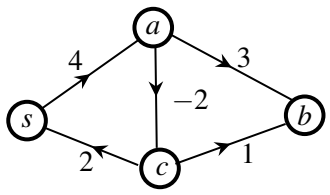
Bellman-Ford Algorithm

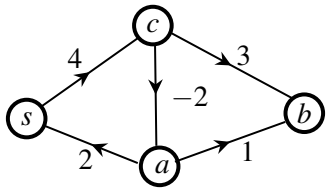
See two examples of running Bellman-Ford algorithm below.



graph

	s	a	b	c
init	0	∞	∞	∞
$R1$	0	4	∞	∞
$R2$	0	4	3	2
$R3$	0	4	3	2

dist array



graph

	s	a	b	c
init	0	∞	∞	∞
$R1$	0	∞	∞	4
$R2$	0	2	7	4
$R3$	0	2	3	4

dist array

Figure 1: The dist array (after each round) running Bellman-Ford algorithm on each example. In each round, we choose to update all edges in lexicographic order, i.e., $(a,b),(a,c),(c,b),(c,s),(s,a)$.

To prove the correctness of Bellman-Ford algorithm, we first introduce some definitions and prove some properties.

A negative cycle C in a graph is a cycle with negative length, i.e., $l(C) := \sum_{e \in C} l(e) < 0$. In the presence of negative cycle, if we do not limit the number of edges in a path, then the length of a path could goes to negative infinity. In other words, the shortest path may not exist in graphs with negative cycles. We therefore need to be careful when talking about shortest paths or $distance(\cdot,\cdot)$, as they have to exist first. It suffices to require the graph to be negative cycle free. Later in this Lecture we will also introduce an algorithm to detect if a given graph contains negative cycle or not.

A path p in a graph is *simple* if p does not have repeating vertices. If a graph G does not contain negative cycle, then for any pair of vertices u and v, if u can reach v, then there always exists a simple shortest path from u to v, as otherwise we can skip the cycle in it to get a better or same-length path. If all cycles in graph G are positive then every shortest path is simple. Since any simple path contains at most $|V|-1$ edges, we have the following:

Fact 1. If G does not contain negative cycles, for every $u, v \in V$ where v is reachable from u, there exists a shortest path from u to v with at most $(|V|-1)$ edges.

Shortest path admits the following *optimal substructure* property. Intuitively, this property states that, the shortest path from u to v contains the shortest path from u to any internal vertex on this path (formally described below). Essentially, this is why shortest path problem can be solved efficiently.

Fact 2. Let $p = v_1 \to v_2 \to v_3 \to \cdots \to v_{k-1} \to v_k$ be the shortest path from v_1 to v_k. Then for any $1 \le i \le k$, $p_i = v_1 \to v_2 \to \cdots \to v_i$, i.e., the portion of p from v_1 to v_i, is the shortest path from v_1 to v_i.

Proof. Suppose that $p_i = v_1 \to v_2 \to \cdots \to v_i$ is not the shortest path from v_1 to v_i. Assume that q is the shorest path from v_1 to v_i. Then we can construct a path from v_1 to v_k shorter than p, by concatenating q and $v_i \to v_{i+1} \to \cdots \to v_k$. This contradicts to the fact that p is the shortest path from v_1 to v_k. □

The above fact immediately implies the following:

Fact 3. If there exists one shortest path from s to v such that (u,v) is the last edge on this shortest path, then we have that $distanct(s,v) = distance(s,u) + l(u,v)$.

Now let's go back to the Bellman-Ford algorithm. We first show an invariant about its data structure the *dist* array:

Fact 4. Throughout the Bellman-Ford algorithm, if $dist[v] \neq \infty$ then $dist[v]$ represents the length of some path from s to v.

Proof. Clearly, in the initialization step which sets $dist[s] = 0$ and $dist[v] = \infty$ for all $v \neq s$, above claim holds, as $dist[s]$ stores a path from s to s without any edge and therefore its length is 0. Now to show above fact is correct throughout the algorithm, we just need to show that the "update" operation keeps this invariant (as this algorithm does nothing else but "update"). We can prove this by induction. Assume that up to the n-th update opertion above claim holds, i.e., $dist[v]$ stores the length of some path from s to v when $dist[v] \neq \infty$. Now consider the $(n+1)$-th update operation on edge $e = (u,v)$. Assume that $dist[v] > dist[u] + l(u,v)$, as otherwise this operation does not change *dist* and the claim continues to be true. Now $dist[v]$ is updated as $dist[u] + l(u,v)$. Since, according to the inductive assumption, $dist[u]$ stores the length of some path from s to u, we have that $dist[v]$ stores the length of the path that consists of the aforementioned path from s to u followed by edge (u,v). □

Assume that the shortest path from s to v exists. Following above fact, $dist[v] \geq distance(s,v)$ throughout the algorithm, as $dist[v]$ represents the length of *some* path from s to v, while $distance(s,v)$ represents the length of the *shortest* path from s to v. In Bellman-Ford algorithm, $dist[v]$ starts from a trivial upper bound (i.e., infinity) of $distance(s,v)$, and will get closer and closer to $distance(s,v)$ through updates, and eventually reach $distance(s,v)$. We now state the conditions for this to happen.

Fact 5. If edge (u,v) is the last edge on one shortest path from s to v and $dist[u] = distance(s,u)$, then after $update(u,v)$ we will have $dist[v] = distance(s,v)$.

Proof. Since edge (u,v) is the last edge on one shortest path from s to v, according to Fact 3, we know that $distance(s,v) = distance(s,u) + l(u,v) = dist[u] + l(u,v)$. Through $update(u,v)$, $dist[v]$ will be compared with $dist[u] + l(u,v) = distance(s,v)$. The first case will be that $dist[v] \leq dist[u] + l(u,v) = distance(s,v)$. Notice that in this case we must have $dist[v] = distance(s,v)$ according to Fact 4, i.e., $dist[v]$ already stores the distance. The second case will be that $dist[v] > dist[u] + l(u,v) = distance(s,v)$, and in this case the "update" function will set $dist[v] = dist[u] + l(u,v) = distance(s,v)$. Hence, in either case, we will have $dist[v] = distance(s,v)$ after updating edge (u,v). □

Suppose that $s \to v_1 \to v_2 \to \cdots \to v$ is one shortest path from s to v. In the initialization step we have $dist[s] = distance(s,s) = 0$. If at a later time, $update(s,v_1)$ is executed, then following above Fact 5, we know that $dist[v_1] = distance(s,v_1)$ after this update (reasoning: $dist[s] = distance(s,s)$, and (s,v_1) is the last edge on one shortest path from s to v_1 according to the optimal substrcture property). Once $dist[v_1]$ becomes $distance(s,v_1)$, $dist[v_1]$ will stay as $distance(s,v_1)$ according to Fact 4. If at a later time $update(v_1,v_2)$ happens then following Fact 5, we have that $dist[v_2] = distance(s,v_2)$. Note that it does not matter if additional updates happen between $update(s,v_1)$ and $update(v_1,v_2)$. We can continue this argument; a general form is summarized below.

Fact 6. If there exists a sequence of udpate procedures (not necessarily consecutive, i.e., there can be update(s) between any two in this sequence) that update all the edges following one shortest path from s to v, then after that we will have $dist[v] = distance(s,v)$.

But we do not know the the shortest path in advance. That's fine. As in graphs without negative cycles the number of edges in the (simple) shortest path will not exceed $(|V|-1)$, the Bellman-Ford algorithm simply update *all* edges in each round, and do this $(|V|-1)$ times. This therefore guarantees that the i-th edge on the shortest path can be updated during the i-th round. Consequently, this guarantees the existence of

a sequence of update procedures that update all edges following the shortest path. This analysis leads to the following conclusion, which proves the correctness of Bellman-Ford algorithm. See an illustration in Figure 2.

Fact 7. If G does not contain negative cycle, then we have $dist[v] = distance(s,v)$ for every $v \in V$ after $|V|-1$ rounds of updates. In particular, let p be one shortest path from s to v with k edges. Then after k rounds of updates, $dist[v] = distance(s,v)$.

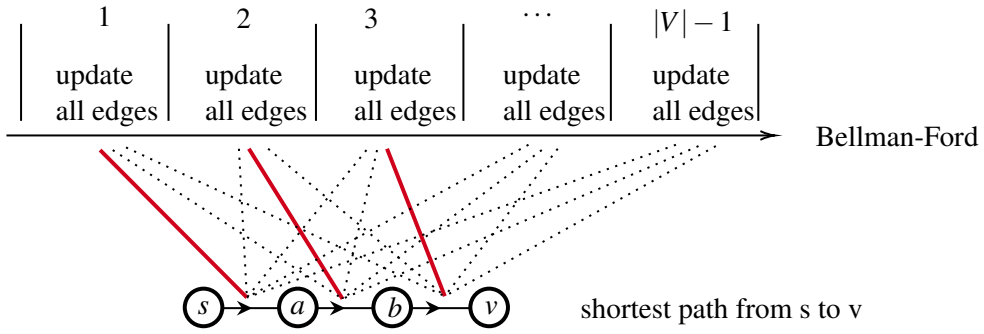


Figure 2: Illustration of the correctness of the Bellman-Ford algorithm. Dotted lines represent additional updates on the corresponding edge.

Detecting Negative Cycles

We can slightly modify Bellman-Ford algorithm to detect if a given graph contains negative cycle that is reachable from s. The algorithm does one more round of updates, in which it determines if some $dist$ value can be further reduced.

```
Algorithm Bellman-Ford-Detect-Negative-Cycle (G = (V,E), l(e) for any e ∈ E, s ∈ V)
    init an array dist of size |V|;
    dist[s] = 0; dist[v] = ∞ for any v ≠ s;
    for k = 1 → |V| − 1
        for each edge (u,v) ∈ E
            update(u,v);
        end for;
    end for;
    for each edge (u,v) ∈ E
        if (dist[v] > dist[u] + l(u,v)): report G contains negative cycle and exit
    end for;
    report that G does not contain negative cycle
end algorithm;
```

Let's show that above algorithm is correct. We first prove that, if G does not contain negative cycle (reachable from s), then in above additional round $dist[v] > dist[u] + l(u,v)$ will never happen, i.e., we will get the report that "G does not contain negative cycle". As per Fact 7 and the assumption that G does not contain negative cycle, we know that $dist[v] = distance(s,v)$ after $|V|-1$ rounds. Also, according to Fact 4, update function will never make $dist[v]$ smaller than $distance(s,v)$ when G does not contain negative cycle. Hence, during

the $|V|$-th round in above algorithm, none of the *dist* value can be further reduced.

We then prove that, if G contains negaive cycle (reachable from s), then in above additional round, there must exist an edge (u,v) such that $dist[v] > dist[u] + l(u,v)$. Suppose conversely that, in above additional round, all edges satisfy $dist[v] \leq dist[u] + l(u,v)$. Let $C = v_1 \to v_2 \to \cdots \to v_{k-1} \to v_k \to v_1$ be one negative reachable from s. We have $\sum_{e \in C} l(e) < 0$ as C is a nagative cycle. Applying $dist[v] \leq dist[u] + l(u,v)$ to all edges in C gives:

$$\begin{array}{rcl} dist[v_2] & \leq & dist[v_1] + l(v_1, v_2) \\ dist[v_3] & \leq & dist[v_2] + l(v_2, v_3) \\ & \cdots & \\ dist[v_k] & \leq & dist[v_{k-1}] + l(v_{k-1}, v_k) \\ dist[v_1] & \leq & dist[v_k] + l(v_k, v_1) \end{array}$$

Summing up both sides of all above inequalities gives $\sum_{e \in C} l(e) \geq 0$, a contradiction.