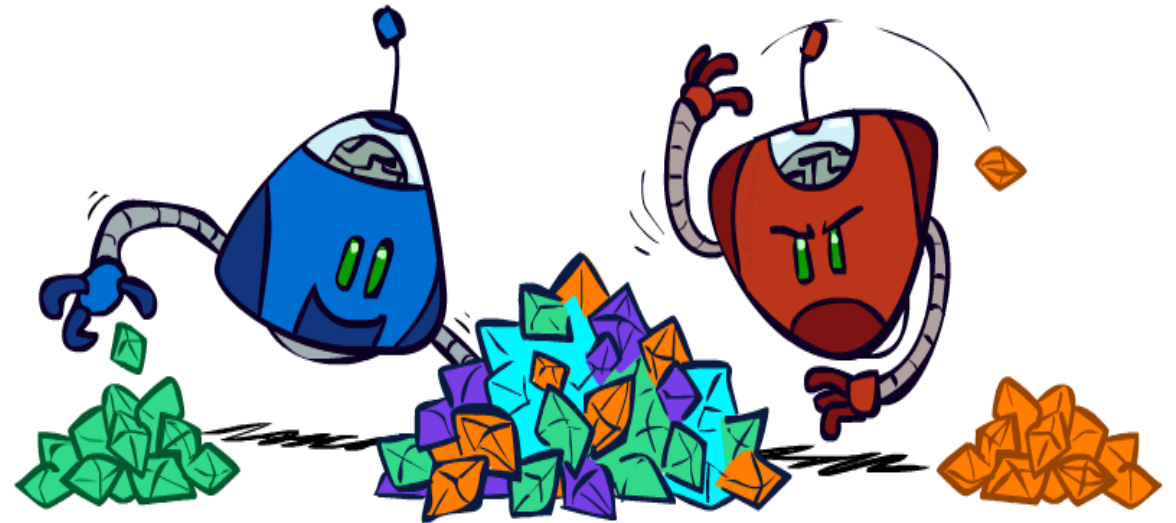
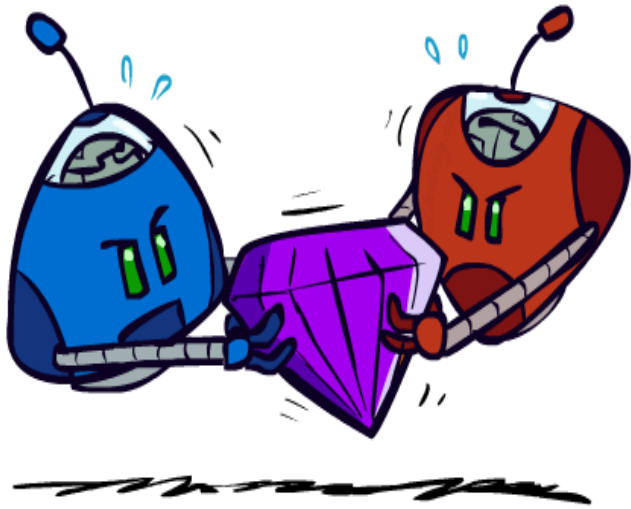
- Want algorithms for calculating a **strategy (policy)** which recommends a move from each state

Deterministic Games

- Many possible formalizations, one is:
 - States: S (start at s_0)
 - Players: $P=\{1\dots N\}$ (usually take turns)
 - Actions: A (may depend on player / state)
 - Transition Function: $S \times A \rightarrow S$
 - Terminal Test: $S \rightarrow \{t, f\}$
 - Terminal Utilities: $S \times P \rightarrow R$
- Solution for a player is a **policy**: $S \rightarrow A$



Zero-Sum Games



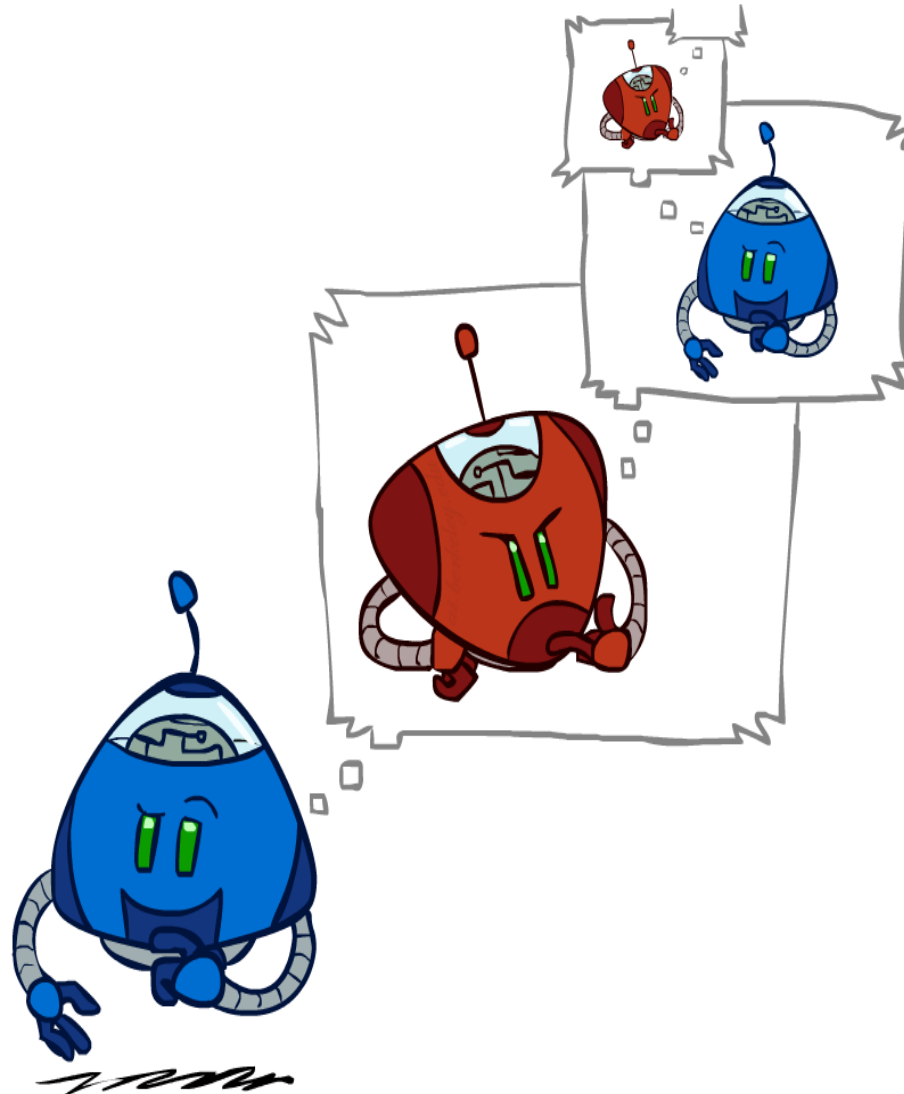
- Zero-Sum Games

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

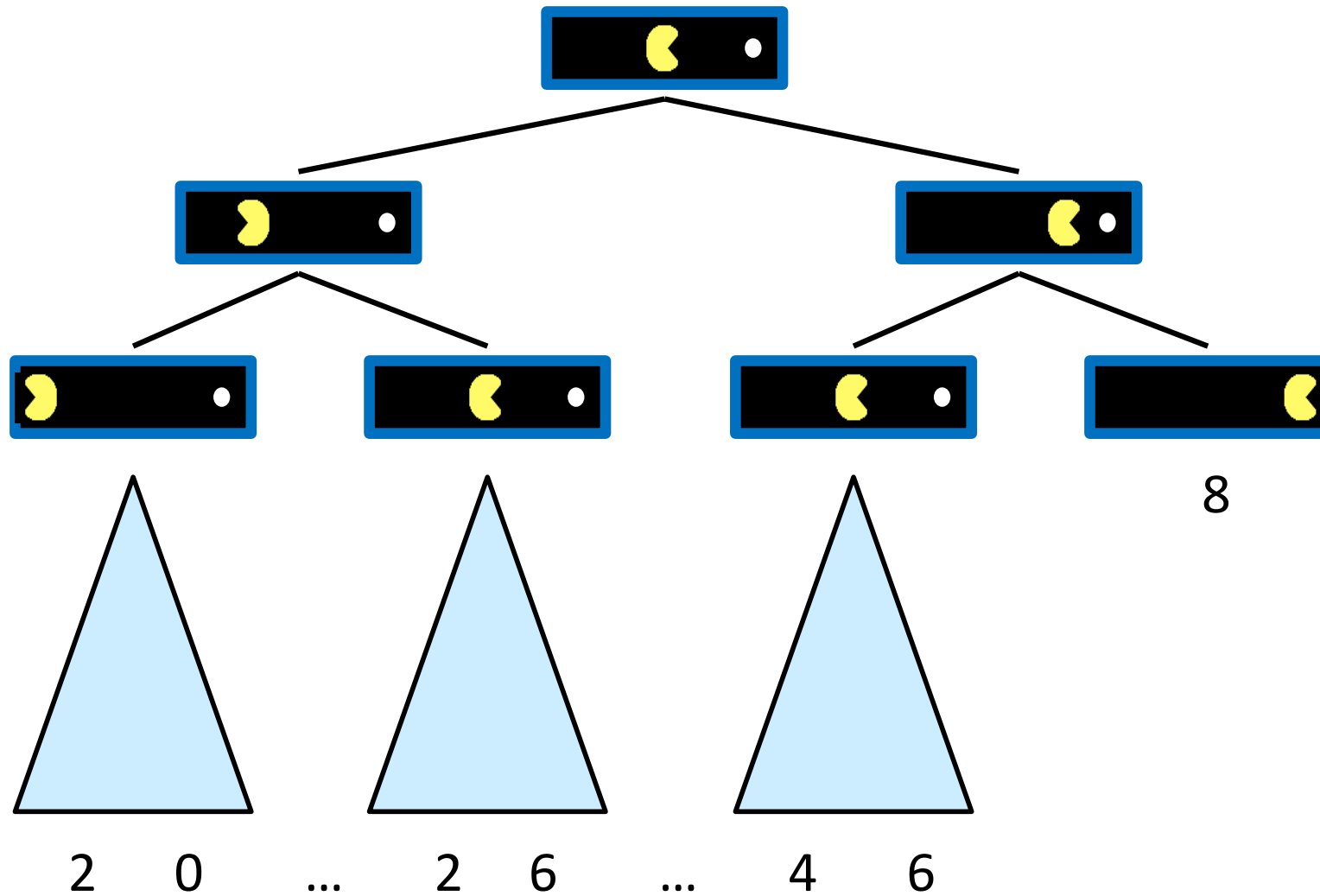
- General Games

- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible

Adversarial Search

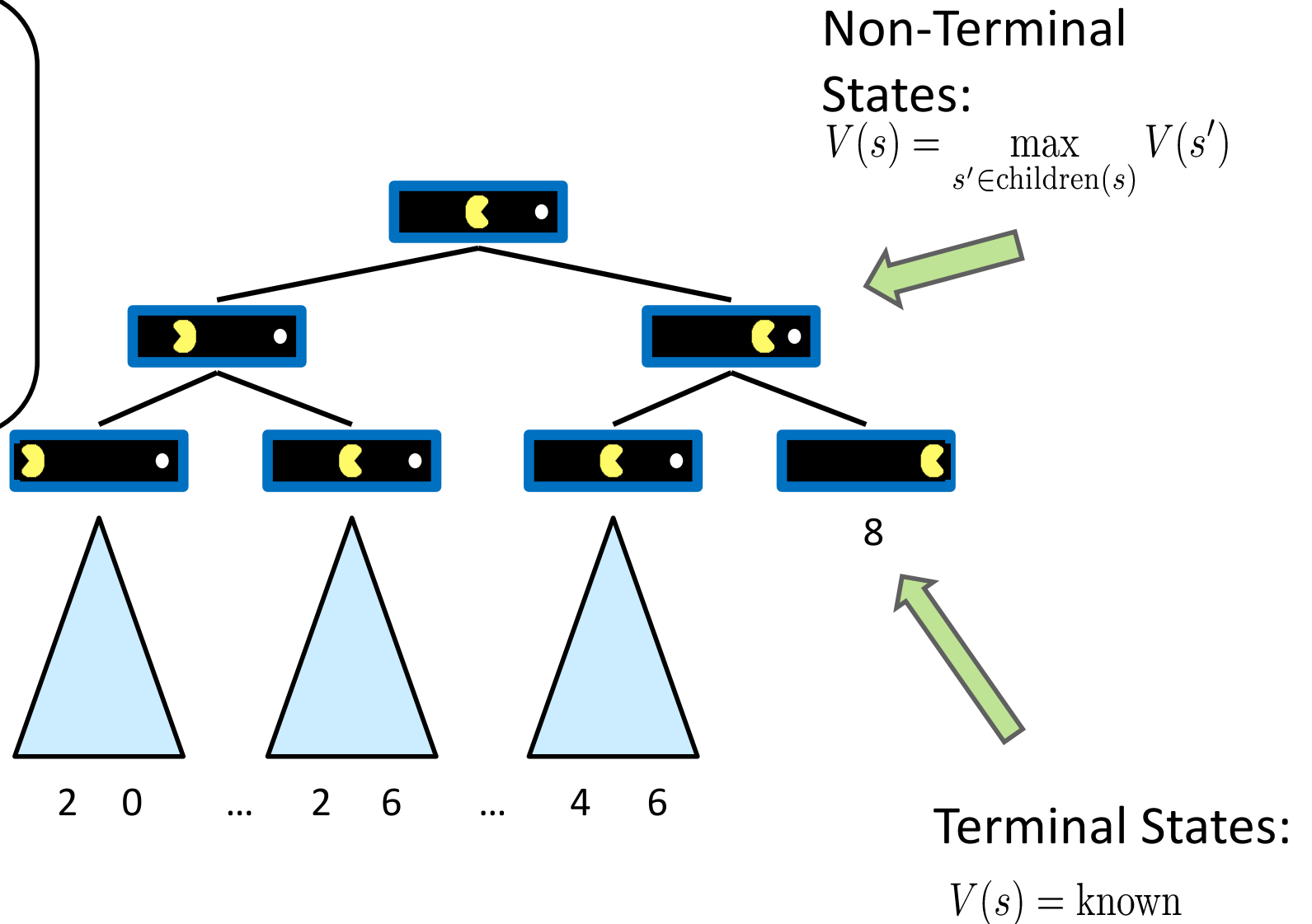


Single-Agent Trees

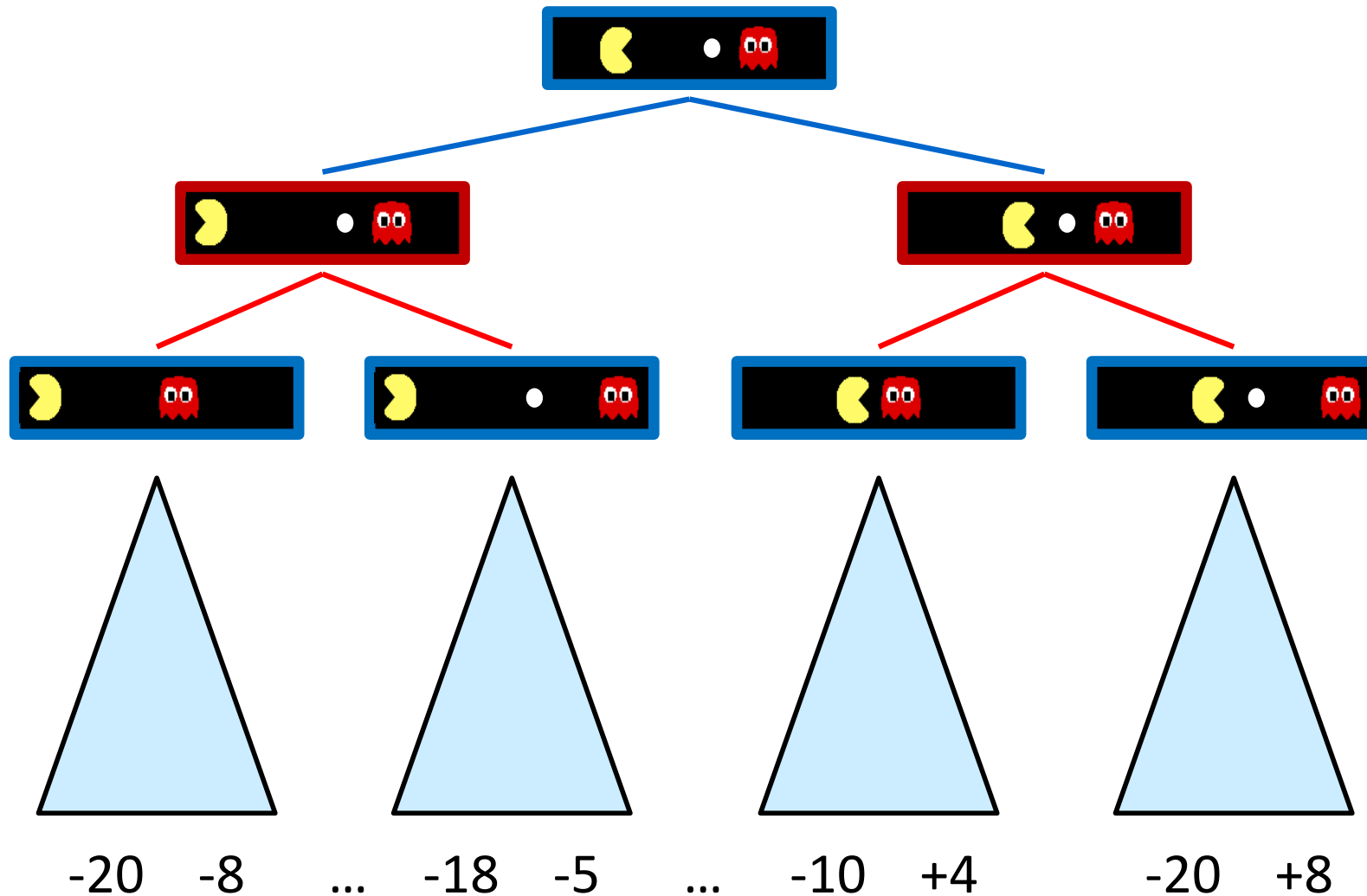


Value of a State

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



Adversarial Game Trees



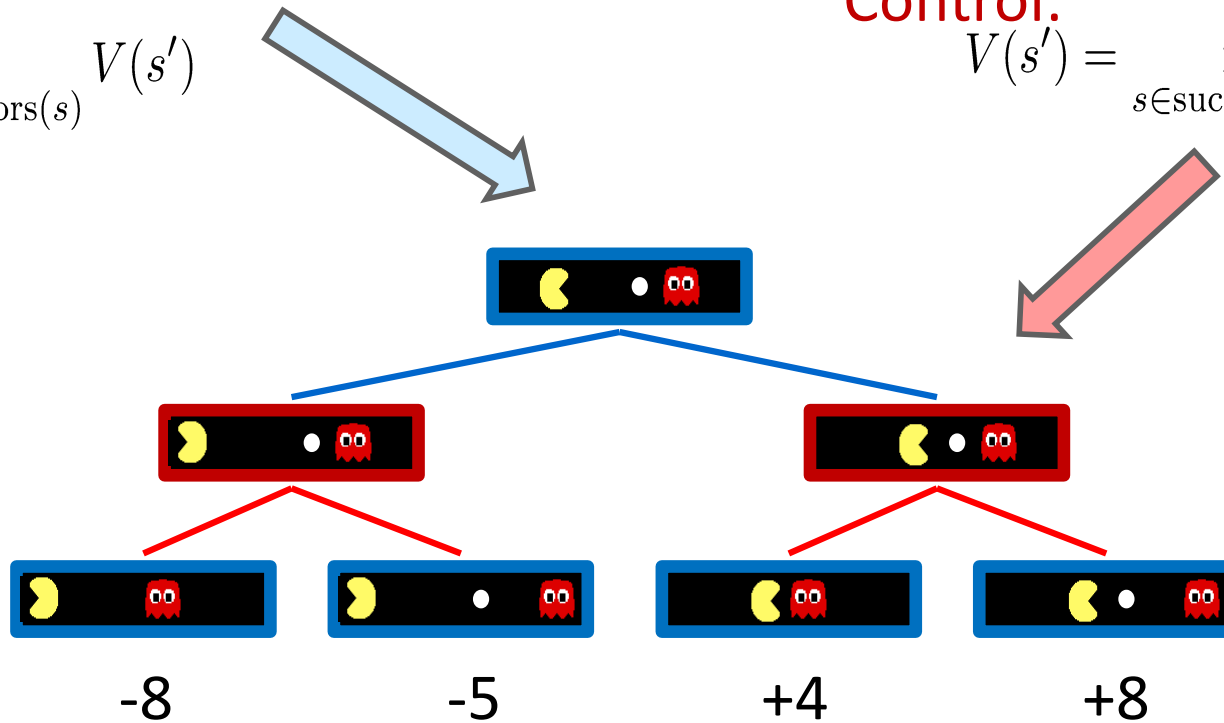
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

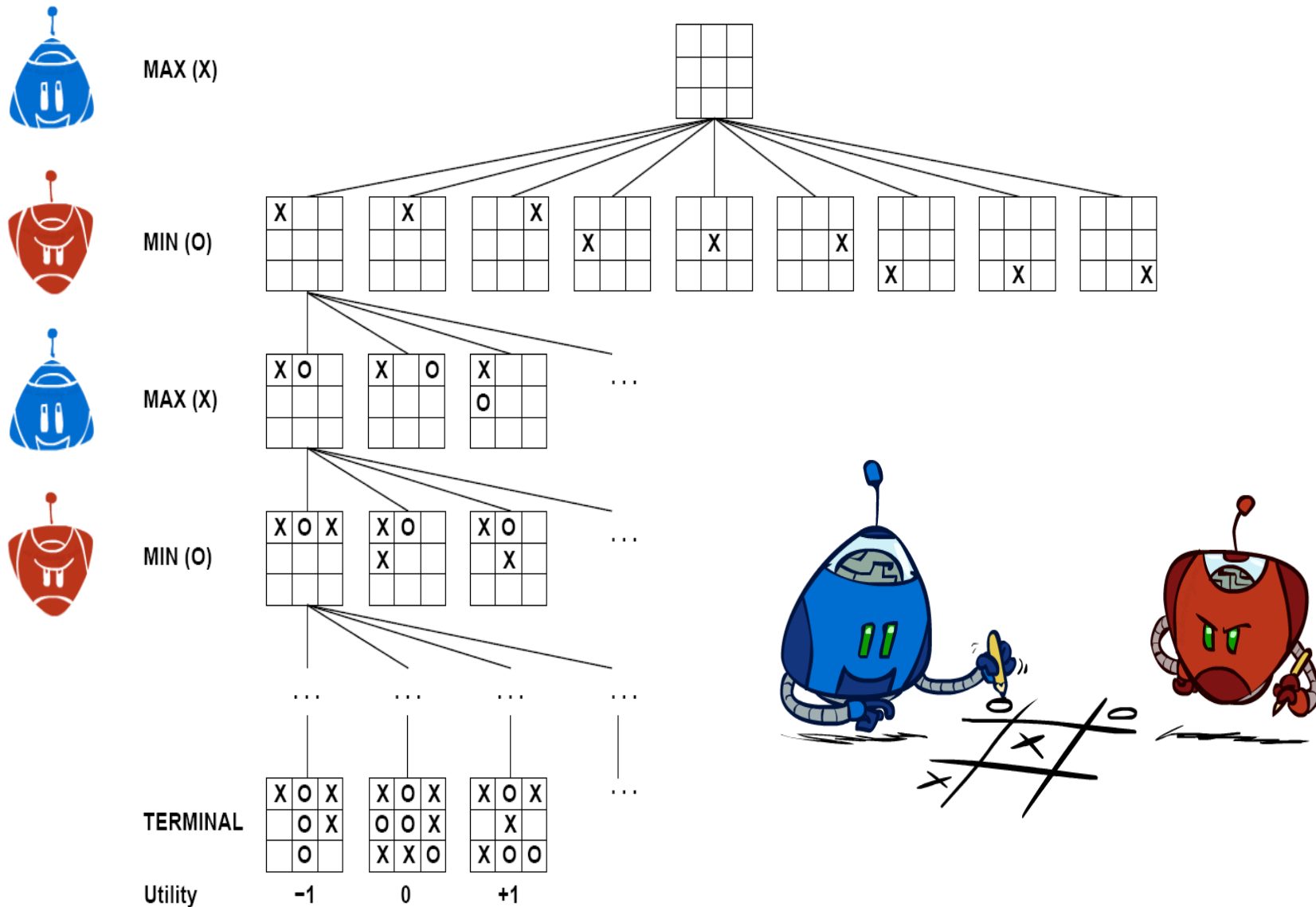
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

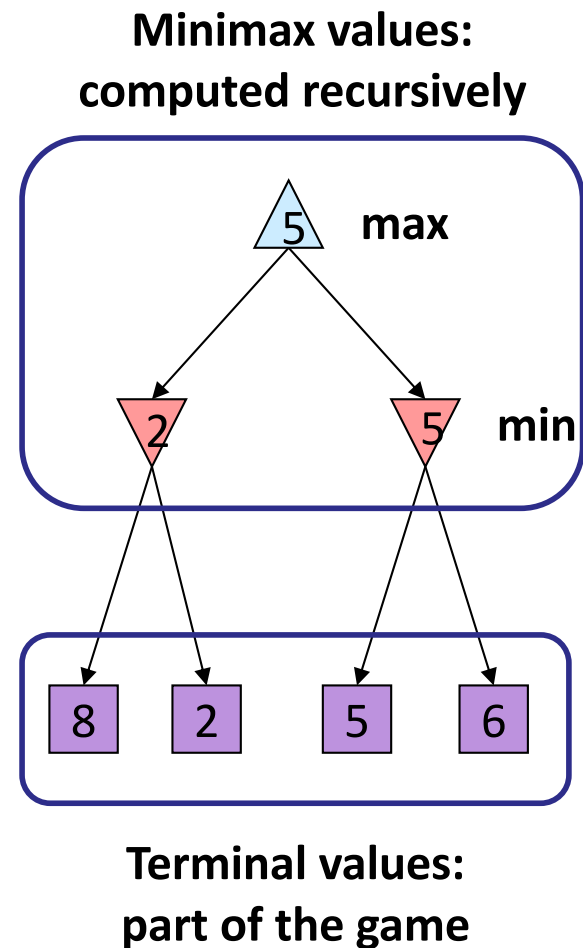
$$V(s) = \text{known}$$

Tic-Tac-Toe Game Tree



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 Implementation

```
def max-value(state):
```

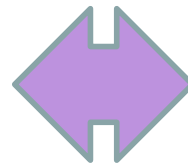
```
    initialize v =  $-\infty$ 
```

```
    for each successor of state:
```

```
        v = max(v, min-  
            value(successor))
```

```
    return v
```

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



```
def min-value(state):
```

```
    initialize v =  $+\infty$ 
```

```
    for each successor of state:
```

```
        v = min(v, max-  
            value(successor))
```

```
    return v
```

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

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  $v = -\infty$ 
```

```
    for each successor of state:
```

```
         $v = \max(v,$   
             $\text{value(successor)})$ 
```

```
    return v
```

```
def min-value(state):
```

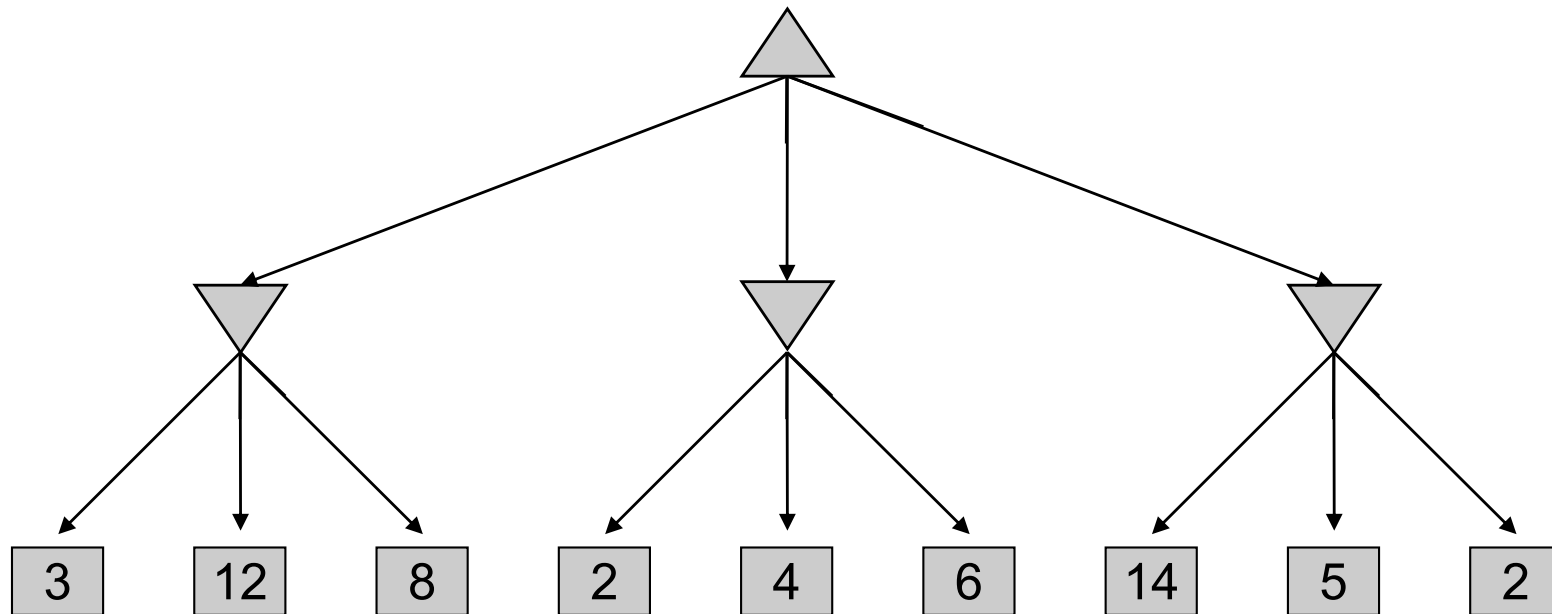
```
    initialize  $v = +\infty$ 
```

```
    for each successor of state:
```

```
         $v = \min(v,$   
             $\text{value(successor)})$ 
```

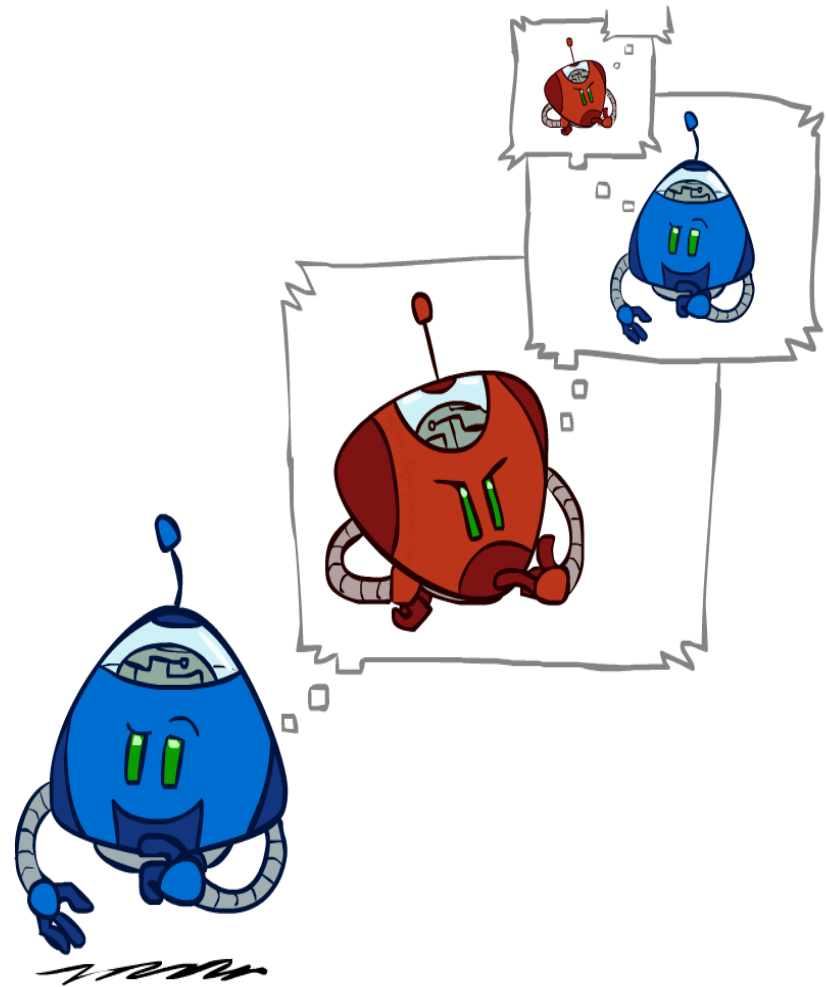
```
    return v
```

Minimax Example



Minimax Efficiency

- How efficient is minimax?
 - Just like (exhaustive) DFS
 - 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?

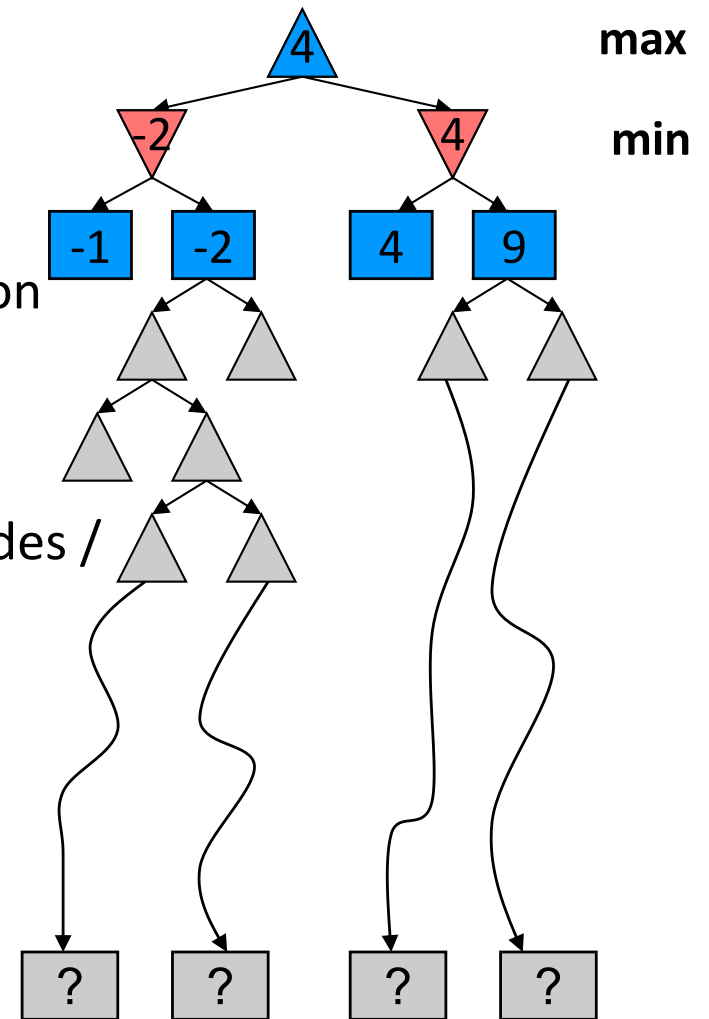


Resource Limits



Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
 - Instead, search only to a limited depth in the tree
 - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
 - Suppose we have 100 seconds, can explore 10K nodes / sec
 - So can check 1M nodes per move
 - α - β reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



Depth Matters

- Evaluation functions are always imperfect
- 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

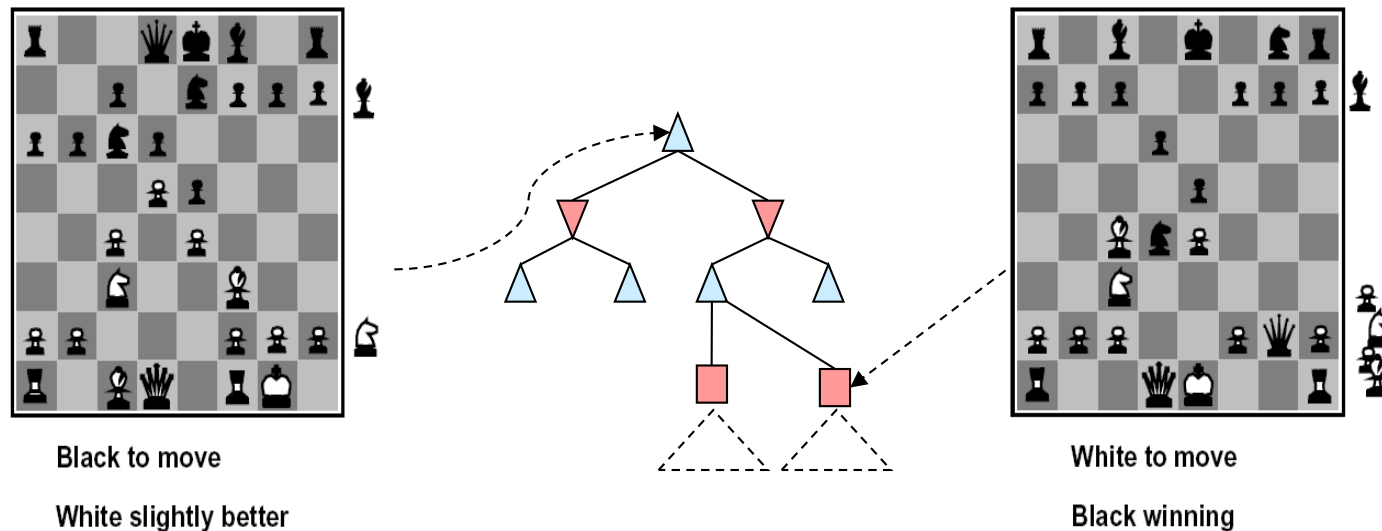


Evaluation Functions



Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search

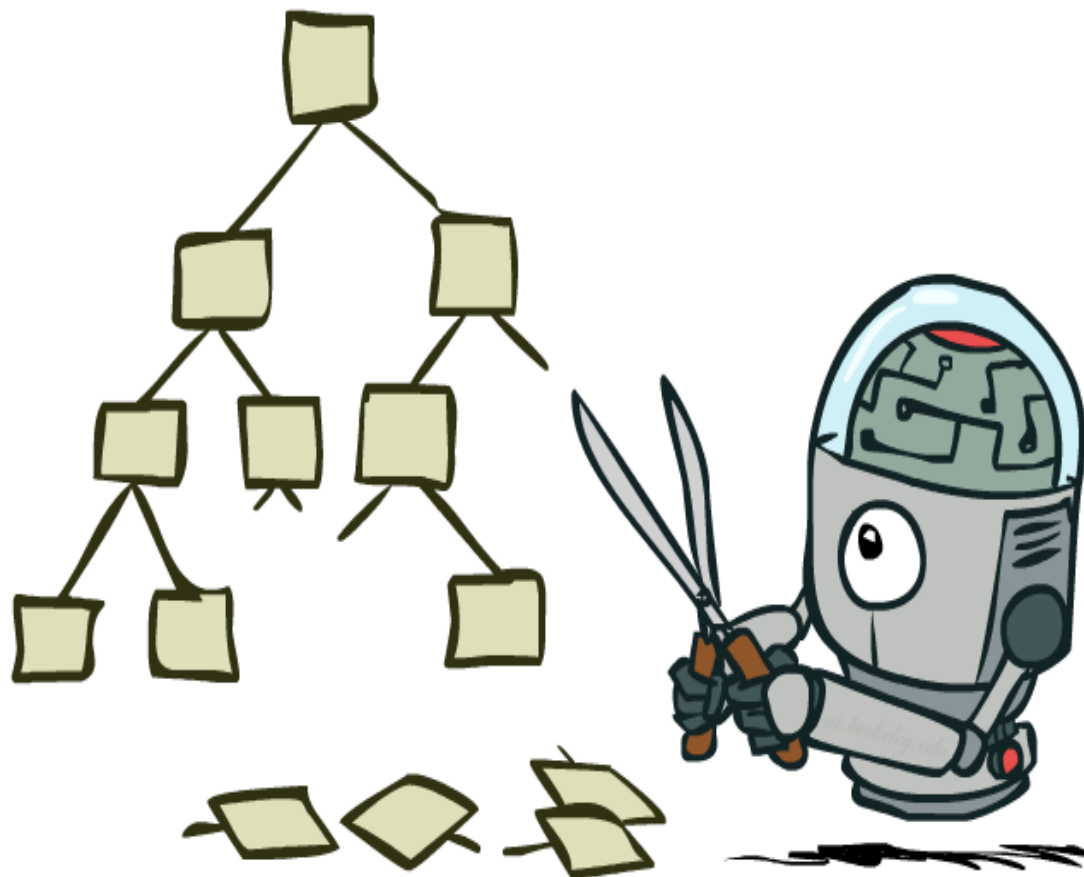


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

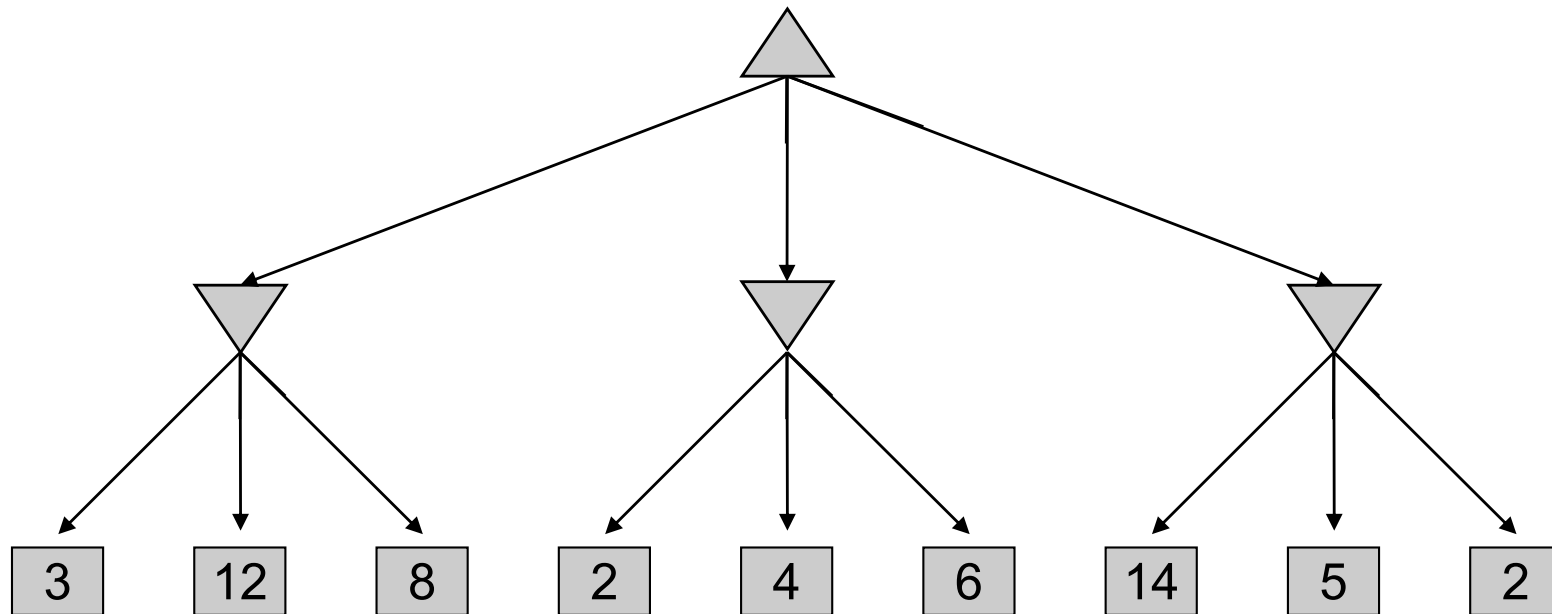
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$$

- e.g. $f_1(s) = (\text{num white queens} - \text{num black queens})$, etc.

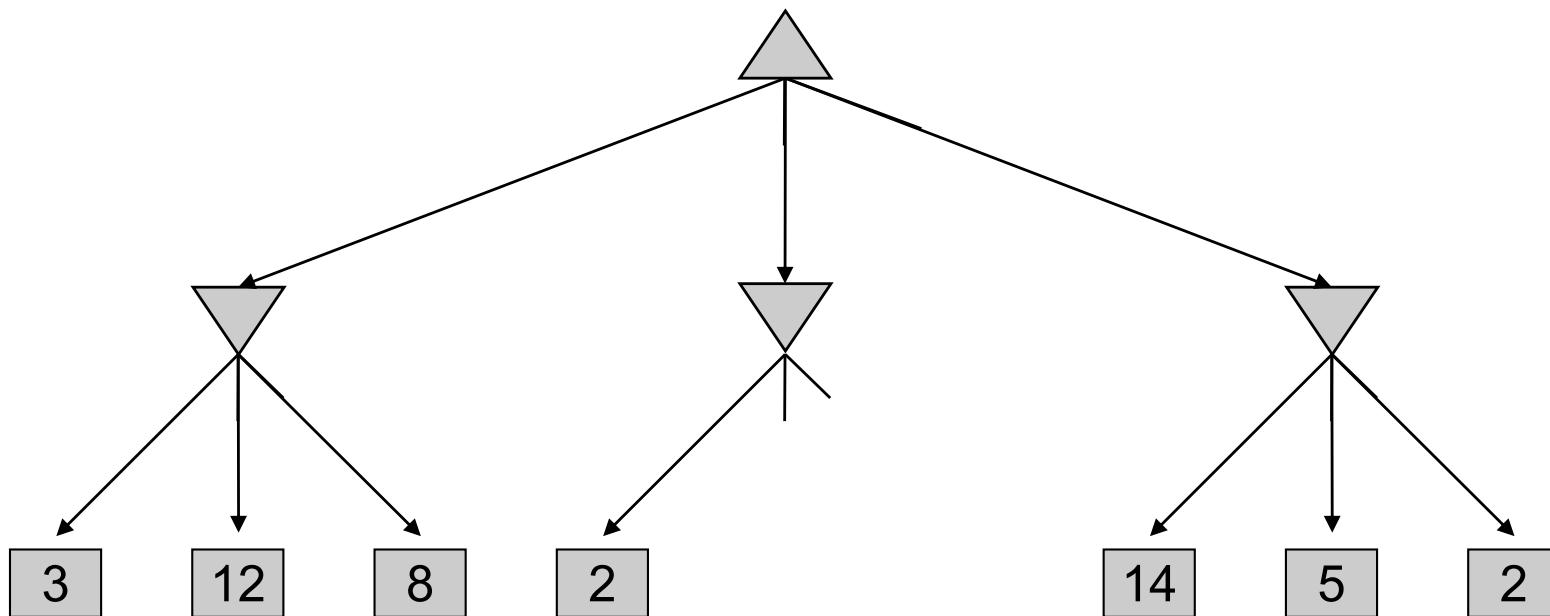
Game Tree Pruning



Minimax Example



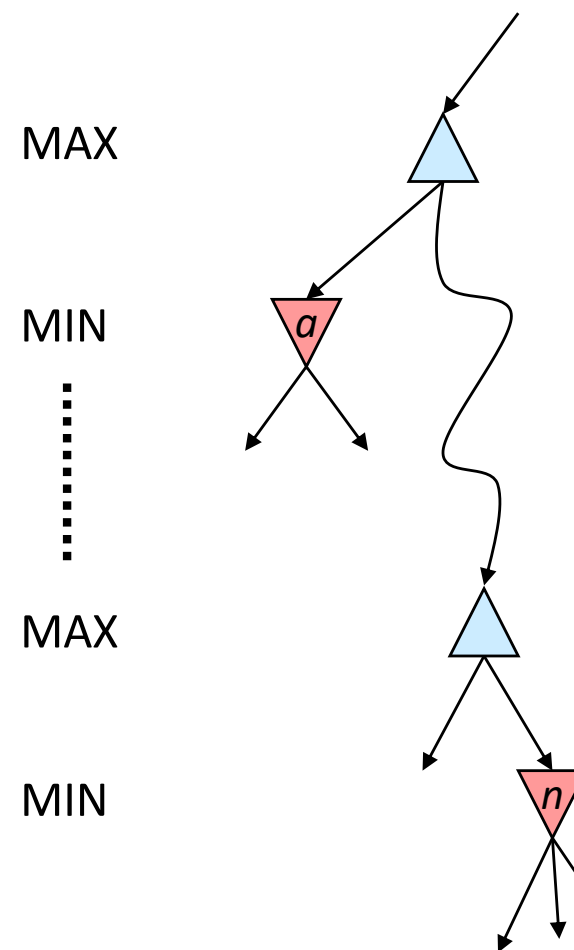
Minimax Pruning



Alpha-Beta Pruning

- General configuration (MIN version)

- We're computing the MIN-VALUE at some node n
- We're looping over n 's children
- n 's estimate of the childrens' min is dropping
- Who cares about n 's value? MAX
- Let a be the best value that MAX can get at any choice point along the current path from the root
- If n becomes worse than a , MAX will avoid it, so we can stop considering n 's other children (it's already bad enough that it won't be played)

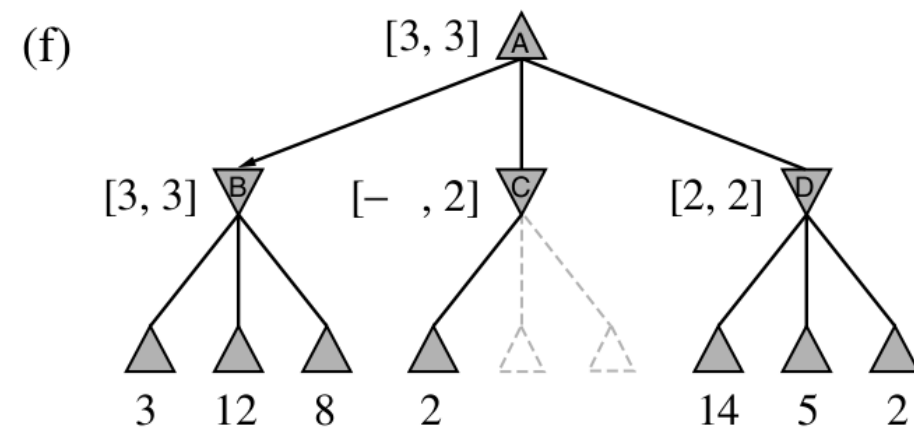
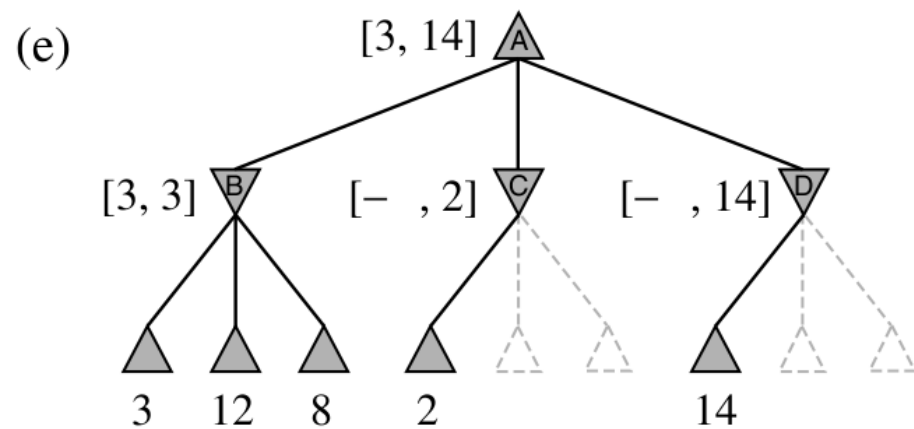
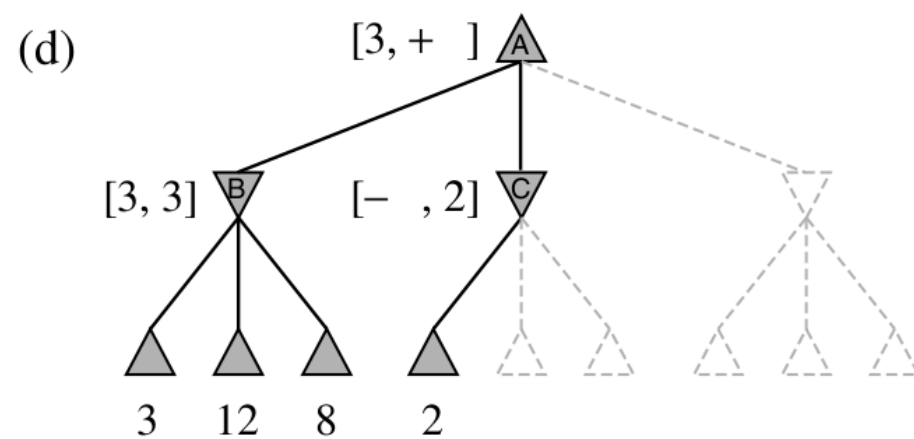
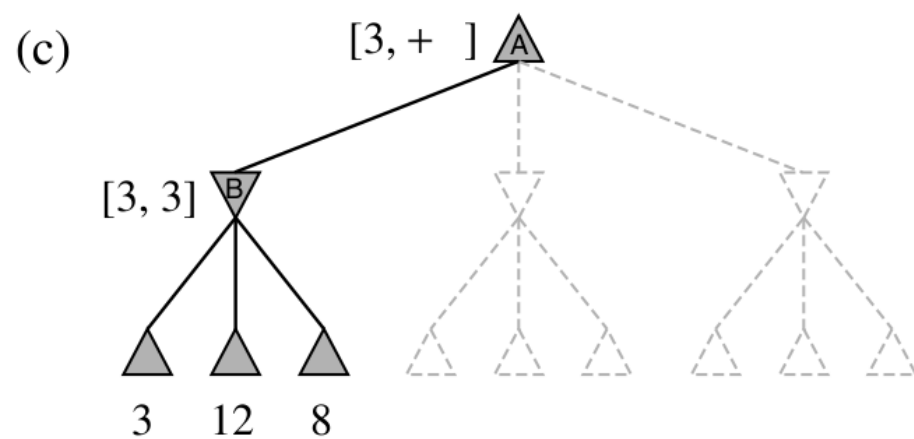
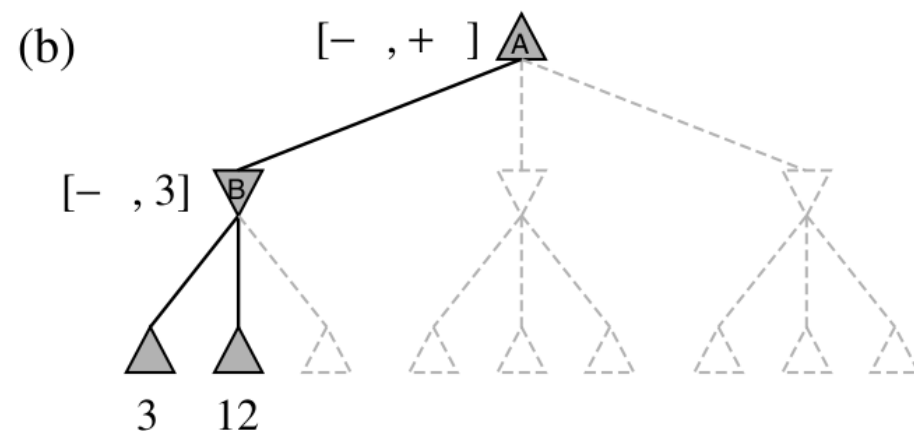
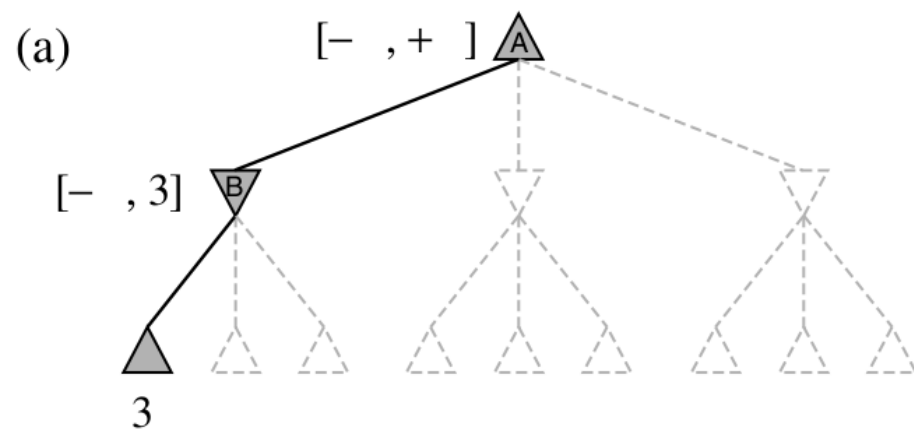


Alpha-Beta Implementation

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

```
def max-value(state,  $\alpha$ ,  $\beta$ ):  
    initialize  $v = -\infty$   
    for each successor of state:  
         $v = \max(v,$   
             $\text{value}(\text{successor}, \alpha, \beta))$   
        if  $v \geq \beta$  return  $v$   
         $\alpha = \max(\alpha, v)$   
    return  $v$ 
```

```
def min-value(state,  $\alpha$ ,  $\beta$ ):  
    initialize  $v = +\infty$   
    for each successor of state:  
         $v = \min(v,$   
             $\text{value}(\text{successor}, \alpha, \beta))$   
        if  $v \leq \alpha$  return  $v$   
         $\beta = \min(\beta, v)$   
    return  $v$ 
```



Alpha-Beta Pruning Properties

- This pruning has **no effect** on minimax value computed for the root!
- Good child ordering improves effectiveness of pruning
- With “perfect ordering”:
 - Time complexity drops to $O(b^{m/2})$
 - Doubles solvable depth!
 - Full search of, e.g. chess, is still hopeless...
- This is a simple example of **metareasoning** (computing about what to compute)

Alpha-Beta

Step-By-Step Examples

