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

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 ← 6 5 3 |

Review
Definition: search problem
- $s_{\text{start}}$: starting state
- $\text{Actions}(s)$: possible actions
- $\text{Cost}(s, a)$: action cost
- $\text{Succ}(s, a)$: successor
- $\text{IsEnd}(s)$: reached end state?

Objective: find the minimum cost path from $s_{\text{start}}$ to an $s$ satisfying $\text{IsEnd}(s)$.
• Recall the magic tram example from the last lecture. Given a search problem (specification of the start state, end, 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.

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):

Cost(s, a) \rightarrow (a_1, \ldots, a_k)

Inverse problem (learning):

(a_1, \ldots, a_k) \rightarrow 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.

• 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.

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.

• Intuitively, we want the tram cost to be more and the walk cost to be less. Specifically, let’s increase the walk cost and decrease the tram cost.

Now, the predicted path coincides with the true observed path. Is this a good strategy in general?

**Tweaking costs**

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

Minimum cost path: Minimum cost path:

1 1
walk:3 walk:1

2 walk:3 tram:2

3 walk:3 4 tram:2

walk:3

4

Output $y$: solution path

walk walk walk

**Modeling costs (simplified)**

Assume costs depend only on the action:

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

Candidate output path:

\[ y: s_0 : w[a_1] \xrightarrow{} s_1 : w[a_2] \xrightarrow{} s_2 : w[a_3] \xrightarrow{} s_3 \]

Path cost:

$$\text{Cost}(y) = w[a_1] + w[a_2] + w[a_3]$$
For each action $a$, we define a weight $w[a]$ 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.

So far, the cost of an action $a$ is simply $w[a]$. We can generalize this to allow the cost to be a general dot product $w \cdot \phi(y)$, where $\phi(y) = \phi(s_0, a_1) + \phi(s_1, a_2)$ is the sum of the feature vectors over all actions.

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 $\text{Loss}(x, y, w) = \max(0, -\langle w, \phi(x, y) \rangle)$, 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 $\text{Loss}(x, y, w) = \max_x \{ \sum_a w[a] - \sum_y \phi(s, a) \}$ (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.

So far, the cost of an action $a$ is simply $w[a]$. We can generalize this to allow the cost to be a general dot product $w \cdot \phi(s, a)$, 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 $w \cdot \phi(y)$, 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

**Algorithm: Structured Perceptron (simplified)**

- For each action: $w[a] \leftarrow 0$
- For each iteration $t = 1, \ldots T$:
  - For each training example $(x, y) \in D_{\text{train}}$:
    - Compute the minimum cost path $y'$ given $w$
    - For each action $a \in y$: $w[a] \leftarrow w[a] - 1$
    - For each action $a \in y'$: $w[a] \leftarrow w[a] + 1$
  - Try to decrease cost of true $y$ (from training data)
  - Try to increase cost of predicted $y'$ (from search)

Generalization to features (skip)

Costs are parametrized by feature vector:

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

Example:

- $\phi(y) = \phi(s_0, a_1) + \phi(s_1, a_2)$
- $y = [a_0, a_1, a_2]$ with $a_1 : w \cdot [1, 0, 1]$, $a_2 : w \cdot [1, 2, 0]$
- $w = [3, -1, -1]$

Path cost:

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

Learning algorithm (skip)

**Algorithm: Structured Perceptron [Collins, 2002]**

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

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 $s_{\text{start}}$ to $s$

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

- The structured Perceptron was first used for natural language processing tasks. Given its 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.

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)

\[ S_{\text{start}} \rightarrow S \rightarrow S_{\text{end}} \]

PastCost(s) FutureCost(s)

Ideal: explore \[ \text{in order of PastCost(s) + FutureCost(s)} \]

Definition: Heuristic function

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

A* search


Run uniform cost search with modified edge costs:

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

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

Example:

\[ \begin{array}{c}
0 & 2 & 2 & 0 & 0 \\
A & B & C & D & E \\
\end{array} \]

\( h(s) = 4 \)

\[ \begin{array}{c}
3 & 2 & 1 & 0 \\
\end{array} \]

An example heuristic

Will any heuristic work?

No.

Counterexample:

\[ \begin{array}{c}
0 & 0 & 0 \\
A & 1000 & C \\
\end{array} \]

Doesn’t work because of negative modified edge costs!
Consistent heuristics

Definition: consistency
A heuristic \( h \) is consistent if:
- \( \text{Cost}'(s, a) = \text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s) \geq 0 \)
- \( h(\text{send}) = 0 \).

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

\[
\text{Cost}(s, a) \rightarrow \text{Succ}(s, a) \rightarrow h(\text{Succ}(s, a)) \rightarrow \text{send}
\]

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

Correctness of A*

Proposition: correctness
If \( h \) is consistent, A* returns the minimum cost path.

Proof of A* correctness

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

\[
\begin{align*}
\text{Cost}(s_0, a_1) + h(s_1) & = \text{Cost}'(s_0, a_1) + h(s_1) \\
\text{Cost}(s_1, a_2) + h(s_2) & = \text{Cost}'(s_1, a_2) + h(s_2)
\end{align*}
\]

- Key identity:

\[
\sum_{i=1}^{L} \text{Cost}'(s_{i-1}, a_i) = \sum_{i=1}^{L} \text{Cost}(s_{i-1}, a_i) + h(s_L) - h(s_0)
\]

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

- 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, \( h(\text{send}) = 0 \), 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.

- 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.

- To show the correctness of A*, let’s take any path of length \( L \) from \( s_0 = \text{start} \) to \( s_L = \text{send} \). 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 \( (c_1 + h_1) + (c_2 + h_2) + \ldots + (c_L + h_L) \). 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

\[
\text{PastCost}(s) + h(s) \leq \text{PastCost}(s_{\text{end}})
\]

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.

A* search

Key idea: distortion

A* distorts edge costs to favor end states.

- 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 \( \text{PastCost}(s) + h(s) - h(s_{\text{end}}) \) is less than \( \text{PastCost}(s_{\text{end}}) = \text{PastCost}(s_{\text{end}}) + h(s_{\text{end}}) - h(s_{\text{end}}) \), 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 ≤ 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).

- 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.

- 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 \( \text{FutureCost}(s) \)

---

### Roadmap

- **Learning costs**
- **A**\(^*\) search
- **Relaxation**

### 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 \( \text{A}^* \) 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

- Relaxation overview
- reduce costs
- remove constraints
- closed form solution
- easier search
- independent subproblems
- combine heuristics using max

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.

- 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.

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., $h((1, 1)) = 5$
• 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?

Relevant sections from the image:

- **Easier search**
  - Compute relaxed $\text{FutureCost}_r(location)$ for each location $(1, \ldots, n)$ using dynamic programming or UCS.
  
  - Example: reversed relaxed problem.

- **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.

- **General framework**
  - Removing constraints (knock down walls, walk/tram freely, overlap pieces)
  - Reducing edge costs (from $\infty$ to some finite cost)

- **Example**:
  - Original: $\text{Cost}((1, 1), \text{East}) = \infty$
  - Relaxed: $\text{Cost}_r((1, 1), \text{East}) = 1$

---

**Notes**:

- 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.
• 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:

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

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

General framework

Theorem: consistency of relaxed heuristics
Suppose $h(s) = \text{FutureCost}_{rel}(s)$ for some relaxed problem $P_{rel}$.
Then $h(s)$ is a consistent heuristic.

Proof:

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

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

Tradeoff

Efficiency:

$h(s) = \text{FutureCost}_{rel}(s)$ 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

- **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...