Lecture 6: Search II

Announcements

- For homework 2 (sentiment), Thursday 11:00pm is 2 late day **hard deadline**.

- Section on Thursday at 3:30pm: search problems (uniform cost search, dynamic programming, A*)

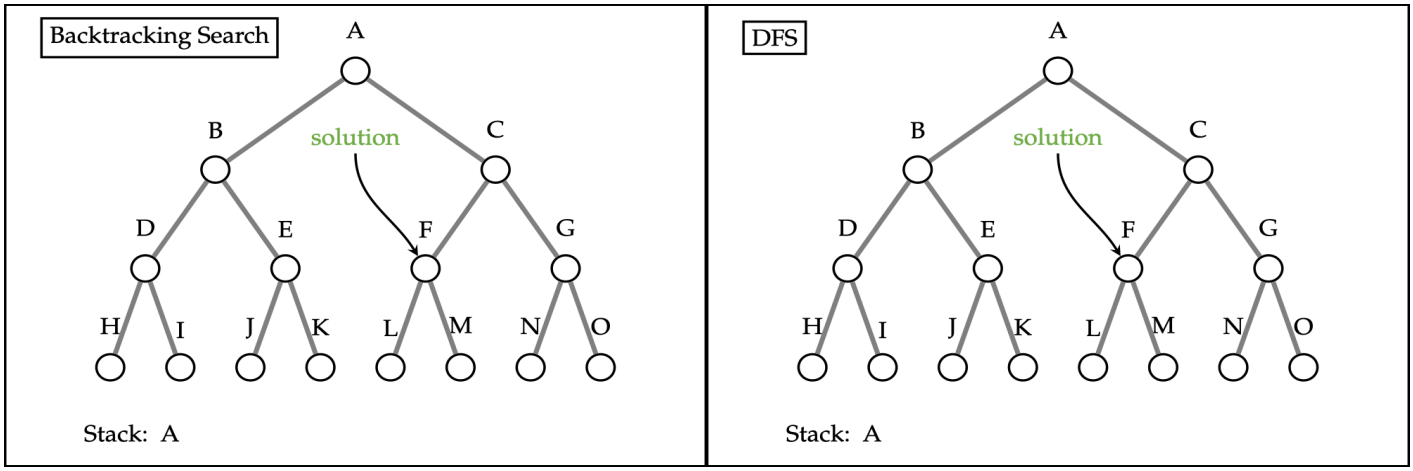
Previous Lecture

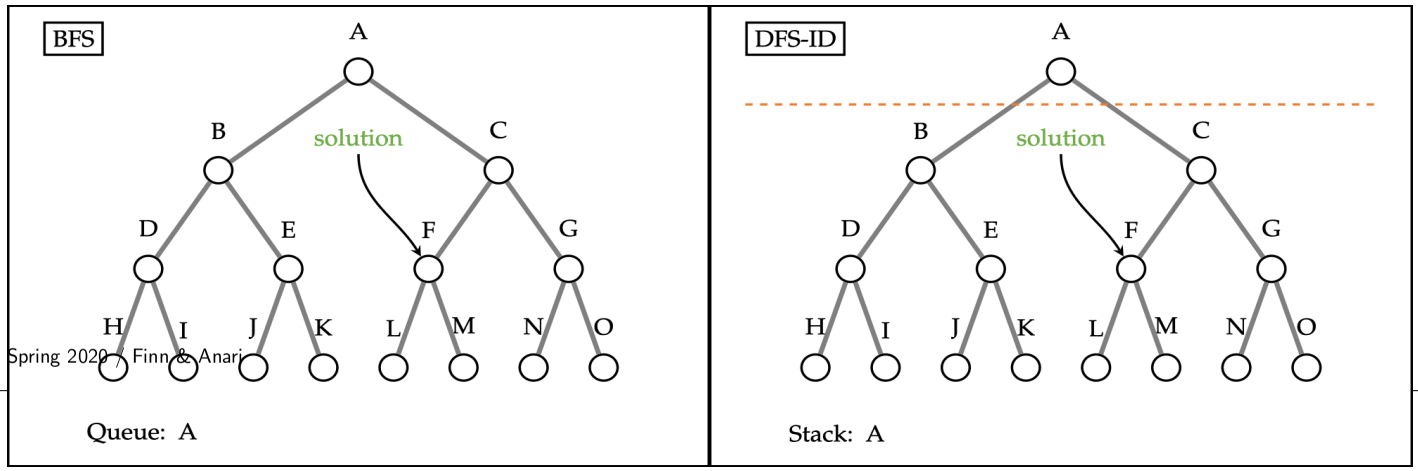
Tree search

Dynamic programming

Uniform cost search

Tree Search Review





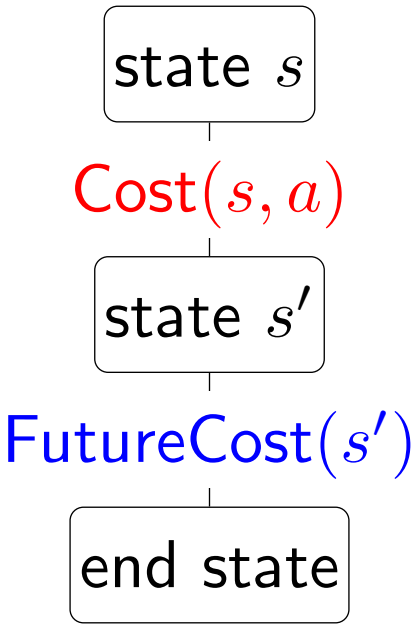
Previous Lecture

Tree search

Dynamic programming

Uniform cost search

Dynamic Programming Review



$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s,a) + \text{FutureCost}(\text{Succ}(s,a))] & \text{otherwise} \end{cases}$$

Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

Previous Lecture

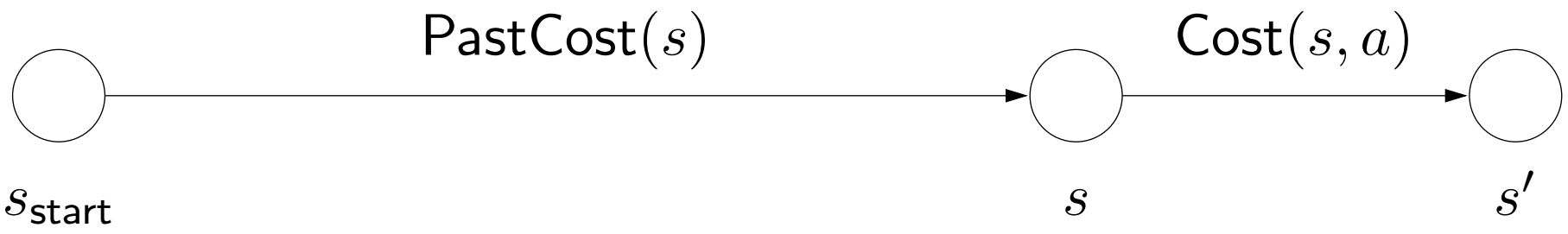
Tree search

Dynamic programming

Uniform cost search

Ordering the states

Observation: prefixes of optimal path are optimal



Key: if graph is acyclic, dynamic programming makes sure we compute PastCost(s) before PastCost(s')

If graph is cyclic, then we need another mechanism to order states...

- Recall that we used dynamic programming to compute the future cost of each state s, the cost of the minimum cost path from s to a end state.
- We can analogously define PastCost(s), the cost of the minimum cost path from the start state to s. If instead of having access to the successors via Succ(s, a), we had access to predecessors (think of reversing the edges in the state graph), then we could define a dynamic program to compute all the PastCost(s).
- Dynamic programming relies on the absence of cycles, so that there is always a clear order in which to compute all the past costs. If the past costs of all the predecessors of a state s are computed, then we could compute the past cost of s by taking the minimum.
- Note that PastCost(s) will always be computed before PastCost(s') if there is an edge from s to s'. In essence, the past costs will be computed according to a topological ordering of the nodes.
- However, when there are cycles, no topological ordering exists, so we need another way to order the states.

Uniform cost search (UCS)

Key idea: state ordering

UCS enumerates states in order of increasing past cost.

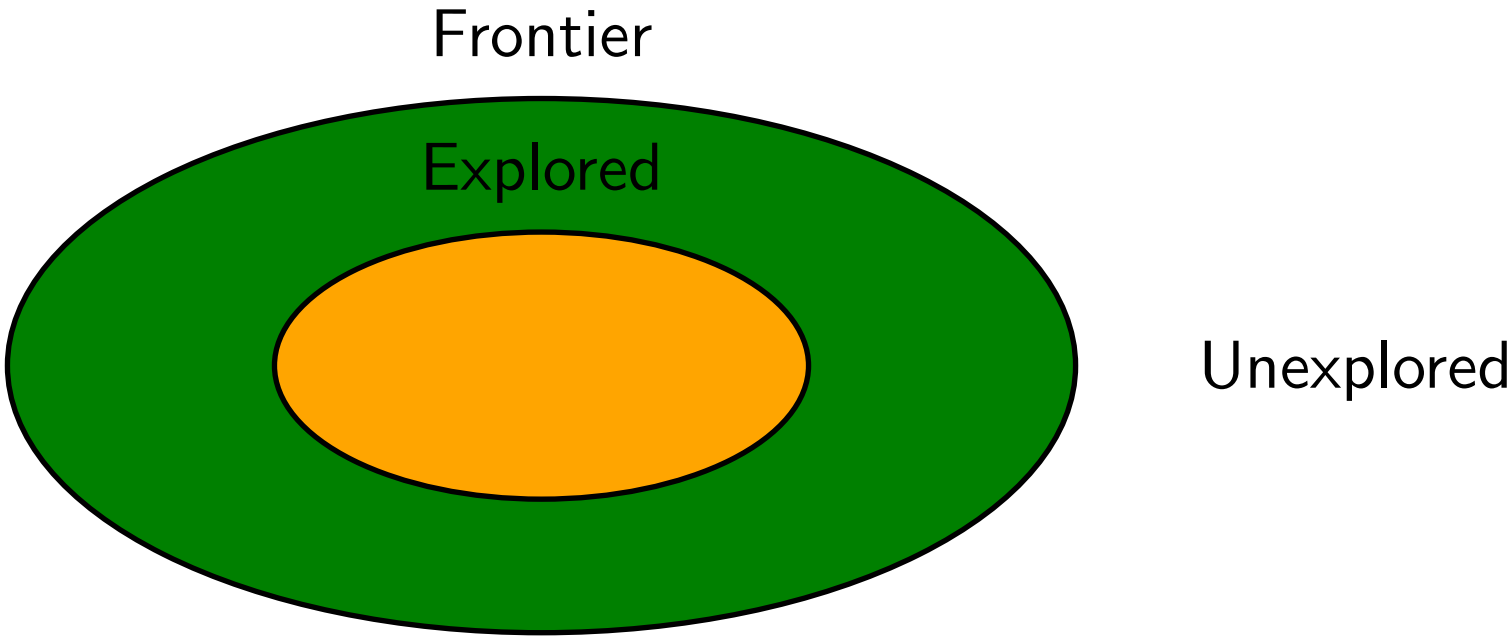
Assumption: non-negativity

All action costs are non-negative: $\text{Cost}(s, a) \geq 0$.

UCS in action:

- The key idea that uniform cost search (UCS) uses is to compute the past costs in order of increasing past cost. To make this efficient, we need to make an important assumption that all action costs are non-negative.
- This assumption is reasonable in many cases, but doesn't allow us to handle cases where actions have payoff. To handle negative costs (positive payoffs), we need the Bellman-Ford algorithm. When we talk about value iteration for MDPs, we will see a form of this algorithm.
- Note: those of you who have studied algorithms should immediately recognize UCS as Dijkstra's algorithm. Logically, the two are indeed equivalent. There is an important implementation difference: UCS takes as input a **search problem**, which implicitly defines a large and even infinite graph, whereas Dijkstra's algorithm (in the typical exposition) takes as input a fully concrete graph. The implicitness is important in practice because we might be working with an enormous graph (a detailed map of world) but only need to find the path between two close by points (Stanford to Palo Alto).
- Another difference is that Dijkstra's algorithm is usually thought of as finding the shortest path from the start state to every other node, whereas UCS is explicitly about finding the shortest path to an end state. This difference is sharpened when we look at the A* algorithm next time, where knowing that we're trying to get to the goal can yield a much faster algorithm. The name uniform cost search refers to the fact that we are exploring states of the same past cost uniformly (the video makes this visually clear); in contrast, A* will explore states which are biased towards the end state.

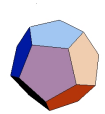
High-level strategy



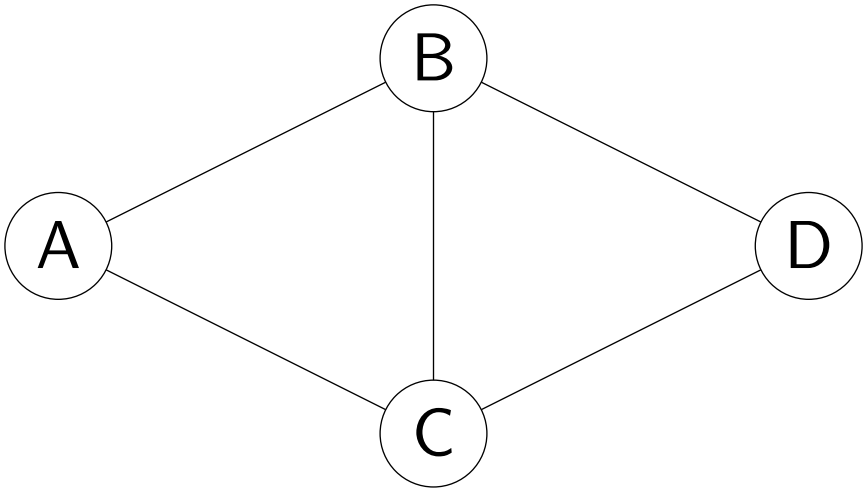
- Explored: states we've found the optimal path to
- Frontier: states we've seen, still figuring out how to get there cheaply
- Unexplored: states we haven't seen

- The general strategy of UCS is to maintain three sets of nodes: explored, frontier, and unexplored. Throughout the course of the algorithm, we will move states from unexplored to frontier, and from frontier to explored.
- The key invariant is that we have computed the minimum cost paths to all the nodes in the explored set. So when the end state moves into the explored set, then we are done.

Uniform cost search example



Example: UCS example



Start state: A, end state: D

[whiteboard]

Minimum cost path:

$$A \rightarrow B \rightarrow C \rightarrow D \text{ with cost } 3$$

- Before we present the full algorithm, let's walk through a concrete example.
- Initially, we put A on the frontier. We then take A off the frontier and mark it as explored. We add B and C to the frontier with past costs 1 and 100, respectively.
- Next, we remove from the frontier the state with the minimum past cost (priority), which is B. We mark B as explored and consider successors A, C, D. We ignore A since it's already explored. The past cost of C gets updated from 100 to 2. We add D to the frontier with initial past cost 101.
- Next, we remove C from the frontier; its successors are A, B, D. A and B are already explored, so we only update D's past cost from 101 to 3.
- Finally, we pop D off the frontier, find that it's a end state, and terminate the search.

Uniform cost search (UCS)

Algorithm: uniform cost search [Dijkstra, 1956]

Add s_{start} to **frontier** (priority queue)

Repeat until frontier is empty:

- Remove s with smallest priority p from frontier
- If IsEnd(s): return solution
- Add s to **explored**
- For each action $a \in$ Actions(s):
  - Get successor $s' \leftarrow$ Succ(s, a)
  - If s' already in explored: continue
  - Update **frontier** with s' and priority $p +$ Cost(s, a)

[semi-live solution: `Uniform Cost Search`]

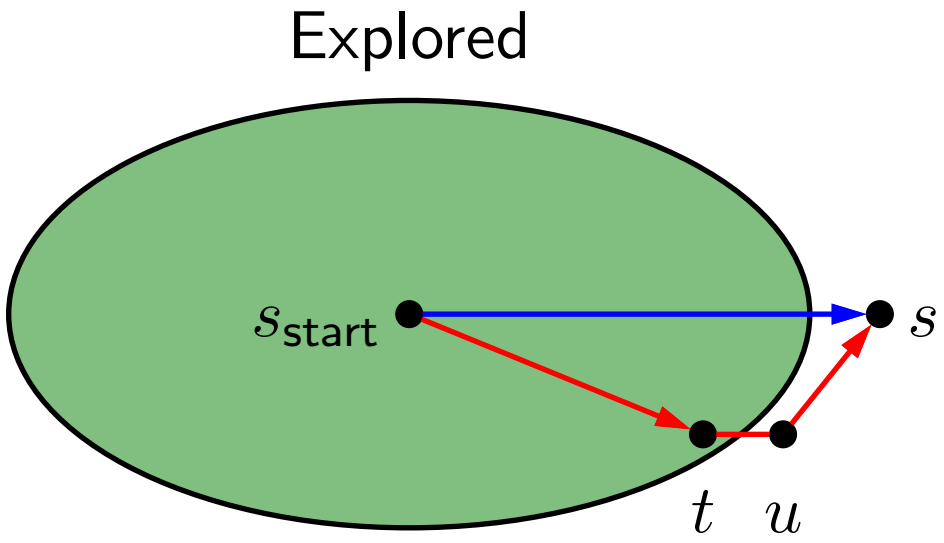
- Implementation note: we use `util.PriorityQueue` which supports `removeMin` and `update`. Note that `frontier.update(state, pastCost)` returns whether `pastCost` improves the existing estimate of the past cost of `state`.

Analysis of uniform cost search

Theorem: correctness

When a state s is popped from the frontier and moved to explored, its priority is PastCost(s), the minimum cost to s.

Proof:



- Let p_s be the priority of s when s is popped off the frontier. Since all costs are non-negative, p_s increases over the course of the algorithm.
- Suppose we pop s off the frontier. Let the blue path denote the path with cost p_s.
- Consider any alternative red path from the start state to s. The red path must leave the explored region at some point; let t and $u = \text{Succ}(t, a)$ be the first pair of states straddling the boundary. We want to show that the red path cannot be cheaper than the blue path via a string of inequalities.
- First, by definition of $\text{PastCost}(t)$ and non-negativity of edge costs, the cost of the red path is at least the cost of the part leading to u, which is $\text{PastCost}(t) + \text{Cost}(t, a) = p_t + \text{Cost}(t, a)$, where the last equality is by the inductive hypothesis.
- Second, we have $p_t + \text{Cost}(t, a) \geq p_u$ since we updated the frontier based on (t, a).
- Third, we have that $p_u \geq p_s$ because s was the minimum cost state on the frontier.
- Note that p_s is the cost of the blue path.

DP versus UCS

N total states, n of which are closer than end state

Algorithm	**Cycles?**	**Action costs**	**Time/space**
DP	no	any	$O(N)$
UCS	yes	≥ 0	$O(n \log n)$

Note: UCS potentially explores fewer states, but requires more overhead to maintain the priority queue

Note: assume number of actions per state is constant (independent of n and N)

- DP and UCS have complementary strengths and weaknesses; neither dominates the other.
- DP can handle negative action costs, but is restricted to acyclic graphs. It also explores all N reachable states from s_{start}, which is inefficient. This is unavoidable due to negative action costs.
- UCS can handle cyclic graphs, but is restricted to non-negative action costs. An advantage is that it only needs to explore n states, where n is the number of states which are cheaper to get to than any end state. However, there is an overhead with maintaining the priority queue.
- One might find it unsatisfying that UCS can only deal with non-negative action costs. Can we just add a large positive constant to each action cost to make them all non-negative? It turns out this doesn't work because it penalizes longer paths more than shorter paths, so we would end up solving a different problem.

answer in chat

Question

Suppose we want to travel from city 1 to city n (going only forward) and back to city 1 (only going backward). It costs $c_{ij} \geq 0$ to go from i to j. Which of the following algorithms can be used to find the minimum cost path (select all that apply)?

- depth-first search
- breadth-first search
- dynamic programming
- uniform cost search

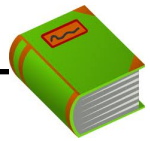
- Let's first start by figuring out what the search problem actually is. Any action sequence needs to satisfy the constraint that we move forward to n and then move backwards to 1. So we need to keep track of the current city i as well as the direction in the state.
- We can write down the details, but all that matters for this question is that the graph is acyclic (note that the graph implied by c_{ij} over cities is not acyclic, but keeping track of directionality makes it acyclic). Also, all edge costs are non-negative.
- Now, let's think about which algorithms will work. Recall the various assumptions of the algorithms. DFS won't work because it assumes all edge costs are zero. BFS also won't work because it assumes all edge costs are the same. Dynamic programming will work because the graph is acyclic. Uniform cost search will also work because all the edge costs are non-negative.

Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

past actions (all cities)	1 3 4 6 5 3
state (current city)	1 3 4 6 5 3

Review



Definition: search problem

- s_{start}: starting state
- Actions(s): possible actions
- Cost(s, a): action cost
- Succ(s, a): successor
- IsEnd(s): reached end state?

Objective: find the minimum cost path from s_{start} to an s satisfying IsEnd(s).

Paradigm

Modeling

Inference **Learning**

Roadmap

Learning costs

A* search

Relaxation

Search

Transportation example

Start state: 1

Walk action: from s to $s+1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

End state: n

search algorithm

walk walk tram tram tram walk tram tram

(minimum cost path)

- Recall the magic tram example from the last lecture. Given a search problem (specification of the start state, end test, actions, successors, and costs), we can use a search algorithm (DP or UCS) to yield a solution, which is a sequence of actions of minimum cost reaching an end state from the start state.

Learning

Transportation example

Start state: 1

Walk action: from s to $s+1$ (cost: **?**)

Tram action: from s to $2s$ (cost: **?**)

End state: n

walk walk tram tram tram walk tram tram

learning algorithm

walk cost: **1**, tram cost: **2**

- Now suppose we don't know what the costs are, but we observe someone getting from 1 to n via some sequence of walking and tram-taking. Can we figure out what the costs are? This is the goal of learning.

Learning as an inverse problem

Forward problem (search):

$$\text{Cost}(s, a) \longrightarrow (a_1, \ldots, a_k)$$

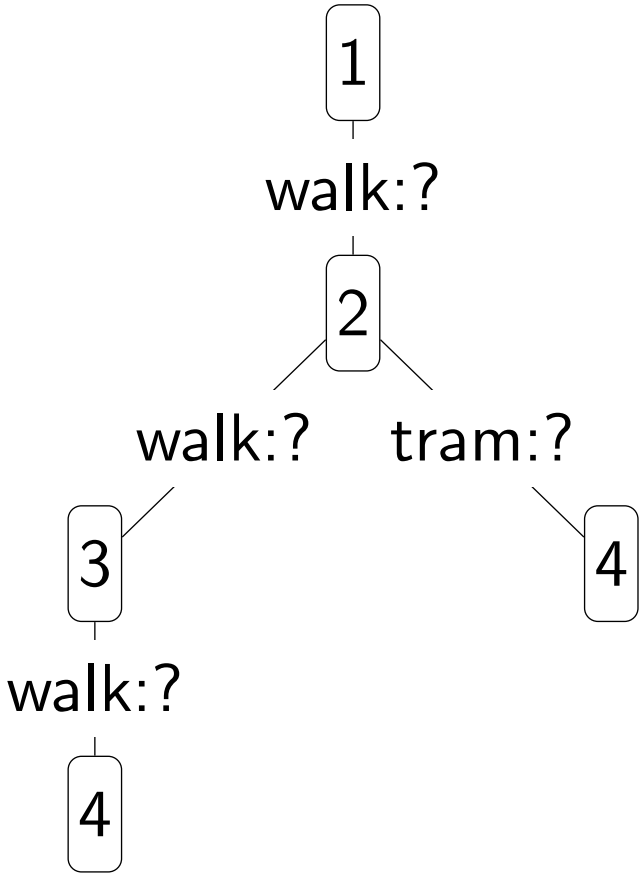
Inverse problem (learning):

$$(a_1, \ldots, a_k) \longrightarrow \text{Cost}(s, a)$$

- More generally, so far we have thought about search as a "forward" problem: given costs, finding the optimal sequence of actions.
- Learning concerns the "inverse" problem: given the desired sequence of actions, reverse engineer the costs.

Prediction (inference) problem

Input x: search problem without costs



Output y: solution path

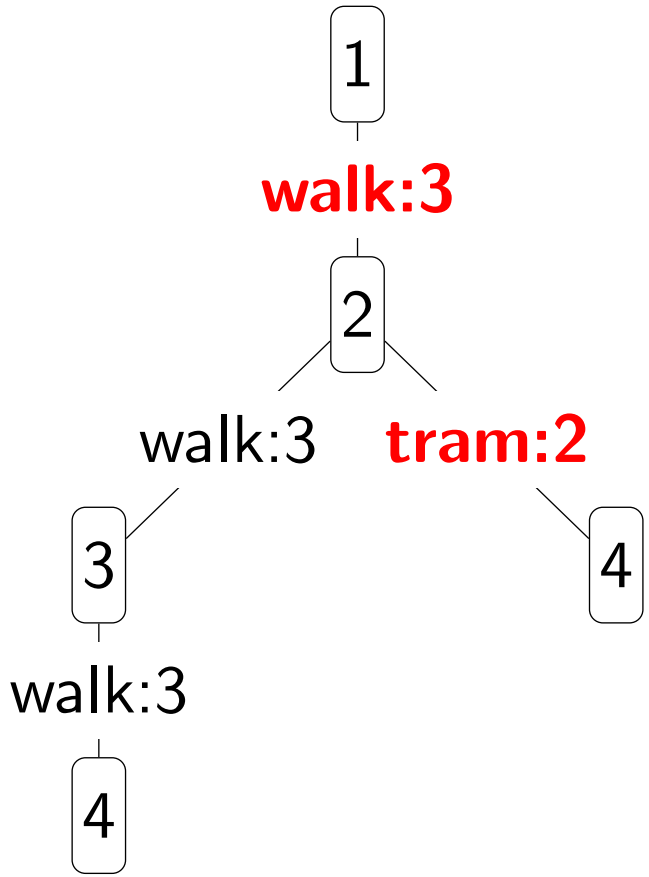
walk walk walk

- Let's cast the problem as predicting an output y given an input x. Here, the input x is the search problem (visualized as a search tree) without the costs provided. The output y is the desired solution path. The question is what the costs should be set to so that y is actually the minimum cost path of the resulting search problem.

Tweaking costs

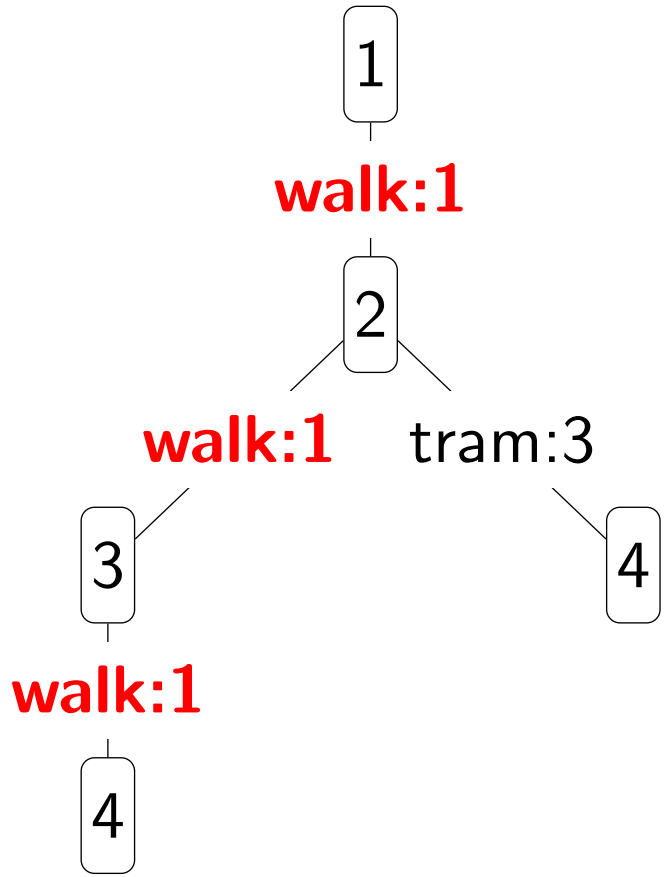
Costs: {walk:3, tram:2}

Minimum cost path:



Costs: {walk:1, tram:3}

Minimum cost path:



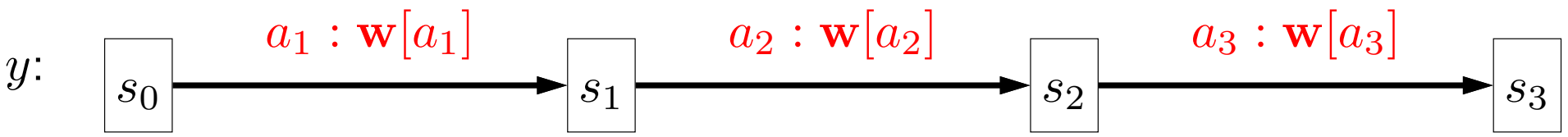
- Suppose the walk cost is 3 and the tram cost is 2. Then, we would obviously predict the [walk, tram] path, which has lower cost.
- But this is not our desired output, because we actually saw the person walk all the way from 1 to 4. How can we update the action costs so that the minimum cost path is walking?
- Intuitively, we want the tram cost to be more and the walk cost to be less. Specifically, let's increase the cost of every action on the predicted path and decrease the cost of every action on the true path. Now, the predicted path coincides with the true observed path. Is this a good strategy in general?

Modeling costs (simplified)

Assume costs depend only on the action:

$$\text{Cost}(s, a) = \mathbf{w}[a]$$

Candidate output path:



Path cost:

$$\text{Cost}(y) = \mathbf{w}[a_1] + \mathbf{w}[a_2] + \mathbf{w}[a_3]$$

- For each action a, we define a weight $\mathbf{w}[a]$ representing the cost of action a. Without loss of generality, let us assume that the cost of the action does not depend on the state s.
- Then the cost of a path y is simply the sum of the weights of the actions on the path. Every path has some cost, and recall that the search algorithm will return the minimum cost path.

Learning algorithm

Algorithm: Structured Perceptron (simplified)

- For each action: $\mathbf{w}[a] \leftarrow 0$
- For each iteration $t = 1, \ldots T$:
  - For each training example $(x, y) \in \mathcal{D}_{\text{train}}$:
    - Compute the minimum cost path y' given $\mathbf{w}$
    - For each action $a \in y$: $\mathbf{w}[a] \leftarrow \mathbf{w}[a] - 1$
    - For each action $a \in y'$: $\mathbf{w}[a] \leftarrow \mathbf{w}[a] + 1$

- Try to decrease cost of true y (from training data)
- Try to increase cost of predicted y' (from search)

[semi-live solution]

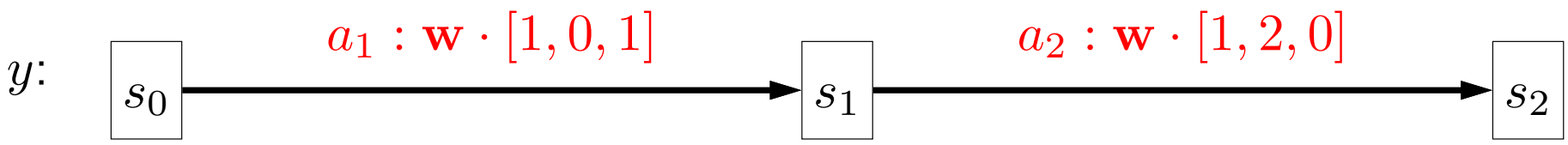
- We are now in position to state the (simplified version of) **structured Perceptron** algorithm.
- Advanced: the Perceptron algorithm performs stochastic gradient descent (SGD) on a modified hinge loss with a constant step size of $\eta = 1$. The modified hinge loss is $\text{Loss}(x, y, \mathbf{w}) = \max\{-(\mathbf{w} \cdot \phi(x))y, 0\}$, where the margin of 1 has been replaced with a zero. The structured Perceptron is a generalization of the Perceptron algorithm, which is stochastic gradient descent on $\text{Loss}(x, y, \mathbf{w}) = \max_{y'}\{\sum_{a \in y} \mathbf{w}[a] - \sum_{a \in y'} \mathbf{w}[a]\}$ (note the relationship to the multiclass hinge loss). Even if you don't really understand the loss function, you can still understand the algorithm, since it is very intuitive.
- We iterate over the training examples. Each (x, y) is a tuple where x is a search problem without costs and y is the true minimum-cost path. Given the current weights w (action costs), we run a search algorithm to find the minimum-cost path y' according to those weights. Then we update the weights to favor actions that appear in the correct output y (by reducing their costs) and disfavor actions that appear in the predicted output y' (by increasing their costs). Note that if we are not making a mistake (that is, if $y = y'$), then there is no update.
- Collins (2002) proved (based on the proof of the original Perceptron algorithm) that if there exists a weight vector that will make zero mistakes on the training data, then the Perceptron algorithm will converge to one of those weight vectors in a finite number of iterations.

Generalization to features (skip)

Costs are parametrized by feature vector:

$$\text{Cost}(s, a) = \mathbf{w} \cdot \phi(s, a)$$

Example:



$$\mathbf{w} = [3, -1, -1]$$

Path cost:

$$\text{Cost}(y) = 2 + 1 = 3$$

- So far, the cost of an action a is simply $\mathbf{w}[a]$. We can generalize this to allow the cost to be a general dot product $\mathbf{w} \cdot \phi(s, a)$, which (i) allows the features to depend on both the state and the action and (ii) allows multiple features per edge. For example, we can have different costs for walking and tram-taking depending on which part of the city we are in.
- We can equivalently write the cost of an entire output y as $\mathbf{w} \cdot \phi(y)$, where $\phi(y) = \phi(s_0, a_1) + \phi(s_1, a_2)$ is the sum of the feature vectors over all actions.

Learning algorithm (skip)

Algorithm: Structured Perceptron [Collins, 2002]

- For each action: $\mathbf{w} \leftarrow 0$
- For each iteration $t = 1, \ldots T$:
  - For each training example $(x, y) \in \mathcal{D}_{\text{train}}$:
    - Compute the minimum cost path y' given $\mathbf{w}$
    - $\mathbf{w} \leftarrow \mathbf{w} - \phi(y) + \phi(y')$

- Try to decrease cost of true y (from training data)
- Try to increase cost of predicted y' (from search)

Applications

- Part-of-speech tagging

Fruit flies like a banana.  Noun Noun Verb Det Noun

- Machine translation

la maison bleue  *the blue house*

- The structured Perceptron was first used for natural language processing tasks. Given it's simplicity, the Perceptron works reasonably well. With a few minor tweaks, you get state-of-the-art algorithms for structured prediction, which can be applied to many tasks such as machine translation, gene prediction, information extraction, etc.
- On a historical note, the structured Perceptron merges two relatively classic communities. The first is search algorithms (uniform cost search was developed by Dijkstra in 1956). The second is machine learning (Perceptron was developed by Rosenblatt in 1957). It was only over 40 years later that the two met.

Roadmap

Learning costs

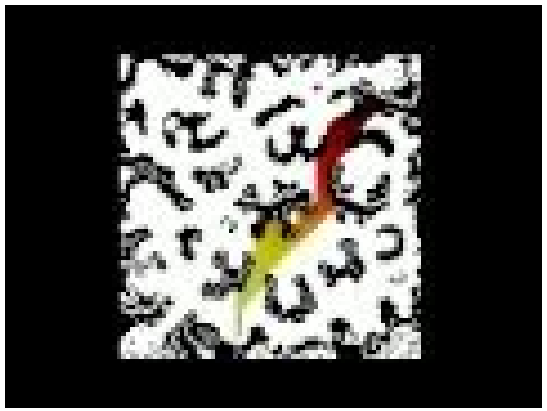
A* search

Relaxation

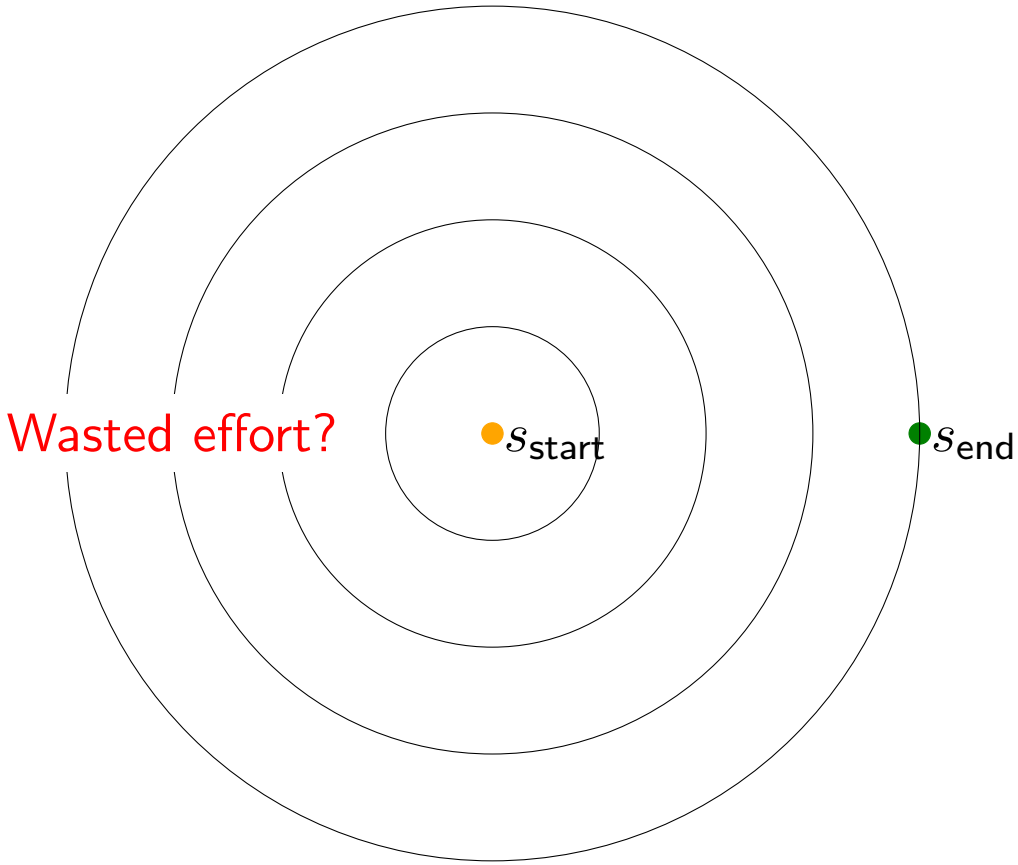
A* algorithm

UCS in action:

A* in action:



Can uniform cost search be improved?



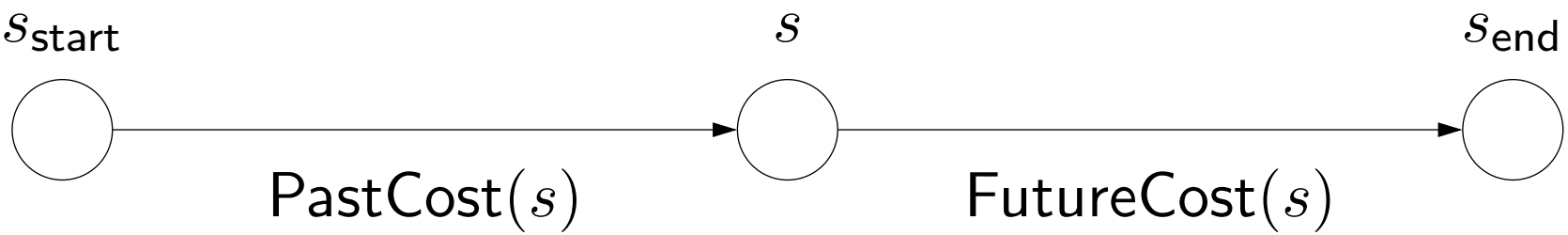
Problem: UCS orders states by cost from s_{start} to s

Goal: take into account cost from s to s_{end}

- Now our goal is to make UCS faster. If we look at the UCS algorithm, we see that it explores states based on how far they are away from the start state. As a result, it will explore many states which are close to the start state, but in the opposite direction of the end state.
- Intuitively, we'd like to bias UCS towards exploring states which are closer to the end state, and that's exactly what A* does.

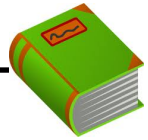
Exploring states

UCS: explore states in order of PastCost(s)



Ideal: explore in order of PastCost(s) + FutureCost(s)

A*: explore in order of PastCost(s) + $h(s)$



Definition: Heuristic function

A heuristic $h(s)$ is any estimate of FutureCost(s).

- First, some terminology: PastCost(s) is the minimum cost from the start state to s, and FutureCost(s) is the minimum cost from s to an end state. Without loss of generality, we can just assume we have one end state. (If we have multiple ones, create a new official goal state which is the successor of all the original end states.)
- Recall that UCS explores states in order of PastCost(s). It'd be nice if we could explore states in order of PastCost(s) + FutureCost(s), which would definitely take the end state into account, but computing FutureCost(s) would be as expensive as solving the original problem.
- A* relies on a **heuristic** $h(s)$, which is an estimate of FutureCost(s). For A* to work, $h(s)$ must satisfy some conditions, but for now, just think of $h(s)$ as an approximation. We will soon show that A* will explore states in order of PastCost(s) + $h(s)$. This is nice, because now states which are estimated (by $h(s)$) to be really far away from the end state will be explored later, even if their PastCost(s) is small.

A* search

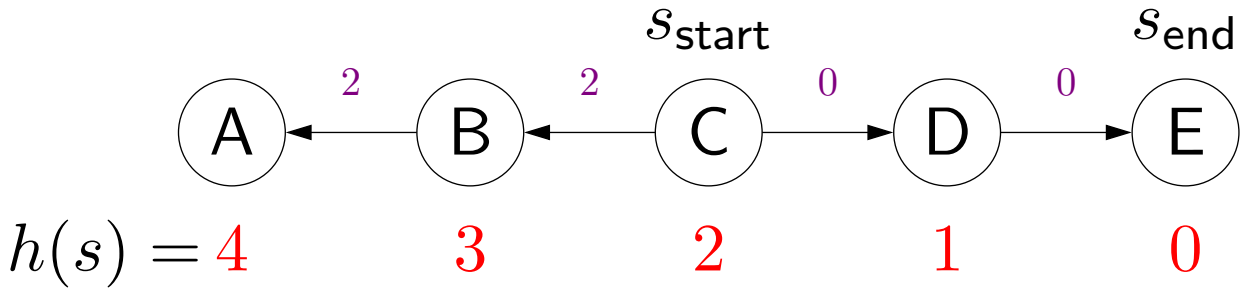
Algorithm: A* search [Hart/Nilsson/Raphael, 1968]

Run uniform cost search with **modified edge costs**:

$$\text{Cost}'(s,a) = \text{Cost}(s,a) + h(\text{Succ}(s,a)) - h(s)$$

Intuition: add a penalty for how much action a takes us away from the end state

Example:



$$\text{Cost}'(C,B) = \text{Cost}(C,B) + h(B) - h(C) = 1 + (3-2) = 2$$

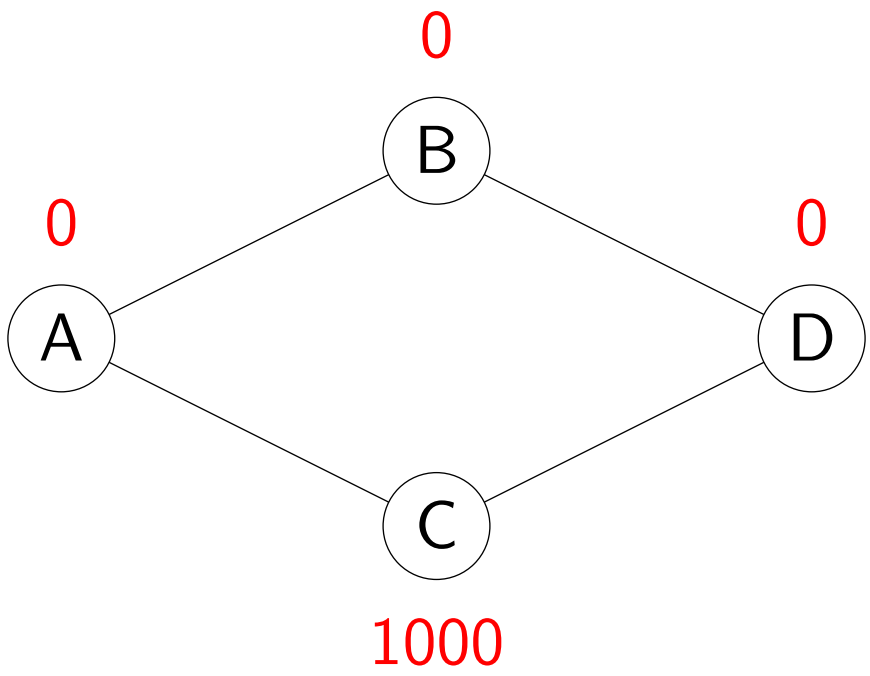
- Here is the full A* algorithm: just run UCS with modified edge costs.
- You might feel tricked because we promised you a shiny new algorithm, but actually, you just got a refurbished version of UCS. (This is a slightly unorthodox presentation of A*. The normal presentation is modifying UCS to prioritize by PastCost$(s) + h(s)$ rather than PastCost(s).) But I think the modified edge costs view shows a deeper connection to UCS, and we don't even have to modify the UCS code at all.
- How should we think of these modified edge costs? It's the same edge cost Cost(s, a) plus an additional term. This term is difference between the estimated future cost of the new state Succ(s, a) and that of the current state s. In other words, we're measuring how much farther from the end state does action a take us. If this difference is positive, then we're penalizing the action a more. If this difference is negative, then we're favoring this action a.
- Let's look at a small example. All edge costs are 1. Let's suppose we define $h(s)$ to be the actual FutureCost(s), the minimum cost to the end state. In general, this is not the case, but let's see what happens in the best case. The modified edge costs are 2 for actions moving away from the end state and 0 for actions moving towards the end state.
- In this case, UCS with original edge costs 1 will explore all the nodes. However, A* (UCS with modified edge costs) will explore only the three nodes on the path to the end state.

An example heuristic

Will any heuristic work?

No.

Counterexample:



Doesn't work because of **negative modified edge costs**!

- So far, we've just said that $h(s)$ is just an approximation of FutureCost(s). But can it be any approximation?
- The answer is no, as the counterexample clearly shows. The modified edge costs would be 1 (A to B), 1002 (A to C), 5 (B to D), and -999 (C to D). UCS would go to B first and then to D, finding a cost 6 path rather than the optimal cost 3 path through C.
- If our heuristic is lying to us (bad approximation of future costs), then running A* (UCS on modified costs) could lead to a suboptimal solution. Note that the reason this heuristic doesn't work is the same reason UCS doesn't work when there are negative action costs.

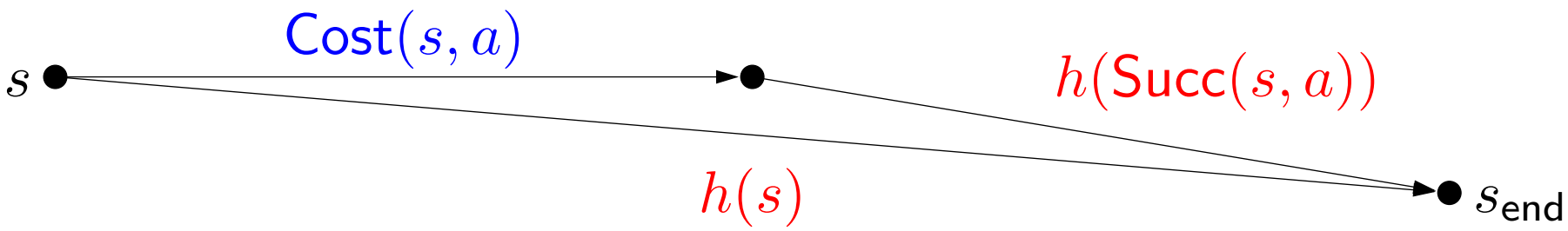
Consistent heuristics

Definition: consistency

A heuristic h is **consistent** if

- $\text{Cost}'(s,a) = \text{Cost}(s,a) + h(\text{Succ}(s,a)) - h(s) \geq 0$
- $h(s_{\text{end}}) = 0$.

Condition 1: needed for UCS to work (triangle inequality).



Condition 2: $\text{FutureCost}(s_{\text{end}}) = 0$ so match it.

- We need $h(s)$ to be **consistent**, which means two things. First, the modified edge costs are non-negative (this is the main property). This is important for UCS to find the minimum cost path (remember that UCS only works when all the edge costs are non-negative).
- Second, $h(s_{\text{end}}) = 0$, which is just saying: be reasonable. The minimum cost from the end state to the end state is trivially 0, so just use 0.
- We will come back later to the issue of getting a hold of a consistent heuristic, but for now, let's assume we have one and see what we can do with it.

Correctness of A*

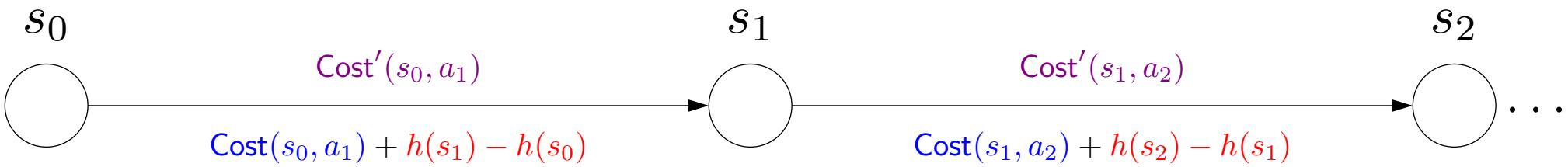
Proposition: correctness

If h is consistent, A* returns the minimum cost path.

- The main theoretical result for A* is that if we use any consistent heuristic, then we will be guaranteed to find the minimum cost path.

Proof of A* correctness

- Consider any path $[s_0, a_1, s_1, \ldots, a_L, s_L]$:



- Key identity:

$$\underbrace{\sum_{i=1}^{L} \text{Cost}'(s_{i-1}, a_i)}_{\text{modified path cost}} = \underbrace{\sum_{i=1}^{L} \text{Cost}(s_{i-1}, a_i)}_{\text{original path cost}} + \underbrace{h(s_L) - h(s_0)}_{\text{constant}}$$

- Therefore, A* (finding the minimum cost path using modified costs) solves the original problem (even though edge costs are all different!)

- To show the correctness of A*, let's take any path of length L from $s_0 = s_{\text{start}}$ to $s_L = s_{\text{end}}$. Let us compute the modified path cost by just adding up the modified edge costs. Just to simplify notation, let $c_i = \text{Cost}(s_{i-1}, a_i)$ and $h_i = h(s_i)$. The modified path cost is $(c_1 + h_1 - h_0) + (c_2 + h_2 - h_1) + \cdots + (c_L + h_L - h_{L-1})$. Notice that most of the h_i's actually cancel out (this is known as **telescoping sums**).
- We end up with $\sum_{i=1}^{L} c_i$, which is the original path cost plus $h_L - h_0$. First, notice that $h_L = 0$ because s_L is an end state and by the second condition of consistency, $h(s_L) = 0$. Second, h_0 is just a constant (in that it doesn't depend on the path at all), since all paths must start with the start state.
- Therefore, the modified path cost is equal to the original path cost plus a constant. A*, which is running UCS on the modified edge costs, is equivalent to running UCS on the original edge costs, which minimizes the original path cost.
- This is kind of remarkable: all the edge costs are modified in A*, but yet the final path cost is the same (up to a constant)!

Efficiency of A*

Theorem: efficiency of A*

A* explores all states s satisfying

$$\text{PastCost}(s) \leq \text{PastCost}(s_{\text{end}}) - h(s)$$

Interpretation: the larger $h(s)$, the better

Proof: A* explores all s such that

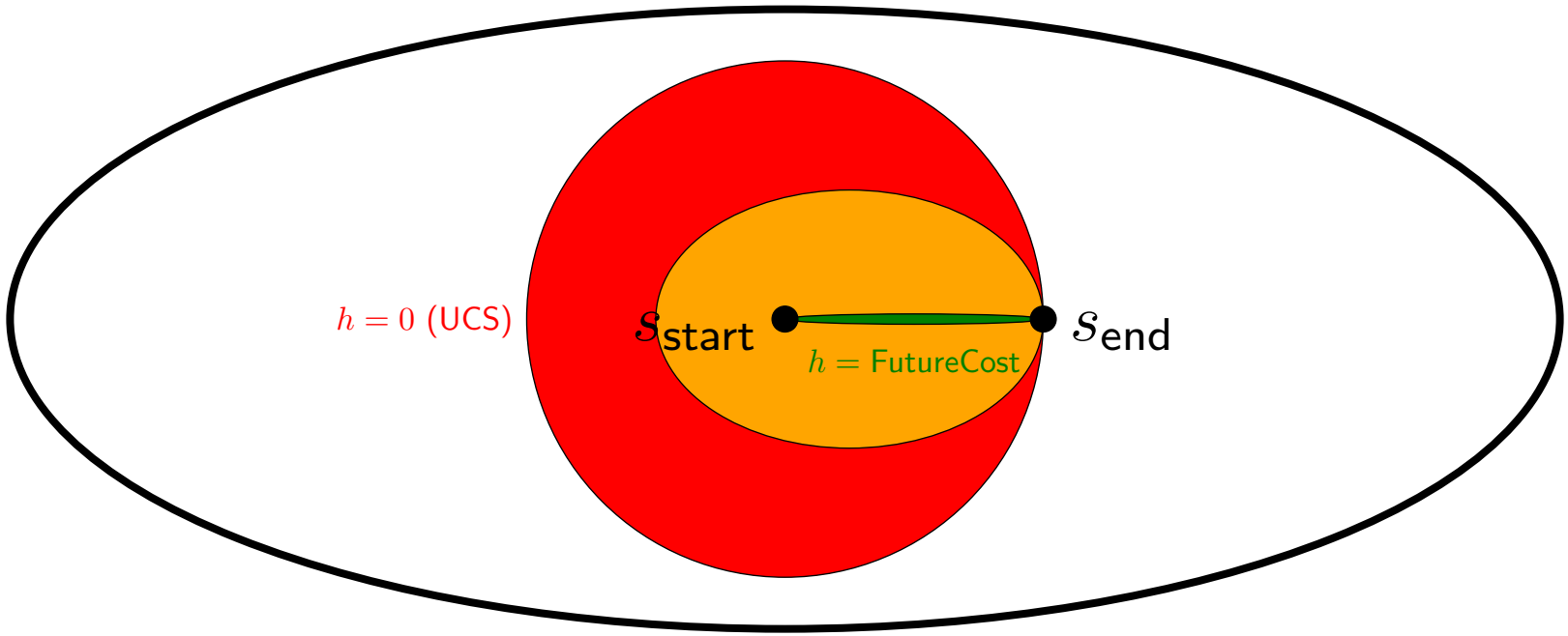
$$\text{PastCost}(s) + h(s)$$

$$\leq$$

$$\text{PastCost}(s_{\text{end}})$$

- We've proven that A* is correct (finds the minimum cost path) for any consistent heuristic h. But for A* to be interesting, we need to show that it's more efficient than UCS (on the original edge costs). We will measure speed in terms of the number of states which are explored prior to exploring an end state.
- Our second theorem is about the efficiency of A*: recall that UCS explores states in order of past cost, so that it will explore every state whose past cost is less than the past cost of the end state.
- A* explores all states for which $\text{PastCost}'(s) = \text{PastCost}(s) + h(s) - h(s_{\text{start}})$ is less than $\text{PastCost}'(s_{\text{end}}) = \text{PastCost}(s_{\text{end}}) + h(s_{\text{end}}) - h(s_{\text{start}})$, or equivalently $\text{PastCost}(s) + h(s) \leq \text{PastCost}(s_{\text{end}})$ since $h(s_{\text{end}}) = 0$.
- From here, it's clear that we want $h(s)$ to be as large as possible so we can push as many states over the $\text{PastCost}(s_{\text{end}})$ threshold, so that we don't have to explore them. Of course, we still need h to be consistent to maintain correctness.
- For example, suppose $\text{PastCost}(s_1) = 1$ and $h(s_1) = 1$ and $\text{PastCost}(s_{\text{end}}) = 2$. Then we would have to explore s_1 ($1 + 1 \leq 2$). But if we were able to come up with a better heuristic where $h(s_1) = 2$, then we wouldn't have to explore s_1 ($1 + 2 > 2$).

Amount explored



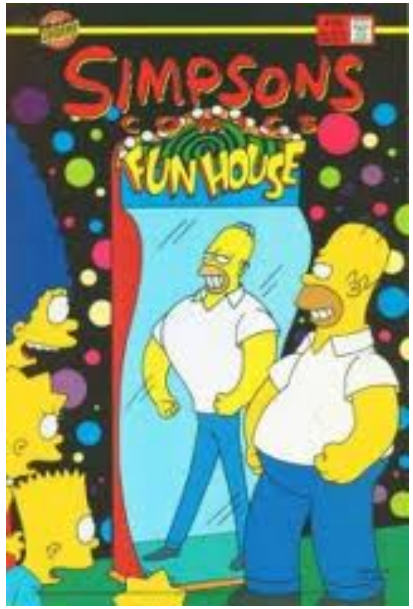
- If $h(s) = 0$, then A* is same as UCS.
- If $h(s) = \text{FutureCost}(s)$, then A* only explores nodes on a minimum cost path.
- Usually $h(s)$ is somewhere in between.

- In this diagram, each ellipse corresponds to the set of states which are explored by A* with various heuristics. In general, any heuristic we come up with will be between the trivial heuristic $h(s) = 0$ which corresponds to UCS and the oracle heuristic $h(s) = \text{FutureCost}(s)$ which is unattainable.

A* search

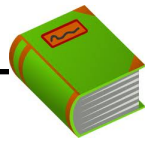
Key idea: distortion

A* distorts edge costs to favor end states.



- What exactly is A* doing to the edge costs? Intuitively, it's biasing us towards the end state.

Admissibility



Definition: admissibility

A heuristic $h(s)$ is admissible if

$$h(s) \leq \text{FutureCost}(s)$$

Intuition: admissible heuristics are optimistic

Theorem: consistency implies admissibility

If a heuristic $h(s)$ is **consistent**, then $h(s)$ is **admissible**.

Proof: use induction on $\text{FutureCost}(s)$

- So far, we've just assumed that FutureCost(s) is the best possible heuristic (ignoring for the moment that it's impractical to compute). Let's actually prove this now.
- To do this, we just have to show that any consistent heuristic $h(s)$ satisfies $h(s) \leq$ FutureCost(s) (since by the previous theorem, the larger the heuristic, the better). In fact, this property has a special name: we say that $h(s)$ is **admissible**. In other words, an admissible heuristic $h(s)$ **underestimates** the future cost: it is optimistic.
- The proof proceeds by induction on increasing FutureCost(s). In the base case, we have $0 = h(s_{\text{end}}) \leq$ FutureCost$(s_{\text{end}}) = 0$ by the second condition of consistency.
- In the inductive case, let s be a state and let a be an optimal action leading to $s' =$ Succ(s, a) that achieves the minimum cost path to the end state; in other words, FutureCost$(s) =$ Cost$(s, a) +$ FutureCost(s'). Since Cost$(s, a) \geq 0$, we have that FutureCost$(s') \leq$ FutureCost(s), so by the inductive hypothesis, $h(s') \leq$ FutureCost(s'). To show the same holds for s, consider: $h(s) \leq$ Cost$(s, a) + h(s') \leq$ Cost$(s, a) +$ FutureCost$(s') =$ FutureCost(s), where the first inequality follows by consistency of $h(s)$, the second inequality follows by the inductive hypothesis, and the third equality follows because a was chosen to be the optimal action. Therefore, we conclude that $h(s) \leq$ FutureCost(s).
- Aside: People often talk about admissible heuristics. Using A* with an admissible heuristic is only guaranteed to find the minimum cost path for tree search algorithms, where we don't use an explored list. However, the UCS and A* algorithms we are considering in this class are graph search algorithms, which require consistent heuristics, not just admissible heuristics, to find the minimum cost path. There are some admissible heuristics which are not consistent, but most natural ones are consistent.

Roadmap

Learning costs

A* search

Relaxation

How do we get good heuristics? Just relax...

Relaxation

Intuition: ideally, use $h(s) = \text{FutureCost}(s)$, but that's as hard as solving the original problem.

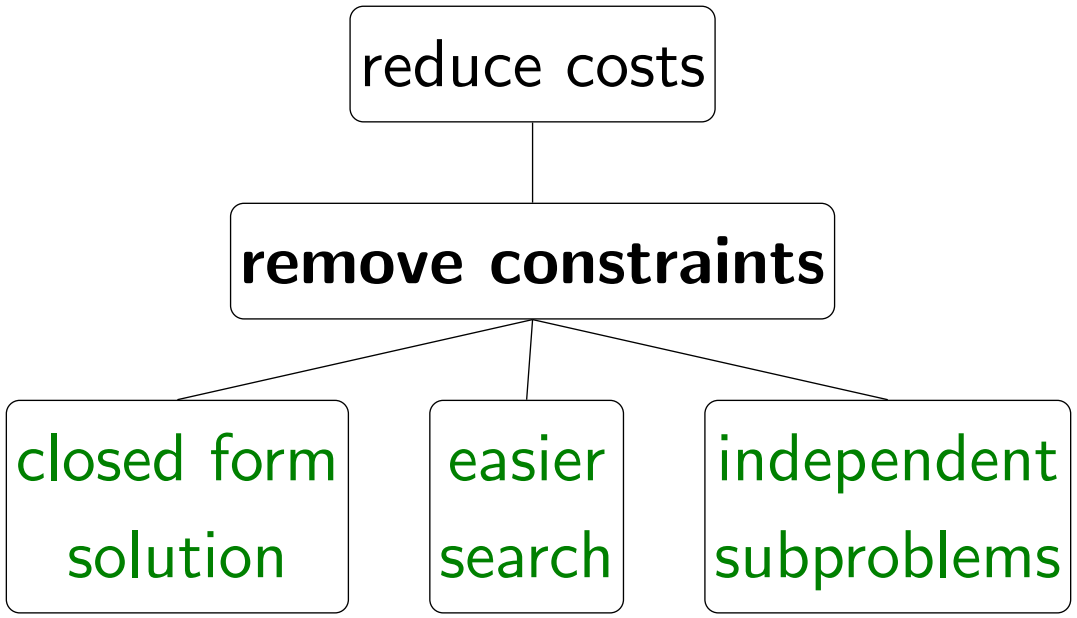
Key idea: relaxation

Constraints make life hard. Get rid of them.

But this is just for the heuristic!

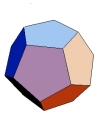
- So far, given a heuristic $h(s)$, we can run A* using it and get a savings which depends on how large $h(s)$ is. However, we've only seen two heuristics: $h(s) = 0$ and $h(s) = \text{FutureCost}(s)$. The first does nothing (gives you back UCS), and the second is hard to compute.
- What we'd like to do is to come up with a general principle for coming up with heuristics. The idea is that of a **relaxation**: instead of computing $\text{FutureCost}(s)$ on the original problem, let us compute $\text{FutureCost}(s)$ on an easier problem, where the notion of easy will be made more formal shortly.
- Note that coming up with good heuristics is about **modeling**, not algorithms. We have to think carefully about our problem domain and see what kind of structure we can exploit in it.

Relaxation overview



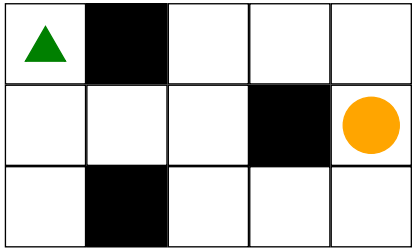
combine heuristics using max

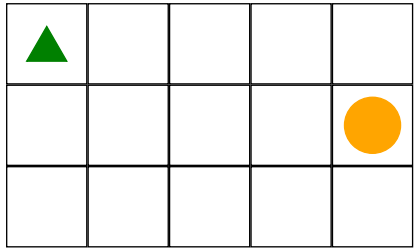
Closed form solution



Example: knock down walls

Goal: move from triangle to circle





Hard

Easy

Heuristic:

$$h(s) = \text{ManhattanDistance}(s, (2, 5))$$

e.g., $h((1, 1)) = 5$

- Here's a simple example. Suppose states are positions (r, c) on the grid. Possible actions are moving up, down, left, or right, provided they don't move you into a wall or off the grid; and all edge costs are 1. The start state is at the triangle at $(1, 1)$, and the end state is the circle at position $(2, 5)$.
- With an arbitrary configuration of walls, we can't compute FutureCost(s) except by doing search. However, if we just **relaxed** the original problem by removing the walls, then we can compute FutureCost(s) in **closed form**: it's just the Manhattan distance between s and s_{end}. Specifically, ManhattanDistance$((r_1, c_1), (r_2, c_2)) = |r_1 - r_2| + |c_1 - c_2|$.

Easier search

Example: original problem

Start state: 1

Walk action: from s to $s+1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

End state: n

Constraint: can't have more tram actions than walk actions.

State: (location, **#walk - #tram**)

Number of states goes from $O(n)$ to $O(n^2)$!

- Let's revisit our magic tram example. Suppose now that a decree comes from above that says you can't have take the tram more times than you walk. This makes our lives considerably worse, since if we wanted to respect this constraint, we have to keep track of additional information (augment the state).
- In particular, we need to keep track of the number of walk actions that we've taken so far minus the number of tram actions we've taken so far, and enforce that this number does not go negative. Now the number of states we have is much larger and thus, search becomes a lot slower.

Easier search

Example: relaxed problem

Start state: 1

Walk action: from s to $s+1$ (cost: 1)

Tram action: from s to $2s$ (cost: 2)

End state: n

~~Constraint: can't have more tram actions than walk actions.~~

Original state: (location, **#walk - #tram**)

Relaxed state: location

- What if we just ignore that constraint and solve the original problem? That would be much easier/faster. But how do we construct a consistent heuristic from the solution from the relaxed problem?

Easier search

- Compute relaxed $\text{FutureCost}_{\text{rel}}(\text{location})$ for **each** location $(1, \ldots, n)$ using dynamic programming or UCS

Example: reversed relaxed problem

Start state: n

Walk action: from s to $s-1$ (cost: 1)

Tram action: from s to $s/2$ (cost: 2)

End state: 1

Modify UCS to compute all past costs in reversed relaxed problem (equivalent to future costs in relaxed problem!)

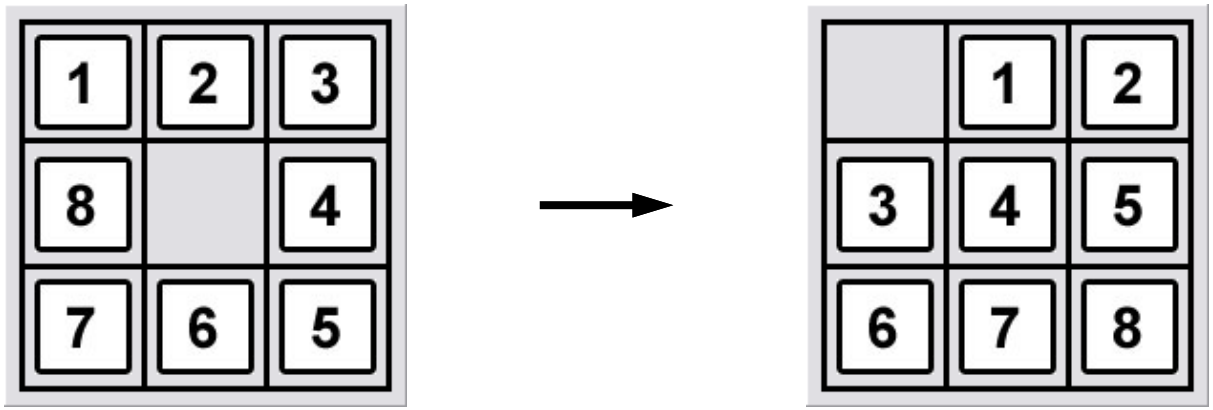
- Define heuristic for original problem:

$$h((\text{location, \#walk-\#tram})) = \text{FutureCost}_{\text{rel}}(\text{location})$$

- We want to now construct a heuristic $h(s)$ based on the future costs under the relaxed problem.
- For this, we need the future costs for all the relaxed states. One straightforward way to do this is by using dynamic programming. However, if we have cycles, then we need to use uniform cost search.
- But recall that UCS only computes the past costs of all states up until the end. So we need to make two changes. First, we simply don't stop at the end, but keep on going until we've explored all the states. Second, we define a **reversed relaxed problem** (where all the edges are just reversed), and call UCS on that. UCS will return past costs in the reversed relaxed problem which correspond exactly to future costs in the relaxed problem.
- Finally, we need to construct the actual heuristic. We have to be a bit careful because the state spaces of the relaxed and original problems are different. For this, we set the heuristic $h(s)$ to the future cost of the relaxed version of s.
- Note that the minimum cost returned by A* (UCS on the modified problem) is the true minimum cost minus the value of the heuristic at the start state.

Independent subproblems

[8 puzzle]



Original problem: tiles cannot overlap (constraint)

Relaxed problem: tiles can overlap (no constraint)

Relaxed solution: 8 indep. problems, each in closed form

Key idea: independence

Relax original problem into independent subproblems.

- So far, we've seen that some heuristics $h(s)$ can be computed in closed form and others can be computed by doing a cheaper search. But there's another way to define heuristics $h(s)$ which are efficient to compute.
- In the 8-puzzle, the goal is to slide the tiles around to produce the desired configuration, but with the constraint that no two tiles can occupy the same position. However, we can throw that constraint out the window to get a relaxed problem. Now, the new problem is really easy, because the tiles can now move **independently**. So we've taken one giant problem and turned it into 8 smaller problems. Each of the smaller problems can now be solved separately (in this case, in closed form, but in other cases, we can also just do search).
- It's worth remembering that all of these relaxed problems are simply used to get the value of the heuristic $h(s)$ to guide the full search. The actual solutions to these relaxed problems are not used.

General framework

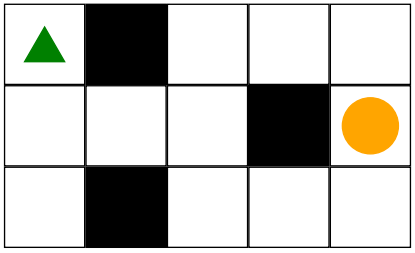
Removing constraints

(knock down walls, walk/tram freely, overlap pieces)

Reducing edge costs

(from ∞ to some finite cost)

Example:

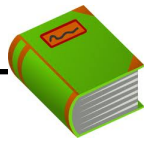


Original: $\text{Cost}((1,1),\text{East}) = \infty$

Relaxed: $\text{Cost}_{\text{rel}}((1,1),\text{East}) = 1$

- We have seen three instances where removing constraints yields simpler solutions, either via closed form, easier search, or independent subproblems. But we haven't formally proved that the heuristics you get are consistent!
- Now we will analyze all three cases in a unified framework. Removing constraints can be thought of as adding edges (you can go between pairs of states that you weren't able to before). Adding edges is equivalent to reducing the edge cost from infinity to something finite (the resulting edge cost).

General framework



Definition: relaxed search problem

A **relaxation** P_{rel} of a search problem P has costs that satisfy:

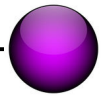
$$\text{Cost}_{\text{rel}}(s, a) \leq \text{Cost}(s, a).$$

Definition: relaxed heuristic

Given a relaxed search problem P_{rel}, define the **relaxed heuristic** $h(s) = \text{FutureCost}_{\text{rel}}(s)$, the minimum cost from s to an end state using $\text{Cost}_{\text{rel}}(s, a)$.

- More formally, we define a relaxed search problem as one where the relaxed edge costs are no larger than the original edge costs.
- The relaxed heuristic is simply the future cost of the relaxed search problem, which by design should be efficiently computable.

General framework



Theorem: consistency of relaxed heuristics

Suppose $h(s) = \text{FutureCost}_{\text{rel}}(s)$ for some relaxed problem P_{rel}.

Then $h(s)$ is a consistent heuristic.

Proof:

$$h(s) \leq \text{Cost}_{\text{rel}}(s, a) + h(\text{Succ}(s, a)) \text{ [triangle inequality]}$$

$$\leq \text{Cost}(s, a) + h(\text{Succ}(s, a)) \text{ [relaxation]}$$

- We now need to check that our defined heuristic $h(s)$ is actually consistent, so that using it actually will yield the minimum cost path of our original problem (not the relaxed problem, which is just a means towards an end).
- Checking consistency is actually quite easy. The first inequality follows because $h(s) = \text{FutureCost}_{\text{rel}}(s)$, and all future costs correspond to the minimum cost paths. So taking action a from state s better be no better than taking the best action from state s (this is all in the search problem P_{rel}).
- The second inequality follows just by the definition of a relaxed search problem.
- The significance of this theorem is that we only need to think about coming up with relaxations rather than worrying directly about checking consistency.

Tradeoff

Efficiency:

$h(s) = \text{FutureCost}_{\text{rel}}(s)$ must be easy to compute

Closed form, easier search, independent subproblems

Tightness:

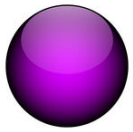
heuristic $h(s)$ should be close to $\text{FutureCost}(s)$

Don't remove too many constraints

- How should one go about designing a heuristic?
- First, the heuristic should be easy to compute. As the main point of A* is to make things more efficient, if the heuristic is as hard as to compute as the original search problem, we've gained nothing (an extreme case is no relaxation at all, in which case $h(s) = \text{FutureCost}(s)$).
- Second, the heuristic should tell us some information about where the goal is. In the extreme case, we relax all the way and we have $h(s) = 0$, which corresponds to running UCS. (Perhaps it is reassuring that we never perform worse than UCS.)
- So the art of designing heuristics is to balance informativeness with computational efficiency.

Max of two heuristics

How do we combine two heuristics?



Proposition: max heuristic

Suppose $h_1(s)$ and $h_2(s)$ are consistent.

Then $h(s) = \max\{h_1(s), h_2(s)\}$ is consistent.

Proof: exercise

- In many situations, you'll be able to come up with two heuristics which are both reasonable, but no one dominates the other. Which one should you choose? Fortunately, you don't have to choose because you can use both of them!
- The key point is the max of two consistent heuristics is consistent. Why max? Remember that we want heuristic values to be larger. And furthermore, we can prove that taking the max leads to a consistent heuristic.
- Stepping back a bit, there is a deeper takeaway with A* and relaxation here. The idea is that when you are confronted with a difficult problem, it is wise to start by solving easier versions of the problem (being an optimist). The result of solving these easier problems can then be a useful guide in solving the original problem.
- For example, if the relaxed problem turns out to have no solution, then you don't even have to bother solving the original problem, because a solution can't possibly exist (you've been optimistic by using the relaxation).

Summary

- Structured Perceptron (reverse engineering): learn cost functions (search + learning)
- A*: add in heuristic estimate of future costs
- Relaxation (breaking the rules): framework for producing consistent heuristics
- Next time: when actions have unknown consequences...