Day 4
Informed Search, Adversarial Search

(some slides based on Downy, Sood, Dan Klein, Pieter Abbeel)

- Class business
  - Lab 1 due April 14
  - Lab 2 due April 21
  - Finish Informed search
- Adversarial search
- Discuss lab 1

A^* search

- Idea: Estimate total path through nodes and avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal (the evaluation of the desirability of n)

A* tree search algorithm –keeping track of visited nodes:

1. start with the initial node as curr
2. have I been to curr before? (is it in CLOSED)
3. is curr the goal?
4. if neither, expand curr - add children/successors to Frontier, add curr to CLOSED
5. choose a node curr from frontier according to the smallest f(n) & go to step 2

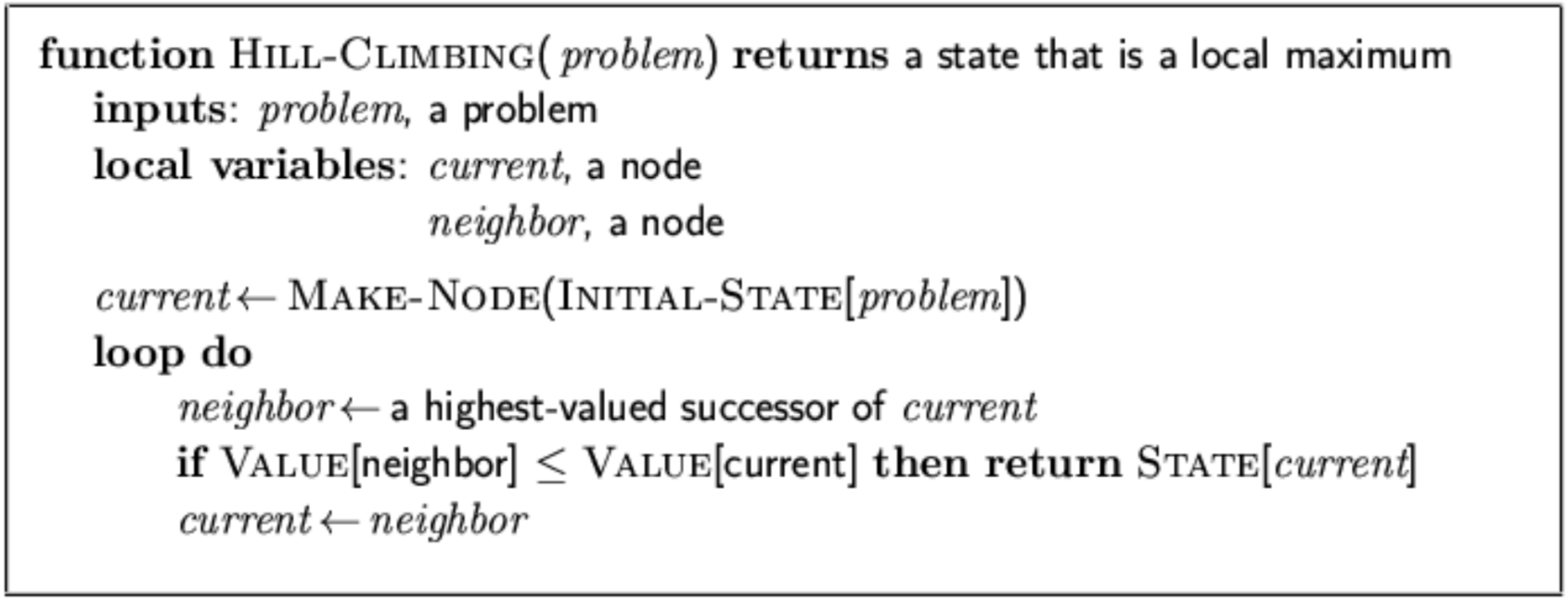
Note: This is basically* lab3!

Local search algorithms

- Many optimization problems the path to the goal is irrelevant, only finding the goal is important.
- State space= set of valid configurations
- Find configuration satisfying constraints
- In such cases, we can use local search algorithms
- For example:
  - Keep a single "current" state, and try to improve it

Greedy Hill climbing

- “like climbing Everest in thick fog with amnesia” *Lab 2*



```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] $\leq$ VALUE[current] then return STATE[current]
        current ← neighbor
```

Greedy Hill climbing random restart

- If stuck at non-goal, choose a new starting state, re-run hill climbing
  - If probability of solving is P
    - Expected restarts till goal found is 1/P
- Can solve the million queens problem in under a min

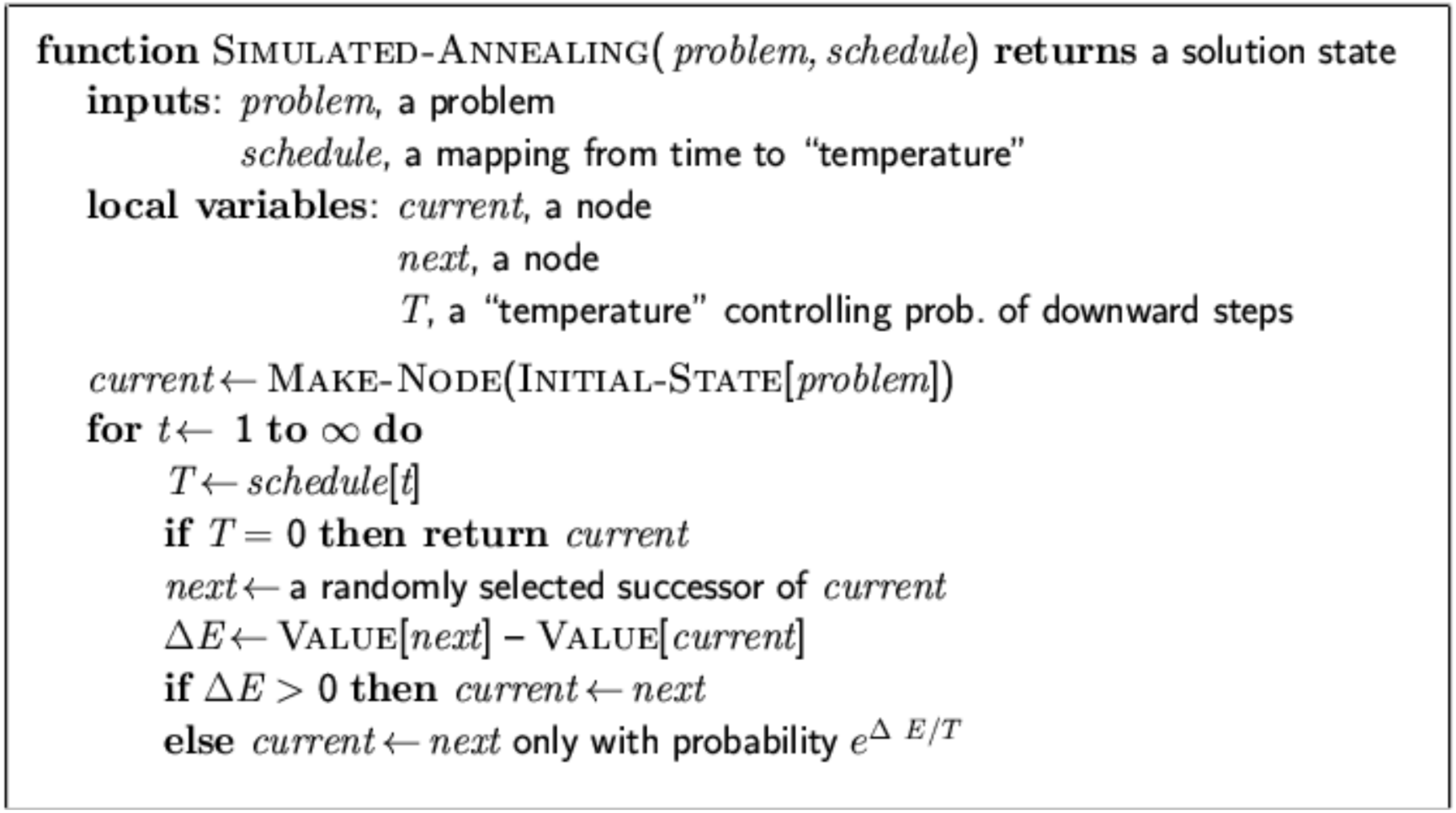
Local beam search

- Keep track of K states rather than 1

- Start with K randomly generated states
- At each iteration, generate all successors for all K states
- If any one is the goal state, stop;
  - Else select the K best successors and repeat
- Stochastic beam search
  - Choose successors at random, with higher likelihood to successors with higher value.

Simulated annealing

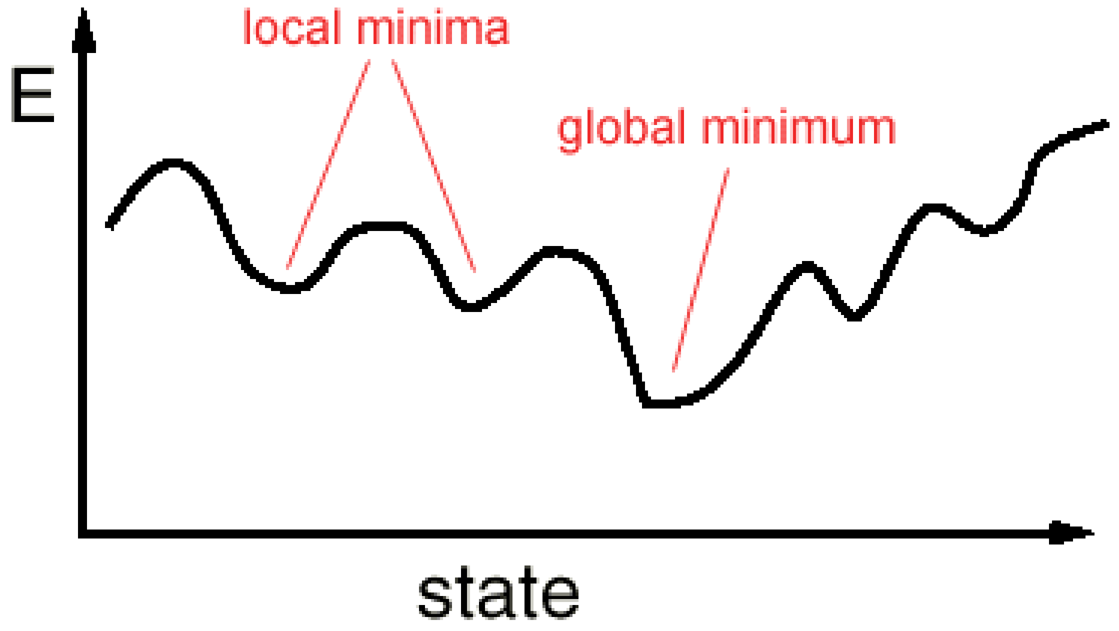
- Idea: escape local maxima by allowing some “bad” moves, but gradually decrease how frequently you allow this



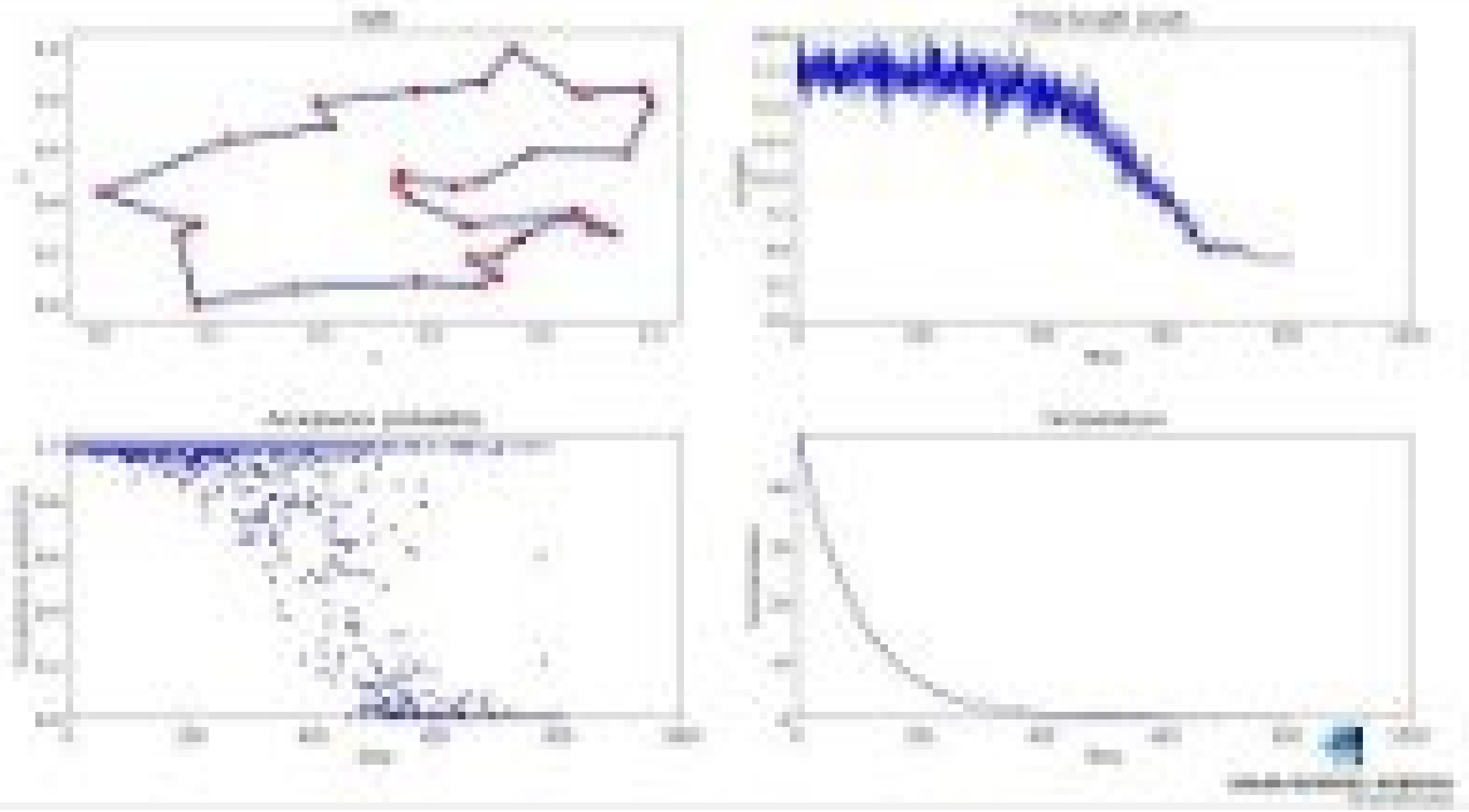
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to “temperature”
    local variables: current, a node
                     next, a node
                     $T$, a “temperature” controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for $t \leftarrow 1$ to $\infty$ do
        $T \leftarrow$ schedule[$t$]
        if $T = 0$ then return current
        next ← a randomly selected successor of current
        $\Delta E \leftarrow$ VALUE[next] – VALUE[current]
        if $\Delta E > 0$ then current ← next
        else current ← next only with probability $e^{\Delta E/T}$
```

Properties of simulated annealing search

- One can prove: if T decreases slowly enough, then simulated annealing will find a global optimum with probability approaching 1



Simulated annealing example

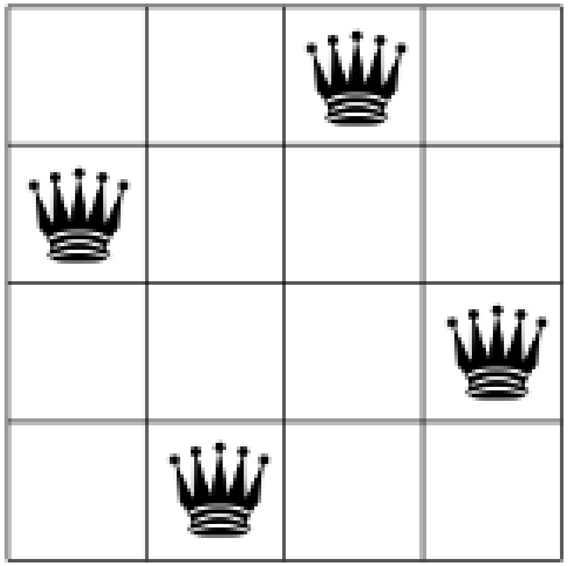


Genetic algorithms

1. Start with K randomly generated states (population)
2. Evaluate the fitness for all states in population using a fitness function
3. Generate next generation of population through reserving, selection, crossover, and mutation.
4. Goto step 2.

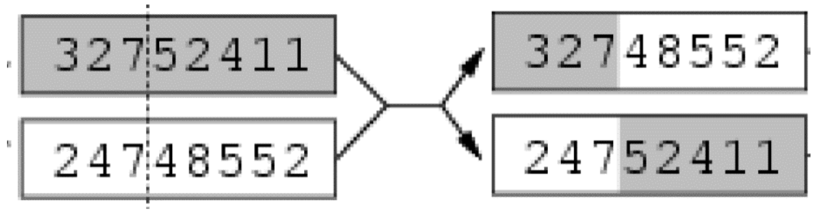
Individuals

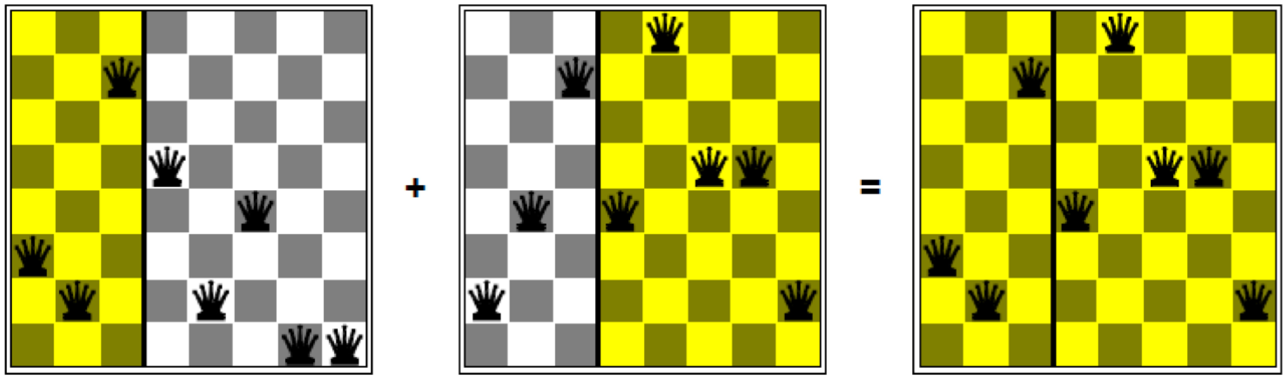
- Each individual consists of an encoded description of its state
- The encoded description is evaluated by the fitness function
- Example:N-queens
  - description
    - 2,4,1,3
  - Fitness function
    - Number of queens attacking



Generate new population

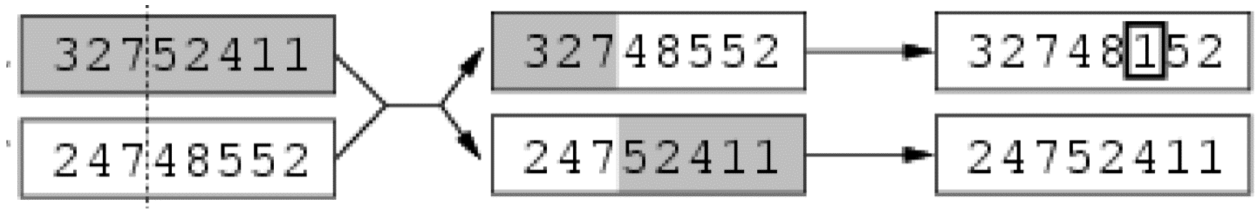
1. Select 2 individuals based on fitness
   - Higher the fitness, more likely the selection (survival of the fittest)
2. Create "offspring" using crossover





Generate new population

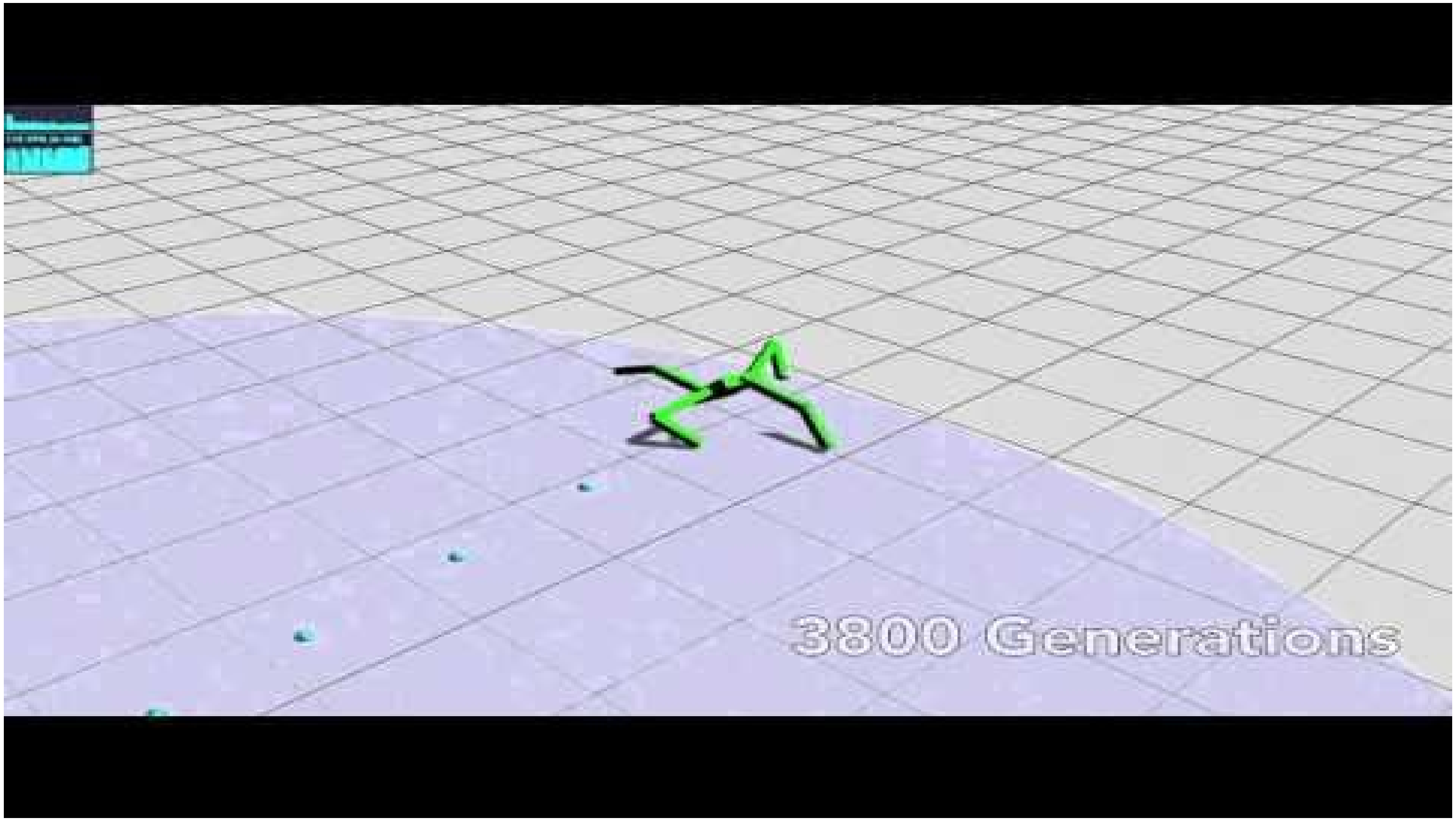
1. Select 2 individuals based on fitness
   - Higher the fitness, more likely the selection (survival of the fittest)
2. Create "offspring" using crossover
3. Modify offspring using mutation



Generate new population

1. Select 2 individuals based on fitness
   - Higher the fitness, more likely the selection (survival of the fittest)
2. Create “offspring” using crossover
3. Modify offspring using mutation
4. Add offspring to next population
5. While (population not full), goto 1

Genetic algorithm example



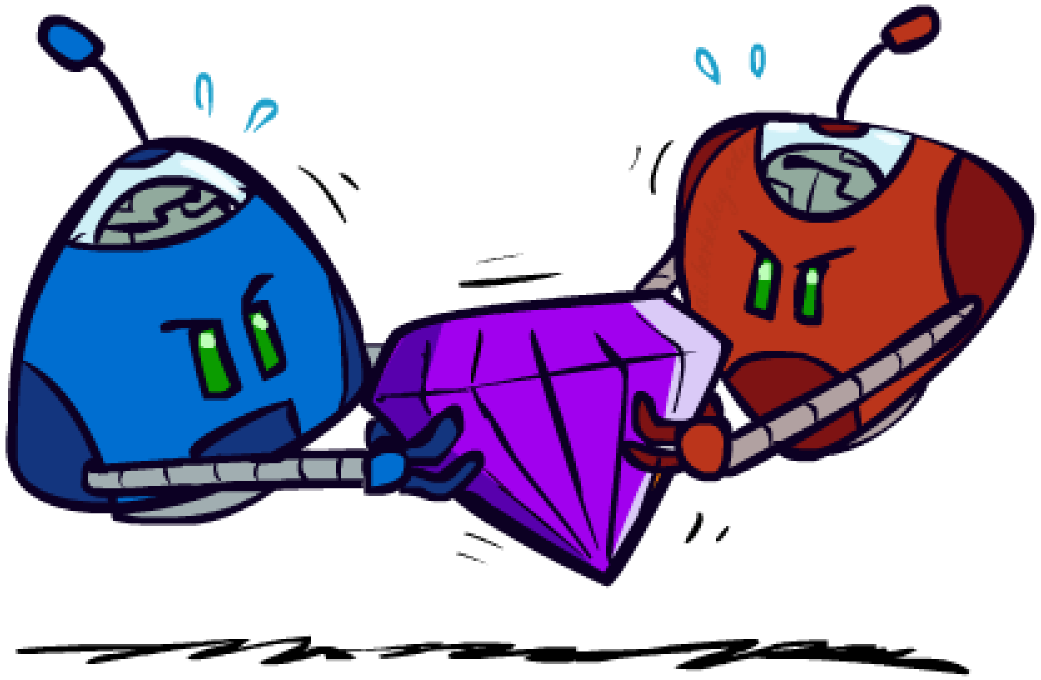
Genetic algorithm example

- Karl Sims 1994

Summary

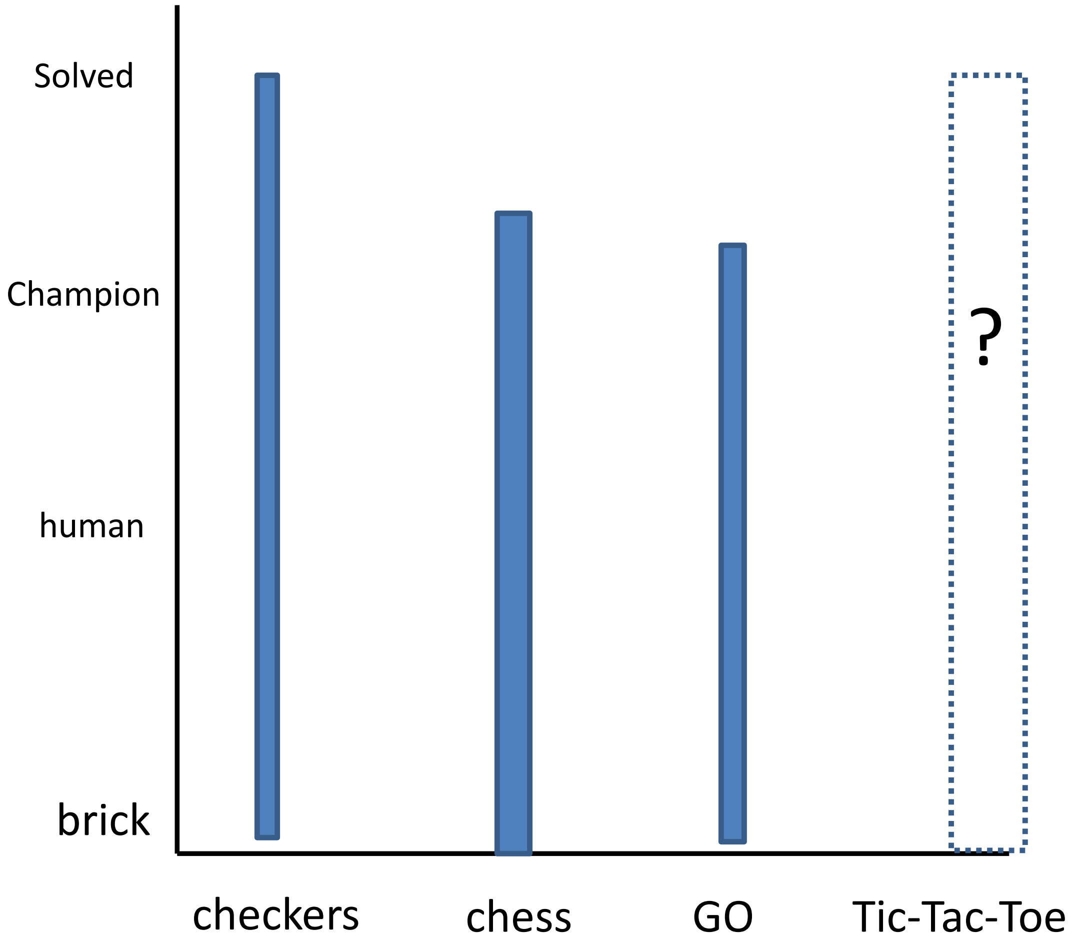
- Informed search
  - Greedy best first
  - A*
- Local search
  - Greedy hill climb
  - Simulated annealing
  - Local beam
  - Genetic algorithm
- Next: adversarial search

Adversarial Search



Game playing state of art

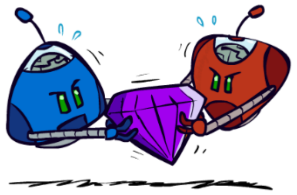
- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M 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:** Alpha Go Defeats human champions in 2016. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.
- **Tic-Tac-Toe-??**



Why games?

- Help understand multi-agent and competitive multi-agent environments
- Reason about stochastic systems
- Forces us to make imperfect, real-time decisions
  - We can’t always evaluate the entire tree.

Types of Games



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

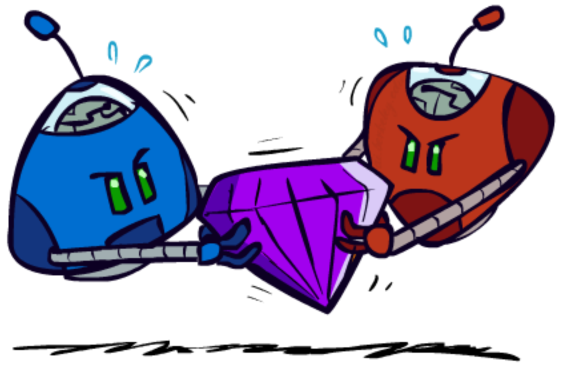
- Games vs search
  - Search – sequence of actions
  - Games – Policy (given observation, what is the right thing to do)

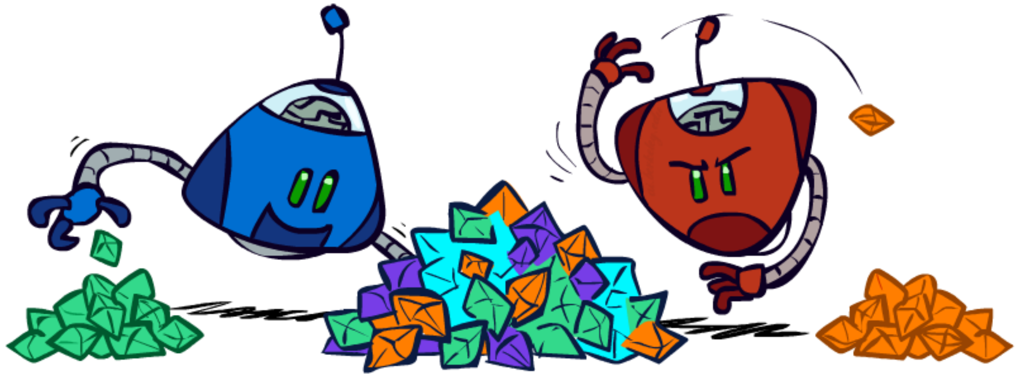
Today: Deterministic games

- 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





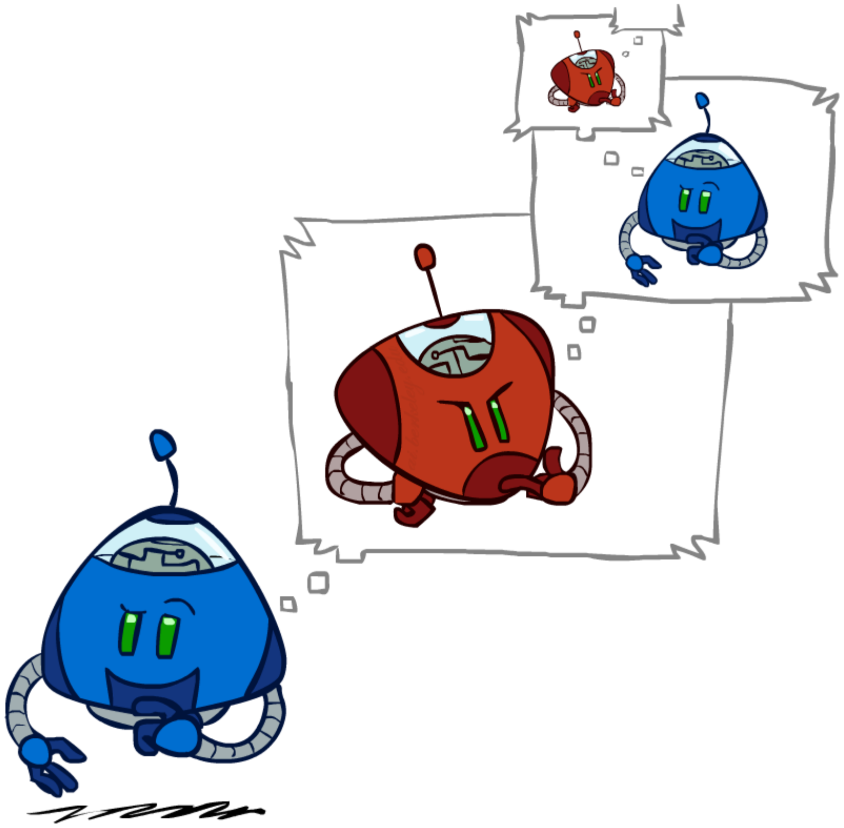
- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition

- 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

What is going on when deciding next move?

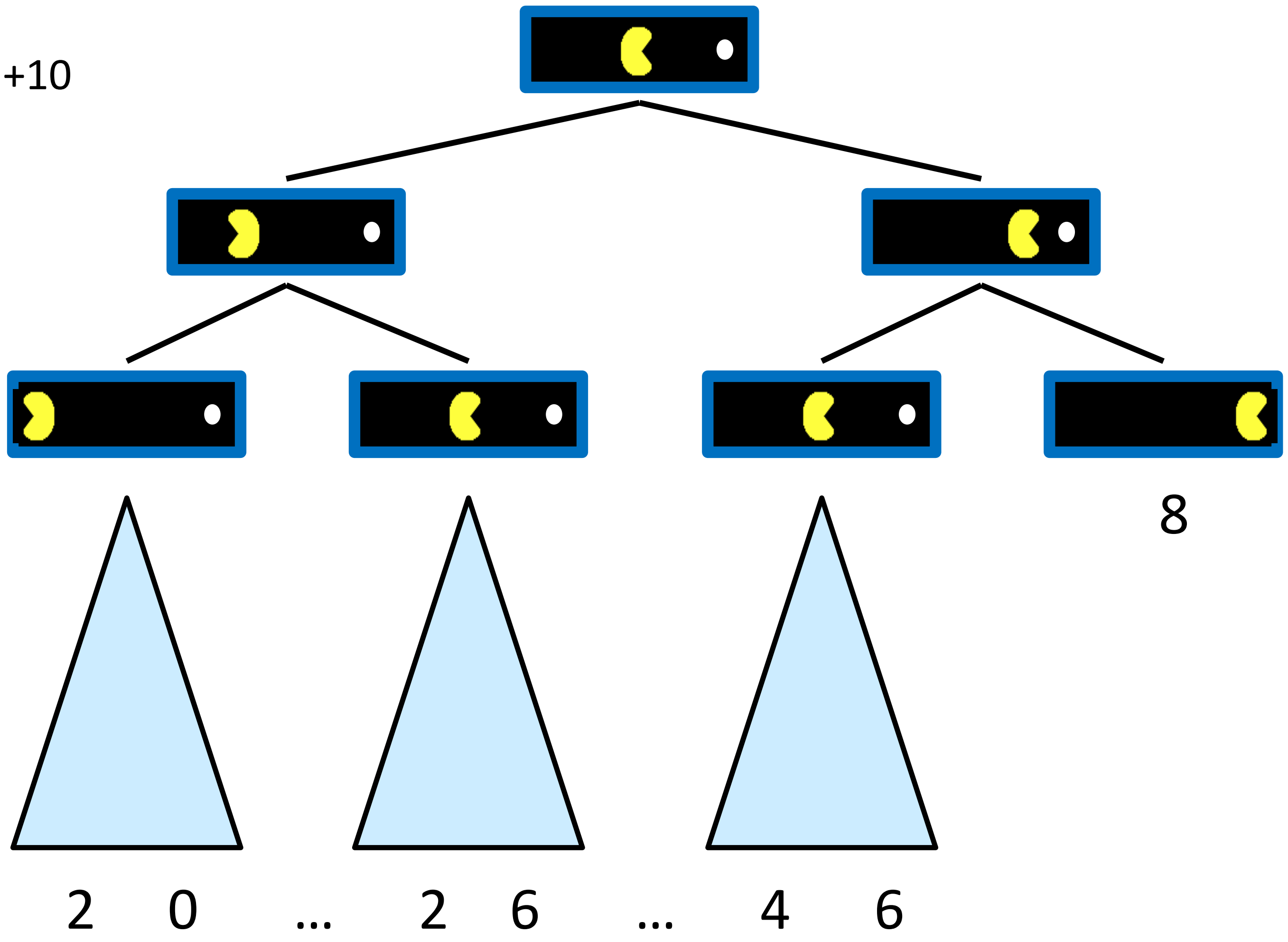
1. What will red do in move 2
2. What will blue do in move 3
3. What will red do in move 4
4. .
5. .
6. .
7. ..
8. .



HD

Single-Agent Trees

Rules:
Move -1
Eat pellet +10

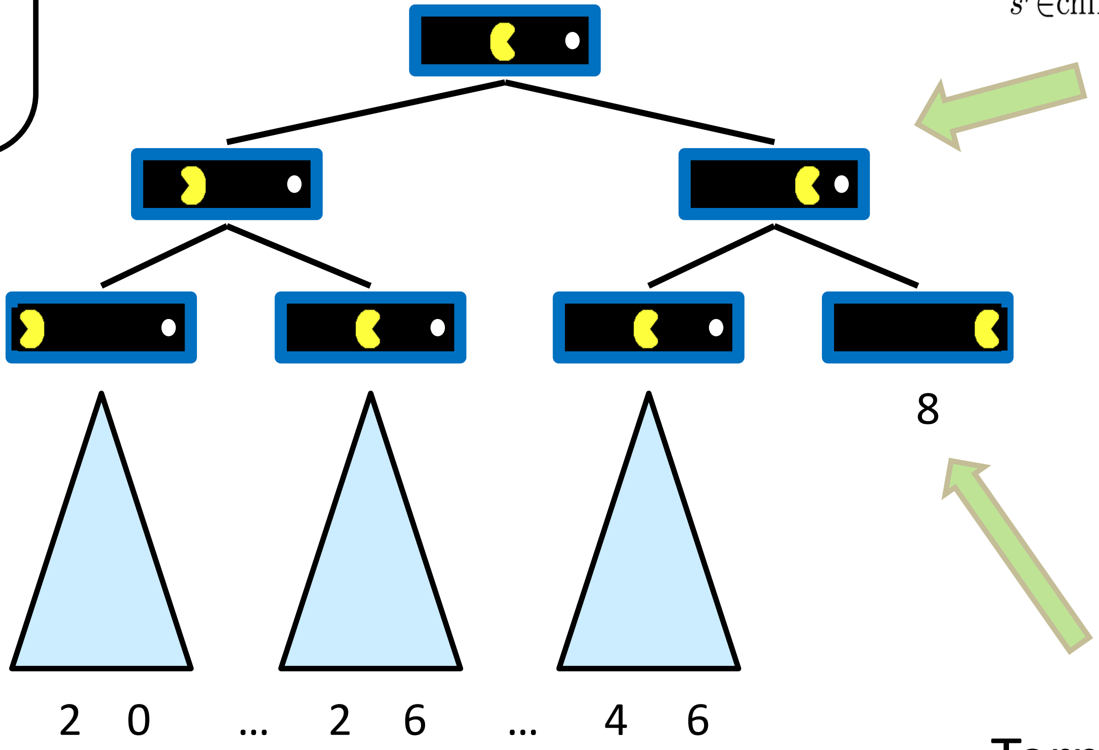


Value of a State

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

Non-Terminal States:

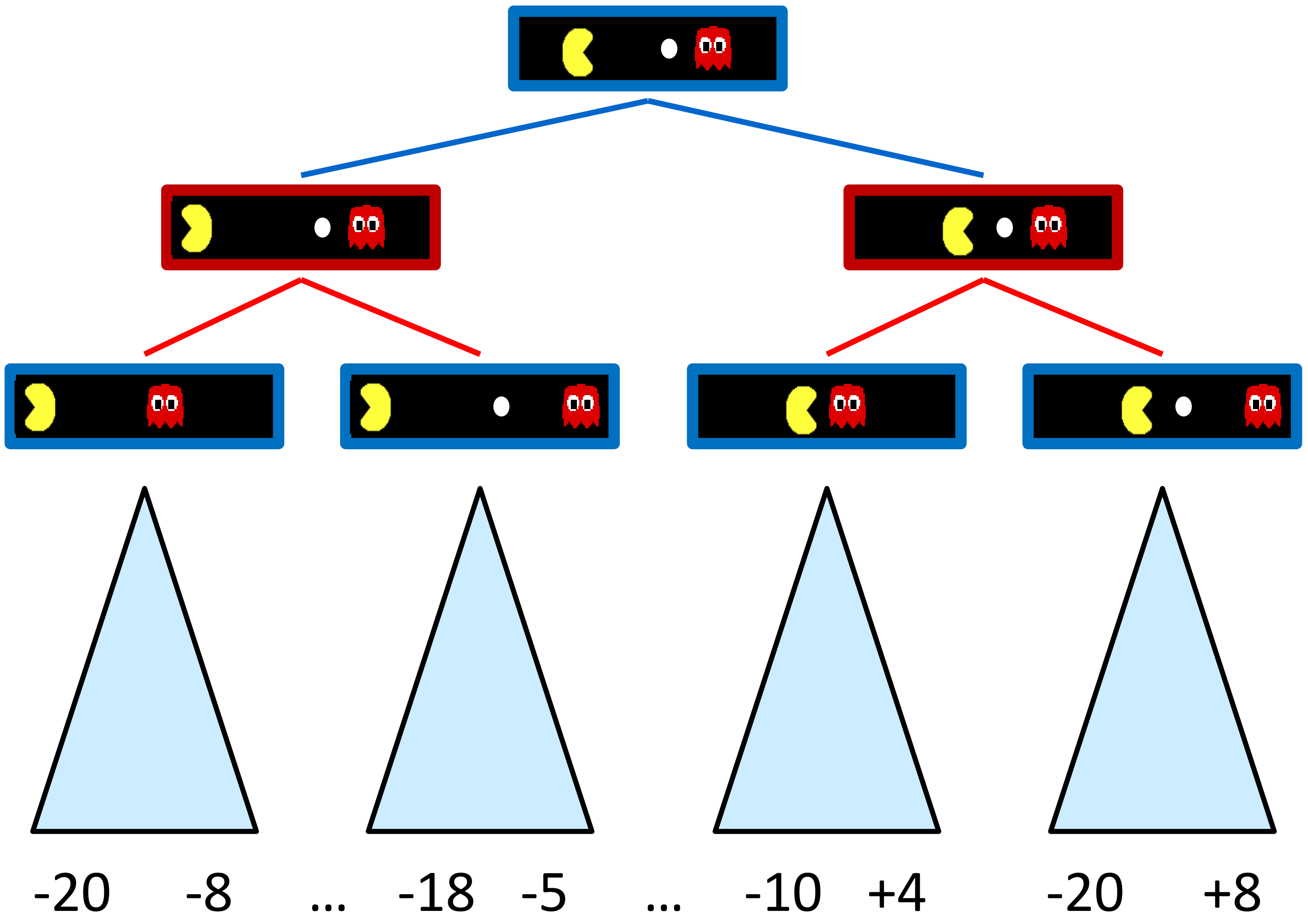
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:

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

Adversarial Game Trees



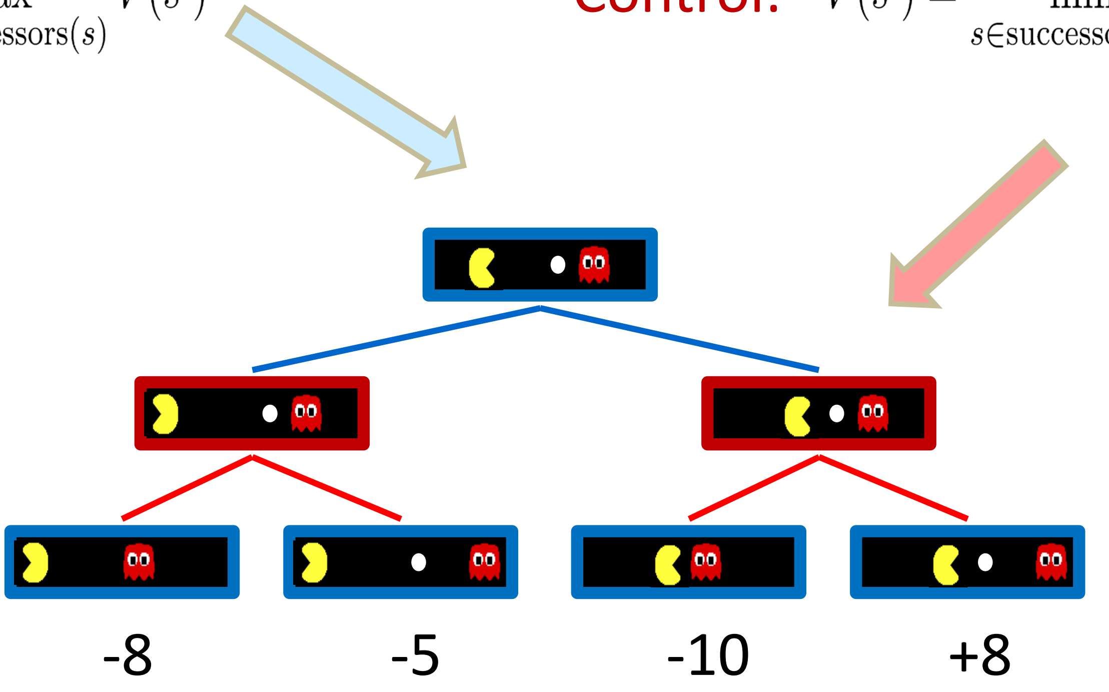
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

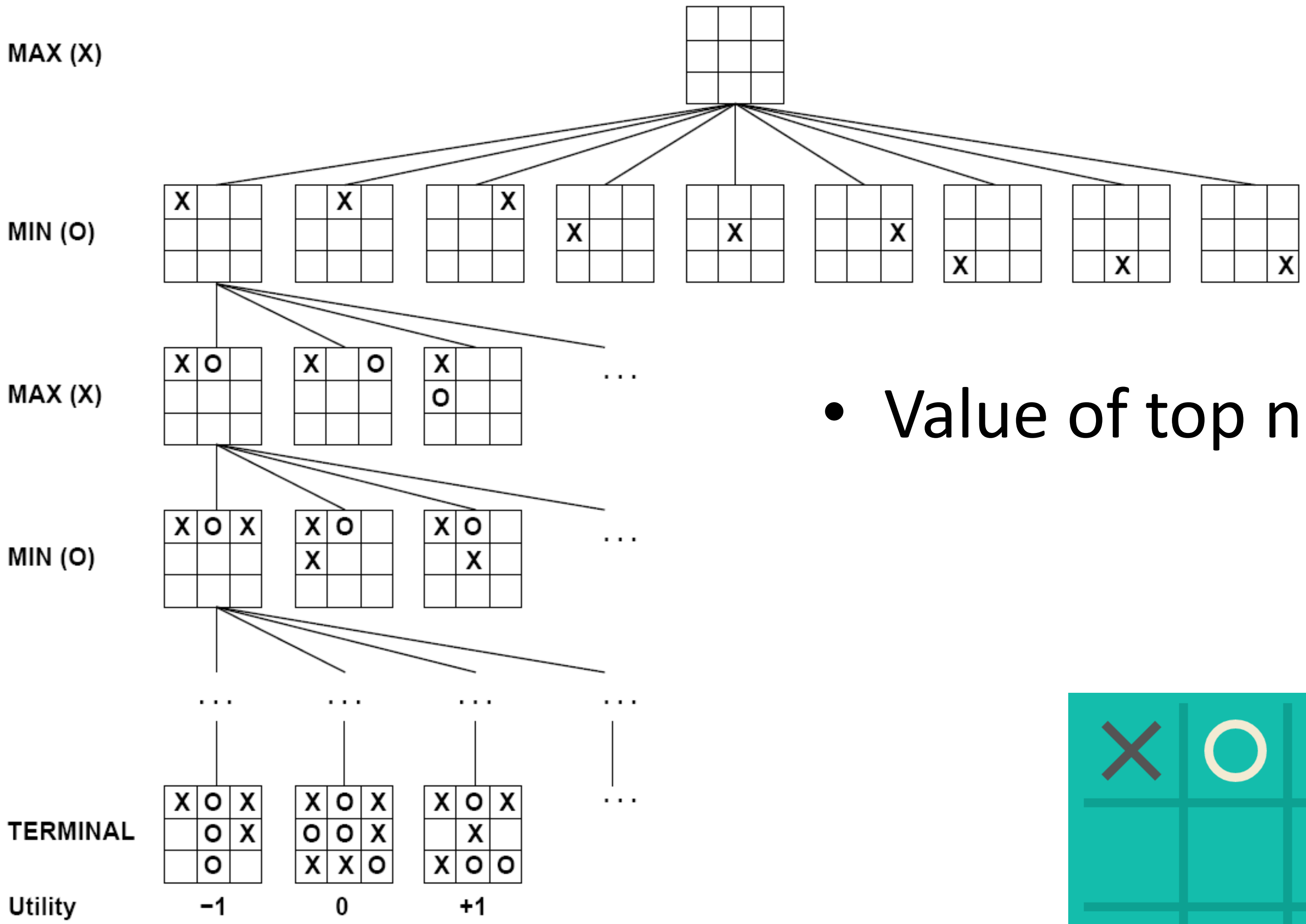
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



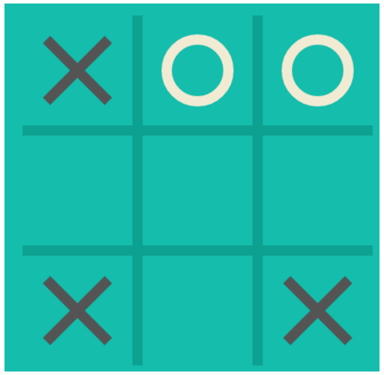
Terminal States:

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

Tic-Tac-Toe game tree



- Value of top node?



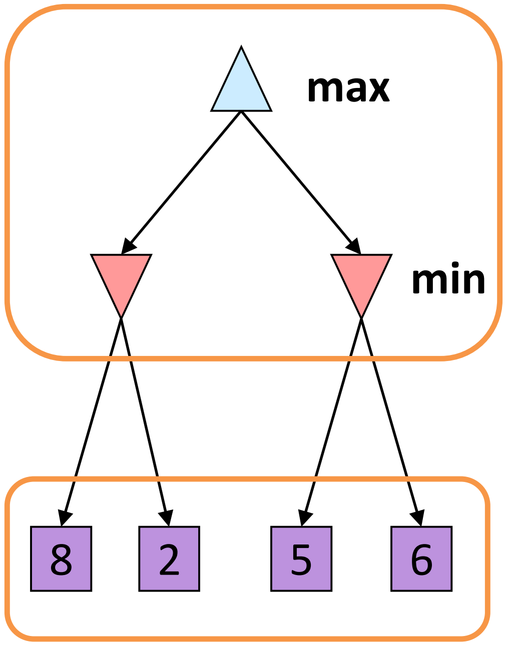
Adversarial Search (Minimax)

- 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

- Is the top max the max value of all terminal states?

Minimax values: computed recursively



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 (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)
```

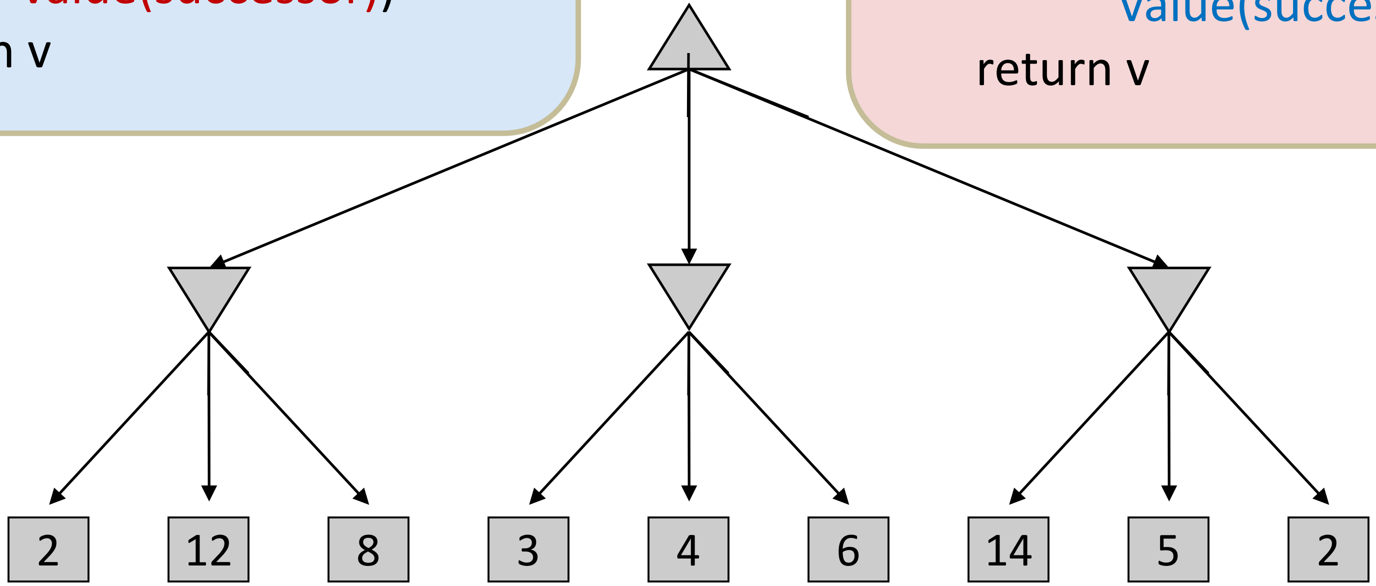
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

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

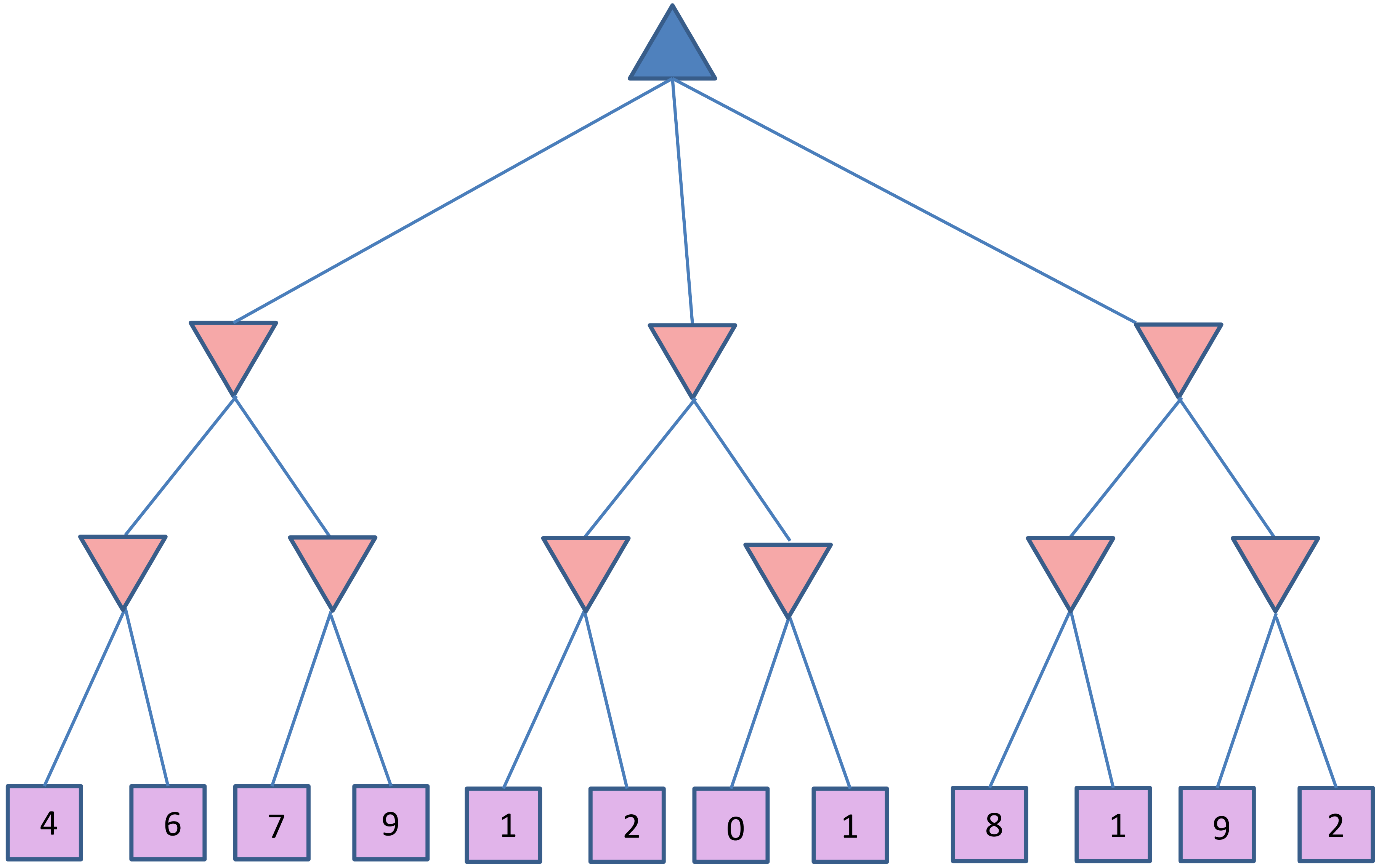
Minimax Example

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

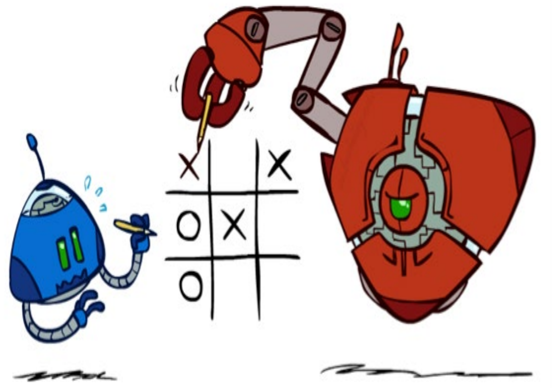
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

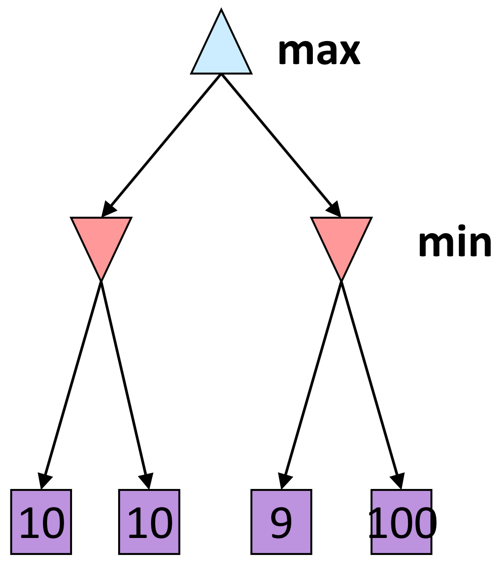


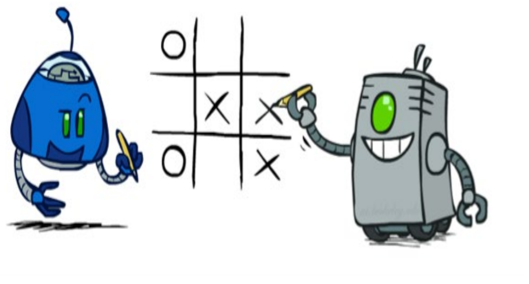
Minimax DIY example



Minimax Properties



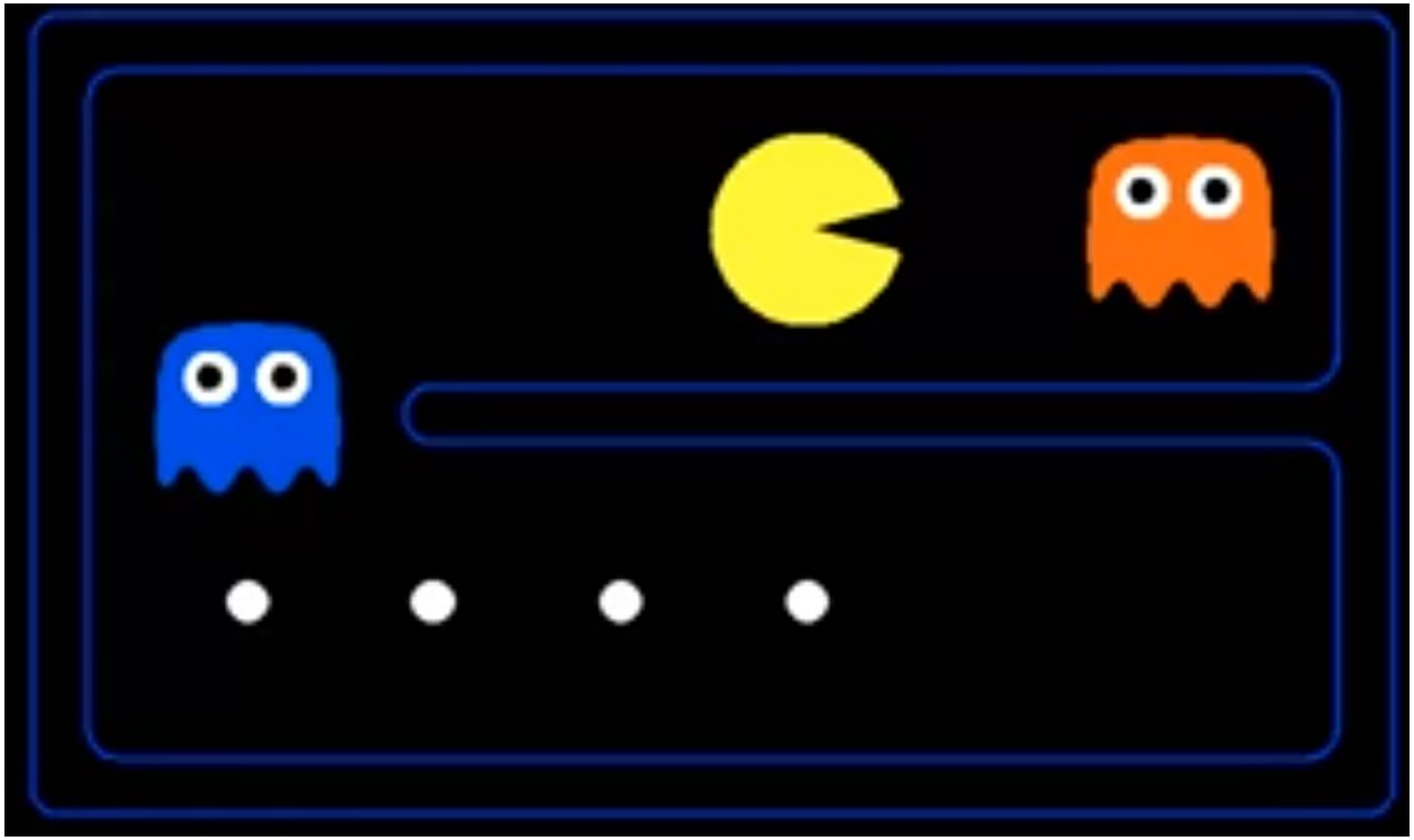




Optimal against a perfect player.
Otherwise?

Optimal play?

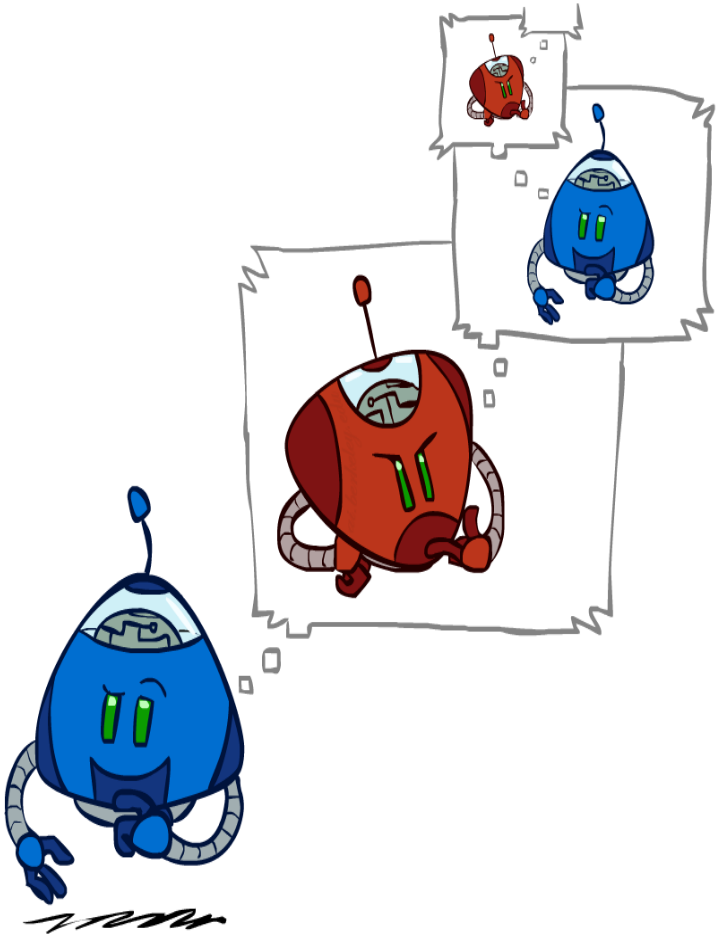
Rules:
Move -1
Eat pellet +10



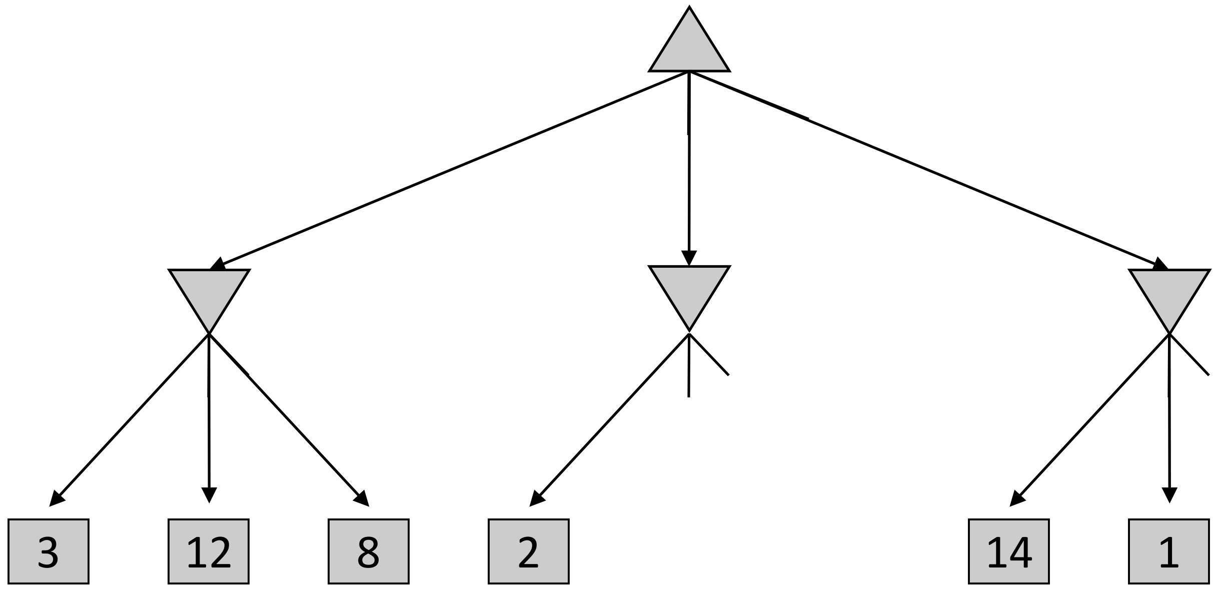
Minimax Efficiency

- 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?

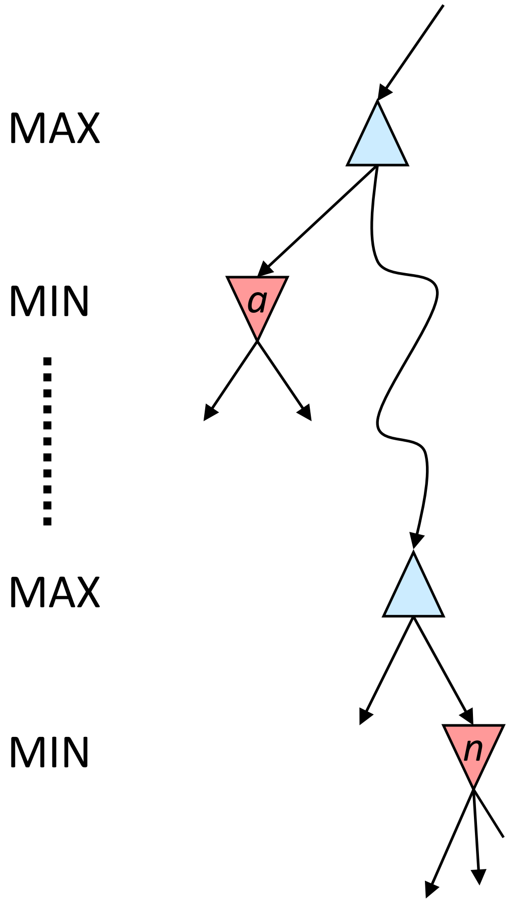


Minimax Pruning



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)
- 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
```

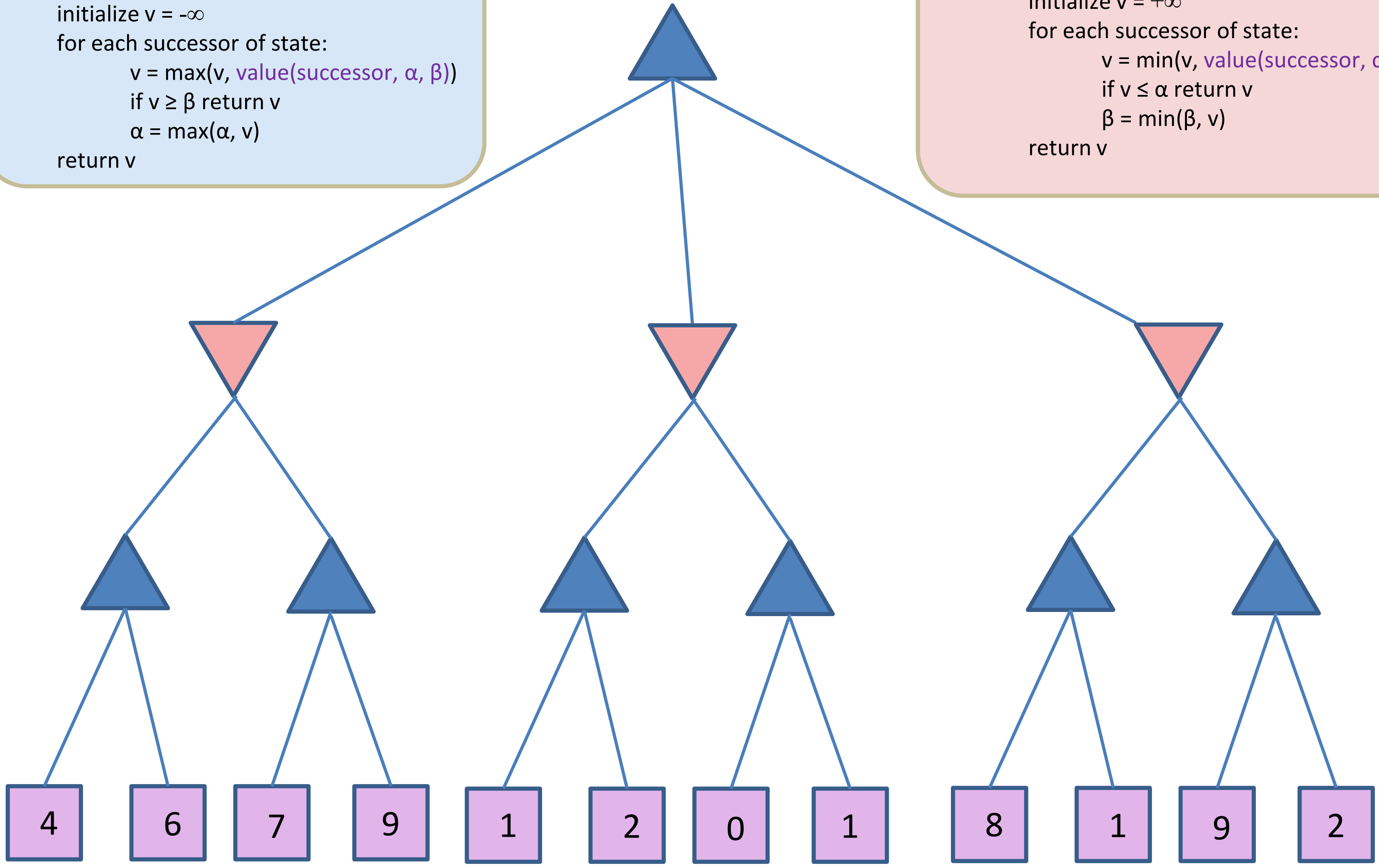
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor,
           α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-beta pruning example

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

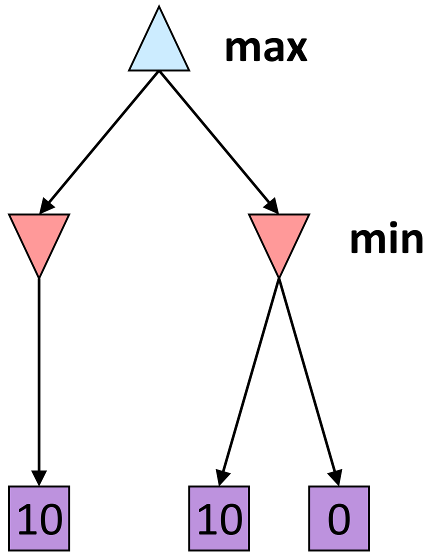
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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

- 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
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
- With random ordering
  - Time complexity is $O(b^{3m/4})$
- This is a simple example of metareasoning (computing about what to compute)



Summary

- Adversarial search
  - Mini-Max
  - Alpha-Beta pruning

- Next class: Imperfect decisions, Stochastic Games (5.4,5.5)

Lab 1

Lab 1