Chapter 21

Shortest Paths II: Negative-Weights in Nearly-Linear Time

We study the breakthrough algorithm of [BNW22] for the *negative weight shortest path* problem in this chapter.

21.1 Negative Weight (Single-Source) Shortest Paths

A standard textbook problem in algorithm design is computing (single-source) shortest paths in graphs with negative edge weights (the *SSSP* problem).

Problem 21.1. Suppose we have a weighted *directed* graph $G = (V, E, w)$ with *integer* weights $w(e) \in \mathbb{Z}$ for each $e \in E$. Given a source vertex $s \in V$, the goal is to compute $\text{dist}_G(s, v)$ for all $v \in V$, namely, the weight of the shortest path from s to each vertex.

Let n denote the number of vertices and m be the number of edges in the graph G we are working with. Throughout this chapter, we make the following assumption.

Assumption 21.2. There is no negative weight cycle in G.

This assumption is without loss of generality in the sense that if the graph has a negative-weight cycle, the algorithms we cover will be able to find it easily and report that. However, finding a shortest path in a graph in the presence of negative weight cycles in general is an **NP**-hard problem and thus we are not planning to work with such graphs in this chapter.

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

Algorithm 21.3.

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 an upper bound on $dist_G(s, v)$ at all times, and that after the i-th iteration, $d[v] = dist_G(s, v)$ for every vertex v whose shortest path from s uses at most i hops (edges).

A recent and remarkable result due to [BNW22] shows that, with the help of randomization, we can solve this problem much faster than this classical approach.

Theorem 21.4 ([BNW22]). *There is a randomized algorithm that for any graph $G = (V, E, w)$ with no negative cycles, 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 21.4 in this chapter.

21.2 Background Tools

This section is dedicated to covering backgrounds before we can start the proof of Theorem 21.4.

21.2.1 Price Function

A classical approach—dating back to [Joh77]—for solving negative weight shortest path is to first make the edge weights non-negative, and then run Dijkstra's algorithm in $O(m \log n)$ time (which only works for graphs with non-negative weights). 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 function**.

Claim 21.5. *Under* any *price function, the shortest paths between vertices do not change.*

Proof. Consider any pair $u, v \in V$ and any u-v path $P_{uv} = (u, z_1, \cdots, z_k, v)$. The sum of new weights basically telescopes out the price function except for u and v:

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

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

The question now is whether there is any price function that can make all edges in the graph non-negative? The answer turns out to be *yes*.

Claim 21.6. *Add a new vertex s^* and connect it to every vertex $v \in V$ with weight $w(s^*, v) = 0$. The price function $\phi(v) = \text{dist}_G(s^*, v)$ makes all edges non-negative.*

Proof. 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). □

[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.

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 when the goal is to solve negative-weight single-source shortest path in the first place! The approach of Theorem 21.4 is to come up with a good price function much more efficiently. To give an example of an easier-to-compute price function, let us consider the case when G is a DAG.

Observation 21.7. *For any directed acyclic graph (DAG) G, we can find a price function in $O(m+n)$ time that makes all edges non-negative.*

Proof. We compute the topological order $v_1, v_2, \cdots, v_n$ of G 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 $w(u,v) \geqslant -W$ 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. □

21.2.2 Reducing Out Degrees

For our next algorithms, it helps if out-degree of no vertex is 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:

1. For any vertex v with $\deg^+(v) \geqslant m/n$, create $k := \left\lceil \frac{\deg^+(v)}{m/n} \right\rceil$ vertices $v_1, \ldots, v_k$.
2. 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 $v_1, \ldots, v_k$.

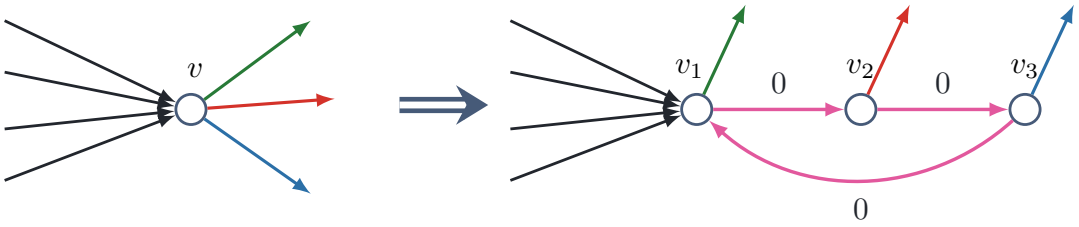

Figure 21.1: An illustration of splitting a vertex with out-degree 3 into 3 vertices with out-degree 2 (an "internal" out-edge with weight 0 and one "external" out-edge corresponding to one out-edge of the original vertex).

It is easy to see that for any vertex $v \in V$ 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.

From now on, without loss of generality, we will make the following assumption.

Assumption 21.8. The input graph to the SSSP problem has m edges, n vertices, and $O(m/n)$ maximum out-degree (we may use these bounds implicitly in the rest of this chapter).

21.2.3 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.

Definition 21.9. 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.

Claim 21.10. *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 21.3):

Algorithm 21.11.

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

For the runtime analysis, recall that in the Bellman-Ford algorithm, $d[v]$ becomes $dist_G(s,v)$ after at most $h_s(v)$ iterations of the for-loop. Thus, we will only go over edges of v for at most $h_s(v)$ times 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 21.12. *Let $b^*_s := \max_{v \in V} b_s(v)$, namely, the maximum 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 Dijkstra's algorithms.

Algorithm 21.13.

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 to update 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 using the initial distances d. I.e., each vertex $v \in V$ is inserted into the priority queue initially with priority $d[v]$ (we 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 Dijkstra'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 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 21.14. *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 21.14 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.

21.2.4 Low Diameter Decomposition for Directed Graphs

The final tool we need is an analogue of low diameter decomposition of Theorem 20.10, which 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 21.15 (Directed Low Diameter Decomposition [BNW22])**.** *There is a randomized algorithm that given any directed graph $G = (V, E, w)$ with non-negative 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 $e \in E$,*
$$\Pr\left(e \in E_{\text{rem}}\right) = w(e) \cdot \frac{O(\log^2 n)}{D} + n^{-10}.$$

The algorithm takes $O(m \log^3 n)$ time with certainty, namely, with probability one (in fact, the runtime is $O(m \log^2 n + n \log^3 n)$ but the distinction is not important for us).

A key difference of Theorem 21.15 from the undirected LDDs we covered in Theorem 20.10 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 the proof of Theorem 20.10 with a lot more technical details. We will not cover this proof and refer the reader to [BNW22, Lemma 1.2] for its proof.

21.3 An $\widetilde{O}(m\sqrt{n})$ Time Algorithm for SSSP

We start by proving a weaker version of Theorem 21.4 which has many of the key ideas. We then sketch how we can extend this to prove Theorem 21.4 also. In particular, our goal now is to prove the following simpler theorem which gives an $\widetilde{O}(m\sqrt{n})$ time algorithm for SSSP.

Theorem 21.16 (Weaker version of Theorem 21.4 [BNW22])**.** *There is a randomized algorithm that for any graph $G = (V, E, w)$ with no negative cycles, finds single-source shortest paths from a given vertex $s \in V$ in $O(m\sqrt{n} \log^2 n \cdot \log(nW))$ expected time.*

The general framework of the proof is as follows: suppose we start with G and a weight function w such that $w(e) \geqslant -2B$ for all $e \in E$ and some integer $B \geqslant 1$. We will find a price function ϕ such that after applying it, we will have $w_\phi(e) \geqslant -B$ for all $e \in E$; i.e., the most negative-weight edge is now at most half as negative as before. We then show that repeating this step for $O(\log W)$ iterations is enough to obtain weights that can be thought of as essentially non-negative. This approach is often called **scaling** and is a classical technique in algorithm design. Let us now formalize this further.

Lemma 21.17. *There is a randomized algorithm that given any graph $G = (V, E, w)$ where $w(e) \geqslant -2B$ for every edge $e \in E$, outputs a price function ϕ such that $w_\phi(e) \geqslant -B$ for every edge $e \in E$. The algorithm has expected $O(m\sqrt{n}\log^2 n)$ time.*

Let us see how to use this lemma now to obtain an $\widetilde{O}(m\sqrt{n})$ time algorithm for SSSP.

Proof of Theorem 21.16 using Lemma 21.17. Given $G = (V, E, w)$ with $W := \max_e |w(e)|$, we first update the weight function w so that $w(e) \leftarrow n \cdot w(e)$. This does not change the shortest path structure (nor change positivity/negativity of any edge). We then run Lemma 21.17 repeatedly for $t := \log(nW)$ iterations on G to obtain price functions $\phi_1, \phi_2, \ldots, \phi_t$, where

$$w(e) \geqslant -nW \implies w_{\phi_1}(e) \geqslant -\frac{nW}{2} \implies w_{\phi_2}(e) \geqslant -\frac{nW}{2^2} \implies \cdots \implies w_{\phi_t}(e) \geqslant -\frac{nW}{2^t} = -1,$$

where each '$\implies$' corresponds to running the algorithm of Lemma 21.17 once. Note that here, each price function ϕ_i is applied on top of the price function ϕ_{i-1}, i.e., is obtained by adding the price function of Lemma 21.17 to the price function ϕ_{i-1}.

At this point, we define a new weight function w' wherein $w'(e) = w_{\phi_t}(e) + 1$. In principle, this can potentially change the shortest path structure. Nevertheless, we prove that, in our special case, this cannot happen. Suppose P and Q are two s-v paths in G (under the original weight function w) and that $w(P) < w(Q)$. We argue that $w'(P) < w'(Q)$ also. We have,

$$w'(P) = w_{\phi_t}(P) + |P| = w(P) + \phi_t(s) - \phi_t(v) + |P| \leqslant w(P) + (n-1) + \phi_t(s) - \phi_t(v),$$

since P can have $n-1$ edges at most. On the other hand

$$w'(Q) = w_{\phi_t}(Q) + |Q| = w(Q) + \phi_t(s) - \phi_t(v) + |Q| \geqslant w(Q) + \phi_t(s) - \phi_t(v).$$

But recall that since we updated the weights w by multiplying them by n, if $w(P) < w(Q)$, then in fact $w(P) \leqslant w(Q) - n$. Thus, we continue to have $w'(P) < w'(Q)$ also as desired.

Since w' is a non-negative weight function, we can simply run Dijkstra's algorithm on G, w' and solve SSSP in $O(m \log n)$ time at this point (and by the previous argument and correctness of price functions, we get the solution is correct). This takes $O(m\sqrt{n} \cdot \log^2 n \cdot \log(nW))$ time in expectation as desired. □

21.3.1 Proof of Lemma 21.17: The Scaling Lemma

We update the graph by adding a vertex s^* that is connected to every vertex with an edge of weight $-B$. Throughout, we use the same set of vertices $\{s^*\} \cup V$ but with different subset of edges (subsets of E which is now updated to include (s^*, v)-edges as well) and different weight functions. In particular, define the following two additional weight functions:

$$w^{2B} : E \to \mathbb{Z} \quad \text{wherein } w^{2B}(e) = w(e) + 2B \text{ for all edges } e \in E;$$
$$w^{B} : E \to \mathbb{Z} \quad \text{wherein } w^{B}(e) = w(e) + B \text{ for all edges } e \in E.$$

Note that both these weight functions can potentially destroy the shortest path structure (but that will not be a concern for us because we will only use them to compute a price function). Moreover, w^{2B} is now a non-negative weight function.

The algorithm, at a high level, is as follows.

Algorithm 21.18.

1. Compute an LDD of $G^{2B} = (\{s^*\} \cup V, E, w^{2B})$ with diameter $D = dB$ using Theorem 21.15. Let $C_1, \ldots, C_k$ be the SCCs and E_{rem} be the removed edges of the LDD.
   ▷ The parameter d will be set later.
2. Use the weights w^B (and not w^{2B}) to find a price function ϕ_1 that makes all edges inside C_i's non-negative in $w^B_{\phi_1}$. ▷ This and next steps are explained in more detail later.
3. Use the updated weights $w^B_{\phi_1}$ to find a price function ϕ_2 that additionally makes the DAG edges of $G \setminus E_{\text{rem}}$ non-negative in $w^B_{\phi_2}$.
4. Use the updated weights $w^B_{\phi_2}$ to find a price function ϕ_3 that additionally makes the edges in E_{rem} non-negative in $w^B_{\phi_3}$.
5. Return w_{ϕ_3} (and not $w^B_{\phi_3}$) as a weight function that satisfies $w_{\phi_3}(e) \geqslant -B$ for all $e \in E$.

We will now go over different steps of this algorithm in detail; Figure 21.2 summarizes the three kinds of edges that arise and the step of the algorithm that makes each of them non-negative.

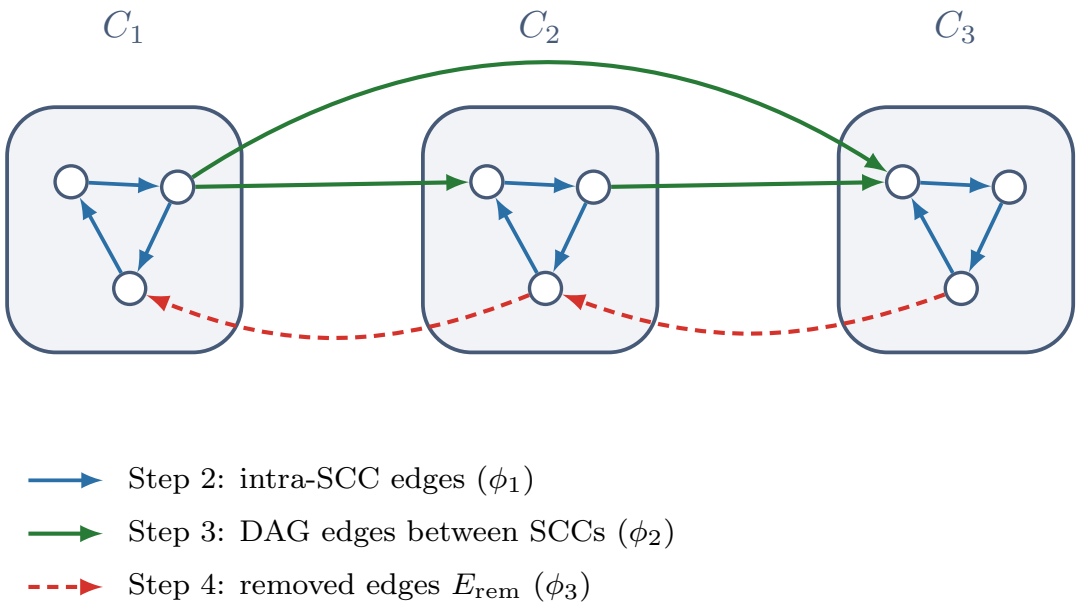

Figure 21.2: The three kinds of edges that remain after the low-diameter decomposition of Step 1 of Algorithm 21.18, and the step that makes each of them non-negative: the edges *inside* each SCC (Step 2, via ϕ_1), the *DAG* edges between SCCs taken in topological order (Step 3, via ϕ_2), and the removed edges E_{rem} (Step 4, via ϕ_3).

Step 1: LDD computation. Recall that an LDD is only defined for graphs with non-negative weights. Since $w(e) \geqslant -2B$ by assumption and $w^{2B}(e) = w(e) + 2B$ by definition, we have w^{2B} is a non-negative weight function. As such, it is valid to apply Theorem 21.15 and obtain a set E_{rem} of edges such that any SCC C of $G \setminus E_{\text{rem}}$ satisfies:

$$\forall u, v \in C \quad dist_{G^{2B}}(u, v) \leqslant dB \quad \text{and} \quad dist_{G^{2B}}(v, u) \leqslant dB, \tag{21.1}$$

and for any edge in G

$$\Pr\left(e \in E_{\text{rem}}\right) = \frac{O(\log^2 n)}{dB} \cdot w^{2B}(e) + n^{-10}. \tag{21.2}$$

This step takes $O(m \log^3 n)$ time.

Step 2: Fixing SCC edges. Consider the graph $G_1 = (\{s^*\} \cup V, E_1, w^B)$ (with weight function w^B) where $E_1 \subseteq E$ only contains the edges inside SCCs of $G \setminus E_{\text{rem}}$, together with the edges outsides out of s^*. Note that the edges of s^* to all other vertices have weight 0 under w^B (as they had weight $-B$ under w). We claim that the shortest paths from s^* in this graph have "few" negative edges.

Claim 21.19. *For any $v \in V$, the hop distance of s^* to v in G_1 is at most d.*

Proof. Consider a shortest path P_{s^*v} in G_1 from s^* to v in G_1. We know that $w^B(P_{s^*v}) \leqslant 0$ as s^* is connected to v by an edge of weight 0. If $w^B(P_{s^*v}) = 0$, the hop distance of s^* to v will simply be one by taking the (s^*, v) edge directly. Otherwise, we have $w^B(P_{s^*v}) < 0$ and thus P_{s^*v} starts by going from s^* to some vertex u in the same SCC as v and then taking the shortest path P_{uv} inside this SCC (recall that the only edges of G_1 are the ones inside SCCs). We will argue that P_{uv} has fewer than d edges. Figure 21.3 illustrates the resulting contradiction.

Suppose towards a contradiction that P_{uv} contains at least d edges in G_1. Then, under the original weight function w, we have,

$$w(P_{uv}) = w^B(P_{uv}) - |P_{uv}| \cdot B < 0 - dB = -dB,$$

since $|P_{uv}| \geqslant d$ by our assumption. On the other hand, since u and v are both inside the same SCC of $G \setminus E_{\text{rem}}$, by Eq (21.3), we have

$$dist_{G^{2B}}(v, u) \leqslant dB.$$

This implies that there exists some path Q_{vu} in G (and not necessarily G_1) such that

$$w(Q_{vu}) = w^{2B}(Q_{vu}) - |Q_{vu}| \cdot 2B \leqslant dB - 1.$$

Putting these two implies that in the original graph G, we can go from u to v with a path of weight $< -dB$ and from v to u with a path of weight $< dB$. This implies that we can start from u and return to it by paying a total weight $< -dB + dB = 0$, implying that there must be a negative cycle in G. But this is a contradiction with Assumption 21.2 (equivalently, the hypothesis of Theorem 21.16). □

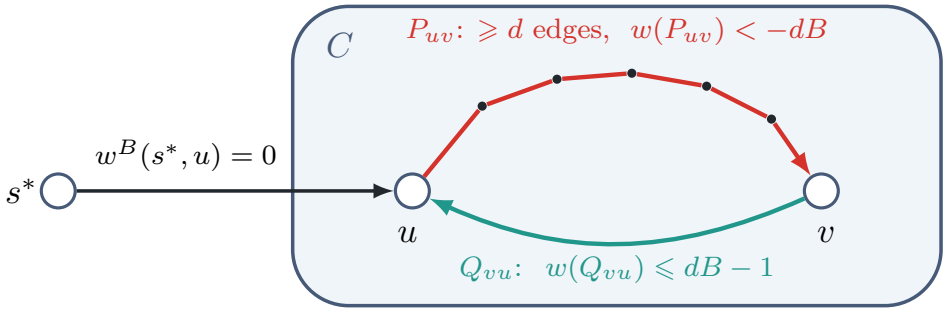

Figure 21.3: Illustration of the proof of Claim 21.19. A shortest path from s^* enters the SCC C at a vertex u along a w^B-weight-0 edge. If its part P_{uv} inside C used at least d edges, then $w(P_{uv}) < -dB$; together with the return path Q_{vu} of weight at most $dB - 1$ (guaranteed by the weak diameter of the decomposition from Step 1), this closes a cycle $u \to v \to u$ of total weight below $-dB + dB = 0$—a negative cycle, contradicting the assumption that G has none.

Combining Claim 21.19 with Lemma 21.14 implies that we can find s^*-shortest paths in G_1 in $O(m \log n \cdot d)$ time. We will then define, for any $v \in V$,

$$\phi_1(v) = dist_{G_1}(s^*, v).$$

This implies that for any edge $(u,v) \in G_1$,

$$w^B_{\phi_1}(u,v) = w^B(u,v) + \phi_1(u) - \phi_1(v) = w^B(u,v) + dist_{G_1}(s^*,u) - dist_{G_1}(s^*,v) \geqslant 0,$$

where the last inequality is by triangle inequality since (u,v) is an edge of G_1. Thus, we made all SCC edges non-negative under $w^B_{\phi_1}$.

Step 3: Fixing DAG edges. This step is quite straightforward: we compute a topological ordering of the SCCs of the graph $G \setminus E_{\text{rem}}$ in $O(m+n)$ time and denote them by $C_1, \ldots, C_k$. For any $v \in C_i$, we define

$$\phi_2(v) = \phi_1(v) + (k-i) \cdot \max_{e \in E} |w^B_{\phi_1}(e)|.$$

Note that for any edge (u,v) inside the same cluster C_i, we have

$$w^B_{\phi_2}(u,v) = w^B(u,v) + \phi_2(u) - \phi_2(v) = w^B(u,v) + \phi_1(u) - \phi_1(v) = w^B_{\phi_1}(u,v) \geqslant 0,$$

as we proved in the last part. For any edge $u \in C_i$ and $v \in C_j$ for $j > i$,

$$\begin{aligned} w^B_{\phi_2}(u,v) &= w^B(u,v) + \phi_2(u) - \phi_2(v) = w^B(u,v) + \phi_1(u) - \phi_1(v) + \max_{e \in E}(j-i) \cdot |w^B_{\phi_1}(e)| \\ &= w^B_{\phi_1}(u,v) + (j-i) \cdot \max_{e \in E} |w^B_{\phi_1}(e)| \geqslant 0. \end{aligned}$$

Finally, since we are working with a topological ordering of a DAG there are no other edges, and thus all edges, except for E_{rem}, have become non-negative in $w^B_{\phi_2}$.

Step 4: Fixing E_{rem} edges. We now consider s^*-shortest paths in the entire graph G under the updated weight function $w^B_{\phi_2}$. We claim that these shortest paths also contain only a "few" negative edges on average.

Claim 21.20. *For any $v \in V$, the expected negative hop distance of s^* to v in G under the weight function $w^B_{\phi_2}$ is less than $\dfrac{O(h_{s^*}(v) \cdot \log^2 n)}{d}$, where $h_{s^*}(v)$ is the hop distance of s^* to v under the weight function w^B.*

Proof. As in Claim 21.19, we consider a path P_{s^*v} that goes from s^* to some vertex u and then takes P_{uv} with $w^B_{\phi_2}(P_{uv}) < 0$ (otherwise, the hop distance of s^* to v will be one). Note that P_{uv} is also the shortest path from u to v in w^B itself since price functions do not change the shortest path structure. But under w^B, we could have again gone from s^* to v with a weight of 0, and thus $w^B(P_{uv}) < 0$. This implies that

$$w^{2B}(P_{uv}) = w^B(P_{uv}) + |P_{uv}| \cdot B \leqslant h_{s^*}(v) \cdot B.$$

At the same time, the number of negative edges in P_{uv} under $w^B_{\phi_2}$ is at most equal to $|P_{uv} \cap E_{\text{rem}}|$ as previous steps made sure all other edges are non-negative. Thus,

$$\begin{aligned} \mathbb{E}\left[\text{negative hop distance of } s^* \text{ to } v \text{ in } w^B_{\phi_2}\right] &\leqslant \mathbb{E}\,|P_{uv} \cap E_{\text{rem}}| \\ &= \sum_{e \in P_{uv}} \Pr\left(e \in E_{\text{rem}}\right) \end{aligned}$$

(by linearity of expectation because P_{uv} is determined by w^B alone, independent of the LDD)

$$= \sum_{e \in P_{uv}} \frac{O(\log^2 n)}{dB} w^{2B}(e) + n^{-10} \quad \text{(by Eq (21.2))}$$

$$= \frac{O(\log^2 n)}{dB} \cdot h_{s^*}(v) \cdot B + n^{-9}$$
(by the above bound on $w^{2B}(P_{uv})$ and since $|P_{uv}| < n$)
$$= \frac{O(h_{s^*}(v) \cdot \log^2 n)}{d},$$
(n^{-9} is absorbed in the $O(\cdot)$ term)

as desired. □

Since the hop distances are always at most $n-1$, by combining Claim 21.20 and Lemma 21.14, we can find s^*-shortest paths in the entire G under the weight function $w^B_{\phi_2}$ in $O(m \log^3 n \cdot \frac{n}{d})$ expected time. By setting

$$\phi_3(v) = \phi_2(v) + dist_{w^B_{\phi_2}}(s^*, v),$$

for all $v \in V$, we can make all edges of G non-negative under the weight function $w^B_{\phi_3}$. As such, under the original weight function w but with the price function ϕ_3, for any $e \in E$, we have

$$w_{\phi_3}(e) = w^B_{\phi_3}(e) - B \geqslant -B.$$

The expected runtime of the algorithm is now

$$O(m \log^3 n + m \log n \cdot d + m \log^3 n \cdot \frac{n}{d}),$$

and thus by setting $d = \sqrt{n} \log n$, we obtain the expected runtime of

$$O(m\sqrt{n} \log^2 n),$$

concluding the proof of Lemma 21.17.

21.4 The Final $\widetilde{O}(m \log W)$ Time Algorithm

The algorithm in Theorem 21.4 can also be obtained in a very similar manner, using the following improved scaling lemma compared to Lemma 21.17.

Lemma 21.21. *There is a randomized algorithm that given any graph $G = (V, E, w)$ where $w(e) \geqslant -2B$ for every edge $e \in E$, outputs a price function ϕ such that $w_\phi(e) \geqslant -B$ for every edge $e \in E$. The algorithm has expected $O(m \log^4 n)$ time.*

Theorem 21.4 follows from Lemma 21.21 the same exact way Theorem 21.16 followed from Lemma 21.17. We now show how to prove Lemma 21.21.

The idea behind the proof of Lemma 21.21 is to introduce one more level of recursion: instead of balancing the time took in Step 2 and 4 of Algorithm 21.18 that led to an $\widetilde{O}(m\sqrt{n})$ time, we will make Step 4 much faster and then recurse on the graphs of Step 2. The improvement obtained in Step 2 comes from another metric: the negative hop distances of the SCCs still drop by a factor of two, and thus the recursion depth will only be $O(\log n)$ which makes our algorithm fast enough.

The algorithm is as follows. We again use the weight functions w^{2B} and w^B and also add the vertex s^* to the graph as before. The input is a graph $G = (\{s^*\} \cup V, E, w)$ with an additional parameter Δ promised to be an upper bound on the hop distances between all pairs of reachable vertices in V (ignoring s^*) under the weight function w^B. The algorithm returns a price function ϕ such that $w^B_\phi(e) \geqslant 0$ for all $e \in E$. The steps of the algorithm are also explained in more detail later.

Algorithm 21.22.

1. If $\Delta \leqslant 1$, run a base case algorithm (explained below) and return.
2. Compute an LDD of $G^{2B} = (\{s^*\} \cup V, E, w^{2B})$ with diameter $D = dB$ using Theorem 21.15 for a parameter $d = \Delta/2$. Let $C_1, \ldots, C_k$ be the SCCs and E_{rem} be the removed edges of the LDD.
3. Use the weights w^B (not w^{2B}) and recurse on $G_i = (\{s^*\} \cup C_i, E[\{s^*\} \cup C_i], w^B)$ for all $i \in [k]$ with parameter $\Delta/2$ to find a price function ϕ_1 that makes edges inside the C_i's non-negative in $w^B_{\phi_1}$.
4. Use the updated weights $w^B_{\phi_1}$ to find a price function ϕ_2 that additionally makes the DAG edges of $G \setminus E_{\text{rem}}$ non-negative in $w^B_{\phi_2}$.
5. Use the updated weights $w^B_{\phi_2}$ to find a price function ϕ_3 that additionally makes the edges in E_{rem} non-negative in $w^B_{\phi_3}$.
6. Return $w^B_{\phi_3}$ as a weight function that satisfy $w^B_{\phi_3}(e) \geqslant 0$ for all $e \in E$.

Step 1: Base case. For the base case, we simply need to run Lemma 21.14 to find s^*-shortest paths and set

$$\phi(v) = \text{dist}_{G^B}(s^*, v).$$

The correctness follows as before as these distances make w^B non-negative and thus $w_\phi(e) \geqslant -B$ for all $e \in E$. Moreover, the runtime is only $O(m \log n)$ by Lemma 21.14 and the promise that the shortest path between every pair of vertices inside V uses at most one hop. (Technically speaking, we could have just run one iteration of the Bellman-Ford algorithm and solved the problem in $O(m)$ time but the difference is inconsequential).

Step 2: LDD computation. w^{2B} is non-negative and so we can apply Theorem 21.15 to obtain a set E_{rem} of edges such that any SCC C of $G \setminus E_{\text{rem}}$ satisfies:

$$\forall u, v \in C \quad \text{dist}_{G^{2B}}(u, v) \leqslant dB \quad \text{and} \quad \text{dist}_{G^{2B}}(v, u) \leqslant dB, \tag{21.3}$$

and for any edge in G

$$\Pr\left(e \in E_{\text{rem}}\right) = \frac{O(\log^2 n)}{dB} \cdot w^{2B}(e) = \frac{O(\log^2 n)}{\Delta \cdot B} \cdot w^{2B}(e) + n^{-10}. \tag{21.4}$$

This step takes $O(m \log^3 n)$ time.

Step 3: Fixing SCC edges. We do exactly as in Algorithm 21.18 and by Claim 21.19, have that under w^B, the negative hop diameter of every C_i will be $d = \Delta/2$. This means that the recursive call in this step is run with a correct parameter and thus by induction, we will find a price function ϕ_1 that ensures $w^B_{\phi_1}(e) \geqslant 0$ for all e in the SCCs.

Step 4: Fixing DAG edges. This step is as before and can be done in $O(m + n)$ time.

Step 5: Fixing E_{rem} edges. Again, we do exactly as in Algorithm 21.18 and by Claim 21.20, have that under $w^B_{\phi_2}$, the negative hop distance of any vertex in expectation is

$$O(\log^2 n \cdot \frac{h_{s^*}(v)}{d}) = O(\log^2 n) \cdot \frac{\Delta}{\Delta/2} = O(\log^2 n),$$

using the fact that under w^B (by our initial assumption in the recursion), hop diameter of the graph is Δ and since we set $d = \Delta/2$. This means that this step can now be implemented in $O(m \log^3 n)$ expected time using Lemma 21.14.

In conclusion, the algorithm correctly finds a price function ϕ such that w_ϕ^B is non-negative and thus for every $e \in E$, $w_\phi(e) \geqslant -B$ as desired.

For the runtime analysis, the algorithm reduces the value of Δ by a factor of two each time, and we would be calling it with $\Delta = n - 1$ at the beginning since any pair of reachable vertices can have at most $n - 1$ hops between them. This means there are $O(\log n)$ levels of recursion. Each level also takes $O(m \log^3 n)$ expected time at most, leading to a total of $O(m \log^4 n)$ expected time. This concludes the proof of Lemma 21.21 and the entire proof of Theorem 21.4.

Exercises

Exercise 21.1. Claim 21.6 shows that if G has no negative cycle, then *some* price function makes every edge non-negative. This exercise proves the converse, so that price functions exactly characterize Assumption 21.2.

(a) Show that for *every* cycle C of G and *every* price function ϕ, we have $w_\phi(C) = w(C)$. (Compare with the telescoping computation in the proof of Claim 21.5, which was carried out for paths and left a residual $\phi(u) - \phi(v)$; explain why the residual vanishes here.)

(b) Conclude that G admits a price function ϕ with $w_\phi(e) \geqslant 0$ for every $e \in E$ *if and only if* G has no negative cycle.

(c) Given a candidate price function ϕ, how long does it take to check whether $w_\phi(e) \geqslant 0$ for all $e \in E$? Combine your answer with part (b) to argue that the algorithm of Theorem 21.4 can be run on an arbitrary input graph—without assuming Assumption 21.2 in advance—so that it either outputs $\text{dist}_G(s, v)$ for all $v \in V$ or correctly reports that G contains a negative cycle, still in $\widetilde{O}(m \log W)$ time.

Exercise 21.2. In the proof of Theorem 21.16 we first replaced w by $n \cdot w$, and only at the very end replaced w_{ϕ_t} by $w' = w_{\phi_t} + 1$. This exercise is about why the order of these two steps matters, and why the factor n is the right one.

(a) Give a graph G, a source s, a vertex v, and two s-v paths P and Q with $w(P) < w(Q)$, such that adding 1 to the weight of every edge results in $w'(Q) < w'(P)$. Conclude that the "+1" step is not harmless on its own.

(b) Reprove the claim of Theorem 21.16 that $w(P) < w(Q)$ implies $w'(P) < w'(Q)$ once all weights are multiples of n, and pinpoint the exact step at which the bound $|P| \leqslant n - 1$ on the number of edges of a path is used.

(c) Show that a scaling factor of $\Theta(n)$ is necessary: for every integer $c \leqslant n - 3$, construct a graph on n vertices in which multiplying every weight by c and *then* adding 1 to every edge weight changes which of two s-v paths is the shorter one.

Exercise 21.3 (♦). Prove Lemma 21.14, which we stated without proof: there is an algorithm solving SSSP in any graph in $O(m \log n \cdot \overline{b}_s)$ time, where $\overline{b}_s$ is the *average* negative-hop-distance of Definition 21.9. Design an algorithm for this problem now.