Search: A*, heuristics, and learning

Announcements

- Homework 2 (sentiment) due tonight at **11pm**

- Homework 3 (search) out today, due in 1 week

- High-res feedback

- announcements before we get started
- homework 2 is due tonight!
- homework 3 is out today, and this is the last lecture of search, so you should know everything you need. come to office hours if youre still confused
- help us out by filling out the weekly feedback! we are reading it weekly.

Roadmap

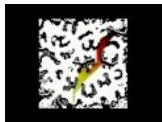**Topics in the lecture:**

A^* (c3.5.2)

Search heuristics (c3.6)

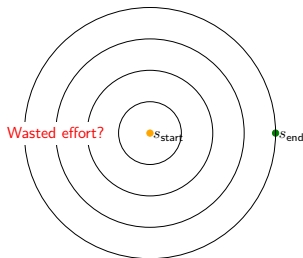Learning: Structured Perceptron

A* algorithm

UCS in action:

A* in action:

Can uniform cost search be improved?

Wasted effort? $\bullet s_\text{start}$ $\bullet s_\text{end}$

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)

s_start s s_end

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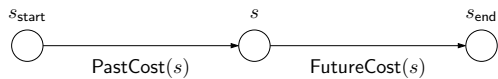
Ideal: explore in order of PastCost(s) + FutureCost(s)

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.
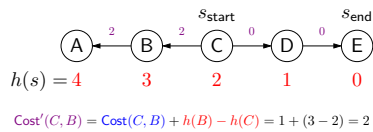
A* search

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:

s_{start} s_{end}

$$A \xleftarrow{2} B \xleftarrow{2} C \xrightarrow{0} D \xrightarrow{0} E$$

$h(s) = 4 \quad\quad 3 \quad\quad 2 \quad\quad 1 \quad\quad 0$

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

- 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 $\text{PastCost}(s) + h(s)$ rather than $\text{PastCost}(s)$.) 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 $\text{Cost}(s, a)$ plus an additional term. This term is difference between the estimated future cost of the new state $\text{Succ}(s, a)$ 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 $\text{FutureCost}(s)$, 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:

0

B

0   0

A   D

C

1000
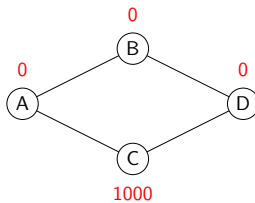
Doesn't work because of **negative modified edge costs**!

- So far, we've just said that $h(s)$ is just an approximation of $\text{FutureCost}(s)$. 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.
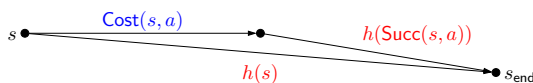
Consistent heuristics

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).

$s \bullet \xrightarrow{\text{Cost}(s,a)} \bullet \xrightarrow{h(\text{Succ}(s,a))} \bullet s_{\text{end}}$

$h(s)$

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.
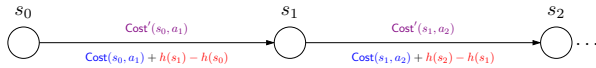
Correctness of A*

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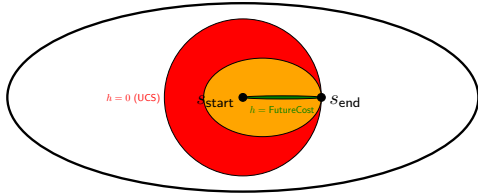
- If $h(s) = 0$, then A* is same as UCS.
- If $h(s) = \text{FutureCost}(s)$, then A* only explores nodes on a minimum cost path.
- Usually $h(s)$ is somewhere in between.

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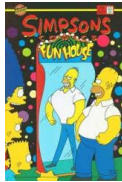
- What exactly is A* doing to the edge costs? Intuitively, it's biasing us towards the end state.

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

Admissibility proof

Theorem: consistency implies admissibility

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

Base case: $0 = h(s_{\text{end}}) \leq \text{FutureCost}(s_{\text{end}}) = 0$

Inductive case: we have state s, optimal action a, successor s'

By optimality: $\text{FutureCost}(s) = \text{Cost}(s, a) + \text{FutureCost}(s') \geq \text{FutureCost}(s')$

By inductive hypothesis, $h(s') \leq \text{FutureCost}(s')$

Thus $h(s) \leq \text{Cost}(s, a) + h(s')$

$\leq \text{Cost}(s, a) + \text{FutureCost}(s') = \text{FutureCost}(s)$

- The proof proceeds by induction on increasing $\text{FutureCost}(s)$. In the base case, we have $0 = h(s_{\text{end}}) \leq \text{FutureCost}(s_{\text{end}}) = 0$ by the second condition of consistency.
- In the inductive case, let s be a state and let a be an optimal action leading to $s' = \text{Succ}(s, a)$ that achieves the minimum cost path to the end state; in other words, $\text{FutureCost}(s) = \text{Cost}(s, a) + \text{FutureCost}(s')$. Since $\text{Cost}(s, a) \geq 0$, we have that $\text{FutureCost}(s') \leq \text{FutureCost}(s)$, so by the inductive hypothesis, $h(s') \leq \text{FutureCost}(s')$. To show the same holds for s, consider: $h(s) \leq \text{Cost}(s, a) + h(s') \leq \text{Cost}(s, a) + \text{FutureCost}(s') = \text{FutureCost}(s)$, 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 $h(s) \leq \text{FutureCost}(s)$.

Summary and Review

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

Run uniform cost search with **modified edge costs**:
$$\text{Cost}'(s, a) = \text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s)$$

Definition: consistency

A heuristic h is **consistent** if
- $\text{Cost}'(s, a) = \text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s) \geq 0$
- $h(s_{\text{end}}) = 0$.

Roadmap

Topics in the lecture:

A^* (c3.5.2)

Search heuristics (c3.6)

Learning: Structured Perceptron

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!

Relaxation overview

reduce costs

remove constraints

closed form solution

easier search

independent subproblems

combine heuristics using max

- 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 FutureCost(s) on the original problem, let us compute 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.
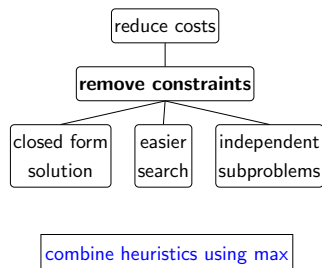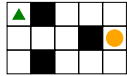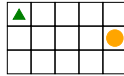
Closed form solution

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:

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

- 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]
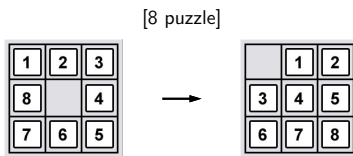
Original problem: tiles cannot overlap (constraint)

Relaxed problem: tiles can overlap (no constraint)

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

> 💡 **Key idea: independence**
>
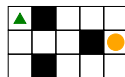> Relax original problem into independent subproblems.

- 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

Summary and Review

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

Remember: relaxed heuristics are consistent!

Roadmap

Topics in the lecture:

A^* (c3.5.2)

Search heuristics (c3.6)

Learning: Structured Perceptron

- 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).

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

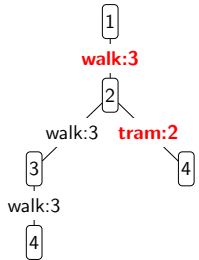Input x: search problem without costs
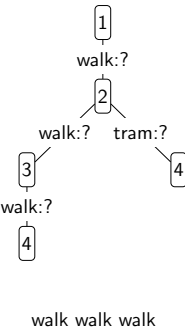
Output y: solution path

walk walk walk

Tweaking costs

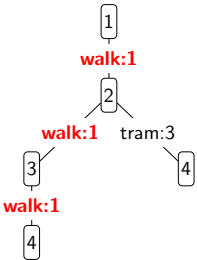Costs: {walk:3, tram:2}   Costs: {walk:1, tram:3}

Minimum cost path: 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:
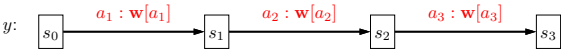
Path cost:

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

• 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)
- **Key idea** : penalize any incorrect predictions, reward the right actions

- 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.

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.

Summary and Review

💡 Key idea: learning action costs

Given observations of optimal actions, find plausible costs

The **structured Perceptron** can find such costs

Growing importance in utility and preference learning