南方科技大学

STA303: Artificial Intelligence

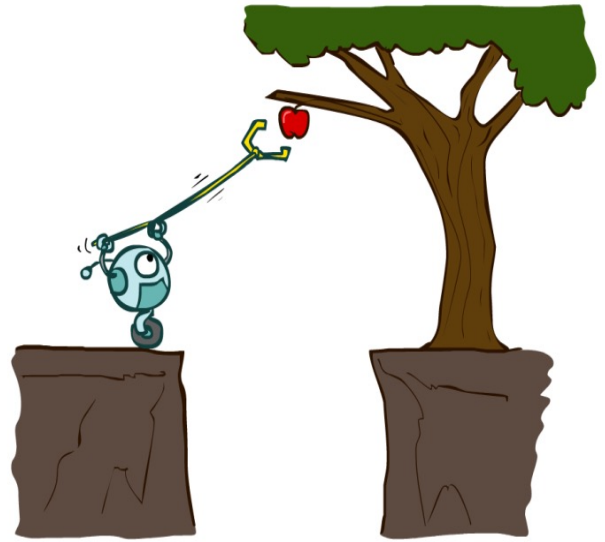
Final Review

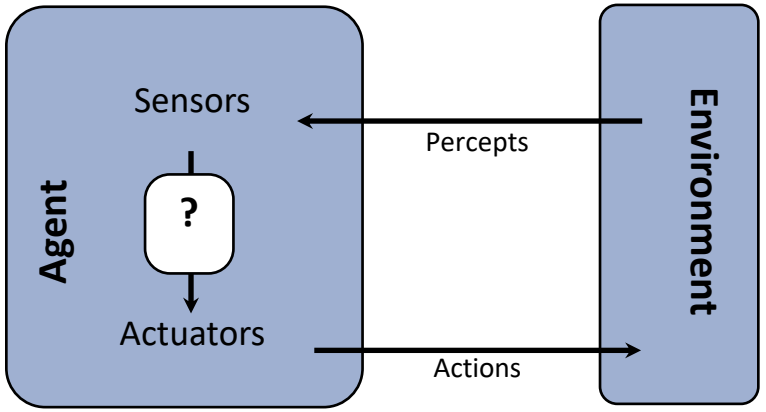
Fang Kong

https://fangkongx.github.io/Teaching/STA303/Fall2025/index.html

This course: Designing rational agents

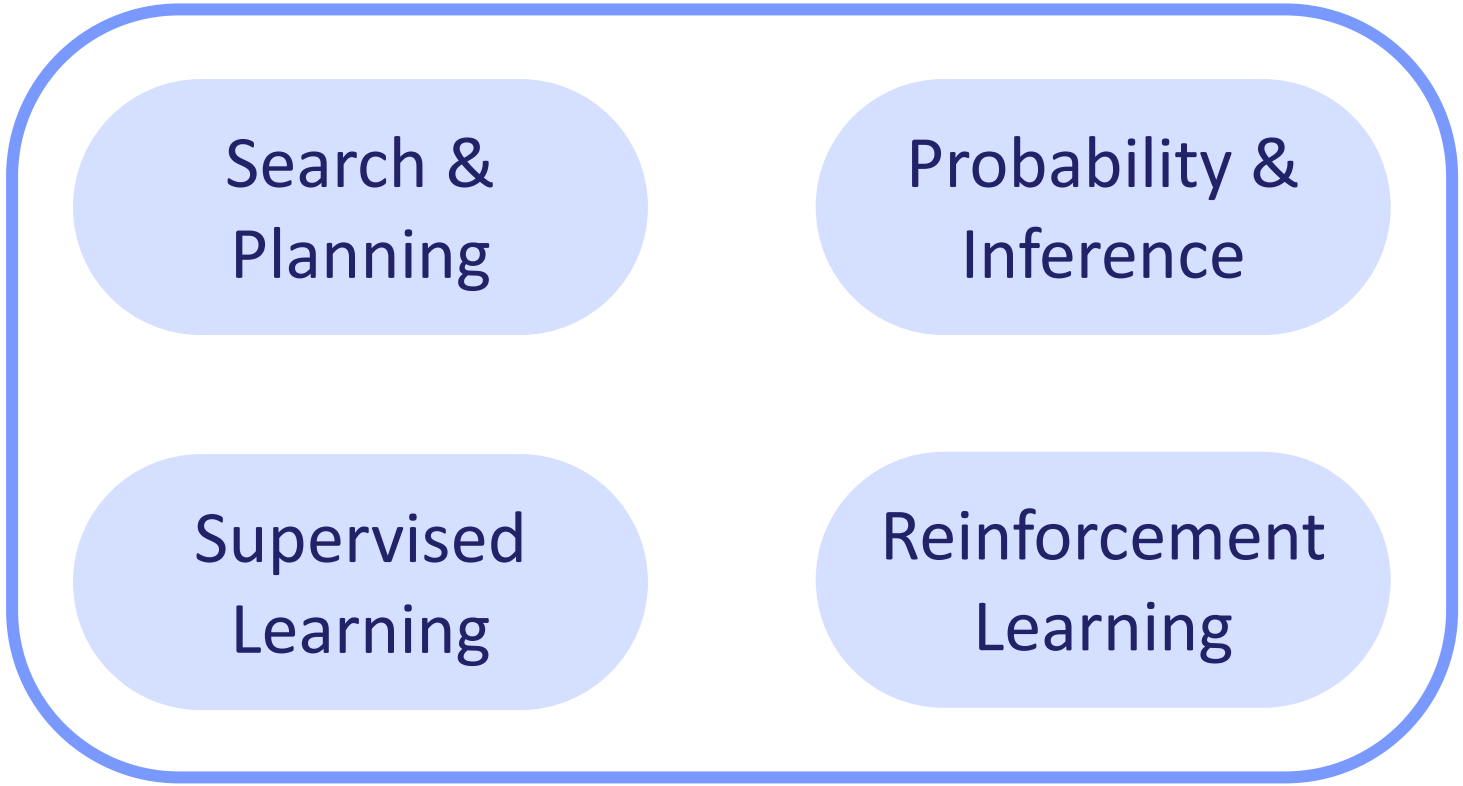
- An **agent** is an entity that perceives and acts.
- A **rational agent** selects actions that maximize its (expected) **utility**.
- Characteristics of the **percepts**, **environment**, and **action space** dictate techniques for selecting rational actions
- This course is about:
  - General AI techniques for a variety of problem types
  - Learning to recognize when and how a new problem can be solved with an existing technique





Course Topics

Core Components of Rational Agents:



南方科技大学

STA303: Artificial Intelligence

Search

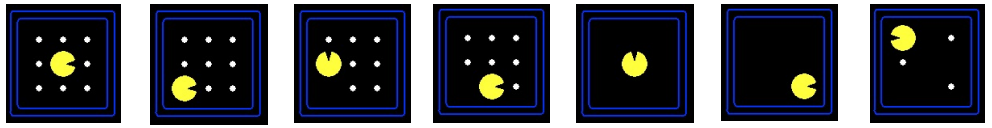
Fang Kong

https://fangkongx.github.io/Teaching/STA303/Fall2025/index.html

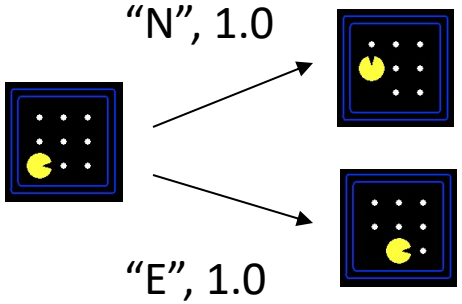
Slide credits: ai.berkeley.edu

Search Problems

- A search problem consists of:
  - A state space



  - A successor function (with actions, costs)

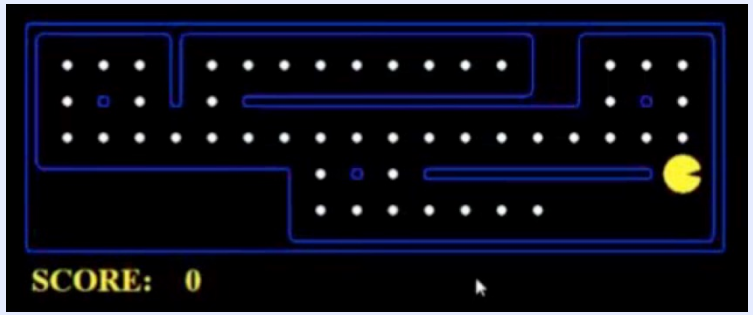


  - A start state and a goal test

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

What's in a State Space?

The world state includes every last detail of the environment



A search state keeps only the details needed for planning (abstraction)

- Problem: Pathing
  - States: (x,y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x,y)=END

- Problem: Eat-All-Dots
  - States: {(x,y), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

State Space Sizes?

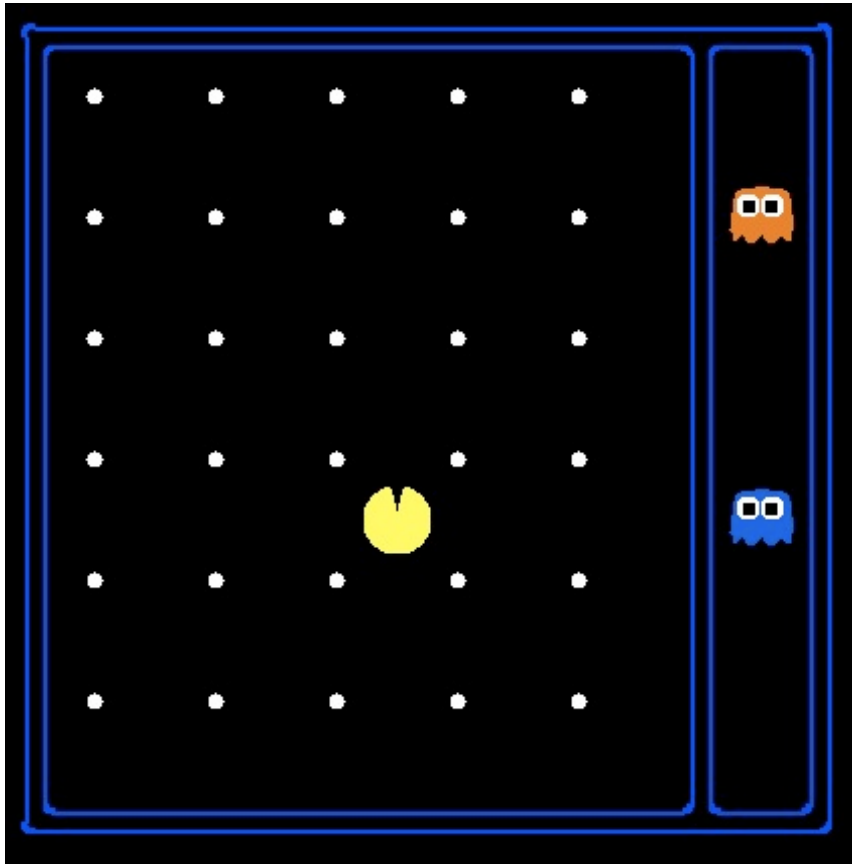
- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW

- How many
  - World states?

    $120x(2^{30})x(12^2)x4$
  - States for pathing?

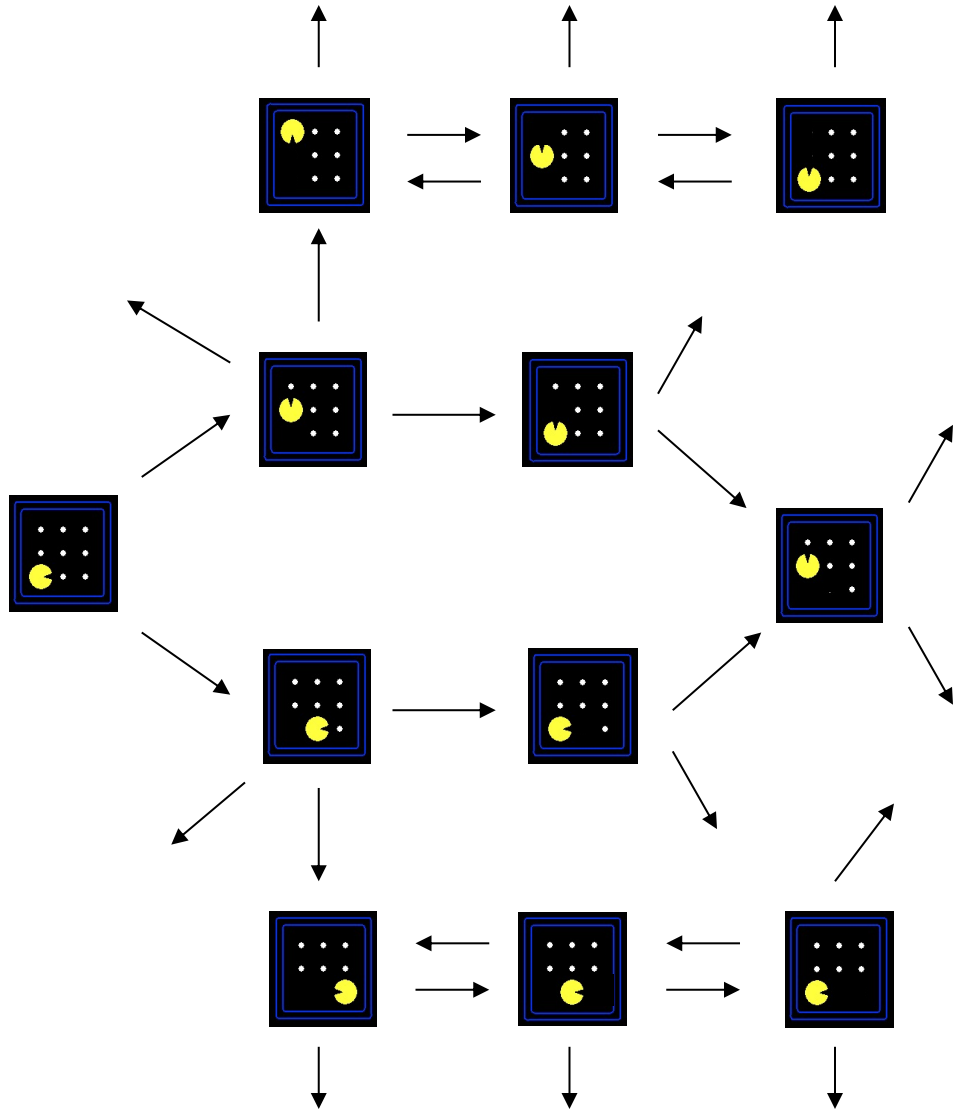
    120
  - States for eat-all-dots?

    $120x(2^{30})$

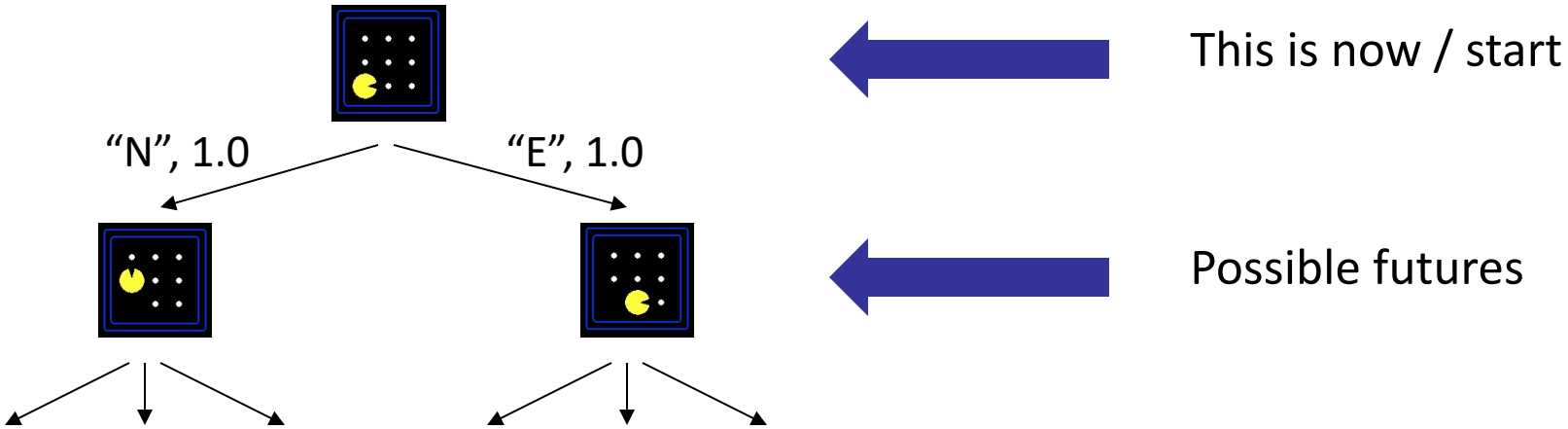


State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea



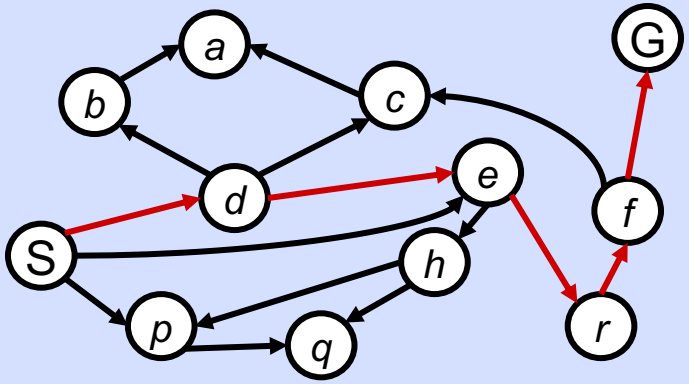
Search Trees



- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

State Space Graphs vs. Search Trees

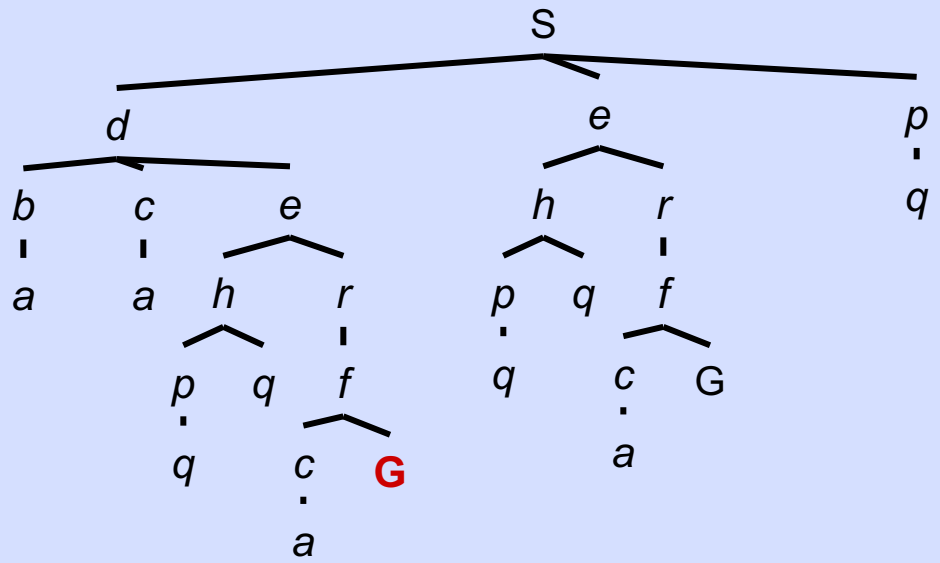
State Space Graph



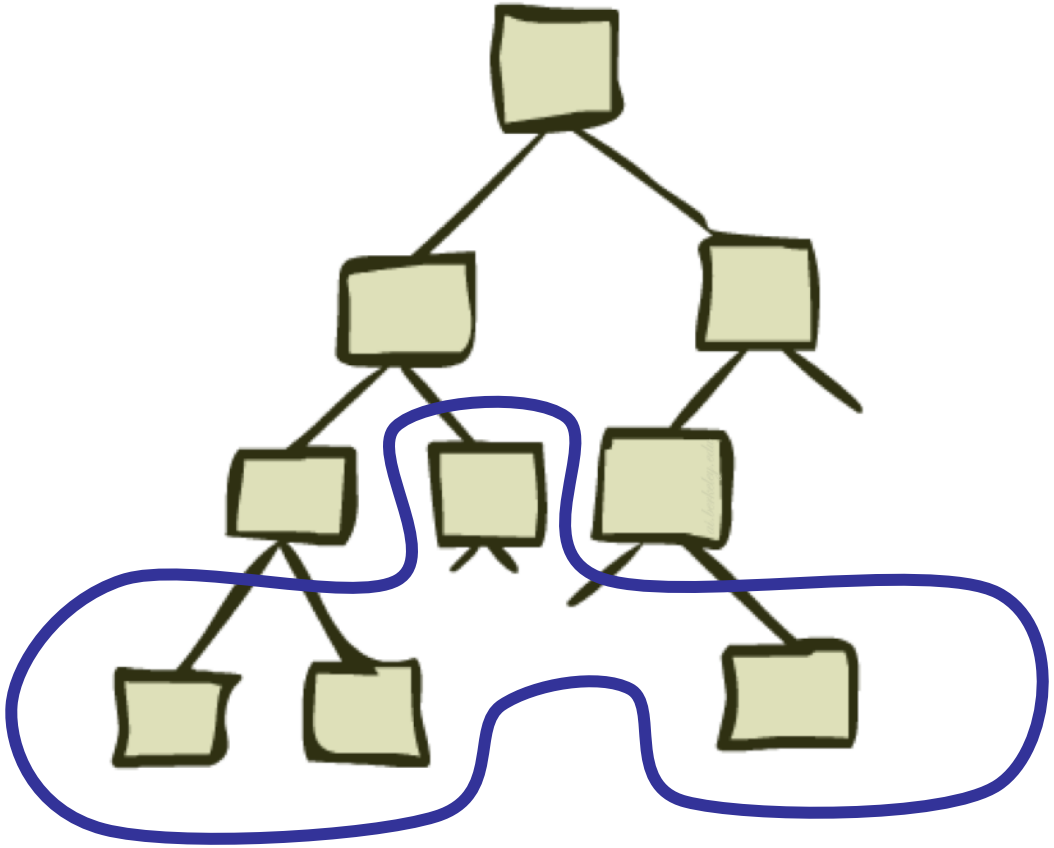
Each NODE in the search tree is an entire PATH in the state space graph.

We construct both on demand – and we construct as little as possible.

Search Tree



Tree Search



General Tree Search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

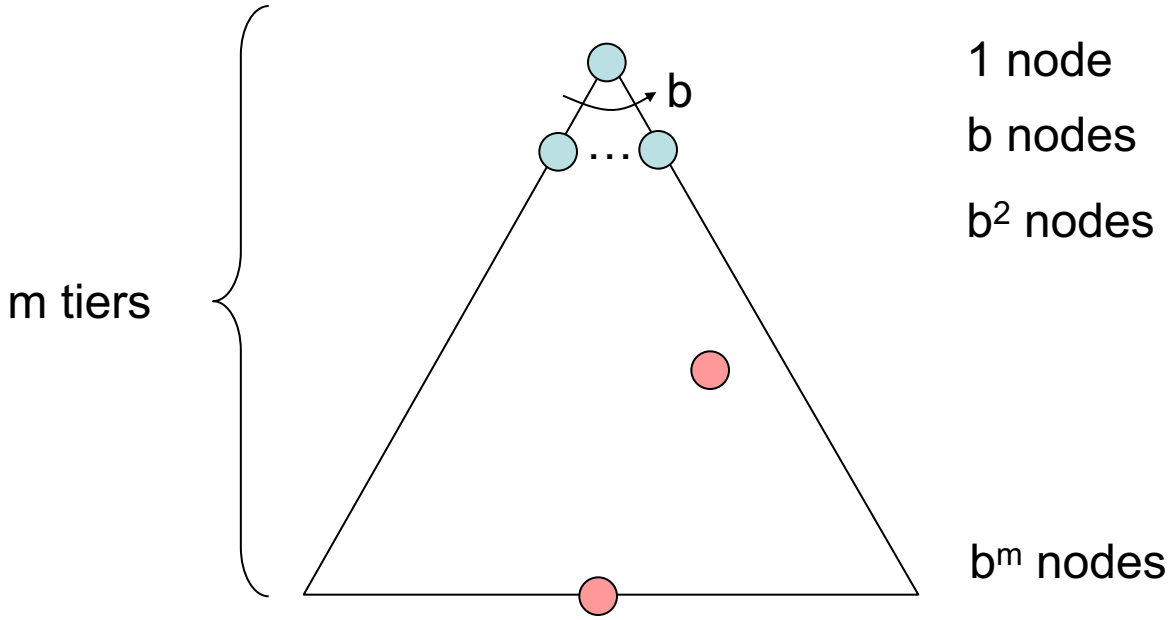
- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy
- Main question: which fringe nodes to explore?

Search Algorithm Properties

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths

- Number of nodes in entire tree?
  - $1 + b + b^2 + \ldots. b^m = O(b^m)$

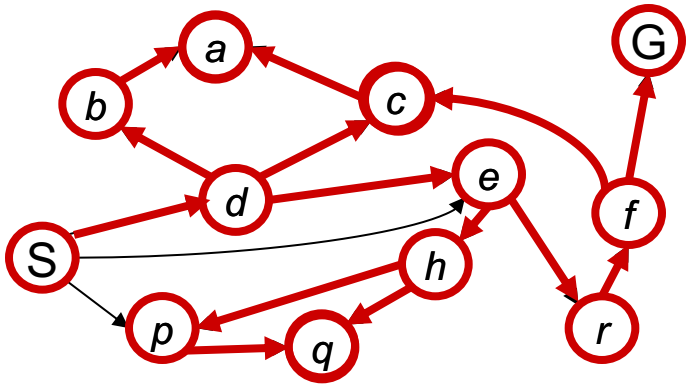


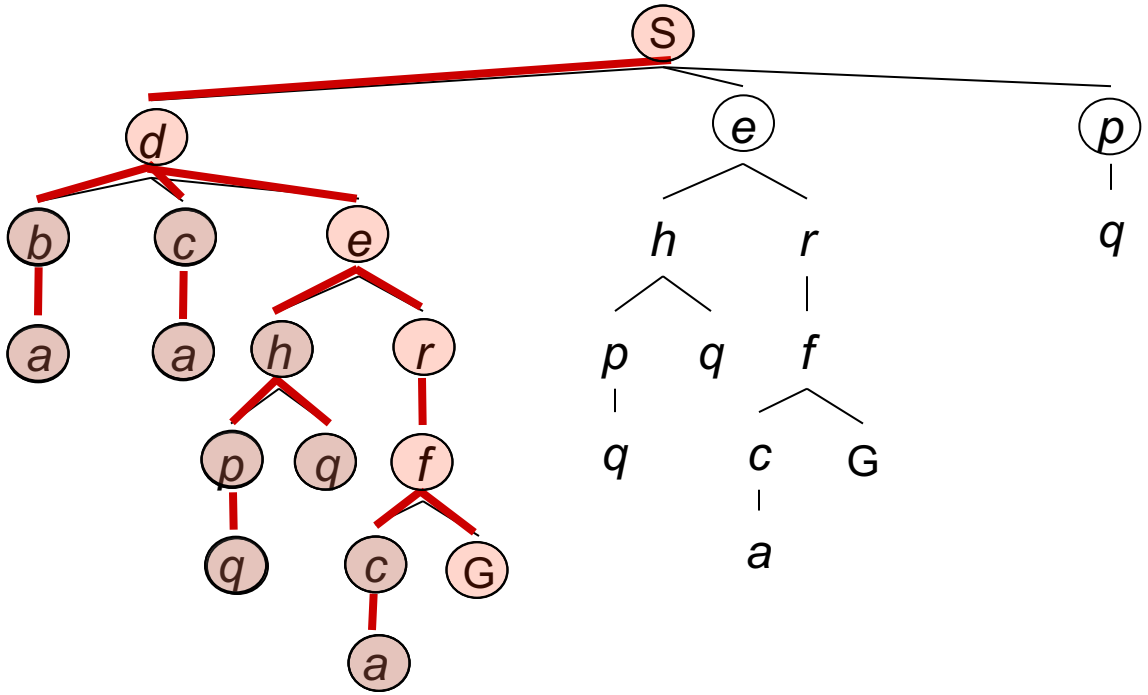
Depth-First Search

Depth-First Search

Strategy: expand a deepest node first

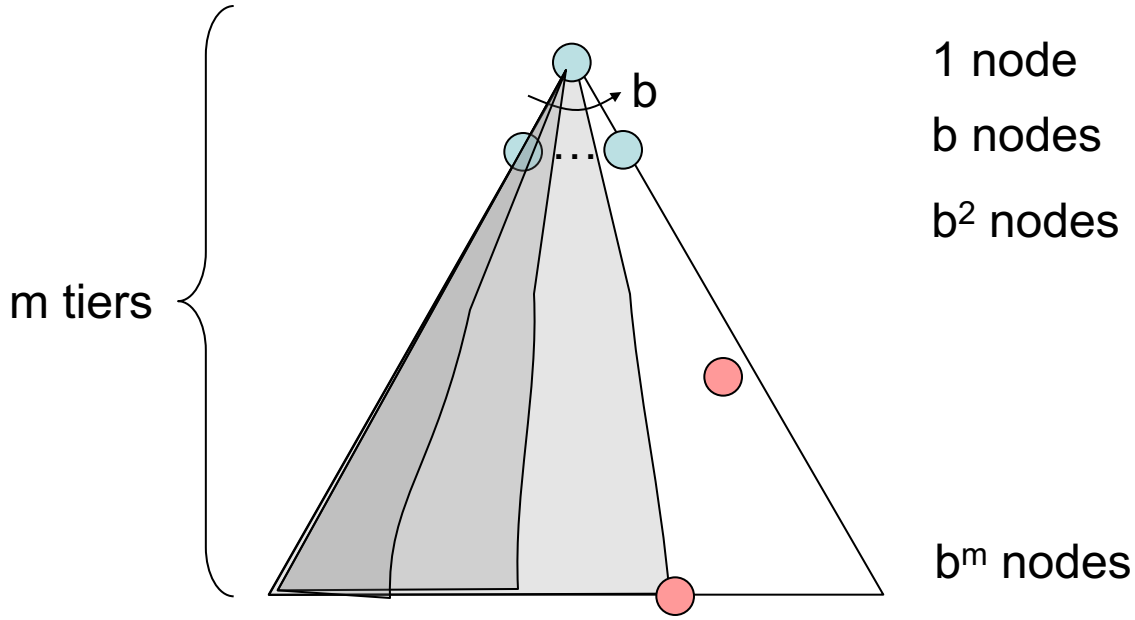
Implementation: Fringe is a LIFO stack



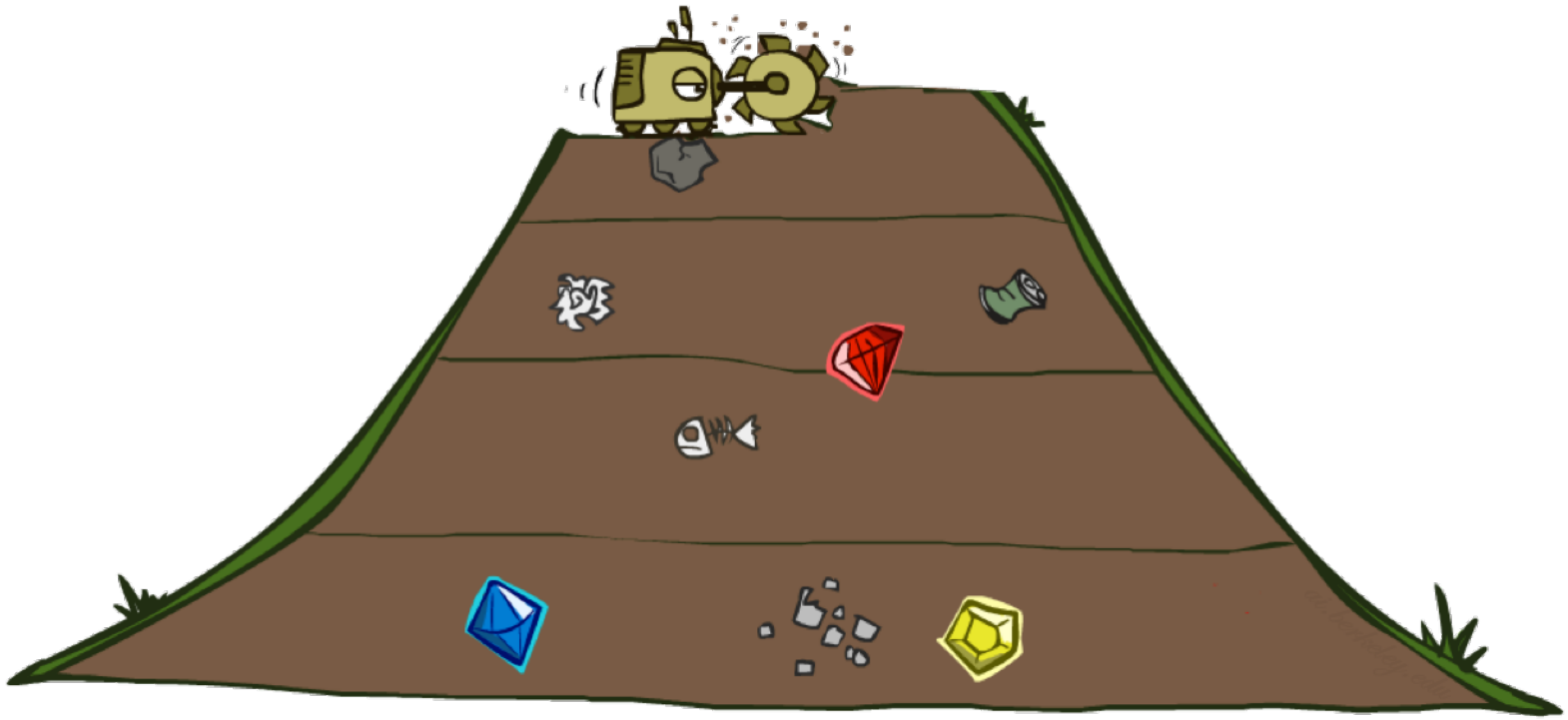


Depth-First Search (DFS) Properties

- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, takes time $O(b^m)$
- How much space does the fringe take?
  - Only has siblings on path to root, so O(bm)
- Is it complete?
  - m could be infinite, so only if we prevent that
- Is it optimal?
  - No, it finds the “leftmost” solution, regardless of depth or cost



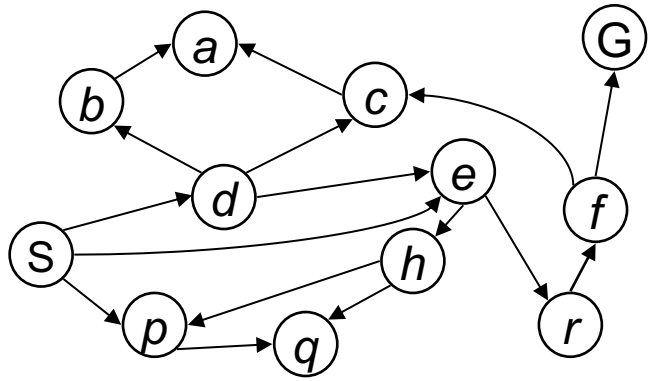
Breadth-First Search

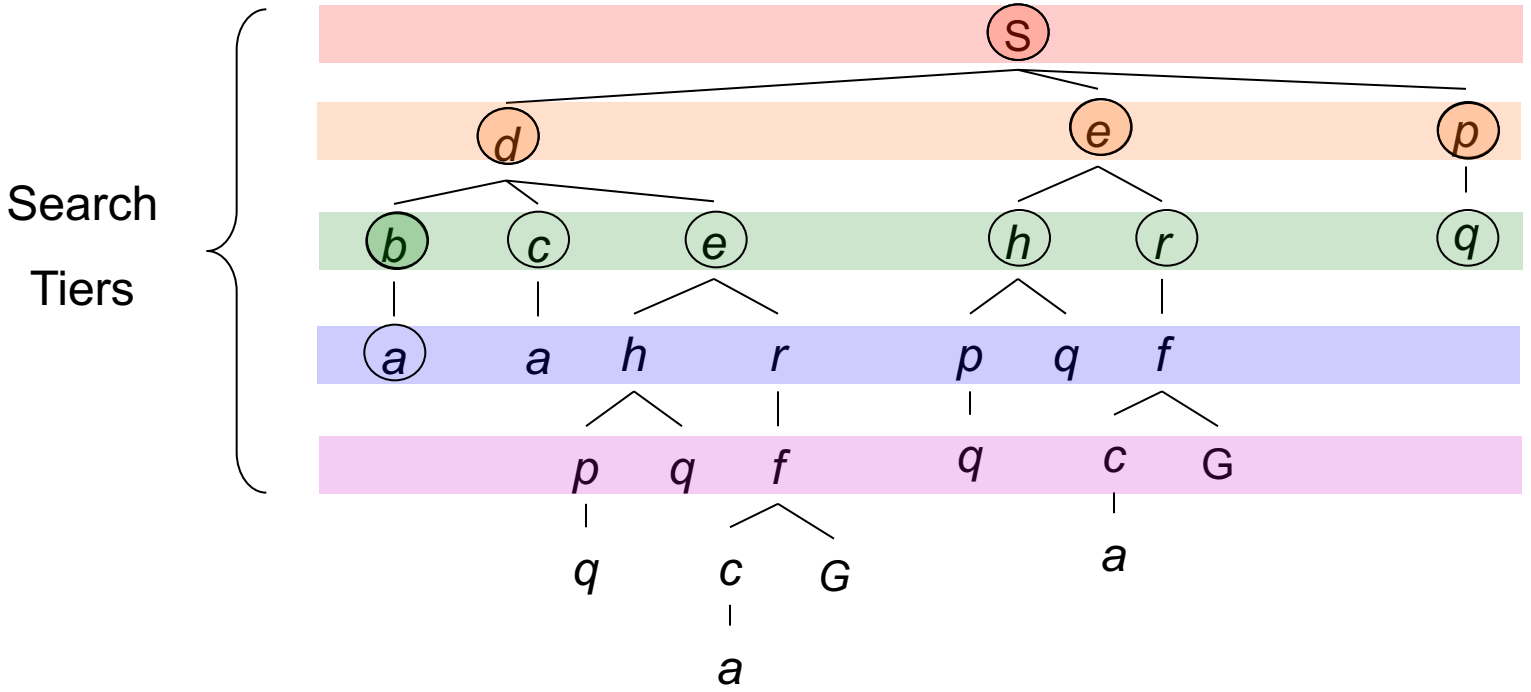


Breadth-First Search

Strategy: expand a shallowest node first

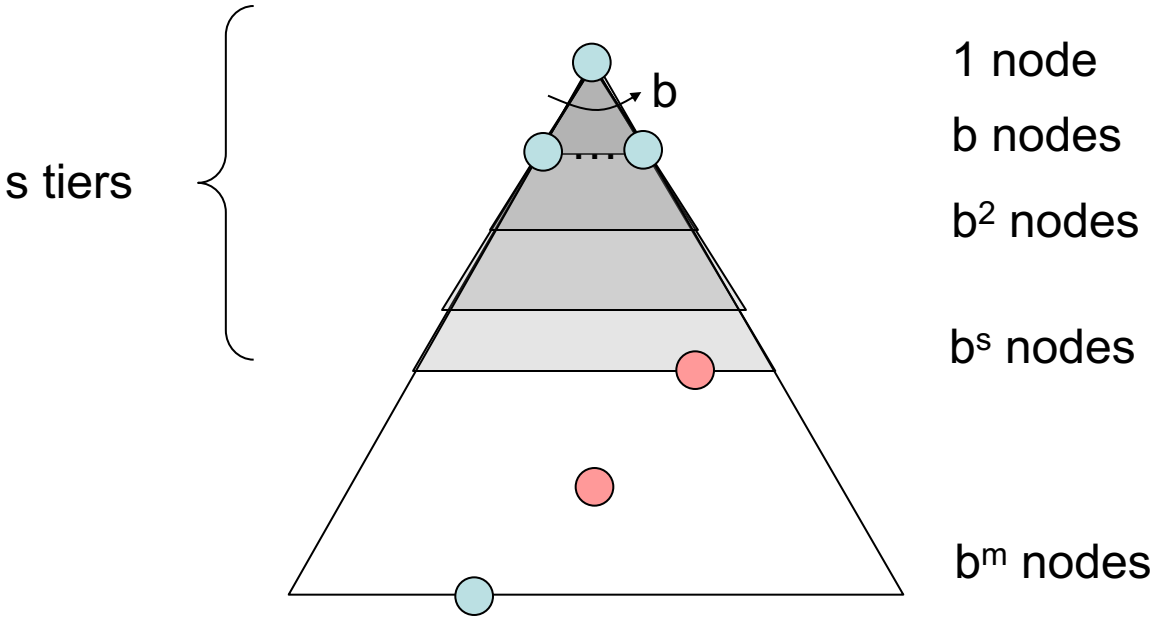
Implementation: Fringe is a FIFO queue





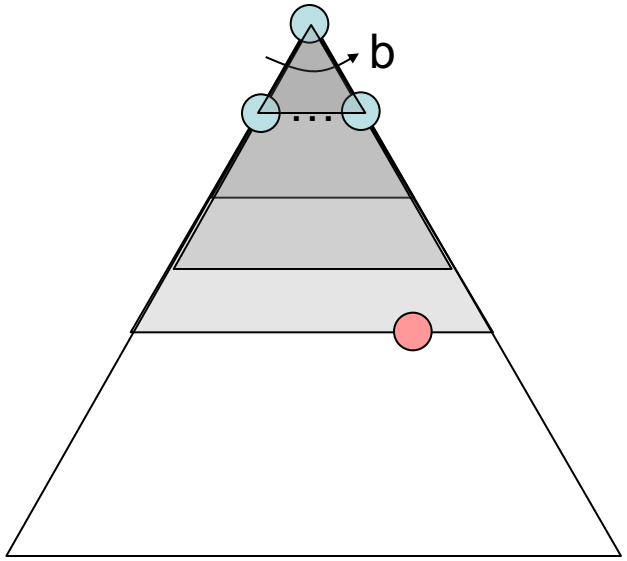
Breadth-First Search (BFS) Properties

- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^s)$
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$
- Is it complete?
  - s must be finite if a solution exists, so yes!
- Is it optimal?
  - Only if costs are all 1 (more on costs later)

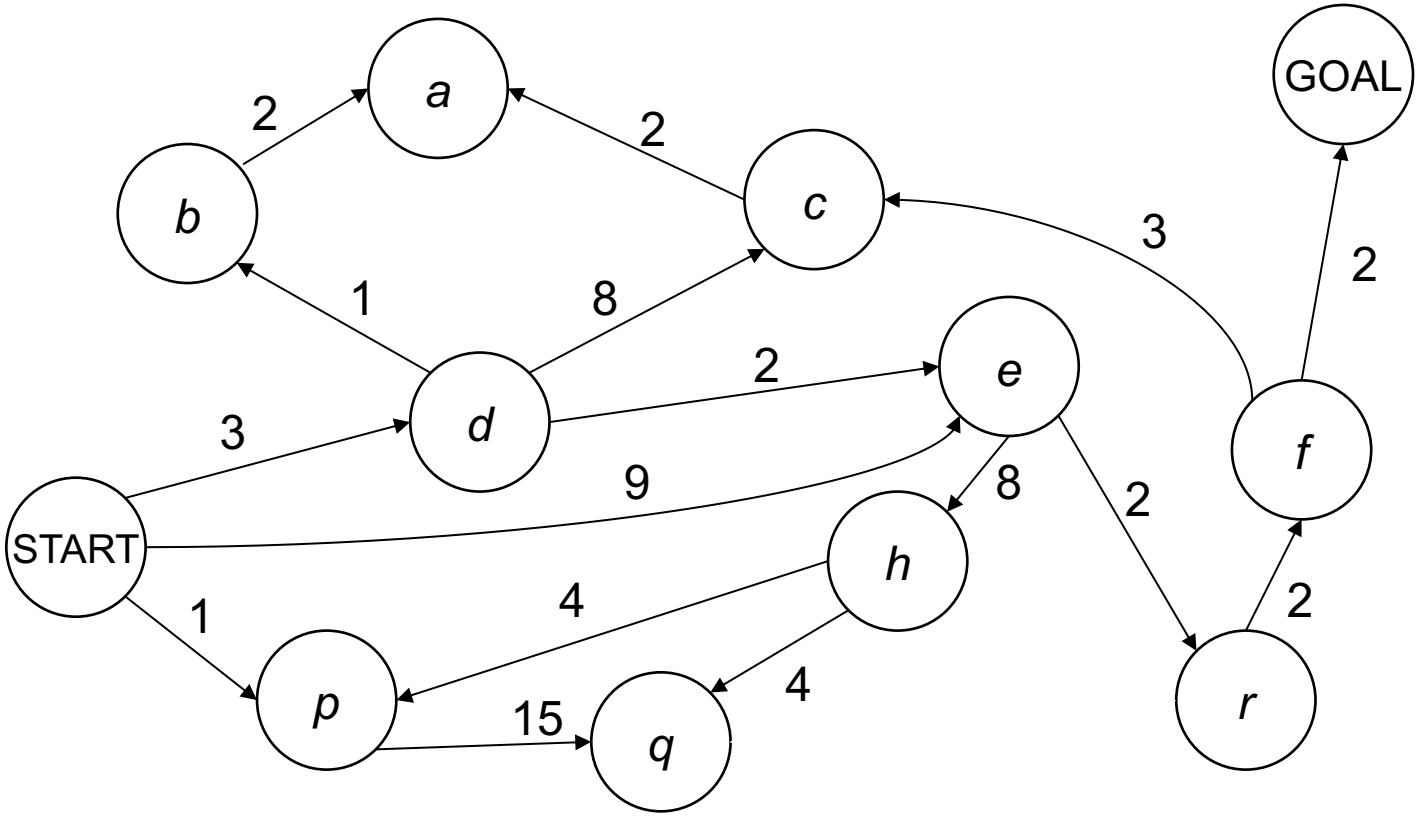


Iterative Deepening

- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution…
  - Run a DFS with depth limit 2. If no solution…
  - Run a DFS with depth limit 3. …..
- Isn't that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!

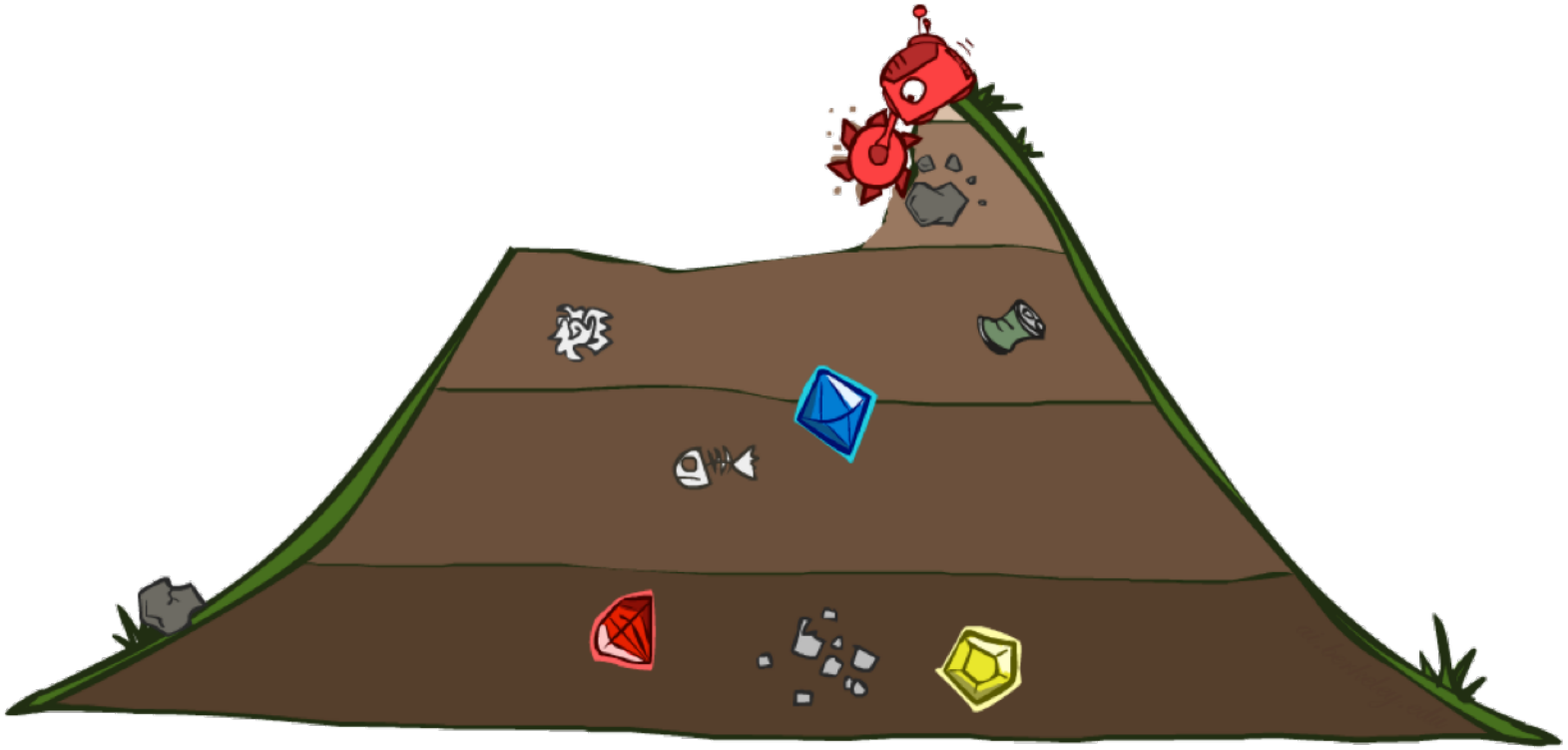


Cost-Sensitive Search



BFS finds the shortest path in terms of number of actions.
It does not find the least-cost path. We will now cover
a similar algorithm which does find the least-cost path.

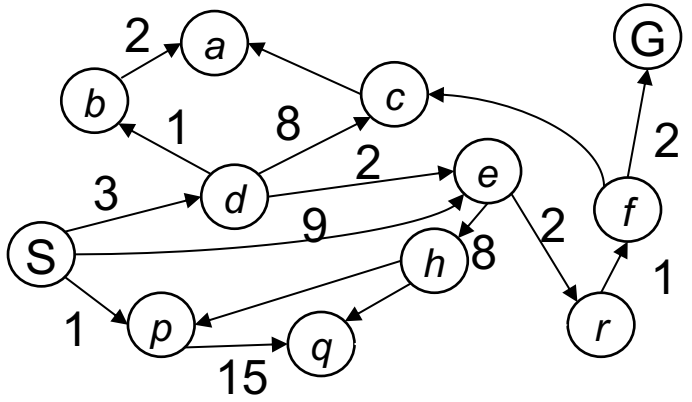
Uniform Cost Search



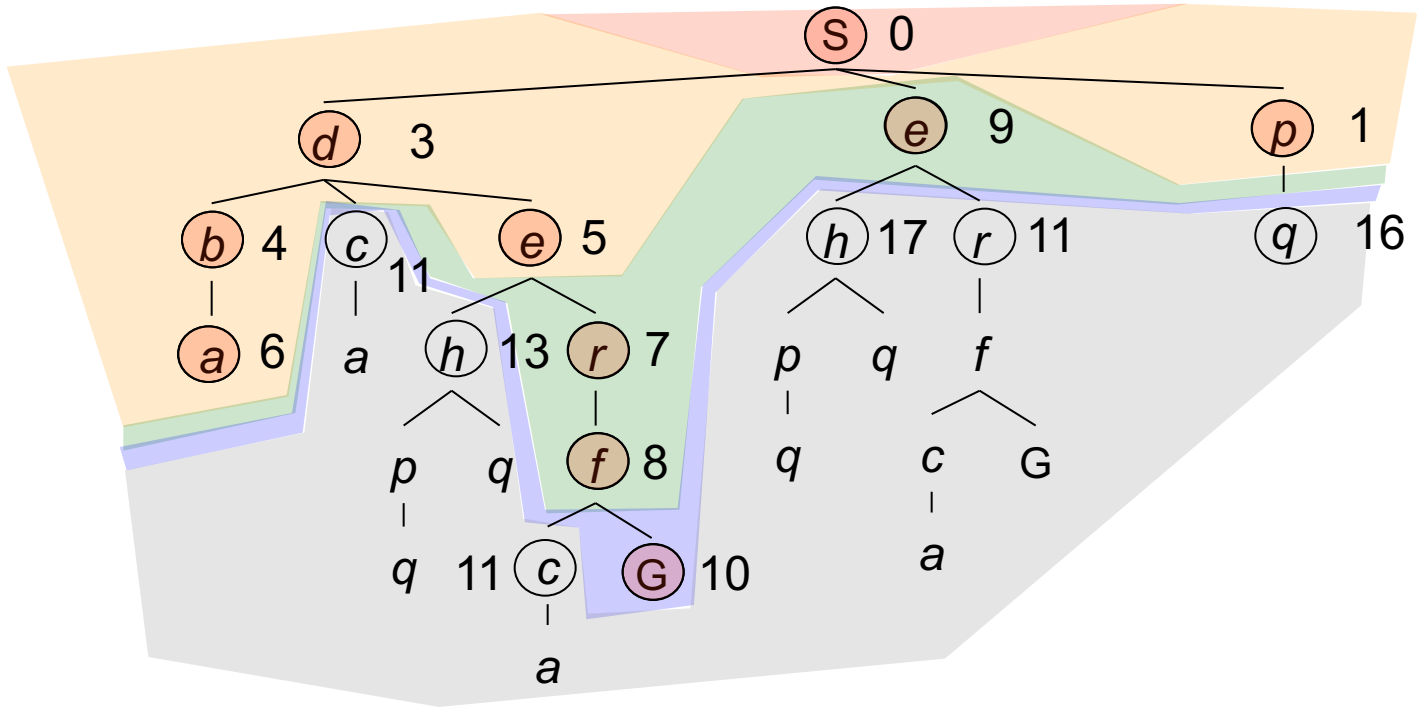
Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)

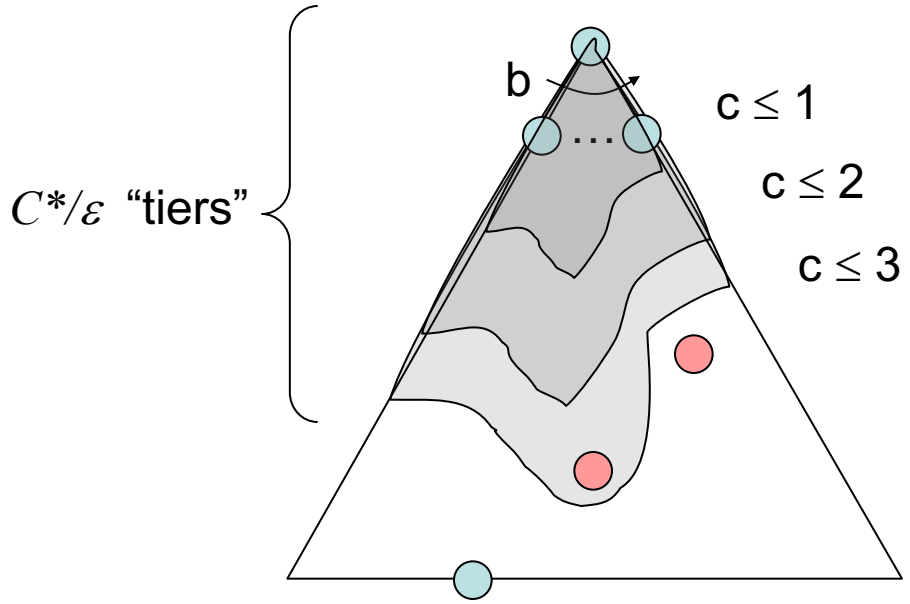


Cost contours



Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)



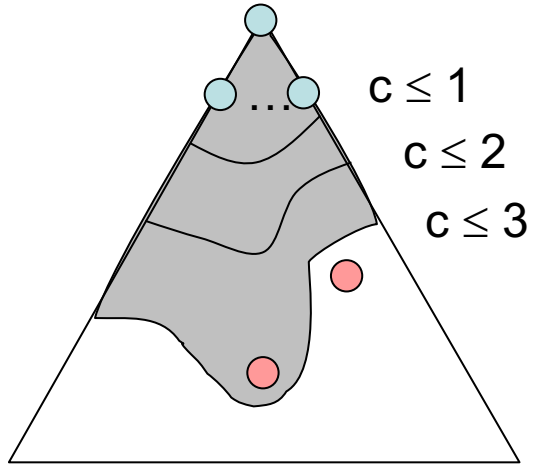
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$

- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!

- Is it optimal?
  - Yes! (Proof next lecture via A*)

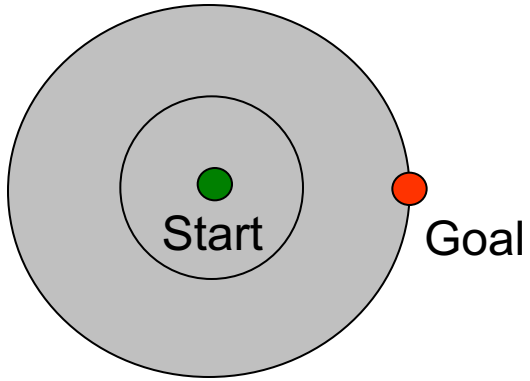
Uniform Cost Issues

- Remember: UCS explores increasing cost contours



- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every “direction”
  - No information about goal location

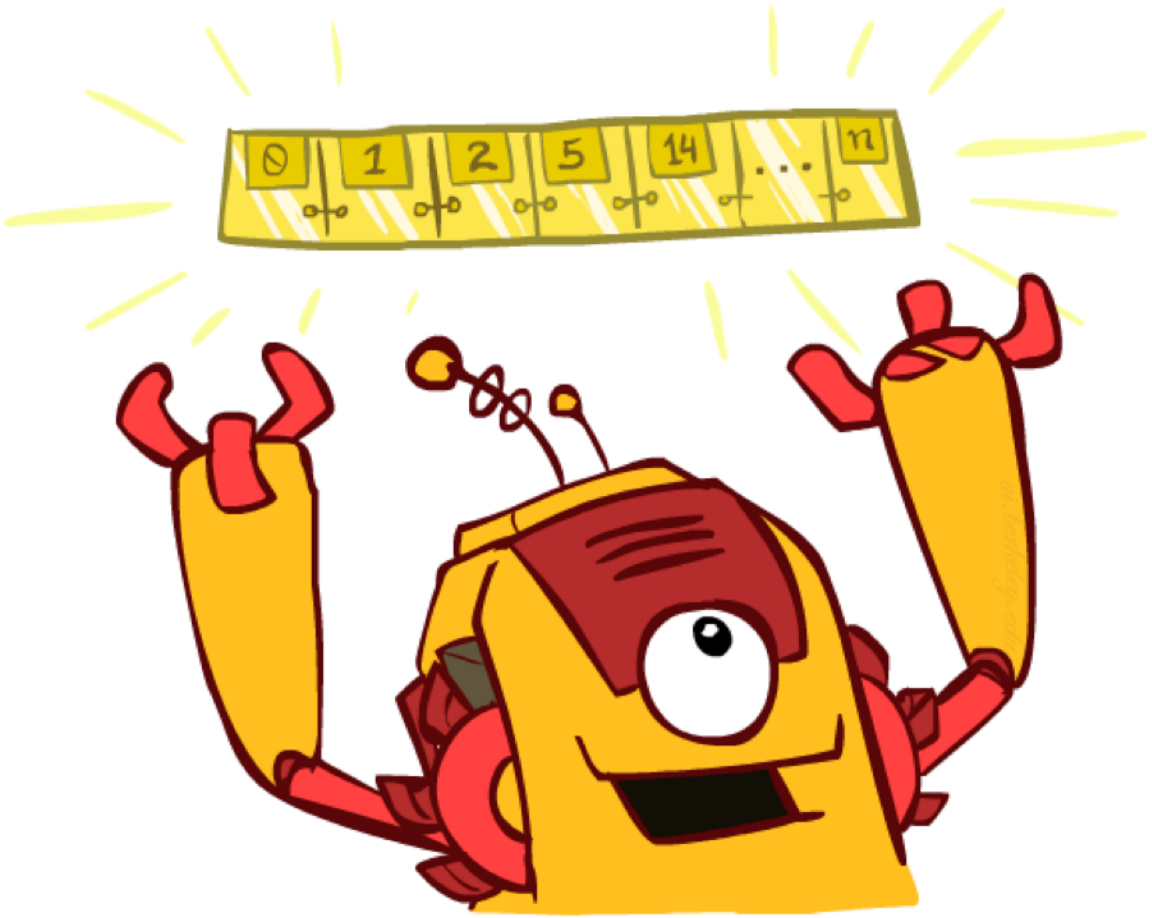


- We’ll fix that soon!

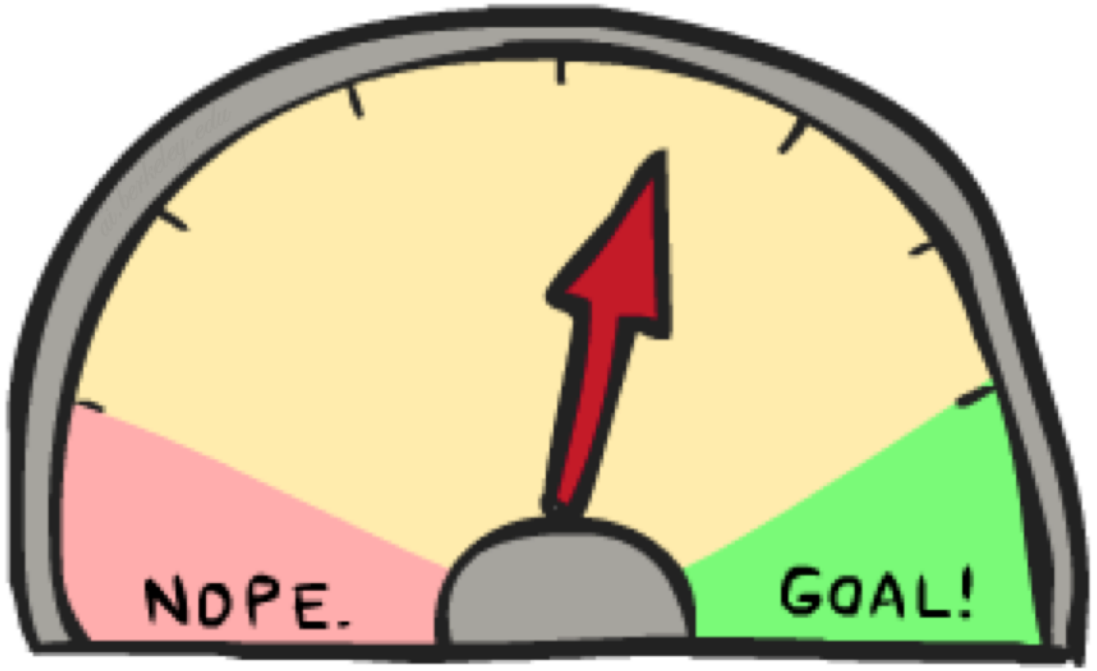
[Demo: empty grid UCS (L2D5)]
[Demo: maze with deep/shallow water DFS/BFS/UCS (L2D7)]

The One Queue

- All these search algorithms are the same except for fringe strategies
  - Conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities)
  - Practically, for DFS and BFS, you can avoid using an actual priority queue, by using stacks and queues
  - Can even code one implementation that takes a variable queuing object

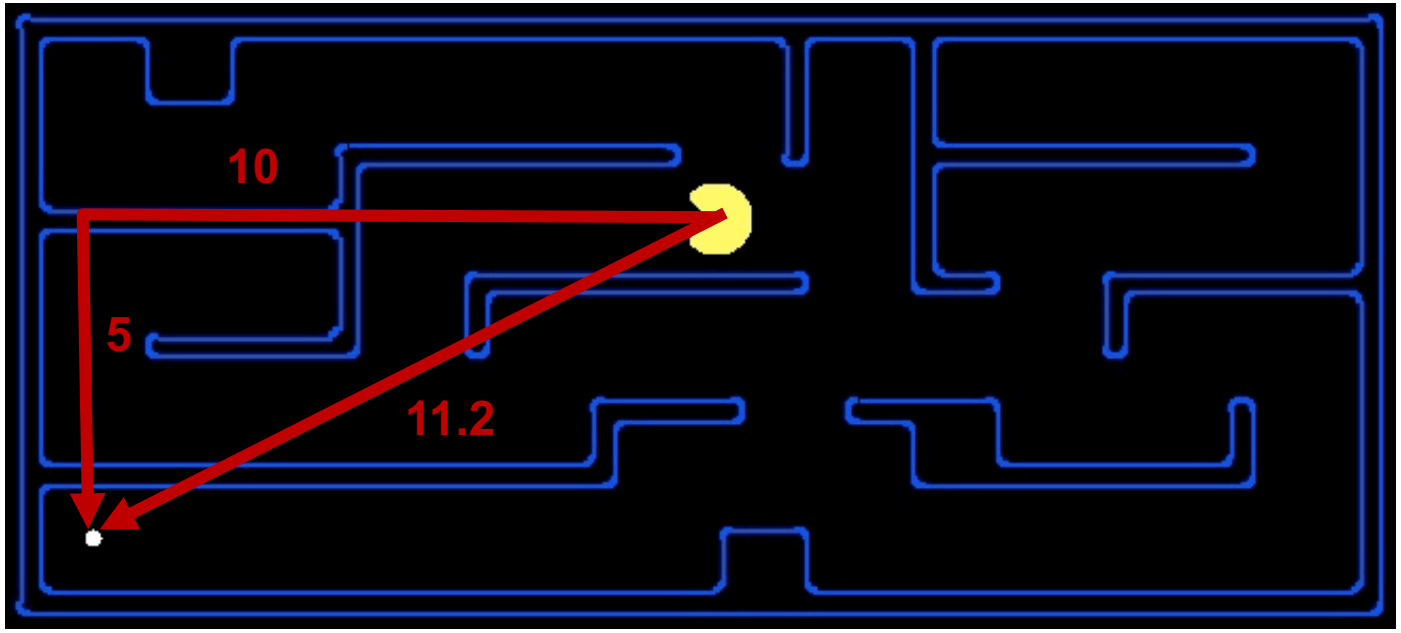


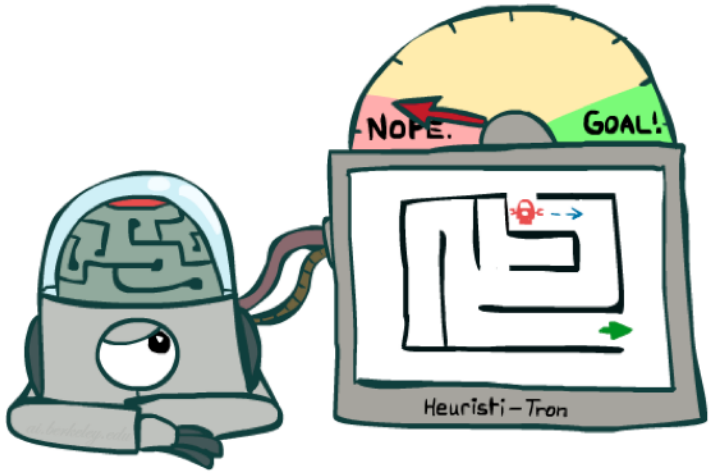
Informed Search

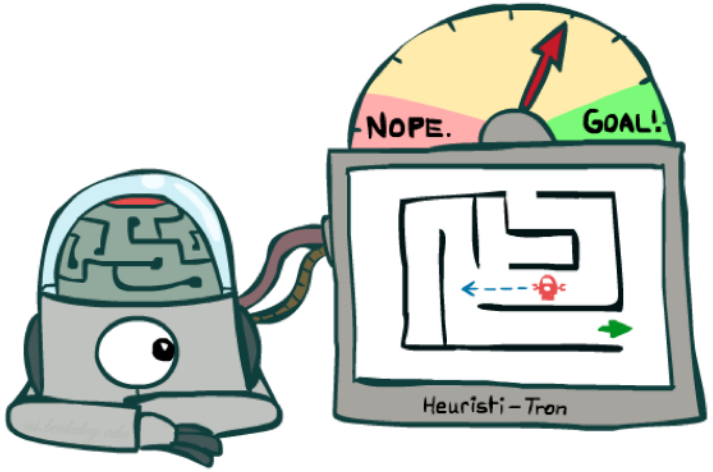


Search Heuristics

- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing

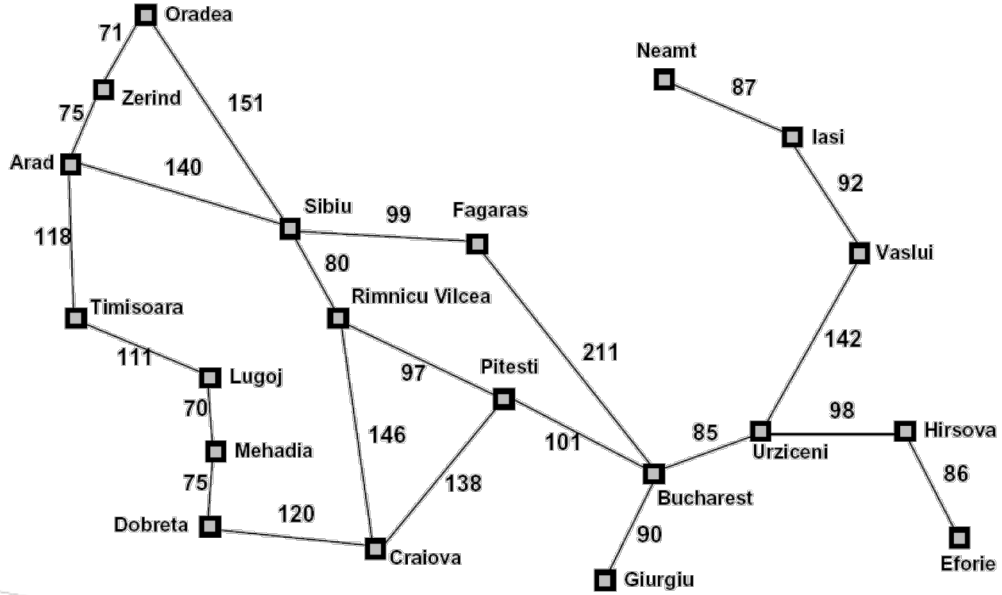




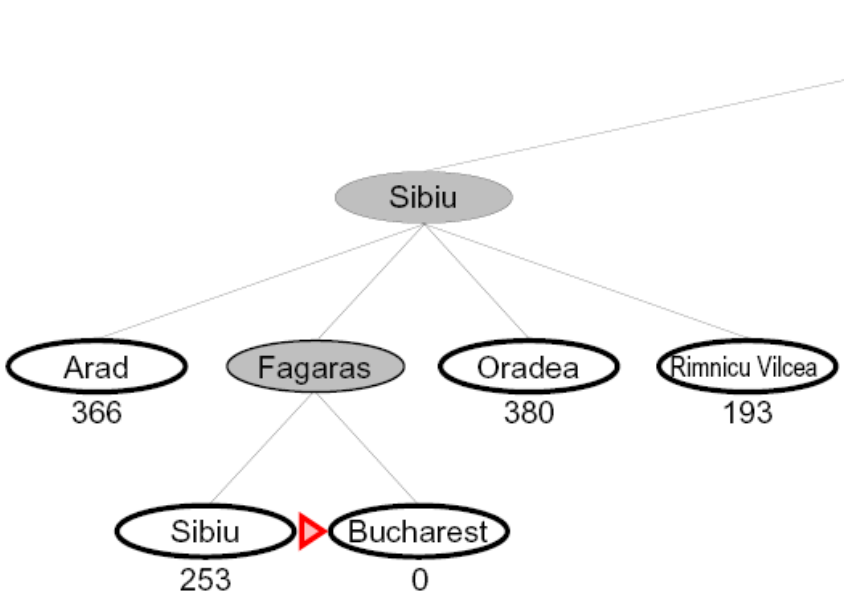


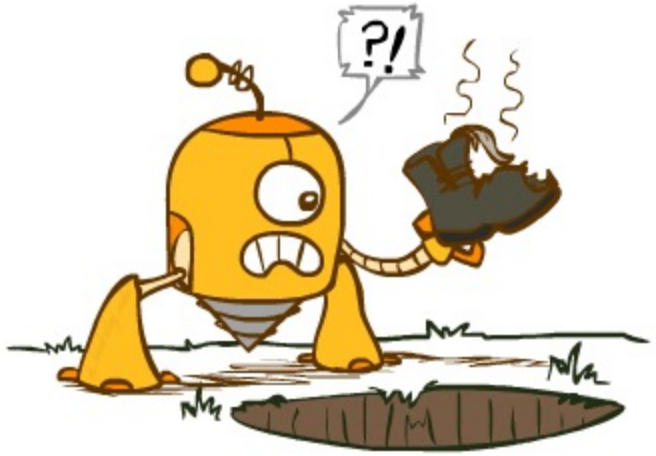
Greedy Search

Greedy Search



- Expand the node that seems closest…

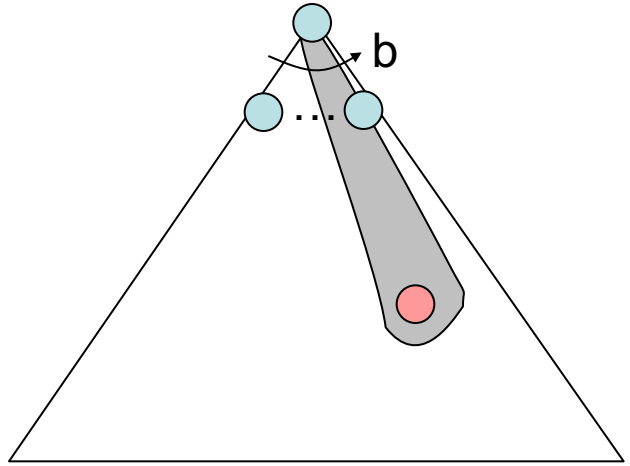




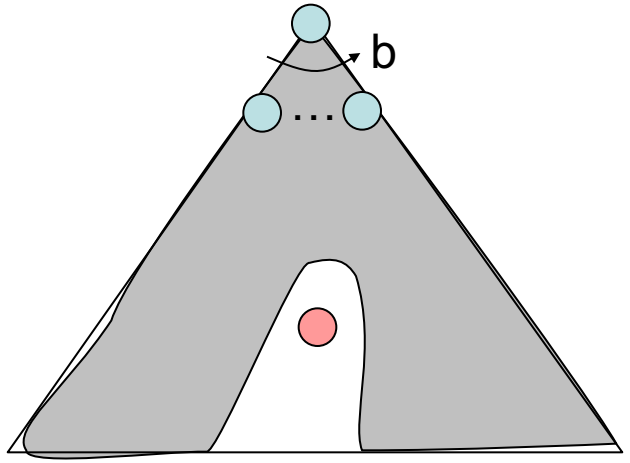
- What can go wrong?

Greedy Search

- Strategy: expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state



- A common case:
  - Best-first takes you straight to the (wrong) goal



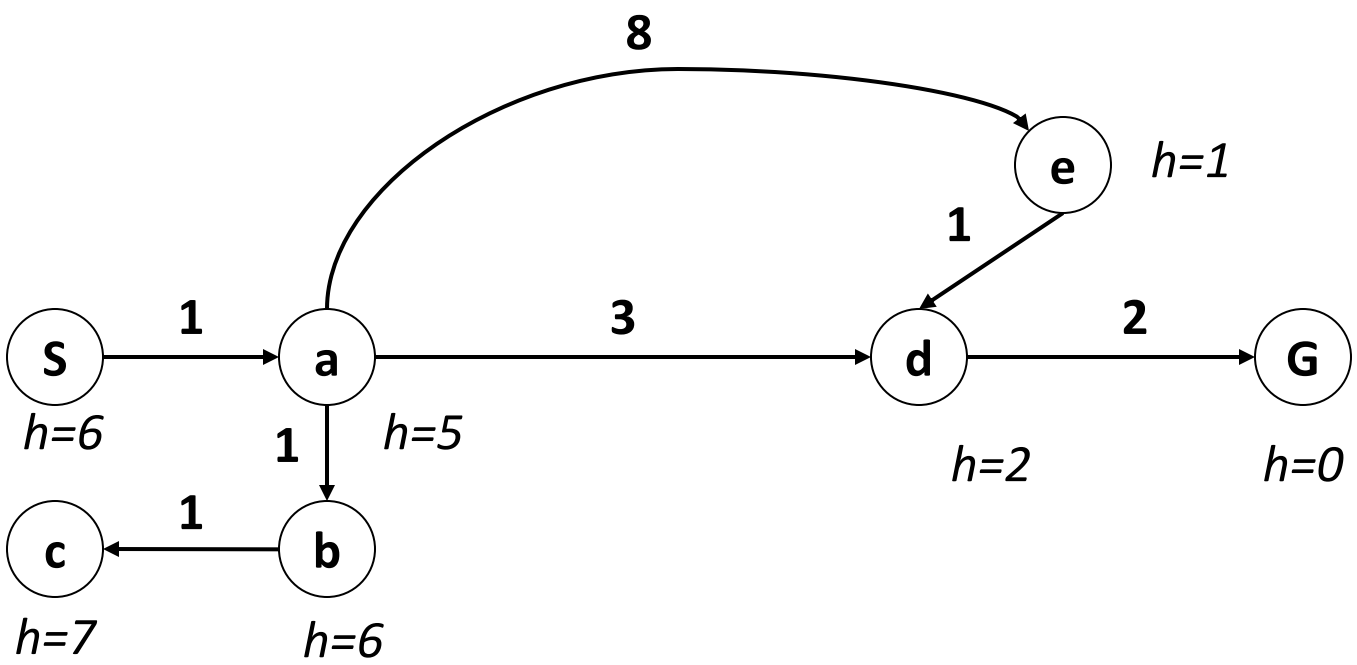
- Worst-case: like a badly-guided DFS

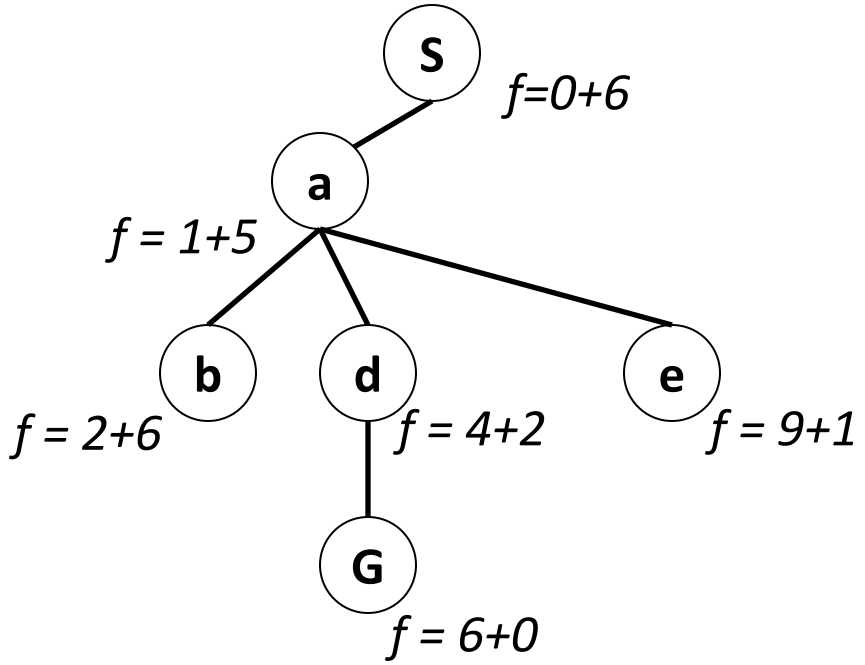
[Demo: contours greedy empty (L3D1)]
[Demo: contours greedy pacman small maze (L3D4)]

A* Search

Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)



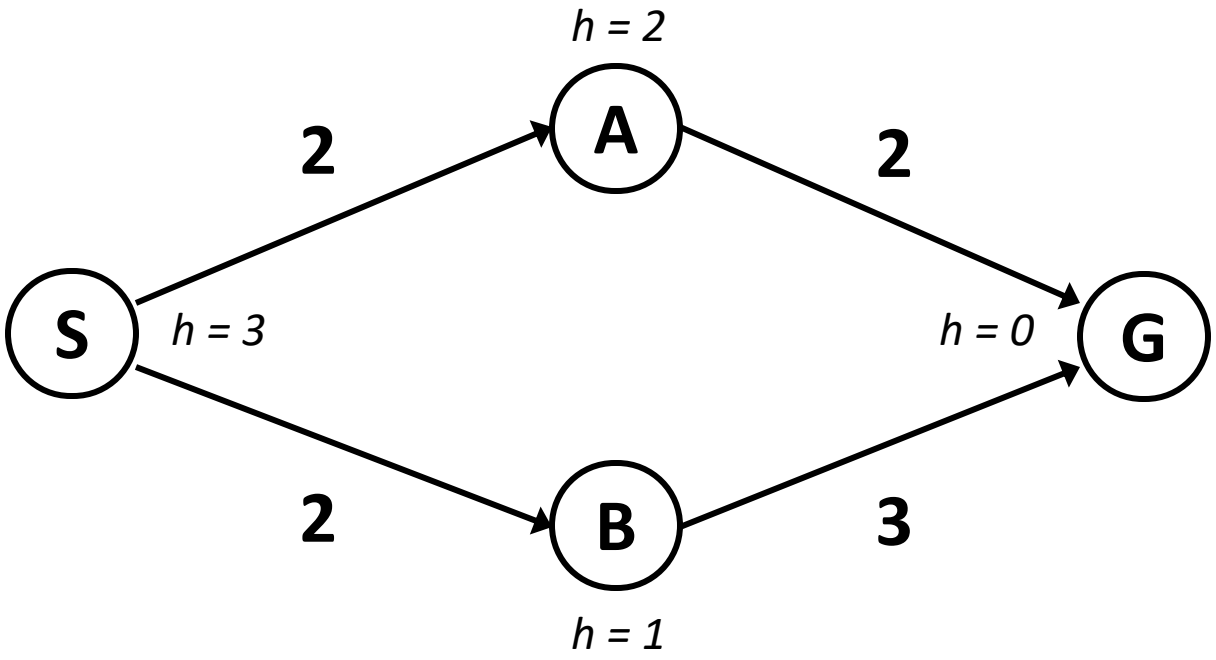


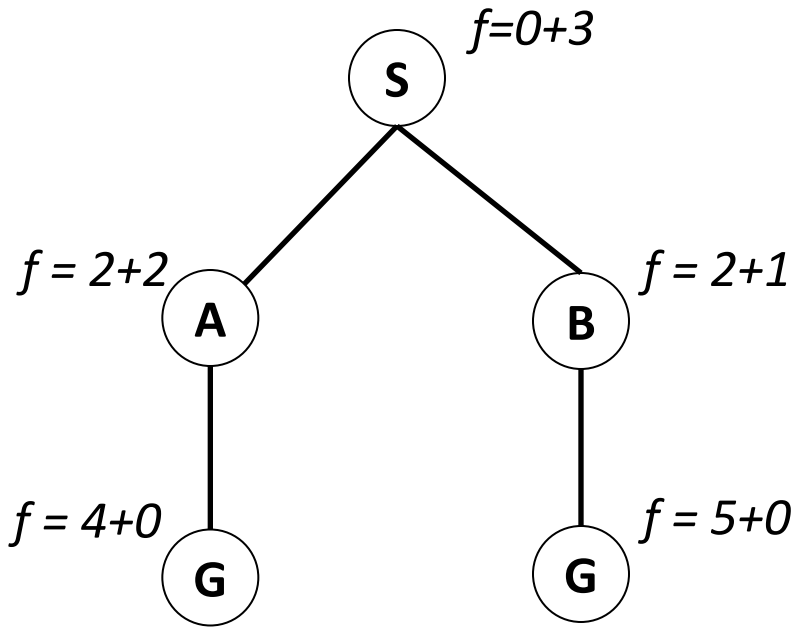
- A* Search orders by the sum: f(n) = g(n) + h(n)

Example: Teg Grenager

When should A* terminate?

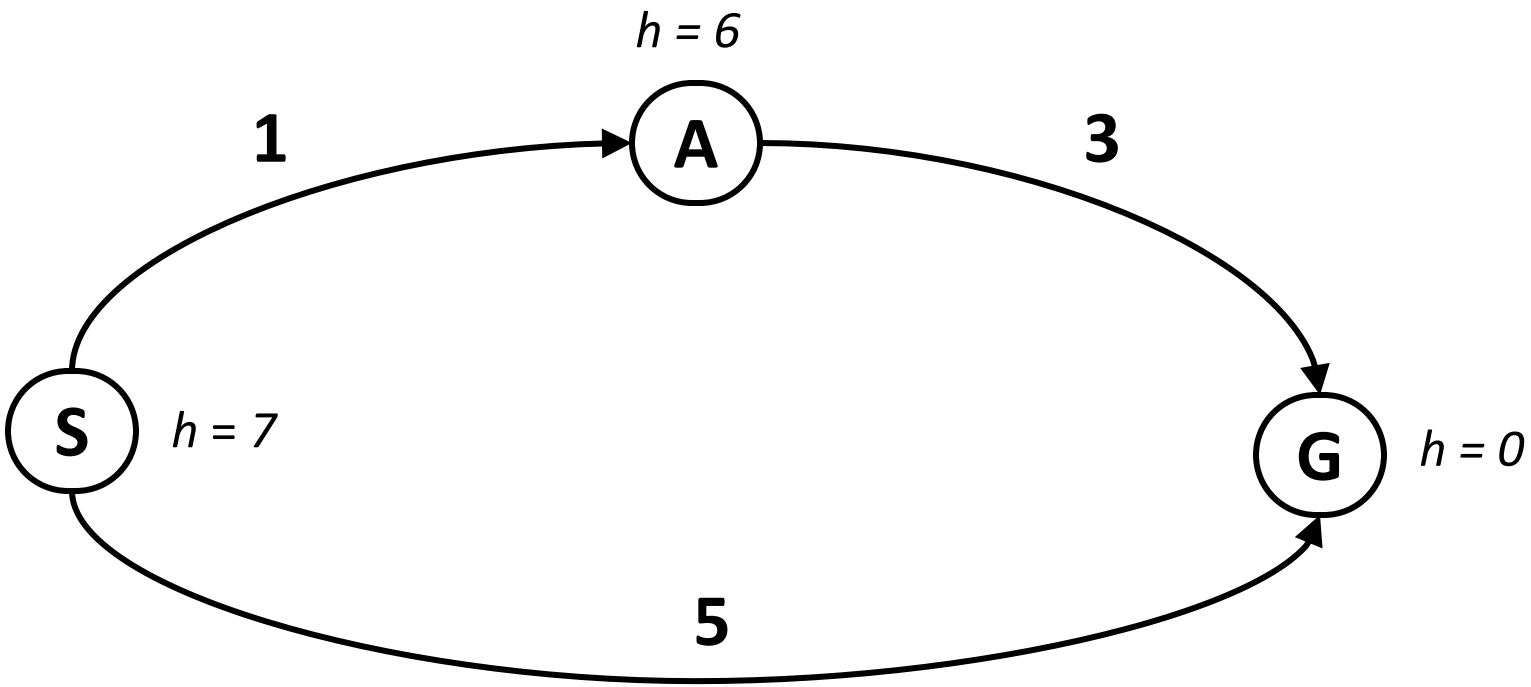
- Should we stop when we enqueue a goal?

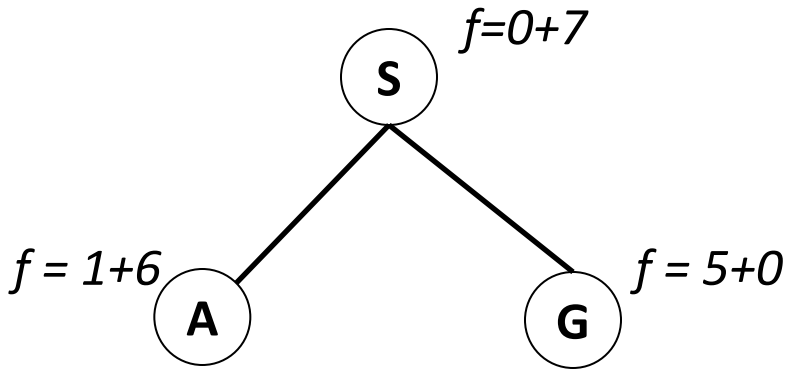




- No: only stop when we dequeue a goal

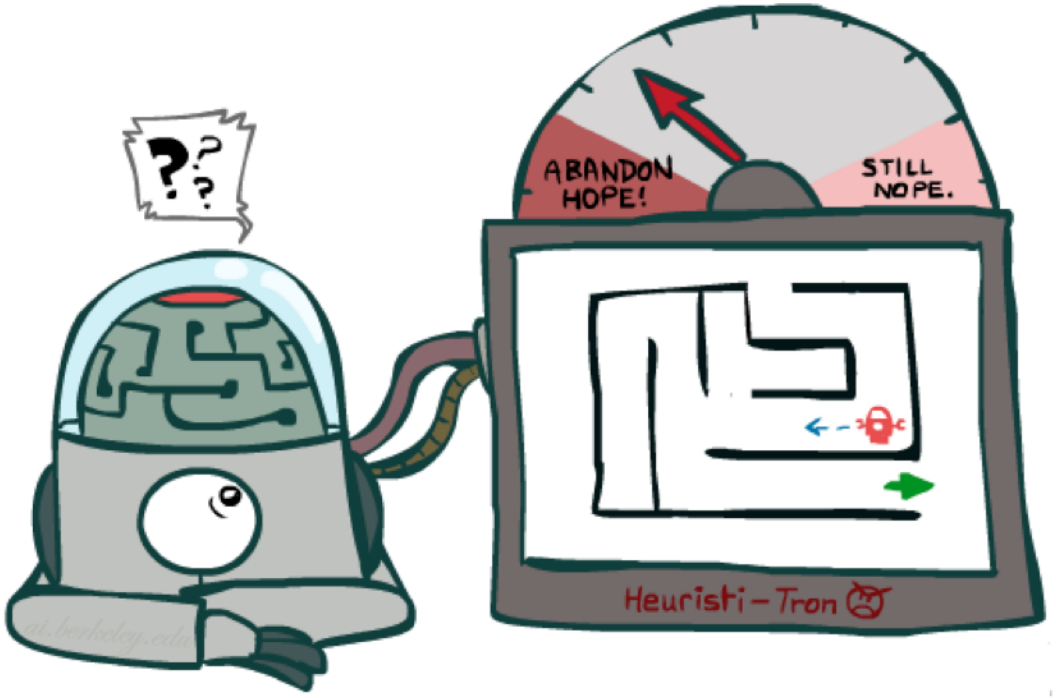
Is A* Optimal?



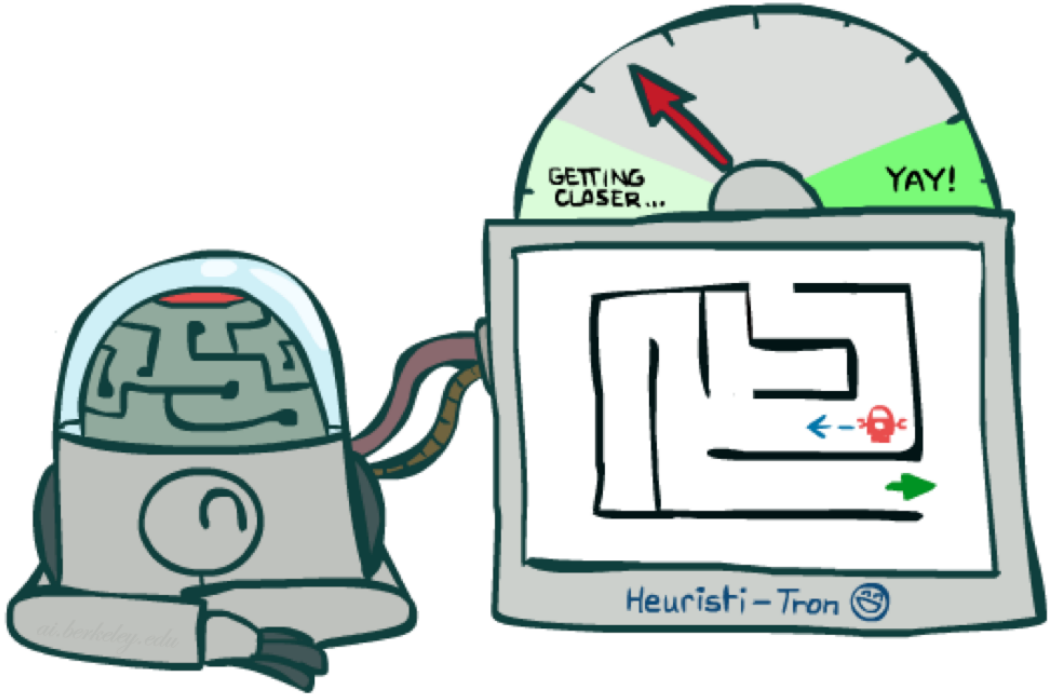


- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!

Idea: Admissibility



Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe



Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs

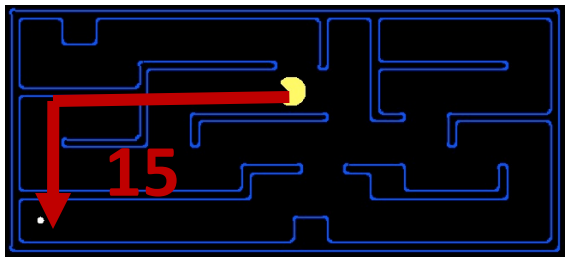
Admissible Heuristics

- A heuristic h is *admissible* (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to a nearest goal

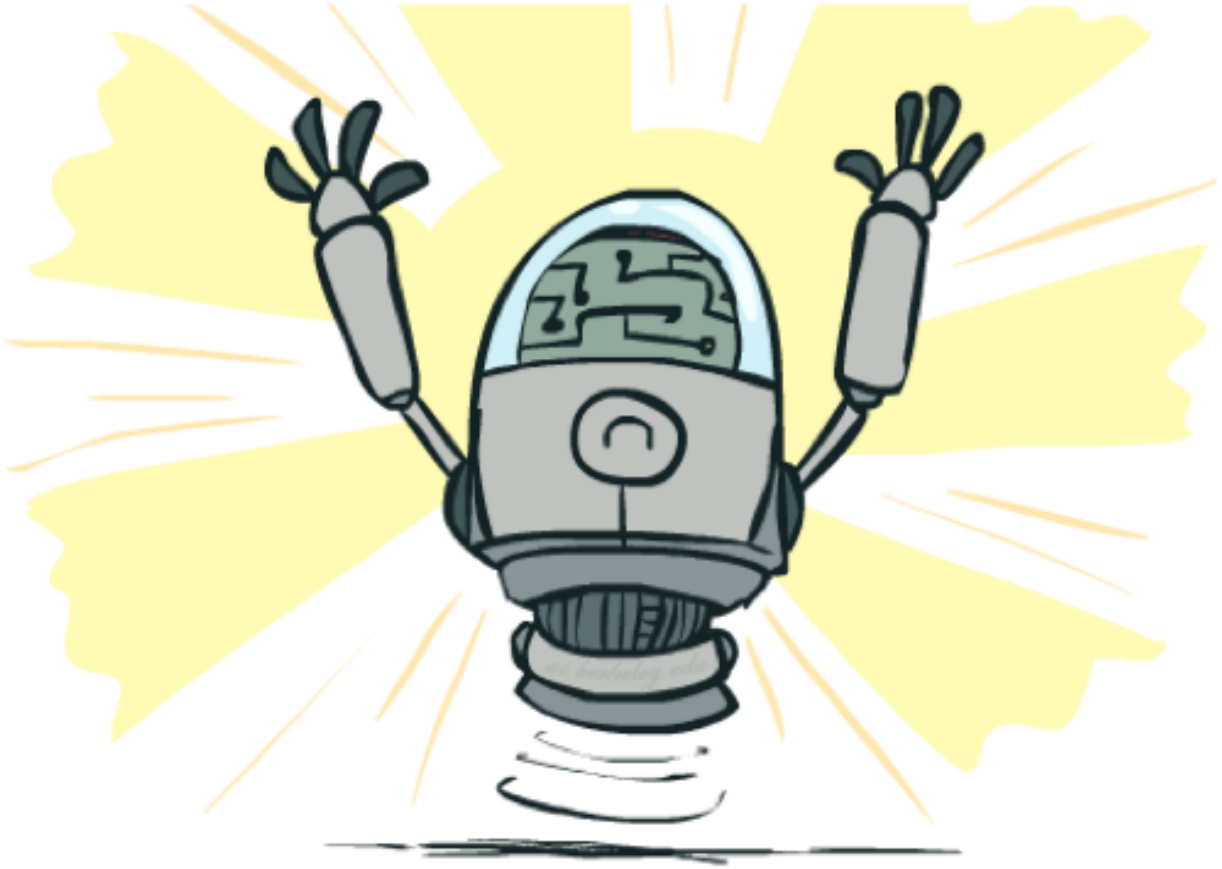
- Examples:



4

- Coming up with admissible heuristics is most of what's involved in using A* in practice.

Optimality of A* Tree Search



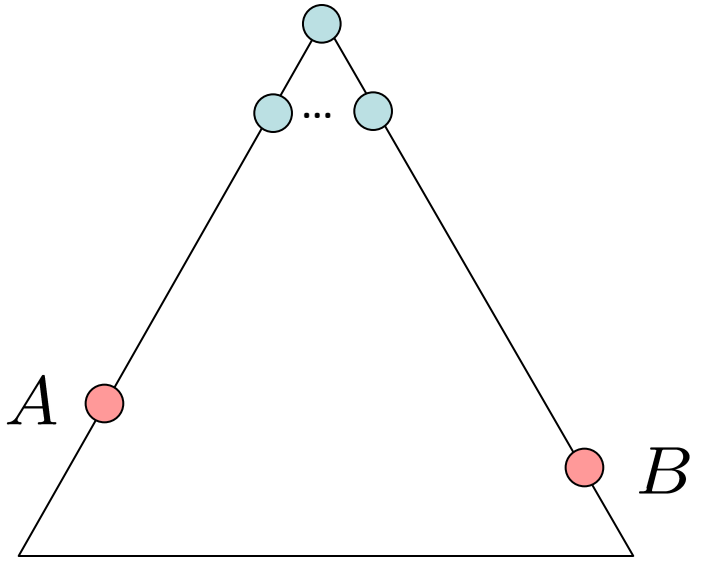
Optimality of A* Tree Search

Assume:

- A is an optimal goal node
- B is a suboptimal goal node
- h is admissible

Claim:

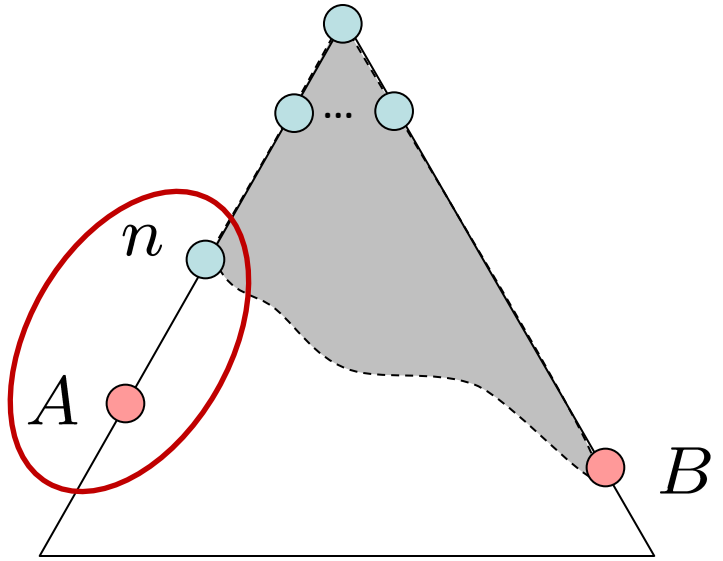
- A will exit the fringe before B



Optimality of A* Tree Search: Blocking

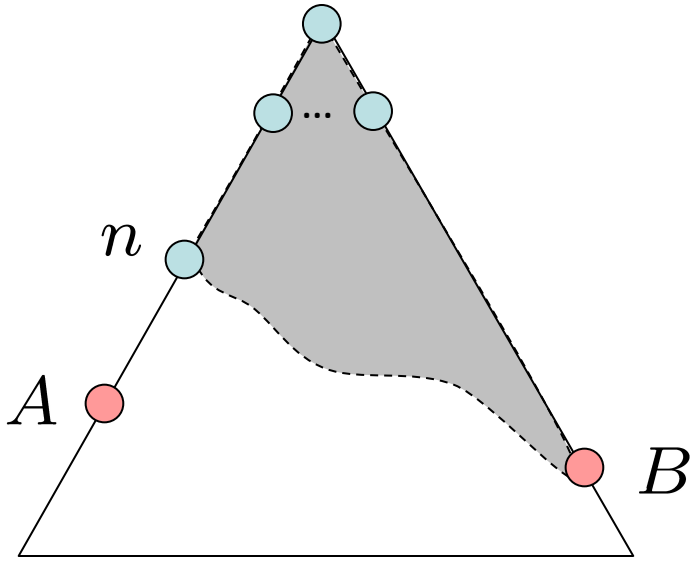
Proof:

- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
  1. f(n) is less or equal to f(A)



Optimality of A* Tree Search: Blocking

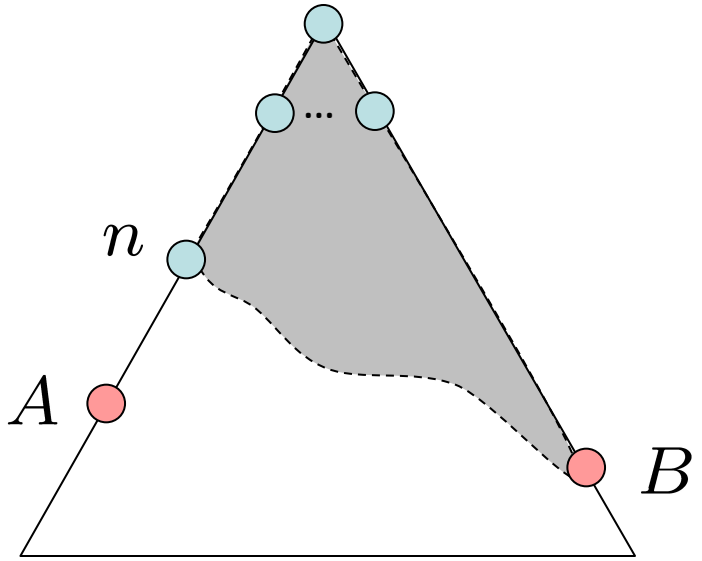
1. f(n) is less than or equal to f(A)
   - Definition of f-cost says:
     f(n) = g(n) + h(n) = (path cost to n) + (est. cost of n to A)
     f(A) = g(A) + h(A) = (path cost to A) + (est. cost of A to A)
   - The admissible heuristic must underestimate the true cost
     h(A) = (est. cost of A to A) = 0
   - So now, we have to compare:
     f(n) = g(n) + h(n) = (path cost to n) + (est. cost of n to A)
     f(A) = g(A) = (path cost to A)
   - h(n) must be an underestimate of the true cost from n to A
     (path cost to n) + (est. cost of n to A) ≤ (path cost to A)
     g(n) + h(n) ≤ g(A)
     f(n) ≤ f(A)



Optimality of A* Tree Search: Blocking

Proof:

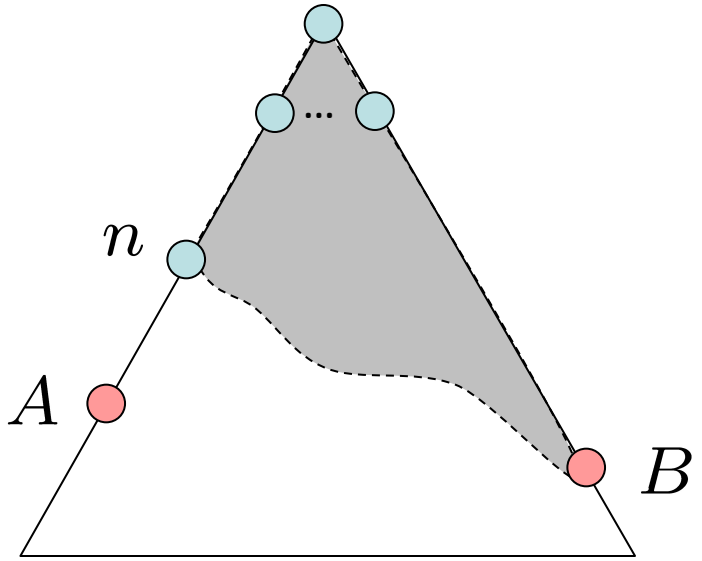
- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
  1. f(n) is less or equal to f(A)
  2. f(A) is less than f(B)



Optimality of A* Tree Search: Blocking

2. f(A) is less than f(B)

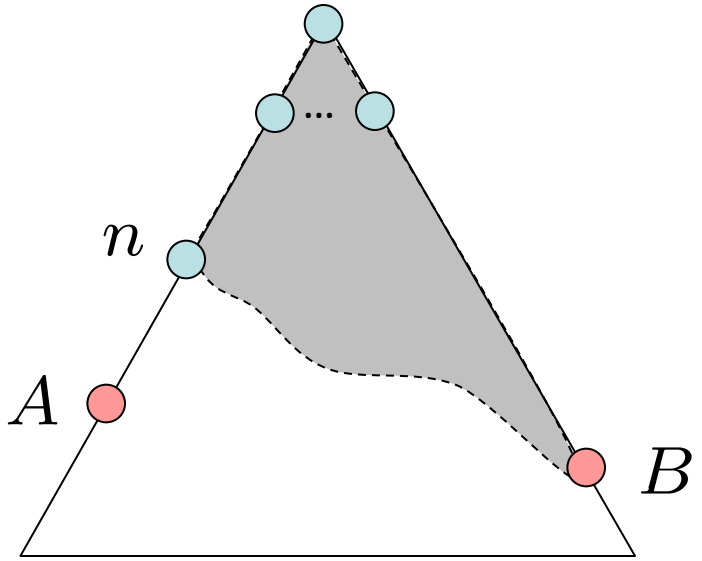
- We know that:
  f(A) = g(A) + h(A) = (path cost to A) + (est. cost of A to A)
  f(B) = g(B) + h(B) = (path cost to B) + (est. cost of B to B)
- The heuristic must underestimate the true cost:
  h(A) = h(B) = 0
- So now, we have to compare:
  f(A) = g(A) = (path cost to A)
  f(B) = g(B) = (path cost to B)
- We assumed that B is suboptimal! So
  (path cost to A) < (path cost to B)
  g(A) < g(B)
  f(A) < f(B)



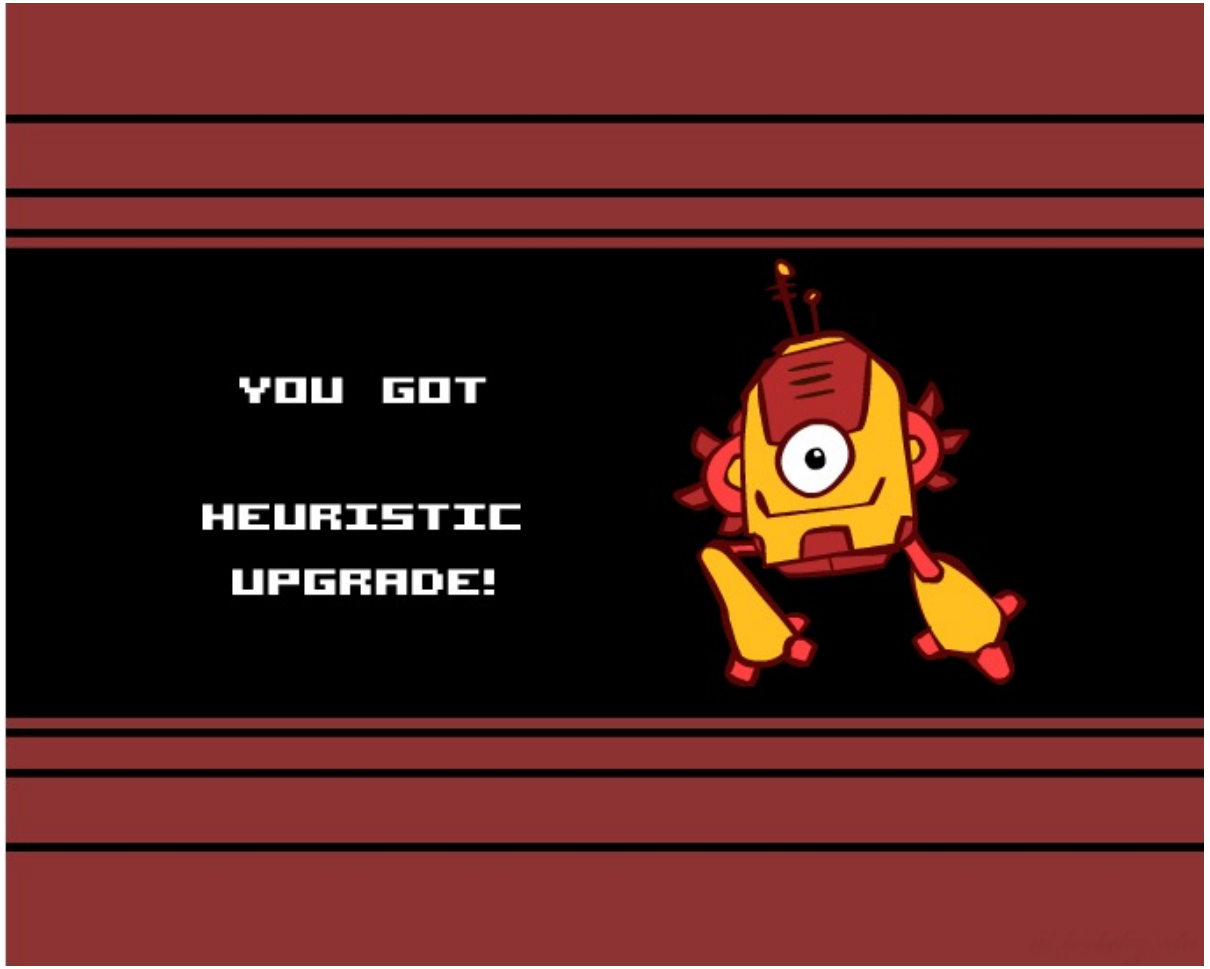
Optimality of A* Tree Search: Blocking

Proof:

- Imagine B is on the fringe
- Some ancestor *n* of A is on the fringe, too (maybe A!)
- Claim: *n* will be expanded before B
  1. f(n) is less or equal to f(A)
  2. f(A) is less than f(B)
  3. *n* expands before B
- All ancestors of A expand before B
- A expands before B
- A* search is optimal



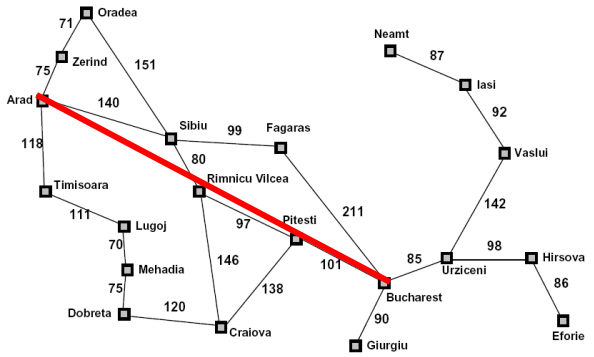
Creating Heuristics

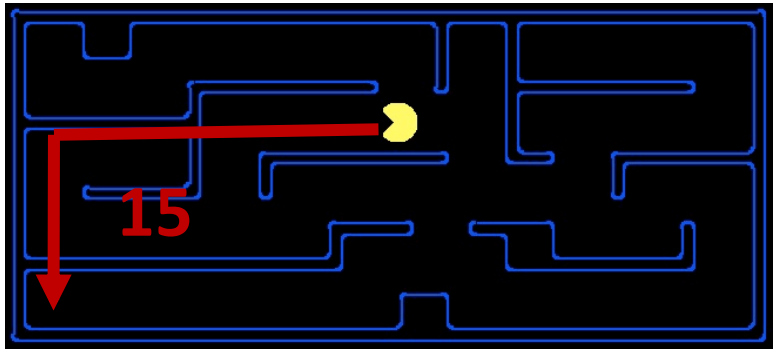


Creating Admissible Heuristics

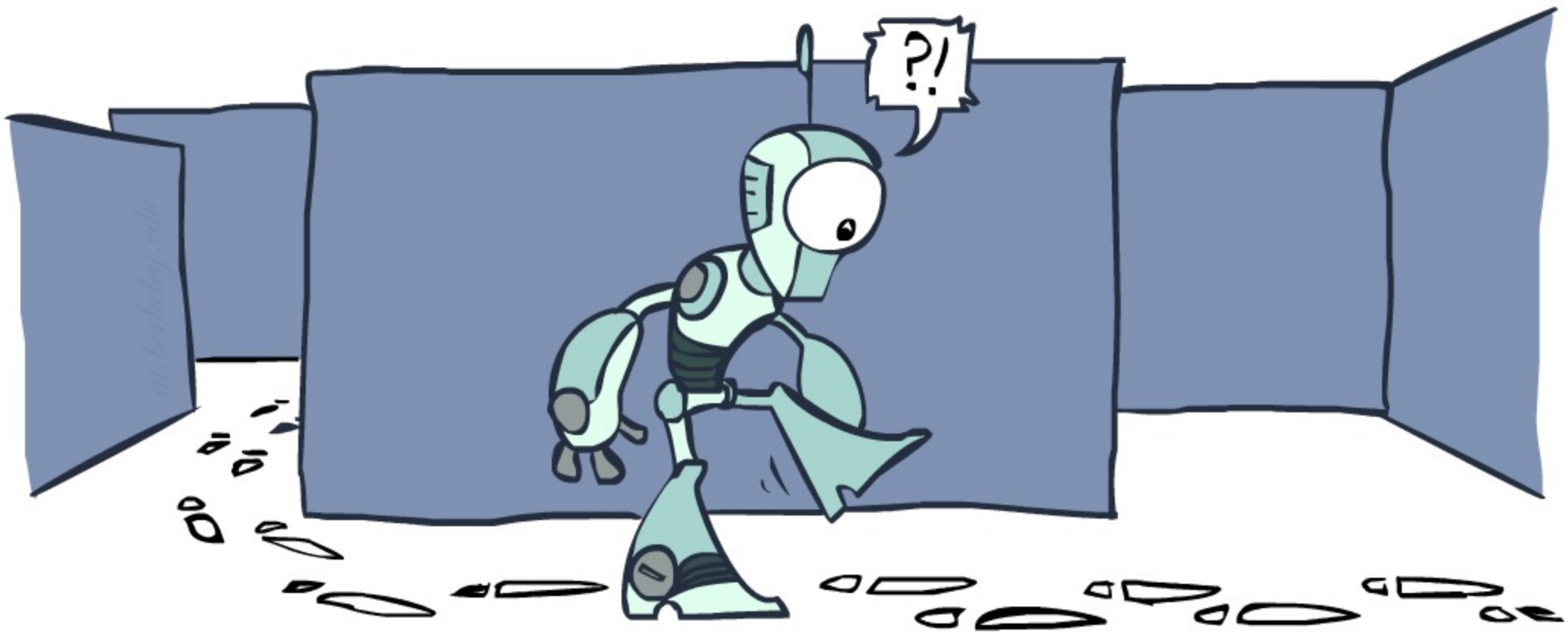
- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics
- Often, admissible heuristics are solutions to *relaxed problems,* where new actions are available

366



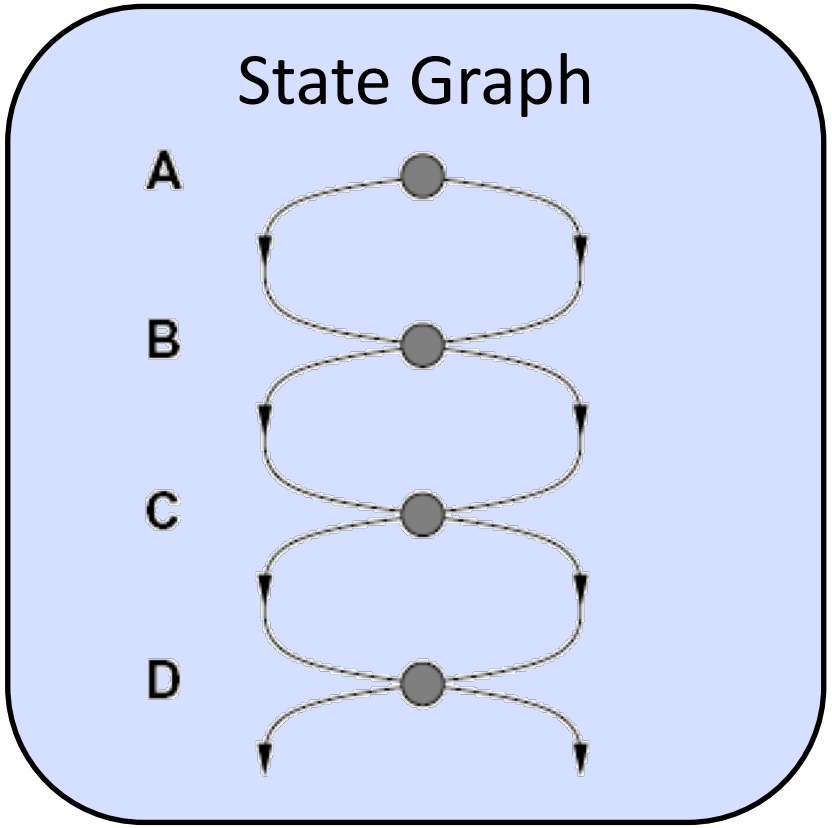


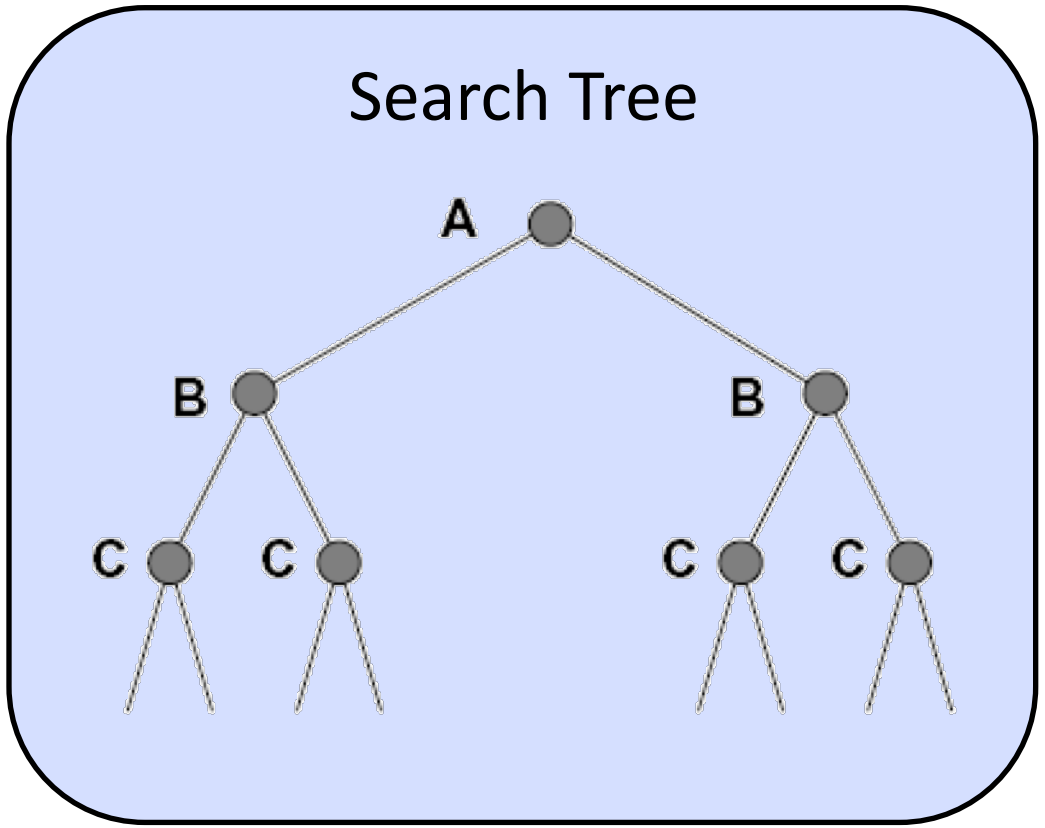
Graph Search



Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work.





Graph Search

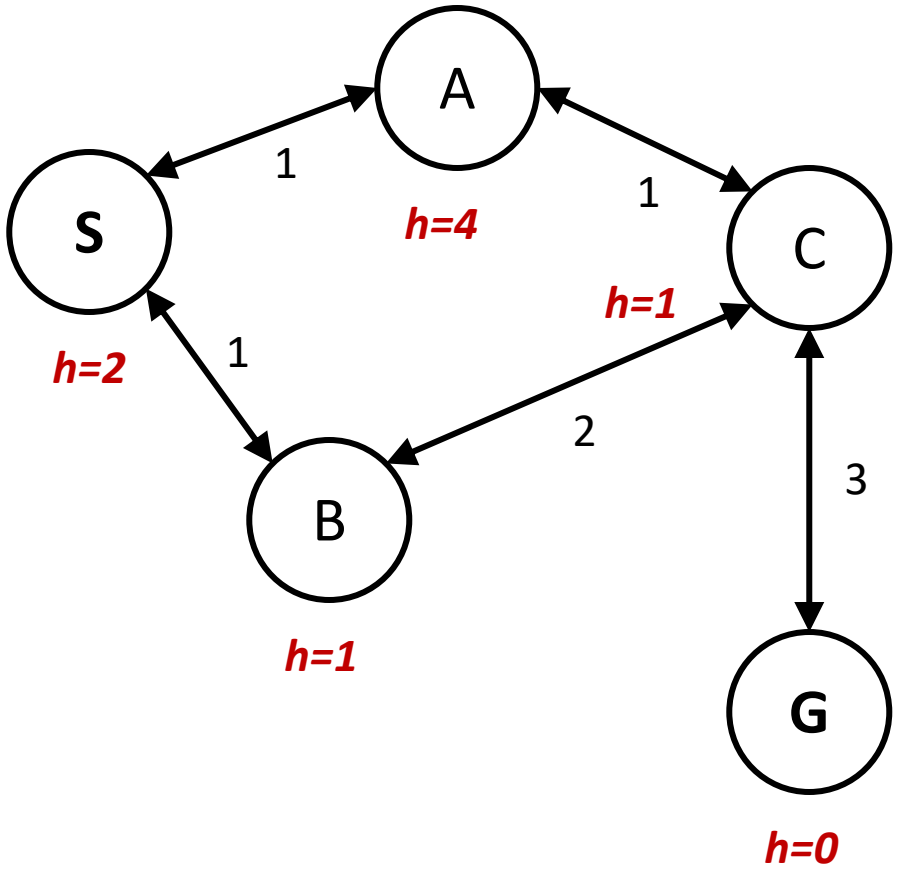
- Idea: never expand a state twice
- How to implement:
  - Tree search + set of expanded states (“closed set”)
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set
- Important: store the closed set as a set, not a list
- Can graph search wreck completeness? Why/why not?
- How about optimality?

Graph Search Pseudo-Code

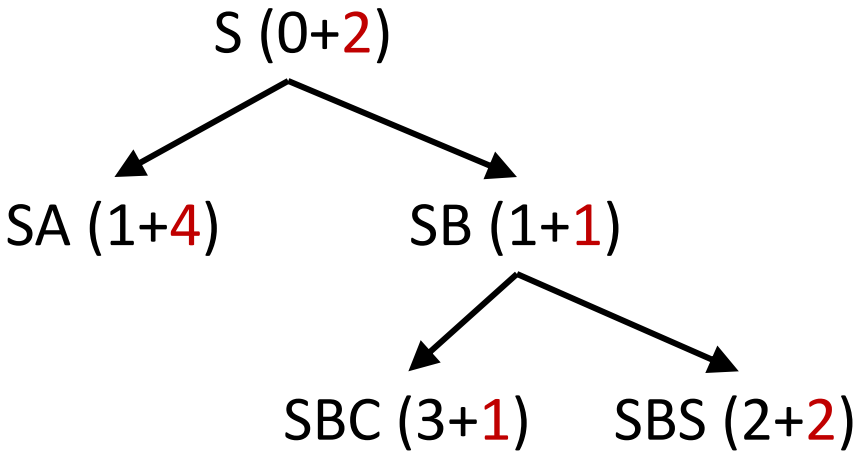
```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

A* Graph Search Gone Wrong?

State space graph



Search tree

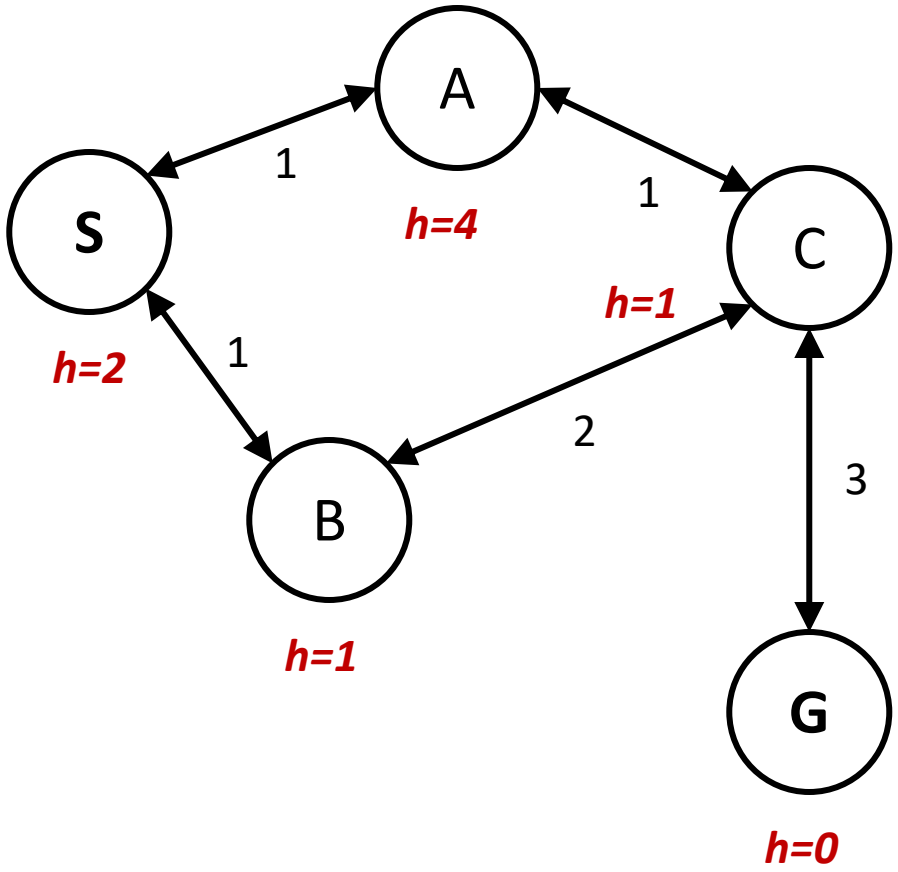


Closed set

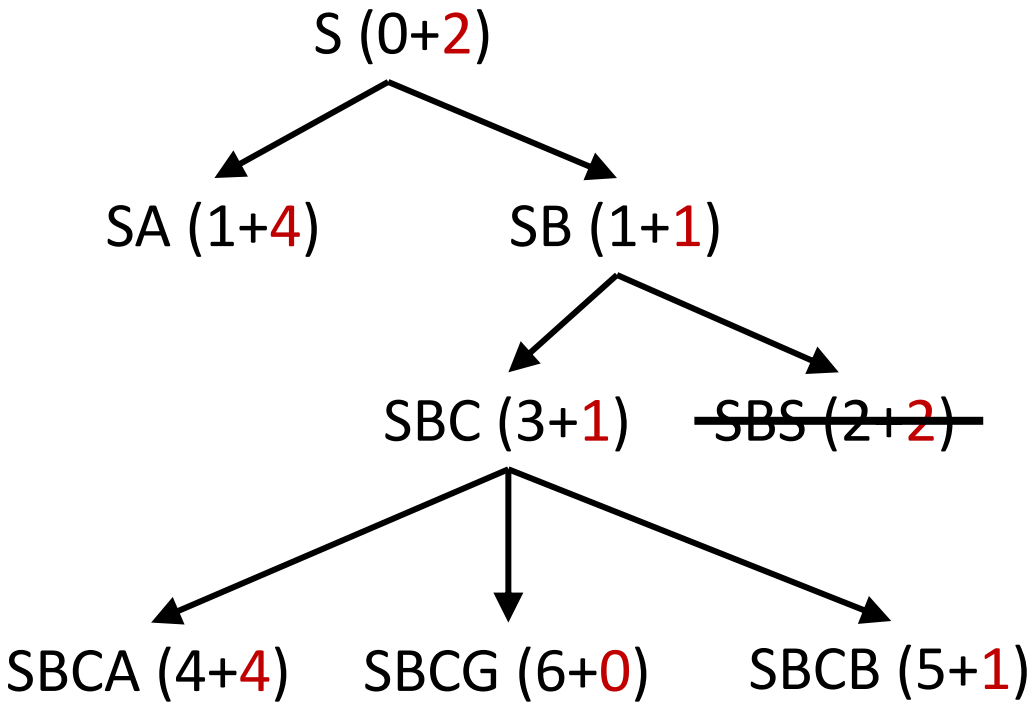
{ S B }

A* Graph Search Gone Wrong?

State space graph



Search tree

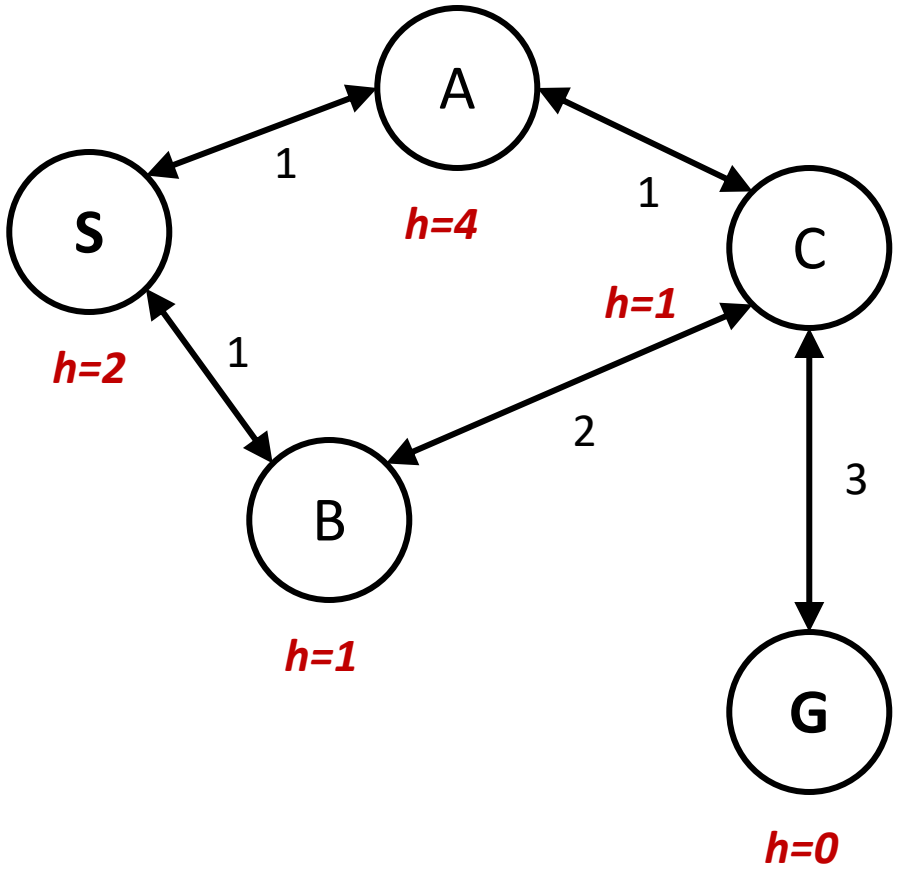


Closed set

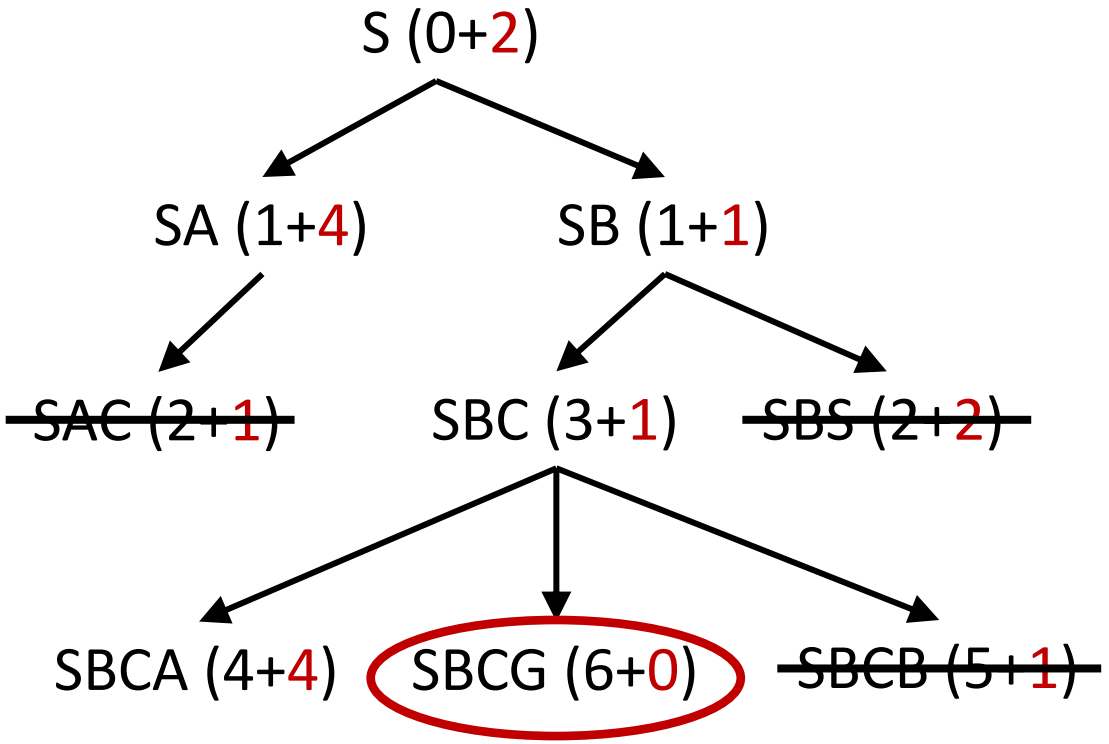
{ S B }

A* Graph Search Gone Wrong?

State space graph



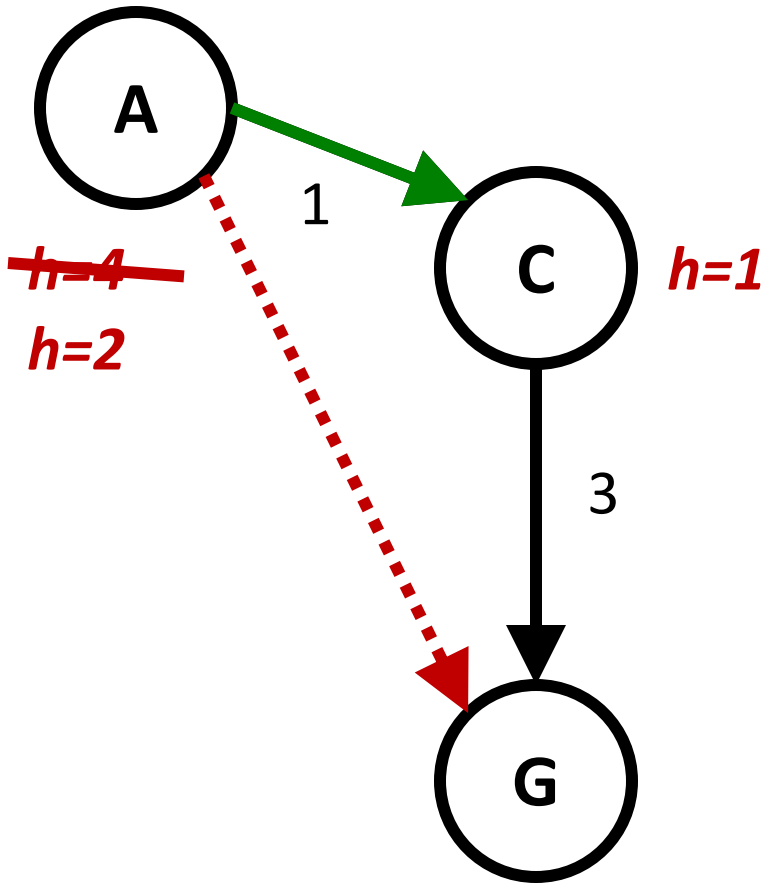
Search tree



Closed set

{ S B C A }

Consistency of Heuristics



- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal

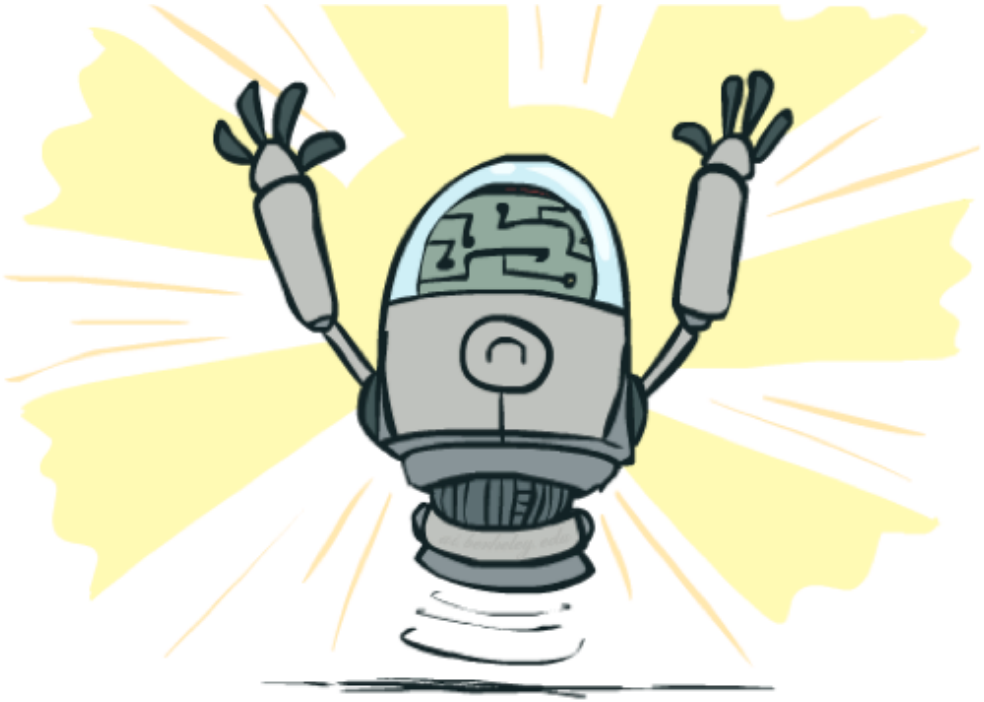
    h(A) ≤ actual cost from A to G
  - Consistency: heuristic "arc" cost ≤ actual cost for each arc

    h(A) – h(C) ≤ cost(A to C)
- Consequences of consistency:
  - The f value along a path never decreases

    h(A) ≤ cost(A to C) + h(C)
  - A* graph search is optimal

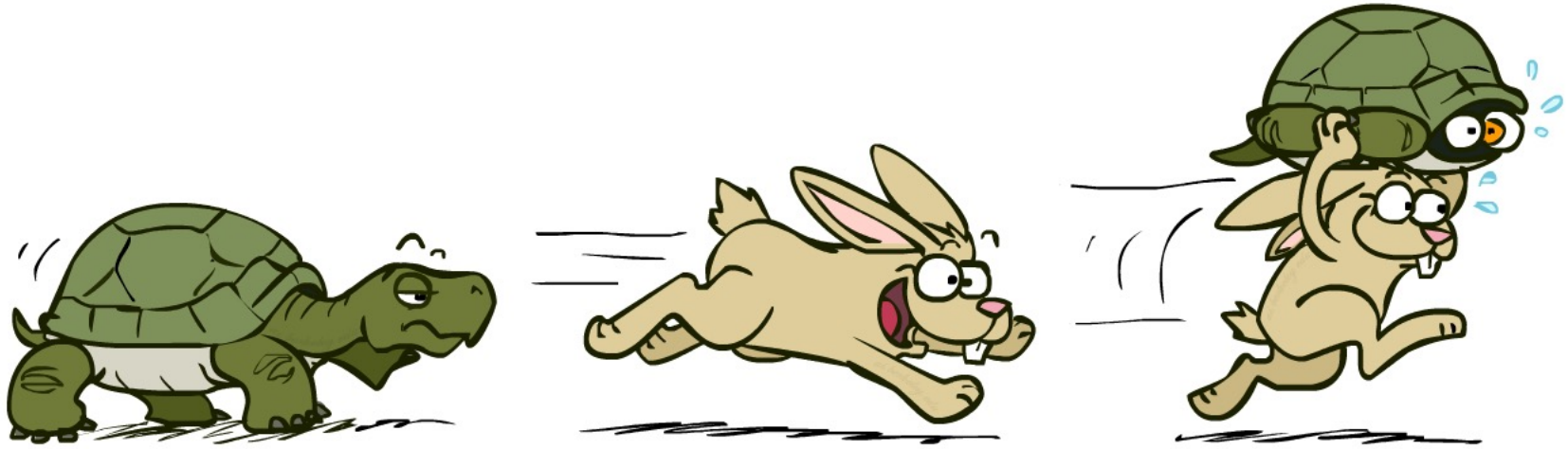
Optimality

- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissibility
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems



A*: Summary

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems



南方科技大学

STA303: Artificial Intelligence

Games

Fang Kong

https://fangkongx.github.io/

Part of slide credits: ai.berkeley.edu

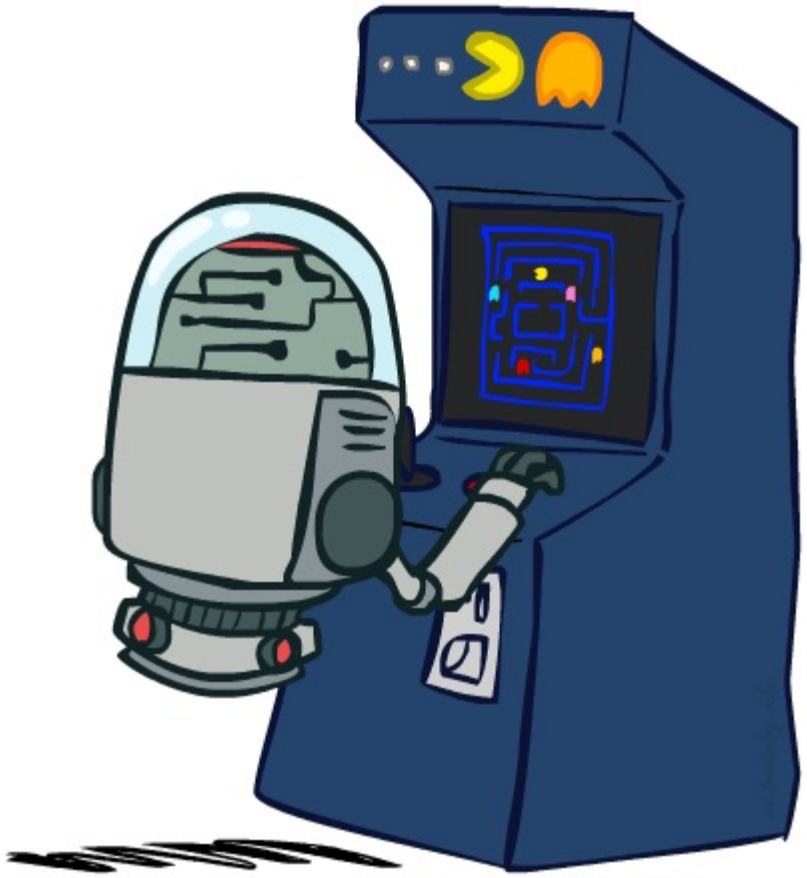
Types of Games

- Game = task environment with > 1 agent
- Axes:
  - Deterministic or stochastic?
  - Perfect information (fully observable)?
  - Two, three, or more players?
  - Teams or individuals?
  - Turn-taking or simultaneous?
  - Zero sum?

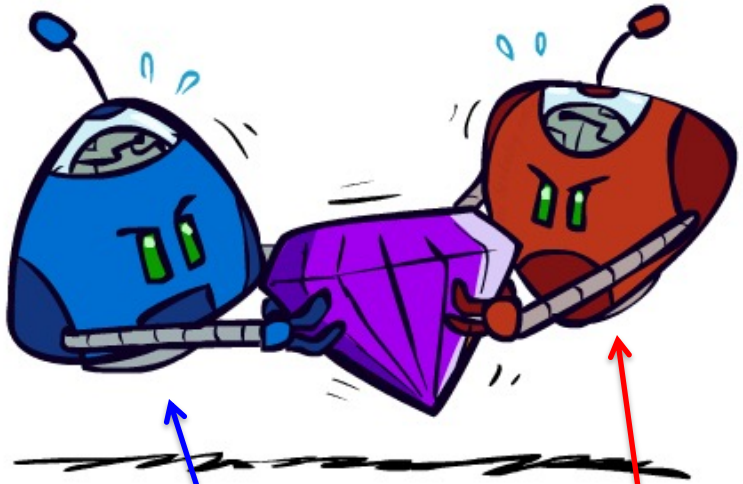
- Want algorithms for calculating a **strategy** (policy) which recommends a move from every possible state

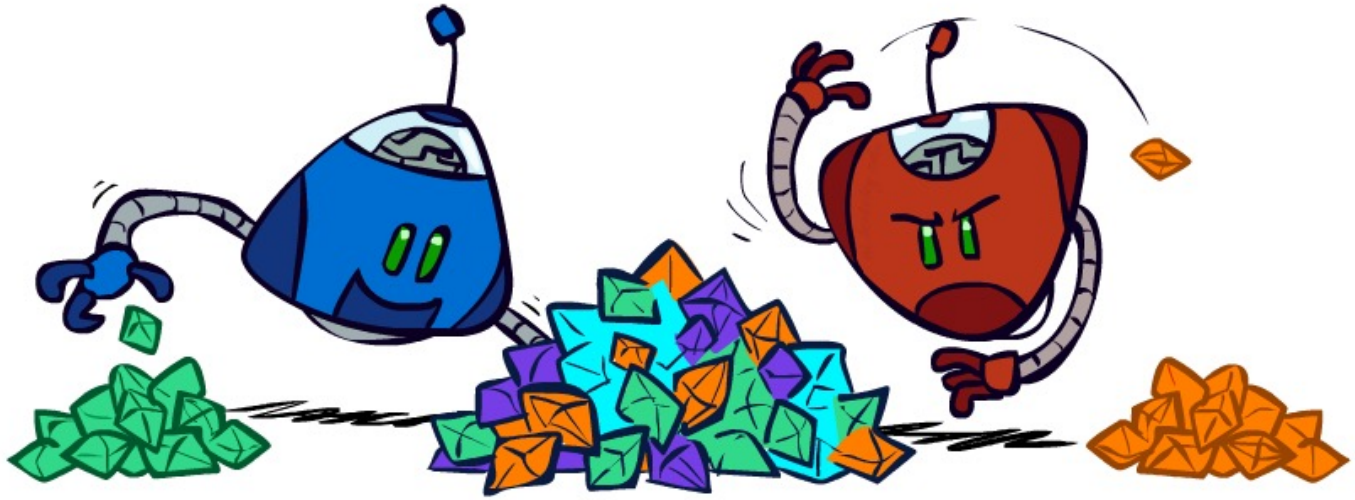
Deterministic Games

- 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: S x A → S
  - Terminal test: S → {true, false}
  - Terminal utilities: S x P → R
- Solution for a player is a policy: S → A



Zero-Sum Games





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

- General-Sum Games
  - Agents have ***independent*** utilities
  - Cooperation, indifference, competition, shifting alliances, and more are all possible
- Team Games
  - Common payoff for all team members

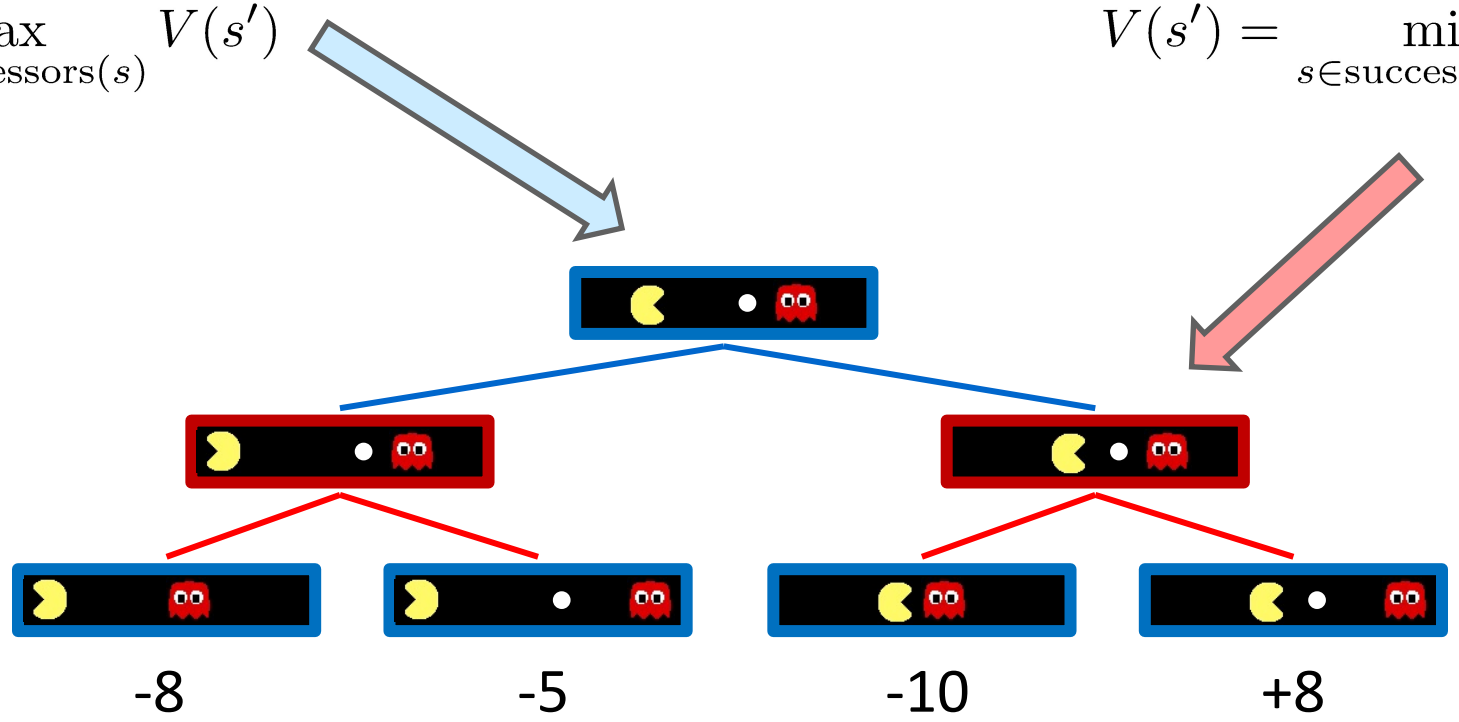
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

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



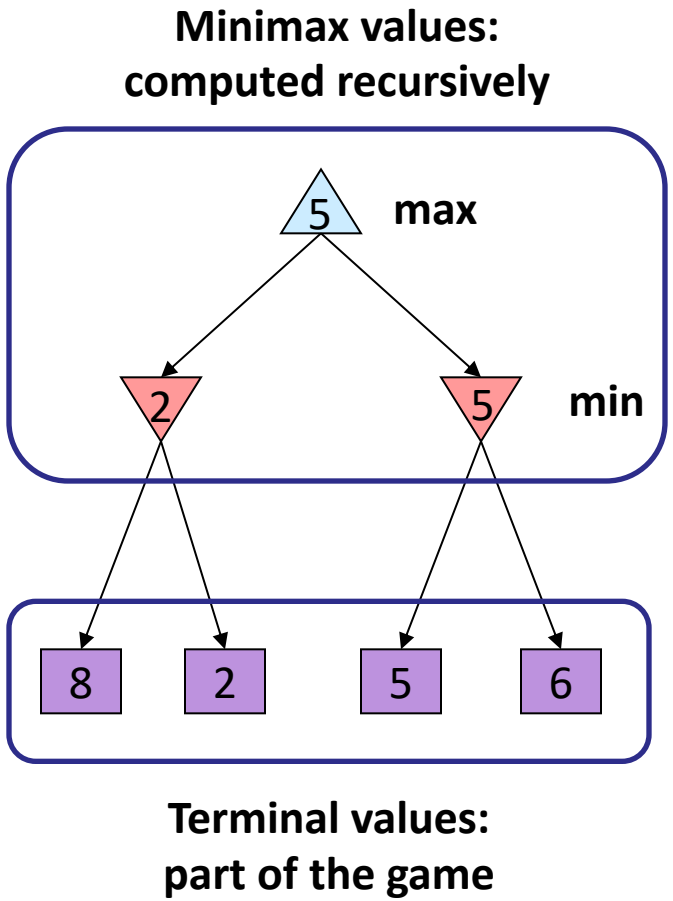
Terminal States:

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

Adversarial Search (Minimax)

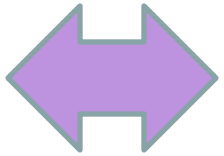
- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

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



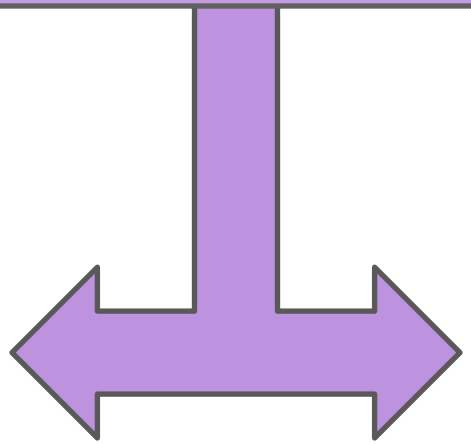
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

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

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

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)
```



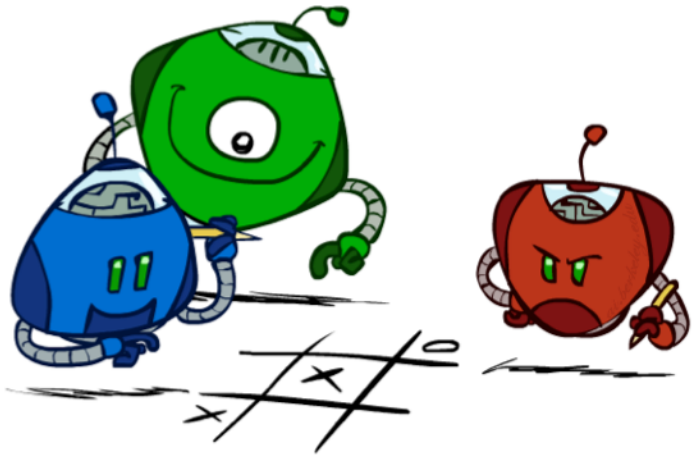
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

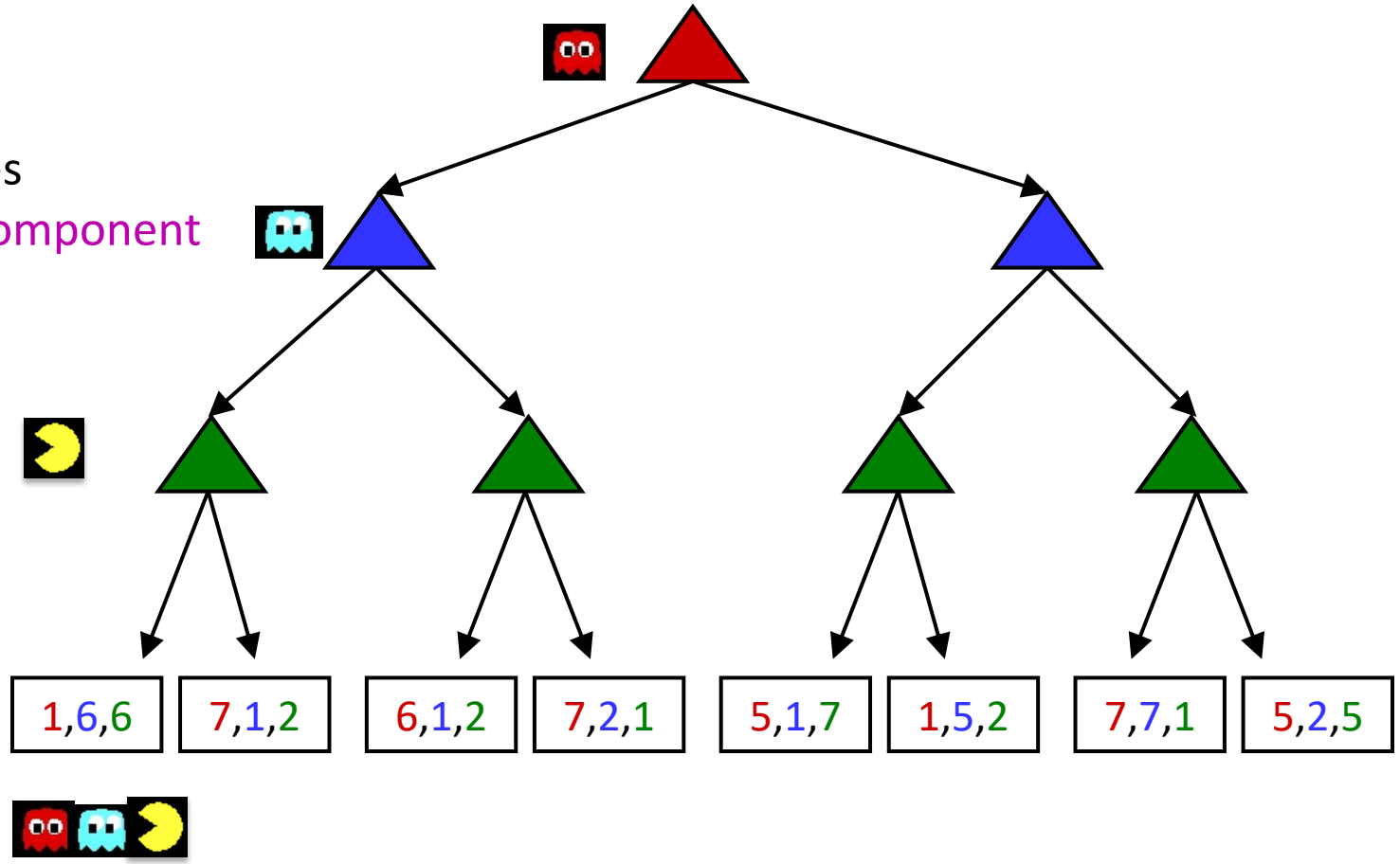
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

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…

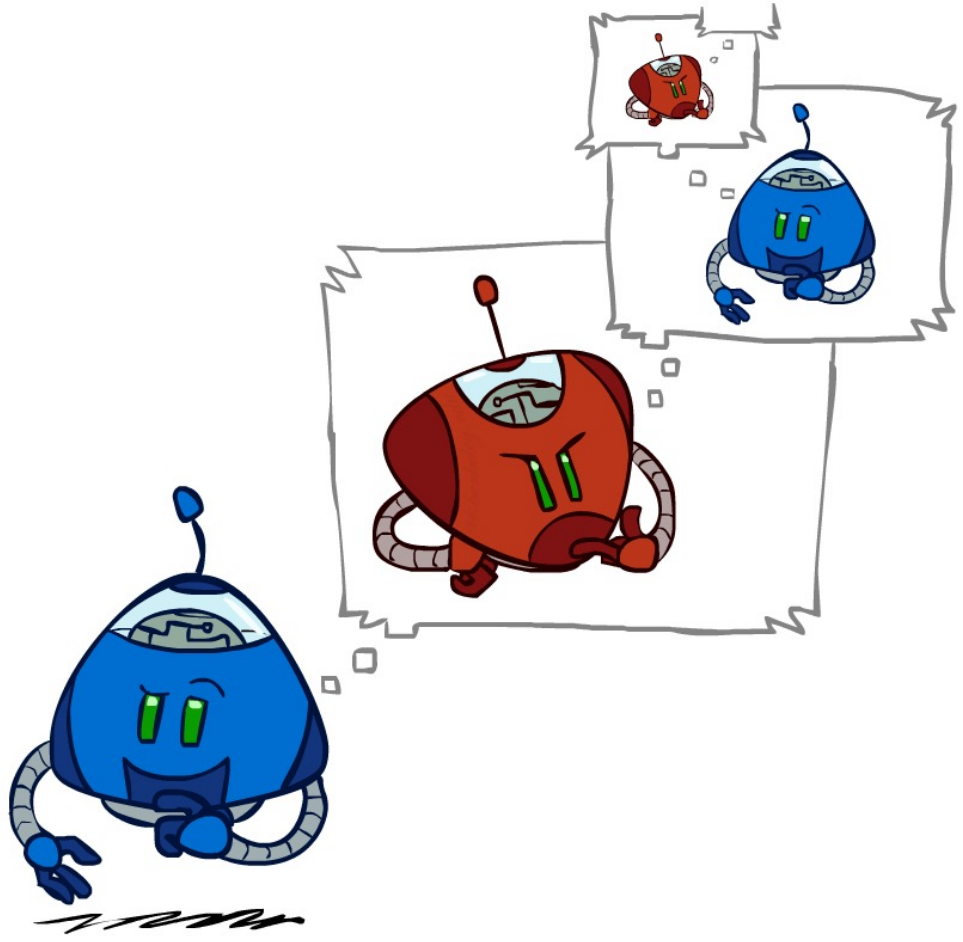




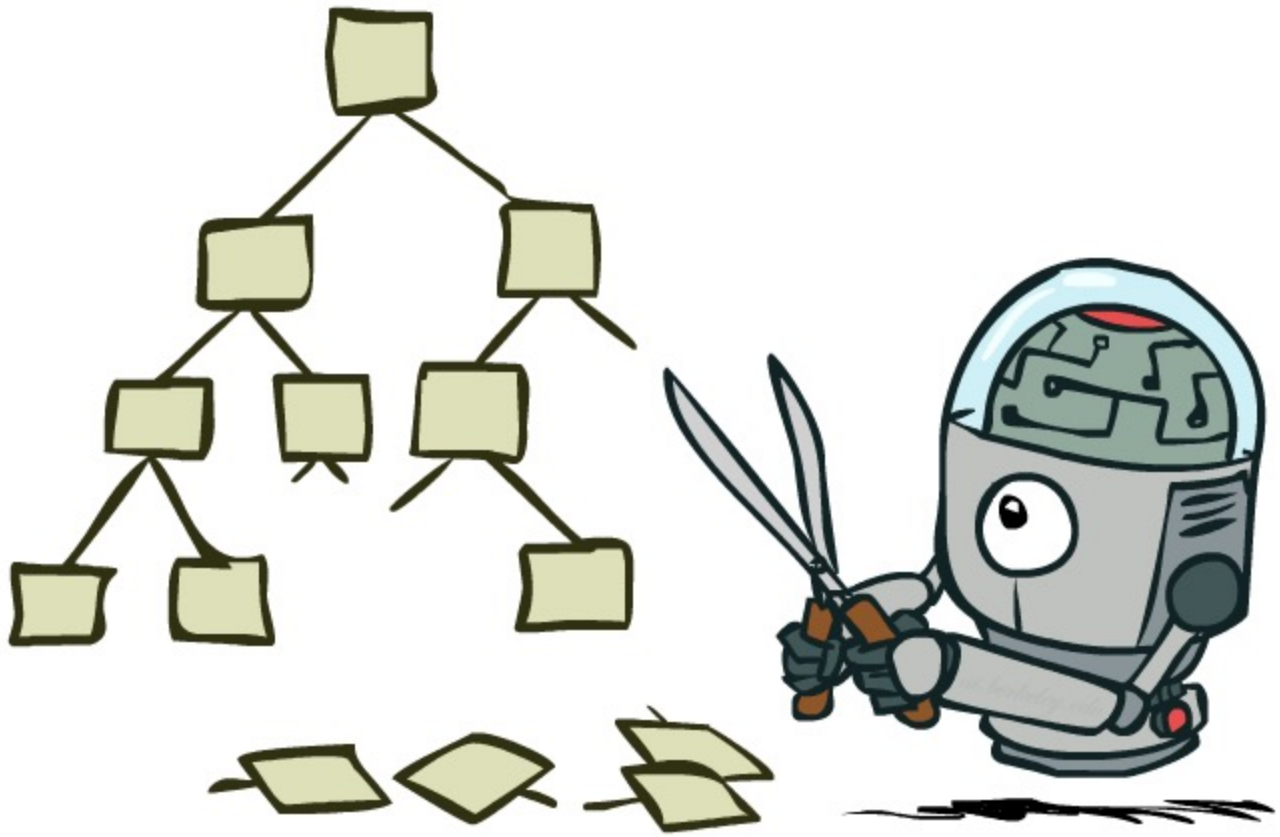
Minimax Efficiency

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

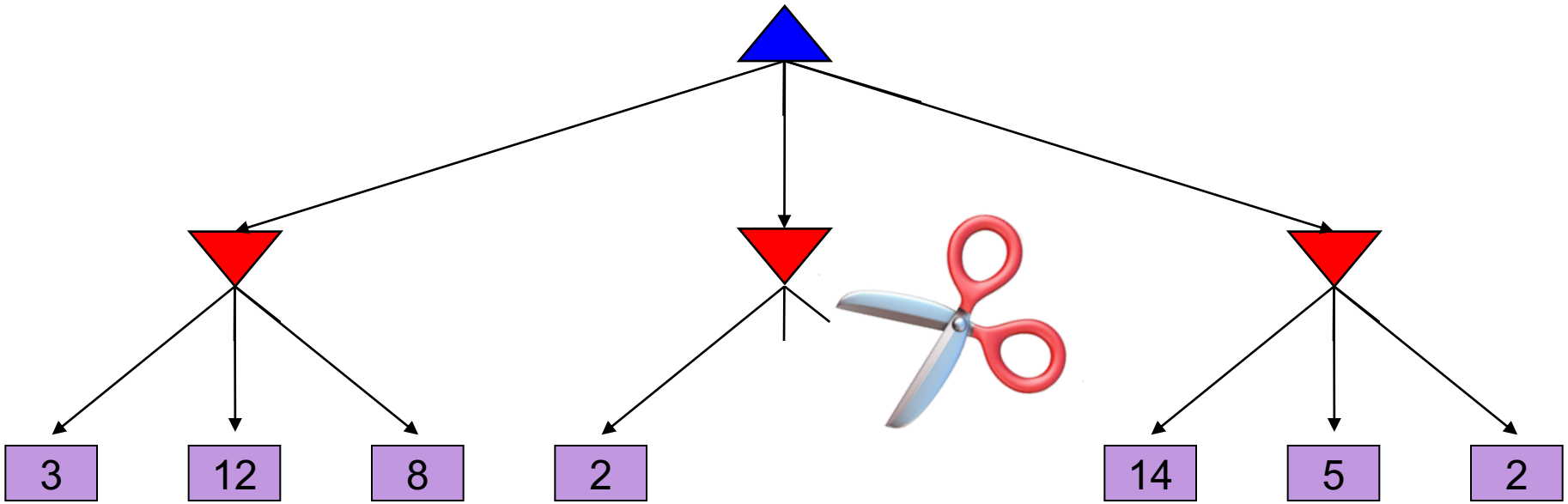
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?



Game Tree Pruning



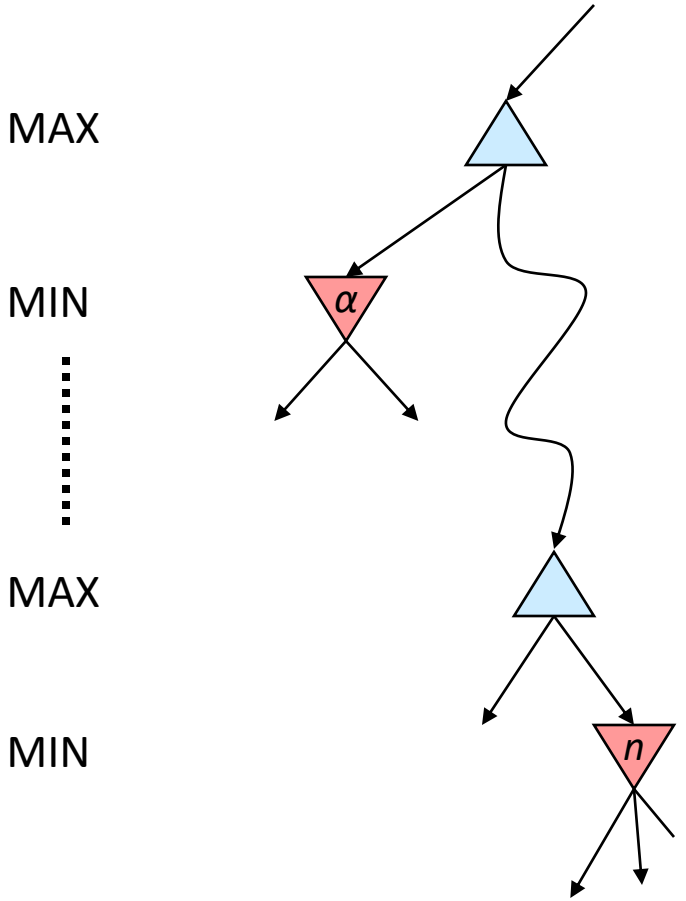
Minimax Pruning



The order of generation matters:
more pruning is possible if good moves come first

Alpha-Beta Pruning

- General case (pruning children of MIN node)
  - 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 ***α*** be the best value that MAX can get so far at any choice point along the current path from the root
  - If *n* becomes worse than ***α***, MAX will avoid it, so we can prune *n*'s other children (it's already bad enough that it won't be played)
- Pruning children of MAX node is symmetric
  - Let **β** be the best value that MIN can get so far at any choice point along the current path from the root



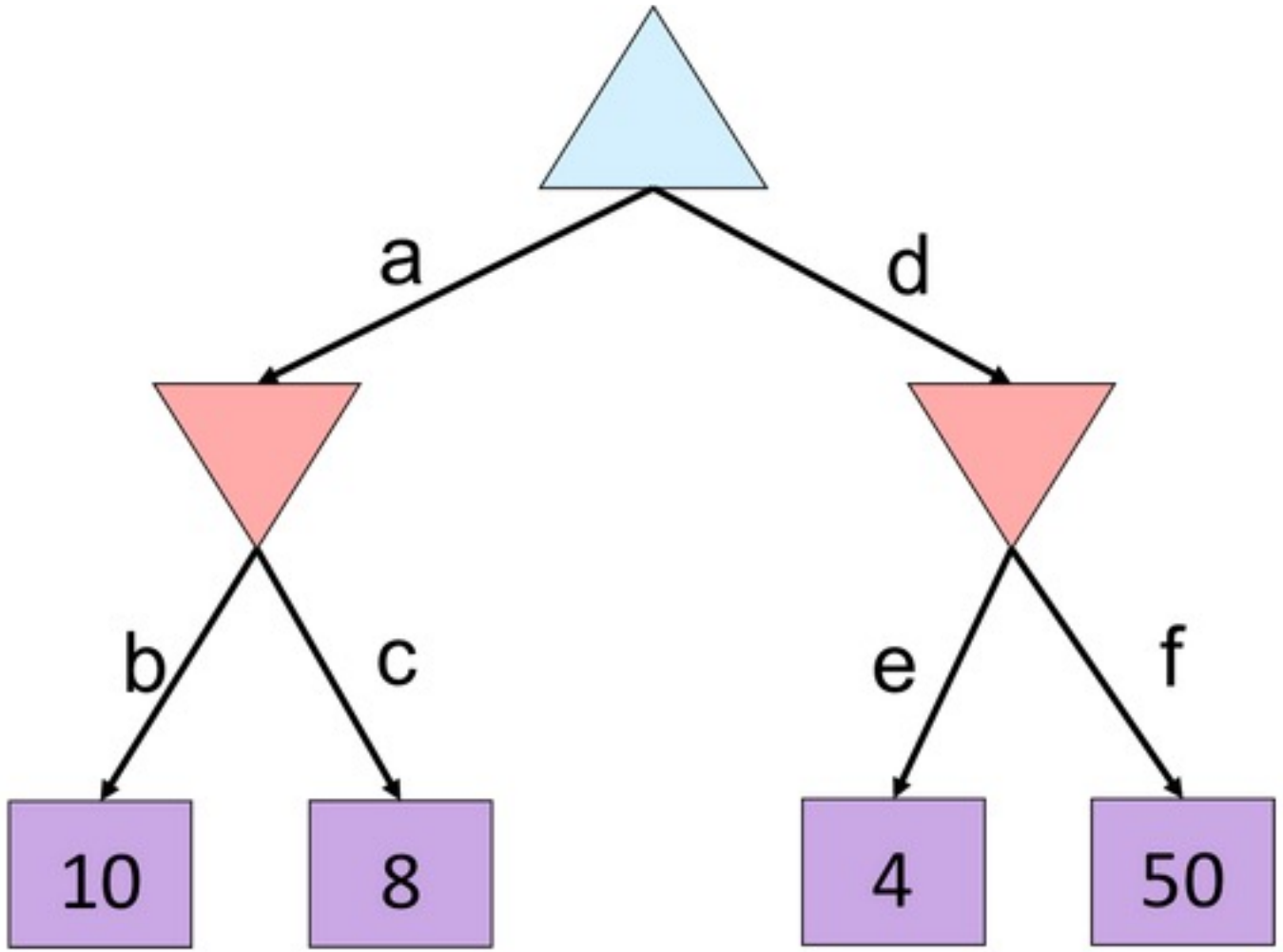
Alpha-Beta Implementation

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
```

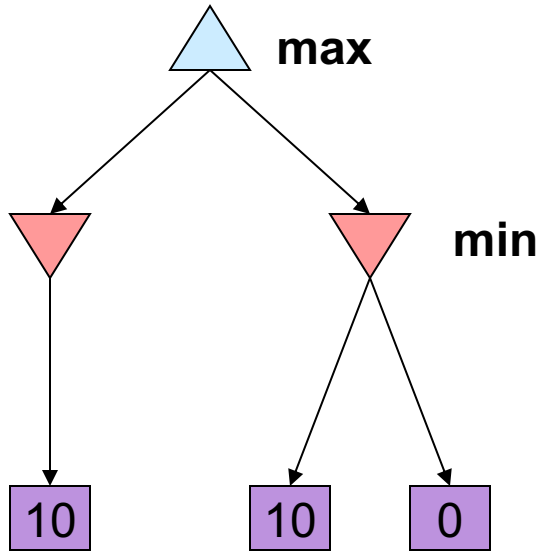
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Quiz



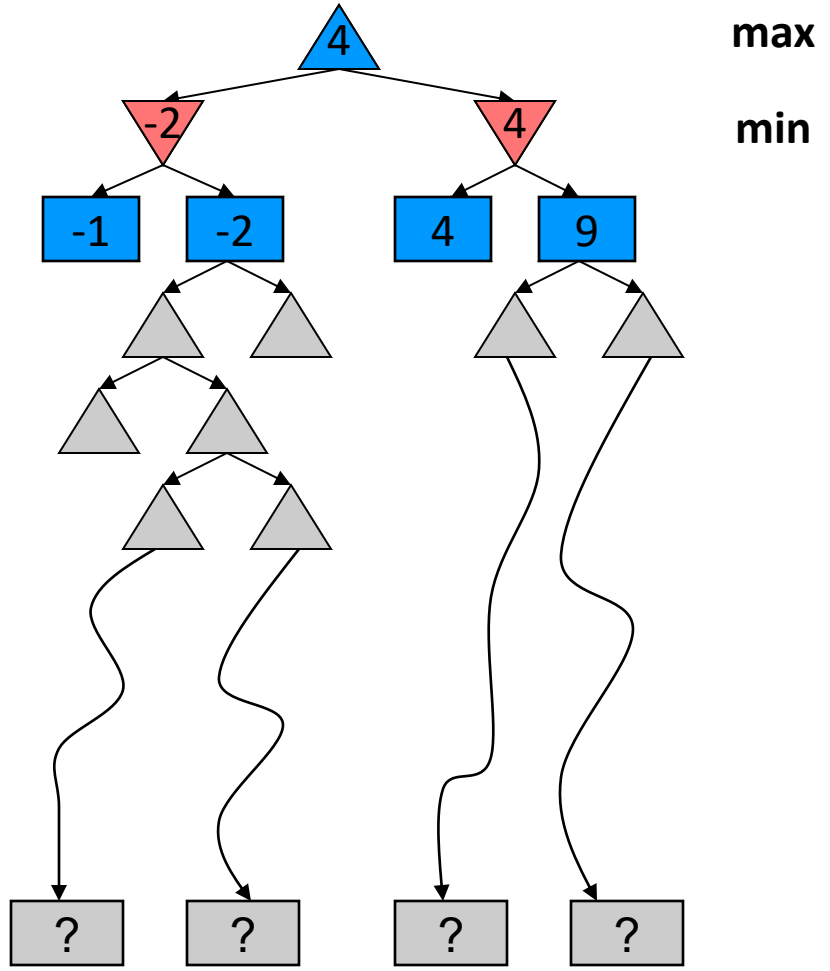
Alpha-Beta Pruning Properties

- 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!
  - Full search of, e.g. chess, is still hopeless…



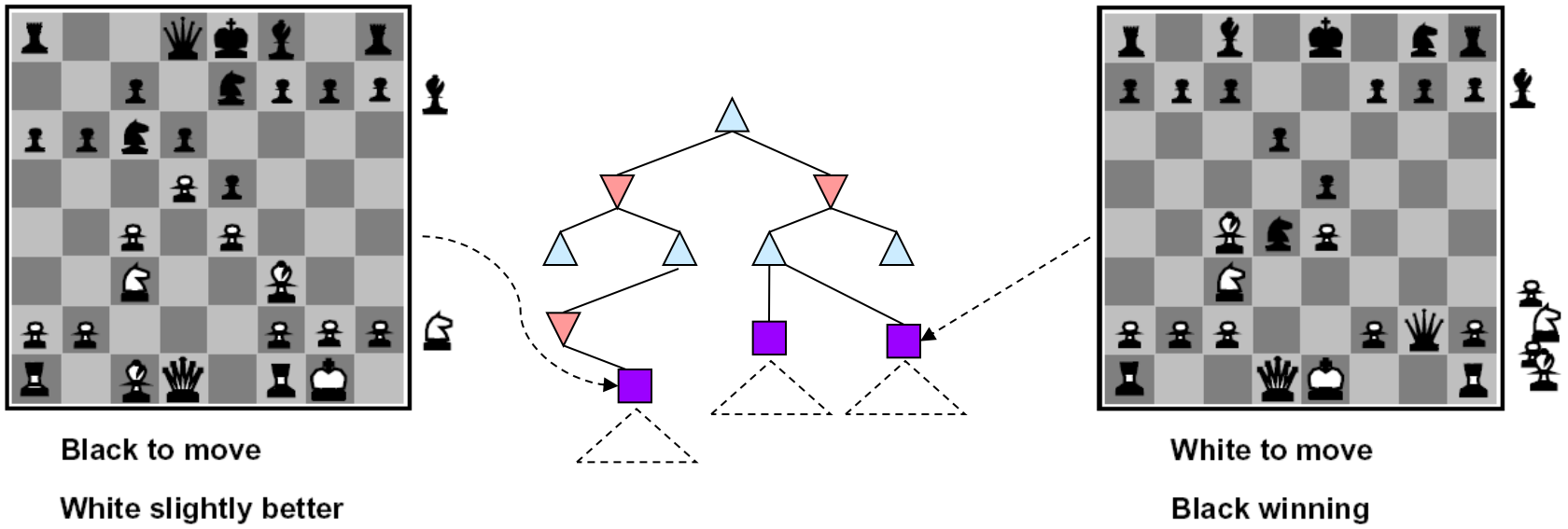
Resource Limits

- 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



Evaluation Functions

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

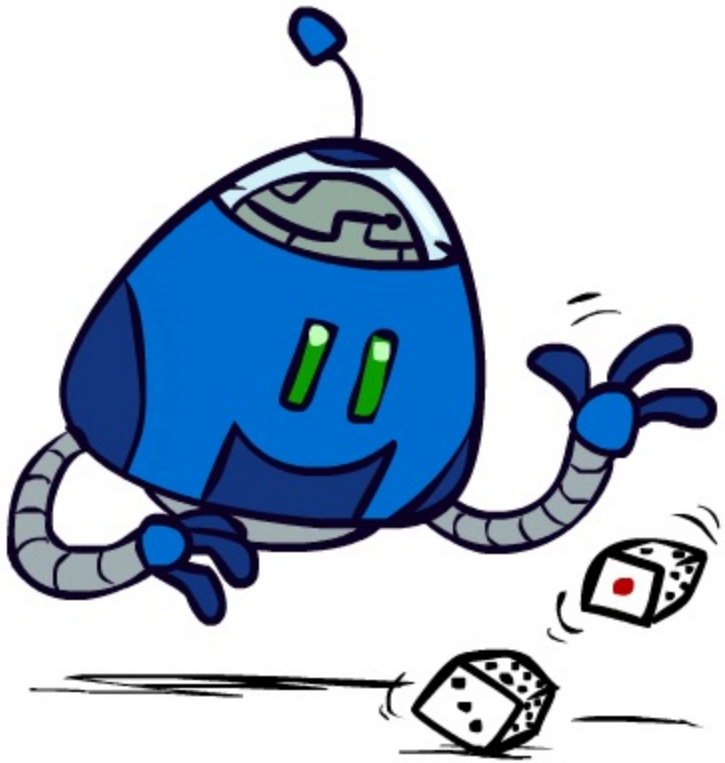


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

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

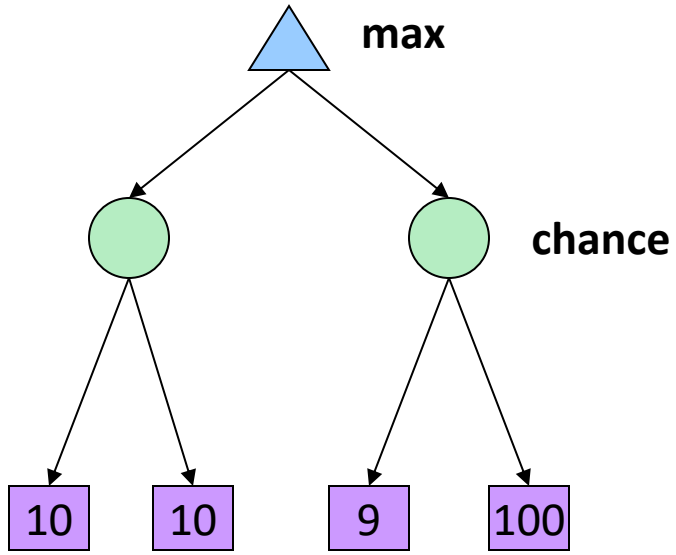
  - E.g. $f_1(s)$ = (num white queens – num black queens), etc.
- Or a more complex nonlinear function (e.g., NN) trained by self-play RL

Uncertain Outcomes



Expectimax Search

- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Actions can fail: when moving a robot, wheels might slip
- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes
- Expectimax search: compute the average score under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but the outcome is uncertain
  - Calculate their expected utilities
  - I.e. take weighted average (expectation) of children
- Later, we'll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes



[Demo: min vs exp (L7D1,2)]

Expectimax Pseudocode

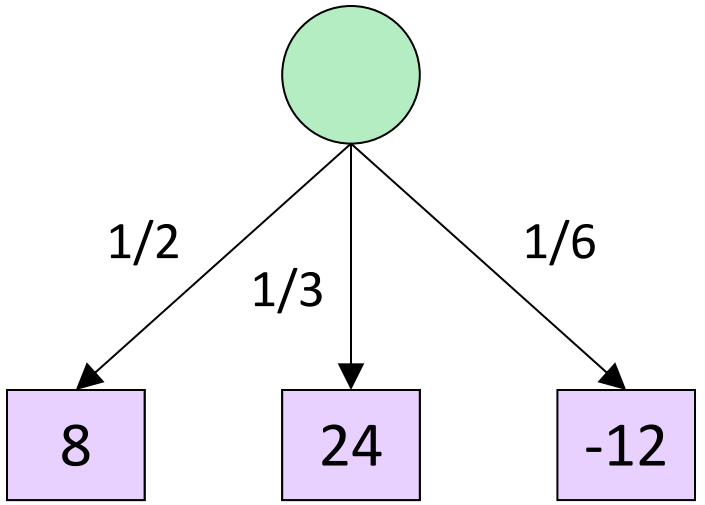
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

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

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

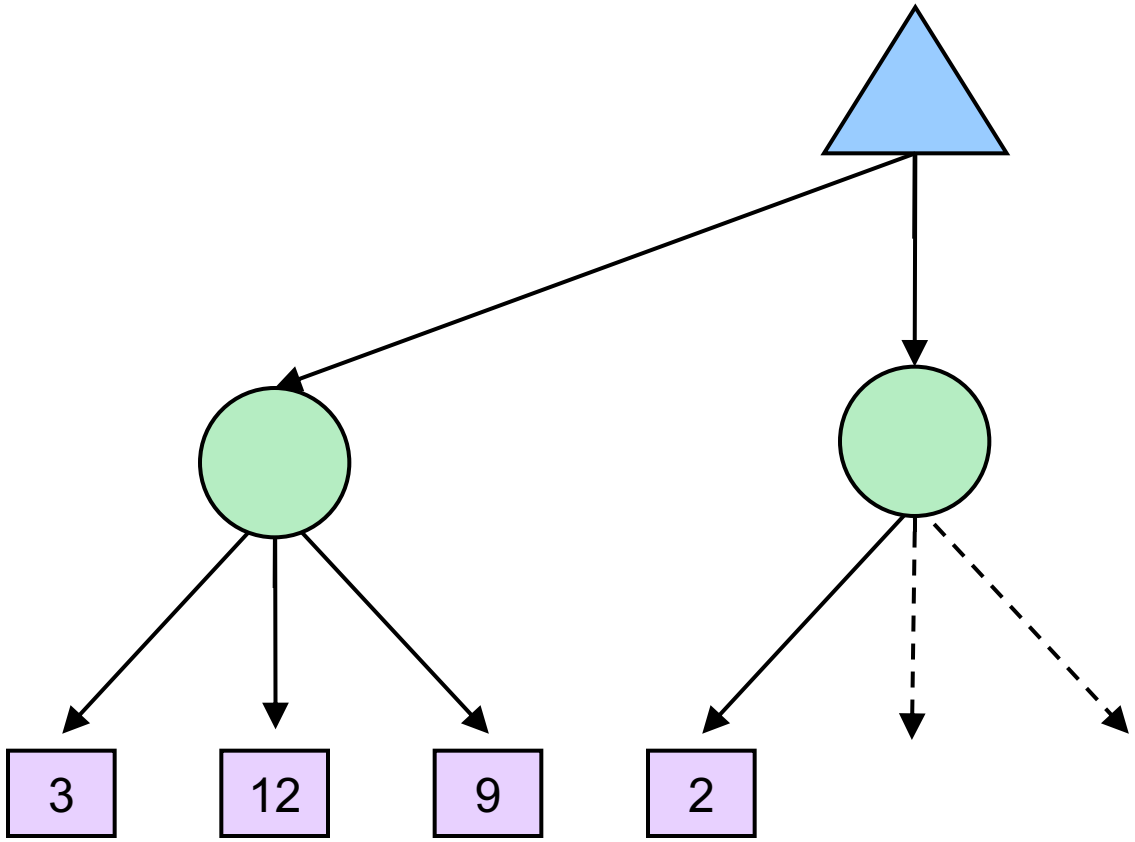
Expectimax Pseudocode

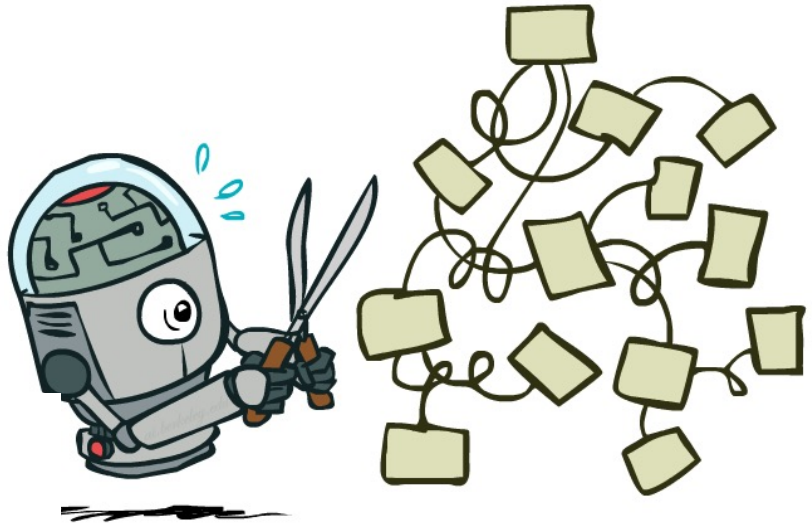
```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```



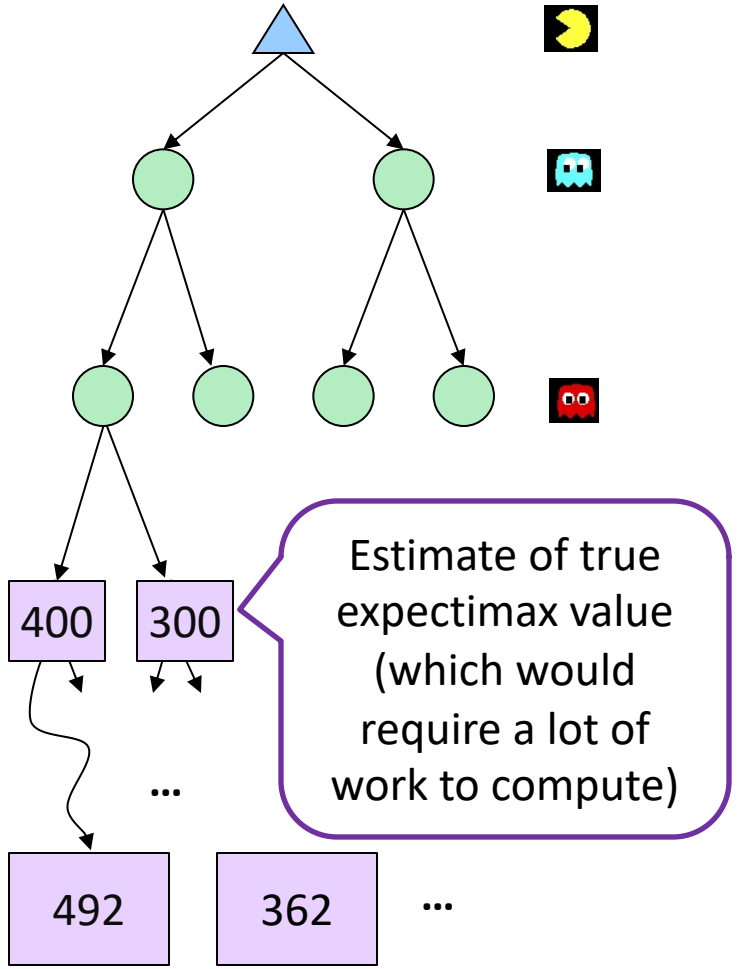
$$v = (1/2)\,(8) + (1/3)\,(24) + (1/6)\,(-12) = 10$$

Expectimax Pruning?





Depth-Limited Expectimax



南方科技大学

STA303: Artificial Intelligence

Markov Decision Processes

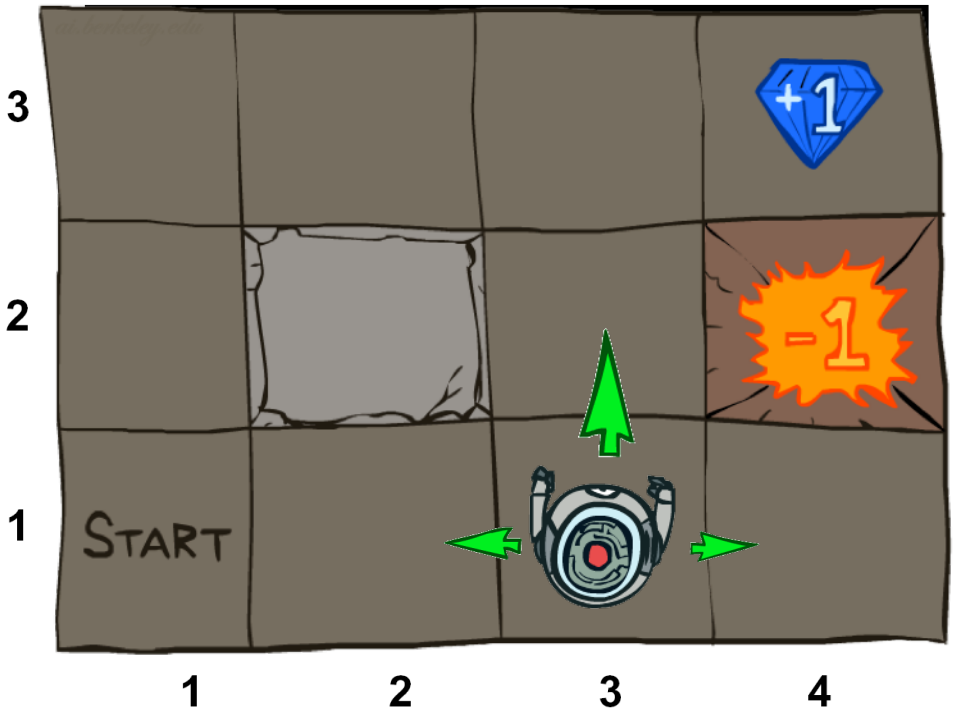
Fang Kong

https://fangkongx.github.io/

Slide credits: ai.berkeley.edu

Markov Decision Processes

- An MDP is defined by:
  - A set of states $s \in S$
  - A set of actions $a \in A$
  - A transition function $T(s, a, s')$
    - Probability that a from s leads to s', i.e., $P(s' | s, a)$
    - Also called the model or the dynamics
  - A reward function $R(s, a, s')$
    - Sometimes just $R(s)$ or $R(s')$
  - A start state
  - Maybe a terminal state



[Demo – gridworld manual intro (L8D1)]

What is Markov about MDPs?

- "Markov" generally means that given the present state, the future and the past are independent

- For Markov decision processes, "Markov" means action outcomes depend only on the current state

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots S_0 = s_0)$$

$$=$$

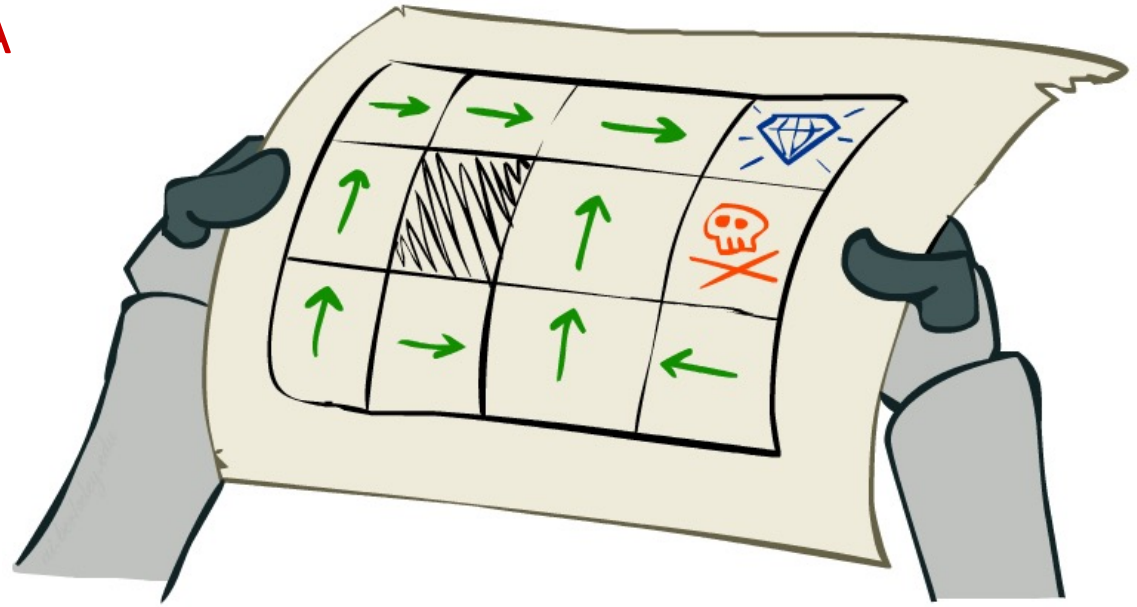
$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

- This is just like search, where the successor function could only depend on the current state (not the history)

Andrey Markov
(1856-1922)

Policies

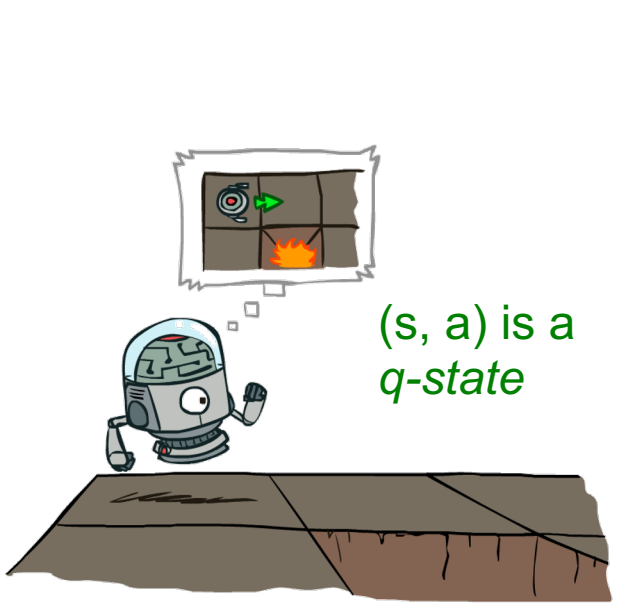
- For MDPs, we want an optimal policy $\pi^*: S \rightarrow A$
  - A policy π gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed

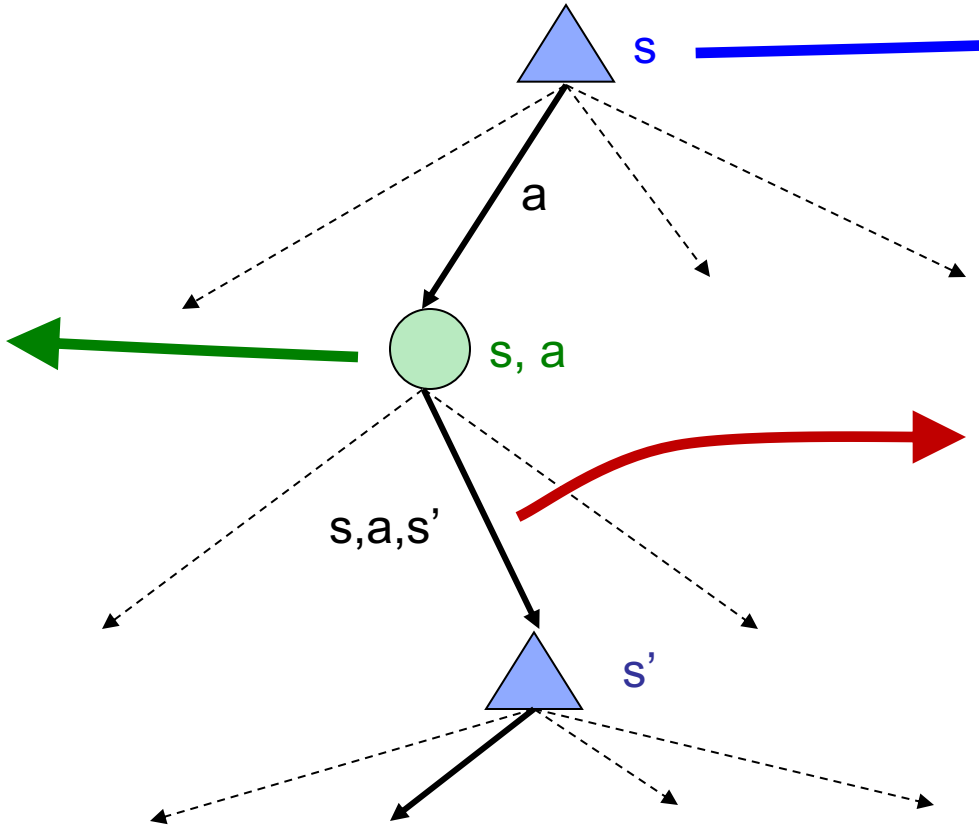


Optimal policy when R(s, a, s') = -0.03
for all non-terminals s

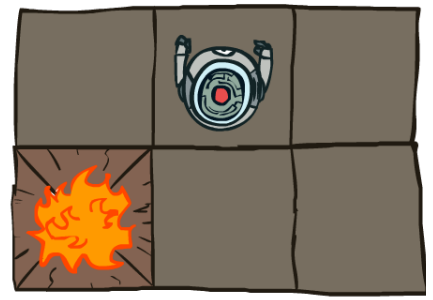
MDP Search Trees

- Each MDP state projects an expectimax-like search tree





s is a *state*



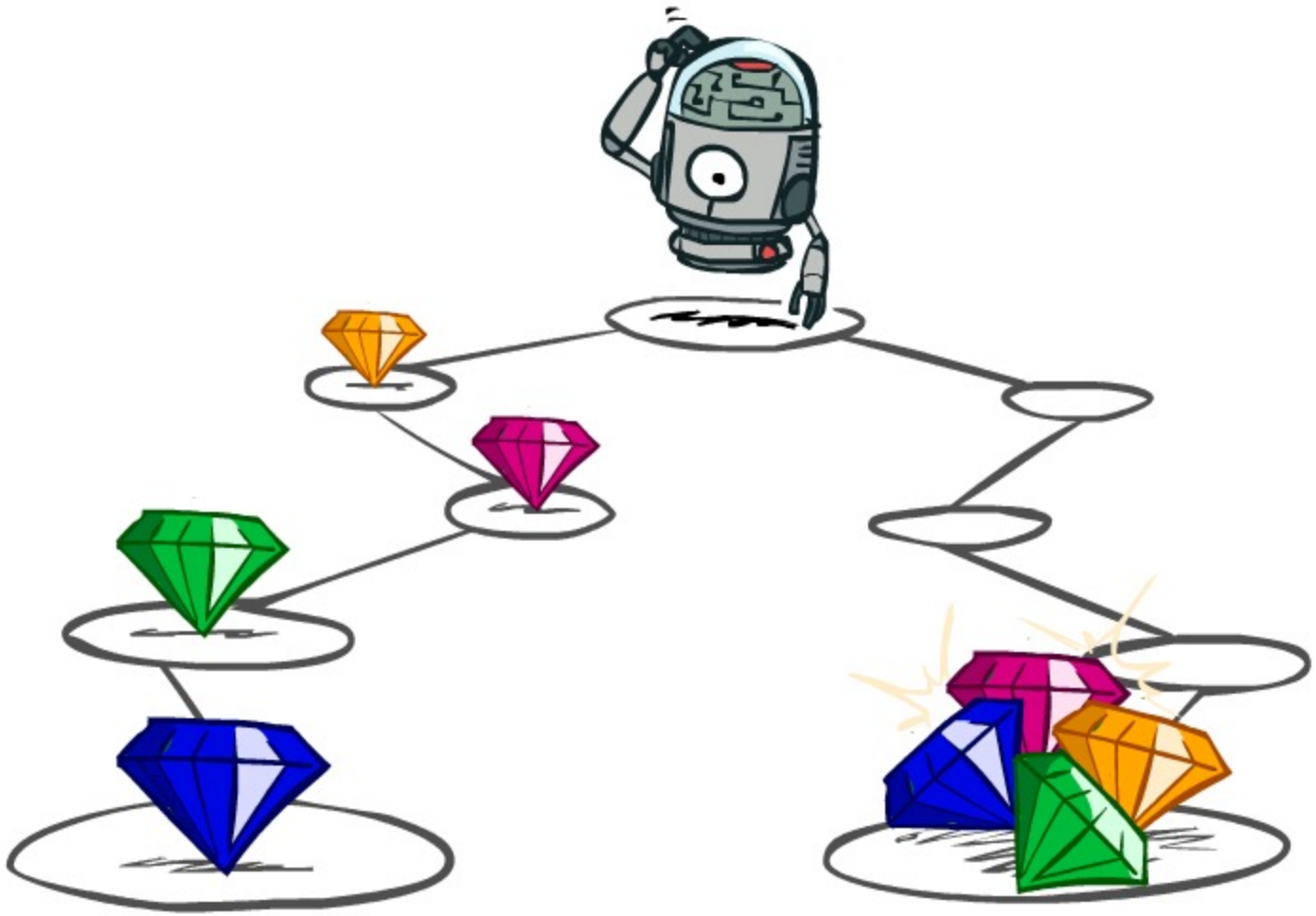
(s, a) is a *q-state*

(s,a,s') called a *transition*

T(s,a,s') = P(s'|s,a)

R(s,a,s')

Utilities of Sequences



Discounting

- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially

1

Worth Now

γ

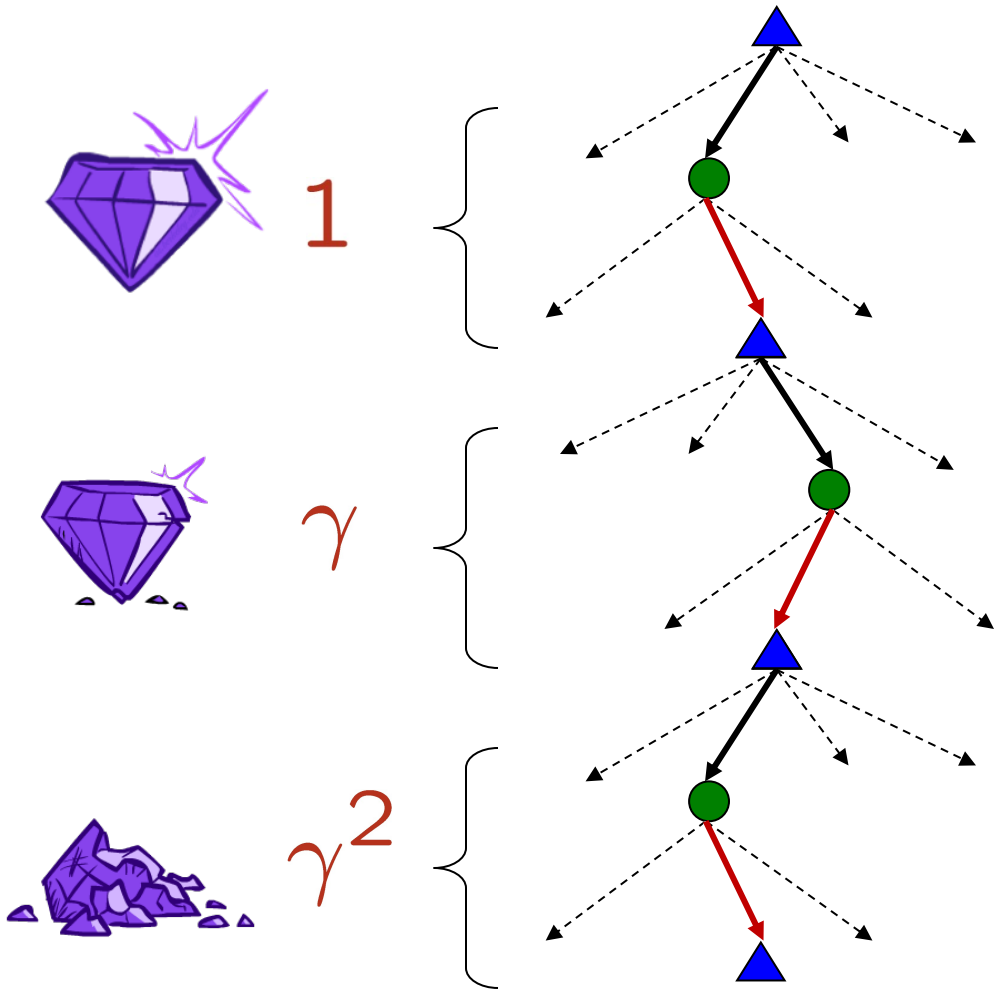
Worth Next Step

γ^2

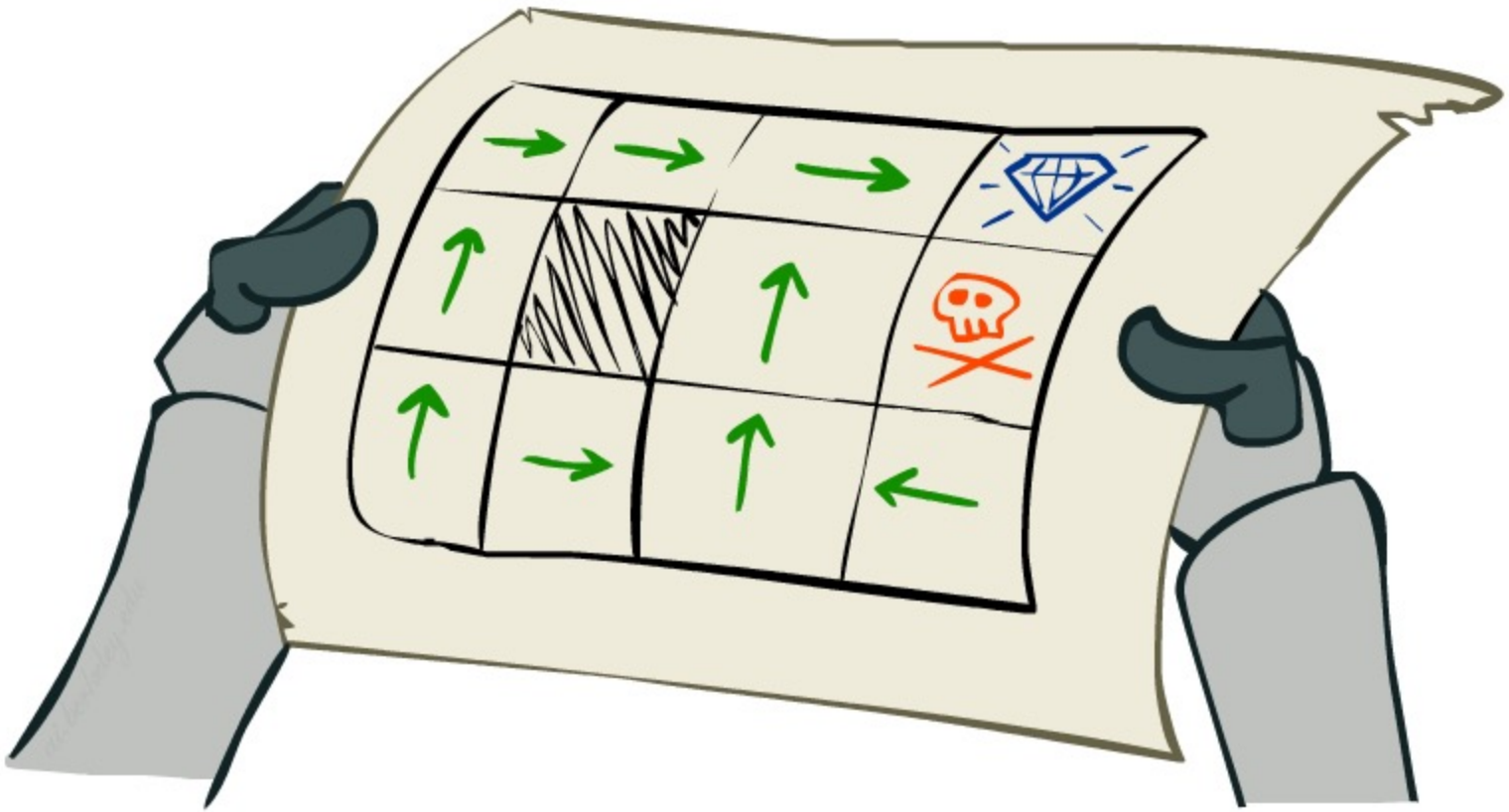
Worth In Two Steps

Visualizing Discounting

- How to discount?
  - Each time we descend a level, we multiply in the discount once
- Why discount?
  - Sooner rewards probably do have higher utility than later rewards
  - Also helps our algorithms converge
- Example: discount of 0.5
  - U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - U([1,2,3]) < U([3,2,1])



Solving MDPs



Optimal Quantities

- The value (utility) of a state s:

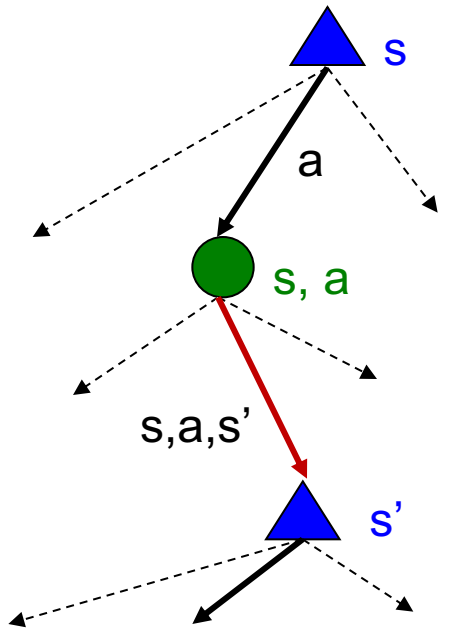
  $V^*(s)$ = expected utility starting in s and acting optimally

- The value (utility) of a q-state (s,a):

  $Q^*(s,a)$ = expected utility starting out having taken action a from state s and (thereafter) acting optimally

- The optimal policy:

  $\pi^*(s)$ = optimal action from state s



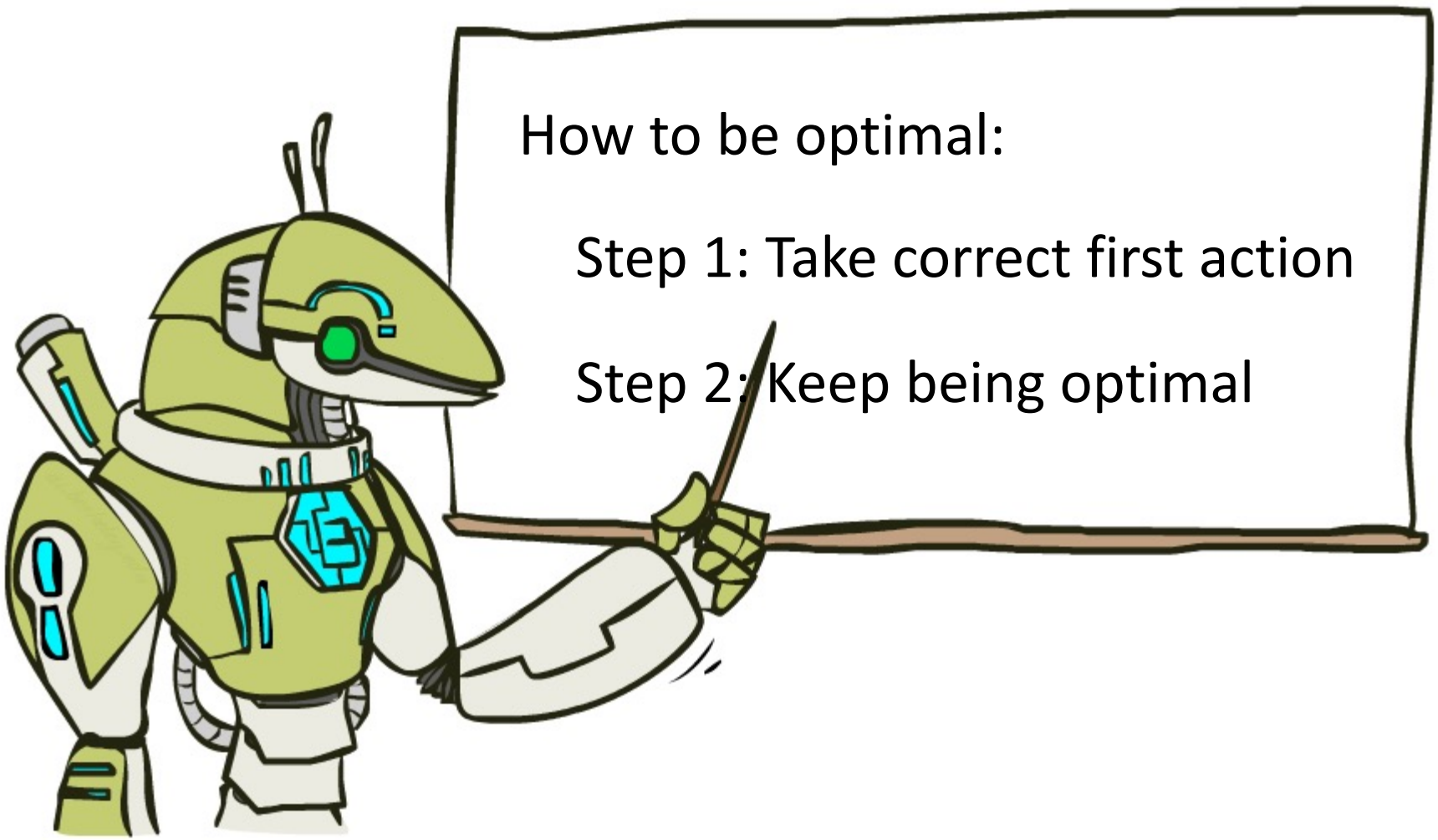
s is a *state*

(s, a) is a *q-state*

(s,a,s') is a *transition*

[Demo: gridworld values (L9D1)]

The Bellman Equations



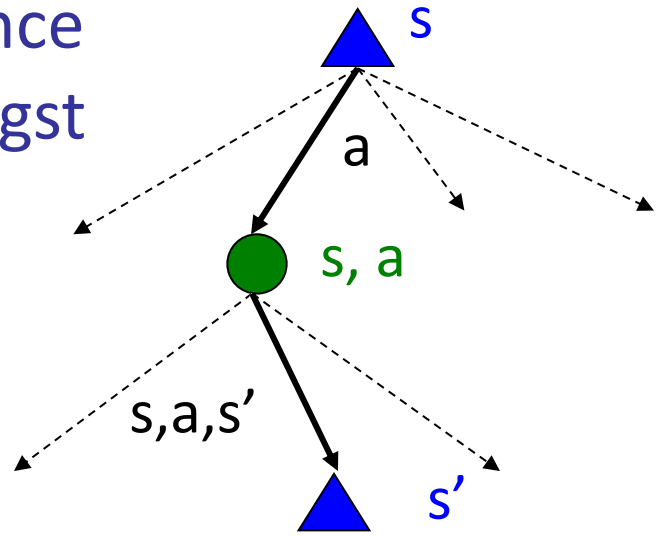
The Bellman Equations

- Definition of "optimal utility" via expectimax recurrence gives a simple one-step lookahead relationship amongst optimal utility values

$$V^*(s) = \max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

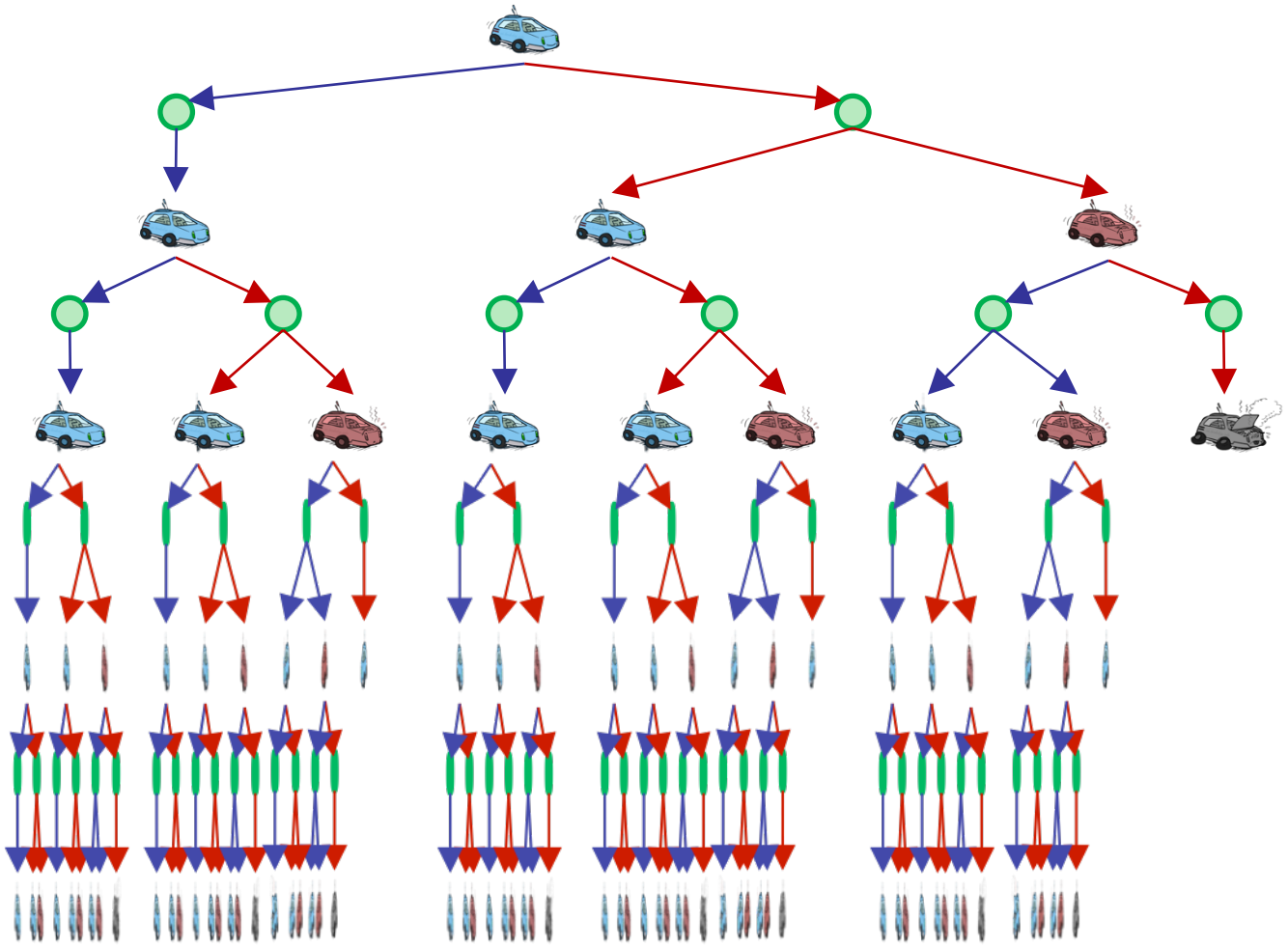
$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$



- These are the Bellman equations, and they characterize optimal values in a way we'll use over and over

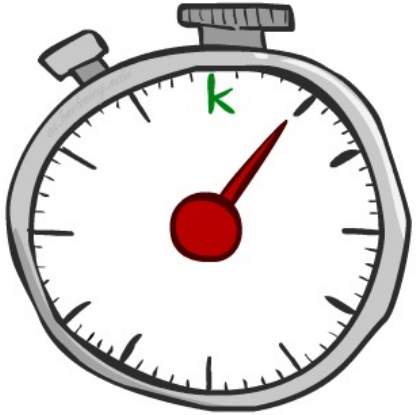
Racing Search Tree

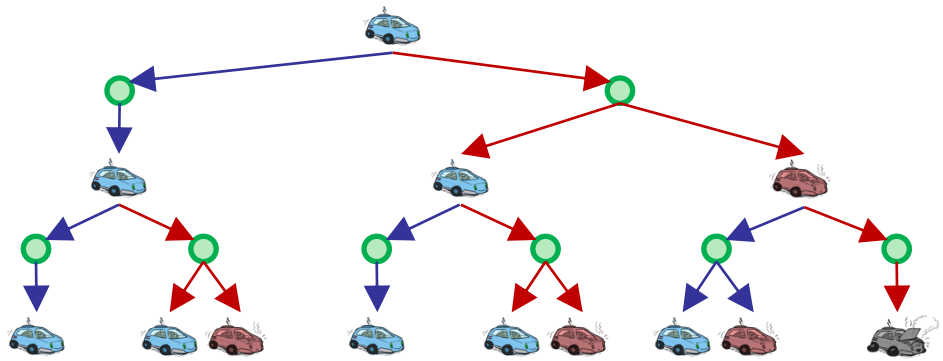
- Problem: States are repeated
  - Idea: Only compute needed quantities once
- Problem: Tree goes on forever
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: deep parts of the tree eventually don't matter if $\gamma < 1$

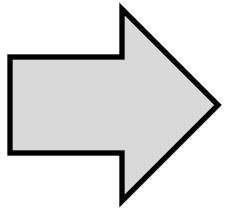


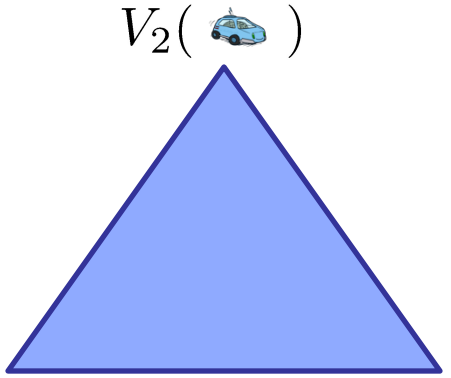
Time-Limited Values

- Key idea: time-limited values
- Define $V_k(s)$ to be the optimal value of s if the game ends in k more time steps
  - Equivalently, it's what a depth-k expectimax would give from s



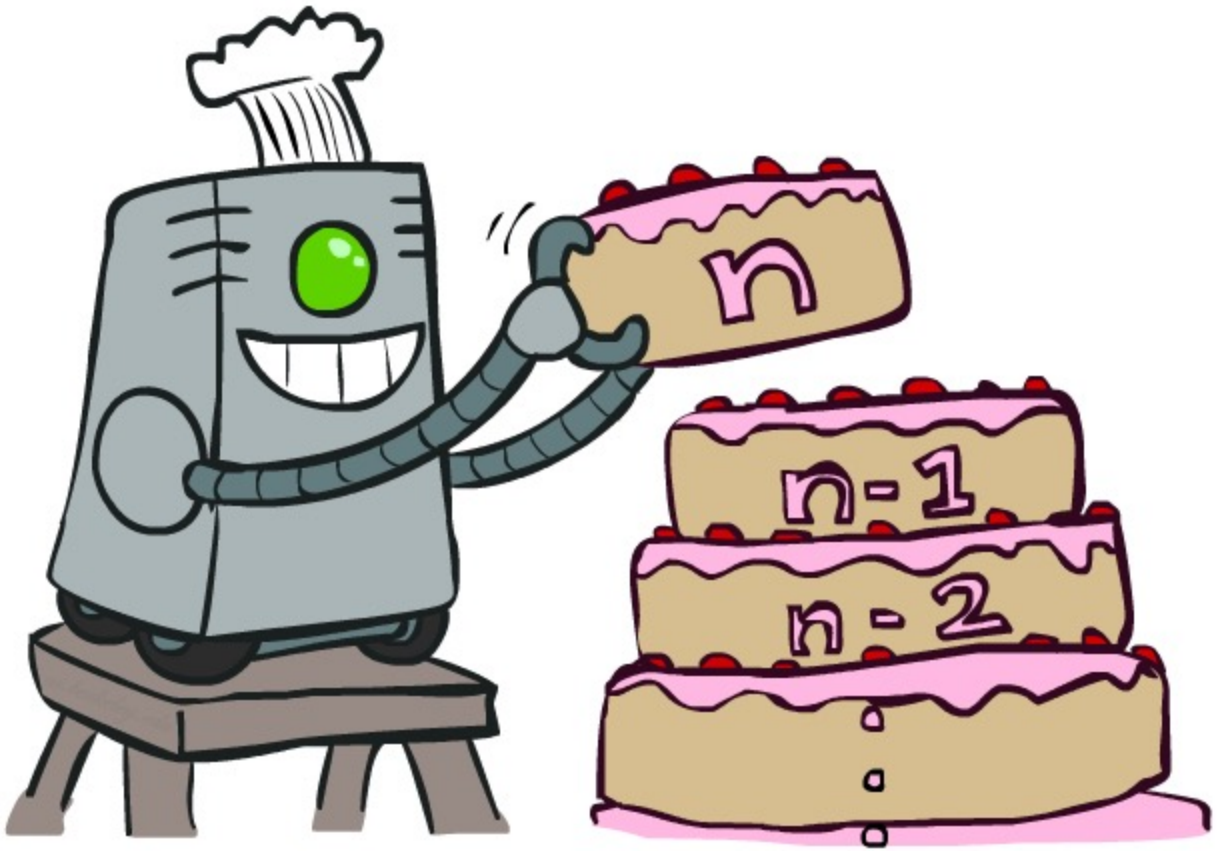






[Demo – time-limited values (L8D6)]

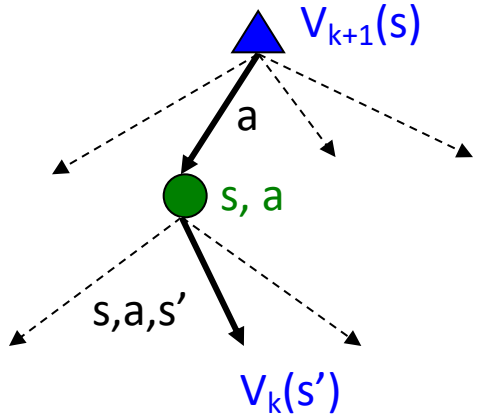
Value Iteration



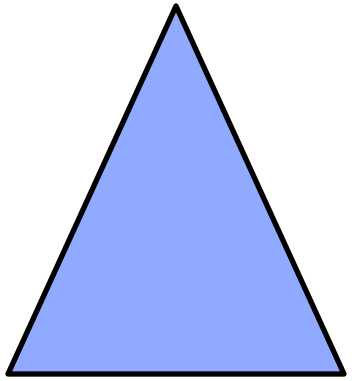
Value Iteration

- Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

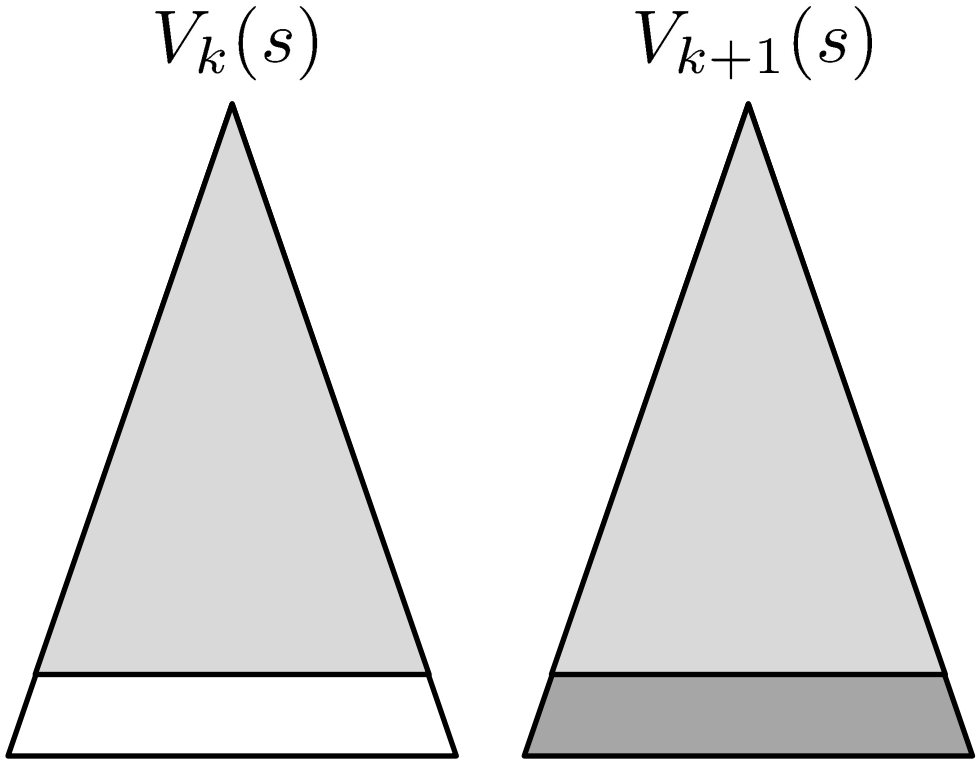


- Repeat until convergence
- Complexity of each iteration: $O(S^2A)$
- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do

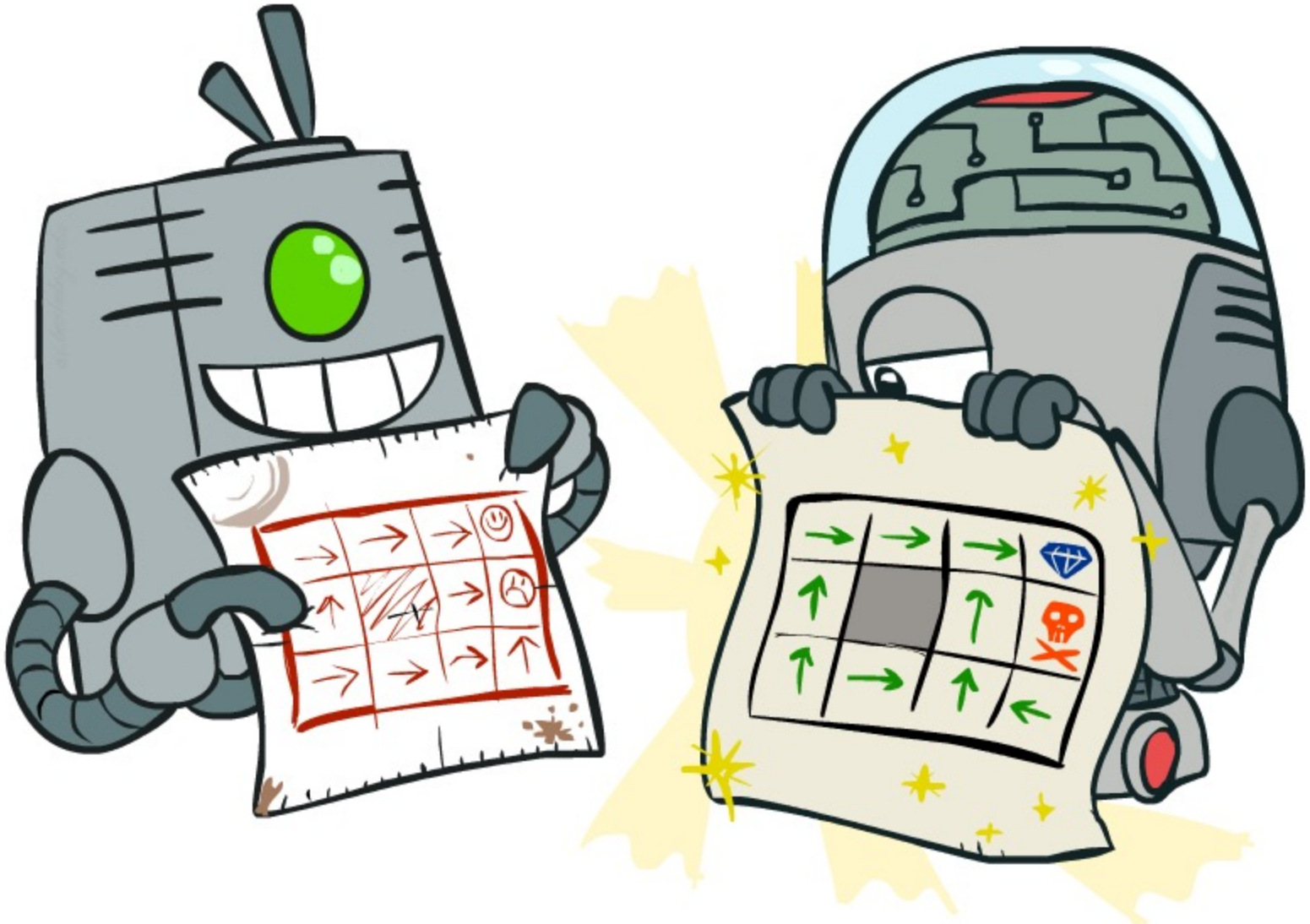


Convergence

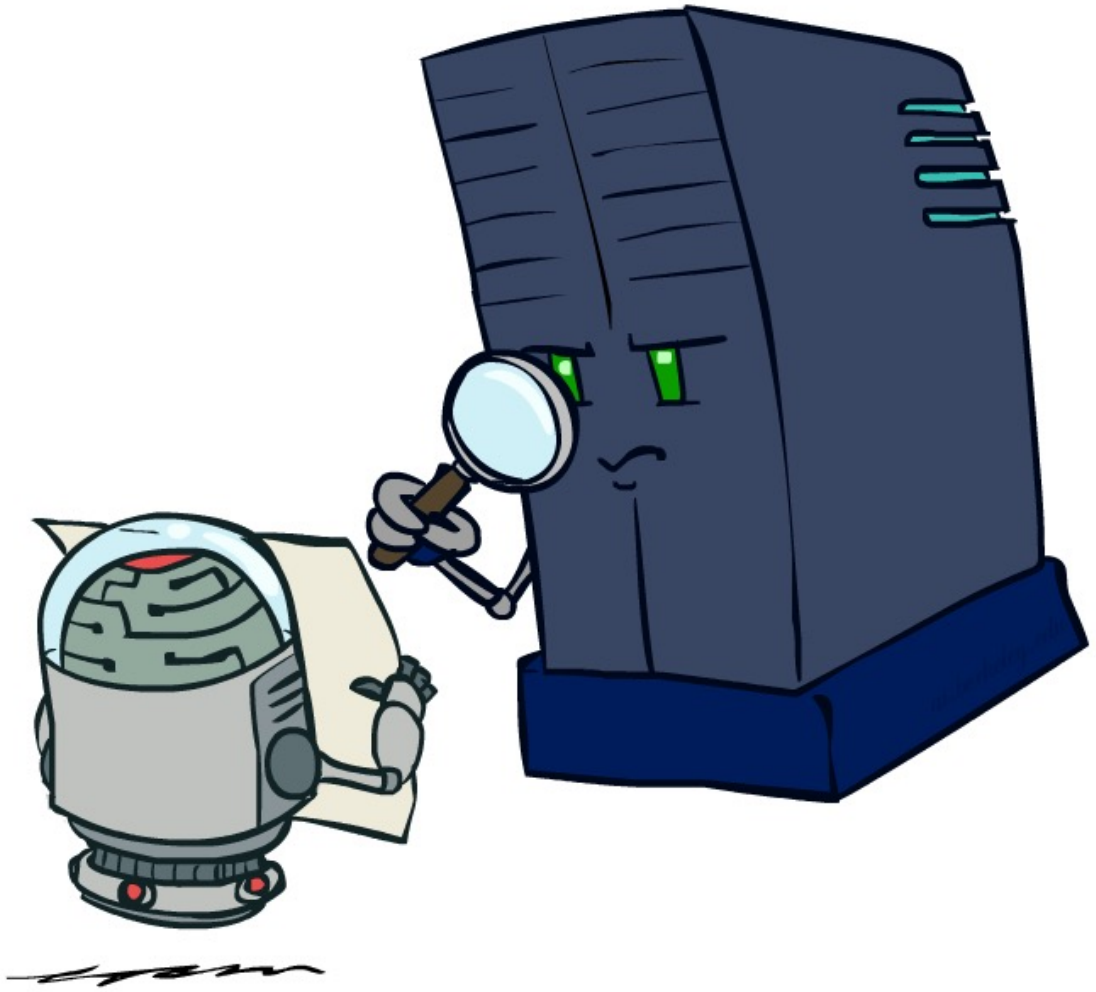
- How do we know the V_k vectors are going to converge?
- Case 1: If the tree has maximum depth M, then V_M holds the actual untruncated values
- Case 2: If the discount is less than 1
  - Sketch: For any state V_k and V_{k+1} can be viewed as depth k+1 expectimax results in nearly identical search trees
  - The difference is that on the bottom layer, V_{k+1} has actual rewards while V_k has zeros
  - That last layer is at best all R_{MAX}
  - It is at worst R_{MIN}
  - But everything is discounted by γ^k that far out
  - So V_k and V_{k+1} are at most γ^k max|R| different
  - So as k increases, the values converge



Policy Methods

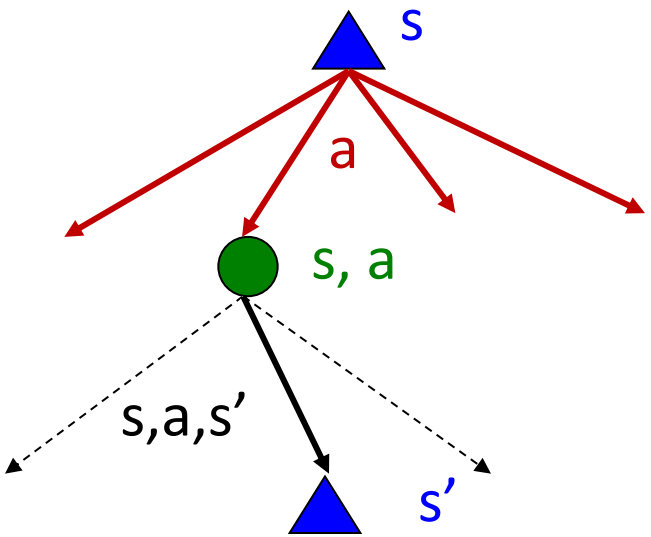


Policy Evaluation

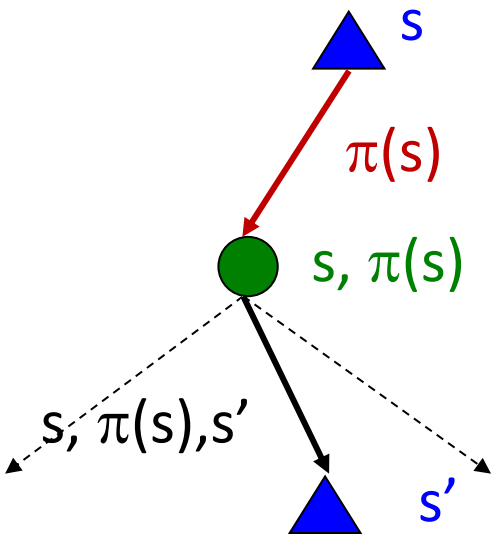


Fixed Policies

Do the optimal action



Do what π says to do



- Expectimax trees max over all actions to compute the optimal values
- If we fixed some policy $\pi(s)$, then the tree would be simpler – only one action per state
  - … though the tree's value would depend on which policy we fixed

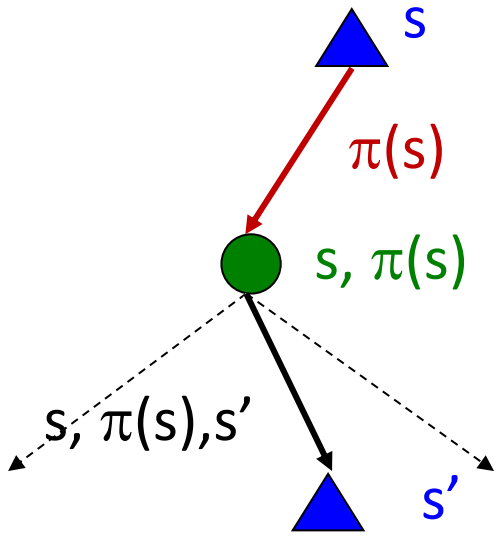
Utilities for a Fixed Policy

- Another basic operation: compute the utility of a state s under a fixed (generally non-optimal) policy

- Define the utility of a state s, under a fixed policy π:

  $V^{\pi}(s)$ = expected total discounted rewards starting in s and following π

- Recursive relation (one-step look-ahead / Bellman equation):



$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

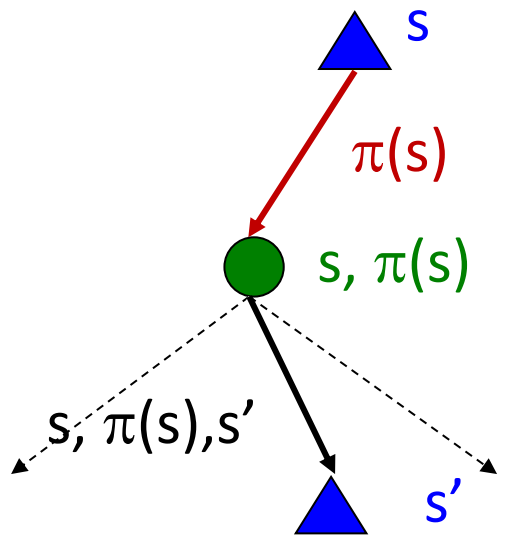
Policy Evaluation

- How do we calculate the V's for a fixed policy π?

- Idea 1: Turn recursive Bellman equations into updates (like value iteration)

$$V_0^\pi(s) = 0$$

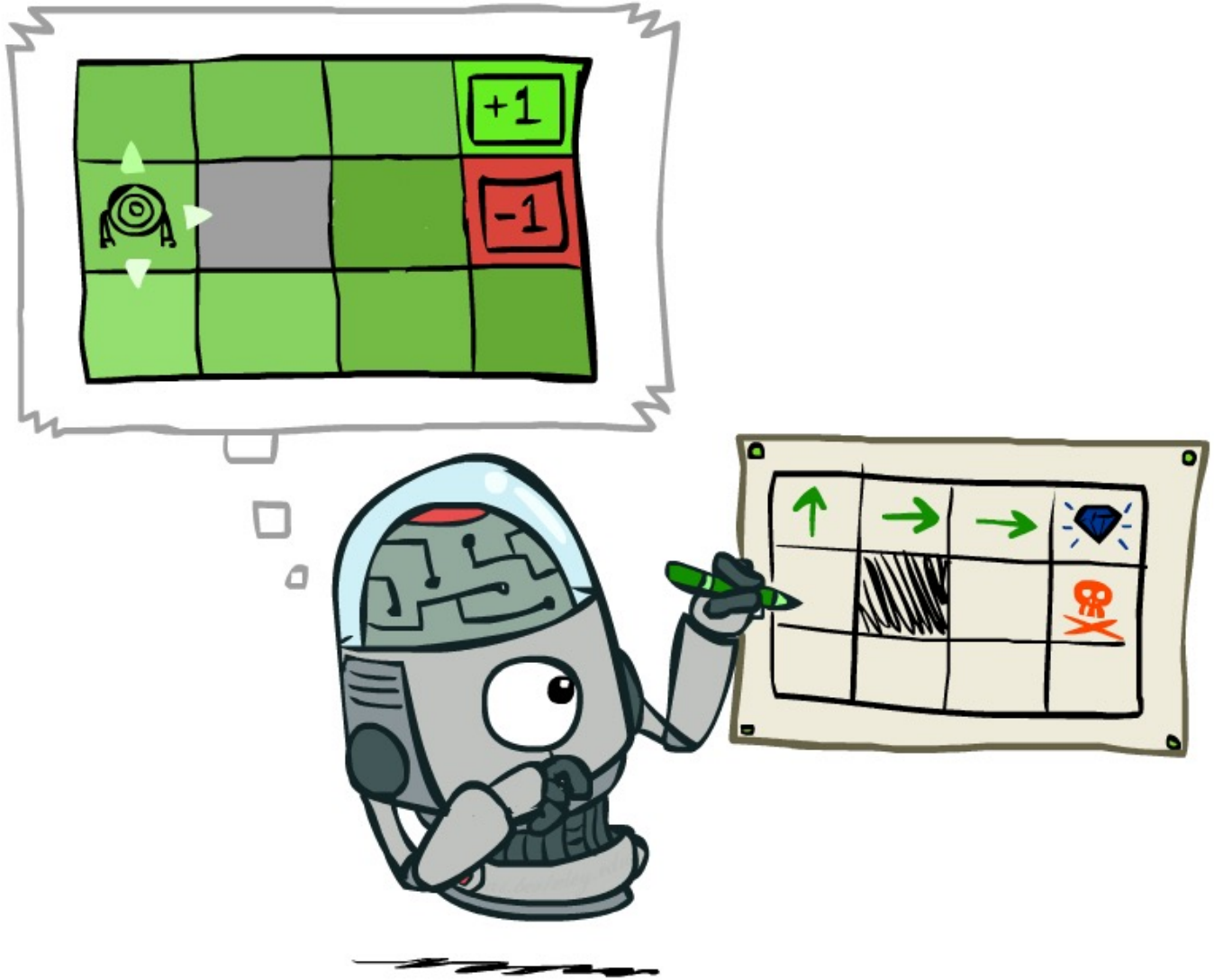
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$



- Efficiency: $O(S^2)$ per iteration

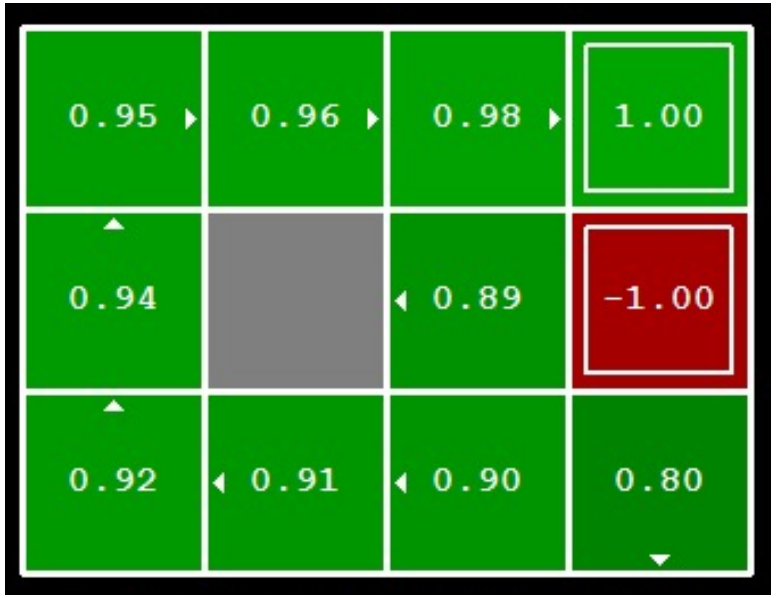
- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve with Matlab (or your favorite linear system solver)

Policy Extraction



Computing Actions from Values

- Let's imagine we have the optimal values V*(s)
- How should we act?
  - It's not obvious!
- We need to do a mini-expectimax (one step)

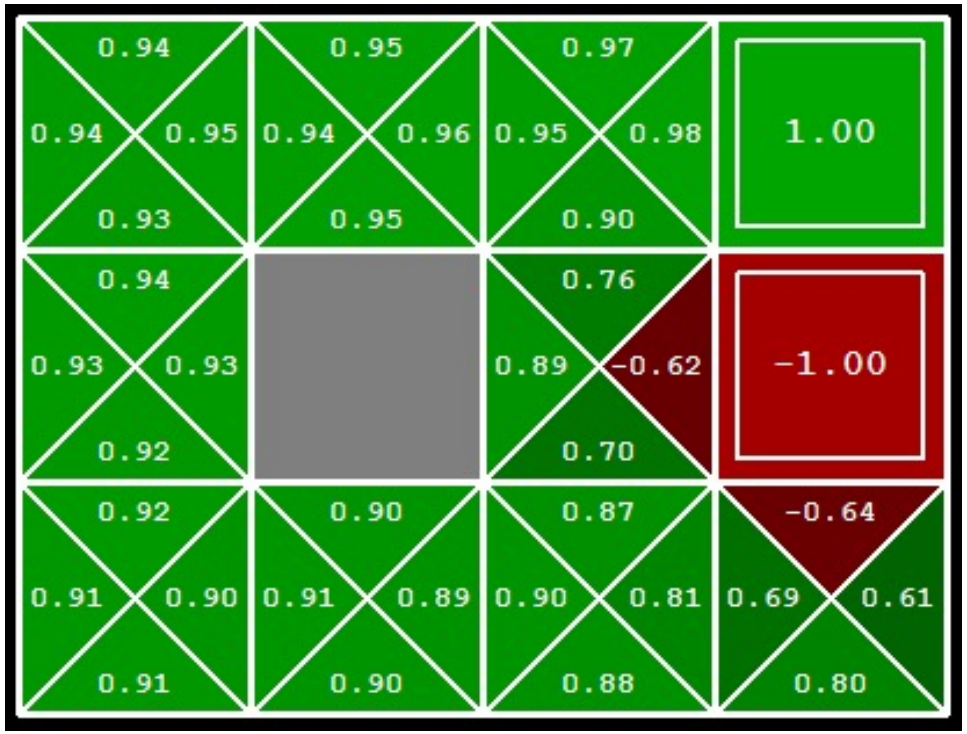


$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

- This is called policy extraction, since it gets the policy implied by the values

Computing Actions from Q-Values

- Let's imagine we have the optimal q-values:

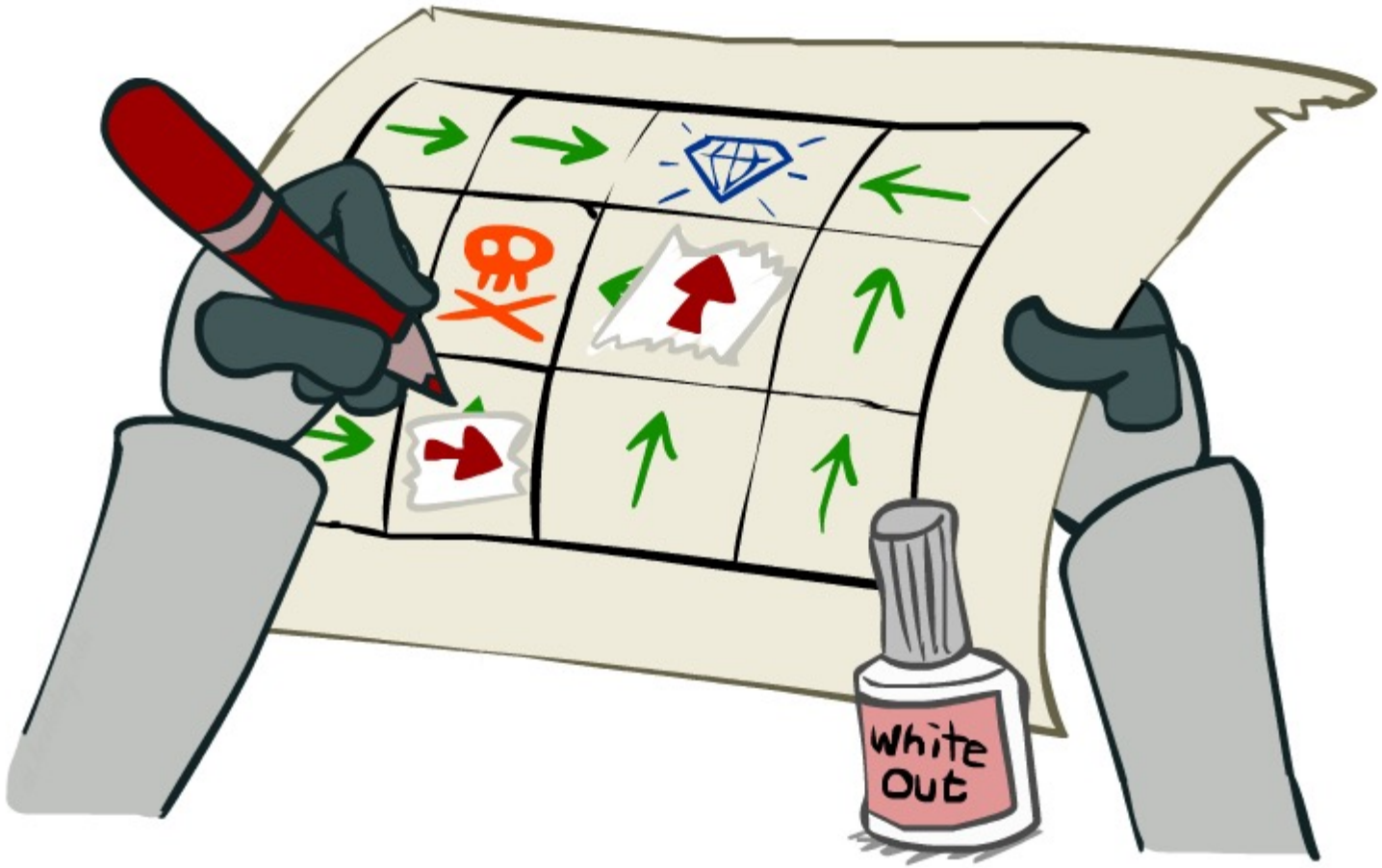


- How should we act?
  - Completely trivial to decide!

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

- Important lesson: actions are easier to select from q-values than values!

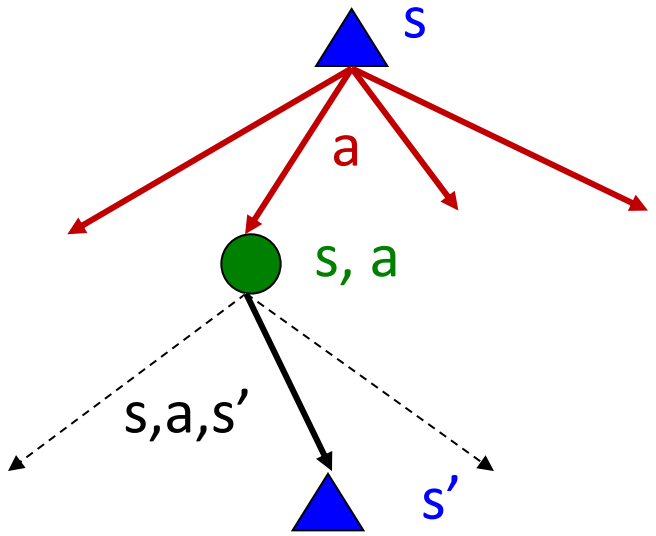
Policy Iteration



Problems with Value Iteration

- Value iteration repeats the Bellman updates:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$



- Problem 1: It's slow – $O(S^2A)$ per iteration
- Problem 2: The "arg max" at each state rarely changes
- Problem 3: The policy often converges long before the values

[Demo: value iteration (L9D2)]

Policy Iteration

- Alternative approach for optimal values:
  - Step 1: Policy evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - Step 2: Policy improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges

- This is policy iteration
  - It's still optimal!
  - Can converge (much) faster under some conditions

Policy Iteration (PI)

- Evaluation: For fixed current policy π, find values with policy evaluation:
  - Iterate until values converge:

$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s') \right]$$

- Improvement: For fixed values, get a better policy using policy extraction
  - One-step look-ahead:

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi_i}(s') \right]$$

Convergence of PI

- 1. Improvement: Does each policy improvement step produce a better policy?
- 2. Convergence: Does PI converge to an optimal policy?

Comparison

- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don't track the policy, but taking the max over actions implicitly recomputes it
- In policy iteration:
  - We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
  - After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
  - The new policy will be better (or we're done)
- Both are dynamic programs for solving MDPs

南方科技大学

STA303: Artificial Intelligence

Reinforcement Learning

Fang Kong

https://fangkongx.github.io/

Slide credits: ai.berkeley.edu

Reinforcement Learning

- Still assume a Markov decision process (MDP):
  - A set of states $s \in S$
  - A set of actions (per state) $A(s)$
  - A transition model $T(s,a,s')$
  - A reward function $R(s,a,s')$
- Still looking for a policy $\pi(s)$


Cool


Warm

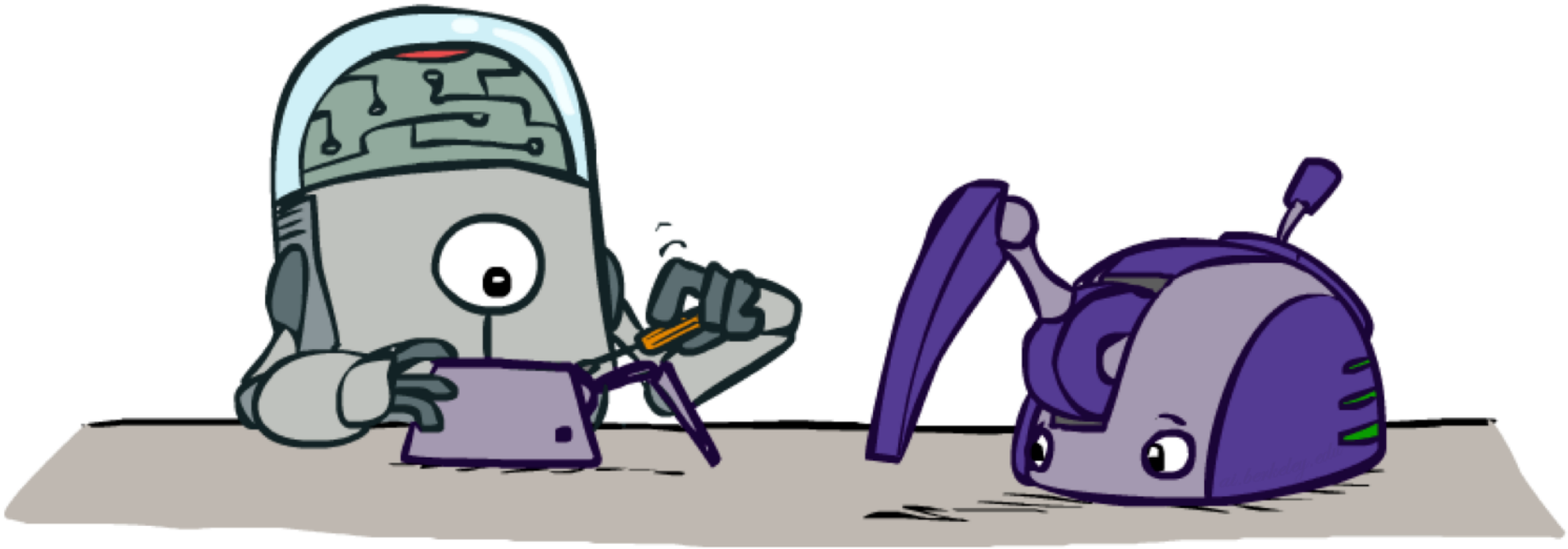

Overheated

- New twist: don't know T or R
  - I.e. we don't know which states are good or what the actions do
  - Must explore new states and actions to discover how the world works

Approaches to reinforcement learning

1. Model-based: Learn the model, solve it, execute the solution
2. Learn values from experiences, use to make decisions
    a. Direct evaluation
    b. Temporal difference learning
    c. Q-learning
3. Optimize the policy directly

Model-Based RL



Model-Based Learning

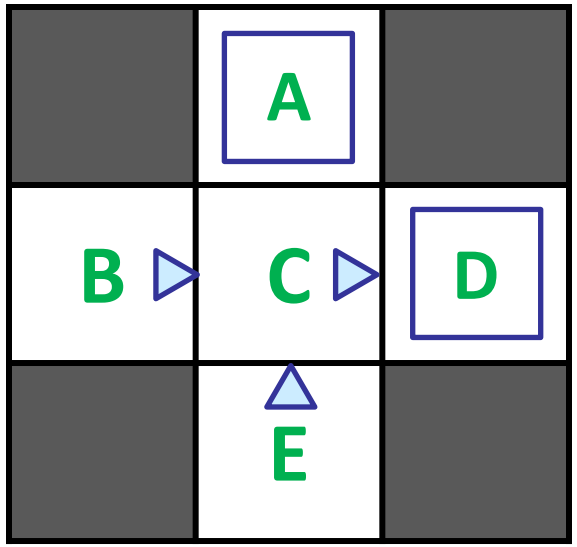
- Model-Based Idea:
  - Learn an approximate model based on experiences
  - Solve for values as if the learned model were correct

- Step 1: Learn empirical MDP model
  - Count outcomes s' for each s, a
  - Directly estimate each entry in $T(s,a,s')$ from counts
  - Discover each $R(s,a,s')$ when we experience the transition

- Step 2: Solve the learned MDP
  - Use, e.g., value or policy iteration, as before

Example: Model-Based Learning

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Learned Model

$T(s,a,s')$

T(B, east, C) = 1.00
P(C, east, D) = 0.75
P(C, east, A) = 0.25
…

$R(s,a,s')$

R(B, east, C) = -1
R(C, east, D) = -1
R(D, exit, x) = +10
…

Pros and cons

- Pro:
  - Makes efficient use of experiences (low ***sample complexity***)
- Con:
  - May not scale to large state spaces
    - Solving MDP is intractable for very large $|S|$
  - RL feedback loop tends to magnify small model errors
  - Much harder when the environment is partially observable

Basic idea of model-free methods

- To approximate expectations with respect to a distribution, you can either
  - Estimate the distribution from samples, compute an expectation
  - Or, bypass the distribution and estimate the expectation from samples directly

Example: Expected Age

Goal: Compute expected age of STA303 students

Known P(A)

$$E[A] = \sum_a P(a) \cdot a = 0.35 \times 20 + \ldots$$

Without P(A), instead collect samples $[a_1, a_2, \ldots a_N]$

"Model Based": estimate P(A):

$$\hat{P}(A=a) = N_a/N$$

$$E[A] \approx \sum_a \hat{P}(a) \cdot a$$

Why does this work? Because eventually you learn the right model.

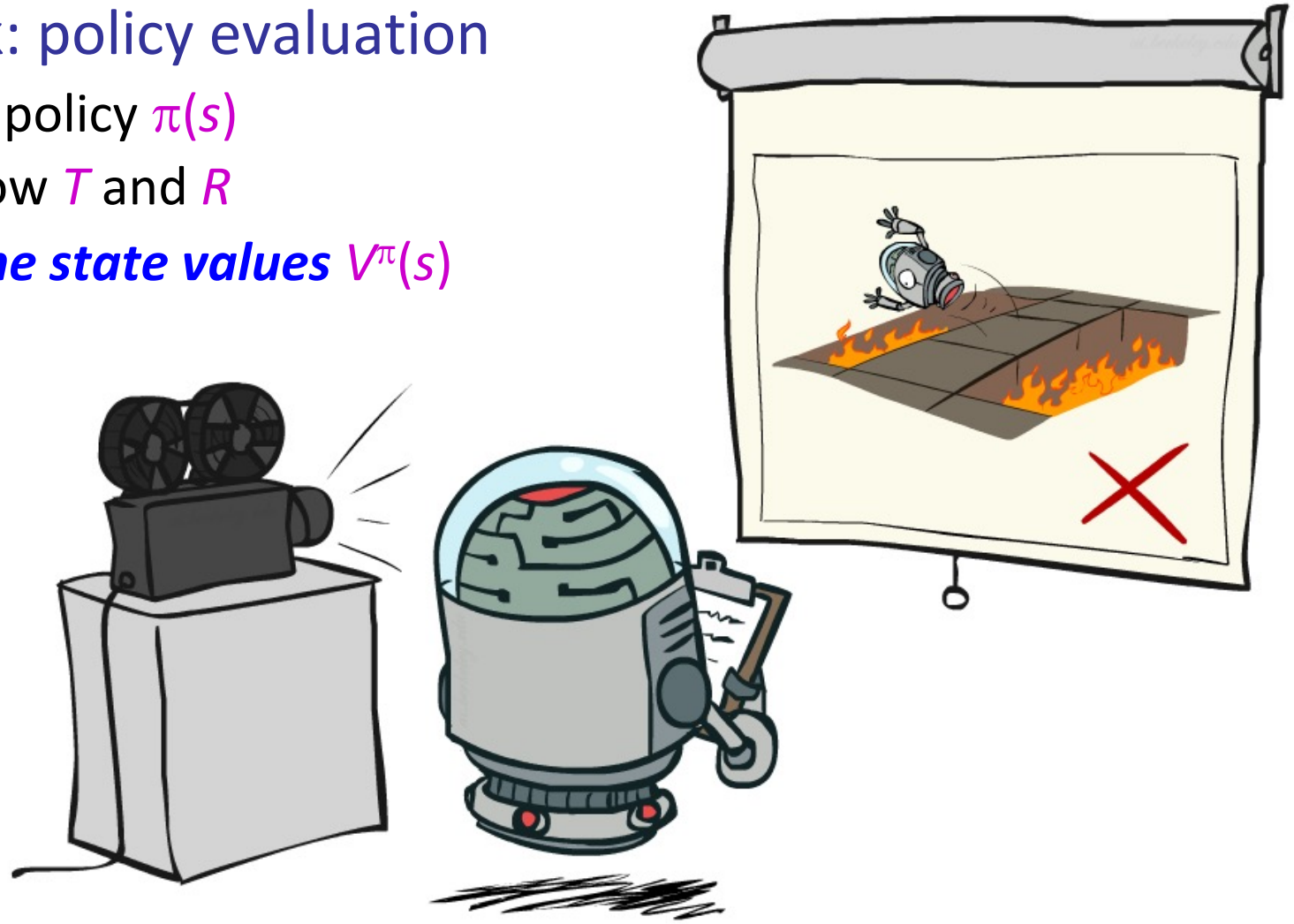
"Model Free": estimate expectation

$$E[A] \approx 1/N \sum_i a_i$$

Why does this work? Because samples appear with the right frequencies.

Passive Reinforcement Learning

- Simplified task: policy evaluation
  - Input: a fixed policy $\pi(s)$
  - You don't know T and R
  - ***Goal: learn the state values*** $V^{\pi}(s)$



Direct evaluation

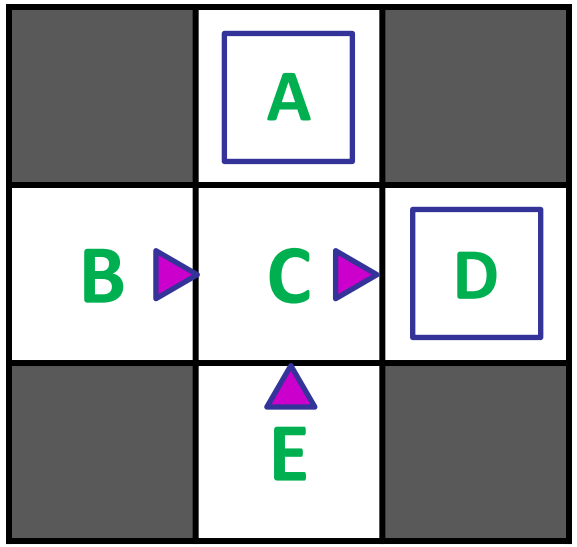
- Goal: Estimate $V^{\pi}(s)$, i.e., expected total discounted reward from s onwards

- Idea:
  - Use ***returns***, the *actual* sums of discounted rewards from s
  - Average over multiple trials and visits to s

- This is called ***direct evaluation*** (or direct utility estimation)

Example: Direct Estimation

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

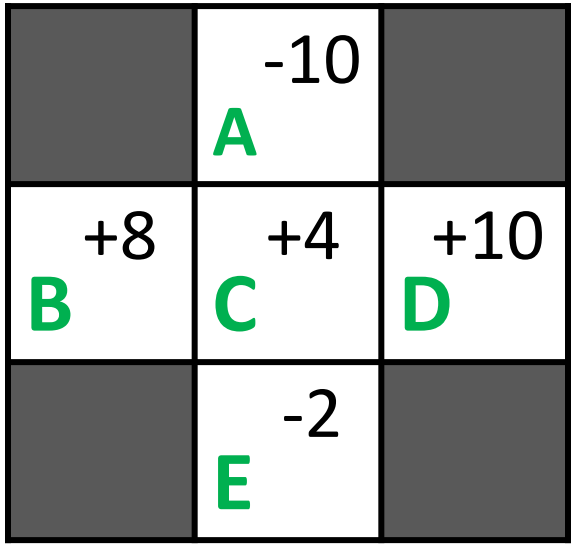
Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Output Values

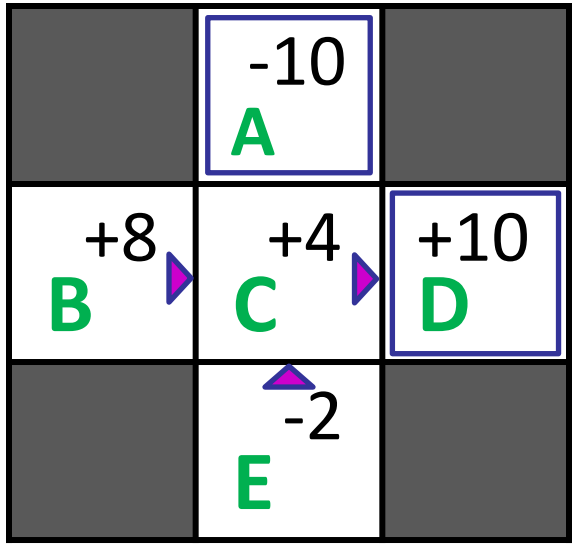


Problems with Direct Estimation

- What's good about direct estimation?
  - It's easy to understand
  - It doesn't require any knowledge of *T* and *R*
  - It converges to the right answer in the limit
- What's bad about it?
  - Each state must be learned separately (fixable)
  - It ***ignores information about state connections***
  - So, it takes a long time to learn

E.g., B=at home, study hard
E=at library, study hard
C=know material, go to exam

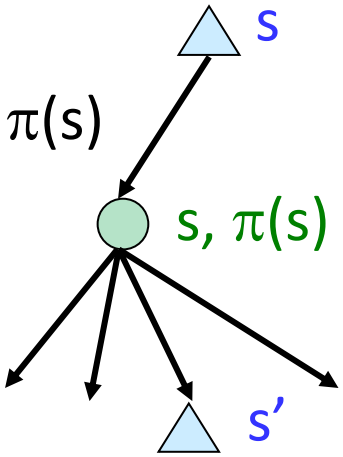
Output Values



If B and E both go to C under this policy, how can their values be different?

Temporal Difference Learning

- Big idea: learn from every experience!
  - Update V(s) each time we experience a transition (s, a, s', r)
  - Likely outcomes s' will contribute updates more often



- Temporal difference learning of values
  - Policy still fixed, still doing evaluation!
  - Move values toward value of whatever successor occurs: running average

Sample of V(s): $sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$

Update to V(s): $V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha)sample$

Same update: $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$

Running averages contd.

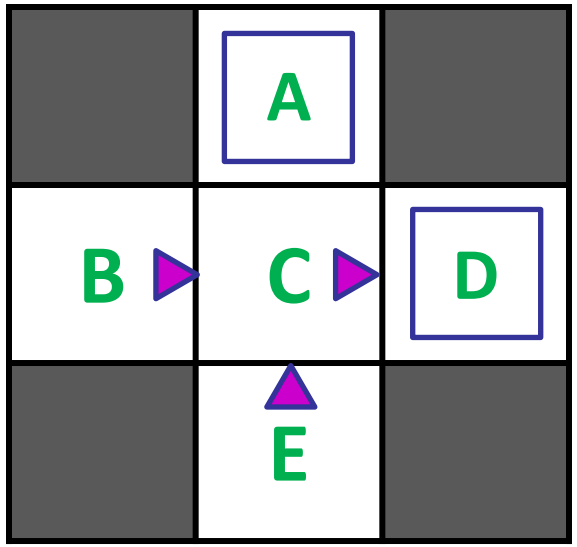
- What if we use a weighted average with a fixed weight?
  - $\mu_n = (1-\alpha)\,\mu_{n-1} + \alpha\, x_n$
  - n=1 $\mu_1 = x_1$
  - n=2 $\mu_2 = (1-\alpha)\cdot\mu_1 + \alpha x_2 = (1-\alpha)\cdot x_1 + \alpha x_2$
  - n=3 $\mu_3 = (1-\alpha)\cdot\mu_2 + \alpha x_3 = (1-\alpha)^2\cdot x_1 + \alpha(1-\alpha)x_2 + \alpha x_3$
  - n=4 $\mu_4 = (1-\alpha)\cdot\mu_3 + \alpha x_4 = (1-\alpha)^3\cdot x_1 + \alpha(1-\alpha)^2 x_2 + \alpha(1-\alpha)x_3 + \alpha x_4$
- I.e., ***exponential forgetting*** of old values
- μ_n is unbiased

TD as approximate Bellman update

- Idea 3: Update values by maintaining a ***running average***
  - sample = $R(s,\pi(s),s') + \gamma V^{\pi}(s')$
  - $V^{\pi}(s) \leftarrow (1-\alpha) \cdot V^{\pi}(s) + \alpha \cdot \text{sample}$
  - $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha \cdot [\text{sample} - V^{\pi}(s)]$
  - This is the ***temporal difference learning rule***
  - $[\text{sample} - V^{\pi}(s)]$ is the "TD error"
  - α is the ***learning rate***
- Observe a sample, move $V^{\pi}(s)$ a little bit to make it more consistent with its neighbor $V^{\pi}(s')$

Example: TD Value Estimation

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

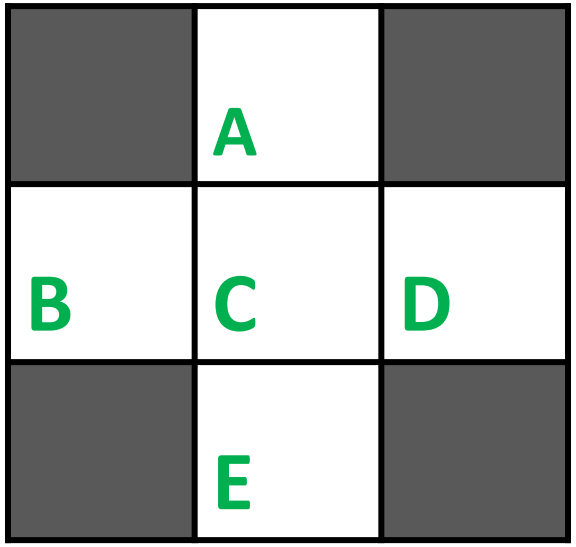
Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Output Values



Example: TD Value Estimation

- Experience transition *i:* (s_i, a_i, s_i', r_i).
- Compute sampled value "target": $r_i + \gamma V^{\pi}(s_i')$.
- Compute "TD error": $\delta_i = \left(r_i + \gamma V^{\pi}(s_i')\right) - V^{\pi}(s_i)$.
- Update: $V^{\pi}(s_i)$ += $\alpha_i \cdot \delta_i$.

s	*V(s)*
A	0
B	-2
C	9
D	10
E	8

i	*s*	*a*	*s'*	*r*	$r + \gamma V^{\pi}(s')$	$V^{\pi}(s)$	δ
1	B	east	C	-1	-1 + 0	0	-1
2	C	east	D	-1	-1 + 0	0	-1
3	D	exit	---	10	10 + 0	0	+10
4	B	east	C	-1	-1 + -1	-1	-1
5	C	east	D	-1	-1 + 10	-1	+10
6	D	exit	---	10	10 + 0	10	0
7	E	north	C	-1	-1 + 9	0	+8

B, east, C, -1
C, east, D, -1
D, exit, x, +10

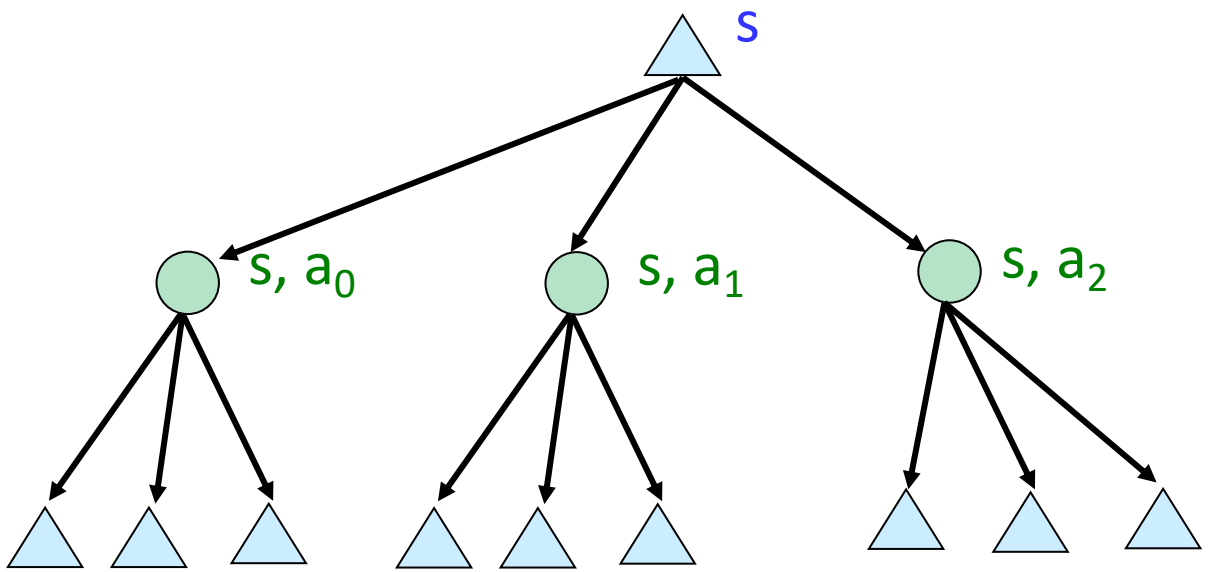
B, east, C, -1
C, east, D, -1
D, exit, x, +10

E, north, C, -1
C, east, D, -1
D, exit, x, +10

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Problems with TD Value Learning

- Model-free policy evaluation! 🎉
- Bellman updates with running sample mean! 🎉



- Need the transition model to improve the policy! 😱

Q-learning as approximate Q-iteration

- Recall the definition of Q values:
  - $Q^*(s,a)$ = expected return from doing a in s and then behaving optimally thereafter; and $\pi^*(s) = \max_a Q^*(s,a)$
- Bellman equation for Q values:
  - $Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$
- Approximate Bellman update for Q values:
  - $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} Q(s',a')]$
- We obtain a policy from learned Q(s,a), with no model!
  - (No free lunch: Q(s,a) table is |A| times bigger than V(s) table)

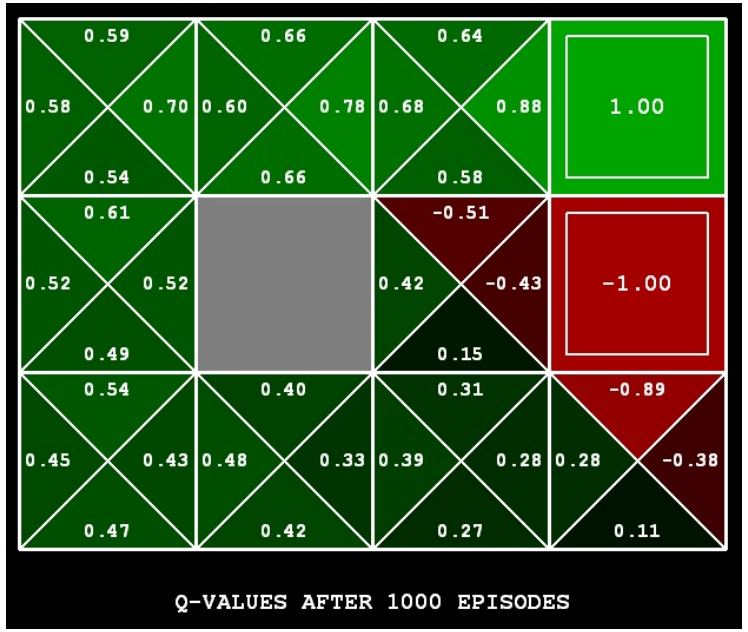
Q-Learning

- Learn Q(s,a) values as you go
  - Receive a sample (s,a,s',r)
  - Consider your old estimate: $Q(s,a)$
  - Consider your new sample estimate:

    $sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$

  - Incorporate the new estimate into a running average:

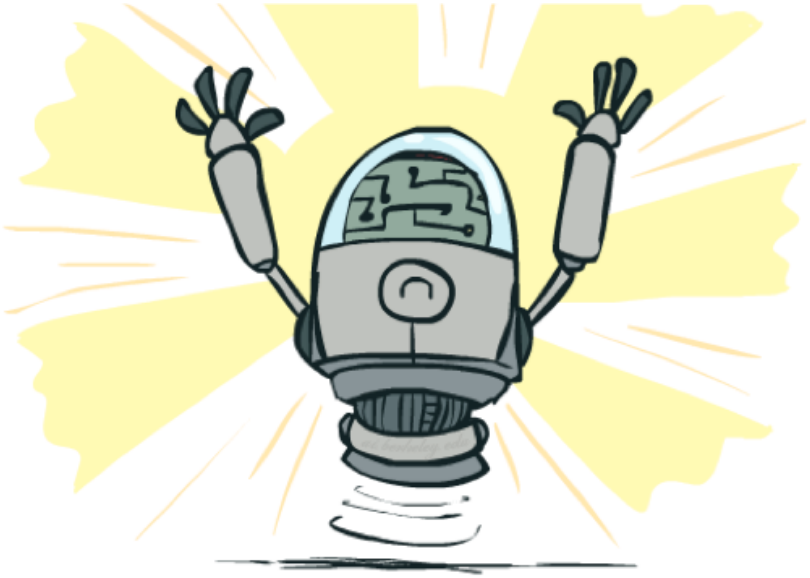
    $Q(s,a) \leftarrow (1-\alpha)\, Q(s,a) + \alpha \cdot [sample]$



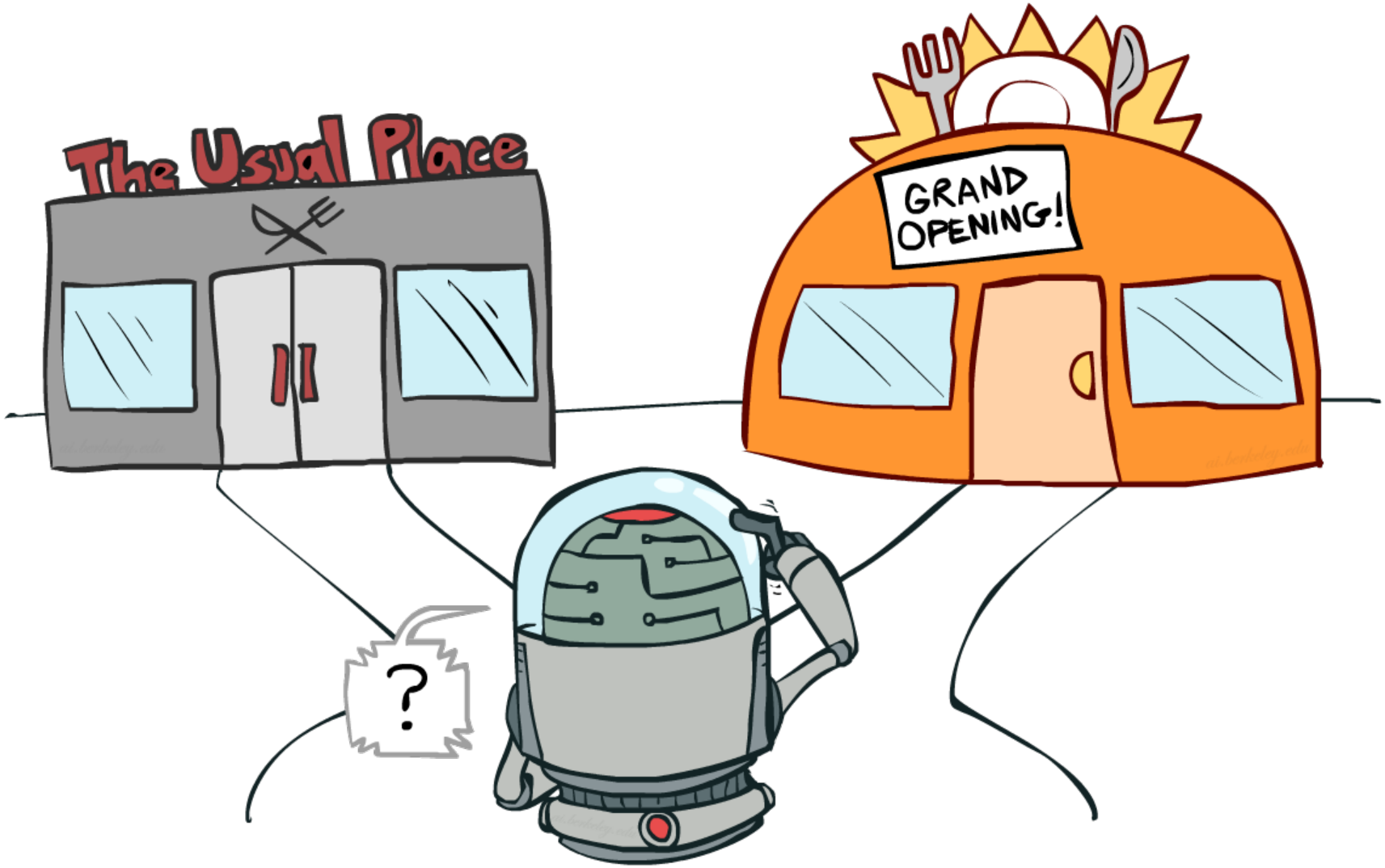
[Demo: Q-learning – gridworld (L10D2)]
[Demo: Q-learning – crawler (L10D3)]

Q-Learning Properties

- Amazing result: Q-learning converges to optimal policy -- even if samples are generated from a suboptimal policy!
- This is called off-policy learning
- Caveats:
  - You have to explore enough
  - You have to eventually make the learning rate small enough
  - … but not decrease it too quickly
  - Basically, in the limit, it doesn't matter how you select actions (!)



Exploration vs. Exploitation



Exploration vs. Exploitation

- ***Exploration***: try new things
- ***Exploitation***: do what's best given what you've learned so far
- Key point: pure exploitation often gets ***stuck in a rut*** and never finds an optimal policy!

Exploration method 1: ε-greedy

- ε-greedy exploration
  - Every time step, flip a biased coin
  - With (small) probability ε, act randomly
  - With (large) probability 1-ε, act on current policy

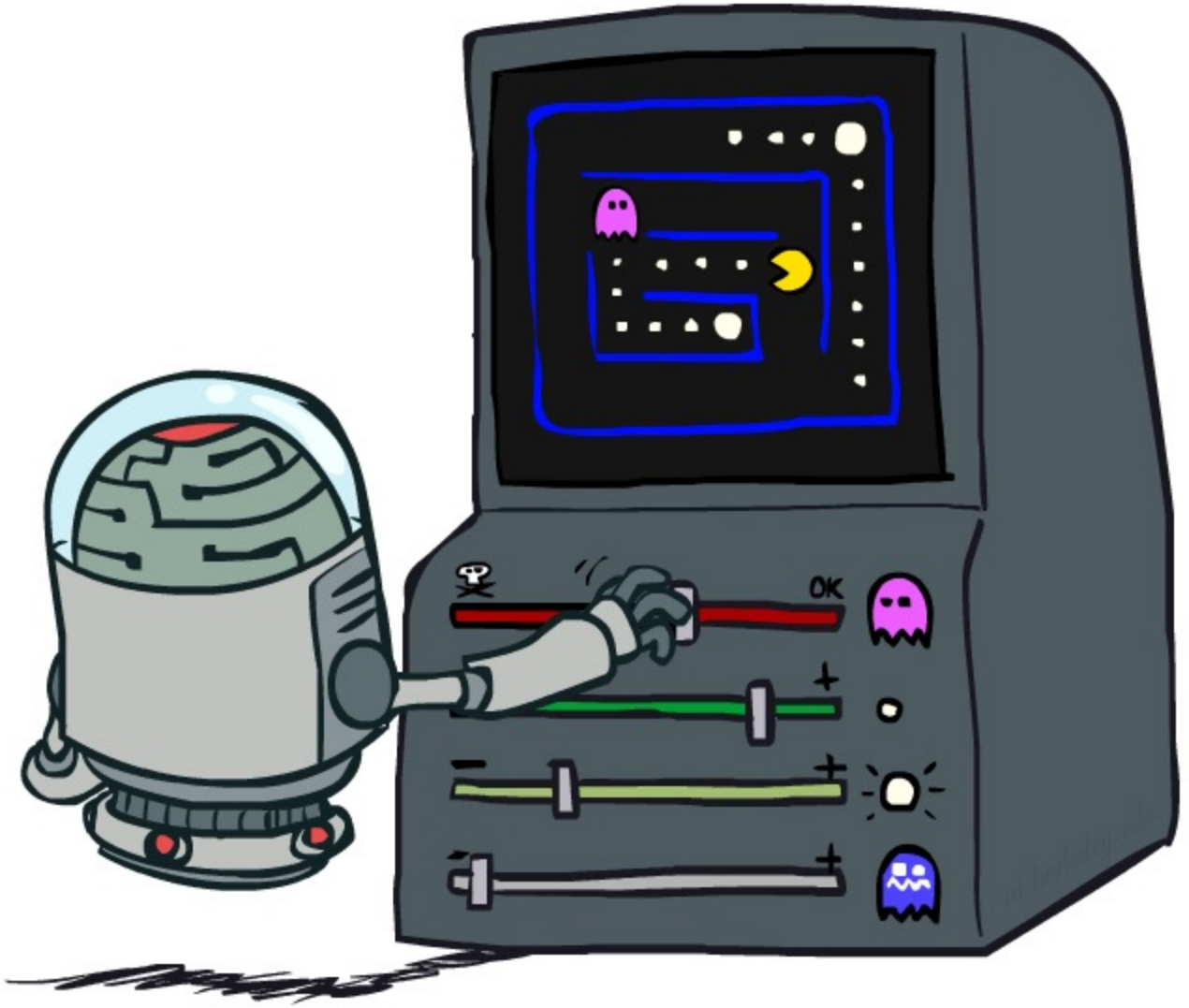
- Properties of ε-greedy exploration
  - Every s,a pair is tried infinitely often
  - Does a lot of stupid things
    - Jumping off a cliff *lots of times* to make sure it hurts
  - Keeps doing stupid things for ever
    - Decay ε towards 0

Method 2: Optimistic Exploration Functions

- ***Exploration functions*** implement this tradeoff
  - Takes a value estimate u and a visit count n, and returns an optimistic utility, e.g., $f(u,n) = u + k/\sqrt{n}$

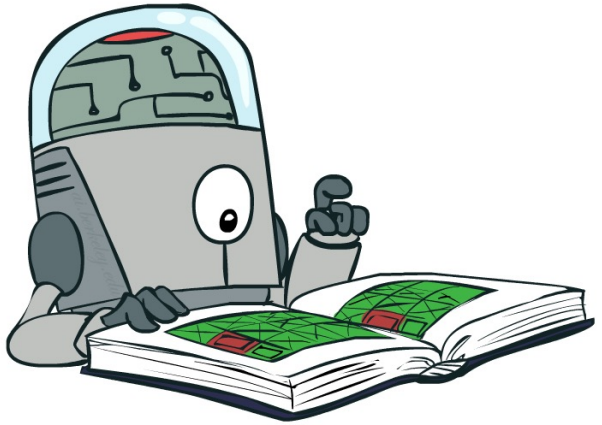
- Regular Q-update:
  - $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_a Q(s',a)]$
- Modified Q-update:
  - $Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_a f(Q(s',a'),n(s',a'))]$
- Note: this propagates the "bonus" back to states that lead to unknown states as well!

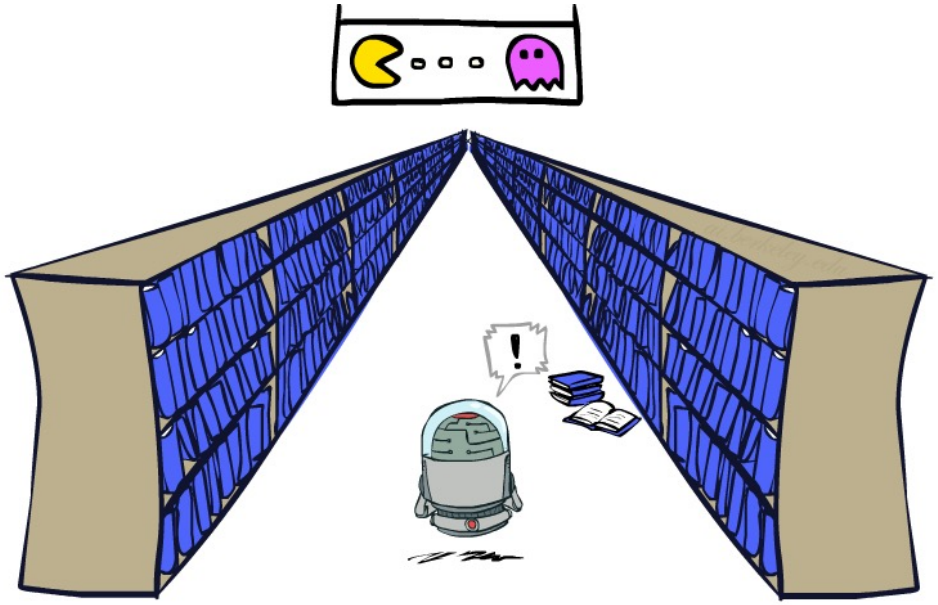
Approximate Q-Learning



Generalizing Across States

- Basic Q-Learning keeps a table of all Q-values
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to hold the Q-tables in memory
- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar situations
  - Can we apply some machine learning tools to do this?

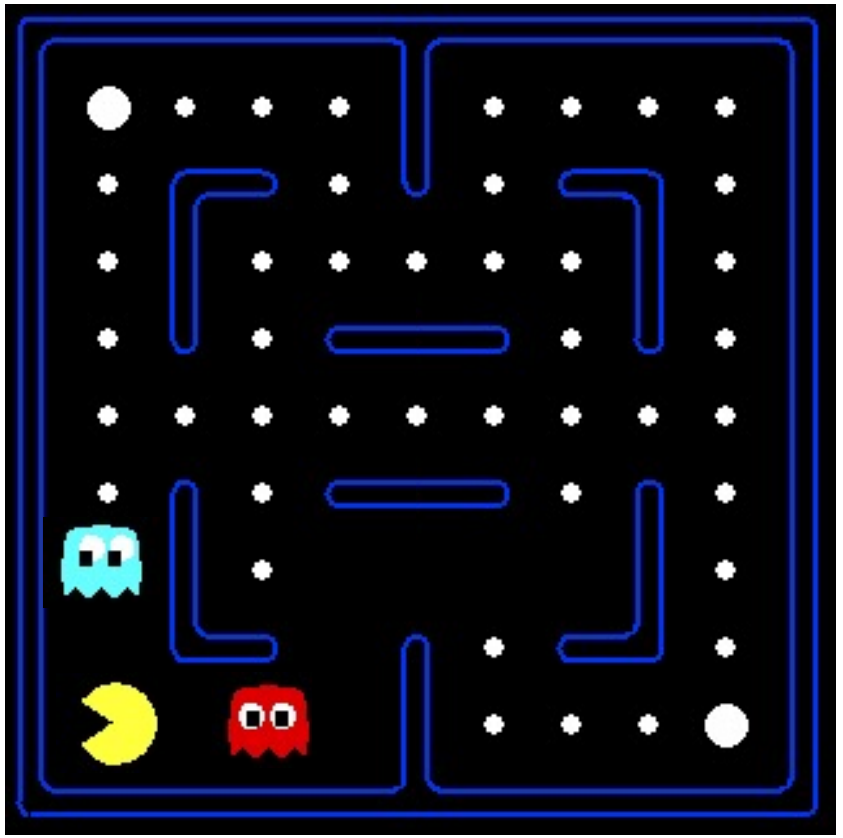




[demo – RL pacman]

Feature-Based Representations

- Solution: describe a state using a vector of ***features***
  - Features are functions from states to real numbers (often 0/1) that capture important properties of the state
  - Example features:
    - Distance to closest ghost f_{GST}
    - Distance to closest dot
    - Number of ghosts
    - 1 / (distance to closest dot) f_{DOT}
    - Is Pacman in a tunnel? (0/1)
    - …… etc.
  - Can also describe a q-state (s, a) with features (e.g., action moves closer to food)



Linear Value Functions

- We can express V and Q (approximately) as weighted linear functions of feature values:
  - $V_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$
  - $Q_\theta(s,a) = \theta_1 f_1(s,a) + \theta_2 f_2(s,a) + \ldots + \theta_n f_n(s,a)$

- Advantage: our experience is summed up in a few powerful numbers
  - Can compress a value function for chess (10^{43} states) down to about 30 weights!

- Disadvantage: states may share features but have very different expected utility!

SGD for Linear Value Functions

- Goal: Find parameter vector θ that minimizes the mean squared error between the true and approximate value function

$$J(\theta) = \mathbb{E}_\pi[\frac{1}{2}\big(V^\pi(s) - V_\theta(s)\big)^2]$$

- Stochastic gradient descent:

$$\theta \leftarrow \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$$

$$= \theta + \alpha\big(V^\pi(s) - V_\theta(s)\big)\frac{\partial V_\theta(s)}{\partial \theta}$$

Supervised Learning for Value Function Approximation

- Let $V^{\pi}(s)$ denote the true target value function
- Use supervised learning on "training data" to predict the value function:

$$\langle s_1, G_1 \rangle, \langle s_2, G_2 \rangle, \ldots, \langle s_T, G_T \rangle$$

- For each data sample

$$\theta \leftarrow \theta + \alpha\big(G_t - V_\theta(s_t)\big) f(s_t)$$

Temporal-Difference (TD) Learning Objective

$$\theta \leftarrow \theta + \alpha\big(V^{\pi}(s) - V_{\theta}(s)\big)\nabla f(s)$$

- In TD learning, $r_{t+1} + \gamma V_{\theta}(s_{t+1})$ is a data sample for the target
- Apply supervised learning on "training data":

$$\langle s_1, r_2 + \gamma V_{\theta}(s_2)\rangle, \langle s_2, r_3 + \gamma V_{\theta}(s_3)\rangle, \ldots, \langle s_T, r_T\rangle$$

- For each data sample, update

$$\theta \leftarrow \theta + \alpha\big(r_{t+1} + \gamma V_{\theta}(s_{t+1}) - V_{\theta}(s_t)\big)\nabla f(s_t)$$

Q-Value Function Approximation

- Approximate the action-value function:

$$Q_\theta(s,a) \simeq Q^\pi(s,a)$$

- Objective: Minimize the **mean squared error**:

$$J(\theta) = \mathbb{E}_\pi\left[\frac{1}{2}(Q^\pi(s,a) - Q_\theta(s,a))^2\right]$$

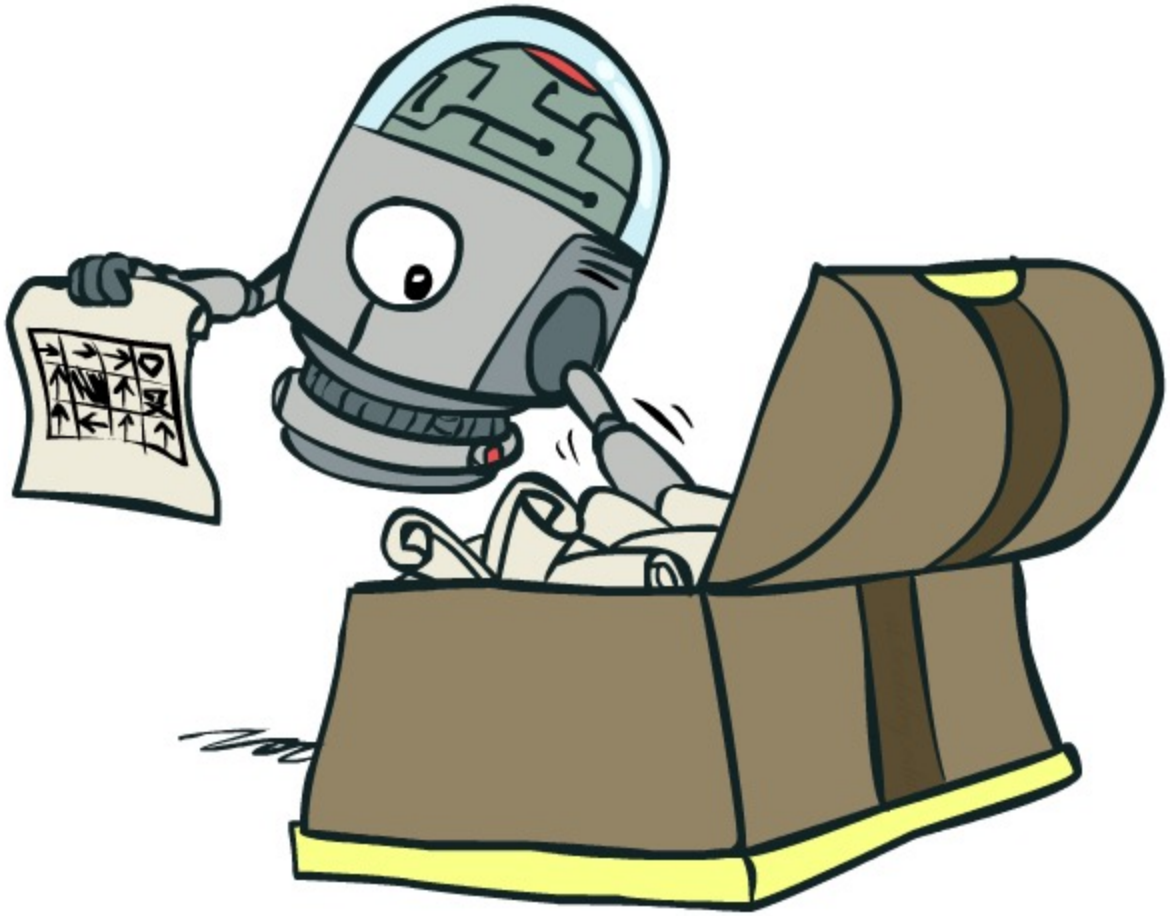
- Stochastic Gradient Descent on a single sample

Intuitive interpretation

- Original Q-learning rule tries to reduce prediction error at s,a:
  - $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
- Instead, we update the weights to try to reduce the error at s,a:
  - $w_i \leftarrow w_i + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)] \partial Q_{\mathbf{w}}(s,a)/\partial w_i$

    $= w_i + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)] f_i(s,a)$
- Intuitive interpretation:
  - Adjust weights of active features
  - If something bad happens, blame the features we saw; decrease value of states with those features. If something good happens, increase value!

Policy Search



Policy Search

- Problem: often the feature-based policies that work well (win games, maximize utilities) aren't the ones that approximate V / Q best
  - E.g. your value functions were probably horrible estimates of future rewards, but they still produced good decisions
  - Q-learning's priority: get Q-values close (modeling)
  - Action selection priority: get ordering of Q-values right (prediction)

- Solution: learn policies that maximize rewards, not the values that predict them

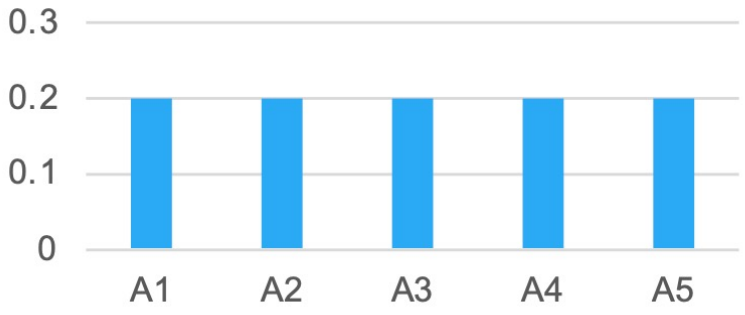
- Policy search: start with an ok solution (e.g. Q-learning) then fine-tune by **hill climbing** (or gradient ascent!) on feature weights

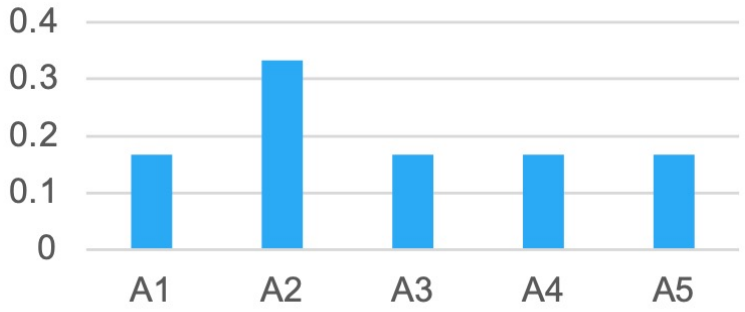
Parameterized Policy

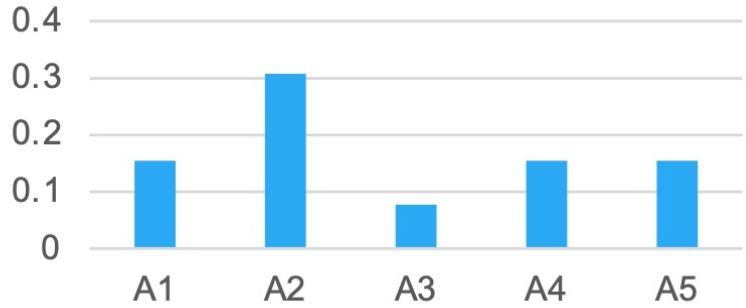
- A policy can be parameterized as $\pi_\theta(a|s)$
- The policy can be deterministic: $a = \pi_\theta(s)$
  - Or stochastic: $\pi_\theta(a|s) = P(a|s;\theta)$
- θ represents the parameters of the policy

Policy Gradient

- Simplest version:
  - Start with initial policy $\pi(s)$ that assigns probability to each action
  - Sample actions according to policy π
  - Update policy:
    - If an episode led to high utility, make sampled actions more likely
    - If an episode led to low utility, make sampled actions less likely







南方科技大学

STA303: Artificial Intelligence

Machine Learning Basics

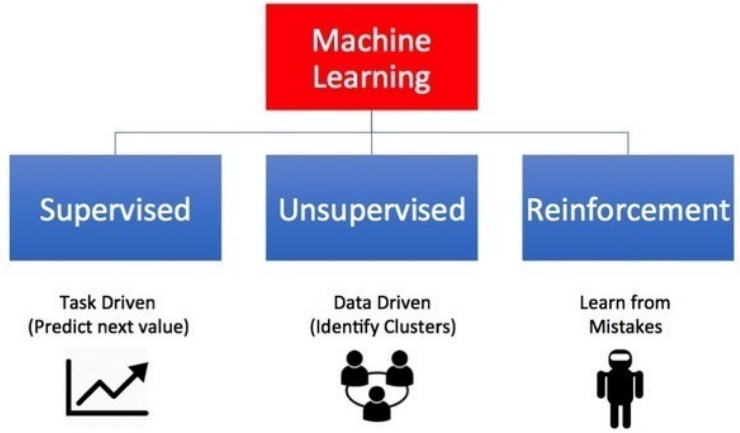
Fang Kong

https://fangkongx.github.io/

Part of slide credits: ai.berkeley.edu; Stanford CS229; https://www.geeksforgeeks.org/

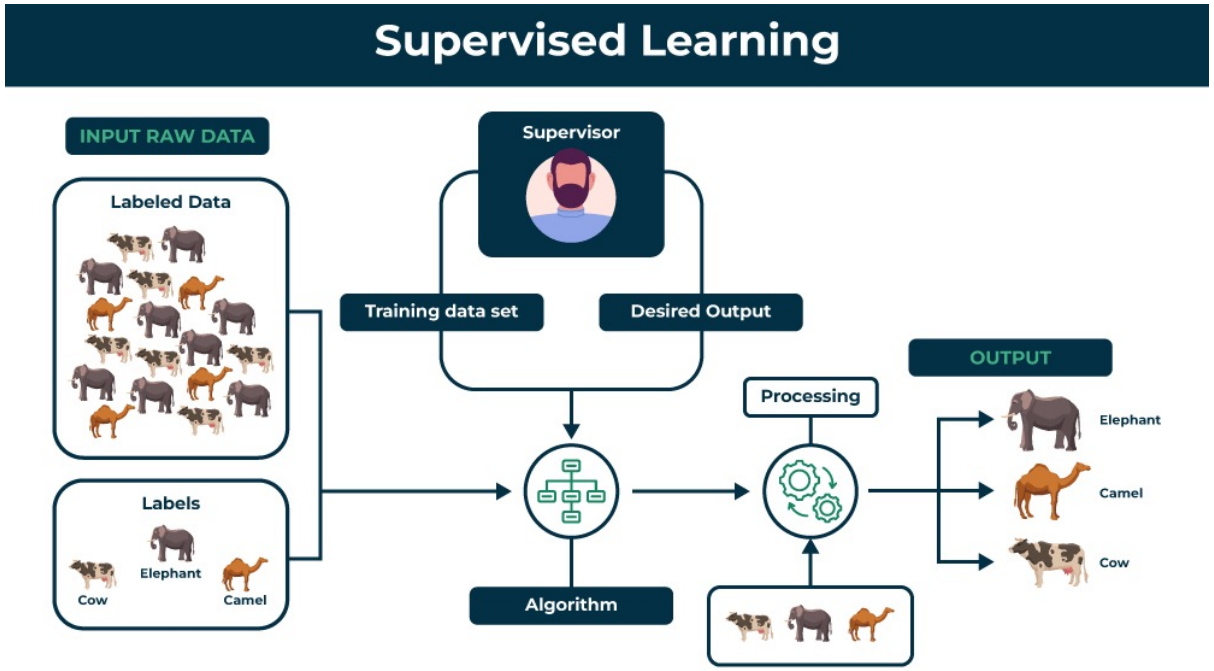
Types of Machine Learning

- Supervised learning
  - Use labeled data to predict on unseen points
- Unsupervised learning
  - No labeled data
- Reinforcement learning
  - Sequentially collect data and learn from feedback

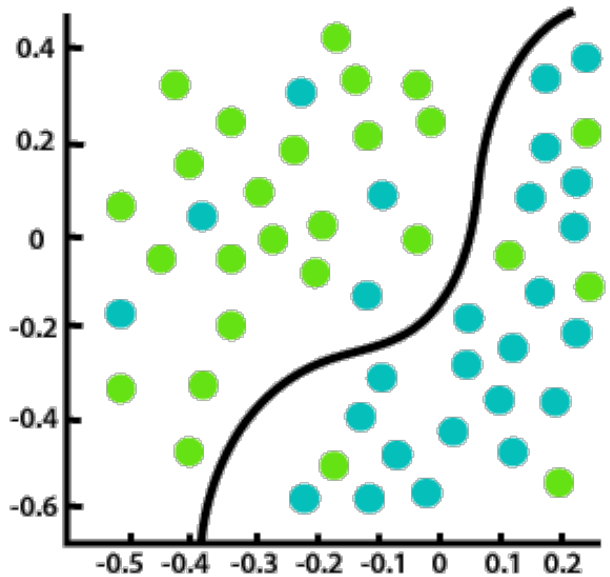


Supervised Learning

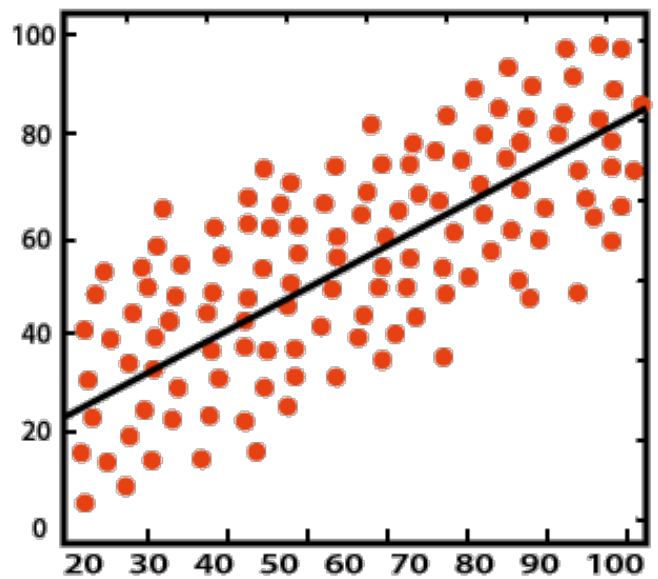
- Trained on a “Labelled Dataset”
- Labelled datasets have both input and output parameters



Tasks in Supervised Learning



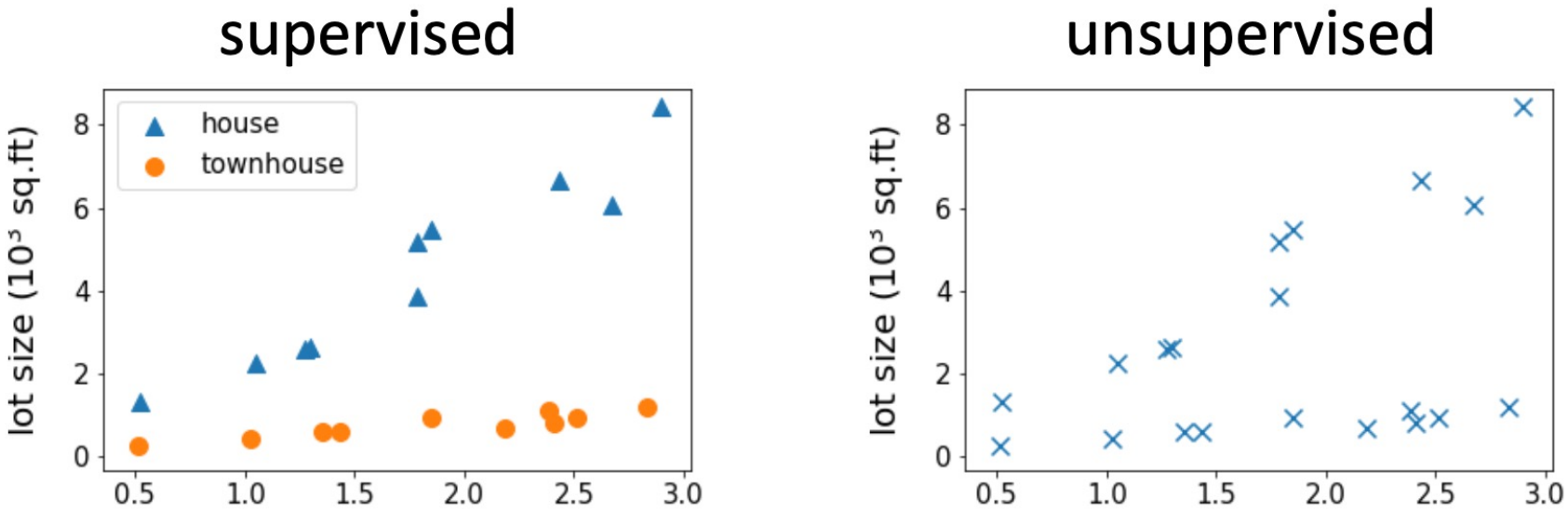
Classification



Regression

Unsupervised Learning

- Discover patterns and relationships using unlabeled data
- Without labeled target outputs



Tasks in Unsupervised Learning

Clustering

- K-Means
- Polynomial
- Hierarchical
- Fuzzy C-Means

Grouping data points into clusters based on their similarity

Dimensionality Reduction

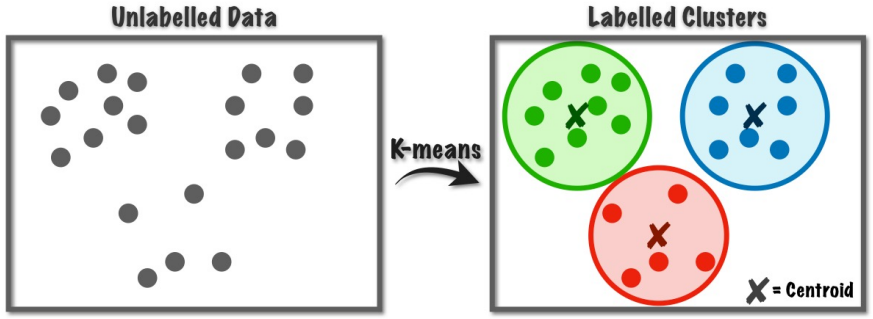
- Principal Component Analysis
- Kernel Principal Analysis

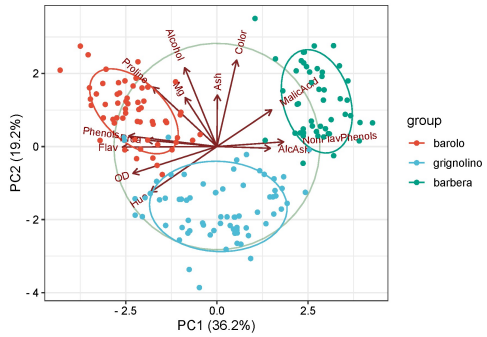
Reduce the dimensionality of data while preserving its essential information

Association (Data Mining)

- Apriori Algorithm
- Eclat Algorithm
- FP-Growth Algorithm

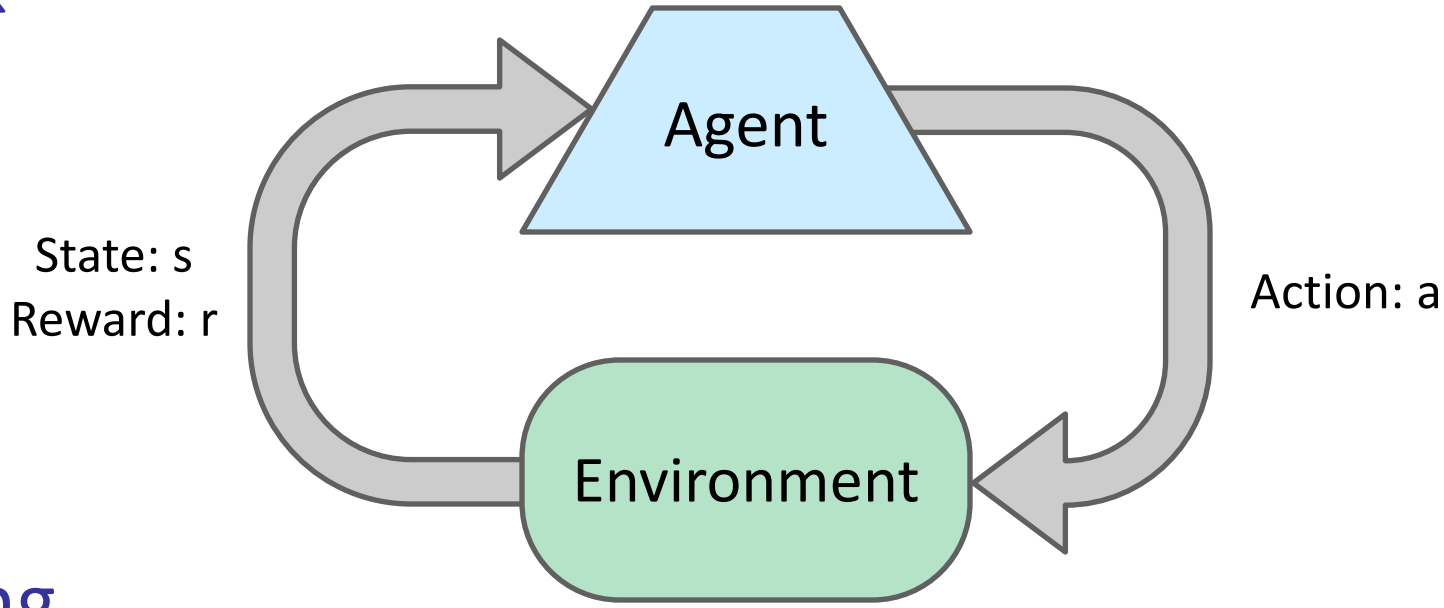
Find the relationships between variables in the large database





Reinforcement Learning

- Interact with the environment by producing actions and receiving feedback



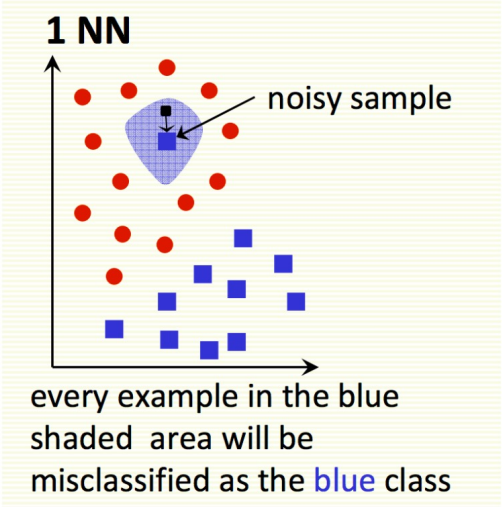
- Q-learning
- Deep Q-learning
- PPO

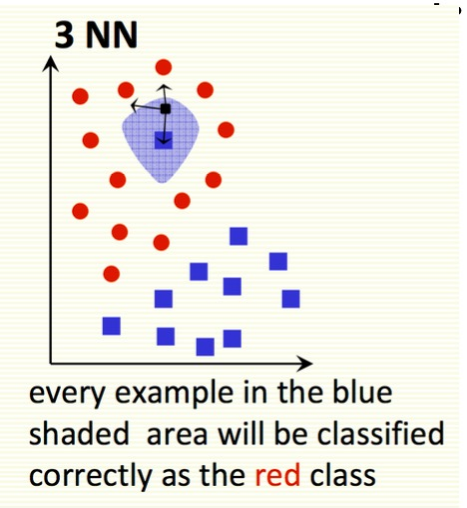
Machine Learning Workflow

- 1. Gather and organize data
  - Preprocessing, cleaning, visualizing
- 2. Choose a model
- 3. Train and test your model, or iterate back to step 2 or 1
- 4. Deploy your model

K-Nearest Neighbors (KNN)

- Nearest neighbors sensitive to noise or mis-labeled data
- Smooth by having k nearest neighbors vote

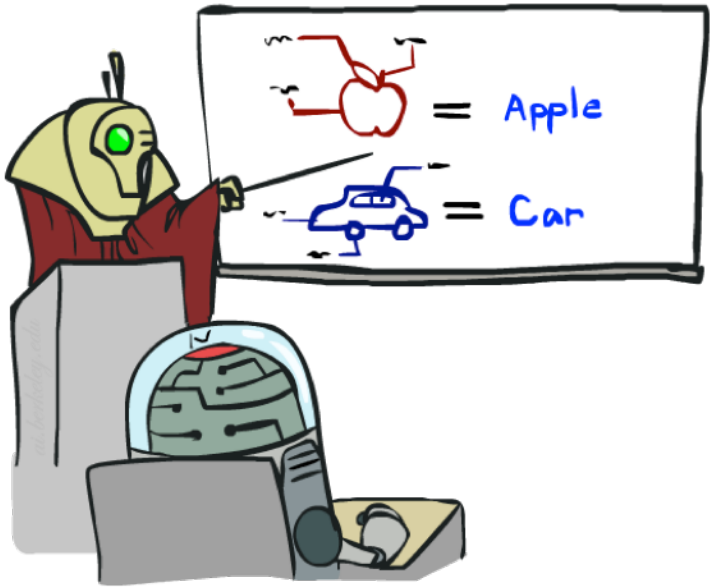


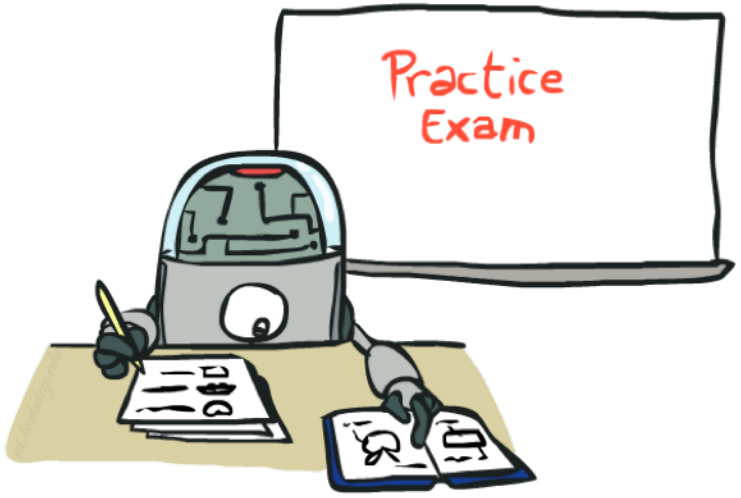


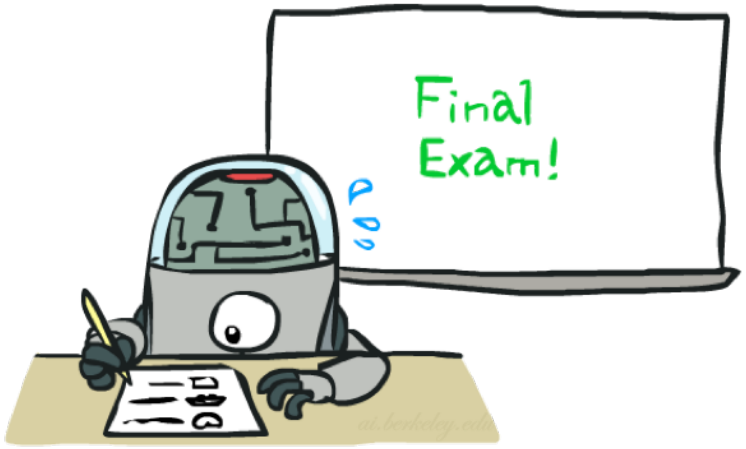
- Voting over k nearest neighbors: classification
- (Weighted) average over k nearest neighbors: regression

[Pic by Olga Veksler]

Step 3: Training and Testing







How to select a model?

- To solve a problem, which model should we choose?
  - KNN or logistic regression?
  - For KNN, which parameter k?

- Denote $\mathcal{M} = \{M_1, \ldots, M_d\}$ as all the models to choose

Select the one with the minimum training loss?

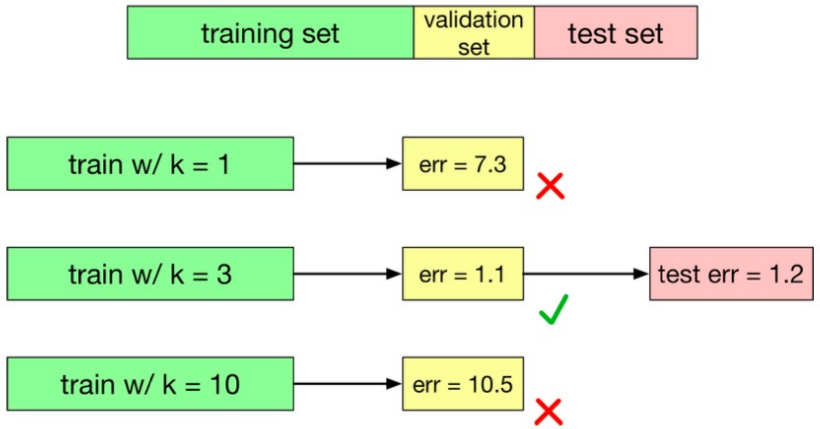
- Given the training set S

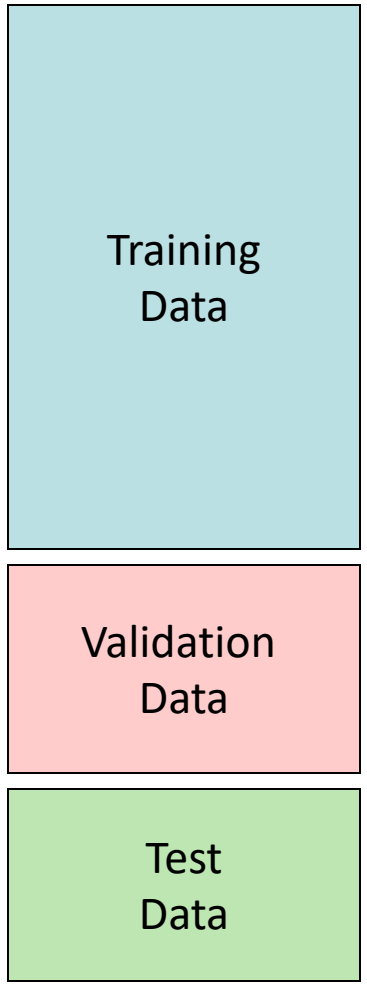
1. Train each model M_i on S, to get some hypothesis h_i.
2. Pick the hypotheses with the smallest training error.

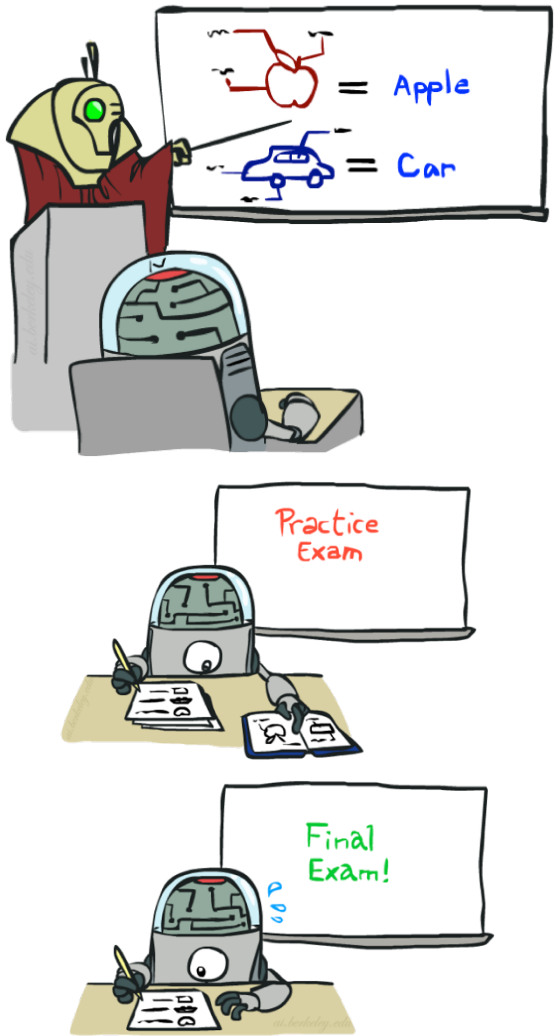
- What's the problem?
  - Lower training error prefers complex models
  - These models usually overfits

Solution: Hold-out cross validation

- How do we check that we're not overfitting during training?
- Split training data into 3 different sets:
  - Training set
  - Validation set
  - Test set
- Experimentation cycle
  - Learn parameters on training set
  - Evaluate models on validation set
  - Very important: never "peek" at the test set!







Hold-out cross validation (cont'd)

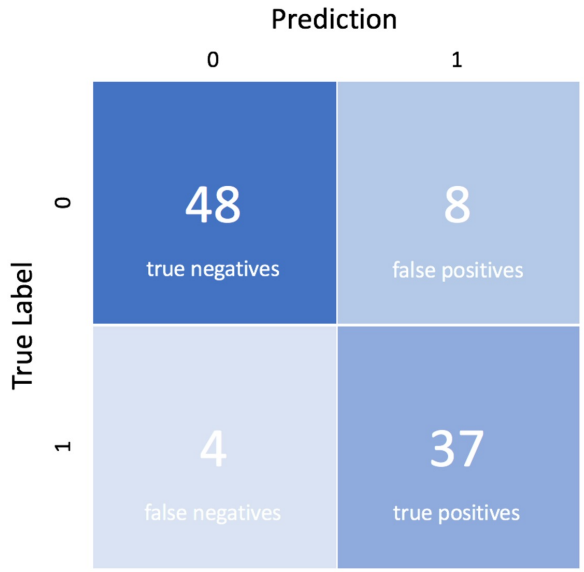
- The final model is only trained on 70% of the training set
- Especially in the case with small training set
  - Waste about 30% of the data

Improvement: k-fold cross validation

Evaluation: Confusion matrix

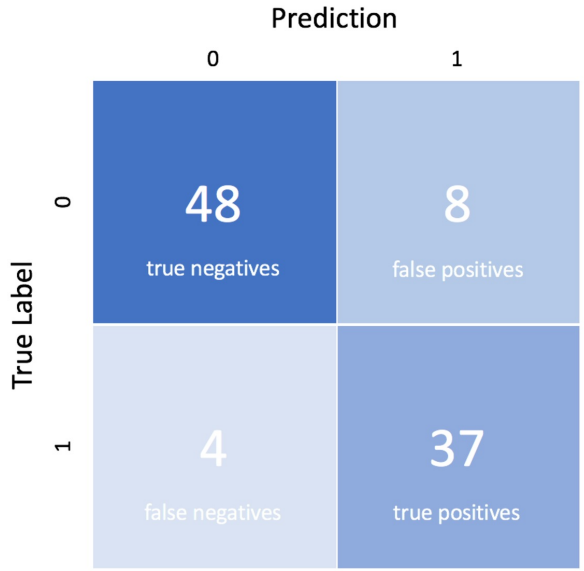
- Given a set of records containing positive and negative results, the computer is going to classify the records to be positive or negative

- Positive: The computer classifies the result to be positive
- Negative: The computer classifies the result to be negative
- True: What the computer classifies is true
- False: What the computer classifies is false



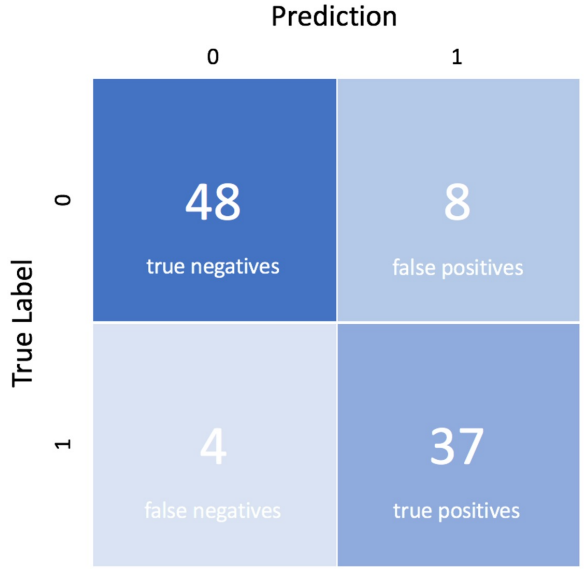
Accuracy

- Accuracy = $\frac{TN+TP}{TN+TP+FN+FP} = \frac{48+37}{48+37+4+8}$



Accuracy

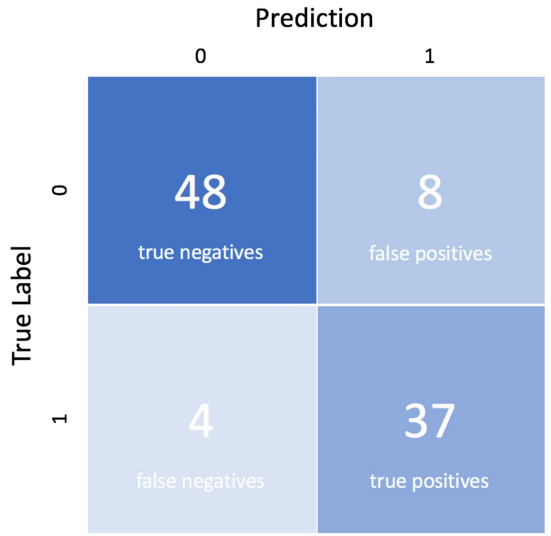
- Accuracy = $\frac{TN+TP}{TN+TP+FN+FP} = \frac{48+37}{48+37+4+8}$

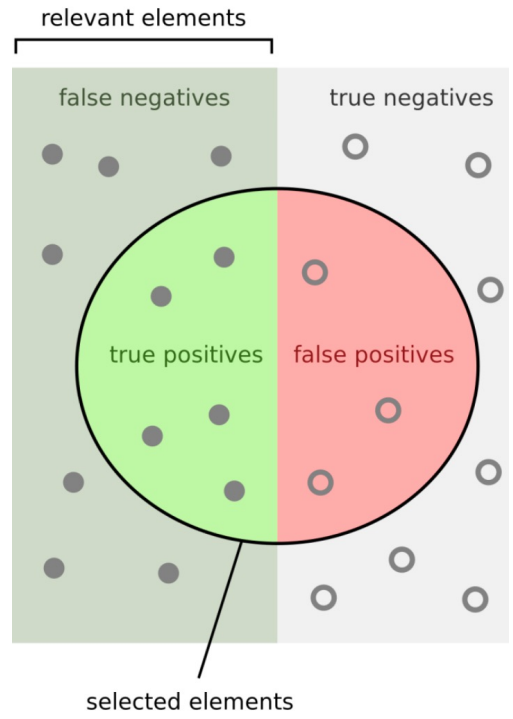


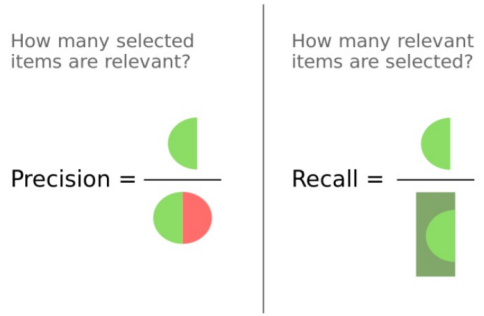
- Limitation
  - Suppose number of class 0 examples = 9990
  - Number of class 1 examples = 10
  - The model predicts every example as 0
  - Then the accuracy is 9990/10000=99.9%
  - The accuracy is misleading because the model does not detect any example in class 1

Other metrics

- Precision = $\frac{TP}{TP+FP} = \frac{37}{37+8}$
- Recall = $\frac{TP}{TP+FN} = \frac{37}{37+4}$
- F-measure = $\frac{2*\text{Precision}*\text{Recall}}{\text{Precision}+\text{Recall}}$







How to understand?

- A school is running a machine learning primary diabetes scanon all of its students
  - Diabetic (+) / Healthy (-)
  - False positive is just a false alarm
  - False negative
    - Prediction is healthy but is diabetic
    - Worst case among all 4 cases
- Accuracy
  - Accuracy = (TP+TN)/(TP+FP+FN+TN)
  - How many students did we correctly label out of all the students?

How to understand?

- A school is running a machine learning primary diabetes scanon all of its students
  - Diabetic (+) / Healthy (-)
  - False positive is just a false alarm
  - False negative
    - Prediction is healthy but is diabetic
    - Worst case among all 4 cases
- Precision
  - Precision = TP/(TP+FP)
  - How many of those who we labeled as diabetic are actually diabetic?

How to understand?

- A school is running a machine learning primary diabetes scanon all of its students
  - Diabetic (+) / Healthy (-)
  - False positive is just a false alarm
  - False negative
    - Prediction is healthy but is diabetic
    - Worst case among all 4 cases
- Recall (sensitivity)
  - Recall = TP/(TP+FN)
  - Of all the people who are diabetic, how many of those we correctly predict?

F1 score (F-Score / F-Measure)

- F1 Score = 2*(Recall * Precision) / (Recall + Precision)
- F1 Score=$\frac{1}{2}$((1/Recall + 1/Precision))$^{-1}$

- Harmonic mean (average) of the precision and recall
- F1 Score is best if there is some sort of balance between precision (p) & recall (r) in the system.
- Oppositely F1 Score isn't so high if one measure is improved at the expense of the other.
- For example, if P is 1 & R is 0, F1 score is 0.

Which to choose?

- Accuracy
  - A great measure
  - But only when you have symmetric datasets
- Precision
  - Want to be more confident of your TP
  - E.g. spam emails. We'd rather have some spam emails in inbox rather than some regular emails in your spam box.

Which to choose?

- Recall
  - If FP is far better than FN or if the occurrence of FN is unaccepted/intolerable
  - Would like more extra FP (false alarms) over saving some FN
  - E.g. diabetes. We'd rather get some healthy people labeled diabetic over leaving a diabetic person labeled healthy
- F1 score
  - If the costs of FP and FN are both important

南方科技大学

STA303: Artificial Intelligence

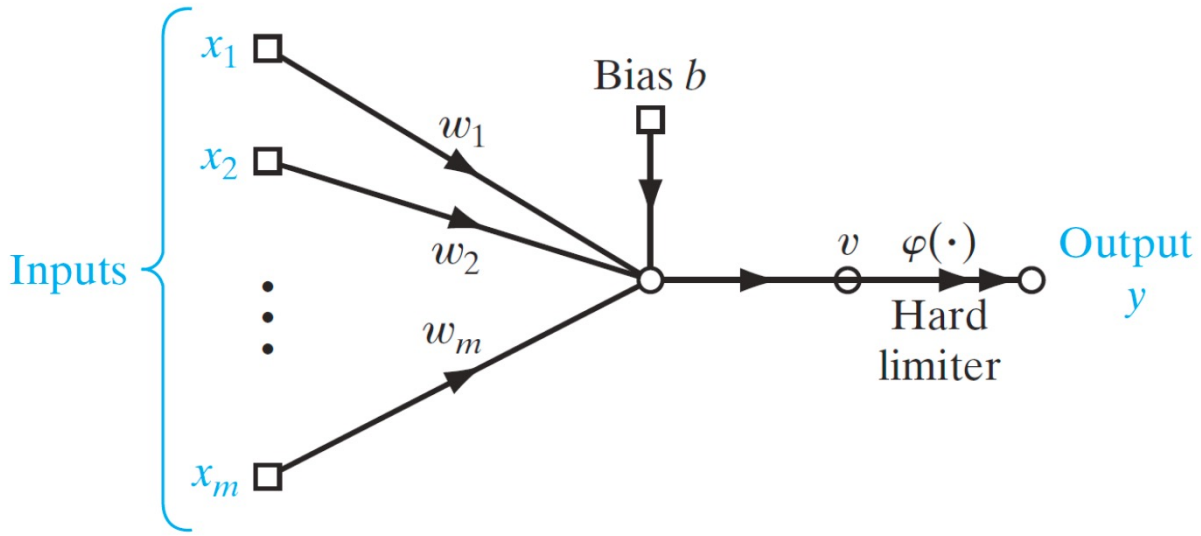
Deep Learning

Fang Kong

https://fangkongx.github.io/

Part of slide credit: ai.berkeley.edu; SJTU VE445

Single-layer perception by Rosenblatt [1958]



- Rosenblatt [1958] 进一步提出感知机作为第一个在“老师”指导下进行学习的模型（即监督学习）
- 专注在如何找到合适的用于二分类任务的权重w_m
  - $y = 1$：类别1
  - $y = -1$：类别2

预测

$$\hat{y} = \sigma\left(\sum_{i=1}^{m} w_i x_i + b\right)$$

激活函数

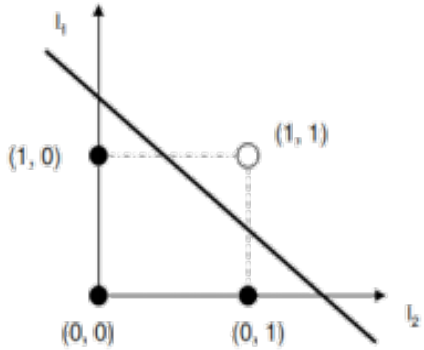
$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Slide credit: Weinan Zhang

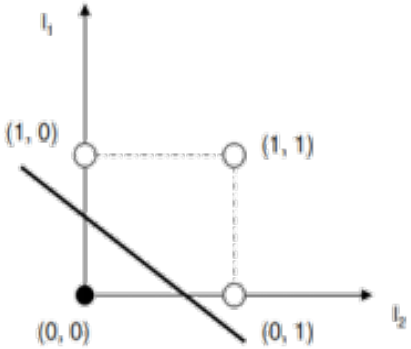
Limitation of perception

- Minsky and Papert [1969] showed that some rather elementary computations, such as XOR problem, could not be done by Rosenblatt's one-layer perceptron
- However Rosenblatt believed the limitations could be overcome if more layers of units to be added, but no learning algorithm known to obtain the weights yet

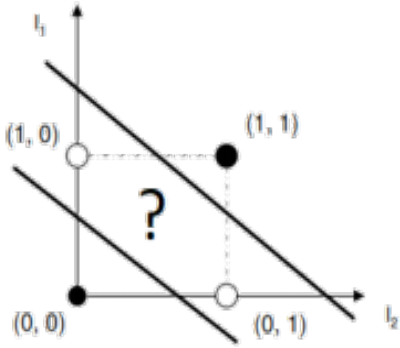
AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1



OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1

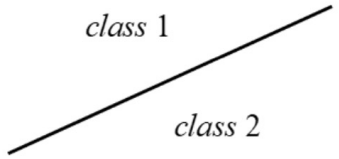


XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0

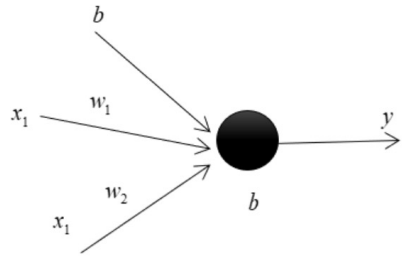


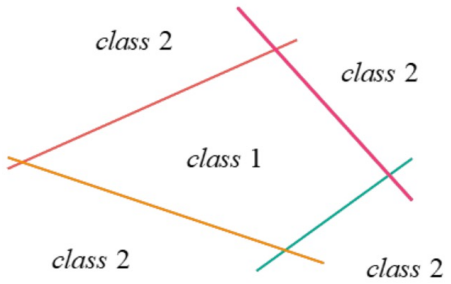
Solution: Add hidden layers

- Adding hidden layers to learn more general scenarios

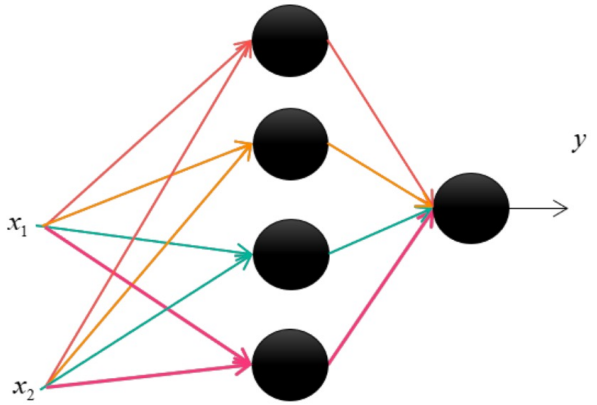


决策边界：$x_1 w_1 + x_2 w_2 + b = 0$





每个隐含节点负责实现凸区域的一条边界线

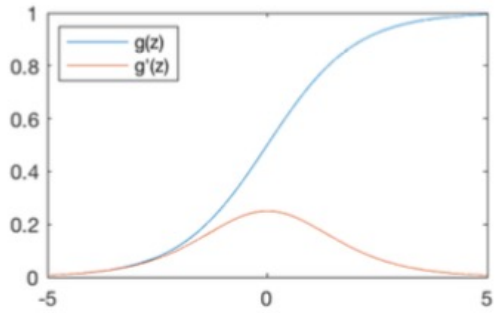


Slide credit: Weinan Zhang

Non-linear activation functions

- Adding non-linearity allows the network to learn and represent complex patterns in the data
- Common non-linear activation functions

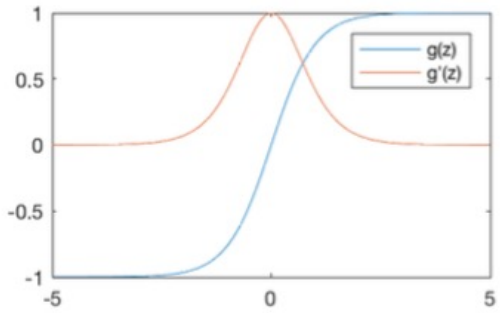
Sigmoid Function



$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

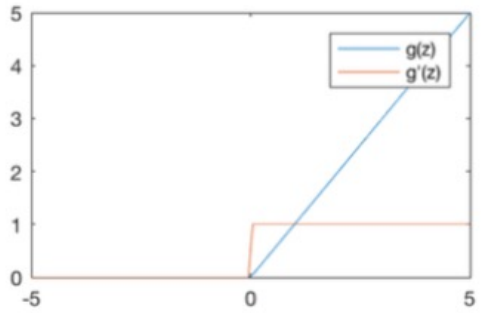
Hyperbolic Tangent



$$\sigma(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$\sigma'(z) = 1 - \sigma(z)^2$$

Rectified Linear Unit (ReLU)

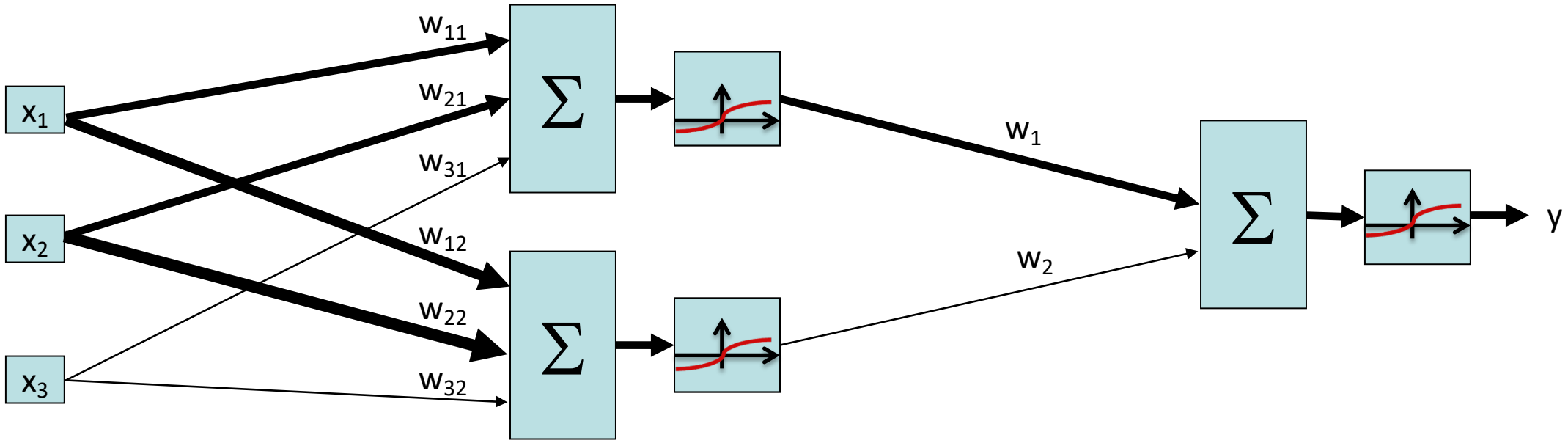


$$\sigma(z) = \max(0, z)$$

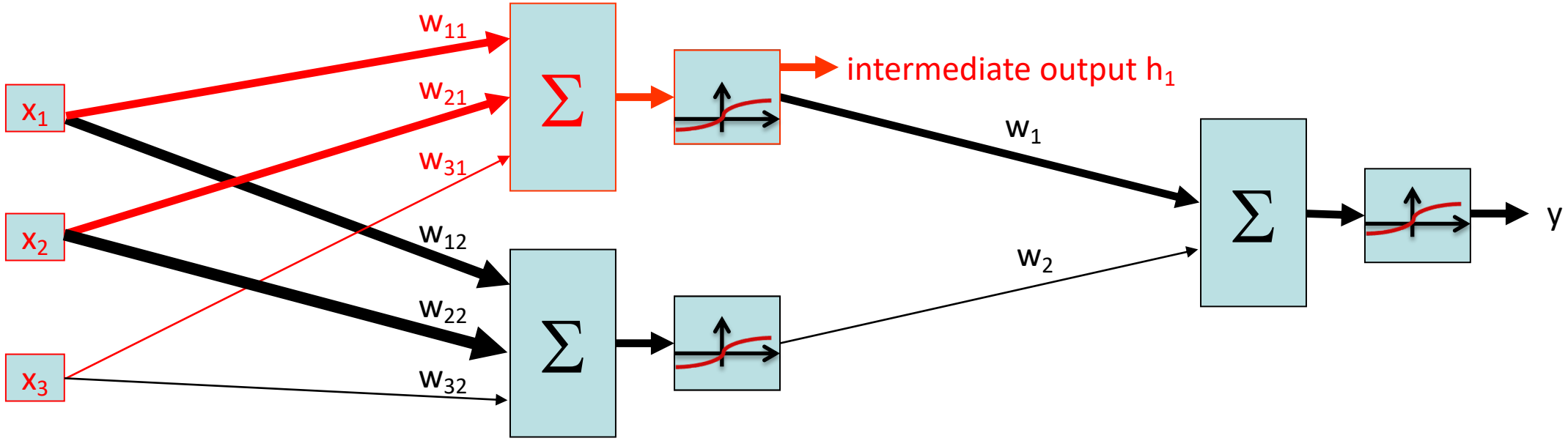
$$\sigma'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

[source: MIT 6.S191 introtodeeplearning.com]

2-Layer, 2-Neuron Neural Network



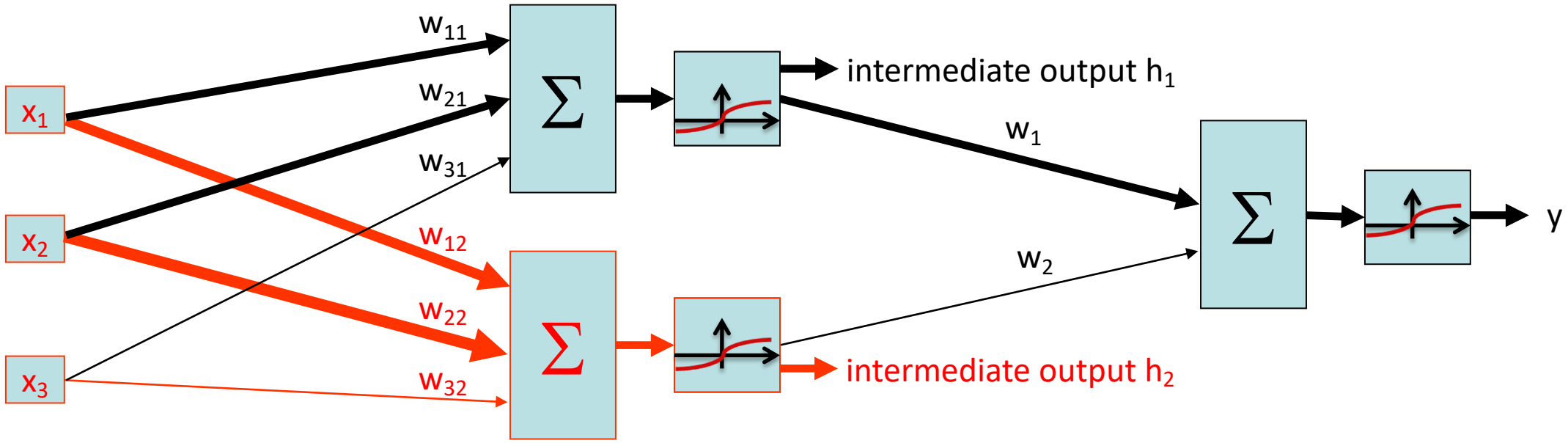
2-Layer, 2-Neuron Neural Network



$$\text{intermediate output } h_1 = \sigma(w_{11}x_1 + w_{21}x_2 + w_{31}x_3)$$

$$= \frac{1}{1 + e^{-(w_{11}x_1 + w_{21}x_2 + w_{31}x_3)}}$$

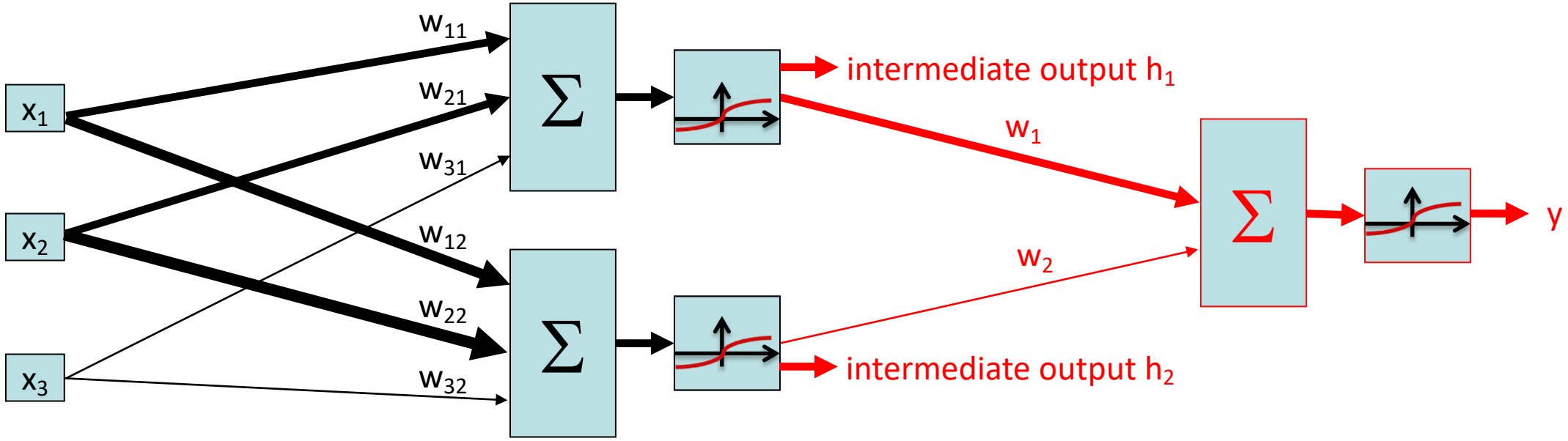
2-Layer, 2-Neuron Neural Network



$$\text{intermediate output } h_2 = \sigma(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)$$

$$= \frac{1}{1 + e^{-(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)}}$$

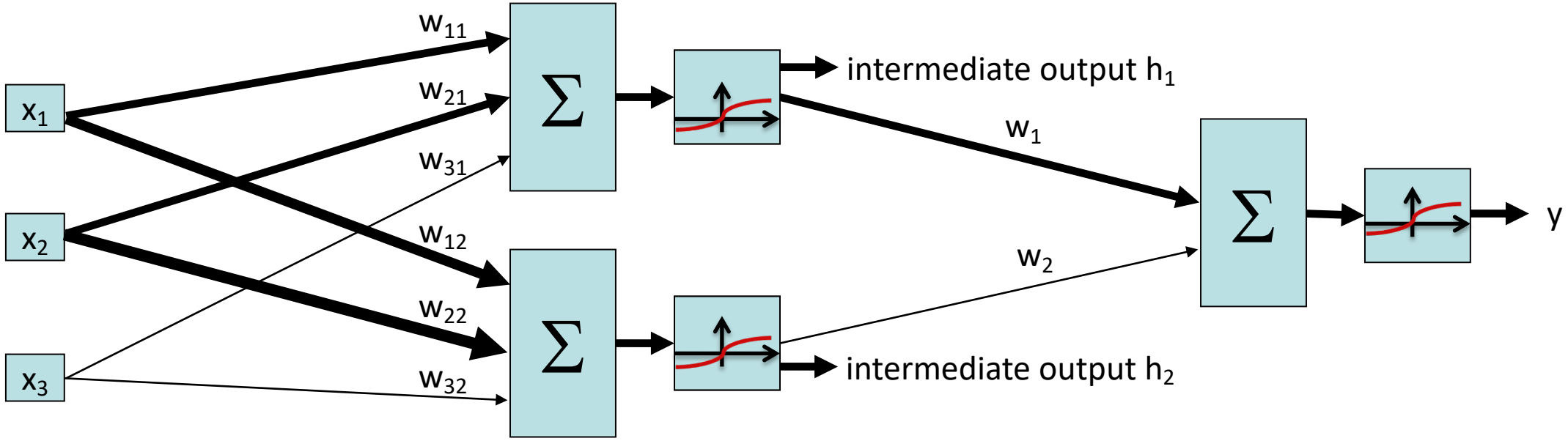
2-Layer, 2-Neuron Neural Network



$$y = \sigma(w_1 h_1 + w_2 h_2)$$

$$= \frac{1}{1 + e^{-(w_1 h_1 + w_2 h_2)}}$$

2-Layer, 2-Neuron Neural Network



$$y = \sigma(w_1 h_1 + w_2 h_2)$$
$$= \sigma(w_1 \sigma(w_{11} x_1 + w_{21} x_2 + w_{31} x_3) + w_2 \sigma(w_{12} x_1 + w_{22} x_2 + w_{32} x_3))$$

Vectorization

$$
\begin{aligned}
y &= \sigma(w_1 h_1 + w_2 h_2) \\
&= \sigma(w_1 \sigma(w_{11} x_1 + w_{21} x_2 + w_{31} x_3) + w_2 \sigma(w_{12} x_1 + w_{22} x_2 + w_{32} x_3))
\end{aligned}
$$

The same equation, formatted with matrices:

$$
\begin{aligned}
&\sigma\left(\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}\right) \\
&= \sigma\left(\begin{bmatrix} w_{11} x_1 + w_{21} x_2 + w_{31} x_3 & w_{12} x_1 + w_{22} x_2 + w_{32} x_3 \end{bmatrix}\right) \\
&= \begin{bmatrix} h_1 & h_2 \end{bmatrix}
\end{aligned}
$$

$$
\sigma\left(\begin{bmatrix} h_1 & h_2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}\right) = \sigma(w_1 h_1 + w_2 h_2) = y
$$

The same equation, formatted more compactly by introducing variables representing each matrix:

$$
\sigma(x \times W_{\text{layer } 1}) = h \qquad \sigma(h \times W_{\text{layer } 2}) = y
$$

2-Layer, 2-Neuron Neural Network

$$\sigma(x \times W_{\text{layer 1}}) = h$$

Shape (1, 3).
Input feature vector.

Shape (3, 2).
Weights to be learned.

Shape (1, 2).
Outputs of layer 1,
inputs to layer 2.

$$\sigma(h \times W_{\text{layer 2}}) = y$$

Shape (1, 2).
Outputs of layer 1,
inputs to layer 2.

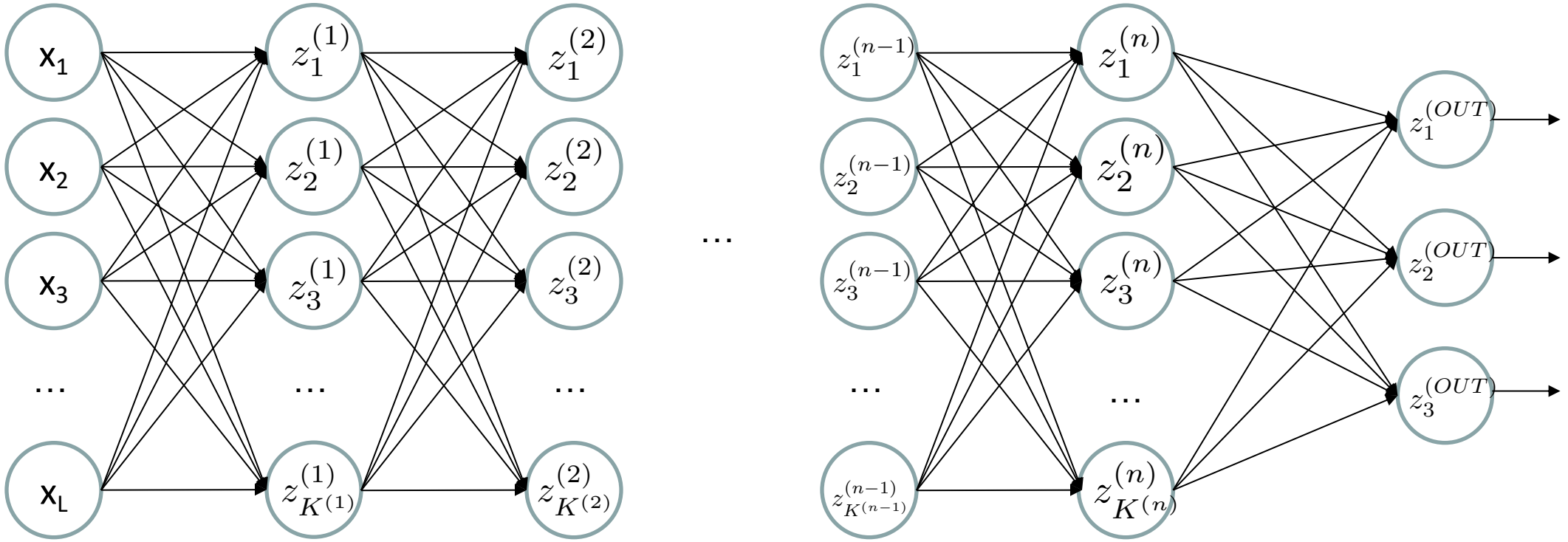
Shape (2, 1).
Weights to be learned.

Shape (1, 1).
Output of network.

Multi-Layer Neural Network

- Input to a layer: some *dim(x)*-dimensional input vector
- Output of a layer: some *dim(y)*-dimensional output vector
  - *dim(y)* is the number of neurons in the layer (1 output per neuron)
- Process of converting input to output:
  - Multiply the (1, *dim(x)*) input vector with a (*dim(x)*, *dim(y)*) weight vector. The result has shape (1, *dim(y)*).
  - Apply some non-linear function (e.g. sigmoid) to the result. The result still has shape (1, *dim(y)*).
- Big idea: Chain layers together
  - The input could come from a previous layer's output
  - The output could be used as the input to the next layer

Deep Neural Network

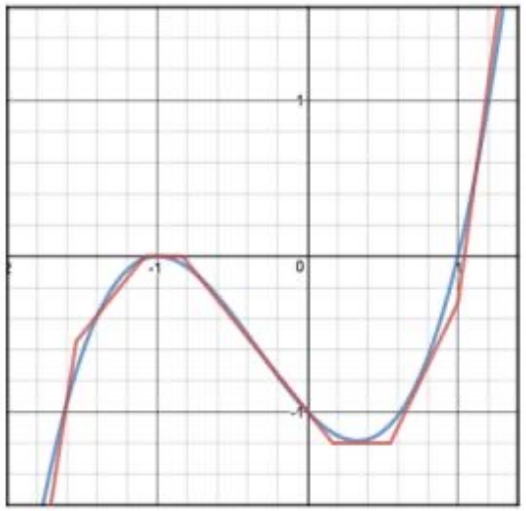


$$z_i^{(k)} = \sigma\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)$$

σ = nonlinear activation function

Universal approximation theorem

- Theorem (Universal Function Approximators). A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.



$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

$$Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x)$$

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." *Neural networks* 2.5 (1989): 359-366

Training: Backpropagation

Training Neural Networks

- Step 1: For each input in the training (sub)set *x*, predict a classification *y* using the current weights

$$\sigma(x \times W_{\text{layer } 1}) = h \qquad \sigma(h \times W_{\text{layer } 2}) = y$$

- Step 2: Compare predictions with the true *y* values, using a **loss function**
  - Higher value of loss function = bad model
  - Lower value of loss function = good model
  - Example: **zero-one loss**: count the number of misclassified inputs
  - Example: **log loss** (derived from maximum likelihood)
  - Example: **sum of squared errors** (more on this soon)
- Step 3: Use numerical method (e.g. gradient descent) to minimize loss
  - Loss is a function of the weights. Optimization goal: find weights that minimize loss

Optimization Procedure: Gradient Descent

- `init` w
- `for iter = 1, 2, …`

$$w \leftarrow w - \alpha \nabla \text{Loss} \quad (w)$$

- α : learning rate --- tweaking parameter that needs to be chosen carefully

Computing Gradients

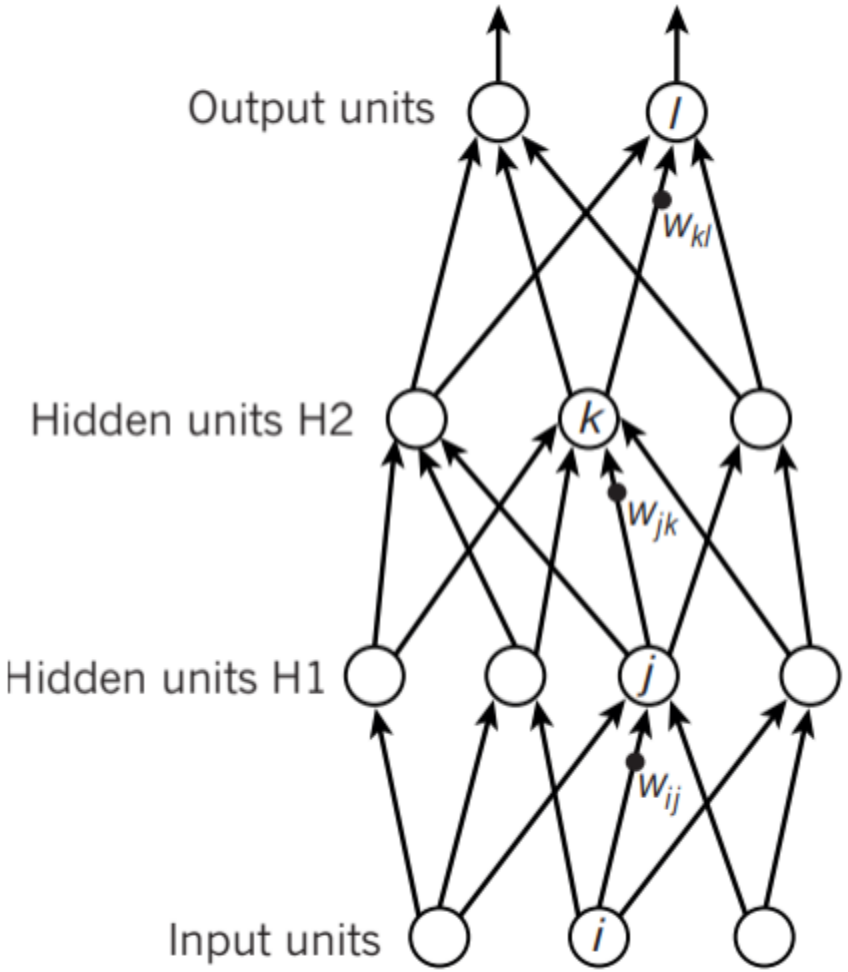
- How do we compute gradients of these loss functions?
  - Repeated application of the chain rule:

If $$f(x) = g(h(x))$$

Then $$f'(x) = g'(h(x))h'(x)$$

→ **Derivatives can be computed by following well-defined procedures**

Feed forward vs. Backpropagation



$$y_l = f(z_l)$$

$$z_l = \sum_{k \,\varepsilon\, \text{H2}} w_{kl}\, y_k$$

$$y_k = f(z_k)$$

$$z_k = \sum_{j \,\varepsilon\, \text{H1}} w_{jk}\, y_j$$

$$y_j = f(z_j)$$

$$z_j = \sum_{i \,\varepsilon\, \text{Input}} w_{ij}\, x_i$$

Compare outputs with correct answer to get error derivatives

$$\frac{\partial E}{\partial y_l} = y_l - t_l$$

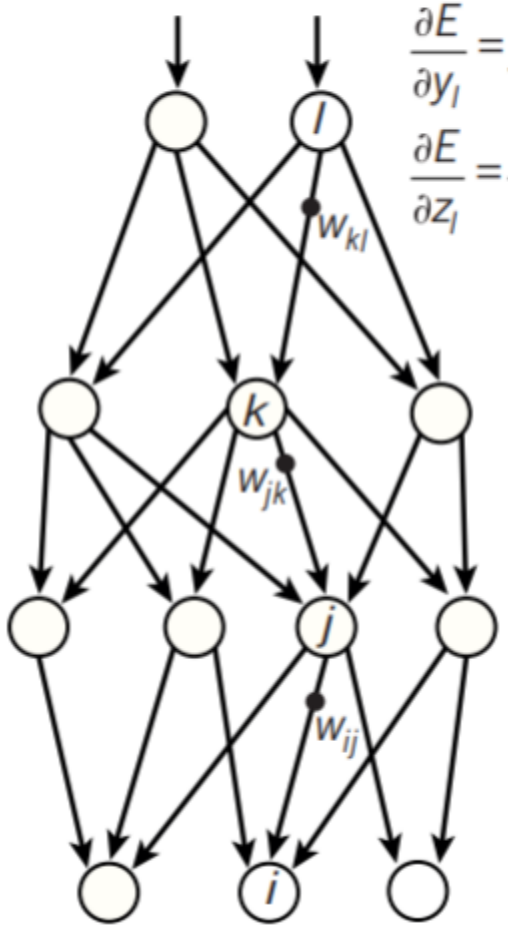
$$\frac{\partial E}{\partial z_l} = \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial z_l}$$

$$\frac{\partial E}{\partial y_k} = \sum_{l \,\varepsilon\, \text{out}} w_{kl}\frac{\partial E}{\partial z_l}$$

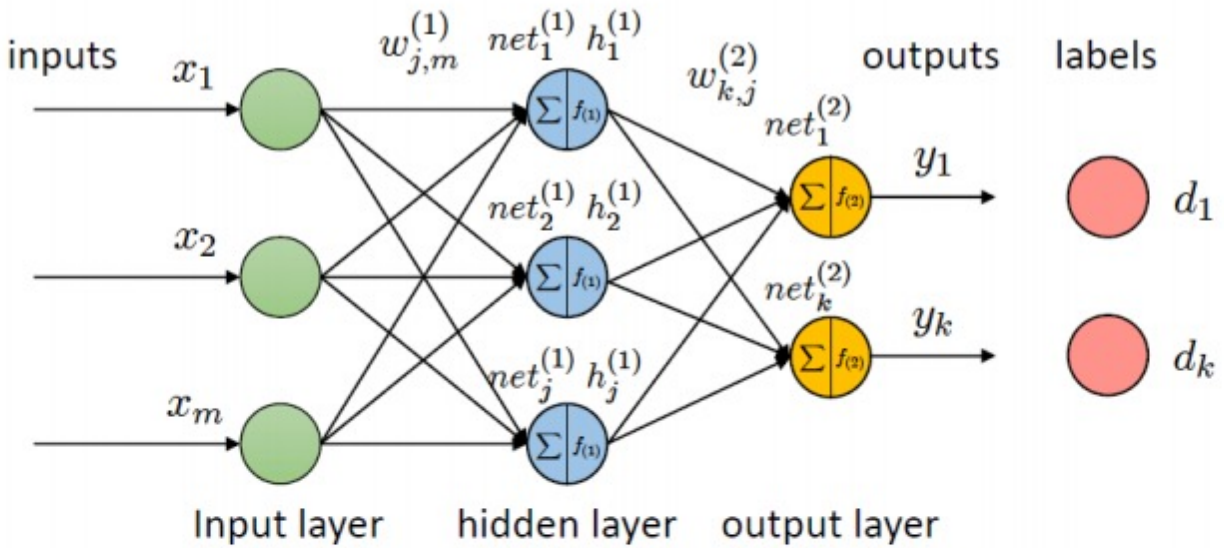
$$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial z_k}$$

$$\frac{\partial E}{\partial y_j} = \sum_{k \,\varepsilon\, \text{H2}} w_{jk}\frac{\partial E}{\partial z_k}$$

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}$$



Make a prediction



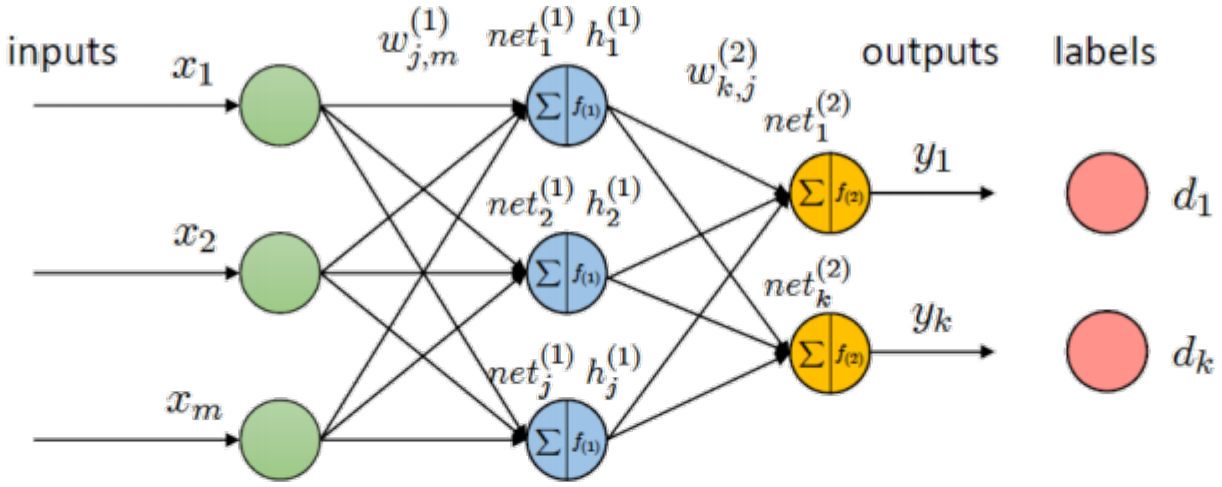
Two-layer feedforward neural network

Feed-forward prediction:

$$x = (x_1, \ldots, x_m) \longrightarrow h_j^{(1)} = f_{(1)}(net_j^{(1)}) = f_{(1)}\left(\sum_m w_{j,m}^{(1)} x_m\right) \longrightarrow y_k = f_{(2)}(net_k^{(2)}) = f_{(2)}\left(\sum_j w_{k,j}^{(1)} h_j^{(1)}\right)$$

where $net_j^{(1)} = \sum_m w_{j,m}^{(1)} x_m$ $\qquad$ $net_k^{(2)} = \sum_j w_{k,j}^{(2)} h_j^{(1)}$

Backpropagation



Two-layer feedforward neural network

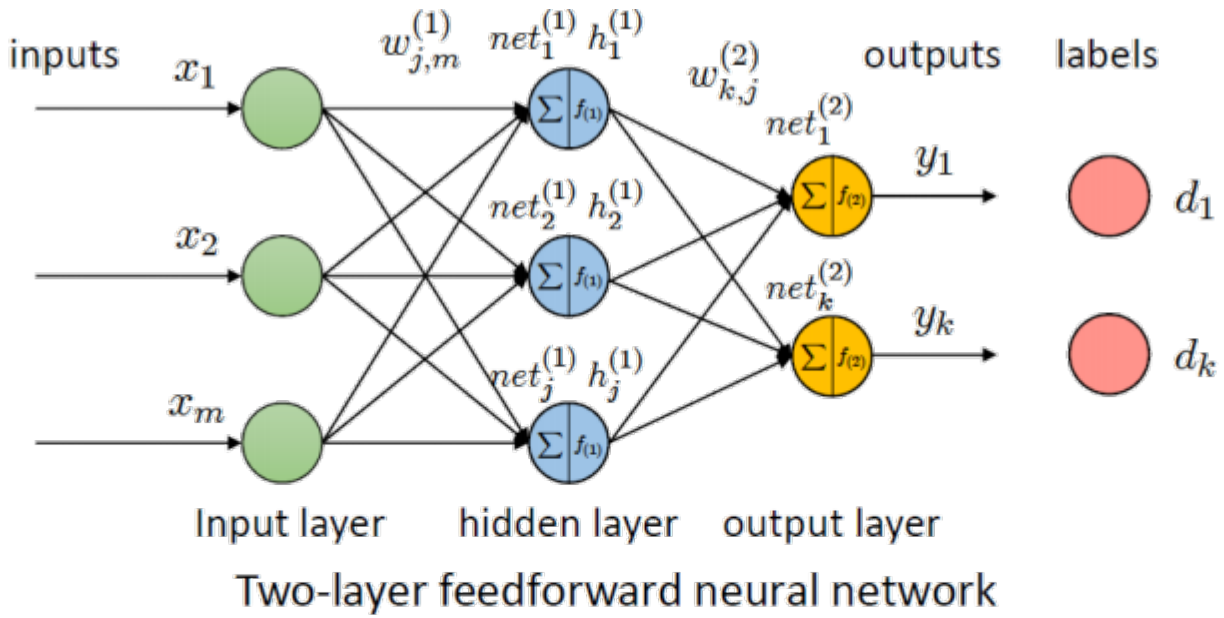
- Assume all the activation functions are sigmoid
- Error function $E = \frac{1}{2}\sum_k (y_k - d_k)^2$
- $\frac{\partial E}{\partial y_k} = y_k - d_k$
- $\frac{\partial y_k}{\partial w_{k,j}^{(2)}} = f'_{(2)}(net_k^{(2)})h_j^{(1)} = y_k(1-y_k)h_j^{(1)}$
- $\Rightarrow \frac{\partial E}{\partial w_{k,j}^{(2)}} = (y_k - d_k)y_k(1-y_k)h_j^{(1)}$
- $\Rightarrow w_{k,j}^{(2)} \leftarrow w_{k,j}^{(2)} - \eta(y_k - d_k)y_k(1-y_k)h_j^{(1)}$

Feed-forward prediction:

$$x = (x_1, \ldots, x_m) \longrightarrow h_j^{(1)} = f_{(1)}(net_j^{(1)}) = f_{(1)}\left(\sum_m w_{j,m}^{(1)} x_m\right) \longrightarrow y_k = f_{(2)}(net_k^{(2)}) = f_{(2)}\left(\sum_j w_{k,j}^{(2)} h_j^{(1)}\right) \longrightarrow y_k$$

where $net_j^{(1)} = \sum_m w_{j,m}^{(1)} x_m$ $\quad net_k^{(2)} = \sum_j w_{k,j}^{(2)} h_j^{(1)}$

Backpropagation (cont.)



Two-layer feedforward neural network

- Error function $E = \frac{1}{2}\sum_k (y_k - d_k)^2$
- $\frac{\partial E}{\partial y_k} = y_k - d_k$
- $\frac{\partial y_k}{\partial h_j^{(1)}} = y_k(1-y_k)w_{k,j}^{(2)}$
- $\frac{\partial h_j^{(1)}}{\partial w_{j,m}^{(1)}} = f'_{(1)}\left(net_j^{(1)}\right)x_m = h_j^{(1)}\left(1-h_j^{(1)}\right)x_m$
- $\Rightarrow \frac{\partial E}{\partial w_{j,m}^{(1)}} = h_j^{(1)}\left(1-h_j^{(1)}\right)\sum_k w_{k,j}^{(2)}(y_k - d_k)y_k(1-y_k)x_m$
- $\Rightarrow w_{j,m}^{(1)} \leftarrow w_{j,m}^{(1)} - \eta h_j^{(1)}\left(1-h_j^{(1)}\right)\sum_k w_{k,j}^{(2)}(y_k - d_k)y_k(1-y_k)x_m$

Feed-forward prediction:

$$h_j^{(1)} = f_{(1)}(net_j^{(1)}) = f_{(1)}\left(\sum_m w_{j,m}^{(1)}x_m\right) \qquad y_k = f_{(2)}(net_k^{(2)}) = f_{(2)}\left(\sum_j w_{k,j}^{(1)}h_j^{(1)}\right)$$

$$x = (x_1, \ldots, x_m) \longrightarrow h_j^{(1)} \longrightarrow y_k$$

where $\quad net_j^{(1)} = \sum_m w_{j,m}^{(1)}x_m \qquad net_k^{(2)} = \sum_j w_{k,j}^{(2)}h_j^{(1)}$

南方科技大学

STA303: Artificial Intelligence

Generalization

Fang Kong

https://fangkongx.github.io/

Part of slide credit: Stanford CS229

Intuition

- Recall in previous classes
  - We typically learn a model h_θ by minimizing the training loss/error
  - $J_\theta = \frac{1}{n}\sum_{i=1}^{n}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$
  - This is not the ultimate goal
- The ultimate goal
  - Sample a test data from the test distribution $\mathcal{D}$
  - Measure the model's error on the test data (test loss/error)

$$L(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[(y - h_\theta(x))^2]$$

  - Can be approximated by the average error on many sampled test examples

Challenges

- The test examples are unseen
  - Even though the training set is sampled from the same distribution $\mathcal{D}$, it can not guaranteed that the test error is close to the training error
  - Minimizing training error may not lead to a small test error
- Important concepts
  - Overfitting: the model predicts accurately on the training dataset but doesn't generalize well to other test examples
  - Underfitting: the training error is relatively large (typically the test error is also relatively large)

- How the test error is influenced by the learning procedure, especially the choice of model parameterizations?

How about fitting a linear model? (cont'd)

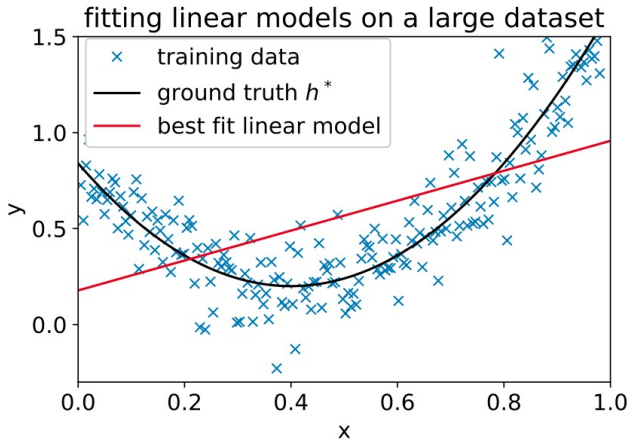


Figure 8.3: The best fit linear model on a much larger dataset still has a large training error.

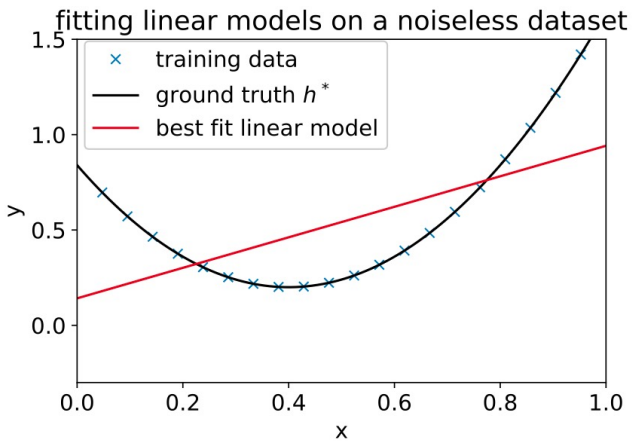


Figure 8.4: The best fit linear model on a noiseless dataset also has a large training/test error.

- Fundamental bottleneck: linear model family's inability to capture the structure in the data
- Define model bias: the test error even if we were to fit it to a very (say, infinitely) large training dataset

How about a 5th-degree polynomial? (cont'd)

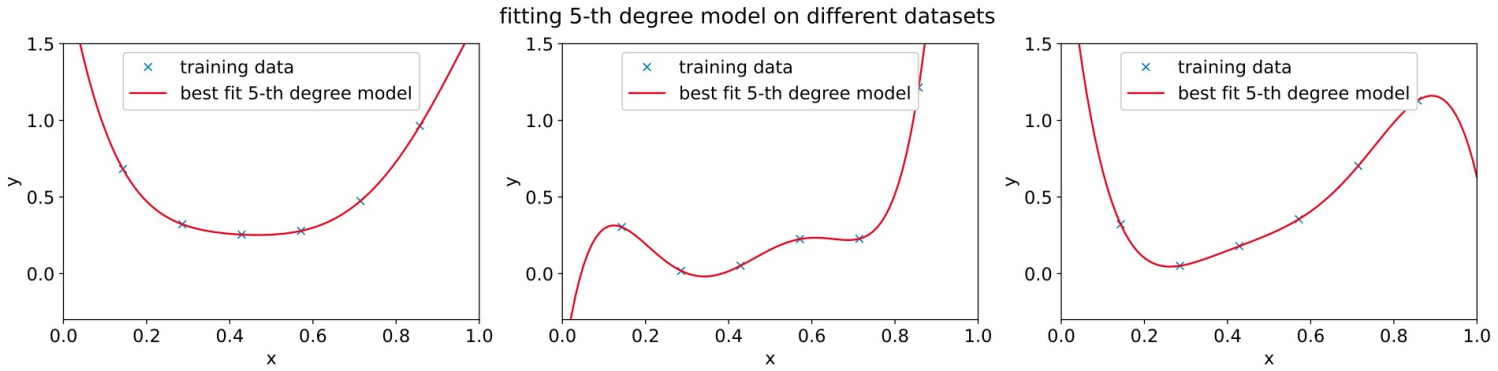


Figure 8.7: The best fit 5-th degree models on three different datasets generated from the same distribution behave quite differently, suggesting the existence of a large variance.

- Failure: fitting patterns in the data that happened to be present in the small, finite training set (NOT the real relationship between x and y)
- Define variance: the amount of variations across models learnt on multiple different training datasets (drawn from the same underlying distribution)

Bias-variance trade-off

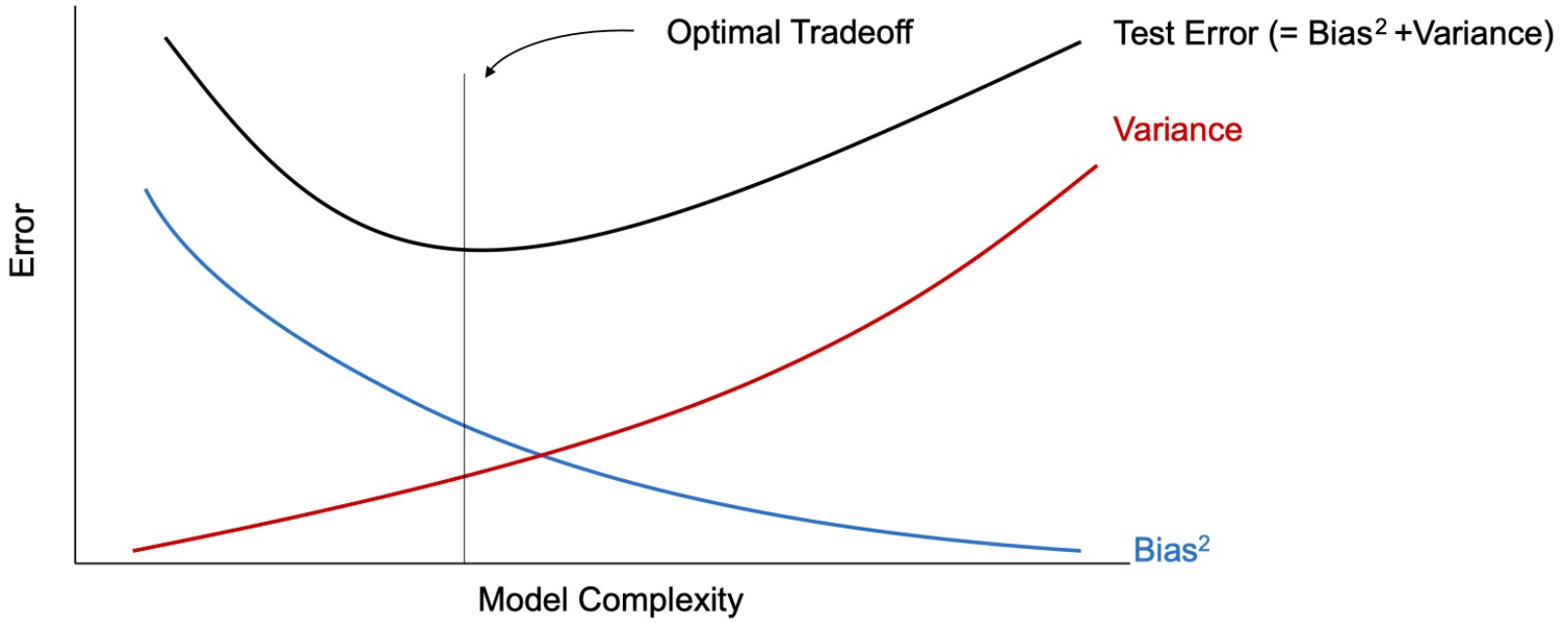


Figure 8.8: An illustration of the typical bias-variance tradeoff.

A mathematical decomposition (for regression)

Problem setting: regression

- Draw a training dataset $S = \{x^{(i)}, y^{(i)}\}_{i=1}^{n}$ such that $y^{(i)} = h^{\star}(x^{(i)}) + \xi^{(i)}$ where $\xi^{(i)} \in N(0, \sigma^2)$.
- Train a model on the dataset S, denoted by $\hat{h}_S$.
- Take a test example (x, y) such that $y = h^{\star}(x) + \xi$ where $\xi \sim N(0, \sigma^2)$, and measure the expected test error (averaged over the random draw of the training set S and the randomness of ξ)

$$\mathrm{MSE}(x) = \mathbb{E}_{S,\xi}[(y - \hat{h}_S(x))^2] \tag{8.2}$$

Decomposition

- $$\begin{aligned}\mathrm{MSE}(x) = \mathbb{E}[(y - h_S(x))^2] &= \mathbb{E}[(\xi + (h^\star(x) - h_S(x)))^2] \\ &= \mathbb{E}[\xi^2] + \mathbb{E}[(h^\star(x) - h_S(x))^2] \\ &= \sigma^2 + \mathbb{E}[(h^\star(x) - h_S(x))^2]\end{aligned}$$

- Define $h_{avg}(x) = \mathbb{E}_S[(h_S(x))]$
  - The model obtained by drawing an infinite number of datasets, training on them, and averaging their predictions on x

- $$\begin{aligned}\mathrm{MSE}(x) &= \sigma^2 + \mathbb{E}[(h^\star(x) - h_S(x))^2] \\ &= \sigma^2 + (h^\star(x) - h_{\mathrm{avg}}(x))^2 + \mathbb{E}[(h_{\mathrm{avg}} - h_S(x))^2] \\ &= \underbrace{\sigma^2}_{\text{unavoidable}} + \underbrace{(h^\star(x) - h_{\mathrm{avg}}(x))^2}_{\triangleq \text{ bias}^2} + \underbrace{\mathrm{var}(h_S(x))}_{\triangleq \text{ variance}}\end{aligned}$$

Sample complexity bounds

Useful lemmas

- **Lemma.** (The union bound). Let $A_1, A_2, \ldots, A_k$ be k different events (that may not be independent). Then

$$P(A_1 \cup \cdots \cup A_k) \leq P(A_1) + \ldots + P(A_k).$$

- **Lemma.** (Hoeffding inequality) Let $Z_1, \ldots, Z_n$ be n independent and identically distributed (iid) random variables drawn from a Bernoulli(ϕ) distribution. I.e., $P(Z_i = 1) = \phi$, and $P(Z_i = 0) = 1 - \phi$. Let $\hat{\phi} = (1/n) \sum_{i=1}^{n} Z_i$ be the mean of these random variables, and let any $\gamma > 0$ be fixed. Then

$$P(|\phi - \hat{\phi}| > \gamma) \leq 2 \exp(-2\gamma^2 n)$$

Problem setting

- To simplify, consider the classification problem with $y \in \{0,1\}$
- Training set $S = \{(x^i, y^i); i = 1,2, \dots, n\}$, drawn iid from $\mathcal{D}$

- For hypothesis h, define training error (empirical risk/error)

$$\hat{\varepsilon}(h) = \frac{1}{n}\sum_{i=1}^{n} 1\{h(x^{(i)}) \neq y^{(i)}\}$$

- Define the generalization error $\varepsilon(h) = P_{(x,y)\sim\mathcal{D}}(h(x) \neq y)$

One of PAC assumption: training and testing set are from the same D

Problem setting (cont'd)

- Consider the linear classification $h_\theta(x) = 1\{\theta^\top x \geq 0\}$
- Objective: minimize the training error

$$\hat{\theta} = \arg\min_\theta \hat{\varepsilon}(h_\theta)$$

$$\hat{h} = h_{\hat{\theta}}$$

empirical risk minimization

- In learning theory, it will be useful to abstract away from the specific parameterization of hypotheses
- Define the hypothesis class $\mathcal{H}$, for linear classification

$$\mathcal{H} = \{h_\theta : h_\theta(x) = 1\{\theta^T x \geq 0\}, \theta \in \mathbb{R}^{d+1}\}$$

Problem setting (cont'd)

- ERM becomes finding $\hat{h} = \arg\min_{h \in \mathcal{H}} \hat{\varepsilon}(h)$
- For simplicity, first consider the finite hypothesis set

$$\mathcal{H} = \{h_1, \ldots, h_k\}$$

- Now, show the guarantee for the generalization error of $\hat{h}$
  - 1. $\forall h, \hat{\varepsilon}(h)$ is a reliable estimate of $\varepsilon(h)$
  - 2. $\hat{h}$ guarantees good generalization error

Guarantee for a fixed hypothesis function

- Fix any hypothesis function $h_i \in \mathcal{H}$
- Define $Z_j = 1\{h_i(x^j) \neq y^j\}$
- The training error is

$$\hat{\varepsilon}(h_i) = \frac{1}{n}\sum_{j=1}^{n} Z_j$$

- The empirical mean of n random variables with expectation $\varepsilon(h_i)$
- Applying Hoeffding inequality,

$$P(|\varepsilon(h_i) - \hat{\varepsilon}(h_i)| > \gamma) \leq 2\exp(-2\gamma^2 n)$$

Guarantee for any hypothesis function

- $$\begin{aligned} P(\exists h \in \mathcal{H}.|\varepsilon(h_i) - \hat{\varepsilon}(h_i)| > \gamma) &= P(A_1 \cup \cdots \cup A_k) \\ &\leq \sum_{i=1}^{k} P(A_i) \\ &\leq \sum_{i=1}^{k} 2\exp(-2\gamma^2 n) \\ &= 2k\exp(-2\gamma^2 n) \end{aligned}$$

- Thus $$\begin{aligned} P(\neg\exists h \in \mathcal{H}.|\varepsilon(h_i) - \hat{\varepsilon}(h_i)| > \gamma) &= P(\forall h \in \mathcal{H}.|\varepsilon(h_i) - \hat{\varepsilon}(h_i)| \leq \gamma) \\ &\geq 1 - 2k\exp(-2\gamma^2 n) \end{aligned}$$

Corollaries

- How large must n be before we can guarantee that with probability at least $1 - \delta$, training error will be within γ of generalization error? (sample complexity)

- What is the distance between the training error and generalization error with training set size n and confidence δ?

Guarantee for the output hypothesis function

- Recall $\hat{h} = \arg\min_{h \in \mathcal{H}} \hat{\varepsilon}(h)$
- Define the best hypothesis is $h^* = \arg\min_{h \in \mathcal{H}} \varepsilon(h)$
- Then
$$\begin{aligned} \varepsilon(\hat{h}) &\leq \hat{\varepsilon}(\hat{h}) + \gamma \\ &\leq \hat{\varepsilon}(h^*) + \gamma \\ &\leq \varepsilon(h^*) + 2\gamma \end{aligned}$$
- If uniform convergence occurs, then the generalization error of h is at most 2γ worse than the best possible hypothesis in $\mathcal{H}$!

Theorem of generalization error

- **Theorem.** Let $|\mathcal{H}| = k$, and let any n, δ be fixed. Then with probability at least $1 - \delta$, we have that

$$\varepsilon(\hat{h}) \leq \left(\min_{h \in \mathcal{H}} \varepsilon(h) \right) + 2\sqrt{\frac{1}{2n} \log \frac{2k}{\delta}}.$$

- Explanation of bias/variance
  - If we switch to a larger function class $\mathcal{H}' \supseteq \mathcal{H}$
  - The first term decreases: lower bias
  - The second term increases as k increases: higher variance

Corollary of sample complexity

- **Corollary.** Let $|\mathcal{H}| = k$, and let any δ, γ be fixed. Then for $\varepsilon(\hat{h}) \leq \min_{h \in \mathcal{H}} \varepsilon(h) + 2\gamma$ to hold with probability at least $1 - \delta$, it suffices that

$$\begin{aligned} n &\geq \frac{1}{2\gamma^2} \log \frac{2k}{\delta} \\ &= O\left(\frac{1}{\gamma^2} \log \frac{k}{\delta}\right), \end{aligned}$$

Extension to infinite $\mathcal{H}$: Intuition

- Usually the hypothesis set is infinite
  - For example, the linear function set contains a infinite number of parameters

- Suppose $\mathcal{H}$ is parameterized by d real numbers
- The computer uses 64 bits to represent a floating point number
- $\mathcal{H}$ contains 2^{64d} different hypotheses
- Existing results show that with fixed γ, δ

$$n \geq O\left(\frac{1}{\gamma^2}\log\frac{2^{64d}}{\delta}\right) = O\left(\frac{d}{\gamma^2}\log\frac{1}{\delta}\right) = O_{\gamma,\delta}(d)$$

VC dimension

- Shatter

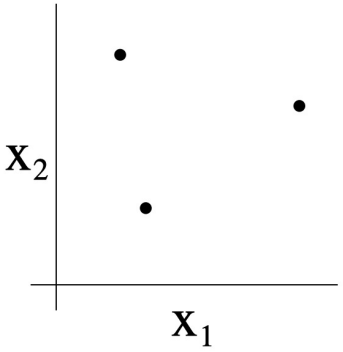
Given a set $S = \{x^{(i)}, \ldots, x^{(\mathbf{D})}\}$ (no relation to the training set) of points $x^{(i)} \in \mathcal{X}$, we say that $\mathcal{H}$ **shatters** S if $\mathcal{H}$ can realize any labeling on S. I.e., if for any set of labels $\{y^{(1)}, \ldots, y^{(\mathbf{D})}\}$, there exists some $h \in \mathcal{H}$ so that $h(x^{(i)}) = y^{(i)}$ for all $i = 1, \ldots \mathbf{D}$.

- VC dimension

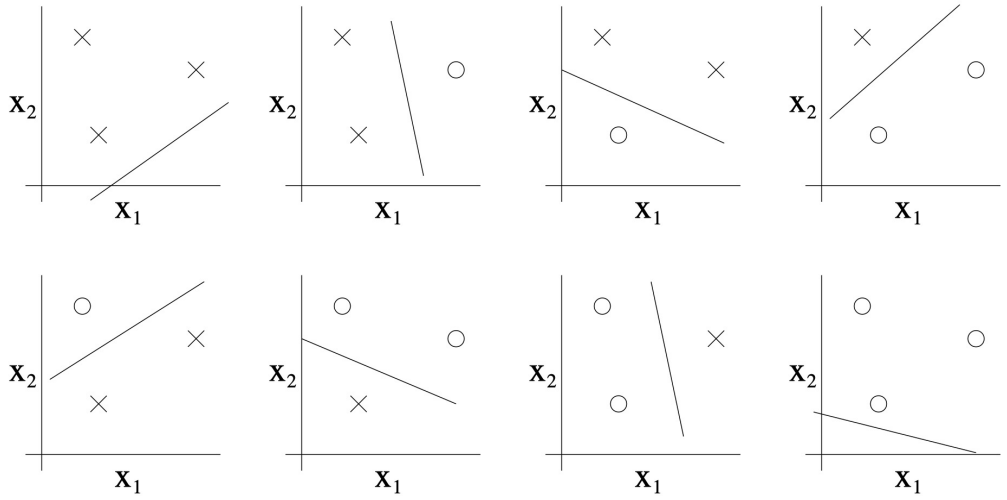
Given a hypothesis class $\mathcal{H}$, we then define its **Vapnik-Chervonenkis dimension**, written $\mathrm{VC}(\mathcal{H})$, to be the size of the largest set that is shattered by $\mathcal{H}$. (If $\mathcal{H}$ can shatter arbitrarily large sets, then $\mathrm{VC}(\mathcal{H}) = \infty$.)

VC dimension: illustration

- Can the set $\mathcal{H}$ of linear classifiers in two dimensions shatter the set below?

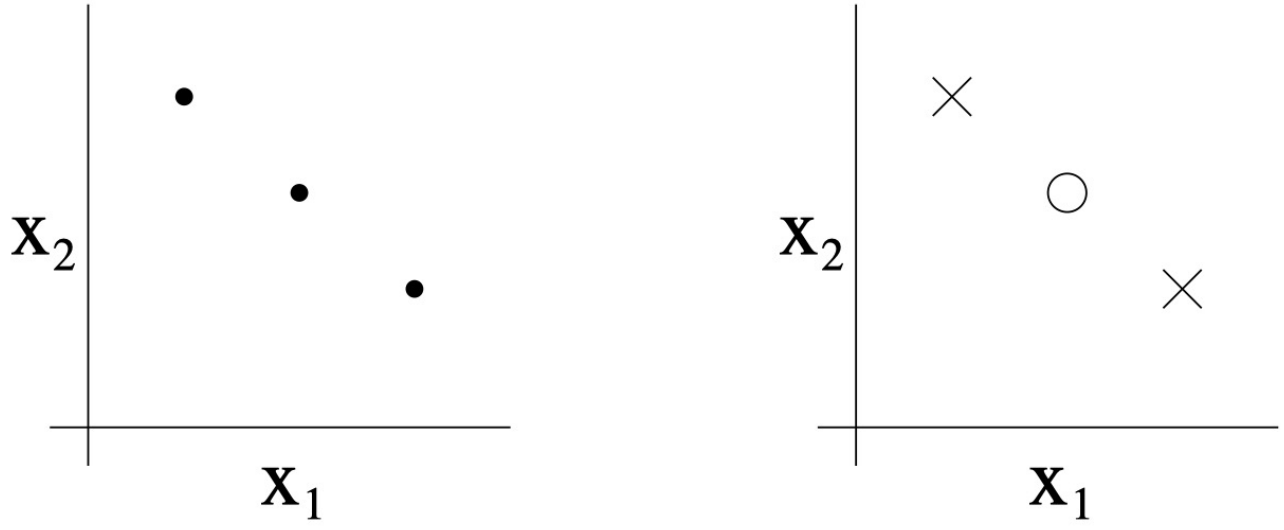


- For any labeling, $\mathcal{H}$ can correctly classify



VC dimension: illustration (cont'd)

- In order to prove that VC($\mathcal{H}$) is at least D, we need to show only that there's **at least one** set of size D that H can shatter (not every set of size D)



Convergence results

- **Theorem.** Let $\mathcal{H}$ be given, and let $\mathbf{D} = \mathrm{VC}(\mathcal{H})$. Then with probability at least $1 - \delta$, we have that for all $h \in \mathcal{H}$,

$$|\varepsilon(h) - \hat{\varepsilon}(h)| \leq O\left(\sqrt{\frac{\mathbf{D}}{n}\log\frac{n}{\mathbf{D}} + \frac{1}{n}\log\frac{1}{\delta}}\right).$$

  Thus, with probability at least $1 - \delta$, we also have that:

$$\varepsilon(\hat{h}) \leq \varepsilon(h^*) + O\left(\sqrt{\frac{\mathbf{D}}{n}\log\frac{n}{\mathbf{D}} + \frac{1}{n}\log\frac{1}{\delta}}\right).$$

Usually the VC dimension is roughly linear in the number of parameters

- **Corollary.** For $|\varepsilon(h) - \hat{\varepsilon}(h)| \leq \gamma$ to hold for all $h \in \mathcal{H}$ (and hence $\varepsilon(\hat{h}) \leq \varepsilon(h^*) + 2\gamma$) with probability at least $1 - \delta$, it suffices that $n = O_{\gamma,\delta}(\mathbf{D})$.

南方科技大学

STA303: Artificial Intelligence

Deep Reinforcement Learning

Fang Kong

https://fangkongx.github.io/

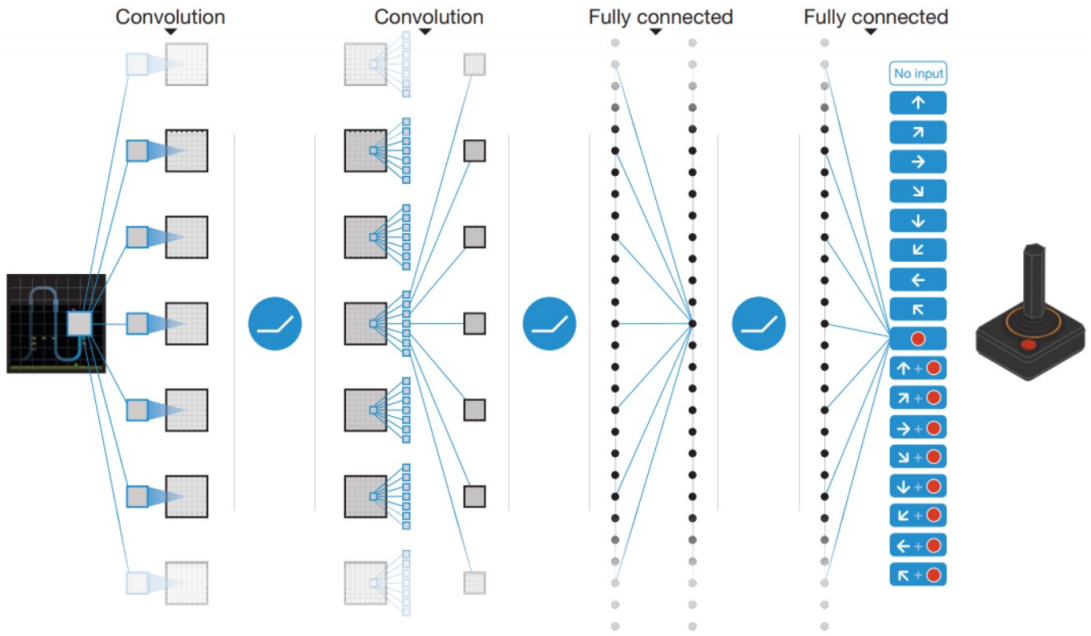
Part of slide credits: Weinan Zhang

Outline

- Deep RL – Value methods
- Deep RL – Policy methods

Value methods: DQN

- Deep Q-Network (DQN)
  - Uses a deep neural network to approximate Q(s,a)
    - → Replaces the Q-table with a parameterized function for scalability
  - The network takes state s as input, outputs Q-values for all actions a simultaneously



Volodymyr Mnih, Koray Kavukcuoglu, David Silver et al. Playing Atari with Deep Reinforcement Learning. NIPS 2013 workshop.

DQN (cont.)

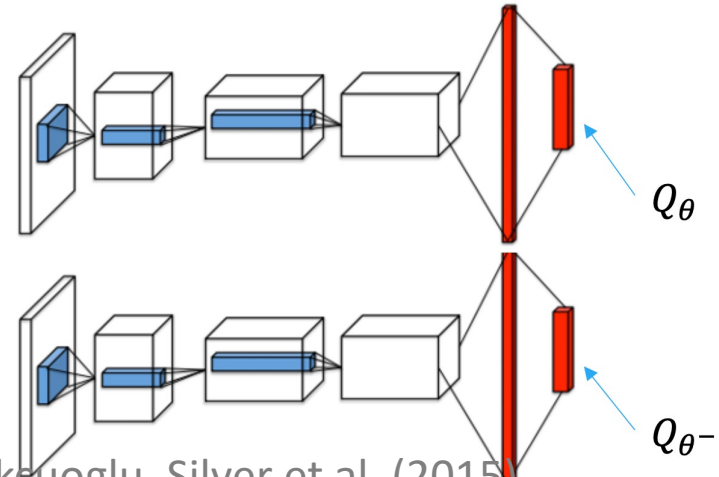
- Intuition: Use a deep neural network to approximate Q(s,a)
  - Instability arises in the learning process
    - Samples $\{(s_t, a_t, s_{t+1}, r_t)\}$ are collected sequentially and do not satisfy the i.i.d. assumption
    - Frequent updates of Q(s,a) cause instability
- Solutions: Experience replay
  - Store transitions $e_t = (s_t, a_t, s_{t+1}, r_t)$ in a replay buffer D
    Sample uniformly from D to reduce sample correlation
  - Dual network architecture: Use an evaluation network and a target network for improved stability

"Human-Level Control Through Deep Reinforcement Learning", Mnih, Kavukcuoglu, Silver et al. (2015)

Target network

- Target network $Q_{\theta^-}(s, a)$
  - Maintains a copy of the Q-network with older parameters θ^-
  - Parameters θ^- are updated periodically (every C steps) to match the evaluation network
- Loss Function (at iteration i)

$$L_i(\theta_i) = \mathbb{E}_{s_t, a_t, s_{t+1}, r_t, p_t \sim D}\left[\frac{1}{2}\omega_t\left(r_t + \gamma \max_{a'} Q_{\theta_i^-}(s_{t+1}, a') - Q_{\theta_i}(s_t, a_t)\right)^2\right]$$



"Human-Level Control Through Deep Reinforcement Learning", Mnih, Kavukcuoglu, Silver et al. (2015)

DQN training procedure

- Collect transitions using an ε-greedy exploration policy
  - Store $\{(s_t, a_t, s_{t+1}, r_t)\}$ into the replay buffer
- Sample a minibatch of k transitions from the buffer
- Update networks:
  - Compute the target using the sampled transitions
  - Update the evaluation network Q_θ
  - Every C steps, synchronize the target network Q_{θ^-} with the evaluation network

"Human-Level Control Through Deep Reinforcement Learning", Mnih, Kavukcuoglu, Silver et al. (2015)

Overestimation in Q-Learning

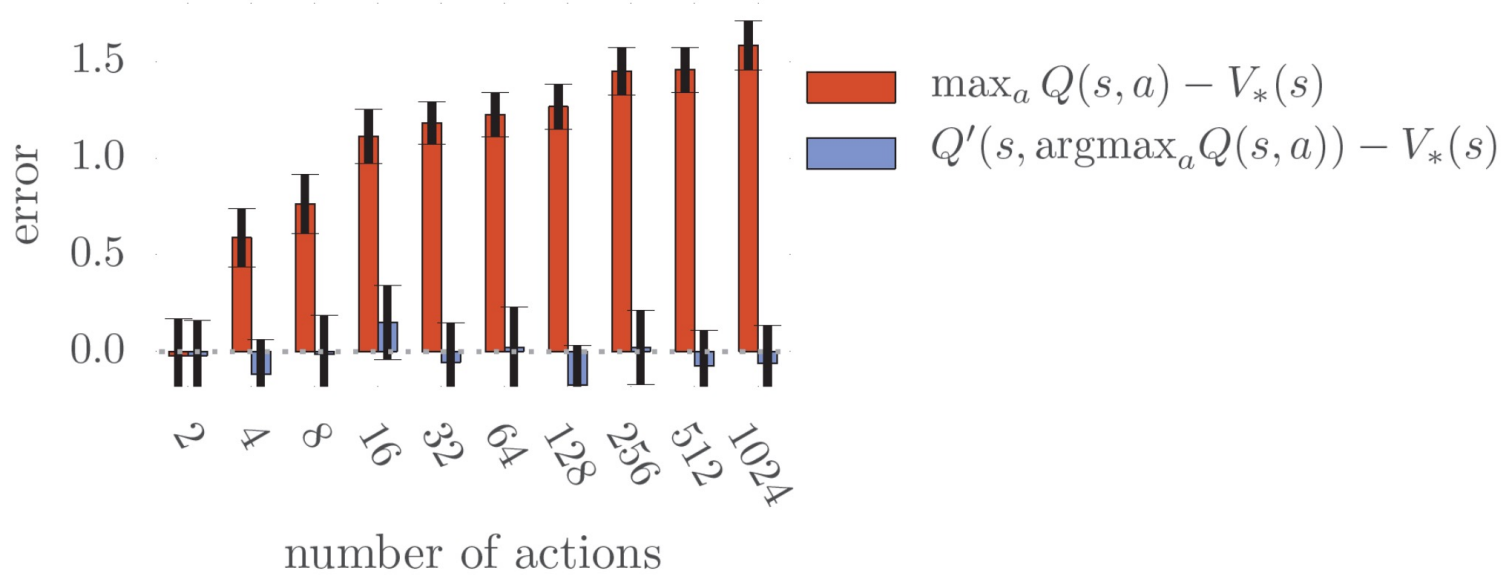
- Q-function overestimation
  - The target value is computed as: $y_t = r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a')$
  - The max operator leads to increasingly larger Q-values, potentially exceeding the true value
- Cause of overestimation

$$\max_{a' \in A} Q_{\theta'}(s_{t+1}, a') = Q_{\theta'}(s_{t+1}, \arg\max_{a'} Q_{\theta'}(s_{t+1}, a'))$$

  - The chosen action might be overestimated due to Q-function error

Degree of overestimation in DQN

- Overestimation increases with the number of candidate actions



- A separately trained Q'-function is used as a reference

Double DQN

- Uses two separate networks for action selection and value estimation, respectively.

$$\text{DQN} \quad y_t = r_t + \gamma Q_\theta(s_{t+1}, \arg\max_{a'} Q_\theta(s_{t+1}, a'))$$

$$\text{Double DQN} \quad y_t = r_t + \gamma Q_{\theta'}(s_{t+1}, \arg\max_{a'} Q_\theta(s_{t+1}, a'))$$

"Double Reinforcement Learning with Double Q-Learning", van Hasselt et al. (2016)

Dueling DQN

- Assume the action-value function follows a distribution:

$$Q(s,a) \sim \mathcal{N}(\mu, \sigma)$$

- Then: $V(s) = \mathbb{E}[Q(s,a)] = \mu$ $\quad$ $Q(s,a) = \mu + \varepsilon(s,a)$

- How do we describe $\varepsilon(s,a)$?

$$\varepsilon(s,a) = Q(s,a) - V(s)$$

  - This term is also known as the Advantage function

"Dueling Network Architectures for Deep Reinforcement Learning", Wang et al. (2016)

Dueling DQN (cont.)

- Advantage function

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

$$Q^{\pi}(s,a) = \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$$

$$V^{\pi}(s) = \mathbb{E}_{a \sim \pi(s)}[Q^{\pi}(s,a)]$$

- Different forms of advantage aggregation

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \max_{a' \in |A|} A(s,a';\theta,\alpha)\right)$$

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

Deep RL – Policy-based methods

Review: The policy gradient theorem

- The policy gradient theorem generalizes the derivation of likelihood ratios to the multi-step MDP setting.
- It replaces the immediate reward r_t with the expected long-term return $Q^{\pi}(s,a)$.

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\pi_\theta}\left[\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} Q^{\pi_\theta}(s,a)\right]$$

Policy Gradient in a Single-Step MDP

- Consider a simple single-step Markov Decision Process (MDP)
  - The initial state is drawn from a distribution: $s \sim d(s)$
  - The process terminates after one action, yielding a reward r_{sa}
- Expected Value of the Policy

$$J(\theta) = \mathbb{E}_{\pi_\theta}[r] = \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(a|s) r_{sa}$$

$$\frac{\partial J(\theta)}{\partial \theta} = \sum_{s \in S} d(s) \sum_{a \in A} \frac{\partial \pi_\theta(a|s)}{\partial \theta} r_{sa}$$

Likelihood Ratio Trick

- Use the identity:

$$\frac{\partial \pi_\theta(a|s)}{\partial \theta} = \pi_\theta(a|s) \frac{1}{\pi_\theta(a|s)} \frac{\partial \pi_\theta(a|s)}{\partial \theta} = \pi_\theta(a|s) \frac{\partial \log \pi_\theta(a|s)}{\partial \theta}$$

- The gradient of the expected return can be written as:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[r] = \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(a|s) r_{sa}$$

$$\frac{\partial J(\theta)}{\partial \theta} = \sum_{s \in S} d(s) \sum_{a \in A} \frac{\partial \pi_\theta(a|s)}{\partial \theta} r_{sa}$$

$$= \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(a|s) \frac{\partial \log \pi_\theta(a|s)}{\partial \theta} r_{sa}$$

$$= \mathbb{E}_{\pi_\theta}\left[\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} r_{sa}\right]$$

Can be approximated sampling s fr d(s) and a fro

Policy network gradient

- For stochastic policies, the probability of selecting an action is typically modeled using a softmax function:

$$\pi_\theta(a|s) = \frac{e^{f_\theta(s,a)}}{\sum_{a'} e^{f_\theta(s,a')}}$$

  - $f_\theta(s,a)$ is a score function (e.g., logits) for the state-action pair
  - Parameterized by θ, often realized via a neural network

- Gradient of the log-form

$$\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} = \frac{\partial f_\theta(s,a)}{\partial \theta} - \frac{1}{\sum_{a'} e^{f_\theta(s,a')}} \sum_{a''} e^{f_\theta(s,a'')} \frac{\partial f_\theta(s,a'')}{\partial \theta}$$

$$= \frac{\partial f_\theta(s,a)}{\partial \theta} - \mathbb{E}_{a' \sim \pi_\theta(a'|s)} \left[\frac{\partial f_\theta(s,a')}{\partial \theta} \right]$$

Policy network gradient (cont.)

- Gradient of the log-form

$$\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} = \frac{\partial f_\theta(s,a)}{\partial \theta} - \mathbb{E}_{a' \sim \pi_\theta(a'|s)} \left[\frac{\partial f_\theta(s,a')}{\partial \theta} \right]$$

- Gradient of the policy network

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\pi_\theta} \left[\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} Q^{\pi_\theta}(s,a) \right]$$

$$= \mathbb{E}_{\pi_\theta} \left[\left(\underbrace{\frac{\partial f_\theta(s,a)}{\partial \theta}}_{\text{Back propagation}} - \mathbb{E}_{a' \sim \pi_\theta(a'|s)} \left[\underbrace{\frac{\partial f_\theta(s,a')}{\partial \theta}}_{\text{Back propagation}} \right] \right) Q^{\pi_\theta}(s,a) \right]$$

1. Rollout
2. Train another network

Comparison: DQN v.s. Policy gradient

- Q-Learning:
  - Learns a Q-value function $Q_\theta(s,a)$ parameterized by θ
  - Objective: Minimize the TD error

$$J(\theta) = \mathbb{E}_{\pi'}\left[\frac{1}{2}\left(r_t + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a') - Q_\theta(s_t, a_t)\right)^2\right]$$

$$\theta \leftarrow \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$$

$$= \theta + \alpha \mathbb{E}_{\pi'}\left[\left(r_t + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a') - Q_\theta(s_t, a_t)\right)\frac{\partial Q_\theta(s,a)}{\partial \theta}\right]$$

Comparison: DQN v.s. Policy gradient

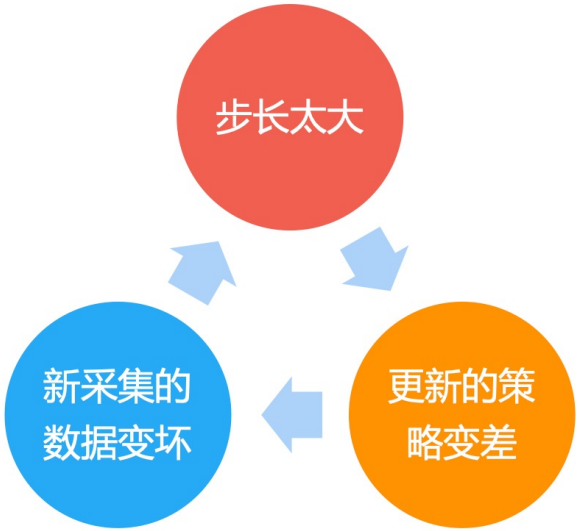
- Q-Learning:
  - Learns a Q-value function $Q_\theta(s, a)$ parameterized by θ
  - Objective: Minimize the TD error
- Policy gradient
  - Learns a policy $\pi_\theta(a \mid s)$ directly, parameterized by θ
  - Objective: Maximize the expected return directly

$$\max_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[Q^{\pi_\theta}(s, a)]$$

$$\theta \leftarrow \theta + \alpha \frac{\partial J(\theta)}{\partial \theta} = \theta + \alpha \mathbb{E}_{\pi_\theta}\left[\frac{\partial \log \pi_\theta(a|s)}{\partial \theta} Q^{\pi_\theta}(s, a)\right]$$

Limitations of policy gradient methods

- Learning rate (step size) selection is challenging in policy gradient algorithms
  - Since the data distribution changes as the policy updates, a previously good learning rate may become ineffective.
  - A poor choice of step size can significantly degrade performance:
    - Too large → policy diverges or collapses
    - Too small → slow convergence or stagnation



Trust Region Policy Optimization (TRPO)

- Two forms of the optimization objective
  - Form 1: Trajectory-based objective $J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[\sum_t \gamma^t r(s_t, a_t)]$
  - Form 2: State-value-based objective $J(\theta) = \mathbb{E}_{s_0 \sim p_\theta(s_0)}[V^{\pi_\theta}(s_0)]$

Optimization gap of the objective function

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[\sum_t \gamma^t r(s_t, a_t)]$$
$$J(\theta) = \mathbb{E}_{s_0 \sim p_\theta(s_0)}[V^{\pi_\theta}(s_0)]$$

- New policy θ' and old policy θ

$$J(\theta') - J(\theta) = J(\theta') - \mathbb{E}_{s_0 \sim p(s_0)}[V^{\pi_\theta}(s_0)]$$

$$= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}[\sum_{t=0} \gamma^t A^{\pi_\theta}(s_t, a_t)]$$

Sampling inconvenience

$$A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)$$

Importance sampling

$$A^{\pi_\theta}(s_t, a_t) = Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)$$

$$J(\theta') - J(\theta)$$

$$= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)} \left[\sum_{t=0}^{\infty} \gamma^t A^{\pi_\theta}(s_t, a_t) \right]$$

$$= \sum_t \mathbb{E}_{s_t \sim p_{\theta'}(s_t)} [\mathbb{E}_{a_t \sim \pi_{\theta'}(a_t|s_t)} [\gamma^t A^{\pi_\theta}(s_t, a_t)]]$$

$$= \sum_t \mathbb{E}_{s_t \sim p_{\theta'}(s_t)} [\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} [\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t)]]$$

$p_{\theta'}$, approximation

Importance sampling

TRPO Policy Constraint

- Use KL divergence to constrain policy update magnitude:

$$\theta' \leftarrow \arg\max_{\theta'} \sum_t \mathbb{E}_{s_t \sim p_\theta(s_t)}[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t)]]$$

$$\text{such that } \mathbb{E}_{s_t \sim p_\theta(s_t)}[D_{KL}(\pi_{\theta'}(a_t|s_t) \parallel \pi_\theta(a_t|s_t))] \leq \epsilon$$

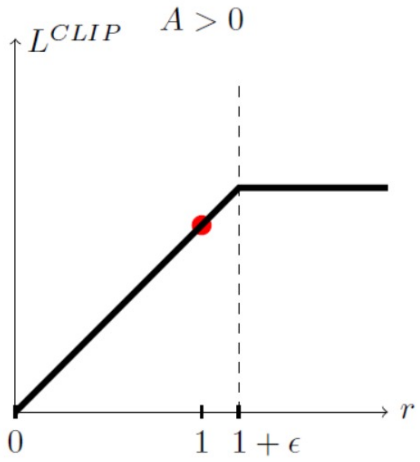
Proximal Policy Optimization (PPO)

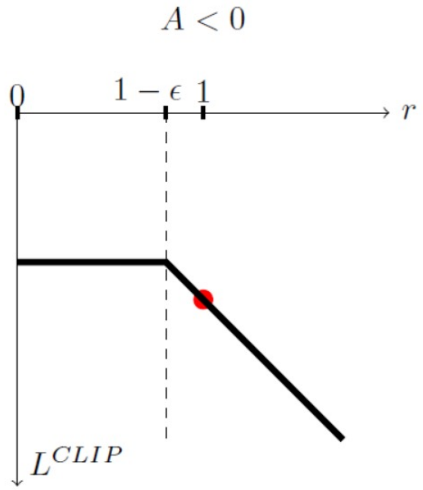
- Clipped Surrogate Objective

conservative policy iteration

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right]$$

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$





Construct the lower bound: $L^{CLIP}(\theta) \leq L^{CPI}(\theta)$

Equivalent at r=1: $L^{CLIP}(\theta) = L^{CPI}(\theta)$

南方科技大学

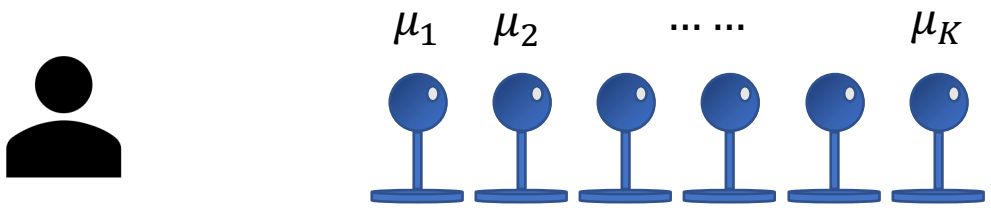
STA303: Artificial Intelligence

Introduction to Multi-armed Bandits

Fang Kong

https://fangkongx.github.io/

Multi-armed bandits (MAB)



- A player and K arms
  - Items, products, movies, companies, …
- Each arm a_j has an unknown reward distribution P_j with unknown mean μ_j
  - CTR, preference value, …
- In each round $t = 1,2, \ldots$:
  - The agent selects an arm $A_t \in \{1,2, \ldots, K\}$
  - Observes reward $X_t \sim P_{A_t}$
    - Click information, satisfaction, …

Assume P_j is supported on [0,1]

Objective

- Maximize the expected cumulative reward in T rounds

$$\mathbb{E}\left[\sum_{t=1}^{T} X_t\right] = \mathbb{E}\left[\sum_{t=1}^{T} \mu_{A_t}\right]$$

- Minimize the regret in T rounds
  - Denote $j^* \in \operatorname{argmax}_j \mu_j$ as the best arm

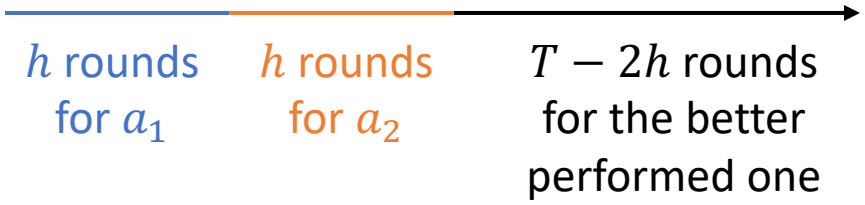
$$Reg(T) = T \cdot \mu_{j^*} - \mathbb{E}\left[\sum_{t=1}^{T} \mu_{A_t}\right]$$

Explore-then-commit (ETC) [Garivier et al., 2016]

- There are $K = 2$ arms (choices/plans/...)
- Suppose
  - $\mu_1 > \mu_2$
  - $\Delta = \mu_1 - \mu_2$

A/B testing

- Explore-then-commit (ETC) algorithm
  - Select each arm h times
  - Find the empirically best arm A
  - Choose $A_t = A$ for all remaining rounds



A soft version: ε-greedy

- For each round t
  - $\varepsilon_t \in (0,1)$
  - With probability ε_t, exploration (uniformly random select arms)
  - With probability $1 - \varepsilon_t$, exploitation (select the best performed arm so far)

- When $\varepsilon_t = \min\left\{1, \frac{c}{t\Delta^2}\right\}$, $Reg(T) = O\left(\frac{\log T}{\Delta}\right)$

Demo: https://cse442-17f.github.io/LinUCB/

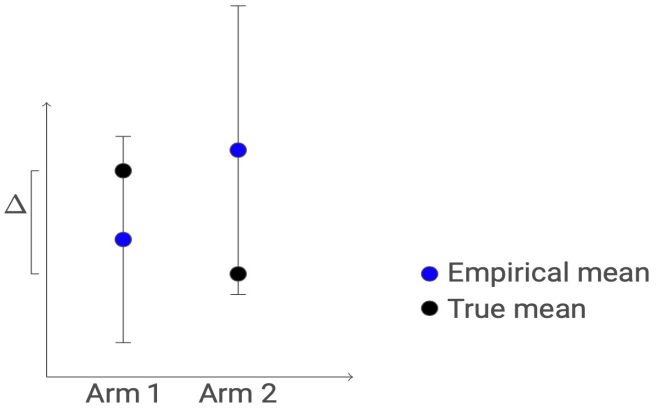
Upper confidence bound (UCB) [Auer et al., 2002]

- With high probability $\geq 1 - \delta$ By Hoeffding's inequality

$$\mu_j \in \left[\hat{\mu}_j - \sqrt{\frac{\log 1/\delta}{T_j}}, \hat{\mu}_j + \sqrt{\frac{\log 1/\delta}{T_j}}\right]$$

Sample mean

Number of selections of a_j



- Optimism: Believe arms have higher rewards, encourage exploration
  - The UCB value represents the reward estimates
- For each round t, select the arm

$$A(t) \in \text{argmax}_{j \in [K]} \left\{\hat{\mu}_j + \sqrt{\frac{\log 1/\delta}{T_j(t)}}\right\}$$

Upper confidence bound (UCB)

Exploitation

Exploration