Bellman-Ford Algorithm

1 Shortest Paths with Dynamic Programming

Now that we know how to use dynamic programming, we can revisit the idea of finding a shortest path when negative edges exist in our graph. For a shortest $s \to t$ path to exist in such a graph, there must not be a negative cycle reachable from s and t. To decompose our large problem into solvable subproblems, we observe that the optimum $s \to t$ path goes from $s \to$ neighbor of t optimally, then takes a last hop to t, minimizing the sum of the two costs.

As a first try, we set $OPT(v)$ = minimum cost path from $v \to t$. Equivalently:

$$OPT(v) = \min_{u:v\to u}(OPT(u) + C_{v,u}) \text{ with } OPT(t) = 0$$

But that makes the task computationally impossible, because in order to calculate $OPT(v)$ we need to know $OPT(u)$ and vice versa. So, we introduce a second variable and reparametrize. Let k be the number of hops in a path; then $OPT(v,k)$ = minimum cost path from $v \to t$ with total hops $\leq k$. For each k-step, one can choose to stay at the current node, or move to an unvisited one. Equivalently:

$$OPT(v,k) = \min(OPT(v,k-1), \min_{u:v\to u}(OPT(u,k-1) + C_{v,u}))$$

$$OPT(t,0) = 0 \text{ and } OPT(v,0) = \infty \text{ for all } v \neq t$$

2 Naive Attempts at Bellman-Ford

2.1 Naive Attempt 1

Algorithm 1 Naive Bellman-Ford 1

```
1: $a[v,0] = \infty$ for all $v \neq t$
2: $a[t,0] = 0$
3: for $k = 1$ to $n$ do
4:     for all $v$ do
5:         $a[v,k] = \min(a[v,k-1], \min_{u:v\to u}(a[u,k-1] + C_{v,u}))$
6:     end for
7: end for
8: return $a[s,n]$
```

We know that this algorithm must run for n iterations, because the shortest path will never contain a cycle (assuming no negative cycles, which we will account for later). Thus, $a[s,n]$ contains the length of the shortest $s \to t$ path. This algorithm runs in $\Theta(mn)$ time and $\Theta(n^2)$ space.

Algorithm 2 Naive Bellman-Ford 2

```
1: $a[v] = \infty, \text{next}(v) = \perp$ for all $v \neq t$
2: $a[t] = 0,\ \text{next}(t) = \perp$
3: for all $n$ do
4:     for all $v$ do
5:         $a[v] = \min(a[v], \min_{u:v\to u}(a[u] + C_{v,u}))$
6:         $\text{next}(v) =$corresponding pointer
7:     end for
8: end for
9: return $a[s]$, $\text{next}(s)$
```

2.2 Naive Attempt 2

In order to reconstruct the shortest path, we adjust the algorithm to contain a next pointer for each node. The next pointer also increase memory efficiency by storing only two path value vectors – the nodes we updated last round, and the current nodes (instead of all nodes, because we can access their necessary information through the pointers).

Claim 1. At termination of Algorithm 2, $a[v] = OPT(v)$.

Proof. $a[v] \leq OPT(v, k) \leq OPT(v)$ after k iterations, because each update can only make $a[v]$ smaller. Because the next pointers always describe a $v \to t$ path of cost $a[v]$, we have $a[v] \geq OPT(v)$. Combining these two inequalities shows that $a[v] = OPT(v)$. □

This more efficient version of Bellman-Ford runs in $\mathcal{O}(mn)$ time and $\Theta(n \log n)$ space, but still cannot handle negative cycles.

3 Negative Cycles

3.1 Finding Negative Cycles

Given a weighted digraph $\mathcal{G}$, find a negative cycle if it exists.

We add a node t with edges of cost 0 from all v and run Bellman-Ford. If the shortest path cost is updated in iteration n (where it would be adding the 0 cost edges to t and thus not updating), there must be a cycle. To locate the cycle, we follow the next pointers of the node that updated.

Claim 2. If the next pointers $\underline{\text{ever}}$ contain a cycle, it is a negative cycle.

Proof. When the node that updated, u, last updated its next pointer, $a[u] = a[\text{next}(u)] + C_{u,\text{next}(u)}$. But $a[\text{next}(u)]$ might later have decreased, so we have $a[u] \geq a[\text{next}(u)] + C_{u,\text{next}(u)}$. Right before updating a neighboring node v, we had $a[v] > a[\text{next}(v)] + C_{v,\text{next}(v)}$. We add these inequalities around the cycle Δ: $\sum_{u\in\Delta} a[u] > \sum_{u\in\Delta} a[u] + \sum_{u\in\Delta} C_{u,\text{next}(u)}$. Then, by canceling, we have $0 > \sum_{u\in cycle} C_{u,\text{next}(u)}$, which means the cycle must be negative. □

3.2 Efficient Termination After Negative Cycle Detection

Our goal now is to improve Bellman-Ford so that it terminates as soon as a negative cycle is detected.

The naive approach is to look for cycles after each update, but the cost is $\Theta(n)$ for the BFS on the next pointer graph. This brings the total runtime to $\Theta(n^2 m)$, which is too high.

Until a cycle appears in the Bellman-Ford graph, we know that it is an **indirected arborescence** into t (a tree where all edges point to t). Thus, setting $\text{next}(v) = u$ would create a cycle if and only if u is in the subtree of the indirected arborescence rooted at v. In addition to storing the outgoing next pointer, we'll also store previous pointers to all u such that $\text{next}(u) = v$. Then we can test if u is in the subtree rooted at v by performing BFS throughout the previous pointers. However, this is still $\Theta(n^2 m)$.

Observation: When no cycle is found after the previous pointer BFS, all $a[u]$ values in the subtree rooted at v are outdated. So, there is no point in using them until u is updated again in the next iteration of Bellman-Ford. We can delete all outgoing $\text{next}(u)$ pointers for all $u \in$ subtree and mark them as inactive until u is updated. This way, each next pointer is used only once in BFS, and the cost can be amortized against the creation of the subsequent next pointer.

By using amortized analysis, we determine that the runtime of this final version of Bellman-Ford is $\Theta(mn)$ and faster in practice.

4 Bellman-Ford Algorithm

4.1 Bellman-Ford Algorithm

Algorithm 3 Bellman-Ford Algorithm

```
1: $a[v] = \infty, \text{next}(v) = \perp$ for all $v \neq t$
2: $a[t] = 0$, $\text{next}(t) = \perp$
3: **for** 1 to $n-1$ **do**
4:     **for** all $v$ **do**
5:         $a[v] = \min(a[v], \min_{u:v\to u}(a[u] + C_{v,u}))$
6:         $\text{next}(v) =$corresponding pointer
7:     **end for**
8: **end for**
9: **for** all $v$ **do**
10:     **if** $\min(a[v], \min_{u:v\to u}(a[u] + C_{v,u})) \neq a[v]$ **then**
11:         return error: negative cycle detected
12:     **end if**
13: **end for**
14: return $a[s]$, $\text{next}(s)$
```

4.2 Applications

In computer networks, each node monitors its outgoing edges and updates their costs if necessary. Thus, nodes can run Bellman-Ford to find the shortest path between points in the network. These are called Distance Vector Protocols. However, the system can be gamed by nodes which announce very short, possibly fake paths, causing everyone to reroute through them. This approach also doesn't deal well with nodes going down or distances increasing; more mature Link State Protocols are normally used instead.

Many dynamic programming problems can be solved with Bellman-Ford on a suitably defined graph (e.g., weighted interval, knapsack, etc).