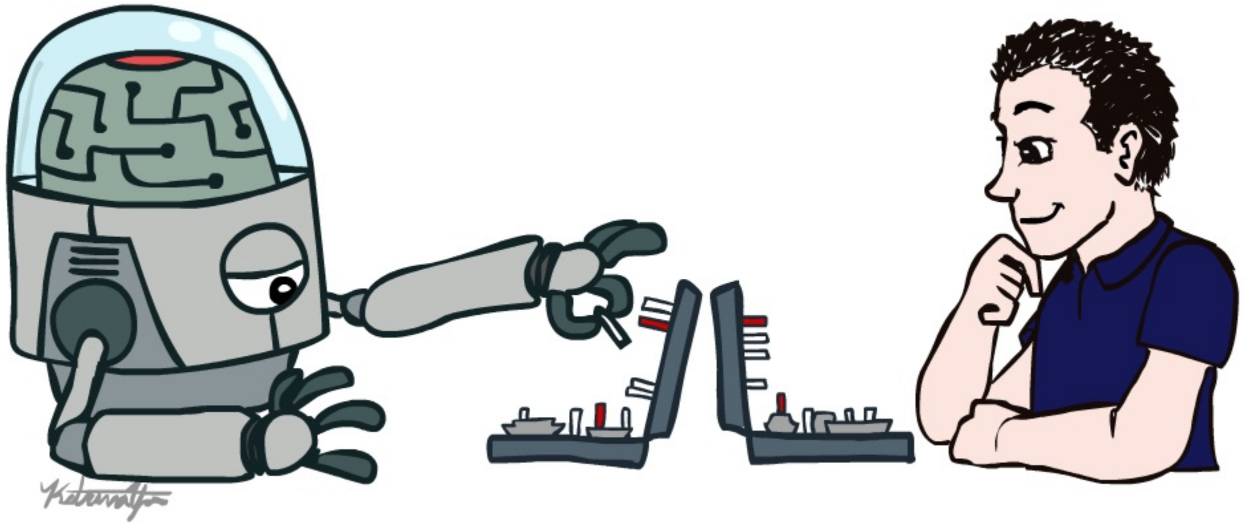
CS 188: Artificial Intelligence

Final Exam Review



Instructor: Nicholas Tomlin

University of California, Berkeley

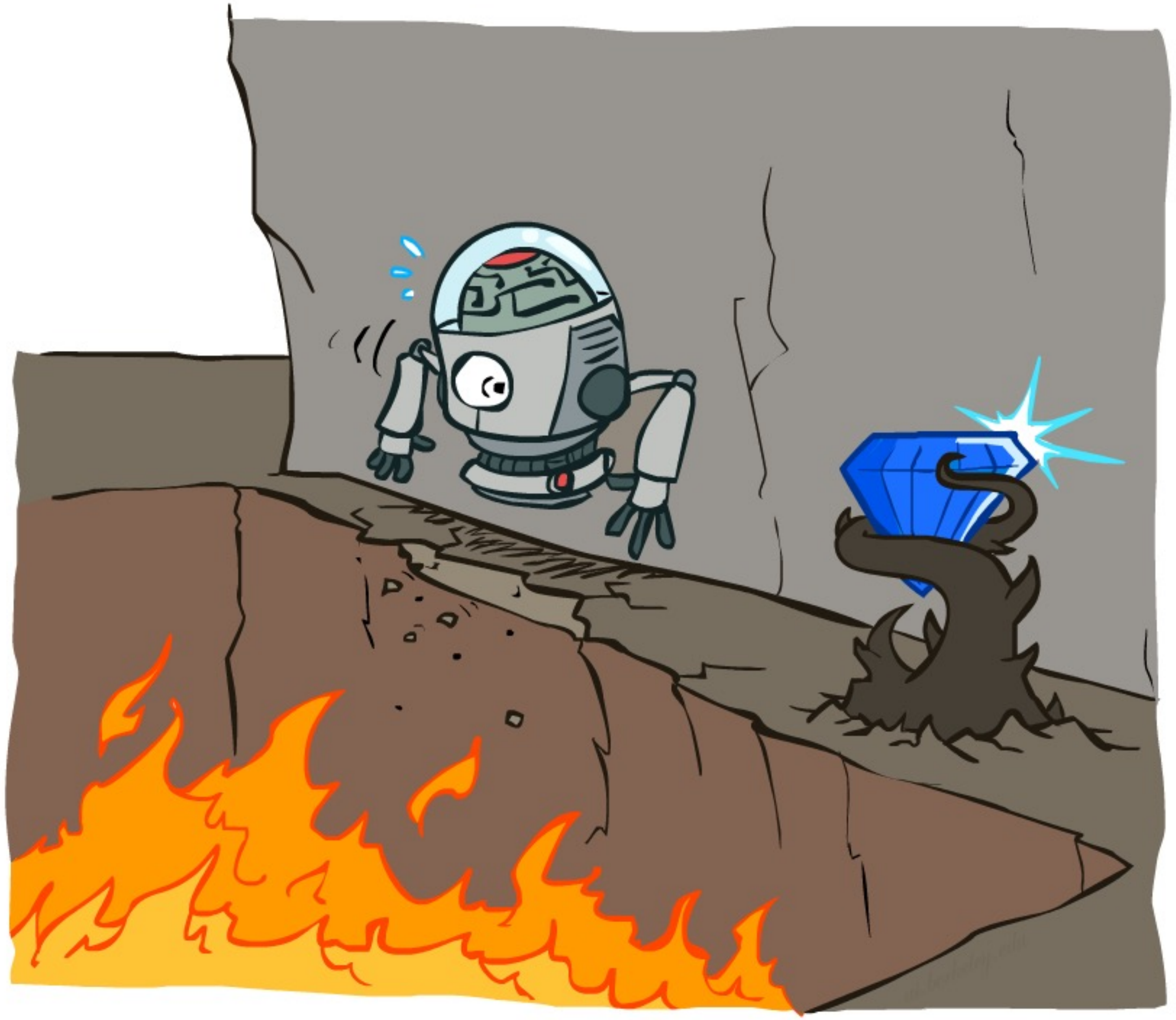
(slides adapted from Dan Klein, Pieter Abbeel, Anca Dragan, Stuart Russell)

Announcements

- All assignments **must** be submitted by 11:59PM tonight; no late submissions, no grace period beyond this point
- Exam tomorrow (Thursday, August 10th):
  - Wheeler 150
  - 7-10PM, but please show up no later than 6:45PM
  - Get enough sleep, drink enough water, etc.
- Get +1% extra credit on the exam by filling out course evaluations: https://course-evaluations.berkeley.edu

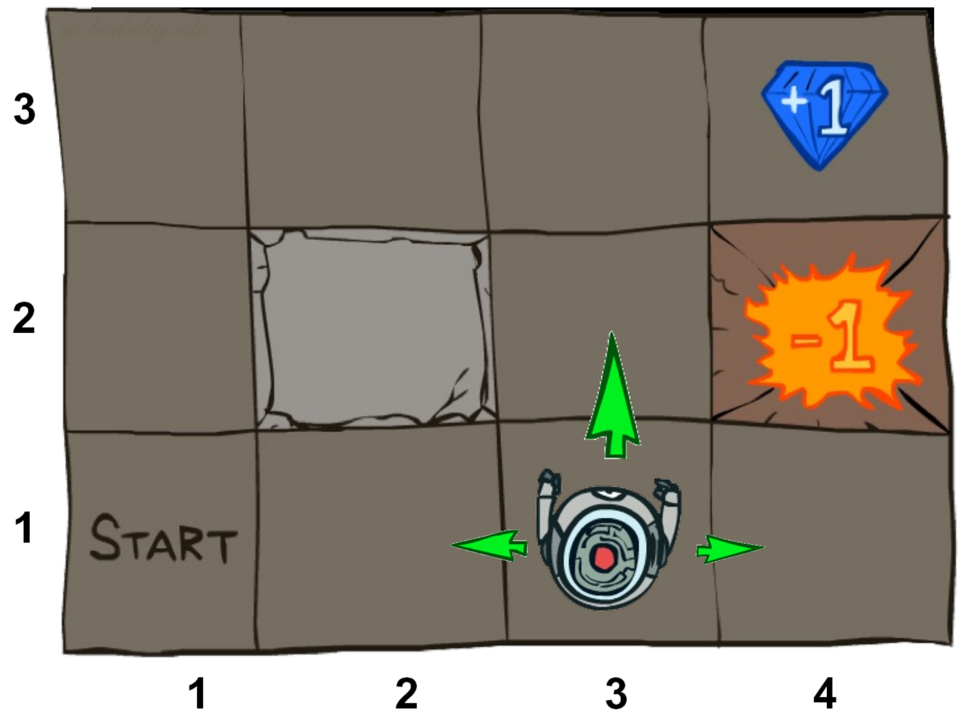
- Today's plan: cover as much material as possible, focused on the second half of the course.

Markov Decision Processes



Markov Decision Processes

- An MDP is defined by:
  - A set of states s ∈ S
  - A set of actions a ∈ 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



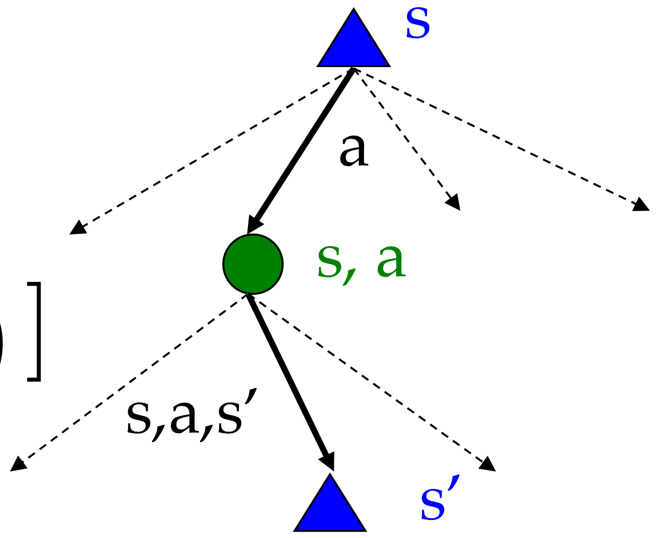
- We care about:
  - Policy = choice of actions for each state
  - Utility = sum of (discounted) rewards

Values of States: Bellman Equation

- Recursive definition of value:

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

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

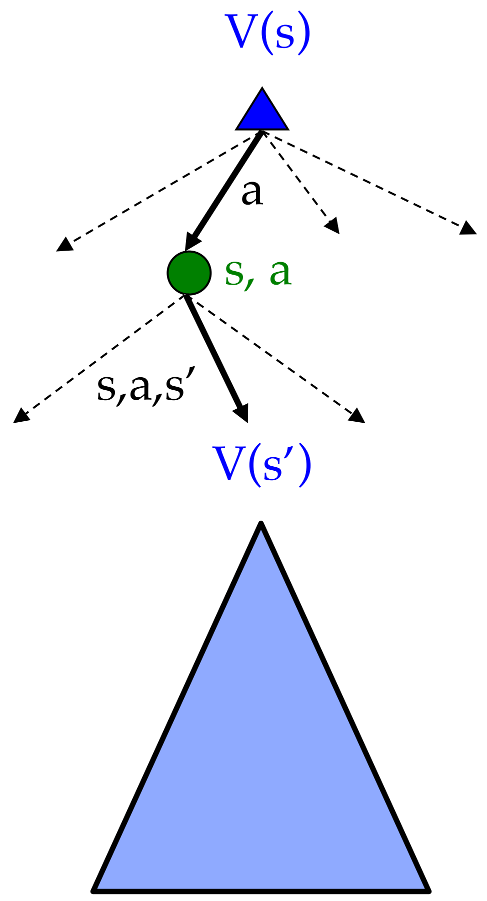


$$V^*(s) = \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$

Value Iteration

- Bellman equations characterize the optimal values:

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



- Value iteration computes them:

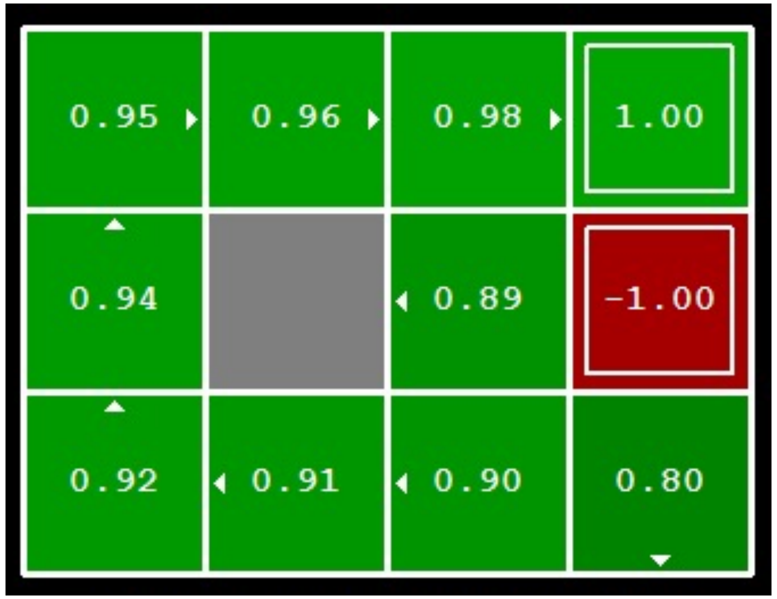
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

"Bellman Update"

- Value iteration is just a fixed point solution method

Policy Extraction 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)

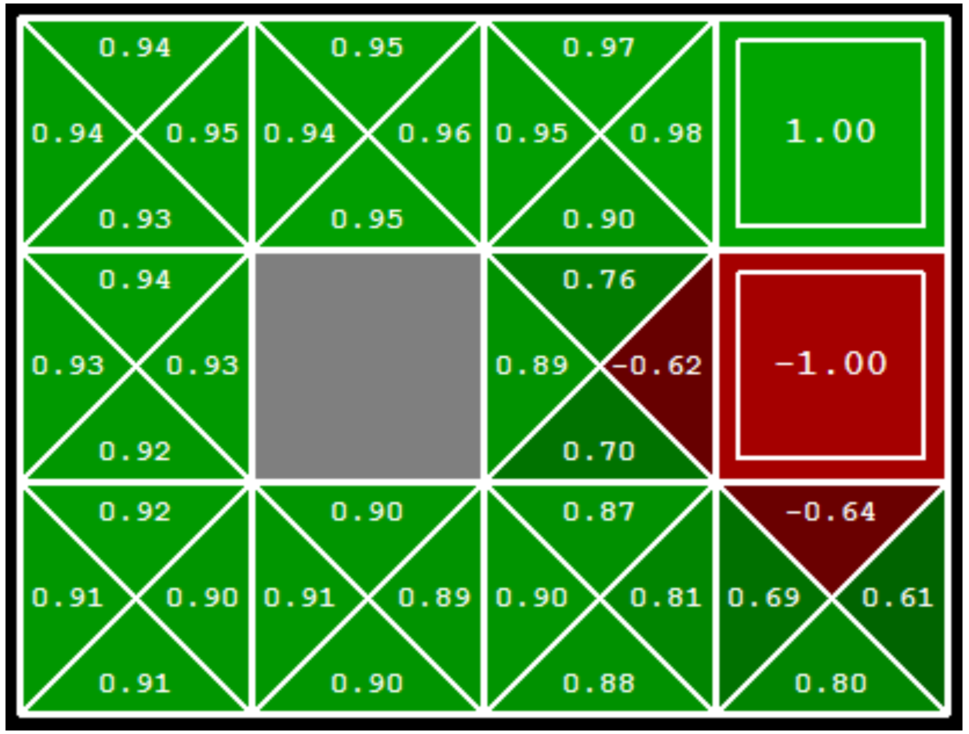


$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

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

Policy Extraction 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 Evaluation

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

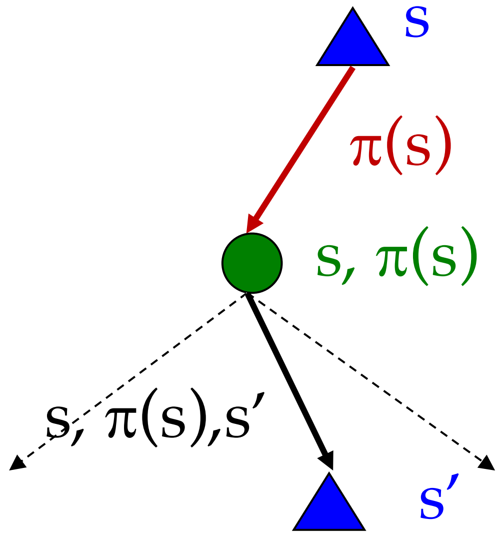
- Idea 1: Turn recursive Bellman equations into updates (like value iteration)

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

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

- Efficiency: O(S^2) per iteration

- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve the system of equations



Policy Iteration

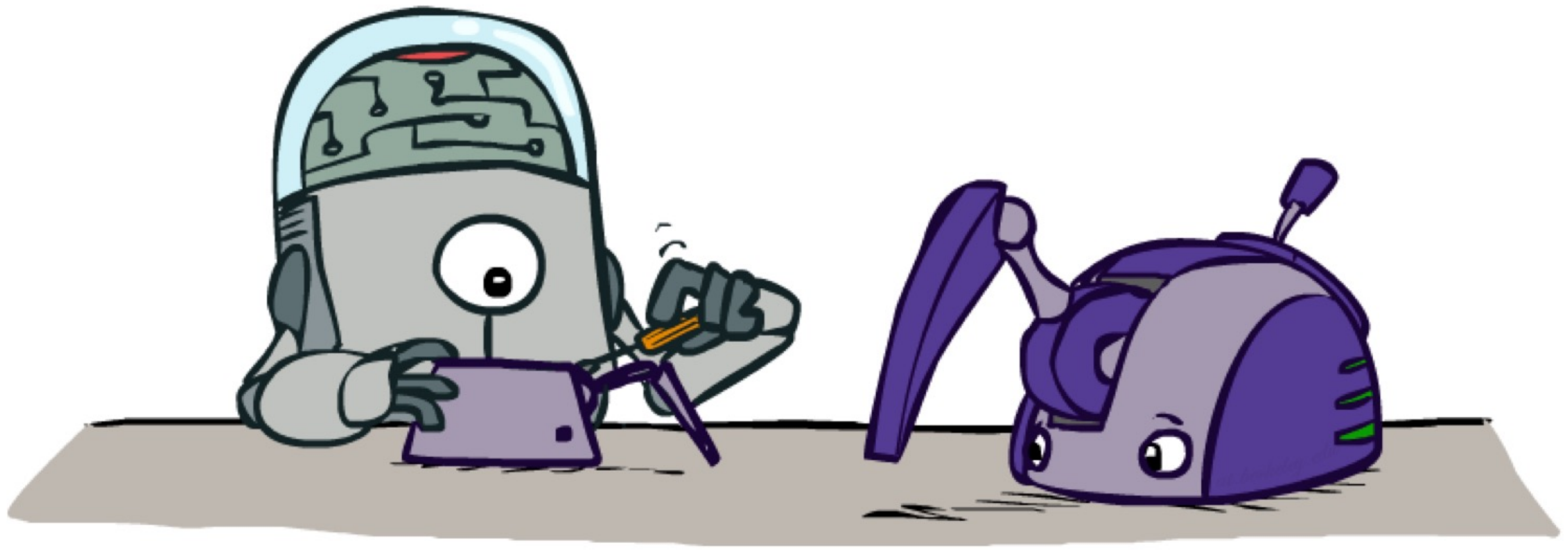
- Evaluation: For fixed current policy π, find values with policy evaluation:
  - Iterate until values converge:

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

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

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

Reinforcement Learning



Map of Reinforcement Learning

Known MDP: Offline Solution

Goal	Technique
Compute V*, Q*, π*	Value / policy iteration
Evaluate a fixed policy π	Policy evaluation

Unknown MDP: Model-Based

Goal	Technique
Compute V*, Q*, π*	VI/PI on approx. MDP
Evaluate a fixed policy π	PE on approx. MDP

Unknown MDP: Model-Free

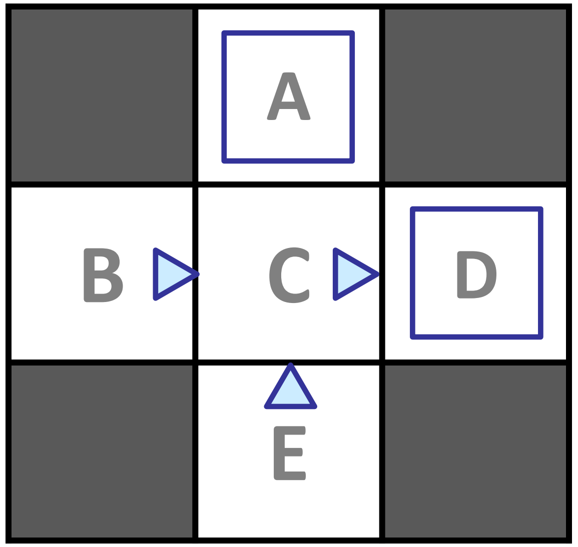
Goal	Technique
Compute V*, Q*, π*	Q-learning
Evaluate a fixed policy π	Value Learning

Direct Evaluation

- Goal: Compute values for each state under π
- Idea: Average together observed sample values
  - Act according to π
  - Every time you visit a state, write down what the sum of discounted rewards turned out to be
  - Average those samples
- This is called direct evaluation

Direct Evaluation

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

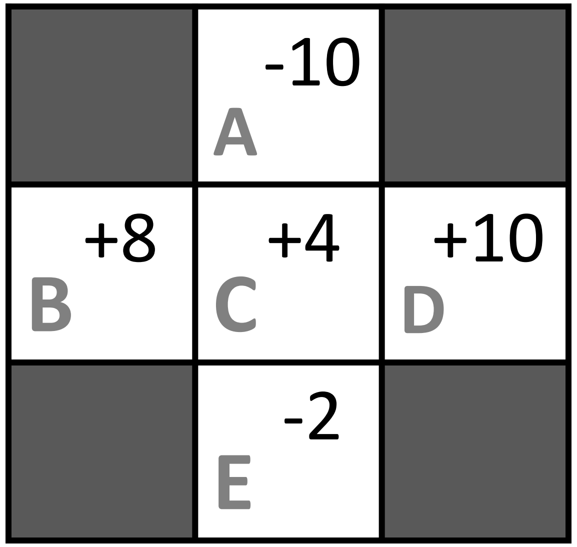
Episode 3

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

Episode 4

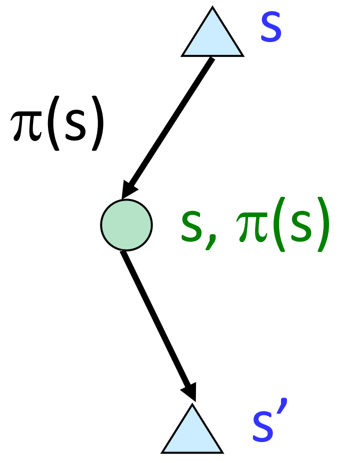
E, north, C, -1
C, east, A, -1
A, exit, x, -10

Output Values



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

Q-Learning

- Q-Learning: sample-based Q-value iteration

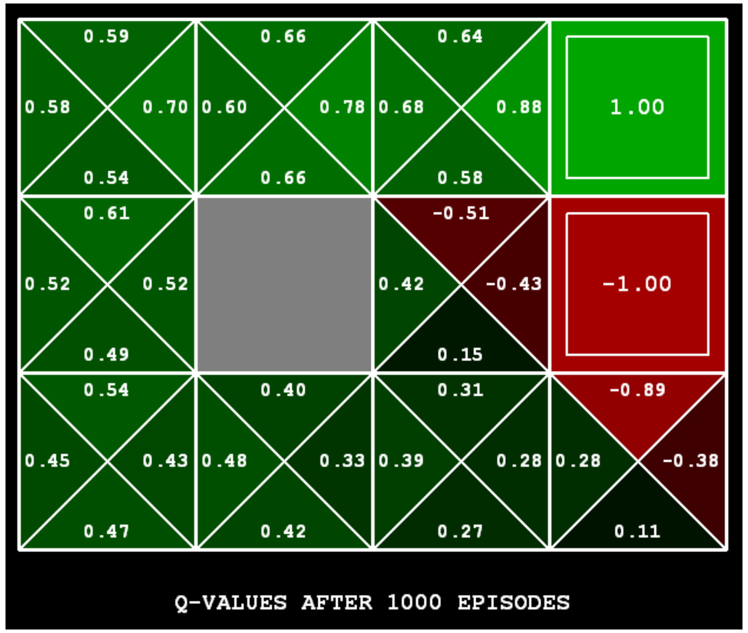
$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

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

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

  - Incorporate the new estimate into a running average:

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



Approximate Q-Learning

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

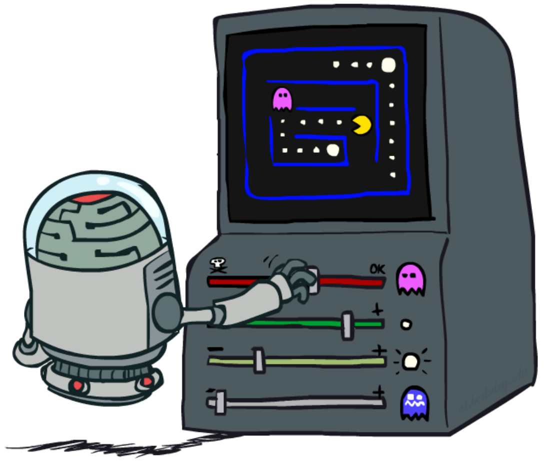
- Q-learning with linear Q-functions:

$$\text{transition} = (s, a, r, s')$$

$$\text{difference} = \left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s,a)$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\text{difference}]$$ Exact Q's

$$w_i \leftarrow w_i + \alpha\,[\text{difference}]\, f_i(s,a)$$ Approximate Q's



- Intuitive interpretation:
  - Adjust weights of active features
  - E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state's features

- Formal justification: online least squares

How to Explore?

- Several schemes for forcing exploration
  - Simplest: random actions (ε-greedy)
    - Every time step, flip a coin
    - With (small) probability ε, act randomly
    - With (large) probability 1-ε, act on current policy
  - Problems with random actions?
    - You do eventually explore the space, but keep thrashing around once learning is done
    - One solution: lower ε over time
    - Another solution: exploration functions

Exploration Functions

- When to explore?
  - Random actions: explore a fixed amount
  - Better idea: explore areas whose badness is no (yet) established, eventually stop exploring

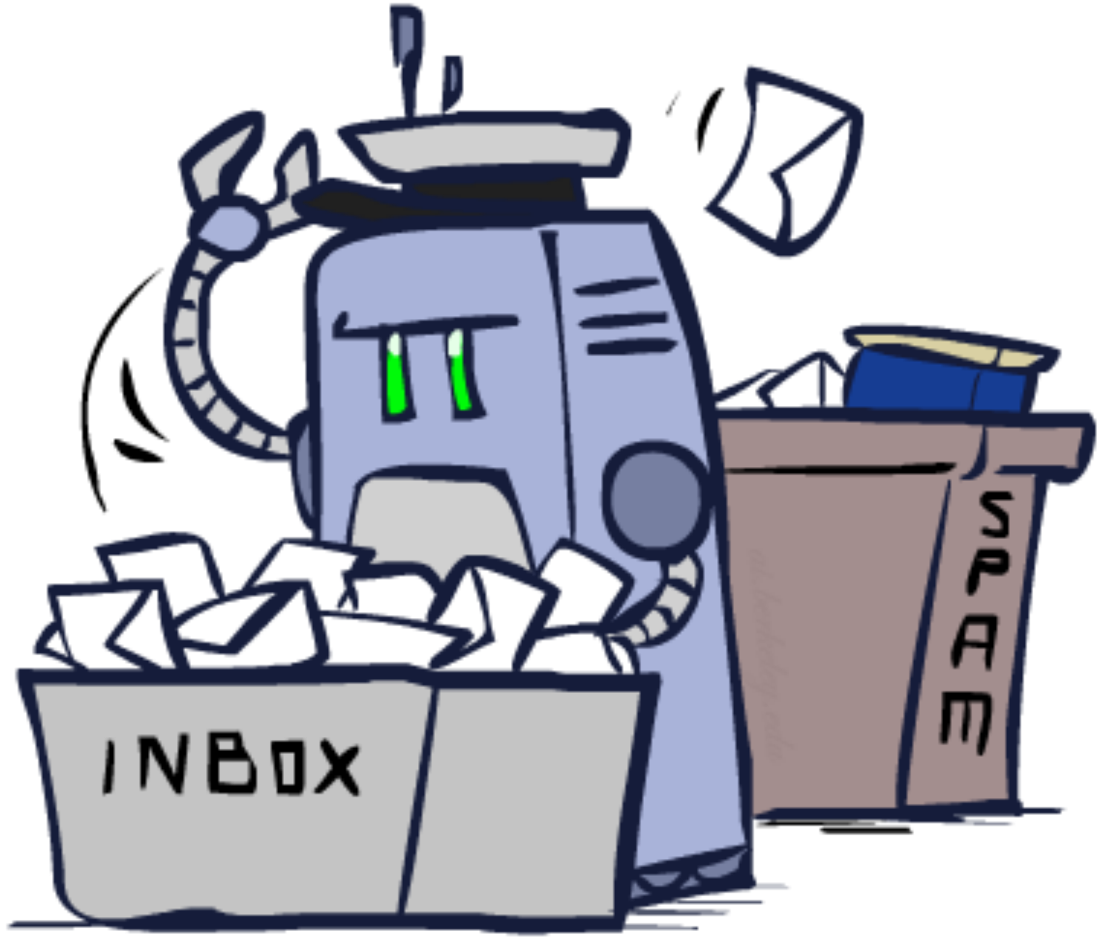
- Exploration function
  - Takes a value estimate u and a visit count n, ai returns an optimistic utility, e.$f(u,n) = u + k/n$

Regular Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} Q(s',a')$

Modified Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} f(Q(s',a'), N(s',a'))$

  - Note: this propagates the "bonus" back to states that lead to unknown states as well!

Machine Learning

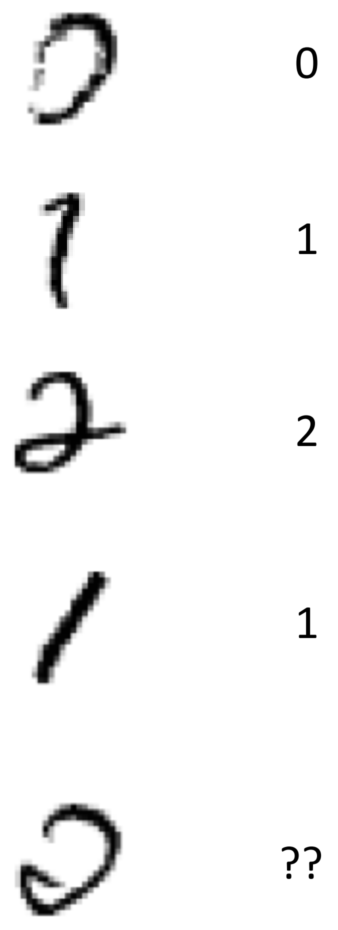


Example: Digit Recognition

- Input: images / pixel grids
- Output: a digit 0-9

- Setup:
  - Get a large collection of example images, each labeled with a digit
  - Note: someone has to hand label all this data!
  - Want to learn to predict labels of new, future digit images

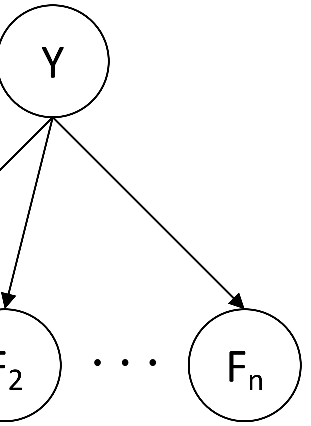
- Features: The attributes used to make the digit decision
  - Pixels: (6,8)=ON
  - Shape Patterns: NumComponents, AspectRatio, NumLoops
  - …
  - Features are increasingly induced rather than crafted



Naïve Bayes for Digits

- Naïve Bayes: Assume all features are independent effects of the label

- Simple digit recognition version:
  - One feature (variable) F_{ij} for each grid position <i,j>
  - Feature values are on / off, based on whether intensity is more or less than 0.5 in underlying image
  - Each input maps to a feature vector, e.g.



$$\rightarrow \langle F_{0,0} = 0 \;\; F_{0,1} = 0 \;\; F_{0,2} = 1 \;\; F_{0,3} = 1 \;\; F_{0,4} = 0 \;\; \ldots F_{15,15} = 0 \rangle$$

  - Here: lots of features, each is binary valued

$$P_{\text{ML}}(x) = \frac{\text{count}(x)}{\text{total samples}}$$

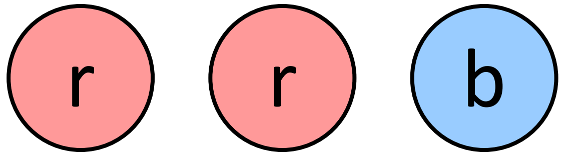
- Naïve Bayes model: $P(Y|F_{0,0} \ldots F_{15,15}) \propto P(Y) \prod_{i,j} P(F_{i,j}|Y)$

- **Conditional probabilities $P(F_{i,j} \mid Y)$ just come from counts in the training data**

Deriving MLEs

- **Model**:

X	red	blue
$P_\theta(x)$	θ	$1-\theta$



- **Data**: draw N balls. N_r come up red, N_b come up blue
  - Dataset: $D = \{x_1, \dots, x_n\}$
  - Ball draws are independent and identically distributed (i.i.d.):

$$P(\,D \mid \theta\,) = \prod_i P(\,x_i \mid \theta\,) = \prod_i P_\theta(x_i) = \theta^{N_r} \cdot (1-\theta)^{N_b}$$

- **Maximum likelihood estimation**: find θ that maximizes $P(\,D \mid \theta\,)$

$$\theta = \underset{\theta}{\operatorname{argmax}}\, P(\,D \mid \theta\,) = \underset{\theta}{\operatorname{argmax}} \log P(\,D \mid \theta\,)$$

  - Approach: take derivative and set to 0

Deriving MLEs

- **Maximum likelihood estimation**: find θ that maximizes $P(\,D \mid \theta\,)$

$$\theta = \underset{\theta}{\operatorname{argmax}}\, P(\,D \mid \theta\,) = \underset{\theta}{\operatorname{argmax}} \log P(\,D \mid \theta\,)$$

$$\begin{aligned}\frac{\partial}{\partial\theta}\log P(\,D \mid \theta\,) &= \frac{\partial}{\partial\theta}\left[N_r \log(\theta) + N_b \log(1-\theta)\right] \\ &= N_r \frac{\partial}{\partial\theta}\log(\theta) + N_b \frac{\partial}{\partial\theta}\log(1-\theta) \\ &= N_r \frac{1}{\theta} - N_b \frac{1}{1-\theta} \\ &= 0\end{aligned}$$

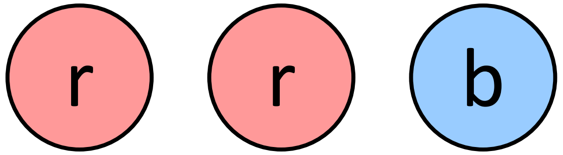
Multiply by $\theta(1-\theta)$:

$$N_r(1-\theta) - N_b\theta = 0$$
$$N_r - \theta(N_r + N_b) = 0$$

$$\boxed{\hat{\theta} = \frac{N_r}{N_r + N_b}}$$

Regularization: Smoothing

- Laplace's estimate:
  - Pretend you saw every outcome once more than you actually did



$$P_{LAP}(x) = \frac{c(x)+1}{\sum_x [c(x)+1]}$$

$$= \frac{c(x)+1}{N+|X|}$$

$$P_{ML}(X) = \left\langle \frac{2}{3}, \frac{1}{3} \right\rangle$$

$$P_{LAP}(X) = \left\langle \frac{3}{5}, \frac{2}{5} \right\rangle$$

- **This is no longer a maximum likelihood estimate**

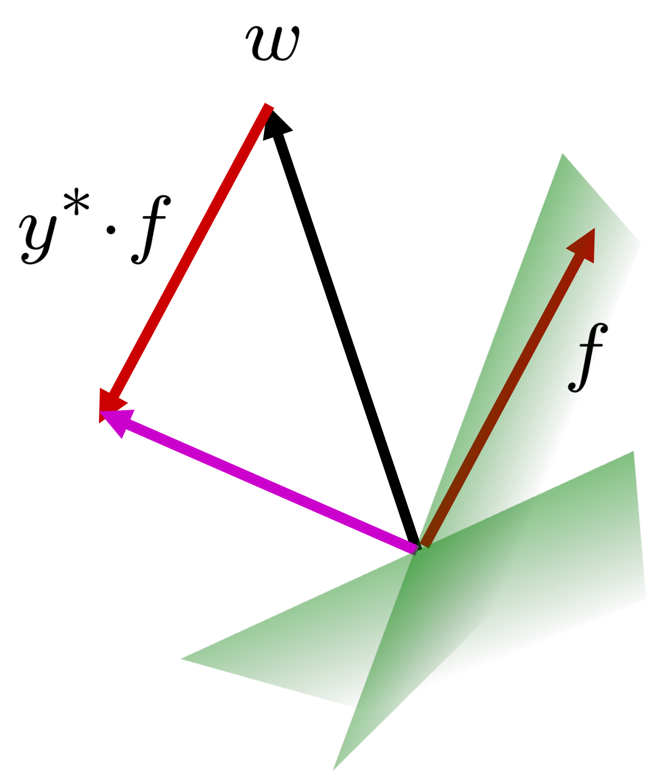
Binary Perceptron

- Start with weights = 0
- For each training instance:
  - Classify with current weights

$$y = \begin{cases} +1 & \text{if } \; w \cdot f(x) \geq 0 \\ -1 & \text{if } \; w \cdot f(x) < 0 \end{cases}$$

  - If correct (i.e., y=y*), no change!
  - If wrong: adjust the weight vector by adding or subtracting the feature vector. Subtract if y* is -1.

$$w = w + y^* \cdot f$$



Multiclass Perceptron

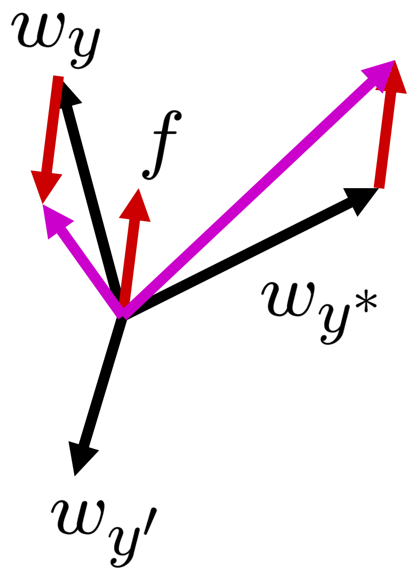
- Start with all weights = 0
- Pick up training examples one by one
- Predict with current weights

$$y = \arg\max_y \; w_y \cdot f(x)$$

- If correct, no change!
- If wrong: lower score of wrong answer, raise score of right answer

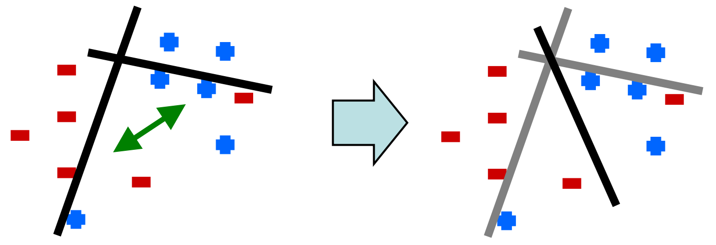
$$w_y = w_y - f(x)$$

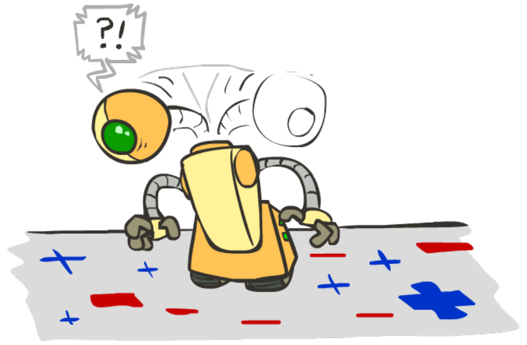
$$w_{y^*} = w_{y^*} + f(x)$$



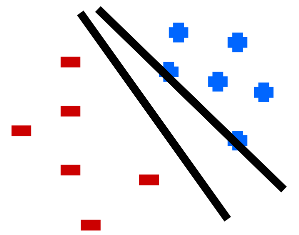
Problems with the Perceptron

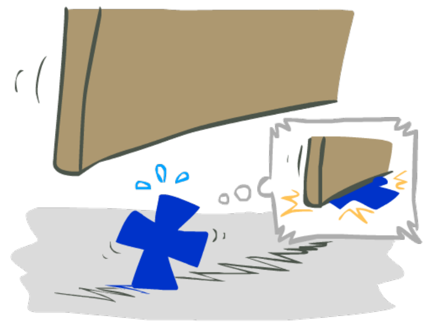
- Noise: if the data isn't separable, weights might thrash
  - Averaging weight vectors over time can help (averaged perceptron)



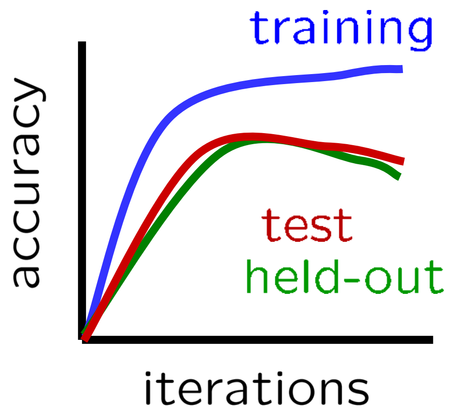


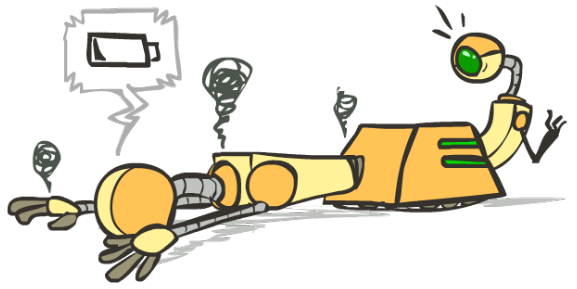
- Mediocre generalization: finds a "barely" separating solution





- Overtraining: test / held-out accuracy usually rises, then falls
  - Overtraining is a kind of overfitting





Logistic Regression

- Maximum likelihood estimation:

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

with:

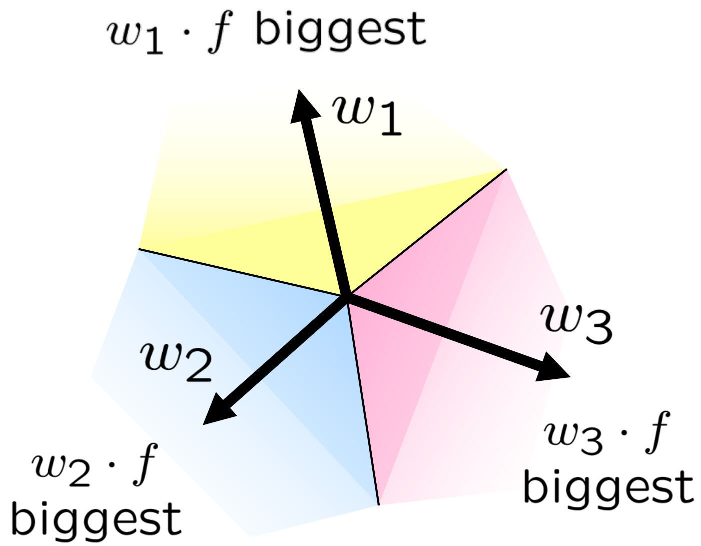
$$P(y^{(i)} = +1 | x^{(i)}; w) = \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}$$

$$P(y^{(i)} = -1 | x^{(i)}; w) = 1 - \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}$$

Aside: linear regression ≠ logistic regression!

Multiclass Logistic Regression

- Recall Perceptron:
  - A weight vector for each class: w_y
  - Score (activation) of a class y: $w_y \cdot f(x)$
  - Prediction highest score wins $y = \arg\max_y \; w_y \cdot f(x)$



- How to make the scores into probabilities?

$$\underbrace{z_1, z_2, z_3}_{\text{original activations}} \rightarrow \underbrace{\frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}}, \frac{e^{z_2}}{e^{z_1}+e^{z_2}+e^{z_3}}, \frac{e^{z_3}}{e^{z_1}+e^{z_2}+e^{z_3}}}_{\text{softmax activations}}$$

Batch Gradient Ascent

$$\max_w \; ll(w) = \max_w \; \underbrace{\sum_i \log P(y^{(i)}|x^{(i)};w)}_{g(w)}$$

```
○ init $w$
○ for iter = 1, 2, …
      $w \leftarrow w + \alpha * \sum_i \nabla \log P(y^{(i)}|x^{(i)};w)$
```

Stochastic Gradient Ascent

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)}|x^{(i)};w)$$

Observation: once gradient on one training example has been computed, might as well incorporate before computing next one

```
init $w$
for iter = 1, 2, …
    pick random j
        $w \leftarrow w + \alpha * \nabla \log P(y^{(j)}|x^{(j)};w)$
```

Mini-batch Gradient Ascent

$$\max_w \; ll(w) = \max_w \; \sum_i \log P(y^{(i)}|x^{(i)};w)$$

Observation: gradient over small set of training examples (=mini-batch) can be computed in parallel, might as well do that instead of a single one

```
o init $w$
o for iter = 1, 2, …
    o pick random subset of training examples J
      $w \leftarrow w + \alpha * \sum_{j \in J} \nabla \log P(y^{(j)}|x^{(j)};w)$
```

Beyond SGD: Second-Order Derivatives

Newton's Method (in 1D):

- Want to optimize: $\max_{\theta} f(\theta)$
- Apply Taylor expansion:

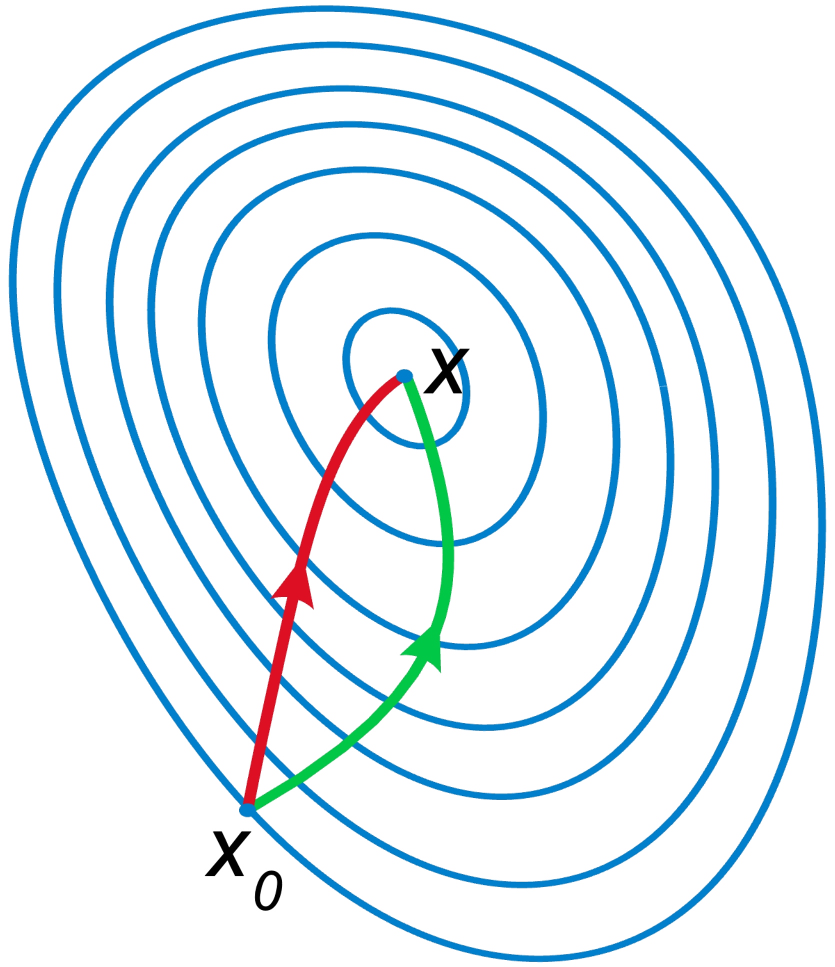
$$f(\theta + h) = f(\theta) + f'(\theta)h + \frac{1}{2}f''(\theta)h^2$$

- Find value of t that maximizes this:

$$0 = \frac{\partial}{\partial h}\left[f(\theta) + f'(\theta)h + \frac{1}{2}f''(\theta)h^2\right]$$
$$= \mathrm{f}'(\theta) + f''(\theta)h$$

- Rearrange terms to get update:

$$h = -\frac{f'(\theta)}{f''(\theta)} \qquad \theta_{t+1} = \theta_t + h = \theta_t - \frac{f'(\theta)}{f''(\theta)}$$



These update equations out of scope for final exam; but high-level concepts are in scope

Beyond SGD: Momentum

- Potential issues with vanilla SGD:
  - Can take a long time to converge if the learning rate is too low
  - Can bounce around in “ravines” without making much progress toward a local optimum

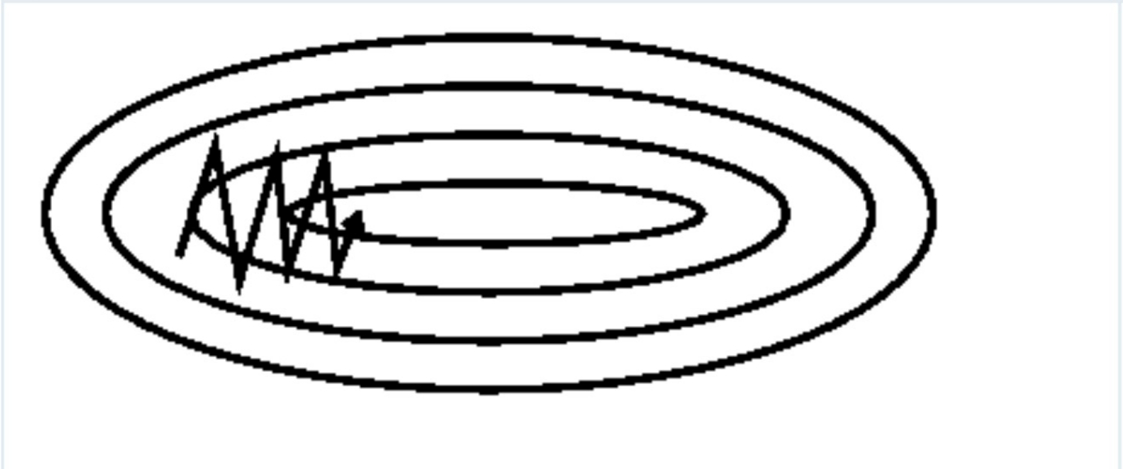


Image 2: SGD without momentum

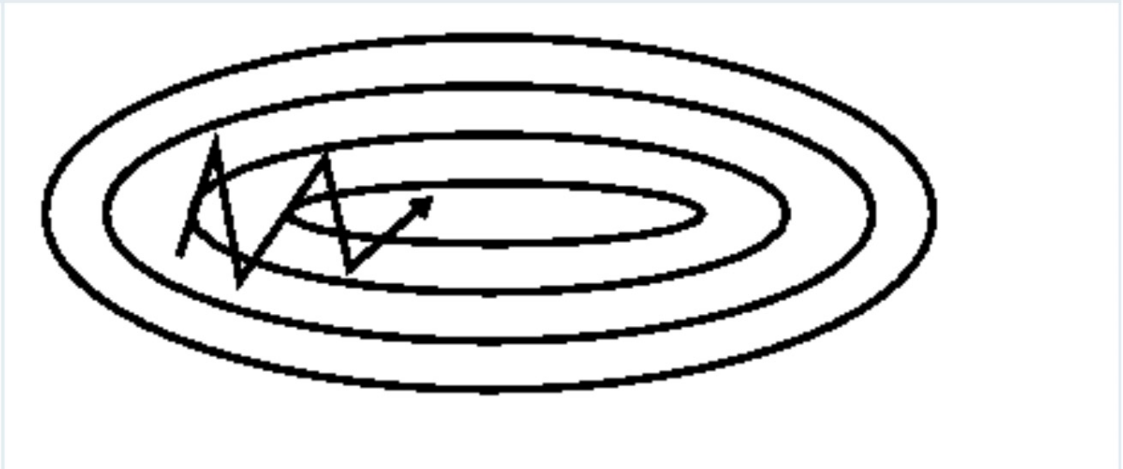


Image 3: SGD with momentum

Beyond SGD: Adaptive Learning Rates

- Recall: learning rates
  - Determines how much we update weights in the direction of the gradient
  - Often: want to set this in terms of how much it updates the weights
  - Often: want to lower learning rate over time (*learning rate scheduling*)

$$\theta_{t+1} = \theta_t - \boxed{\eta}\nabla_\theta \mathrm{f}(\theta_t)$$

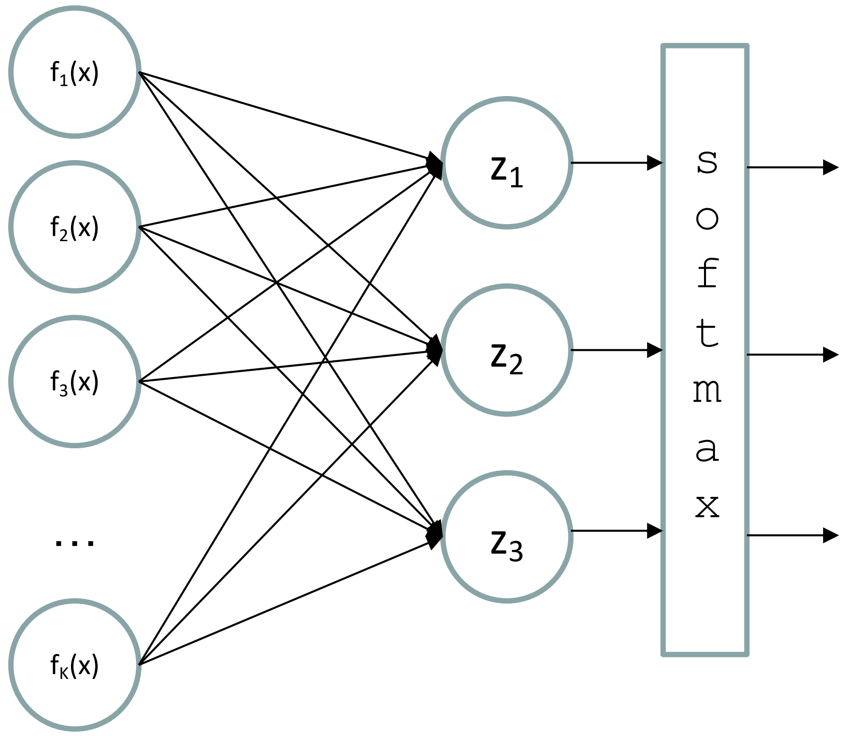
- Key idea: different learning rates for each parameter
  - We can make larger or smaller updates depending on how important a feature is
  - Small updates for frequent features; big updates for rare features
  - This idea underlies: Adagrad, RMSProp, Adam, etc.

Summary: Key Ideas in Optimization

- Gradient descent
  - Batch: update based on the whole dataset
  - SGD: update based on a single randomly chosen training example
  - Minibatch: update based on *k* randomly chosen training examples

- More advanced approaches:
  - Second order optimization (e.g., Newton's method)
  - Momentum (Nesterov's accelerated gradient, Adam)
  - Adaptive learning rates (Adagrad, RMSProp, Adam, etc.)

Logistic Regression

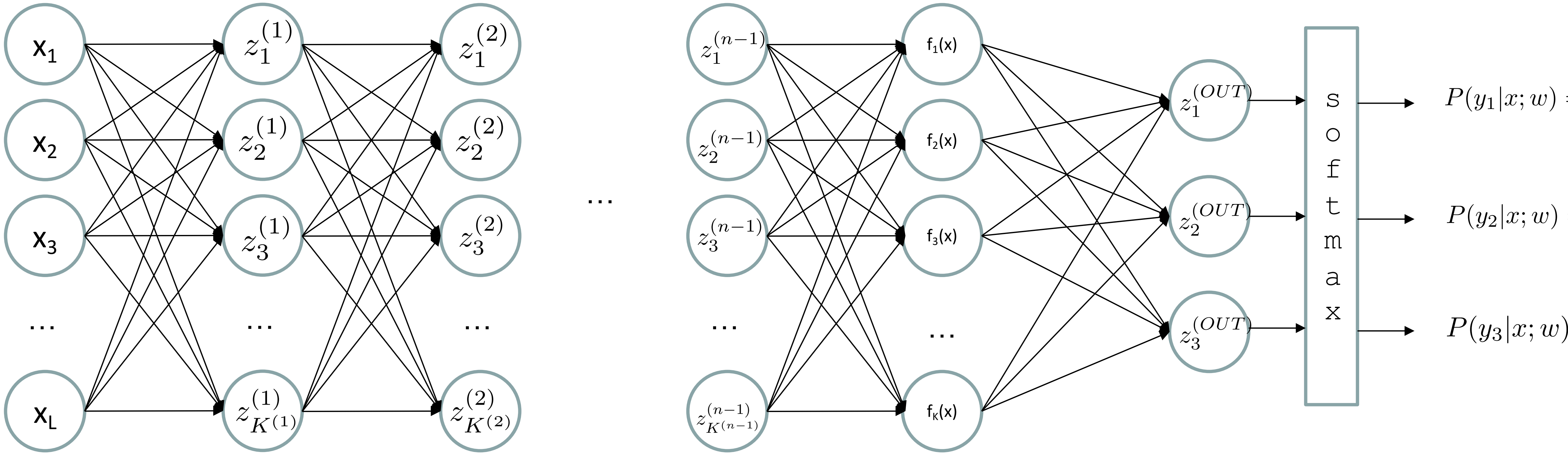


$$P(y_1|x;w) = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$P(y_2|x;w) = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$P(y_3|x;w) = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

Deep Neural Networks

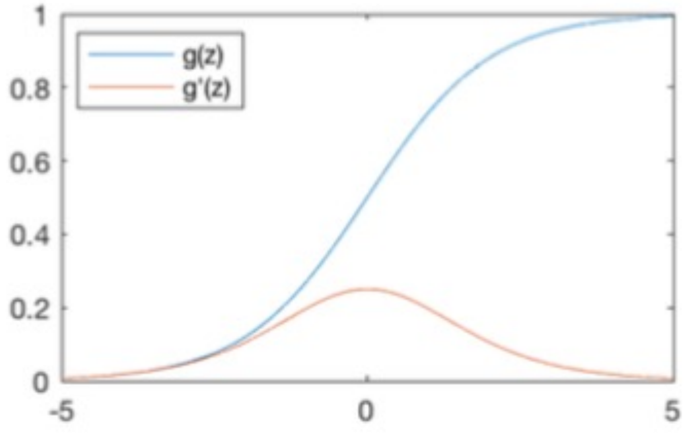


$$z_i^{(k)} = g(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)})$$

g = nonlinear activation function

Common Activation Functions

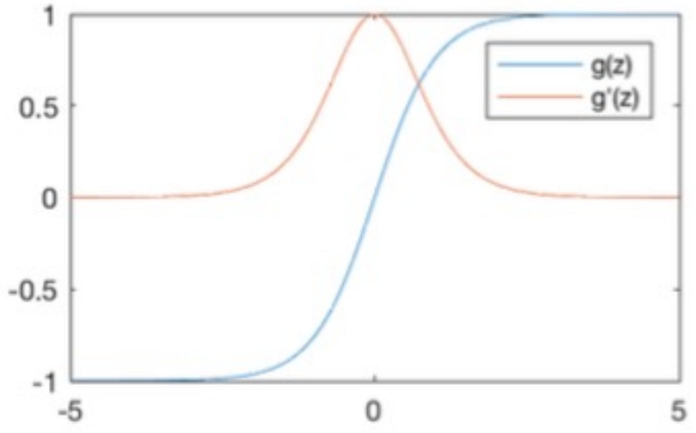
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

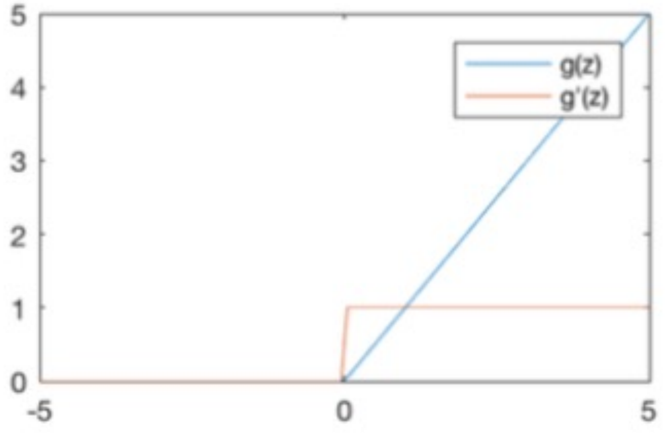
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



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

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

[source: MIT 6.S191 introtodeeplearning.com]

Neural Network Properties

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

- Practical considerations:
  - Must have nonlinear activation function
  - Requires an arbitrarily large number of neurons:
    - Danger for overfitting (hence early stopping!)
    - No guarantee that we can do this on real-world compute
  - Often more efficient in practice to have more layers, less neurons

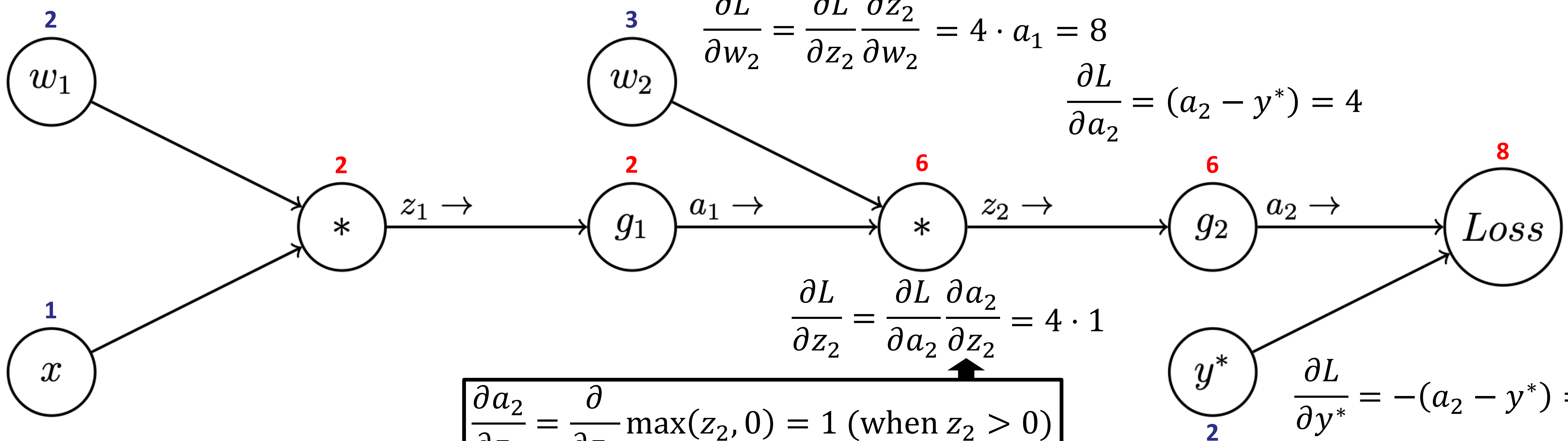
Example: Automatic Differentiation

- Build a computation graph and apply chain rule: $f(x) = g\big(h(x)\big)$ $f'(x) = h'(x) \cdot g'(h(x))$
- Example: neural network with quadratic loss: $L(a_2, y^*) = \frac{1}{2}(a_2 - y^*)^2$ and ReLU activations $g(z) = \max(0, z)$
- $a_2 = g_2(w_2 * g_1(w_1 * x))$

$$\boxed{\frac{\partial z_2}{\partial w_2} = \frac{\partial}{\partial w_2}(w_2 \cdot a_1) = a_1}$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial w_2} = 4 \cdot a_1 = 8$$

$$\frac{\partial L}{\partial a_2} = (a_2 - y^*) = 4$$



$$\frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial z_2} = 4 \cdot 1$$

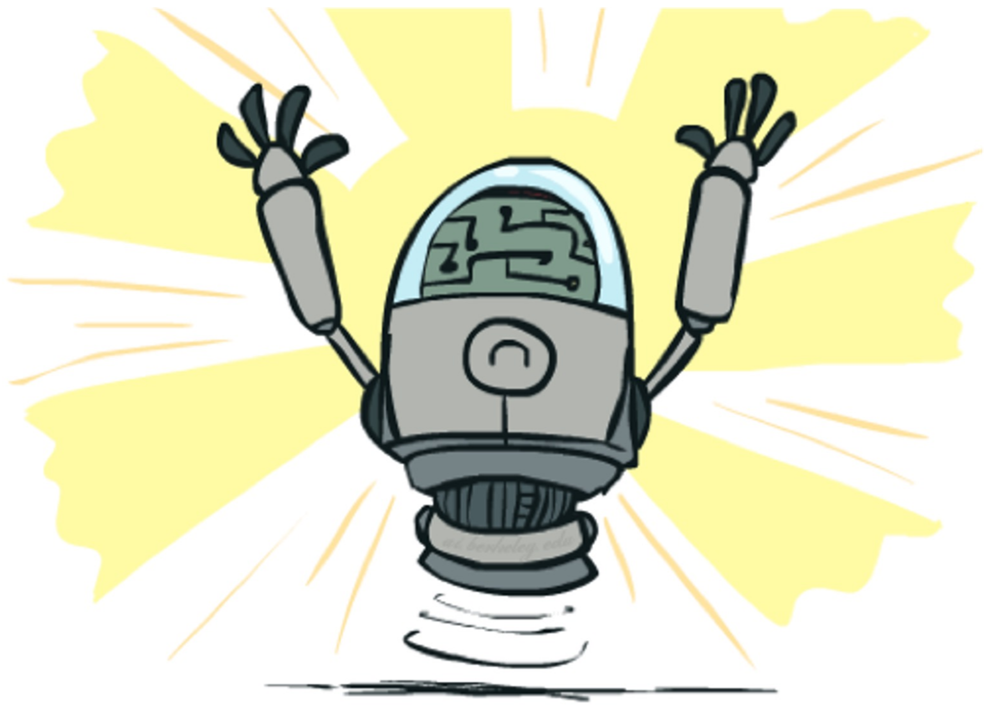
$$\boxed{\frac{\partial a_2}{\partial z_2} = \frac{\partial}{\partial z_2}\max(z_2, 0) = 1 \text{ (when } z_2 > 0)}$$

$$\frac{\partial L}{\partial y^*} = -(a_2 - y^*) = -4$$

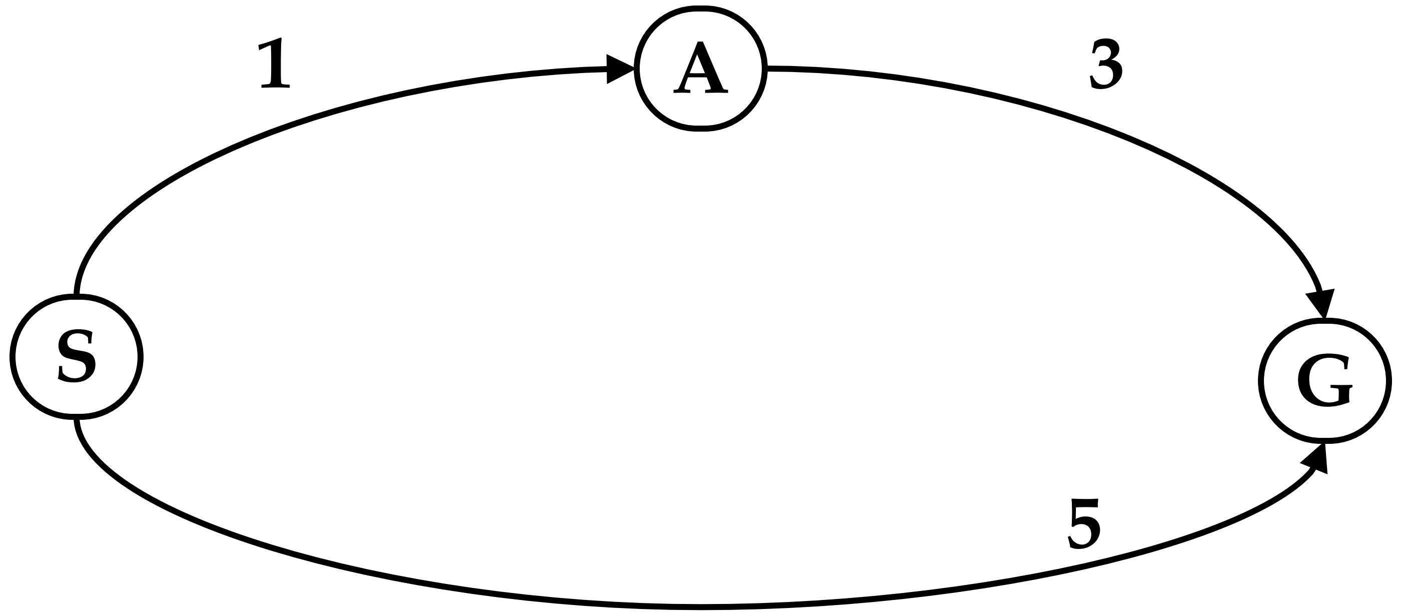
Search

A* Search

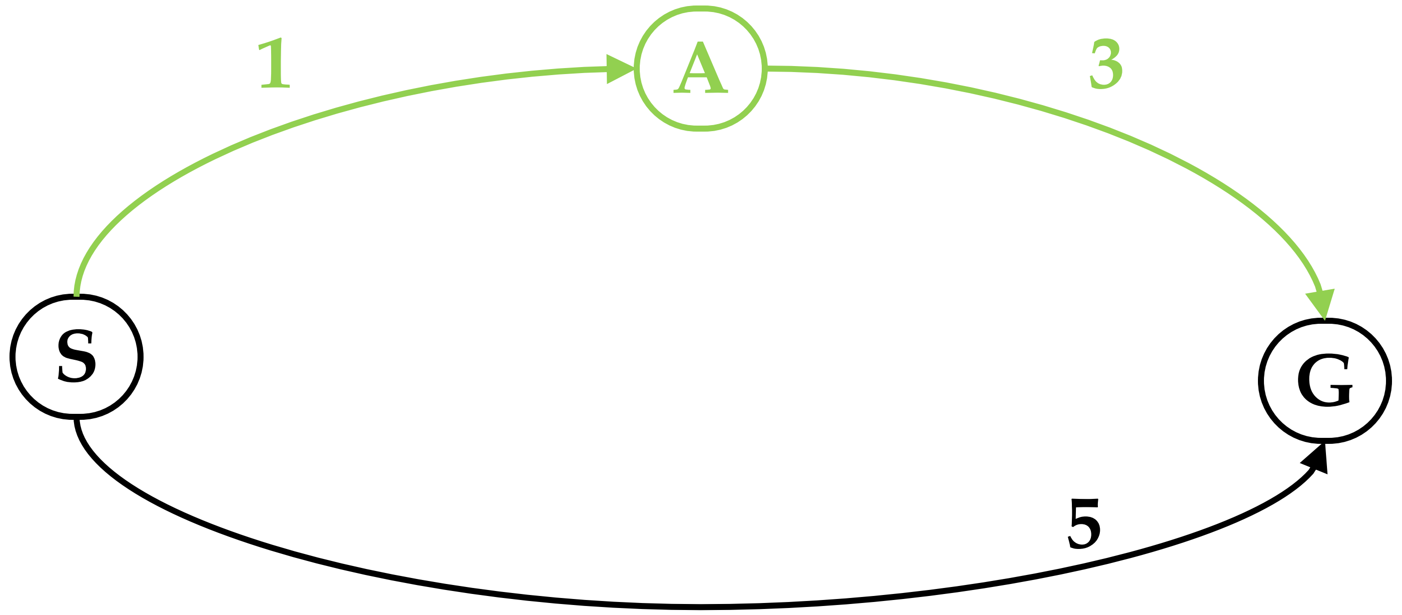
- Expand nodes based on sum: backward cost + forward cost
  - $f(n) = g(n) + h(n)$
  - $g(n)$: cost to get to node
  - $h(n)$: heuristic of future costs
- We ideally want heuristic functions that satisfy:
  - Admissibility: underestimate true cost to the goal
  - Consistency: "triangle inequality"
- Consistency => admissibility



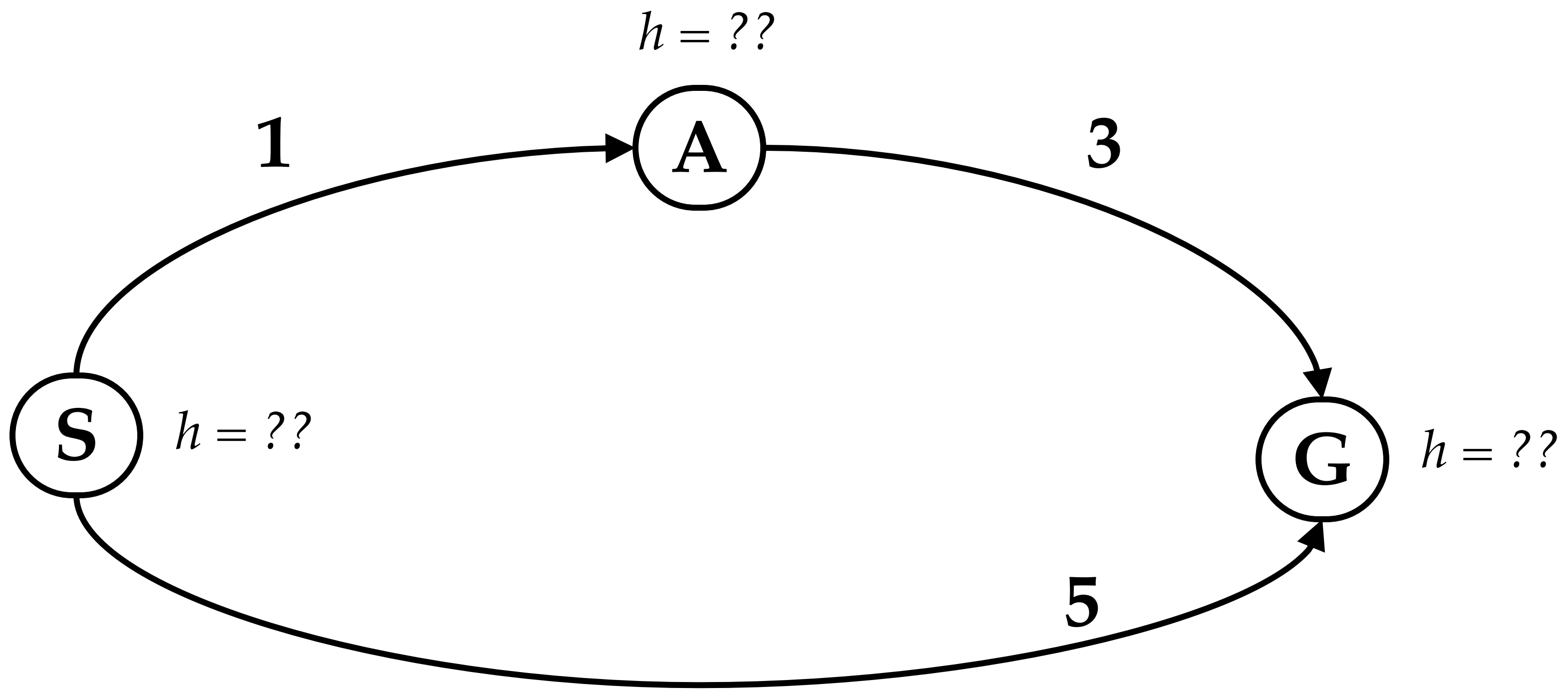
A* Search: Admissibility



A* Search: Admissibility



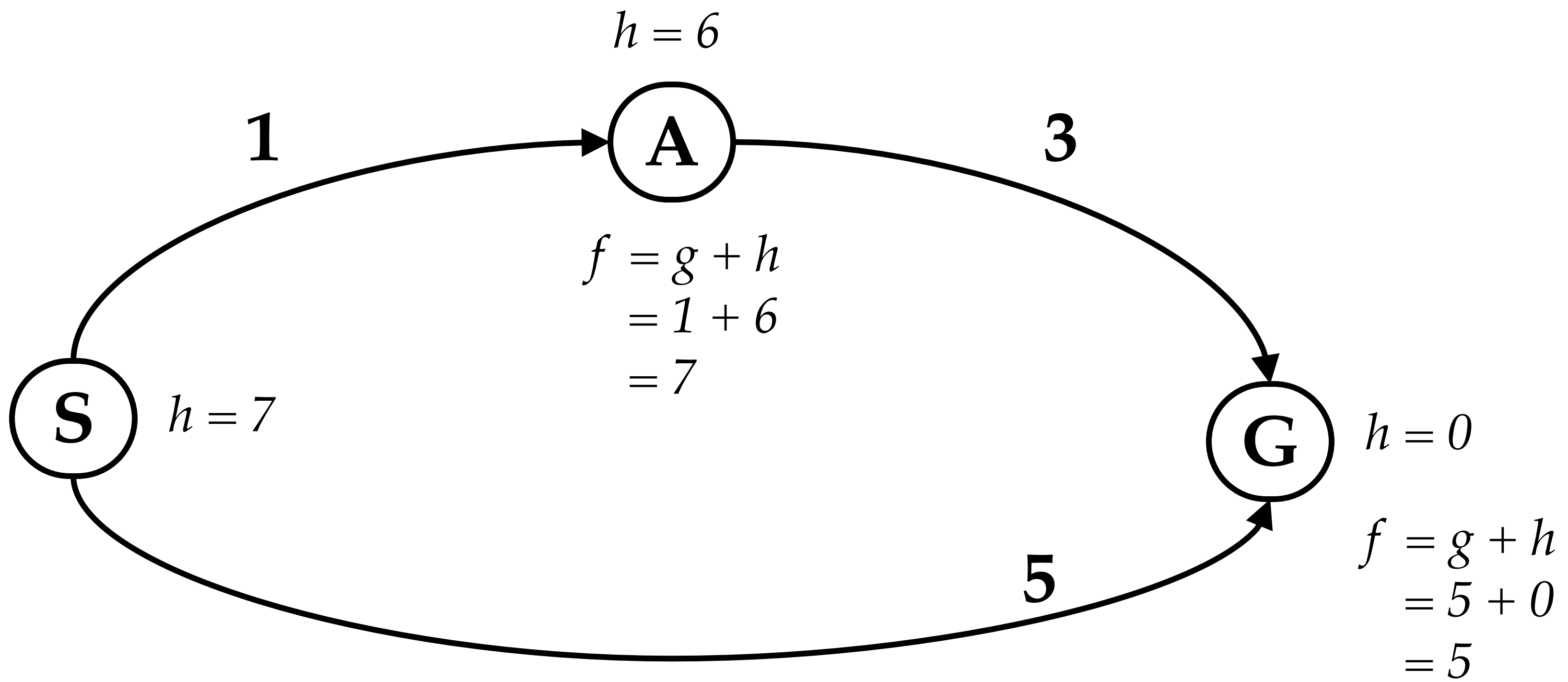
A* Search: Admissibility



A* Search: Admissibility

Q: Where do heuristics come from?

A: We have to create them!

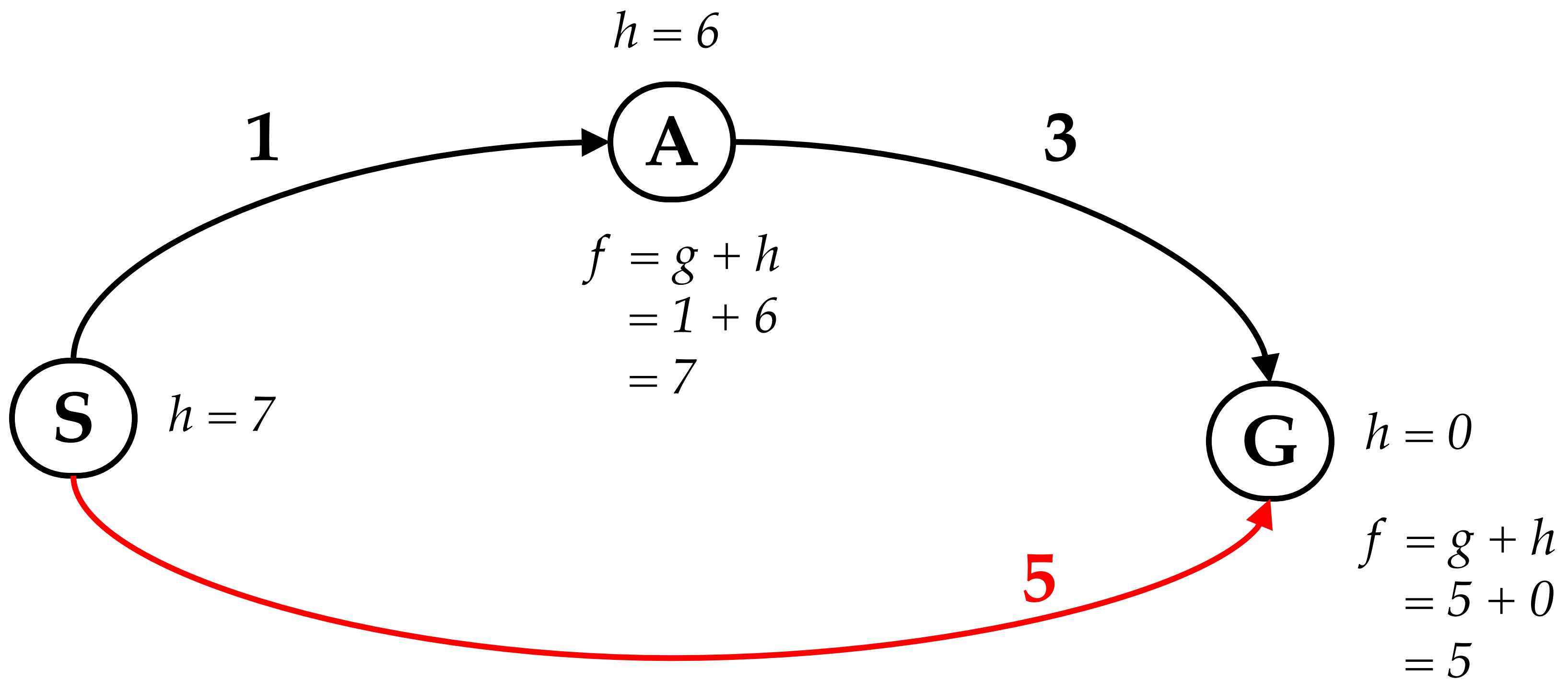


Not the best heuristic…

A* Search: Admissibility

Q: Where do heuristics come from?

A: We have to create them!

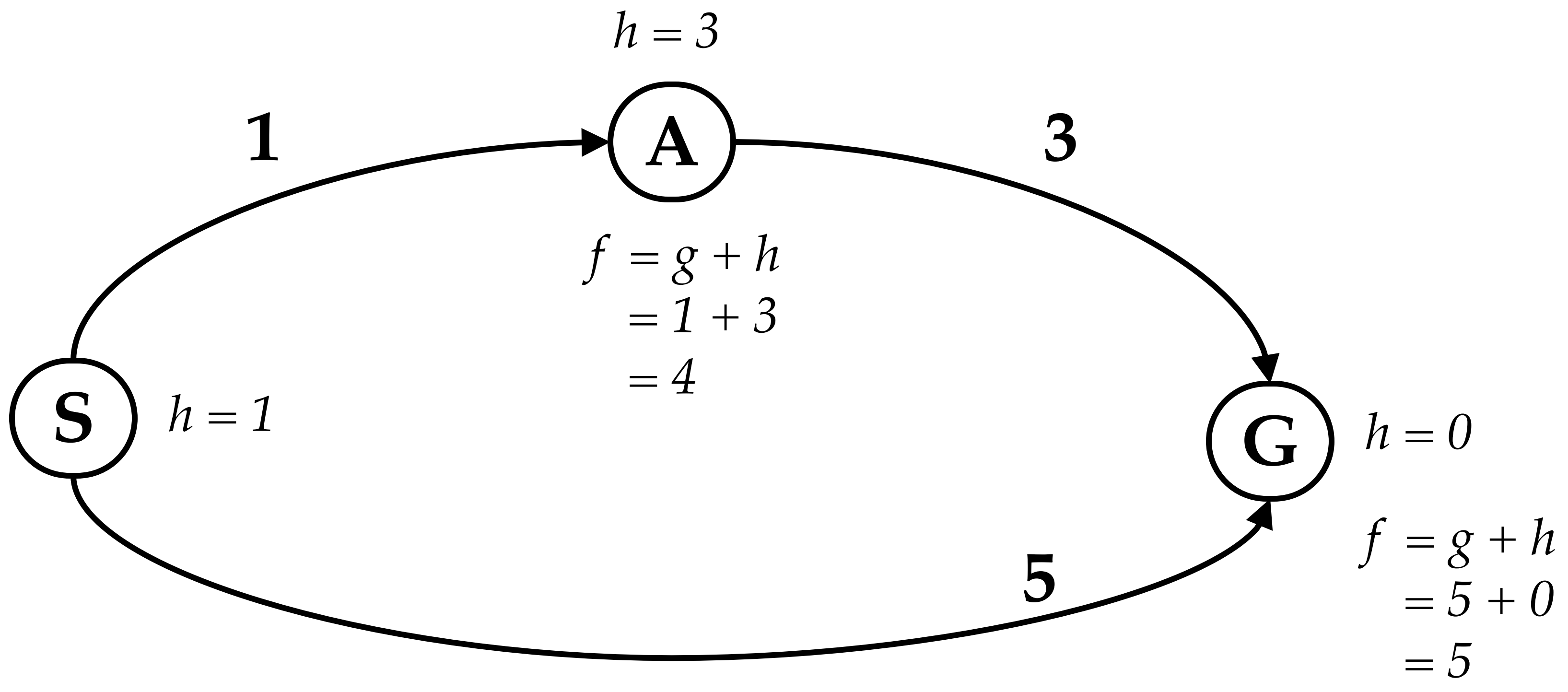


Not the best heuristic…

A* Search: Admissibility

Q: Where do heuristics come from?

A: We have to create them!

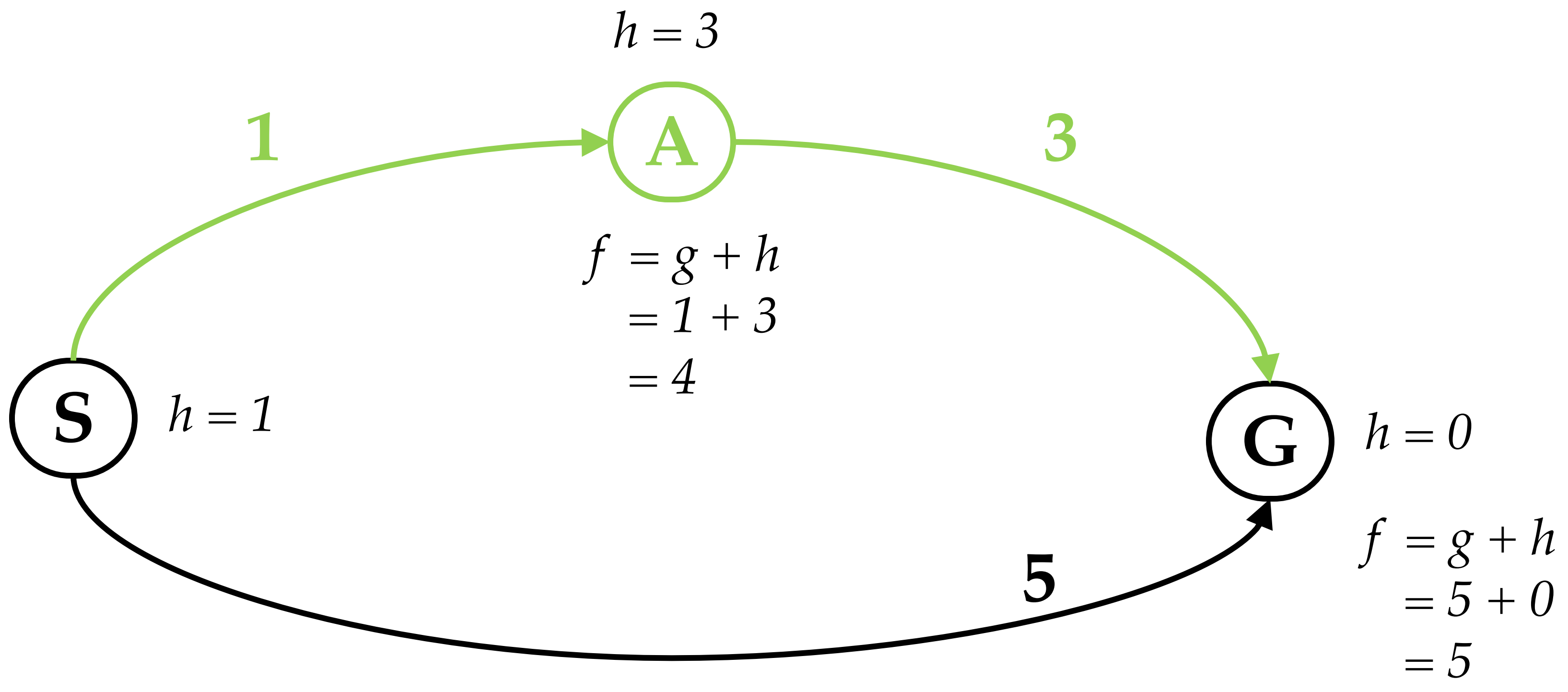


What's a better heuristic?

A* Search: Admissibility

Q: Where do heuristics come from?

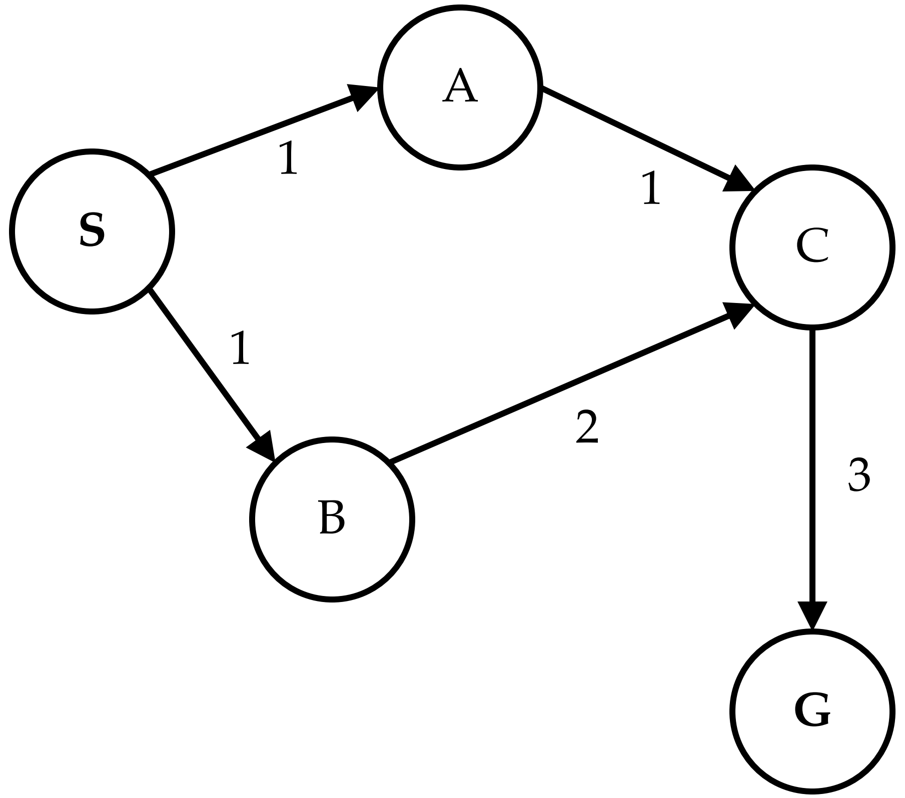
A: We have to create them!



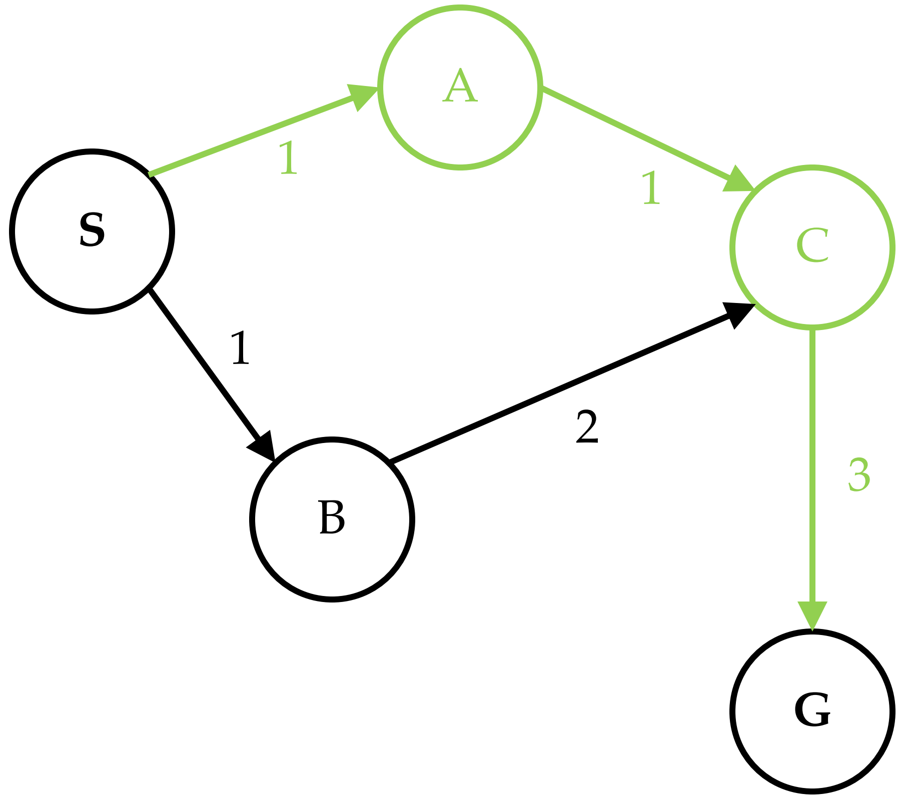
What's a better heuristic?

Admissible = Underestimates Cost to the Goal

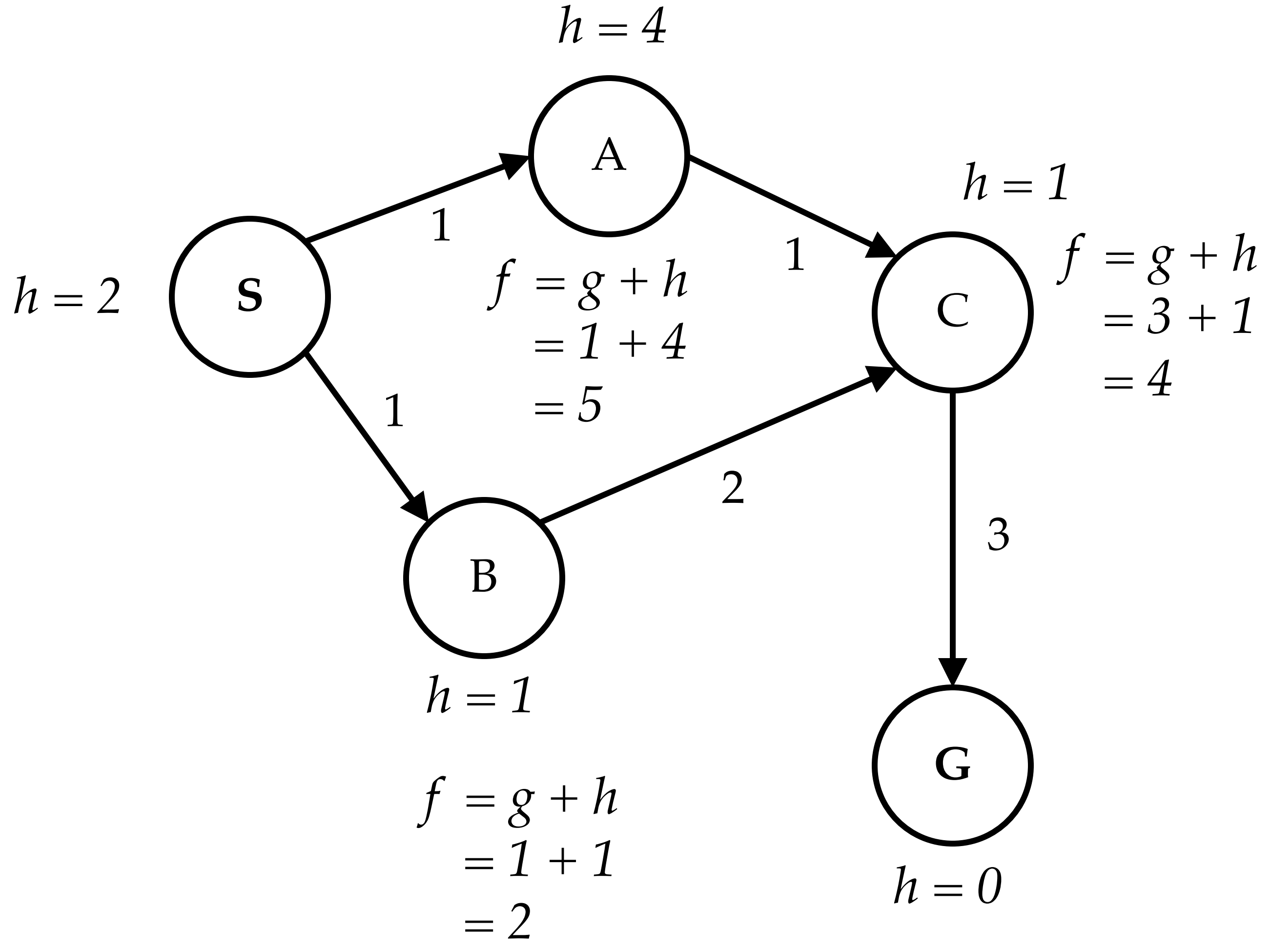
A* Search: Consistency



A* Search: Consistency

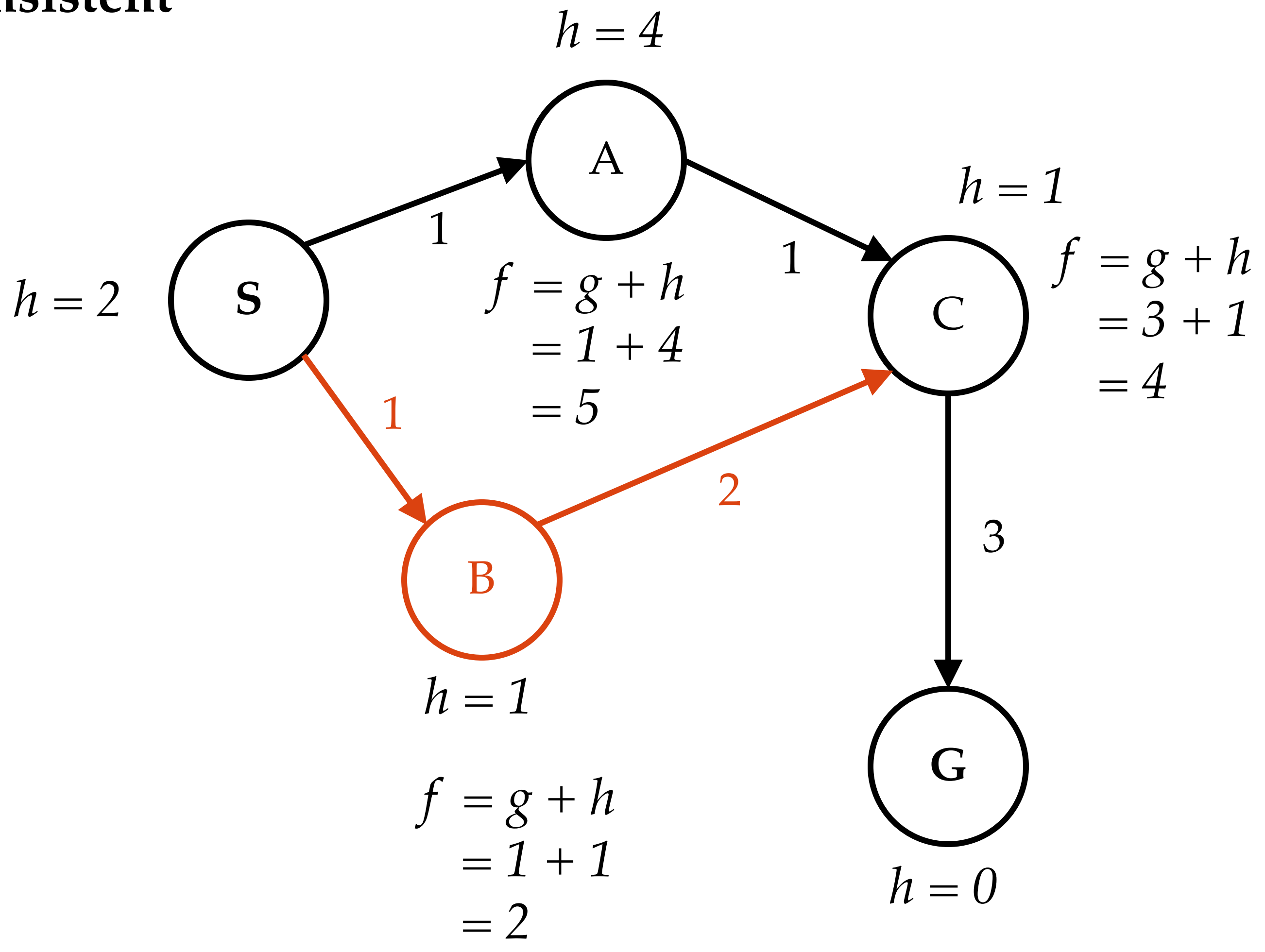


A* Search: Consistency



A* Search: Consistency

This heuristic isn't **consistent**

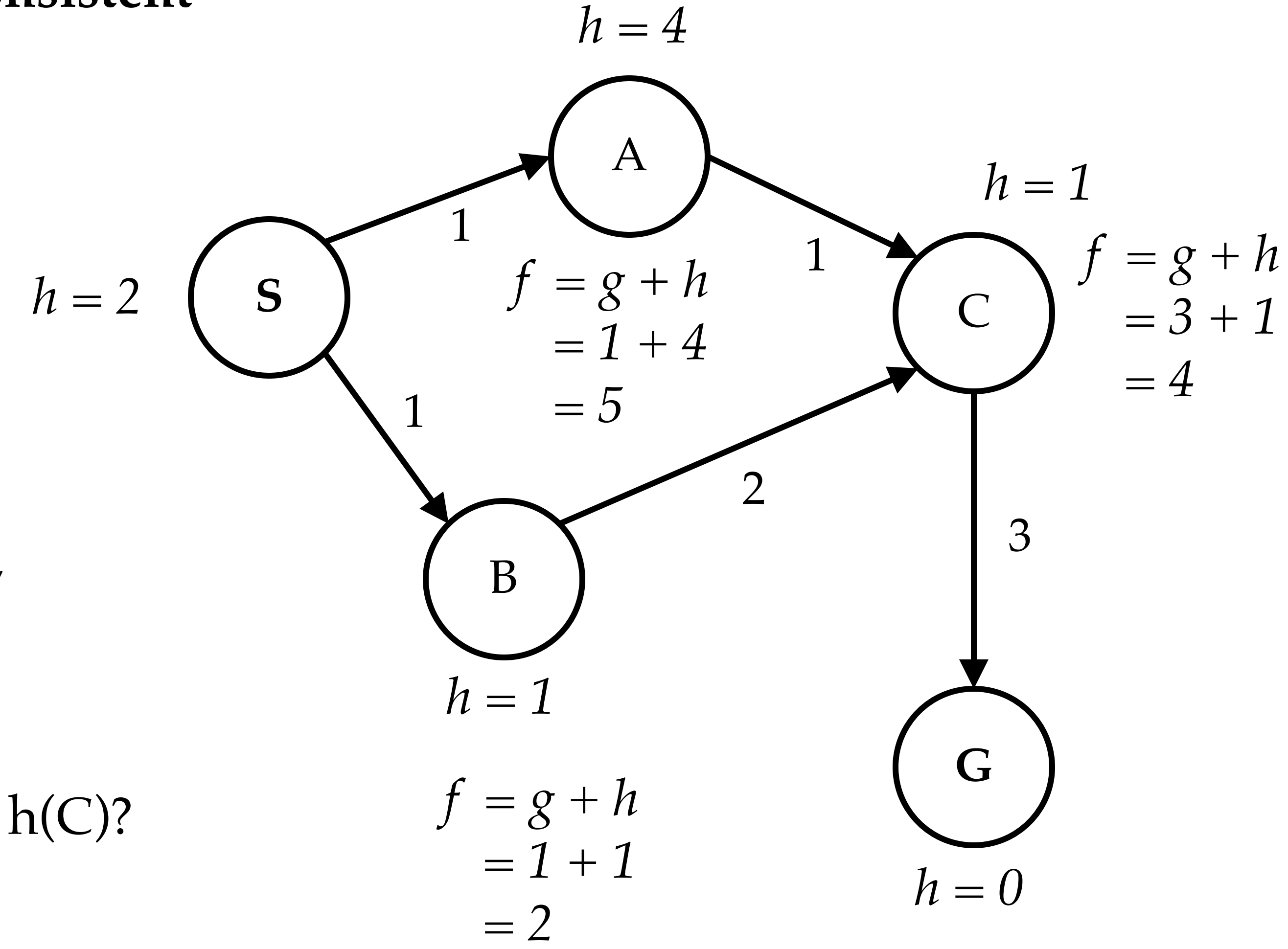


"Triangle inequality"

$h(u) \leq d(u,v) + h(v)$

A* Search: Consistency

This heuristic isn't **consistent**

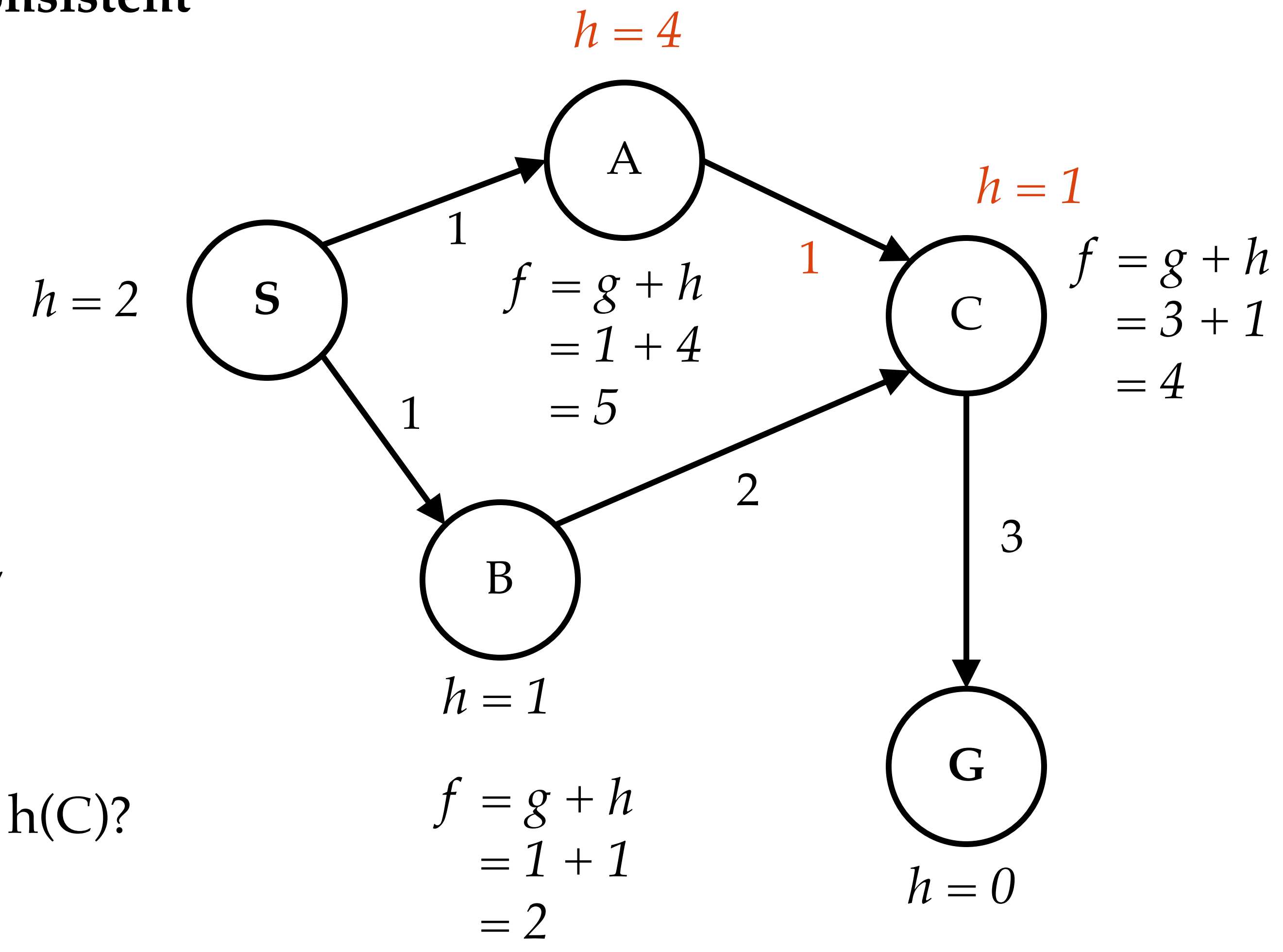


"Triangle inequality"
h(u) ≤ d(u,v) + h(v)

Q: Is h(A) ≤ d(A,C) + h(C)?

A* Search: Consistency

This heuristic isn't **consistent**



"Triangle inequality"
$h(u) \le d(u,v) + h(v)$

Q: Is $h(A) \le d(A,C) + h(C)$?
A: No: $4 \not\le 1 + 1$

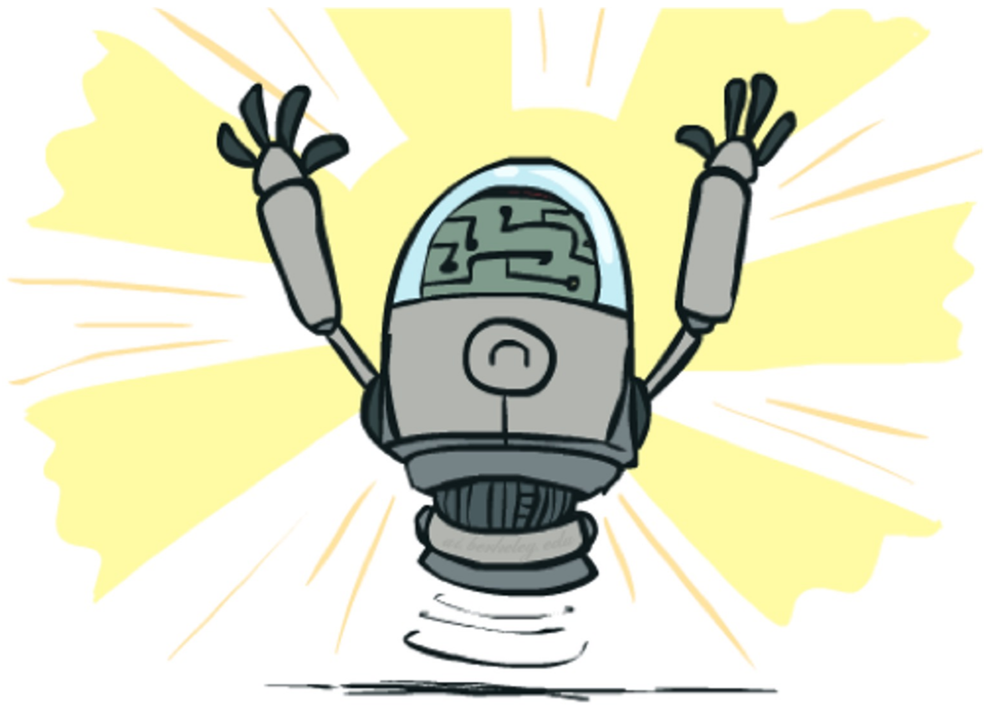
Summary of A*

- 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 it comes from a relaxed problem



Constraint Satisfaction Problems

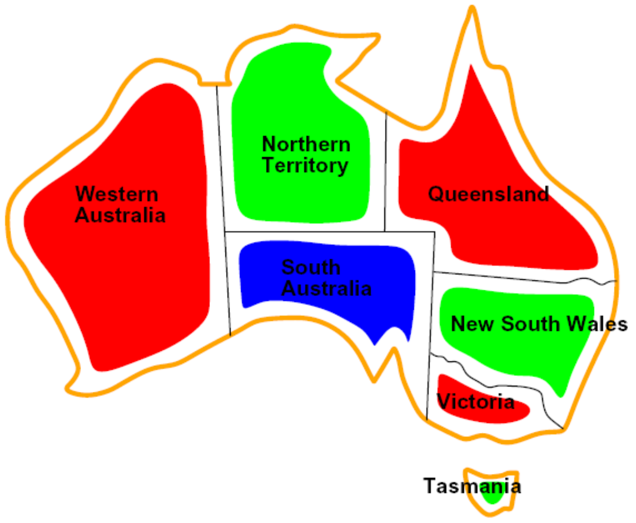
Example: Map Coloring

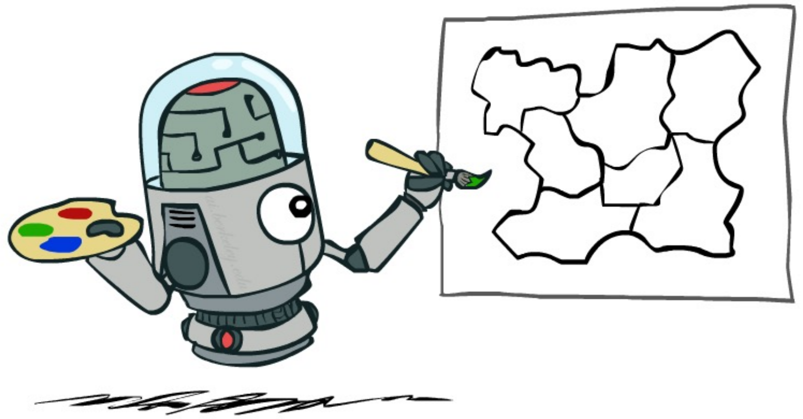
- Variables: $\mathsf{WA,\ NT,\ Q,\ NSW,\ V,\ SA,\ T}$
- Domains: $\mathsf{D = \{red, green, blue\}}$
- Constraints: adjacent regions must have different colors

  Implicit: $\mathsf{WA \neq NT}$

  Explicit: $\mathsf{(WA, NT) \in \{(red, green), (red, blue), \ldots\}}$
- Solutions are assignments satisfying all constraints, e.g.:

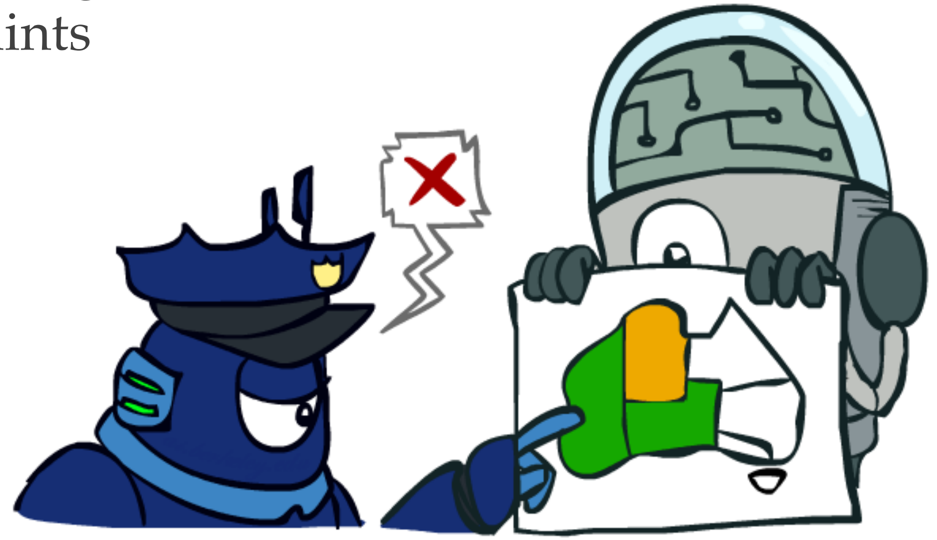
  $\mathsf{\{WA{=}red,\ NT{=}green,\ Q{=}red,\ NSW{=}green,\ V{=}red,\ SA{=}blue,\ T{=}green\}}$





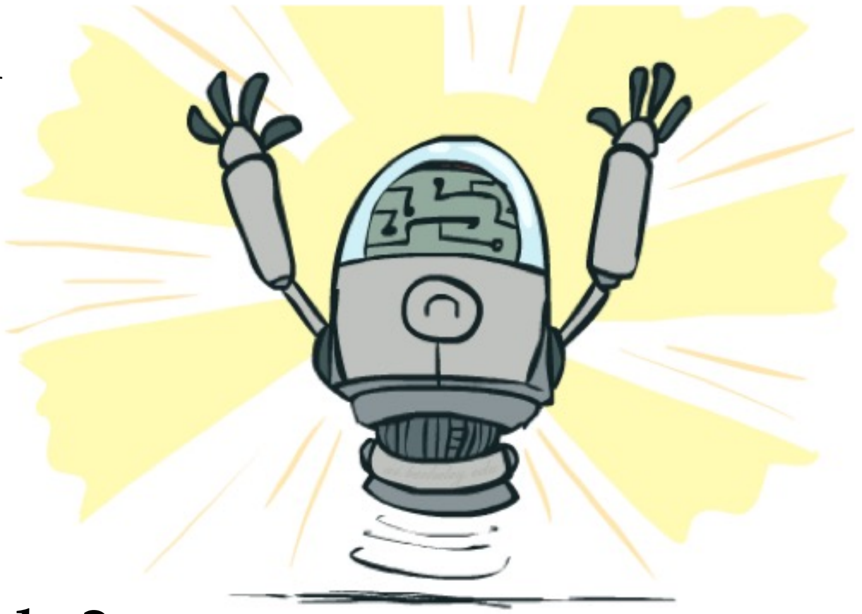
General Approach #1: Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering -> better branching factor!
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - "Incremental goal test"
- Depth-first search with these two improvements is called *backtracking search* (not the best name)
- Can solve n-queens for n ≈ 25



Improving Backtracking

General-purpose ideas give huge gains in speed



1. Ordering:
   - Which variable should be assigned next?
   - In what order should its values be tried?

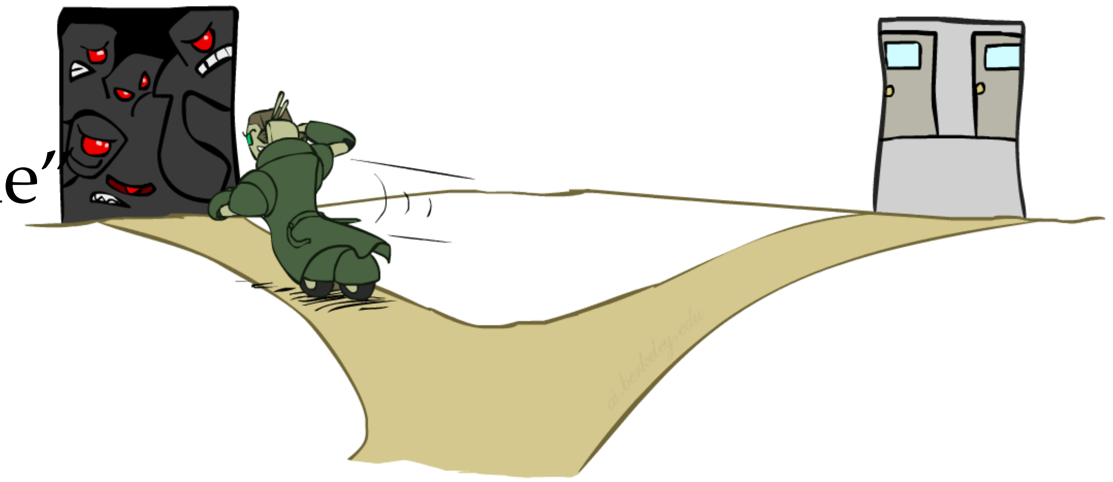
2. Filtering: Can we detect inevitable failure early?

3. Leveraging the structure of the constraint graph

Ordering: Minimum Remaining Values

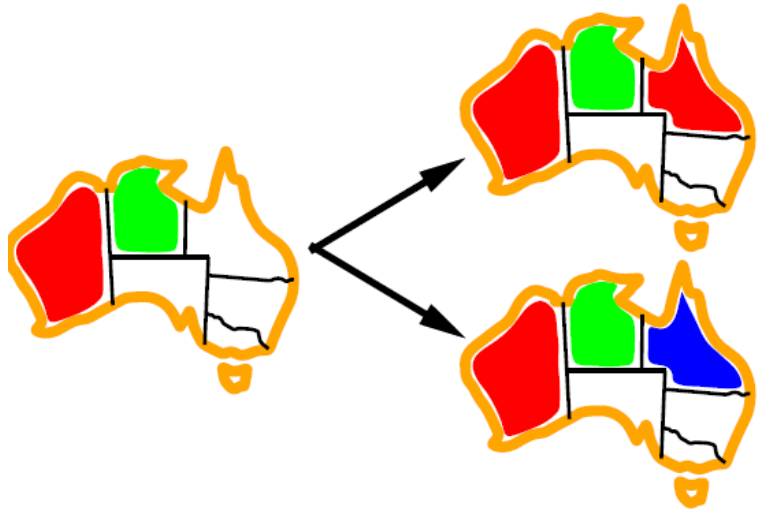
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal values left in its domain

- Why min rather than max?
- Also called "most constrained variable"
- "Fail-fast" ordering

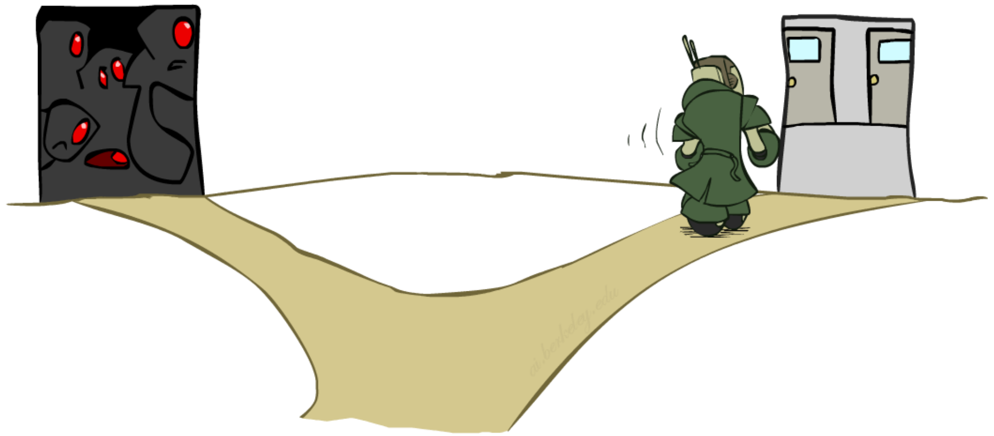


Ordering: Least Constraining Value

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



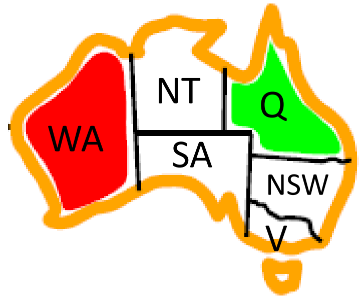
- Why least rather than most?

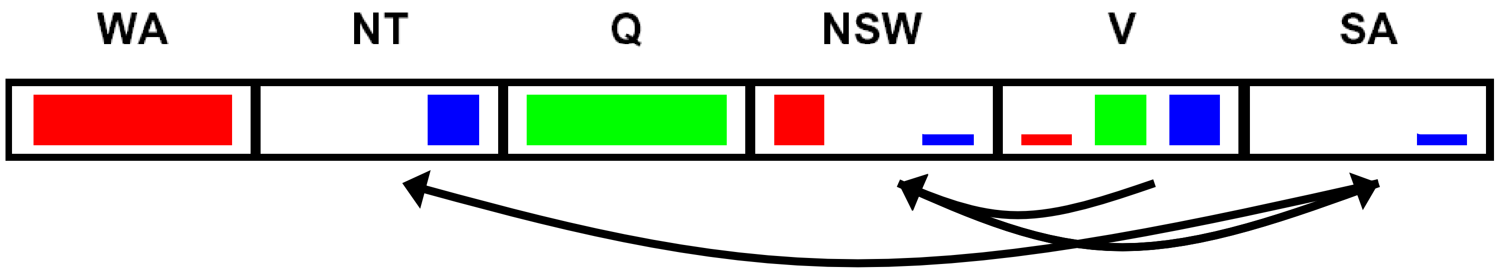


- Combining these ordering ideas makes 1000 queens feasible

Filtering: Arc Consistency

- A simple form of propagation makes sure all arcs are consistent:





- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

Remember: Delete from the tail!

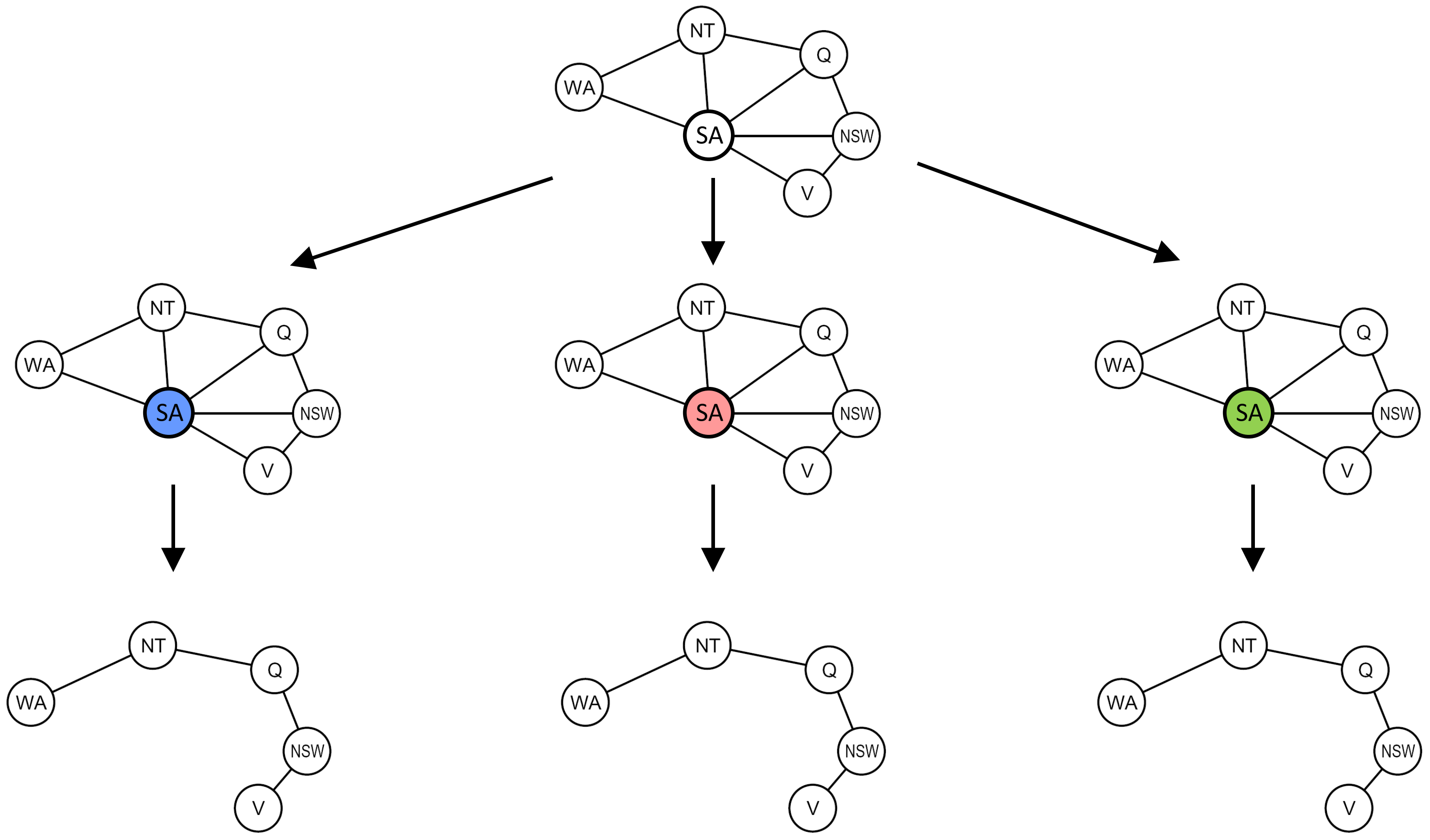
Leveraging Structure: Cutsets

Choose a cutset

Instantiate the cutset
(all possible ways)

Compute residual CSP
for each assignment

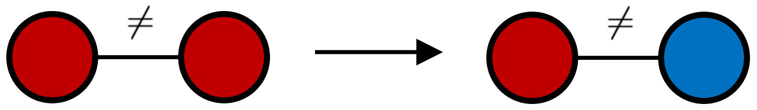
Solve the residual CSPs
(tree structured)



General Approach #2: Iterative Improvement

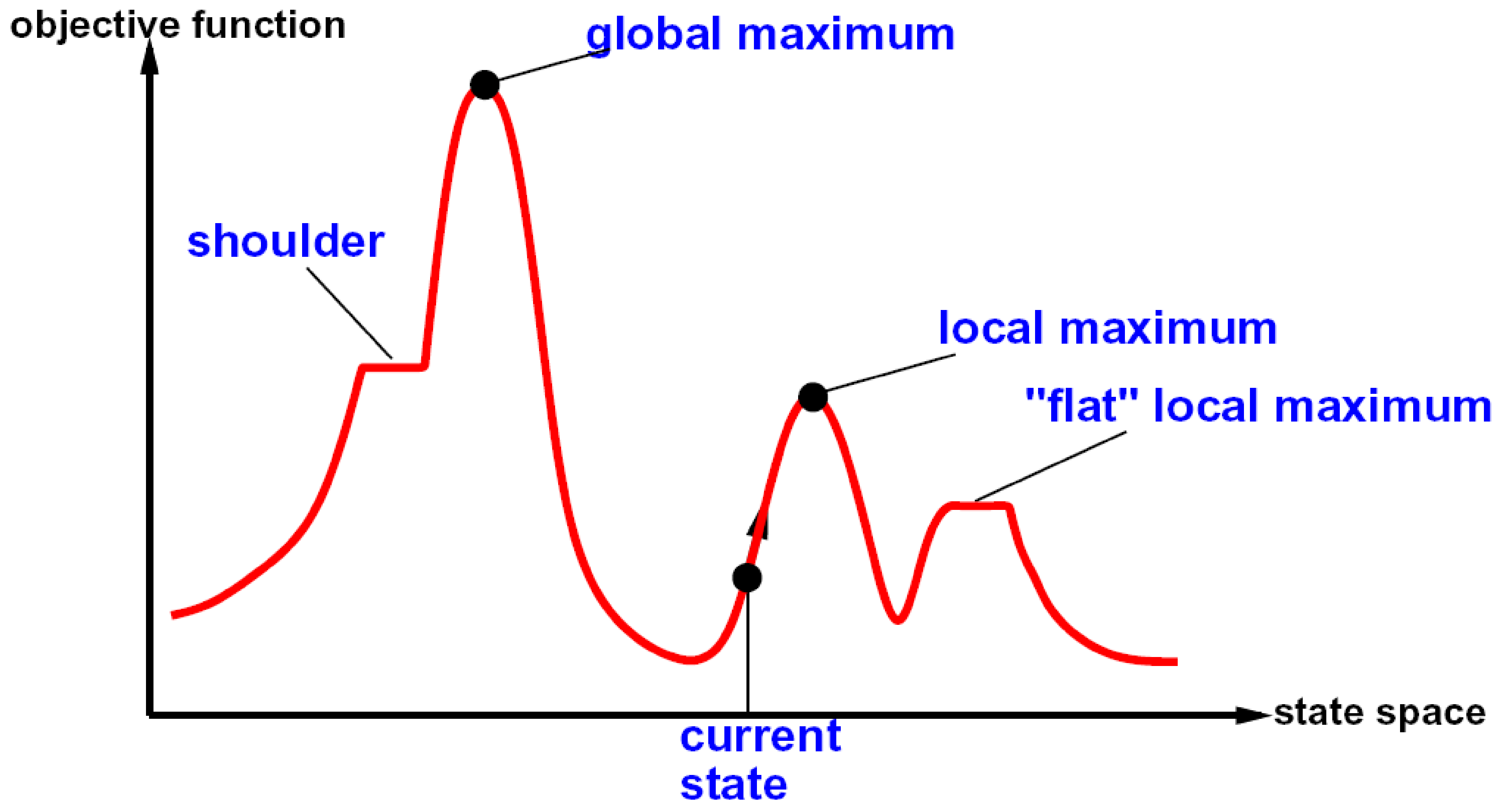
- Local search methods typically work with "complete" states, i.e., all variables assigned

- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.



- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest constraints
    - I.e., hill climb with h(x) = total number of violated constraints

Hill Climbing Diagram

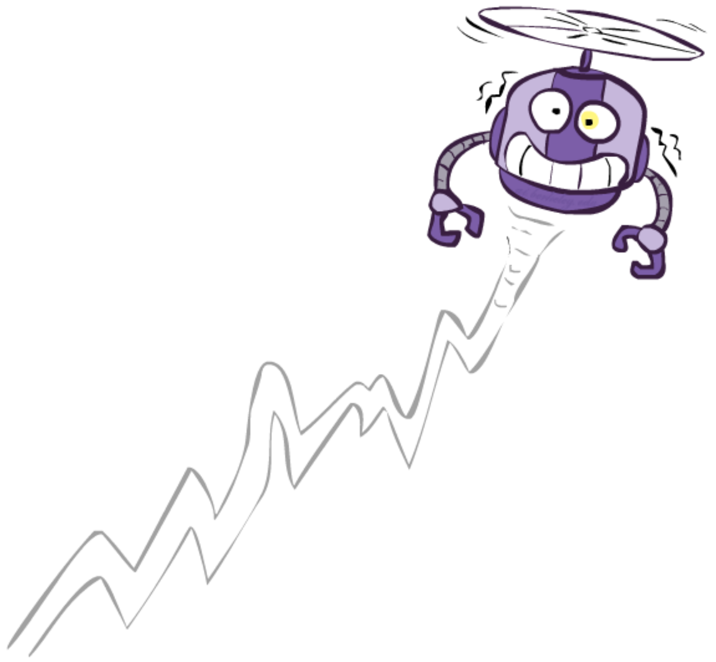


Simulated Annealing

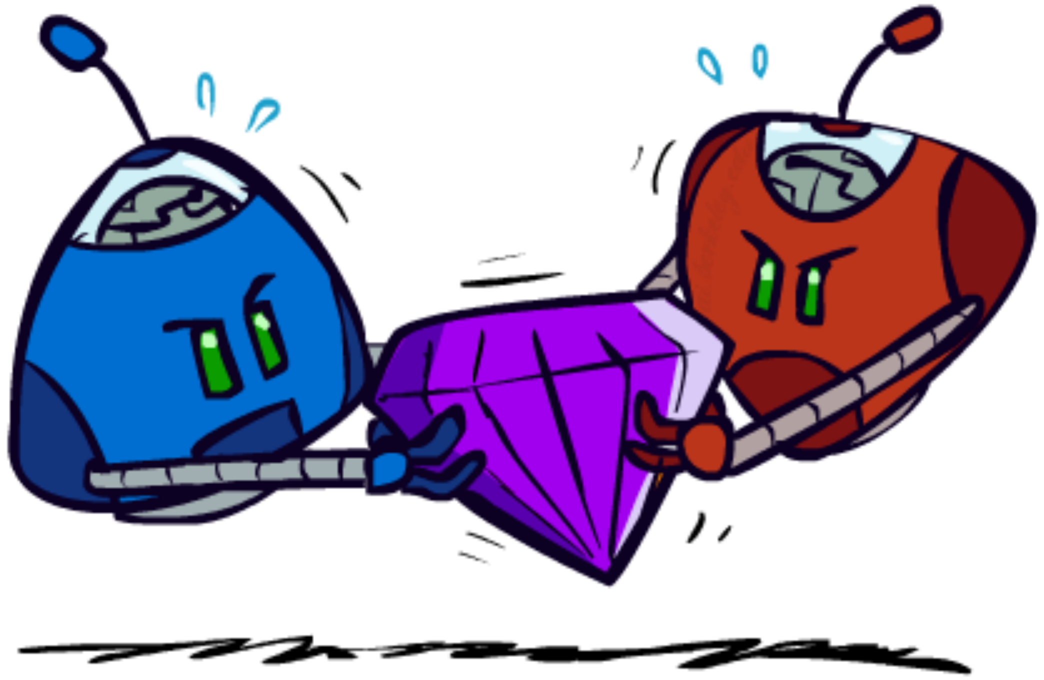
- Idea: Escape local maxima by allowing downhill moves
  - But make them rarer as time goes on

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

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



Game Trees

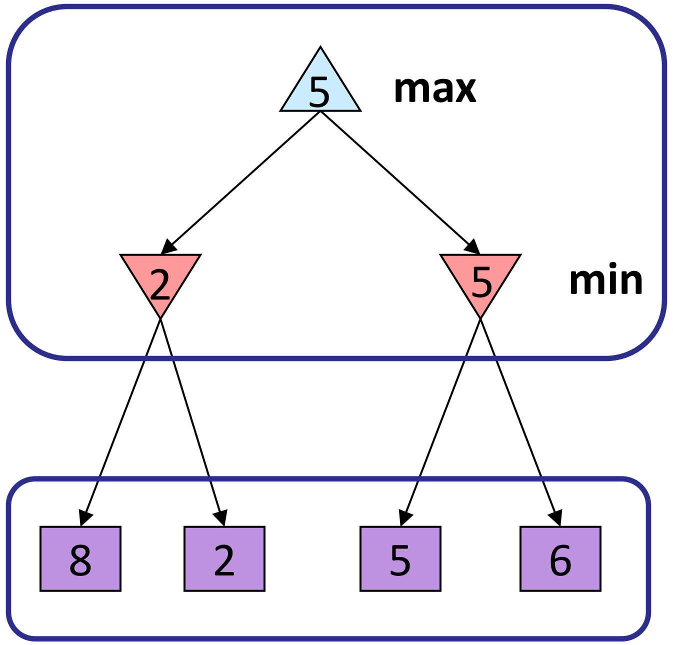


Adversarial Search (Minimax)

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

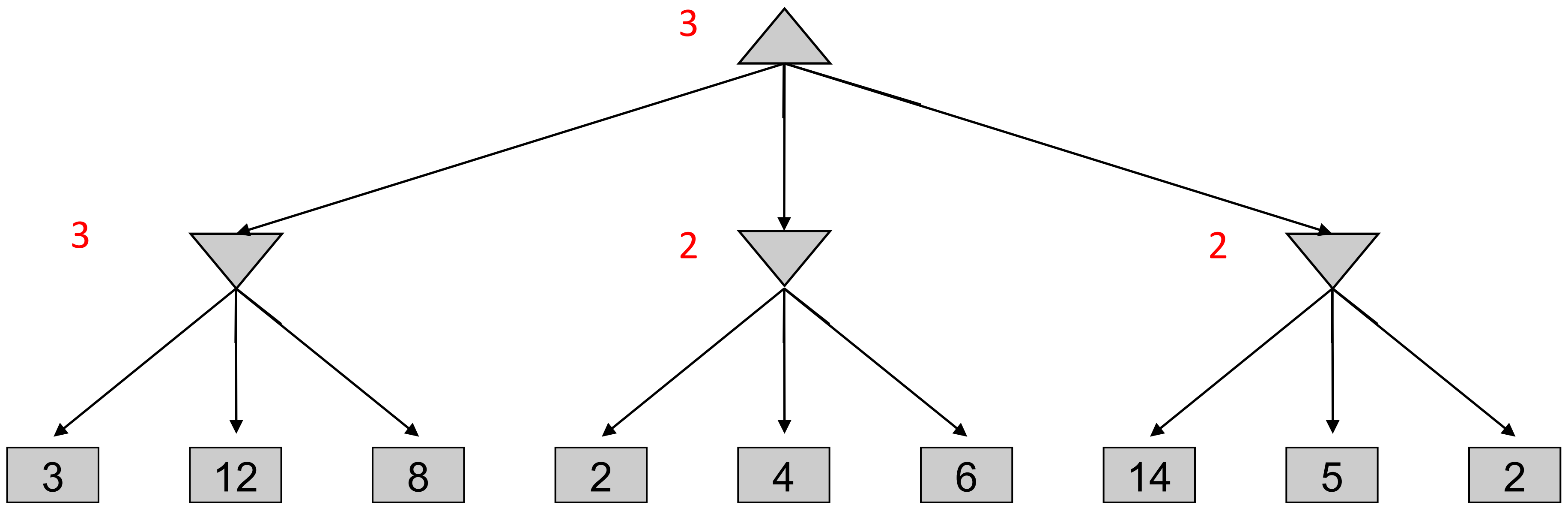
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

Minimax values: computed recursively

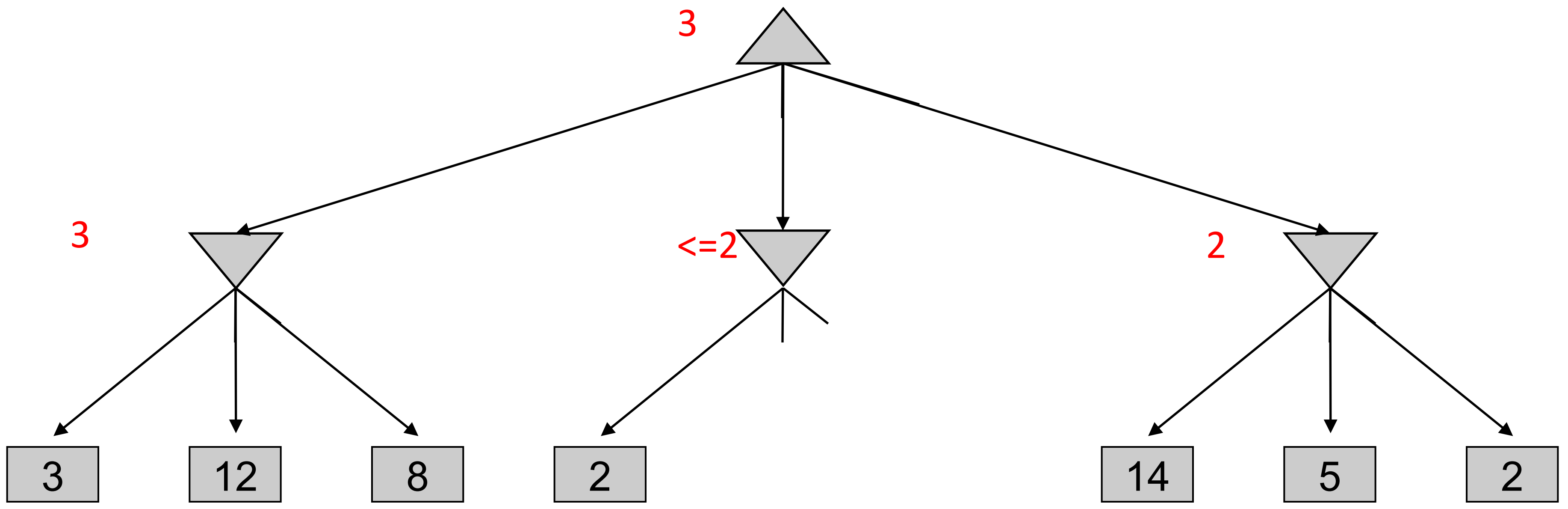


Terminal values: part of the game

Minimax Example

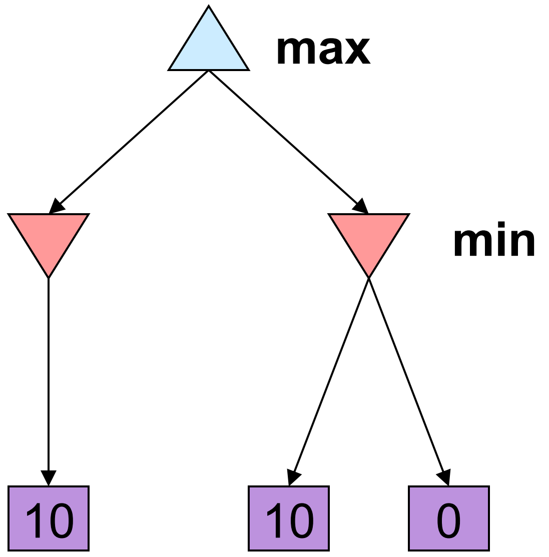


Minimax Example: Pruning

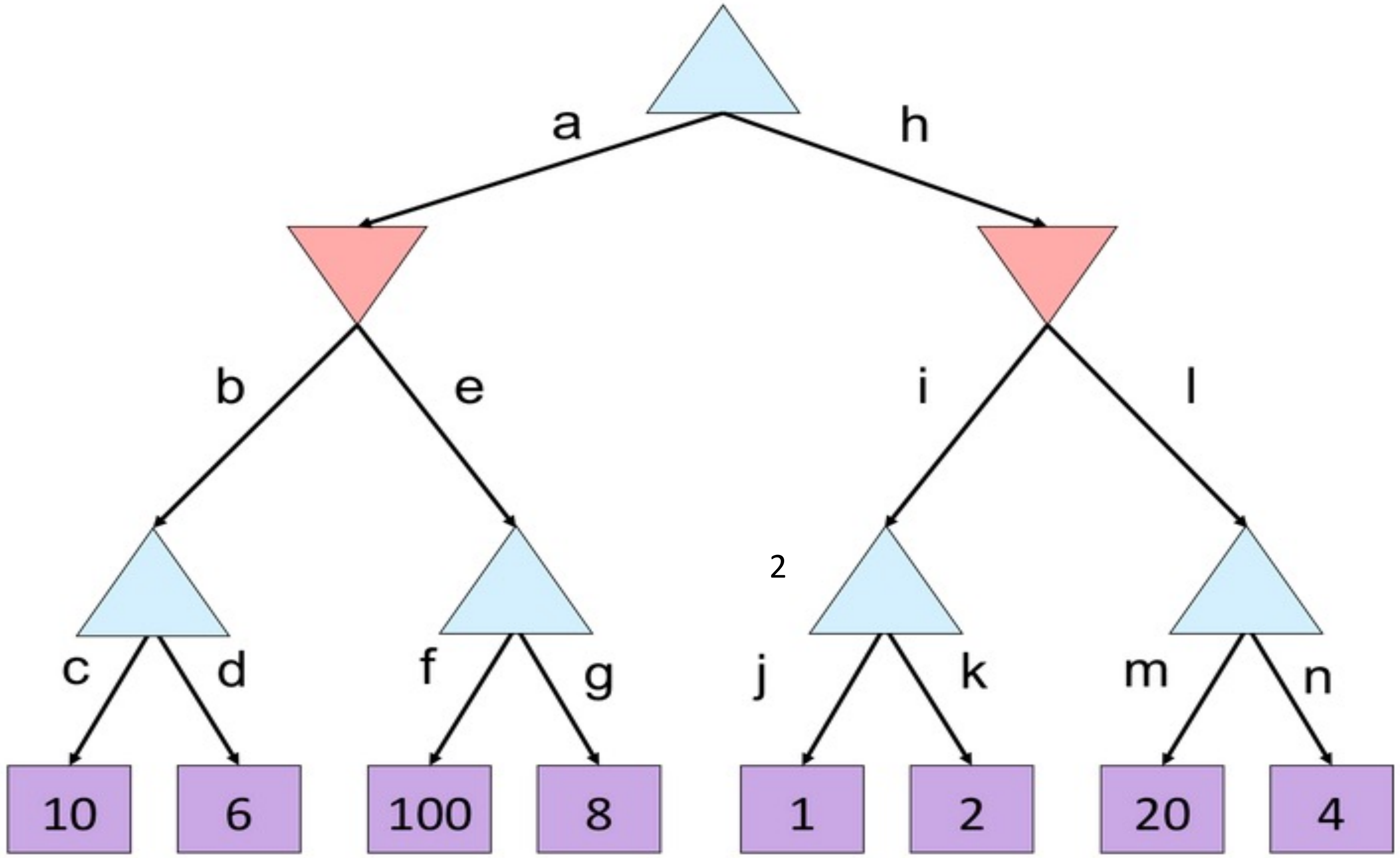


Alpha-Beta Pruning Properties

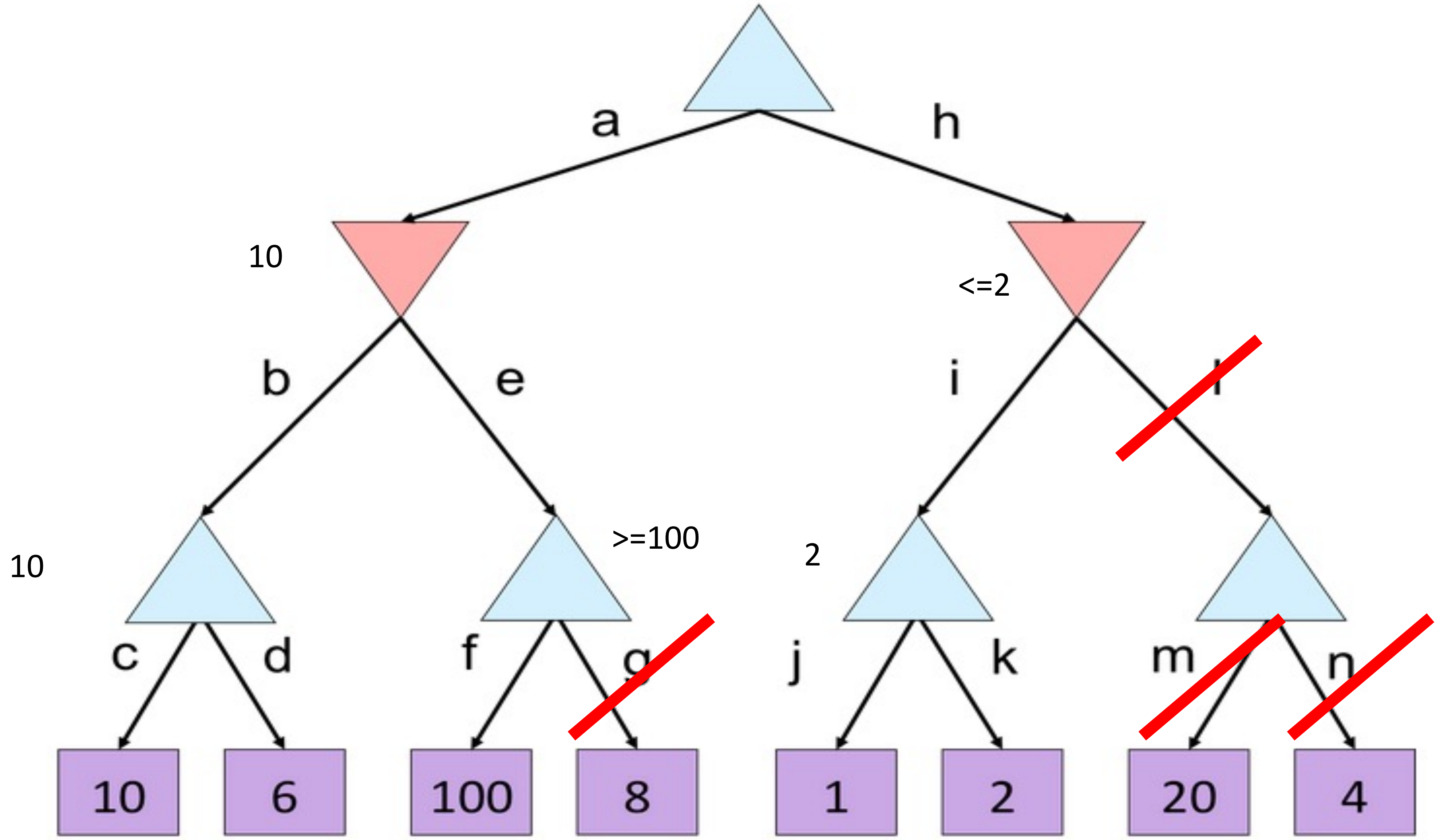
- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…
- This is a simple example of metareasoning (computing about what to compute)



Alpha-Beta Quiz 2



Alpha-Beta Quiz 2

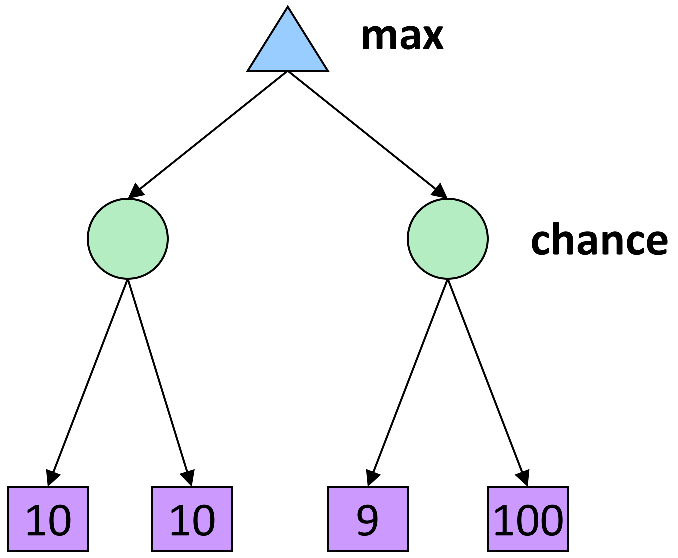


Expectimax Search

- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Unpredictable humans: humans are not perfect
  - 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



Remaining Topics

Bayes Nets:

- Inference by enumeration
- Variable elimination
- D-separation
- Sampling approaches

HMMs:

- Forward algorithm
- Viterbi algorithm
- Particle filtering

Decision networks and VPIs

Out of scope: learning theory, decision tree classifiers, details of non-SGD optimizers (e.g., NAG, Adagrad, Adam), NLP/CV/RL