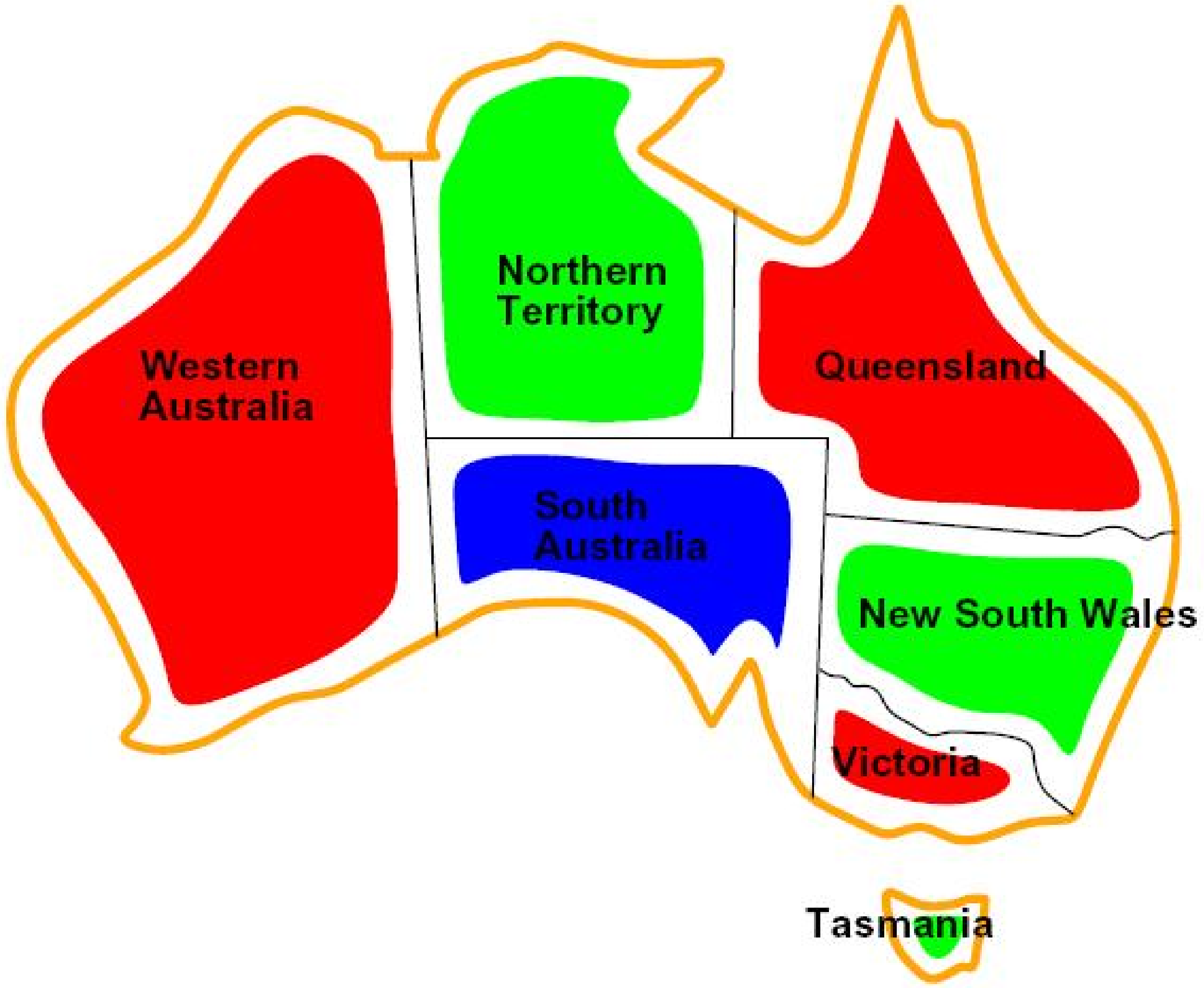
Announcement

- Programming Assignment 1b will be released on 10.10 due 10.24
- HW 1 will be released on 10.11 due 10.18
- Still having class tomorrow morning

Review: Informed Search

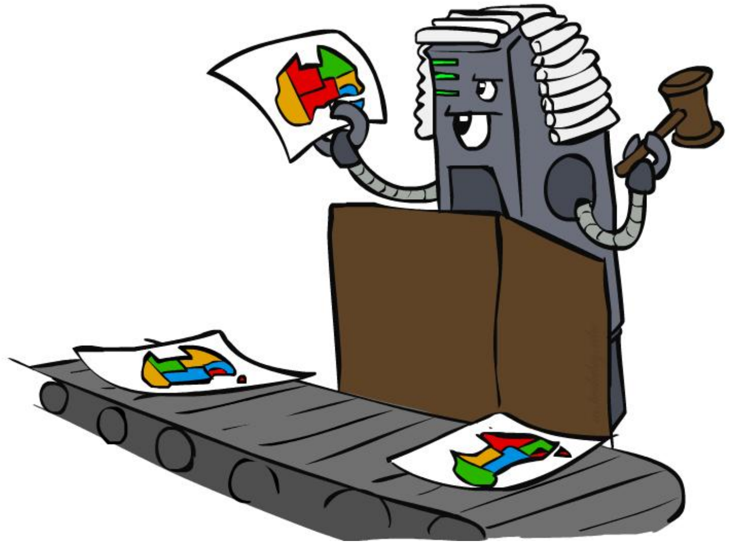
- Informed Search
  - Heuristics
    - A function that *estimates* how close a state is to a goal
  - Greedy Search
  - A* Search
    - Uniform-cost orders by path cost, or *backward cost* g(n)
    - Greedy orders by goal proximity, or *forward cost* h(n)
    - A* Search orders by the sum: f(n) = g(n) + h(n)
    - only stop when we dequeue a goal

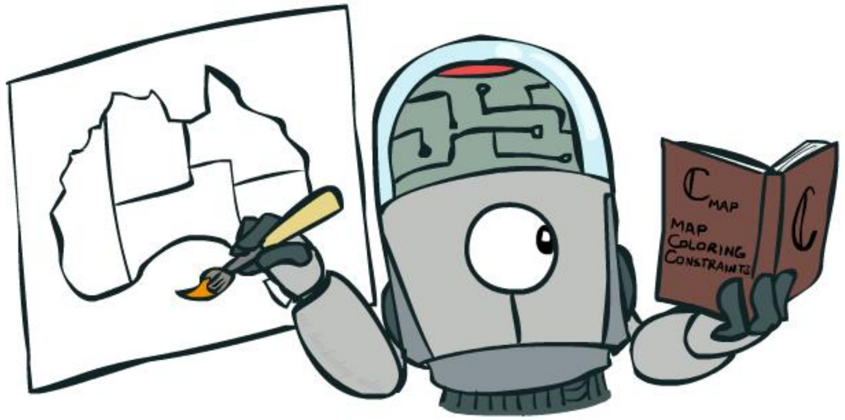
Review: CSP


Western Australia
Northern Territory
South Australia
Queensland
New South Wales
Victoria
Tasmania

Review: CSP

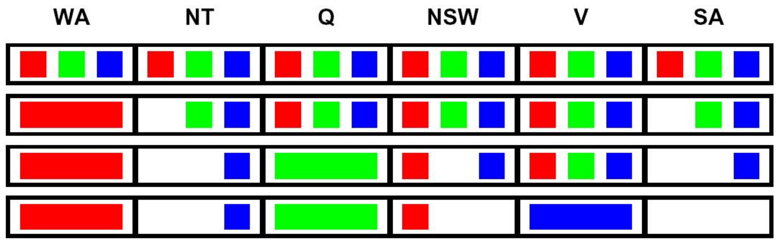
- Standard search problems:
  - State is a "black box": arbitrary data structure
  - Goal test can be any function over states
  - Successor function can also be anything
- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables X_i with values from a domain D (sometimes D depends on i)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables
- CSPs are specialized for identification problems

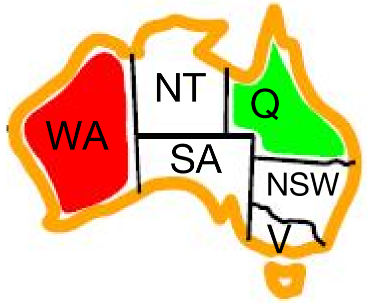




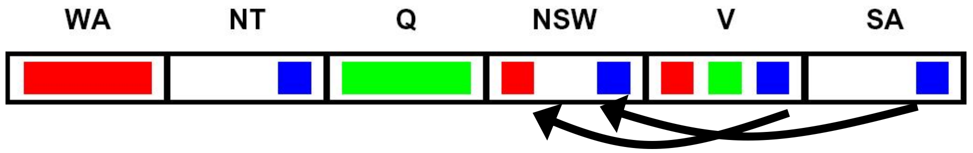
Review: Backtracking Search

- Backtracking = DFS + variable-ordering + fail-on-violation
- Filtering: Keep track of domains for unassigned variables and cross off bad options
  - **Forward checking:** Cross off values that violate a constraint when added to the existing assignment; whenever any variable has no value left, we backtrack





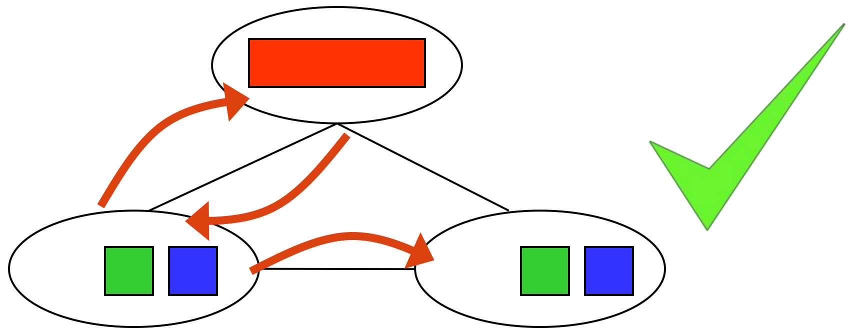
  - **Constraint Propagation:** Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures.
  - An arc X → Y is consistent if for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint
    - Arc consistency detects failure earlier than forward checking
    - *Remember: Delete from the tail!*

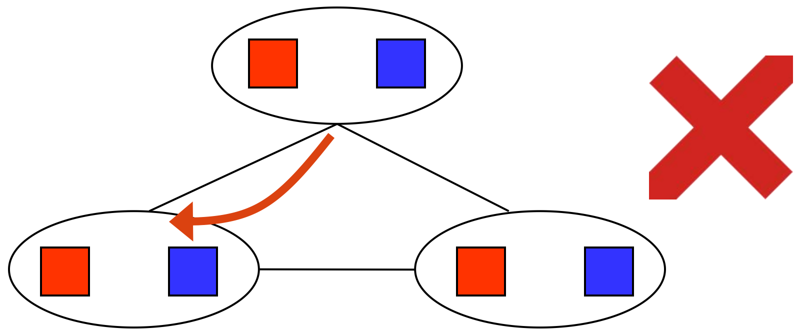


Limitations of Arc Consistency

- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)

- Arc consistency still runs inside a backtracking search!

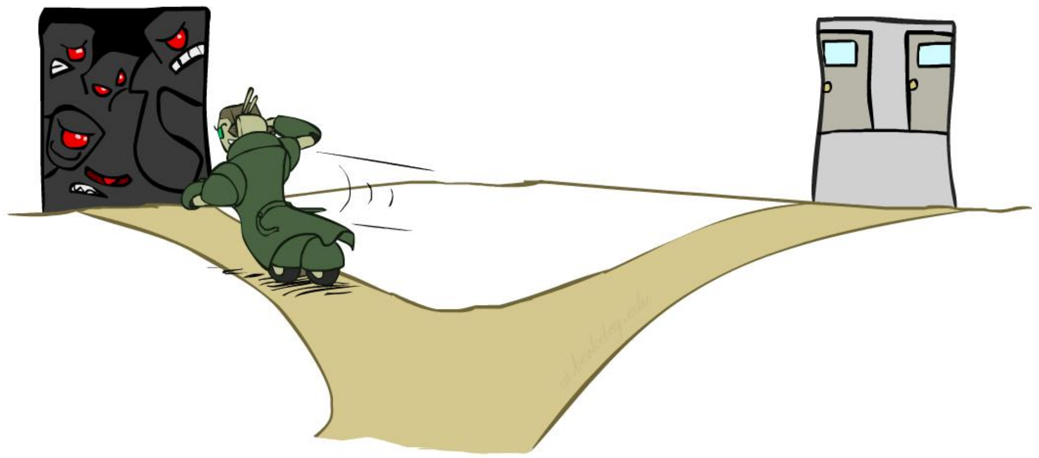




Ordering

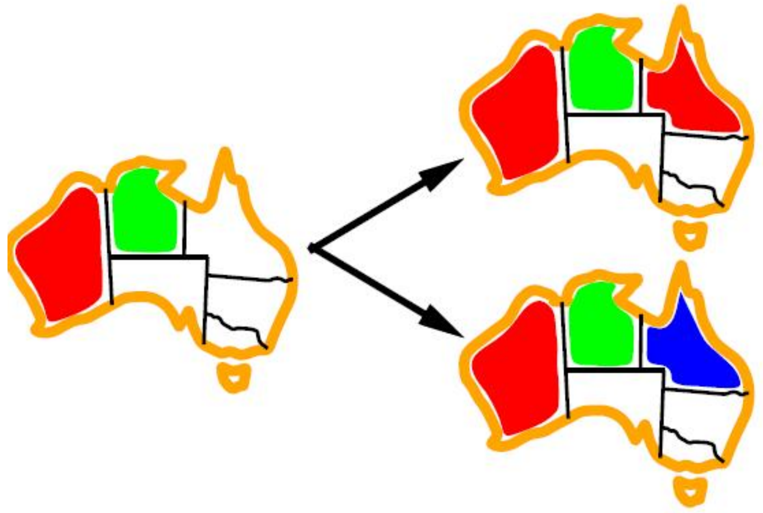
Ordering: Minimum Remaining Values

- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain
  - Also called "most constrained variable"

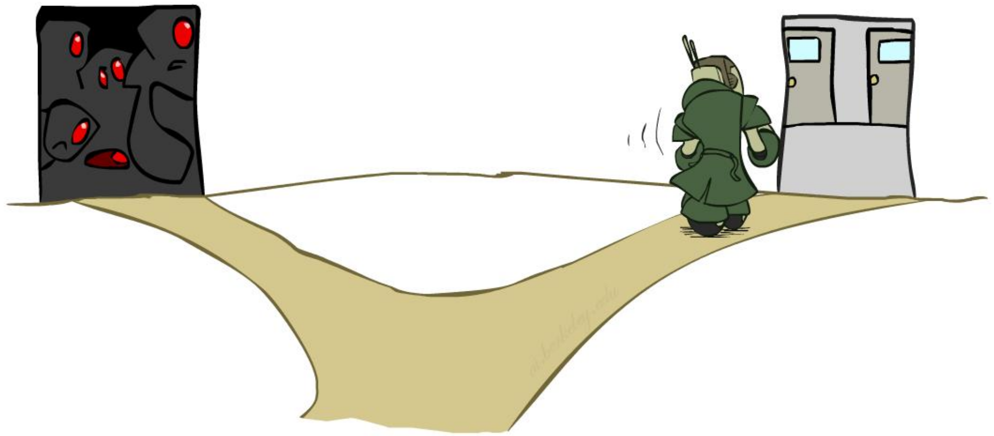


Ordering: Least Constraining Value

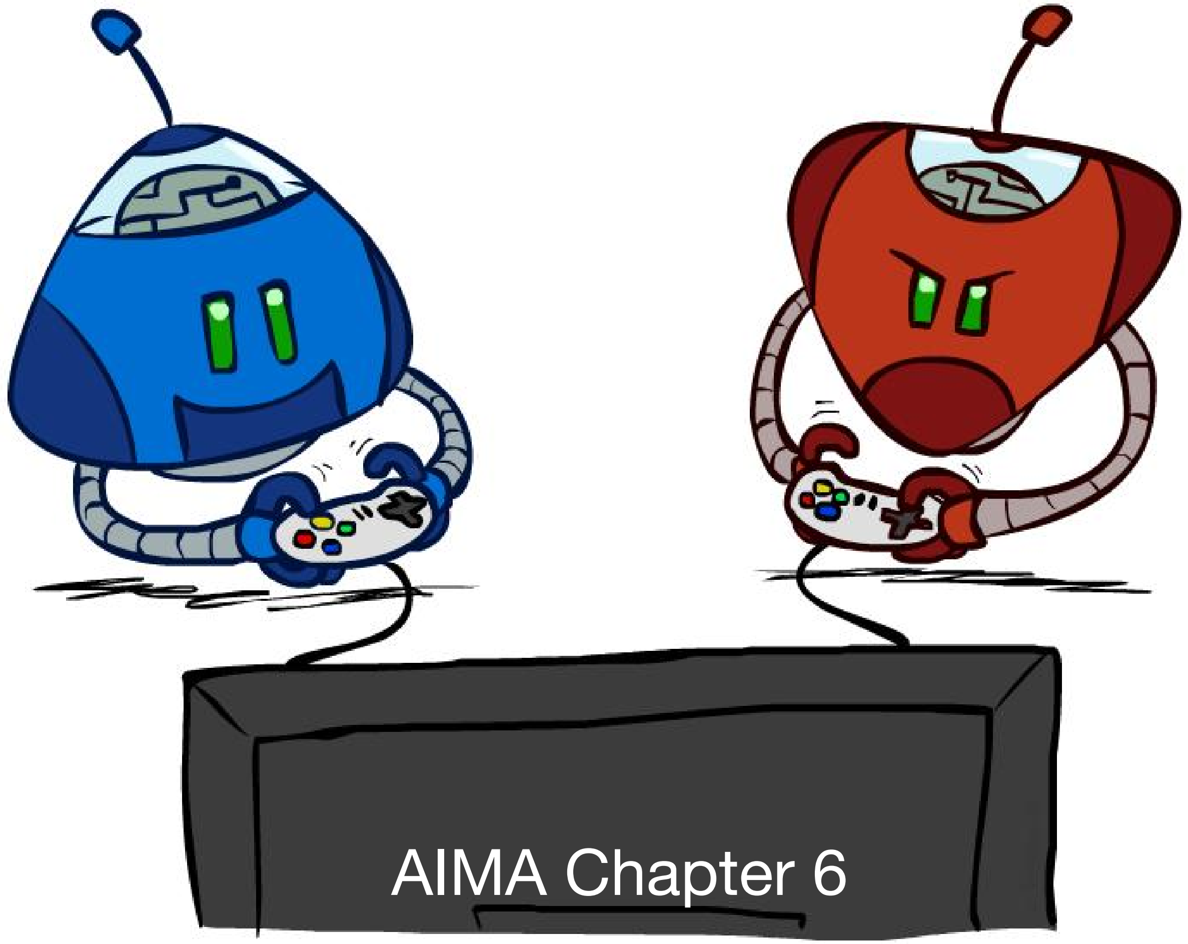
- Value Ordering: Least Constraining Value
  - Given a choice of variable, choose the *least constraining value*
  - I.e., the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (E.g., rerunning filtering)



- Combining these ordering ideas makes 1000 queens feasible

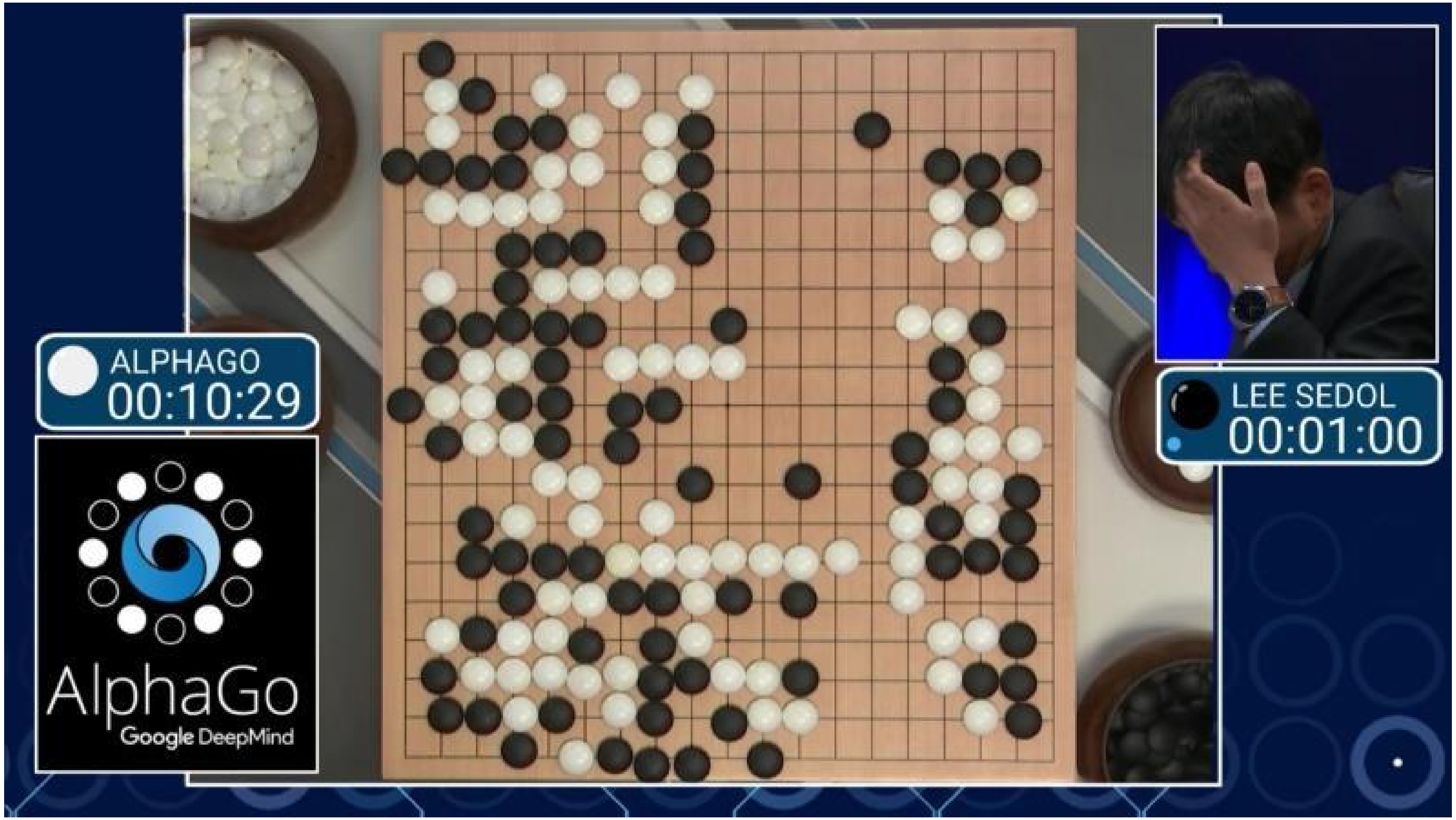


Adversarial Search



[Adapted from slides by Dan Klein and Pieter Abbeel at UC Berkeley (ai.berkeley.edu).]

AlphaGo: the most well-known AI?



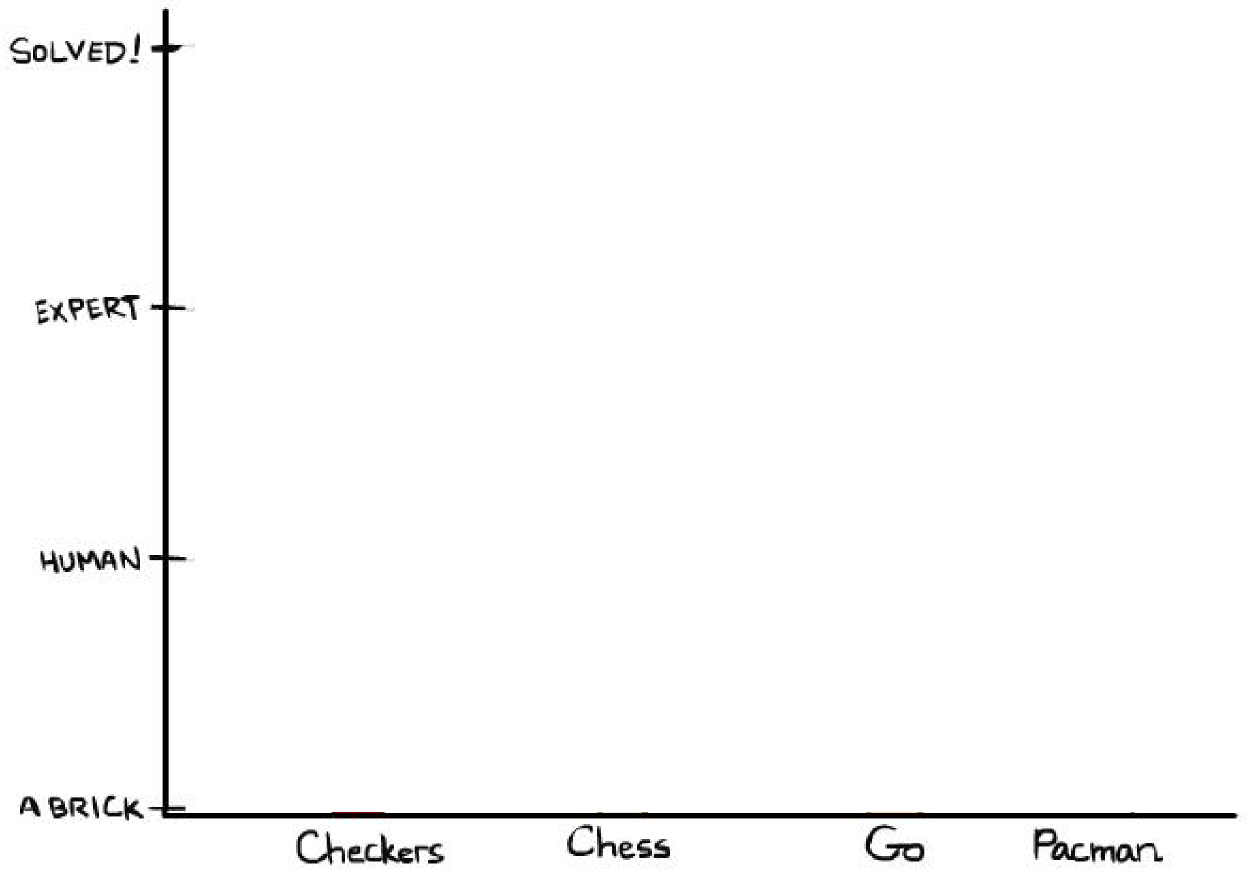
Game Playing State-of-the-Art

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go: 2016: Alpha GO defeats human champion! Uses Monte Carlo Tree Search, learned evaluation function.**

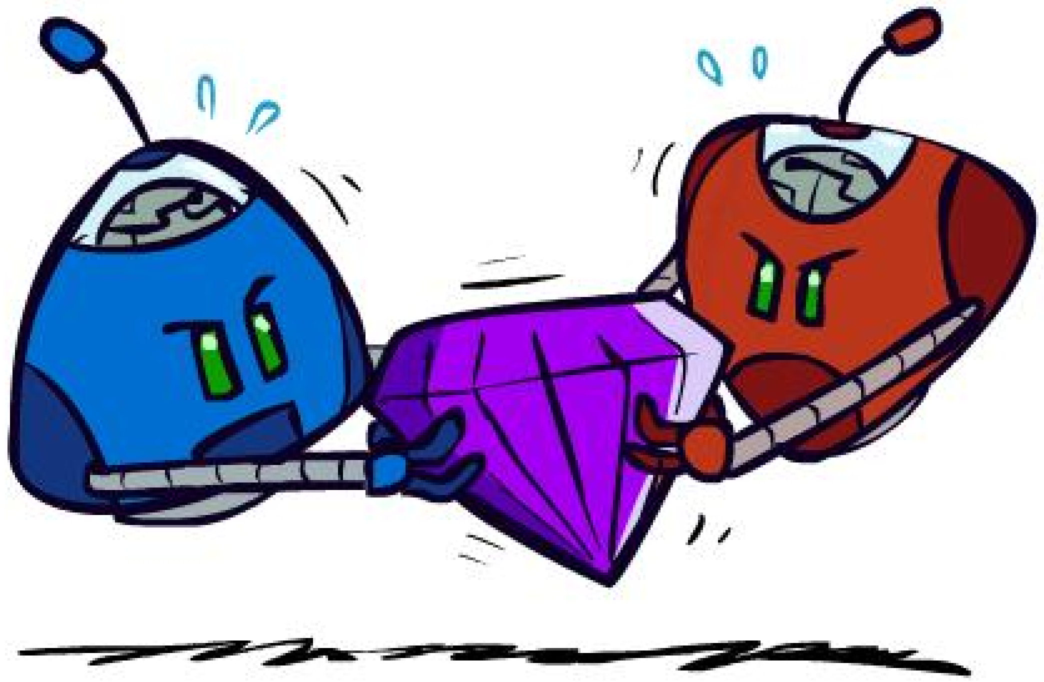
- **Pacman**



Outline

- Adversarial Games
- Adversarial Search
- Resource Limits
- Game Tree Pruning
- Uncertain Outcomes
- Other Game Types

Adversarial Games



Types of Games

- Many different kinds of games!

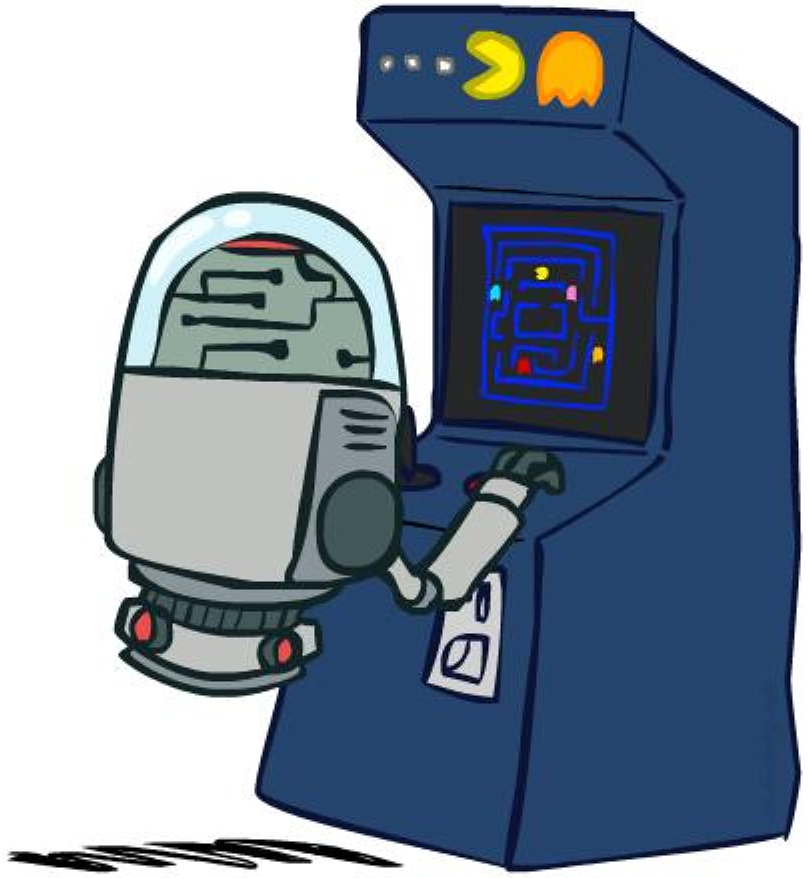
- Differences:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?

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

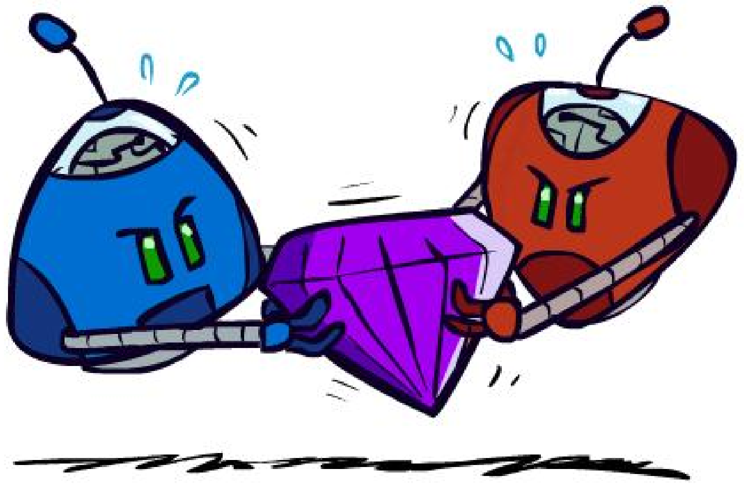
Deterministic Games

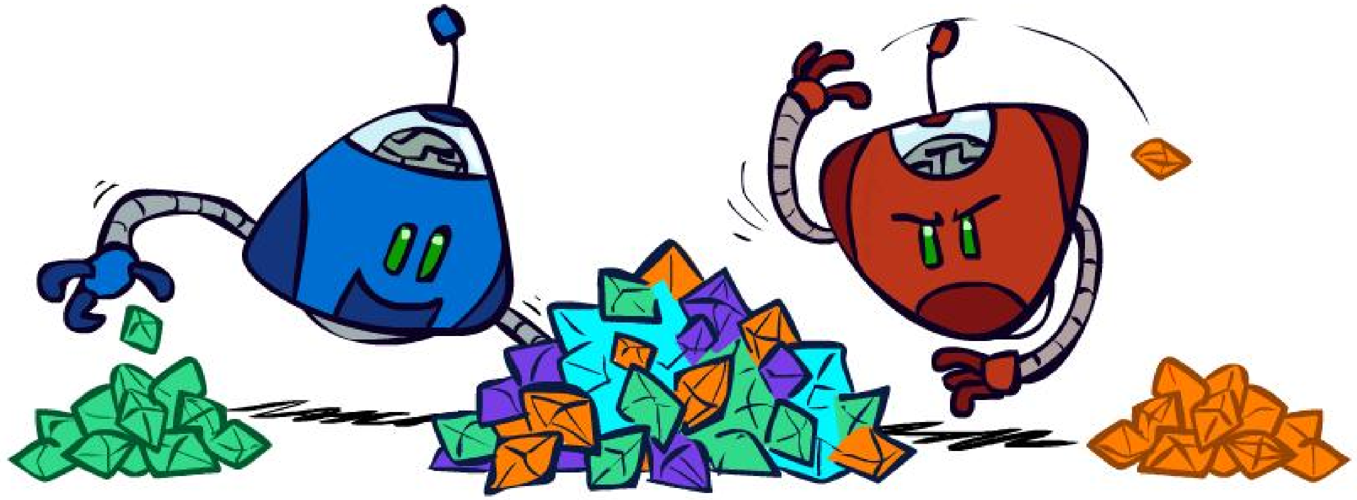
- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: SxA $\rightarrow$ S
  - Terminal Test: S $\rightarrow$ {t,f}
  - Terminal Utilities: SxP $\rightarrow$ R

- Solution for a player is a policy: S $\rightarrow$ A



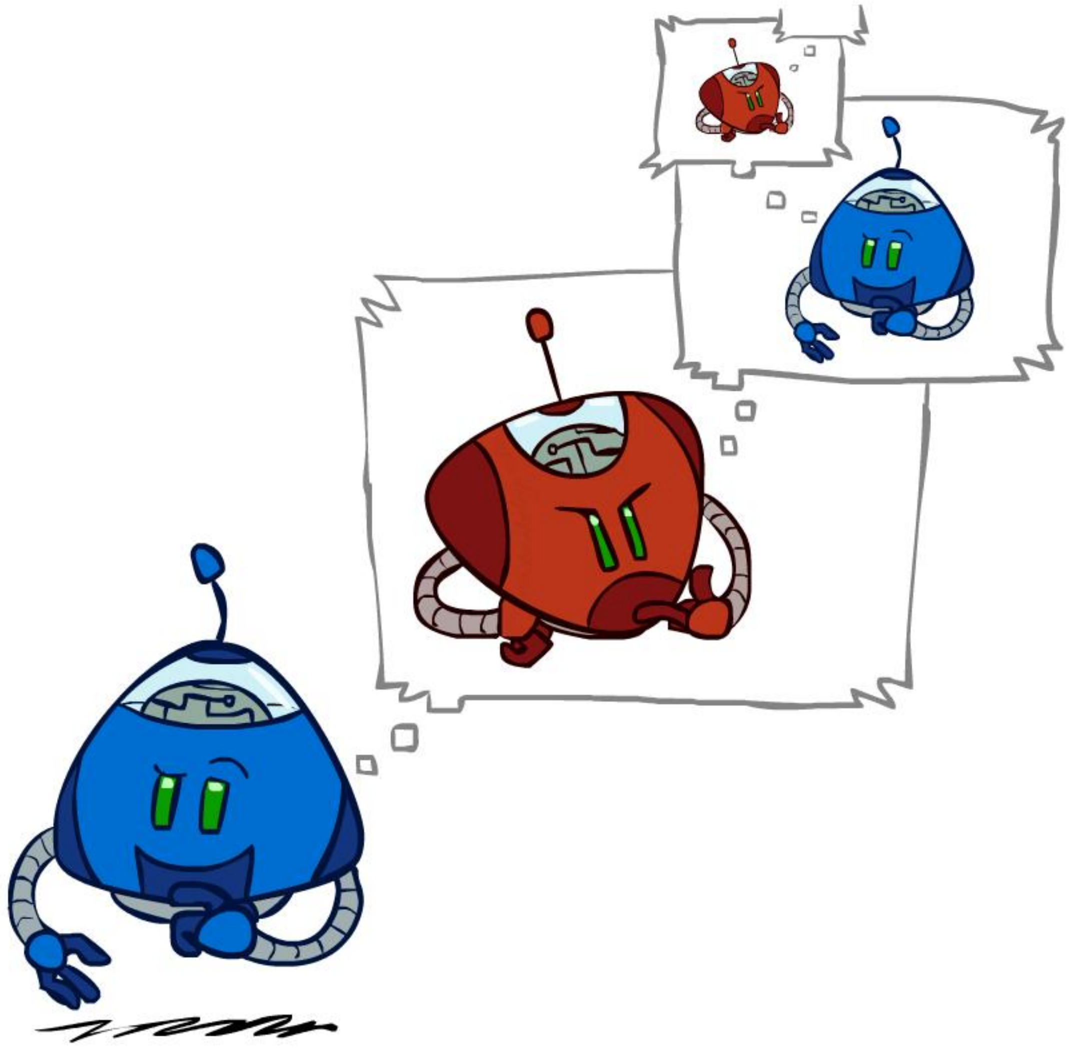
Zero-Sum Games



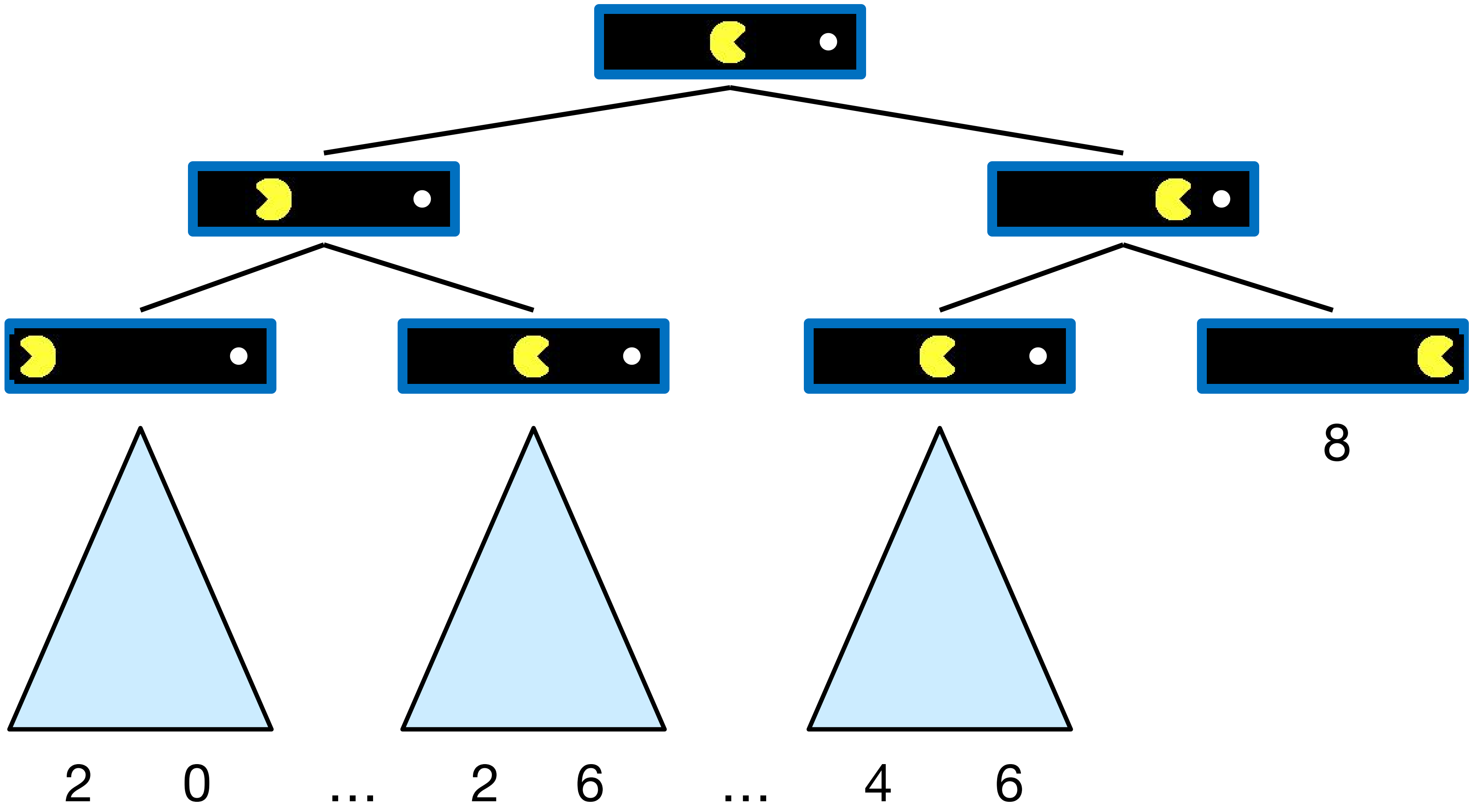


- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition
- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
  - More later on non-zero-sum games

Adversarial Search



Single-Agent Trees



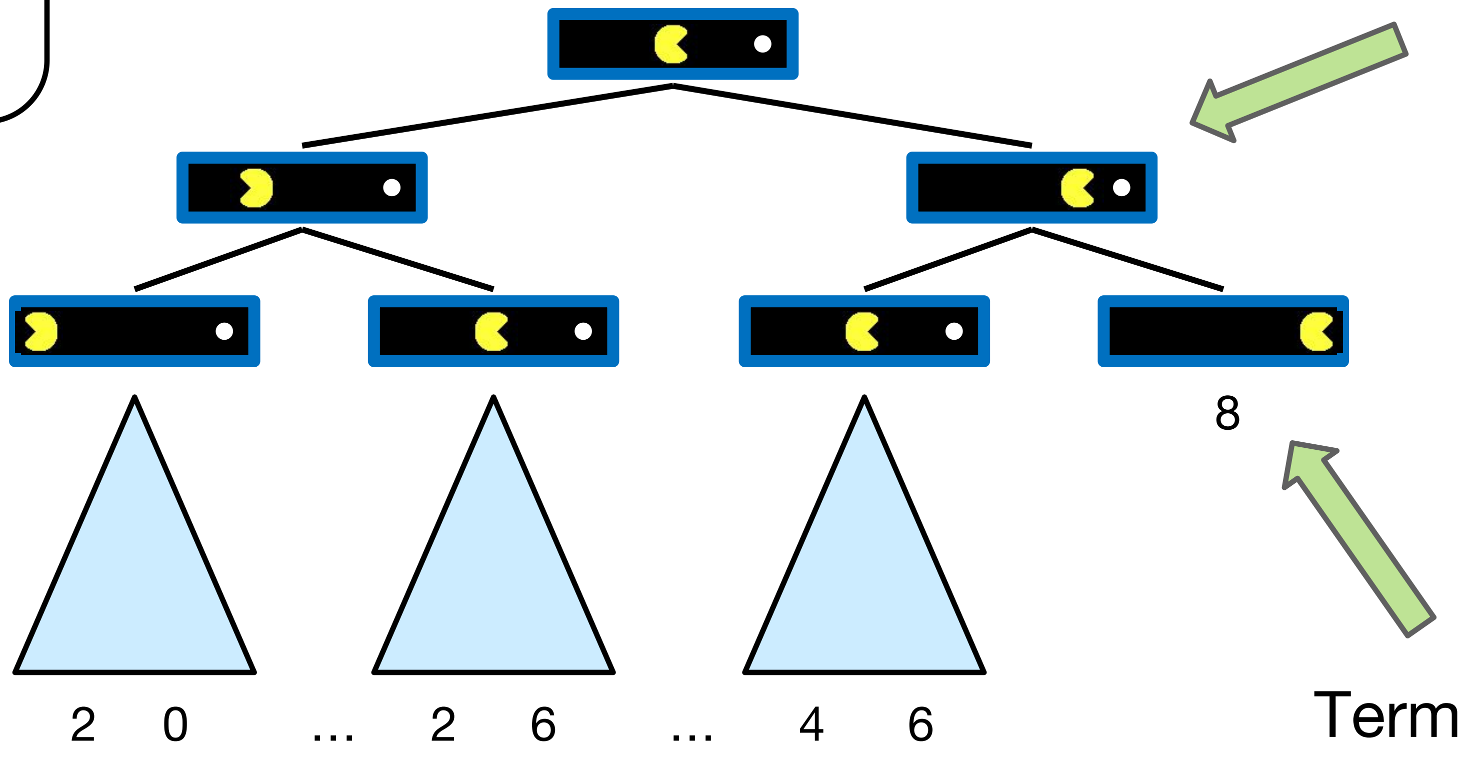
Value of a State

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

Policy: the agent should choose an action leading to the state with the largest value

Non-Terminal States:

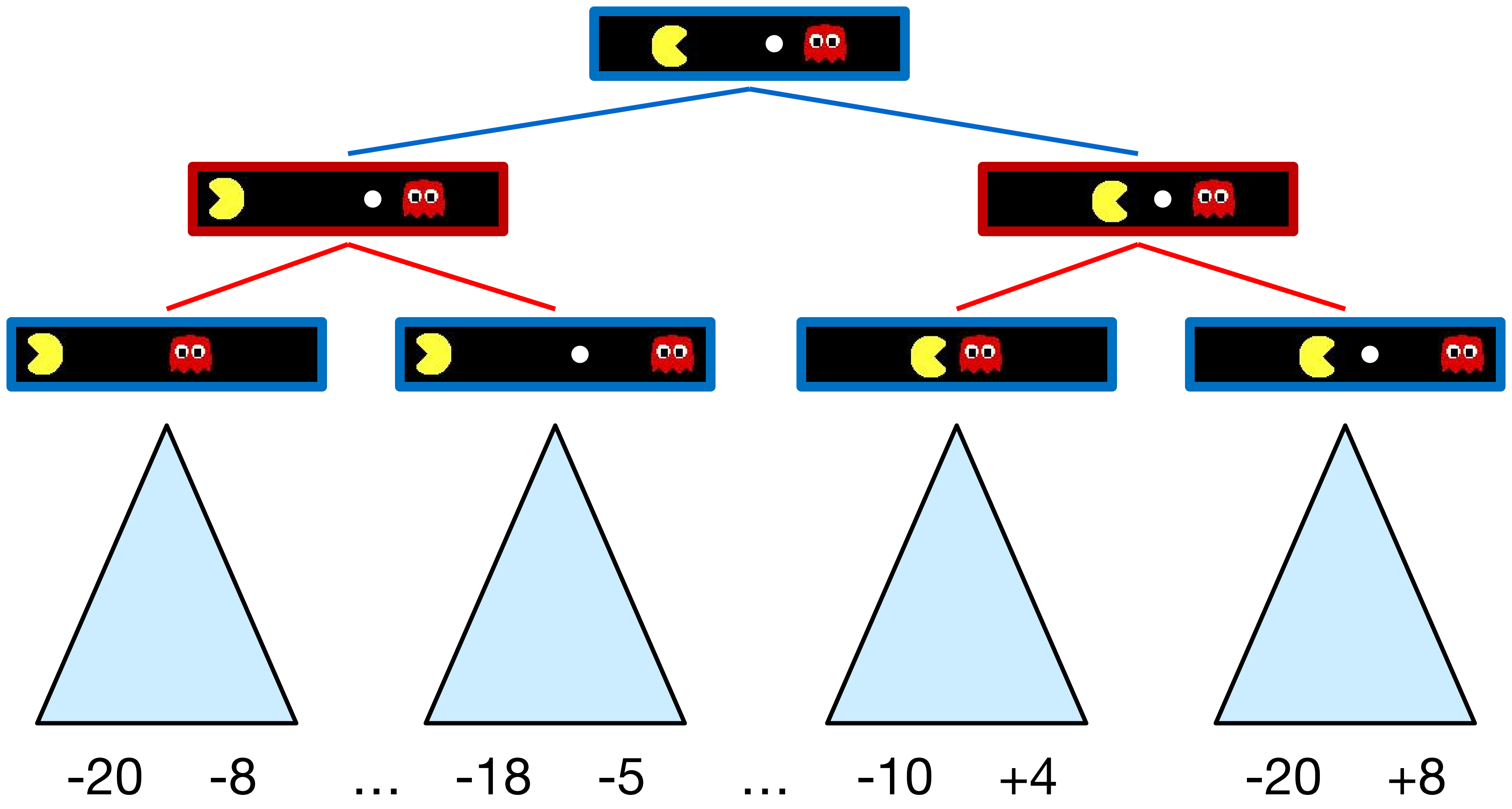
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:

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

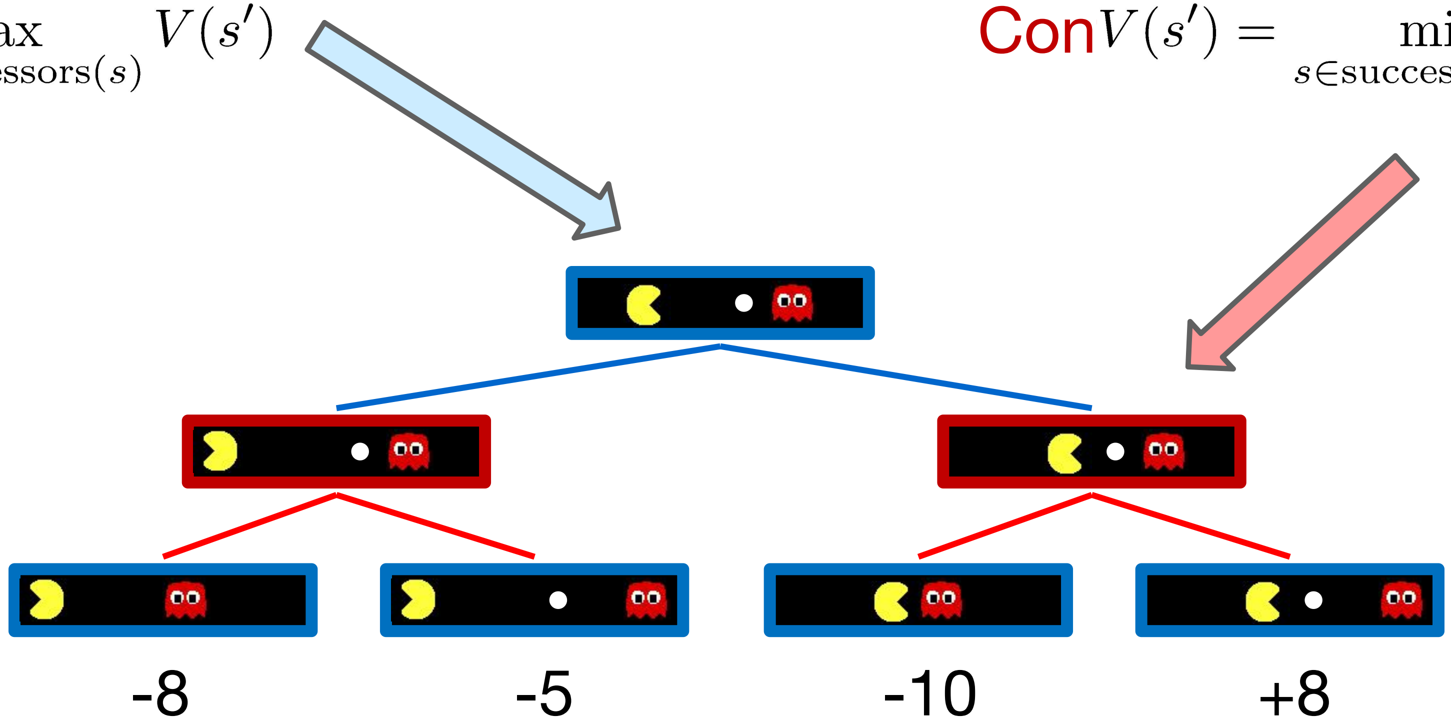
Adversarial Game Trees



Minimax Values

States Under Agent's
C $V(s) = \max_{s' \in \text{successors}(s)} V(s')$

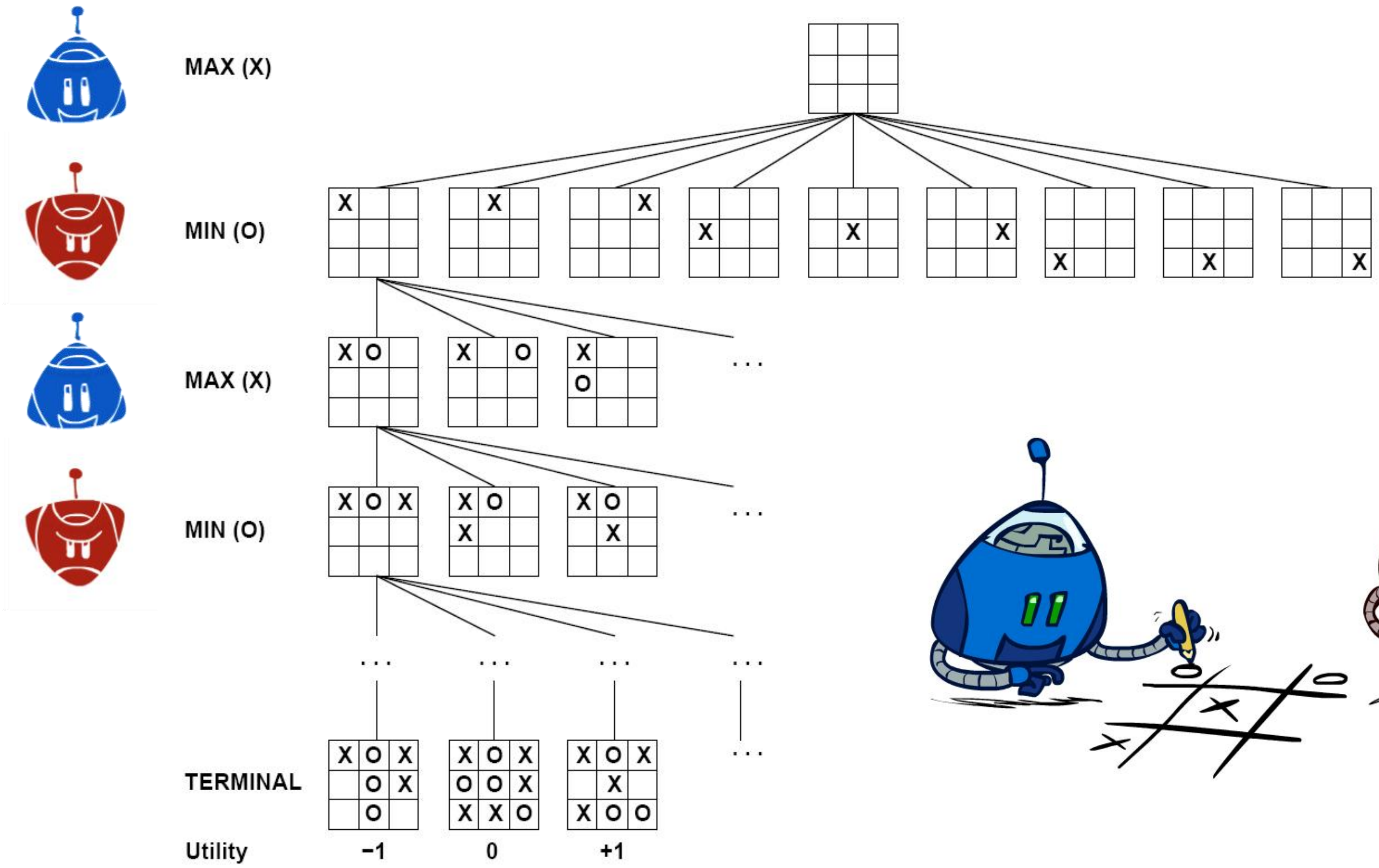
States Under Opponent's
Con $V(s') = \min_{s \in \text{successors}(s')} V(s)$

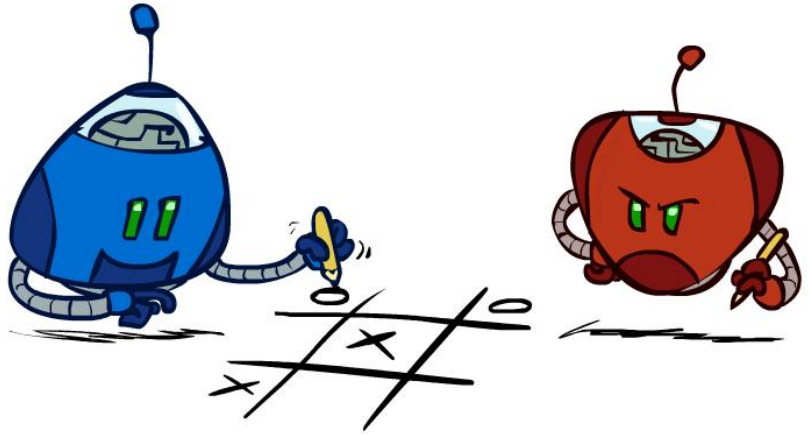


Policy: the agent should choose an action leading to the state with the largest value

Terminal States:
$V(s) = \text{known}$

Tic-Tac-Toe Game Tree

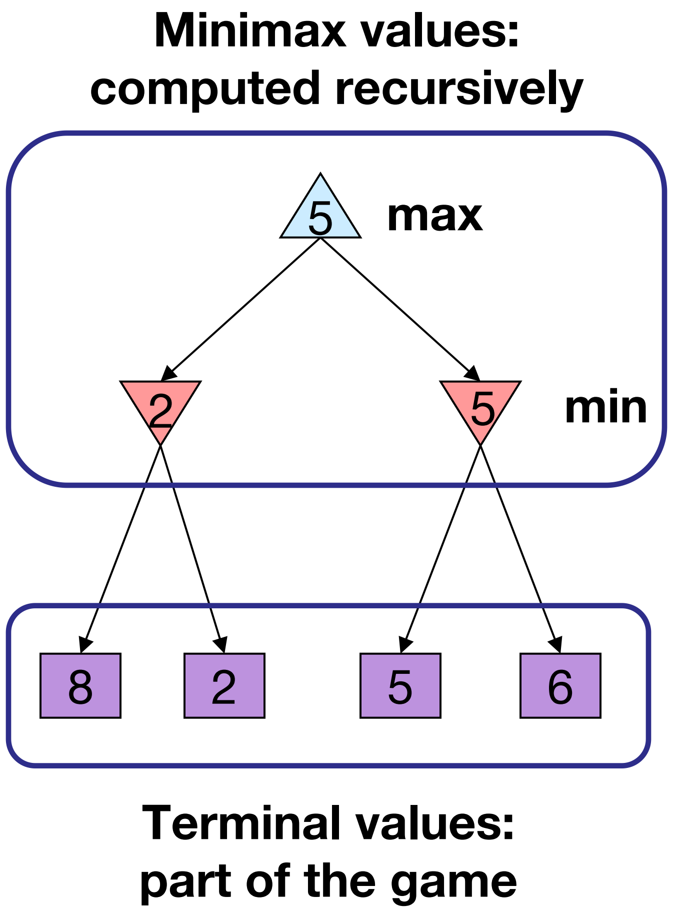




Adversarial Search (Minimax)

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

- Minimax search:
  - A state-space search tree
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary



Minimax values: computed recursively
5
max
2
5
min
8
2
5
6
Terminal values: part of the game

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
```

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

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

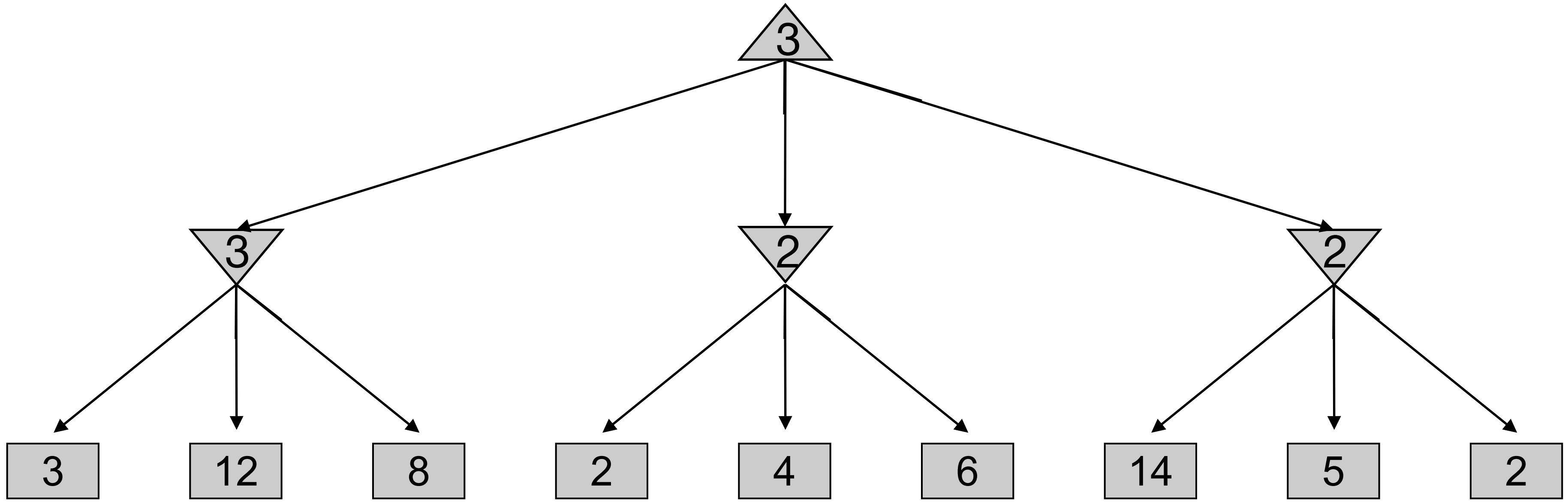
Minimax Implementation

```
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
```

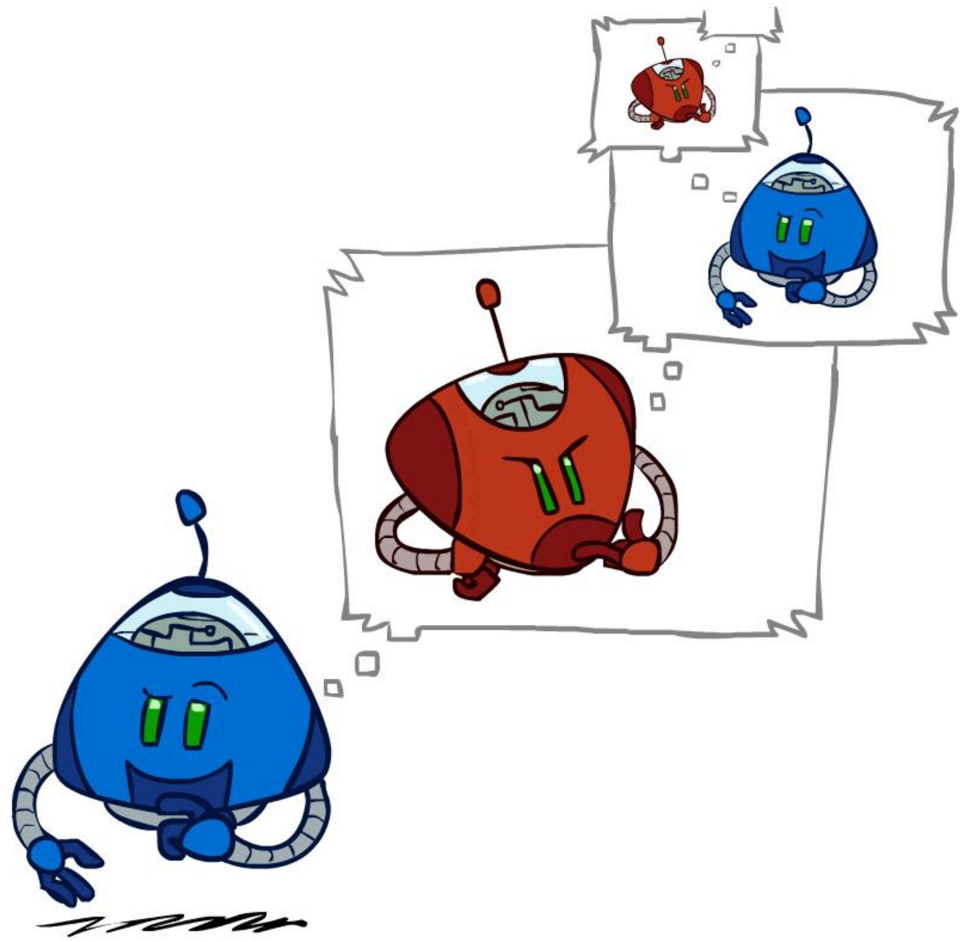
Minimax Example



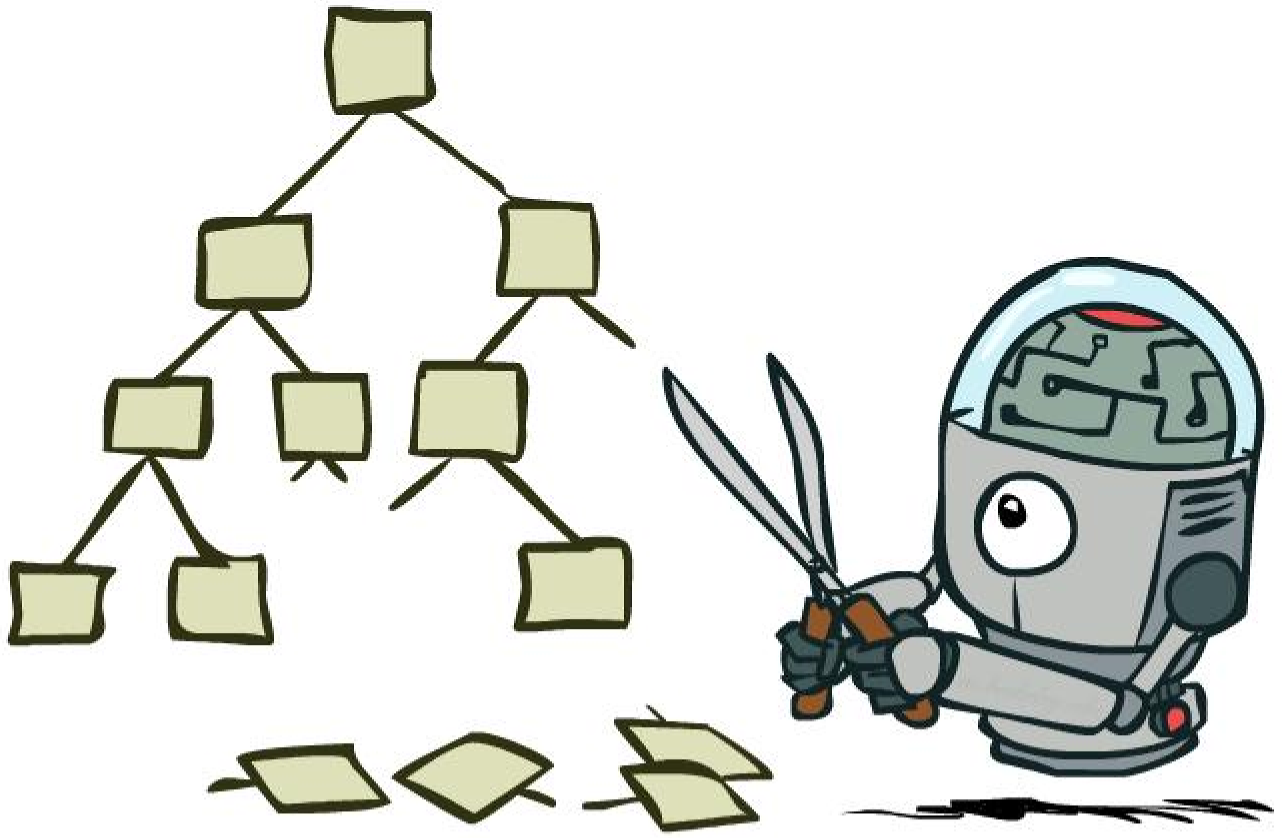
Minimax Efficiency

- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$

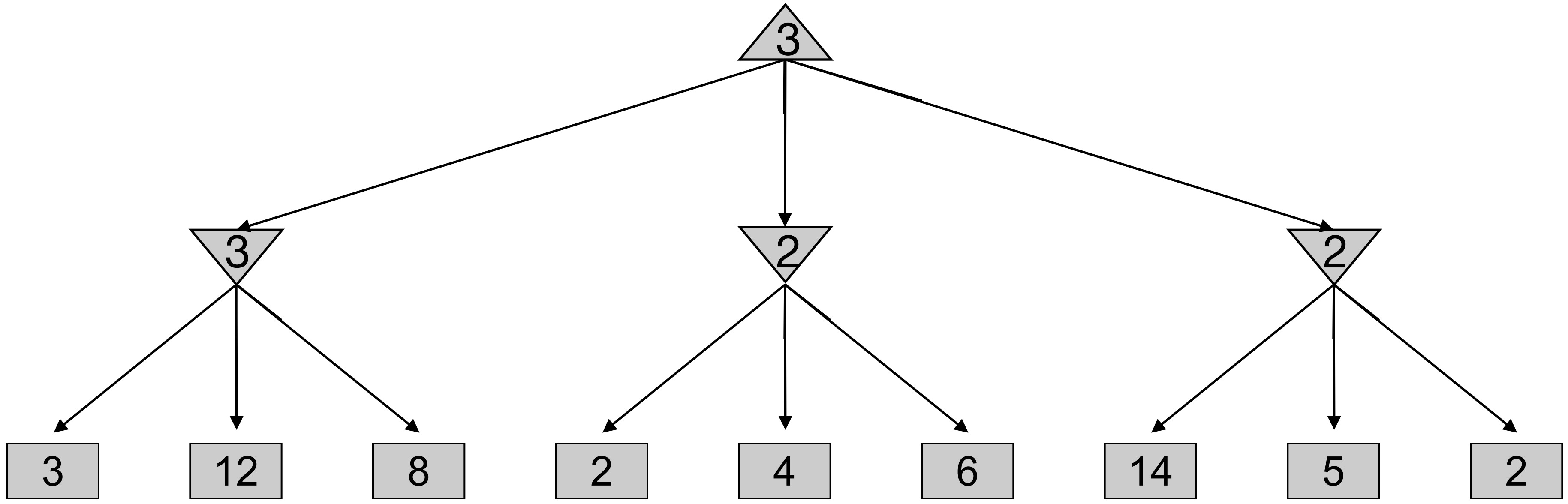
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?



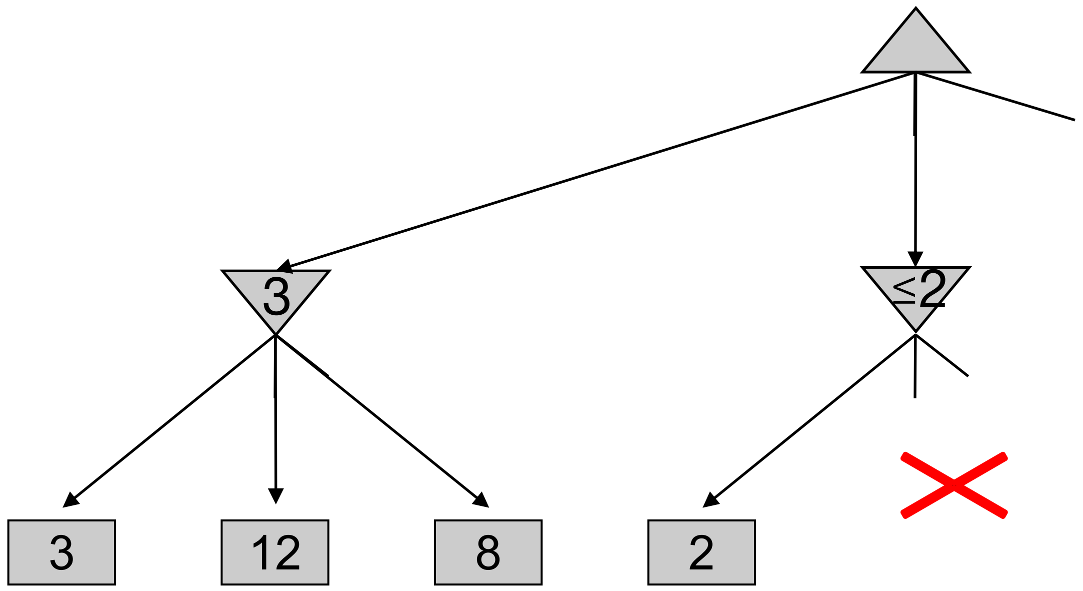
Game Tree Pruning



Minimax Example

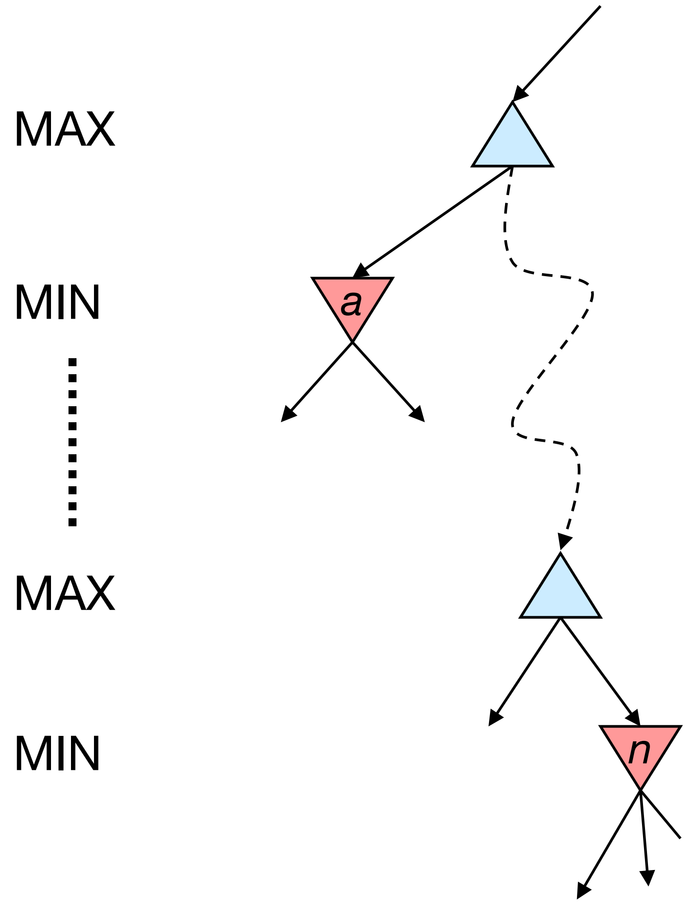


Minimax Pruning



Alpha-Beta Pruning

- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node *n*
  - We're looping over *n*'s children, so *n*'s estimate is decreasing
  - 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*, then we can stop considering *n*'s other children
  - Reason: if *n* is eventually chosen, then the nodes along the path shall all have the value of *n*, but *n* is worse than *a* and hence the path shall not be chosen at the MAX



- MAX version is symmetric

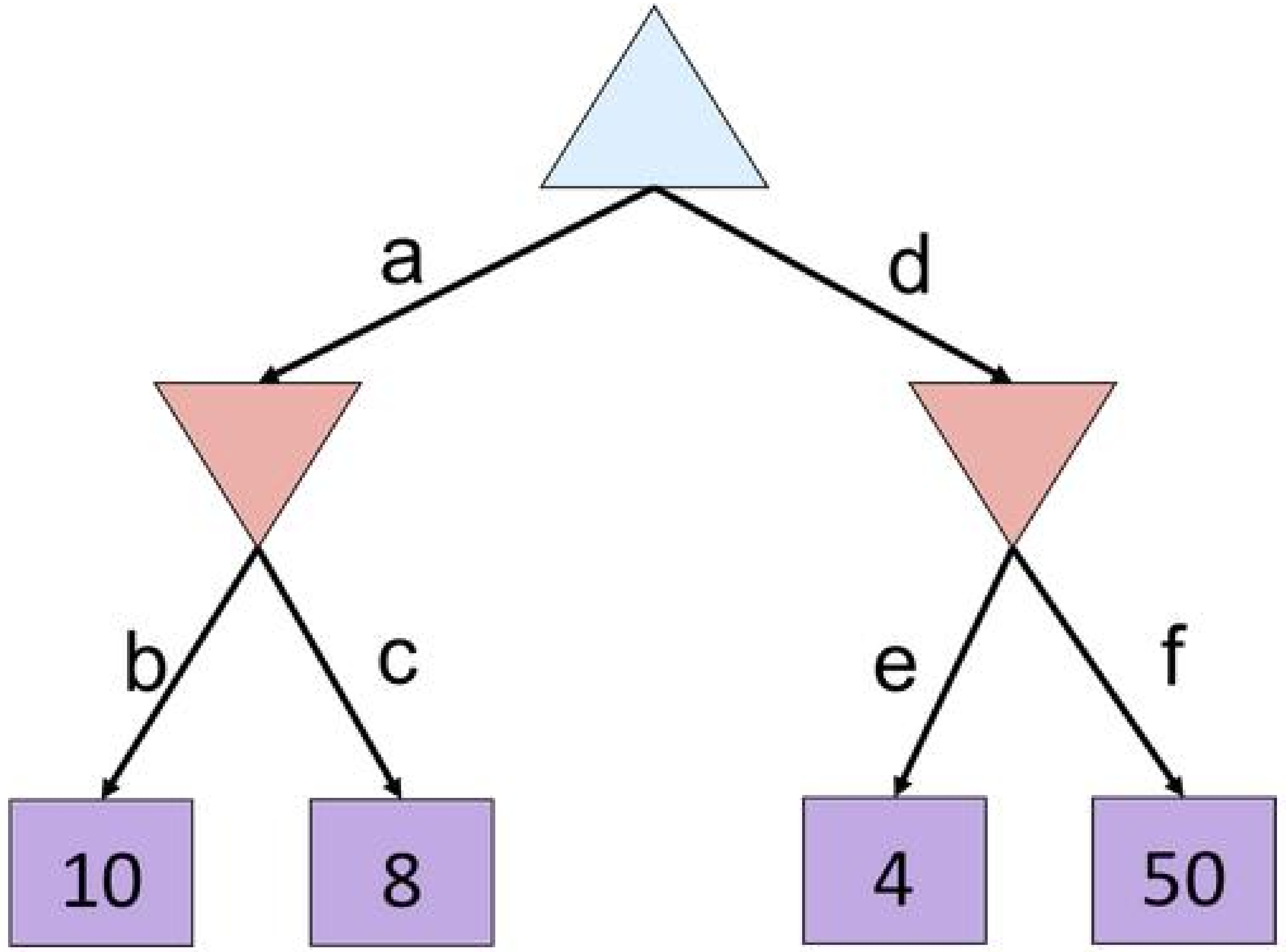
Alpha-Beta Implementation

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 Example



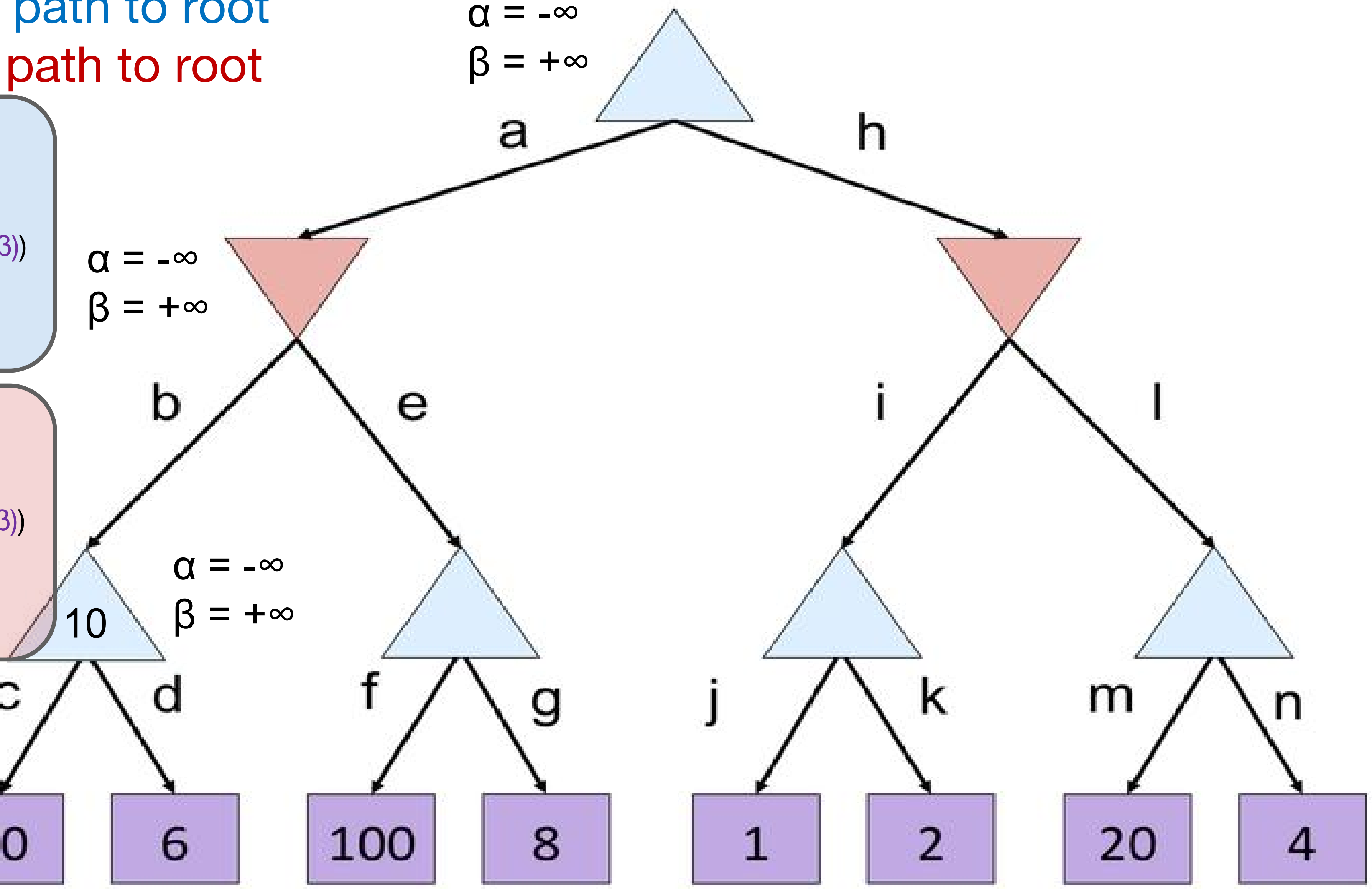
Alpha-Beta Example 2

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
```



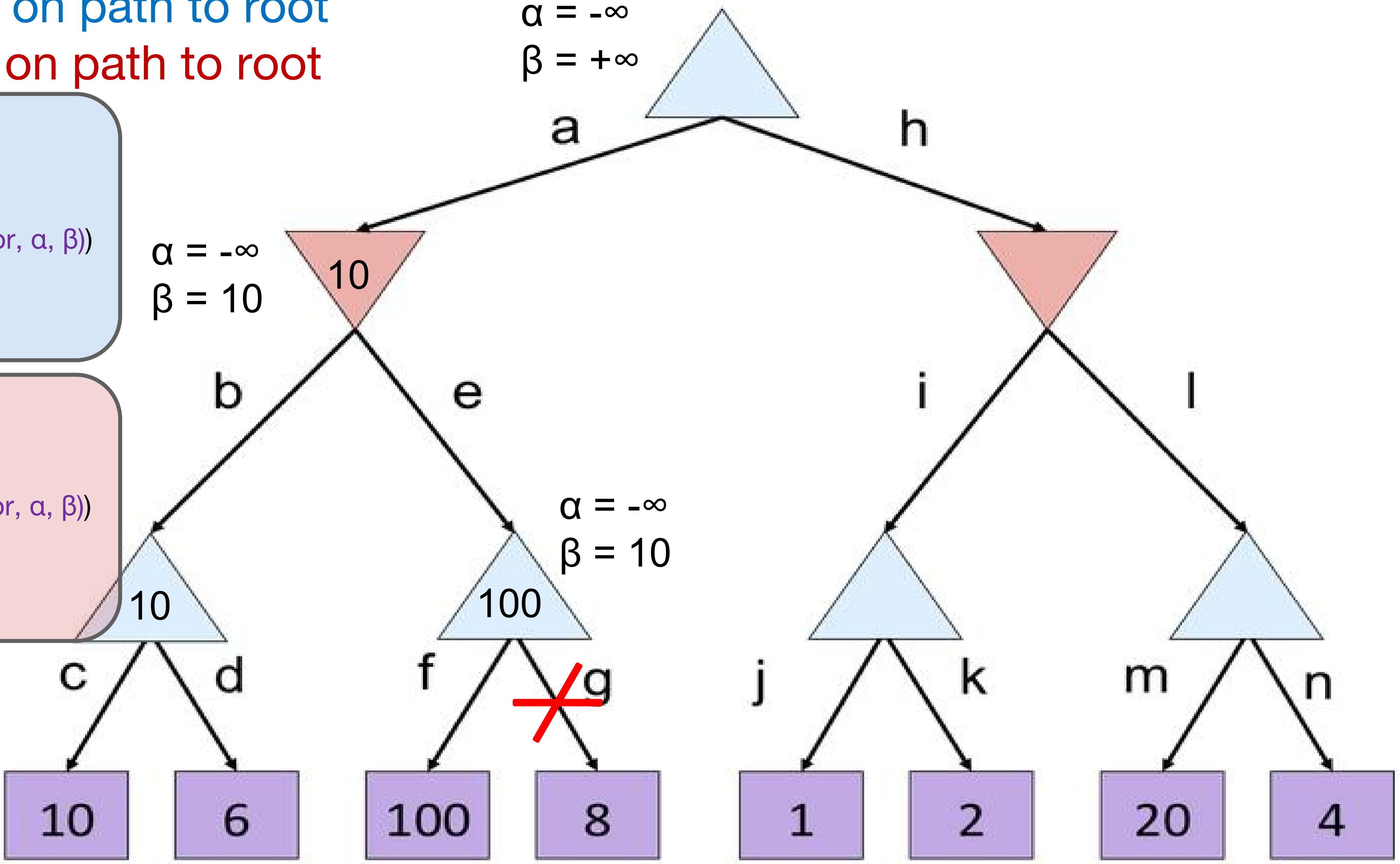
Alpha-Beta Example 2

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
```



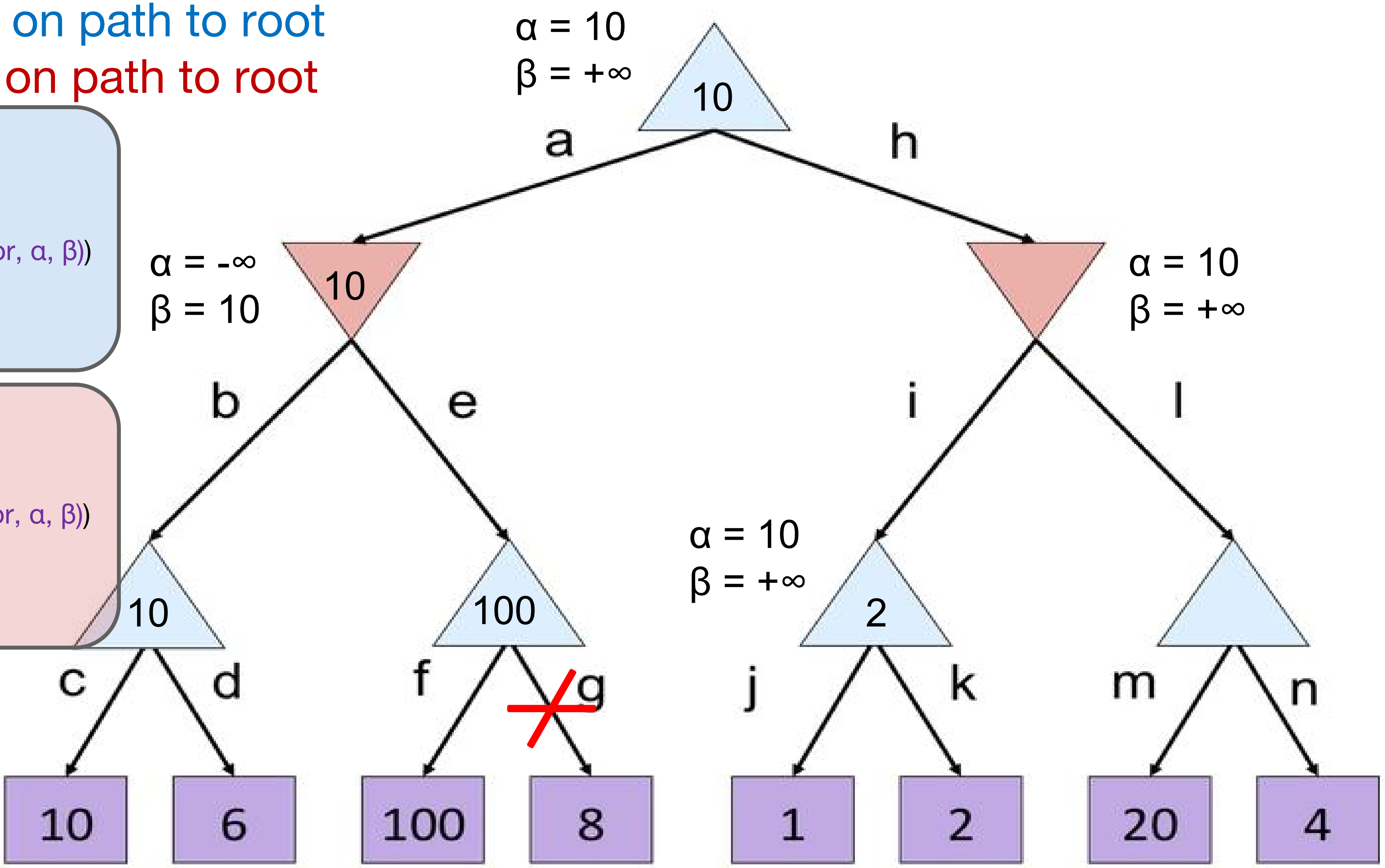
Alpha-Beta Example 2

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
```



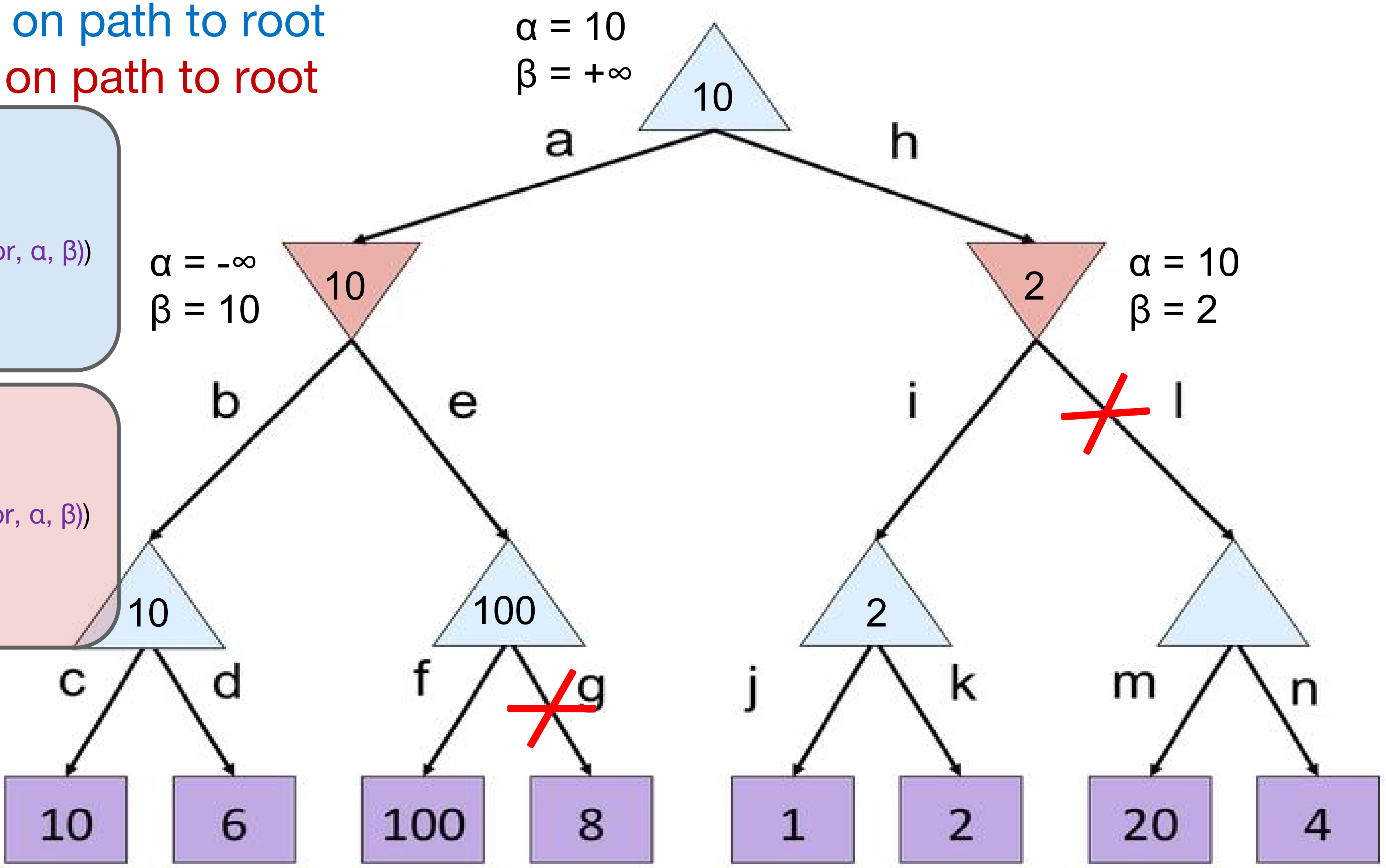
Alpha-Beta Example 2

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
```

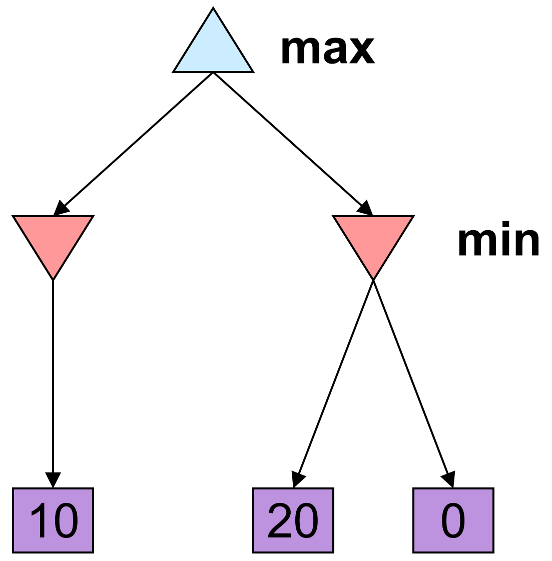
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```



Alpha-Beta Pruning Properties

- Good child ordering improves effectiveness of pruning

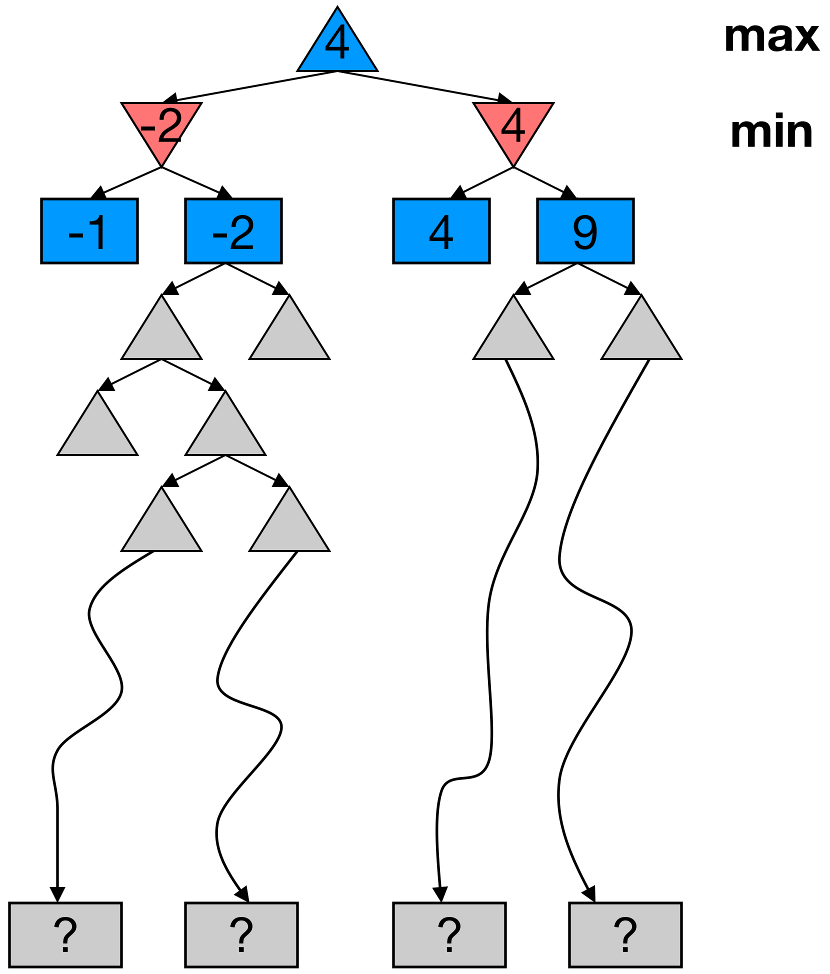
- With “perfect ordering”:
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!



Resource Limits

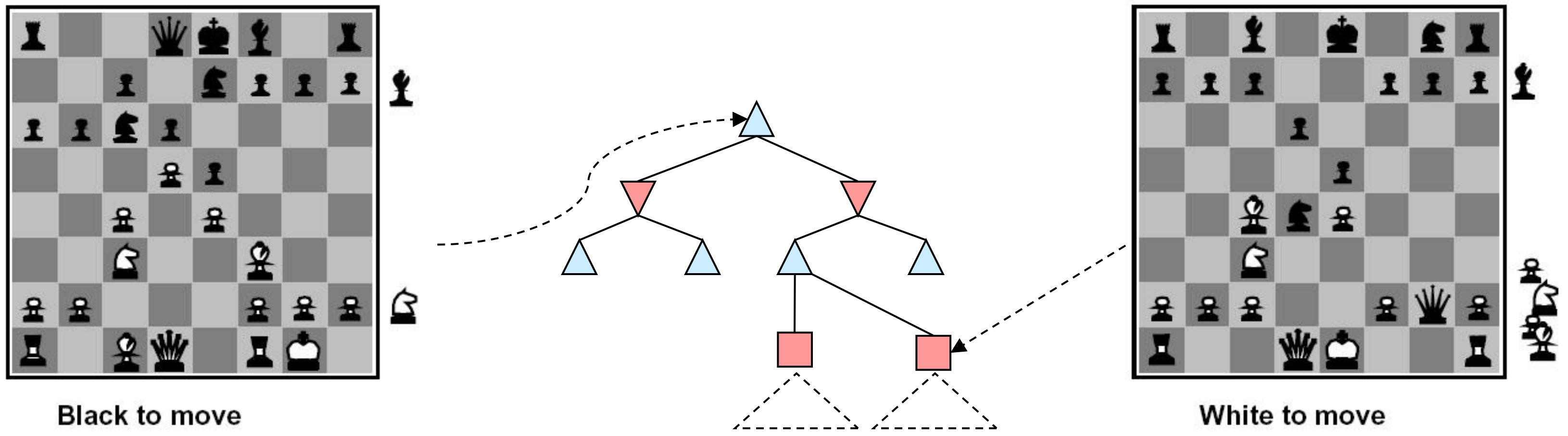
Resource Limits

- 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 depth makes a BIG difference



Evaluation Functions

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



- Ideal function: returns the actual minimax value of the position
- A simple solution in practice: weighted linear sum of features:

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

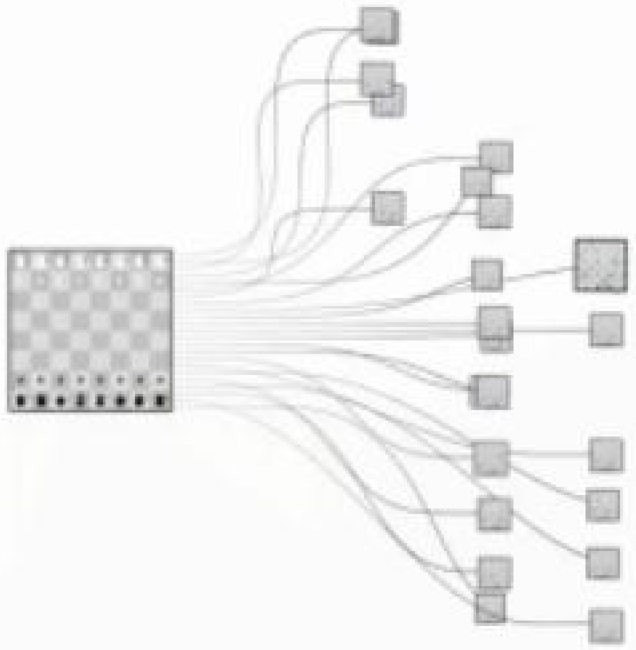
- e.g. $f_1(s)$ = (num white queens – num black queens), etc.

Evaluation Functions

- Recent advances
  - Monte Carlo Tree Search
    - Randomly choose moves until the end of game
    - Repeat for many many times
    - Evaluate the state based on these simulations, e.g., the winning rate
  - Convolutional Neural Network (value network in AlphaGo)
    - Trained from records of game plays to predict a score of the state

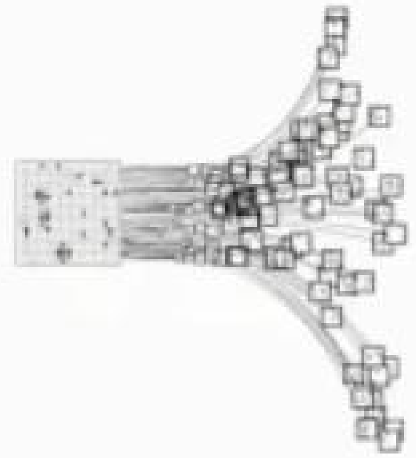
Branching Factor

- Chess



Branching Factor

- Go



Branching Factor

- Go has a branching factor of up to 361
- Idea: limit the branching factor by considering only good moves
  - AlphaGo uses a Convolutional Neural Network (policy network)
    - Trained from records of game plays
    - Trained using reinforcement learning
      - AlphaGo Zero uses RL only