CS 466/666: Algorithm Design and Analysis **University of Waterloo: Fall 2025**

Lecture 23

November 27, 2025

Instructor: Sepehr Assadi

Topics of this Lecture

We study the breakthrough algorithm of [BNW22] for the negative weight shortest path in this and the next lecture.

1 The Negative Weight Single Source Shortest Path Problem

A standard textbook problem in algorithm design is computing (single-source) shortest paths in graphs with negative edge weights (the *SSSP* problem). Specifically, suppose we have a weighted (directed) graph $G = (V, E, w)$ with *integer* weights $w(e)$ for each $e \in E$. Let n denote the number of vertices in G and m be the number of edges. Assume there is no negative weight cycle in G. Given a source vertex $s \in V$, the goal is to compute $dist_G(s, v)$ for all $v \in V$, namely, the weight of the shortest path from s to each vertex.

The classical Bellman-Ford algorithm solves the SSSP problem as follows.

Algorithm 1. The Bellman-Ford algorithm for SSSP.

1. Initialize $d[s] = 0$ and all other vertices $d[v] = \infty$.
2. For $n - 1$ times:
   For every edge $(u, v) \in E$, update $d[v] \leftarrow \min(d[v], d[u] + w(u, v))$.

The analysis of this algorithm is quite standard and we do not repeat it here. We only point out that $d[v]$ is at each step is an upper bound on the $dist_G(s, v)$ and that after the i-th iteration, $d[v] = dist_G(s, v)$ for every vertex v whose shortest (weight) path from s uses at most i hops (edges).

A beautiful result due to [BNW22] from just a couple of years ago shows that, with the help of randomization, we can solve this problem much faster that this classical approach.

Theorem 1 ([BNW22]). *There is a randomized algorithm that for any graph $G = (V, E, w)$ with no negative cycle, finds single-source shortest paths from a given vertex $s \in V$ in $\widetilde{O}(m \log W)$ time*[1] *with high probability where $W := \max_e |w(e)|$, namely, the largest absolute value of any edge weight.*

We will go over the main ideas in the proof of Theorem 1 in this course. Today's lecture is dedicated to covering required backgrounds before we can start the proof of Theorem 1.

2 Background Tools

2.1 Price Function

A classical approach—dating back to Johnson in 1977—for solving negative weight shortest path is to first make the edge weights non-negative, and then simply run Dijkstra's algorithm that works in $O(m \log n)$ time (but only works for non-negative graphs). We just need to ensure that in the process of making edge weights non-negative, we do not change the shortest paths of the graph[2].

Let ϕ be any integer function on vertices, i.e., $\phi : V \to \mathbb{Z}$. For any edge $(u, v) \in E$, define a *new* weight

$$w_\phi(u, v) := w(u, v) + \phi(u) - \phi(v).$$

We refer to ϕ as a **price functions**. We claim that under *any* price function, the shortest paths between vertices do not change. Consider any pair $u, v \in V$ and any u-v path $P_{uv} = (u, z_1, \cdots, z_k, v)$:

$$\begin{aligned} w_\phi(P_{uv}) &= w_\phi(u, z_1) + w_\phi(z_1, z_2) + \cdots + w_\phi(z_k, v) \\ &= w(u, z_1) + \phi(u) - \phi(z_1) + w(z_1, z_2) + \phi(z_1) - \phi(z_2) + \cdots + w(z_k, y) + \phi(z_k) - \phi(y) \\ &= w(u, z_1) + w(z_1, z_2) + \cdots + w(z_k, y) + \phi(u) - \phi(v) \\ &= w(P_{uv}) + \phi(u) - \phi(v). \end{aligned}$$

Consequently, a price function changes the weights of *all* u-v paths by the same amount of value, i.e., $\phi(u) - \phi(v)$. This implies that under *any* price function, the shortest paths of the graph do not change.

The question now is that is there any price function that can make all edges in the graph non-negative? The answer turns out to be *yes*. Add a new vertex s^* and connect it to every vertex $v \in V$ with weight $w(s^*, v) = 0$. Set $\phi(v) := \text{dist}_G(s^*, v)$ to obtain the price function ϕ. Now, for every edge $(u, v) \in E$, the new weight satisfies

$$w_\phi(u, v) = w(u, v) + \phi(u) - \phi(v) = w(u, v) + dist_G(s^*, u) - dist_G(s^*, v) \geqslant 0,$$

where the inequality holds by triangle inequality (since going from s^* to v cannot be costlier than going to u first and then taking the $w(u, v)$ edge).

Thus, we can always make the edge weights non-negative using a price function. The problem with the above approach however is that we have to solve an entire negative-weight single-source shortest path problem before we can obtain ϕ, which defeats the purpose for us. In this lecture, we see how one can find price functions more efficiently. To give an example, let us consider the case when G is a DAG.

Example: price function for DAGs. If G is a directed acyclic graph (DAG), we can compute its topological order $v_1, v_2, \cdots, v_n$ and set $\phi(v_i) = (n - i + 1) \cdot W$. We will have $w_\phi(u, v) \geq 0$ since u has to appear before v in the ordering and $w(u, v) \geq -W$ for any $(u, v) \in E$. Thus, we can find a price function for DAGs in only $O(m + n)$ time using the topological sort algorithm.

[1] Recall that $\widetilde{O}(f) := O(f \cdot \text{poly}\log(f))$.

[2] The "obvious" approach to make edge weights non-negative is to just add a very large number to each edge weight. However, this approach will destroy the shortest paths since for a given pair u, v, the weights of different u-v paths are penalized differently according to the number of edges on the path.

2.2 Reducing Out Degrees

For our next algorithms, it helps if out-degrees of all vertices is not much larger than the average degree. We can achieve this easily when solving SSSP by modifying the graph slightly. Given any directed graph G, consider creating a new directed graph G' as follows:

- Let $\deg^+(v)$ denote the out-degree of v. Split any vertex v with "too high" out-degree: specifically, for any vertex v with $\deg^+(v) \geqslant m/n$, create $k := \left\lceil \frac{\deg^+(v)}{m/n} \right\rceil$ vertices $v_1, \ldots, v_k$.
- Then, add a cycle of weight 0 edges from v_1 to v_2 to v_k and back to v_1. All incoming edges to v now enter v_1 instead. Then split the outgoing edges of v evenly among $v_1, \ldots, v_k$.

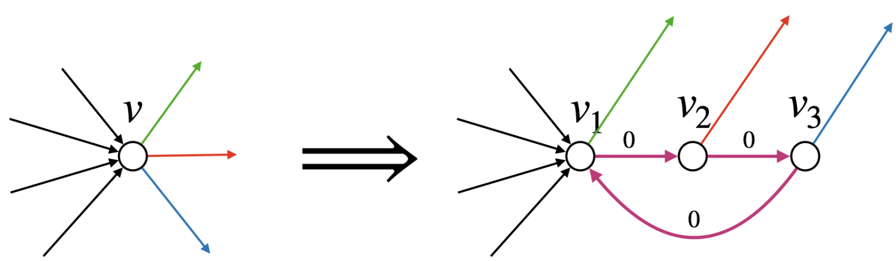

Figure 1: An illustration of splitting a vertex with out-degree 3 into 3 vertices with out-degree 2 (an "internal" out-degree with weight 0 and one "external" out-degree corresponding to one out-edge of the original vertex).

It is easy to see that for any vertex $v \in G$ and any of its copies $v_i \in G'$, we have $dist_G(s, v) = dist_{G'}(s', v_i)$ where s' is the copy s_1 in G'. This is because a path in G can be traversed exactly the same in G' except we use 0-weight edges inside each copy of a vertex until we reach the required out-edge copied from G. Thus, solving SSSP on G' gives us the solution on G as well.

After this step, we have

$$n(G') = \sum_{v \in V} \left\lceil \frac{\deg^+(v)}{m(G)/n(G)} \right\rceil \leqslant \sum_{v \in V} \left(\frac{\deg^+(v)}{m(G)/n(G)} + 1 \right) = \left(\frac{n(G)}{m(G)} \cdot \sum_{v \in V} \deg^+(v) \right) + n(G) = 2 \cdot n(G),$$

as sum of out-degrees is equal to the number of edges. Moreover,

$$m(G') \leqslant m(G) + n(G') = m(G) + 2 \cdot n(G),$$

because every vertex of G' can have at most one additional out-edge compared to G, namely, its 0-weight edge in the split-off cycle it belongs to. Finally, out-degree of any vertex in G' is at most

$$1 + \frac{\deg^+(v)}{\left\lceil \frac{\deg^+(v)}{m(G)/n(G)} \right\rceil} \leqslant 1 + \frac{m(G)}{n(G)}.$$

So, all in all, we transformed G into a graph with $O(n)$ vertices, $O(m)$ edges, and maximum out-degree $O(m/n)$ as desired.

Assumption: From now on, without loss of generality, we assume that we are solving SSSP on an m-edge n-vertex graph with maximum out-degree $O(m/n)$ and we may use these bounds implicitly in our proofs without repeatedly reminding the reader of this assumption.

2.3 Further Properties of Bellman-Ford and Dijkstra's Algorithms

We now list some simple modifications of Bellman-Ford and Dijkstra's algorithms that we use in the rest of this lecture. For this, we need some definition.

Definition 2. For any vertex $v \in V$, we use $h_s(v)$ to denote the smallest number of edges in any shortest path from s to v, i.e.,

$$h_s(v) := \min \{\# \text{ edges in } P_{sv} \ \mid w(P_{sv}) \text{ is minimum among all } s\text{-}v \text{ paths}\}.$$

We refer to $h_s(v)$ as the **hop-distance** of v from s. Define

$$\overline{h}_s := \frac{1}{n} \sum_{v \in V} h_s(v),$$

as the average hop-distance of vertices from s.

Similarly, we use $b_s(v)$ to denote the smallest number of negative-weight edges in any shortest path from s to v and call it the **negative-hop-distance** of v from s. Finally, $\overline{b}_s$ is the average negative-hop-distance of vertices from s.

Let us see these definitions in action.

Claim 3. *There is an algorithm for solving SSSP in any graph in $O(m \cdot \overline{h}_s)$ time.*

Proof. We run the following slight variation of the Bellman-Ford algorithm (Algorithm 1):

Algorithm 2. A modification Bellman-Ford algorithm for SSSP based on average hop-distance.

1. Initialize $d[s] = 0$ and all other vertices $d[v] = \infty$.
2. Let Q be a queue and s to Q.
3. For $i = 1$ to $n - 1$ iterations:
   - While Q is non-empty:
     Dequeue Q to get a vertex u; for any edge $(u, v) \in E$, update $d[v] \leftarrow min(d[v], d[u] + w(u, v))$.
   - Add all vertices whose d-value changed to the Q.

The correctness is the same as the original Bellman-Ford algorithm: we effectively did not change the algorithm at all, and only skipped visiting edges (u, v) in an iteration i of the for-loop, if the d-value of u has not changed from the previous iteration; since those edges were not going to lead to any update of $d[v]$ anyway (otherwise, we would have updated $d[v]$ in the previous iteration of the for-loop), the algorithm and its correctness remains the same.

For the runtime analysis, recall that in the Bellman-Ford algorithm, value of $d[v]$ becomes $dist_G(s, v)$ after at most $h_s(v)$ iteration of the for-loop. Thus, we will only go over edges of v for at most $h_s(v)$ time and each time we spend $\deg^+(v) = O(m/n)$ time. Hence, the total runtime is

$$O(n) + \sum_{v \in V} h_s(v) \cdot O(\frac{m}{n}) = O(n) + O(m) \cdot \frac{1}{n} \sum_{v \in V} h_s(v) = O(n + m \cdot \overline{h}_s) = O(m \cdot \overline{h}_s),$$

as desired. □

Claim 4. *Let $b_s^* := \max_{v \in V} b_s(v)$, namely, the* <u>*maximum*</u> *negative-hop-distance of any vertex v from s. There is an algorithm for solving SSSP in any graph in $O(m \log n \cdot b_s^*)$ time.*

Proof. The algorithm is a direct combination of Bellman-Ford and Disjkstra's algorithms.

Algorithm 3. A combination of Bellman-Ford and Dijkstra's for SSSP based on maximum negative-hop-distance.

1. Initialize $d[s] = 0$ and all other vertices $d[v] = \infty$.
2. Run Dijkstra's algorithm on non-negative edges in G from s and update the distances d.
3. For $i = 1$ to $n - 1$ iterations:
   - Run one iteration of Bellman-Ford update on all negative edges, i.e., go over all negative edges once and update
   $$d[v] \leftarrow min(d[v], d[u] + w(u, v))$$
   for each negative edge (u, v).
   - Run Dijkstra's algorithm on non-negative edges in G using the initial distances d. I.e., each vertex is inserted into the priority queue of Dijkstra's algorithm originally with d-value as its priority (but we can update d-values in Dijkstra's algorithm as is standard).
   - If no distances were updated in this iteration, terminate the for-loop.

The correctness of the algorithm follows by proving that after i-th iteration of the for-loop, $d[v] = dist_G(s, v)$ for any vertex v with $b_s(v) \leqslant i$. This can be proven by induction: for the base case of $i = 0$, running the Dijsktra's algorithm outside the for-loop ensures we compute distances to vertices v with $b_s(v) = 0$ correctly. For the induction step, in iteration i of the for-loop, we start with correct distances for vertices with $b_s(v) \leqslant i - 1$, then, consider taking any one negative edge (in the Bellman-Ford step), and then do a complete Dijkstra step to handle all non-negative edges. If s is reaching v using i negative edges, and the last negative edge is (w, w'), we have $d[w] = dist_G(s, w)$ by induction at the beginning of the for-loop, thus, $d[w'] = dist_G(s, w')$ after running the Bellman-Ford step, and $d[v] = d[w'] + w(P_{w'v})$ where $P_{w'v}$ is the shortest path from w' to v which is not using any non-negative edges (by definition of $b_s(v) \leqslant i$), and thus the Dijkstra step updates $d[v]$ correctly.

By the above argument, the for-loop is run at most $b_s^* + 1$ steps before the algorithm terminates as no distance is being updated. Each for-loop iteration also takes $O(m + m \log n)$ time using standard implementation of Dijkstra's algorithm, giving the final runtime. ☐

The main lemma we need is a combination of the above two algorithms with runtime depending on *average* negative-hop-distances and not maximum.

Lemma 5. *There is an algorithm for solving SSSP in any graph in $O(m \log n \cdot \overline{b}_s)$ time.*

Note that in none of the algorithms in this section, we needed to know the value of parameters $\overline{h}_s, b_s^*$ or $\overline{b}_s$ in advance, and the algorithm figures it out on its own. We will not prove Lemma 5 as its proof is along the lines of the two prior claims but with more care; see [BNW22, Lemma 3.3] for a complete proof.

2.4 Low Diameter Decomposition for Directed Graphs

The final tool we need is an analogue of low diameter decomposition of Lecture 22 that worked on undirected graphs, but now for directed graphs. Directed LDDs were introduced first in [BNW22] and we use the following result from them directly, but note that at this point, stronger guarantees have been obtained in the literature as well.

Theorem 6 (Directed Low Diameter Decomposition [BNW22]). *There is a randomized algorithm that given any directed graph $G = (V, E, w)$ with* <u>*non-negative*</u> *integer weights and an integer $D \geqslant 1$, outputs a set of edges E_{rem} with the following properties:*

- *Let C be any* strongly connected component (SCC) *of $G \setminus E_{\text{rem}}$. Then, C has a "weak diameter" at most D:*
$$\forall u, v \in C \quad dist_G(u, v) \leqslant D \quad \text{and} \quad dist_G(v, u) \leqslant D.$$
- *For any edge $e \in E$,*
$$\Pr\left(e \in E_{\text{rem}}\right) = \frac{O(\log^2 n)}{D} + n^{-10}.$$

The algorithm runs in $O(m \log^3 n)$ time deterministically (in fact, the runtime is $O(m \log^2 n + n \log^3 n)$ but the distinction is not important for us in this lecture).

Notice that a key difference of Theorem 6 for LDDs for undirected graphs we covered in Lecture 22, is that in directed graphs, there are still edges possible between clusters that are not part of E_{rem}. However, these edges form a DAG as we collected all SCCs of G into a cluster each. The proof of this theorem is along the lines of LDDs for undirected graphs covered in Lecture 22 with a lot more technical details. We will not cover this proof in this paper and refer the reader to [BNW22, Lemma 1.2] for its proof.

In the next lecture, we see how a clever combination of these tools allows us to prove Theorem 1.

References

[BNW22] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. Negative-weight single-source shortest paths in near-linear time. In *2022 IEEE 63rd annual symposium on foundations of computer science (FOCS)*, pages 600–611. IEEE, 2022. 1, 2, 5, 6