Search: structured perceptron

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):

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

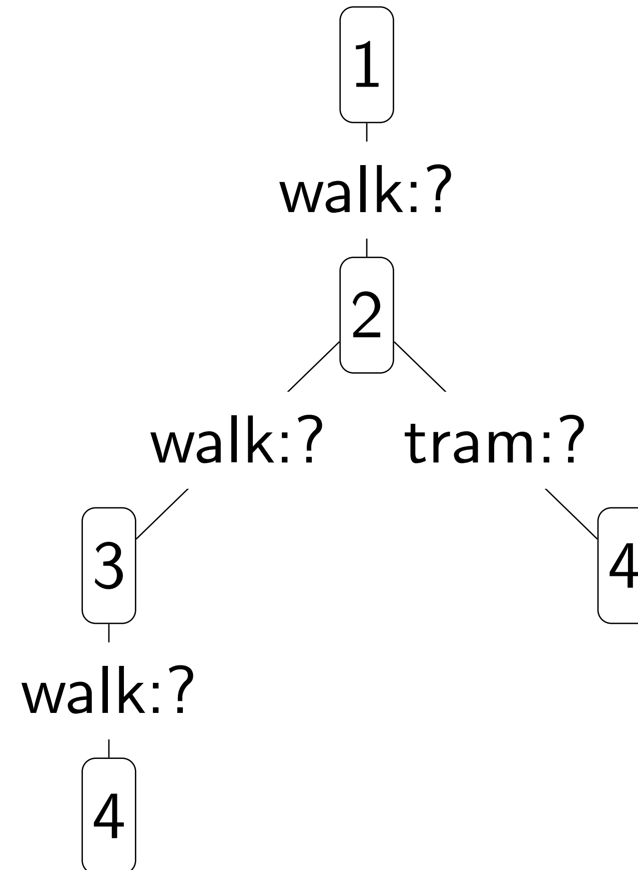
Inverse problem (learning):

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

- 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

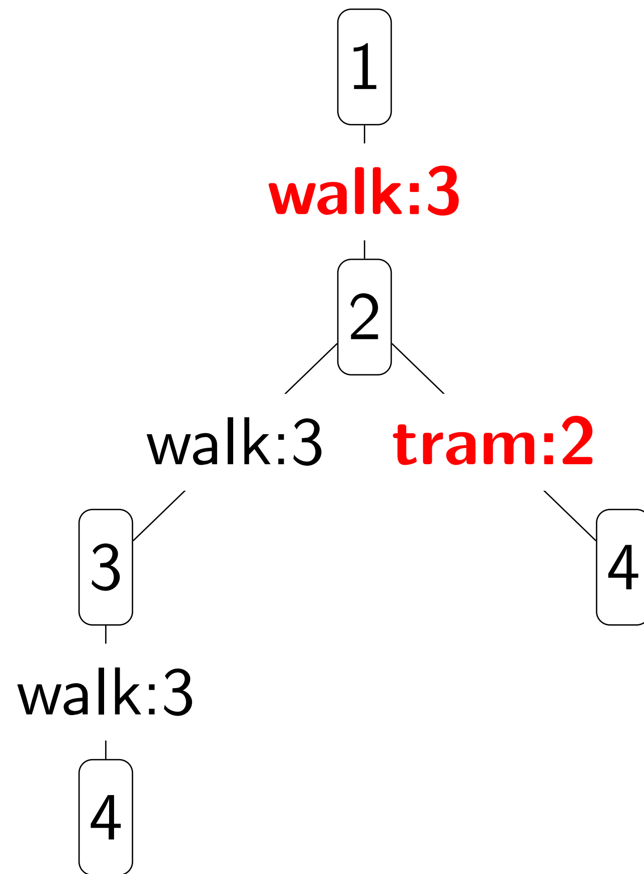
walk walk walk

- 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

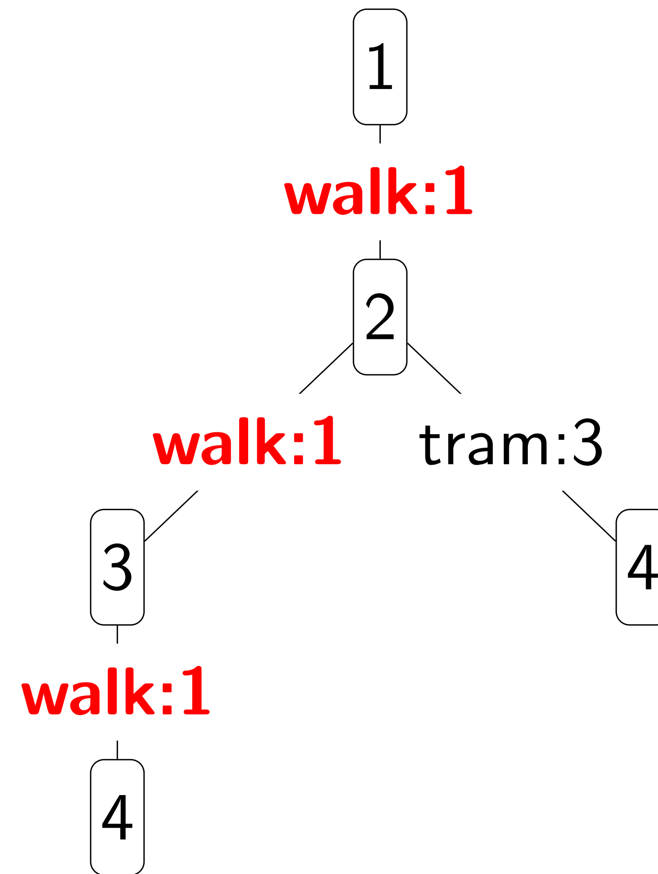
Costs: {walk:3, tram:2}

Minimum cost path:



Costs: {walk:1, tram:3}

Minimum cost path:



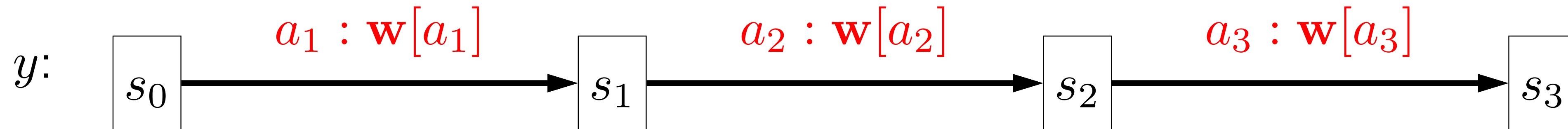
- Suppose the walk cost is 3 and the tram cost is 2. Then, we would obviously predict the [walk, tram] path, which has lower cost.
- But this is not our desired output, because we actually saw the person walk all the way from 1 to 4. How can we update the action costs so that the minimum cost path is 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 $t = 1, \dots T$:
  - 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]

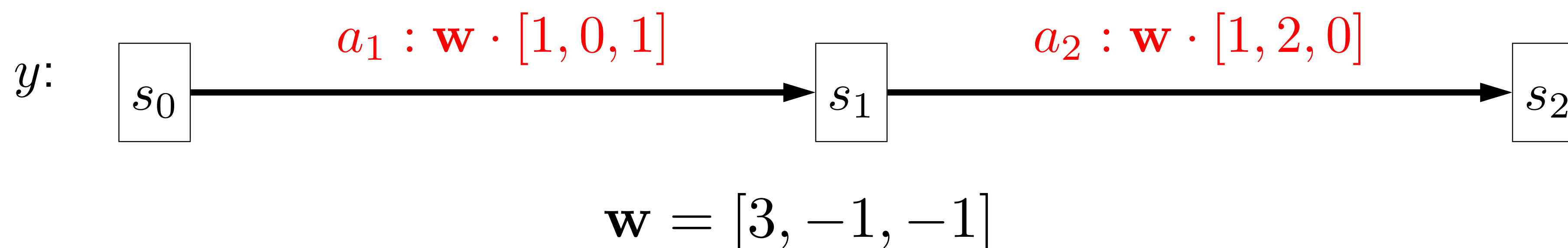
- We are now in position to state the (simplified version of) **structured Perceptron** algorithm.
- Advanced: the Perceptron algorithm performs stochastic gradient descent (SGD) on a modified hinge loss with a constant step size of $\eta = 1$. The modified hinge loss is $\text{Loss}(x, y, \mathbf{w}) = \max\{-(\mathbf{w} \cdot \phi(x))y, 0\}$, 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)

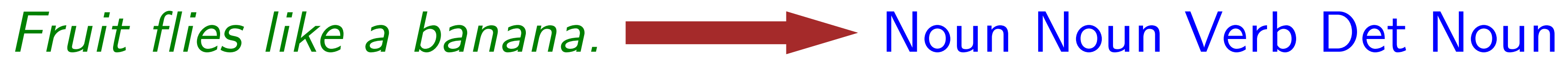
Algorithm: Structured Perceptron [Collins, 2002]

- 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



- Machine translation

- 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.