CSC 2421H : Graphs, Matrices, and Optimization Lecture * : Dec 2022

Negative Weight Shortest Path

Lecturer: Sushant Sachdeva *Scribe: Yanting Yu, Mohamed Khodeir*

1 Introduction

In the single source standard shortest path problem (SSSP), we are given a graph and a source vertex, and we want to find the shortest path from the source to all other vertices in the graph. This problem can be solved efficiently in $O(n \log n)$ time using Dijkstra's algorithm, but only in the case where $w(u, v) \geq 0$.

Definition 1.1. *Single Source Shortest Path (SSSP)*

- ***Input:*** *Directed Graph $G = (V, E, w)$, and a source vertex $s_{in} \in V$*
- ***Output:*** *A shortest path tree starting from node s_{in}*

In this lecture, we will explore the main ideas behind an algorithm proposed by [BNW22] for finding shortest paths in a graph with negative edge weights. The existence of negative edges precludes the use of a greedy algorithm such as Dijkstra's which assumes that the distance to a vertex cannot be decreased by adding edges. In addition, reachable negative cost cycles can cause the shortest path to a vertex to be arbitrarily low, rendering the concept of a shortest path meaningless. Therefore, a good algorithm for negative weight shortest path should also be able to detect the presence of negative cycles. The well-known Bellman-Ford algorithm is able to solve this problem in $O(nm)$ time, but the algorithm we will discuss in this lecture has a time complexity of $O\big(m \cdot \log^8 n \cdot \log W)\big)$ where W is the weight of the most negative edge in the graph.

2 Simplifying Assumptions

Throughout this lecture, we will make the following assumptions so that we can focus on the most important ideas:

1. G has no negative cycles
2. G has constant out-degree
3. G has $w(u, v) \geq -1$ for all edges $(u, v) \in E$

3 Price Functions

The first idea we need was actually introduced in 1977 by [Joh77], and it gives a way to transform the weights of a graph without changing the shortest paths between any of the vertices.

Definition 3.1. *Price Function Consider a graph $G = (V, E, w)$ and let ϕ be **any** function: $V \to \mathbb{Z}$. Then, we define w_ϕ to be the weight function:*

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

Lemma 3.2. *Consider a graph $G = (V, E, w)$ and price function ϕ. For any path P between s and $t \in V$ we have*

$$w_\phi(P) = w(P) + \phi(s) - \phi(t)$$

Proof. Let P be any path from vertex s to t going through vertices $\{v_1, \dots v_k\}$. Then we have:

$$w_\phi(P) = w_\phi(s, v_1) + \sum_{i=1}^{k-1} w_\phi(v_i, v_{i+1}) + w_\phi(v_k, t)$$

$$= \big(w(s, v_1) + \phi(s) - \phi(v_1)\big) + \left(\sum_{i=1}^{k-1} w(v_i, v_{i+1}) + \phi(v_i) - \phi(v_{i+1})\right) + \big(w(v_k, t) + \phi(v_k) - \phi(t)\big)$$

$$= \left(w(s, v_1) + \sum_{i=1}^{k-1} w(v_i, v_{i+1}) + w(v_k, t)\right) + \left(\phi(s) + \sum_{i=1}^{k} (\phi(v_i) - \phi(v_i)) - \phi(t)\right)$$

$$= w(P) + \big(\phi(s) - \phi(t)\big)$$

□

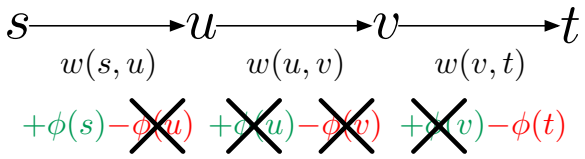

Figure 1: Example s-t path. The prices for all 'transit' vertices cancel out, and we are left with $\phi(s) - \phi(t)$.

Since all paths between any two vertices s and t are shifted up by the same constant $\phi(s) - \phi(t)$, the shortest paths in the graph remain unchanged! This leads very naturally to the question: *can we use price functions to make all the weights nonnegative, and then just use Dijkstra's?*

4 Graphs With Only a Few Negative Edges

The second idea is that if a graph has only a few negative edges on any shortest path between two vertices, then we can efficiently compute a price function that makes all the weights in the graph nonnegative.

Definition 4.1 ($E^{neg}(G)$)**.** *For any weighted graph $G = (V, E, w)$, define*

$$E^{neg}(G) := \{e \in E | w(e) < 0\}$$

Definition 4.2 (G_s)**.** *Given any graph $G = (V, E, w)$,*

$$G_s = (V \cup \{s\}, E \cup \{(s, v)\}_{v \in V}, w_s)$$

refer to the graph G with a dummy source s added, where there is an edge of weight 0 from s to v for every $v \in V$ and edges into s.

Definition 4.3 ($\eta_G(v), \eta(G)$)**.** *For any graph $G = (V, E, w)$, let s be the dummy source in G_s. Define*

$$\eta_G(v) := \begin{cases} \infty & \text{if } dist_{G_s}(s, v) = -\infty \\ min\{|E^{neg}(G) \cap P| : P \text{ is a shortest } sv\text{-path in } G_s\}; & \text{otherwise.} \end{cases}$$

$$\eta(G) = max_{v \in V} \eta_G(v)$$

Definition 4.4 ($dist_i(v)$)**.** *For each $v \in V$ and each integer $i \geq 0$, define $dist_i(v)$ to be the weight of a shortest path from s to v among all s-to-v paths in G containing at most i edges of $E^{neg}(G)$.*

Lemma 4.5 ([Joh77])**.** *Let $G = (V, E)$ be a directed graph with no negative-weight cycle and let s be the dummy source in G_s. Let $\phi(v) = dist_{G_s}(s, v)$ for all $v \in V$. Then, all edge weights in G_ϕ are non-negative.*

Proof. We have the fact that $dist(s, v) \leq dist(s, u) + w(u, v)$. So,

$$w_\phi(u, v) = w(u, v) + \phi(u) - \phi(v) = w(u, v) + \text{dist}_{G_s}(s, u) - \text{dist}_{G_s}(s, v) \geq 0$$

□

Lemma 4.6 (ElimNeg)**.** *There exists an algorithm* ElimNeg(G) *that takes as input a graph $G = (V, E, w)$ in which all vertices have constant out-degree. If G has no negative cycles, the algorithm outputs a price function ϕ such that $w_\phi(e) \geq 0$ for all $e \in E$ and has running time $O(\log n \cdot (n + \sum_{u \in V} \eta_G(v)))$.*

4.1 Algorithm: Dijkstra + Bellman Ford

Algorithm: Algorithm for ElimNeg(G)

```
1  Set d(s) ← 0 and d(v) ← ∞ for v ≠ s
2  Initialize priority queue Q and add s to Q.
3  Initially, every vertex in unmarked.
   // Dijkstra Phase
4  while Q is non-empty do
5      Let v be vertex in Q with minimum d(v)
6      Extract v from Q and mark v
7      foreach edge (v, x) ∈ E\E^neg(G) do
8          if d(v) + w(v, x) < d(x) then
9              add x to Q ;                    // x may already be marked or in Q.
10             d(x) ← d(v) + w(v, x)
   // Bellman-Ford Phase
11 foreach marked vertex v do
12     foreach edge (v, x) ∈ eneg(G) do
13         if d(v) + w(v, x) < d(x) then
14             Add x to Q
15             d(x) ← d(v) + w(v, x)
16     Unmark v
17 If Q is empty: return d(v) for each v ∈ V ;   // d(x) does not change so we get
    final distances.
18 Go to Line 4 ;                                // Q is non-empty.
```

The priority queue Q is implemented as a binary heap, supporting each queue operation in $O(\log n)$ time. In the following, we say that an edge (v, x) of G is active if $d(v) + w(v, x) < d(x)$ and inactive otherwise.

4.2 Correctness

Lemma 4.7. *After any execution of the Dijkstra Phase, all edges of $E\backslash E^{neg}(G)$ are inactive.*

Proof. For iteration i = 0, we run Dijkstra's algorithm on $E\backslash E^{neg}(G)$, so all edges in $E\backslash E^{neg}(G)$ become inactive. For iteration i ¿ 0, we assume that the lemma is true for iteration $i-1$. At the end of the Dijkstra phase of iteration $i-1$, all edges in $E\backslash E^{neg}(G)$ are inactive. The Bellman Ford phase of iteration $i-1$ might decrease some of $d(v)$, which will cause some edges in $E\backslash E^{neg}(G)$ become active. We add these vertex v to Q and execute the Dijkstra phase of iteration i, then all edges in $E\backslash E^{neg}(G)$ will be inactive. □

Lemma 4.8. *After any execution of Bellman-Ford Phase, for any edge $(u, v) \in E^{neg}(G)$, if (u, v) is active then $u \in Q$.*

Proof. We claim that: if any edge $(u, v) \in E^{neg}(G)$ is active, $u \in Q$ or u is marked. According to the algorithm, we know that all vertices will be unmarked after Bellman Ford phase. So the lemma can be proved with this claim.

To prove this claim, we consider the situation where the claim might be false. In line 6, we remove v from Q. But we mark it immediately. In line 10 and 15, $d(x)$ is decreased, so some edges (x, y) will become active. But we have already add x to Q in line 9 and 14. In line 16, we unmark v. But we decrease $d(v)$ for all active edge (u, v), which makes them inactive. Therefore, the claim stands up. □

Corollary 4.9. *If the algorithm terminates, all edges are inactive.*

Lemma 4.10. *If G has no negative cycles then after the Dijkstra Phase in iteration $i \geq 0$, $d(v) < dist_i(s, v)$ (Definition 4.4) for each $v \in V$.*

Proof. For iteration $i = 0$, the lemma follows from Lemma 4.7. For $i > 0$, we assume the lemma holds for iteration $i - 1$. Let P be the shortest path from s to v containing at most i edges of $E^{neg}(G)$. We assume P has exactly i edges of $E^{neg}(G)$. Partition P into maximal subpaths $P_0, e_0, P_1, e_1, ..., e_{i-1}, P_i$, where P_j contains only edges from $E \backslash E^{neg}(G)$ and $e_j \in E^{neg}(G)$. Let s_j be the first vertex and t_j be the last vertex of subpath P_j. Let j be the first iteration where $d(t_i - 1) \leq dist_{i-1}(s, t_{i-1})$ after the Dijkstra Phase, so $j \leq i - 1$. Now we want to prove t_{i-1} is marked after the Dijkstra Phase of iteration j. If the Dijkstra Phase of iteration j reduced $d(t_{i-1})$, t_{i-1} is marked. So we assume the opposite situation. Because of the minimality of j, $d(t_{i-1})$ is reduced and t_{i-1} is added to Q in the Bellman-Ford Phase of $j - 1$. So, t_{i-1} will be extracted from Q and marked in the Dijkstra Phase of iteration j. And the subsequent Bellman-Ford Phase relaxes (t_{i-1}, s_i). Then, we know $d(s_i) \leq dist_i(s, s_i)$ at the beginning of iteration i. By Lemma 4.7, we can get the result $d(v) \leq dist_i(s, v)$ after the Dijkstra Phase in iteration i. □

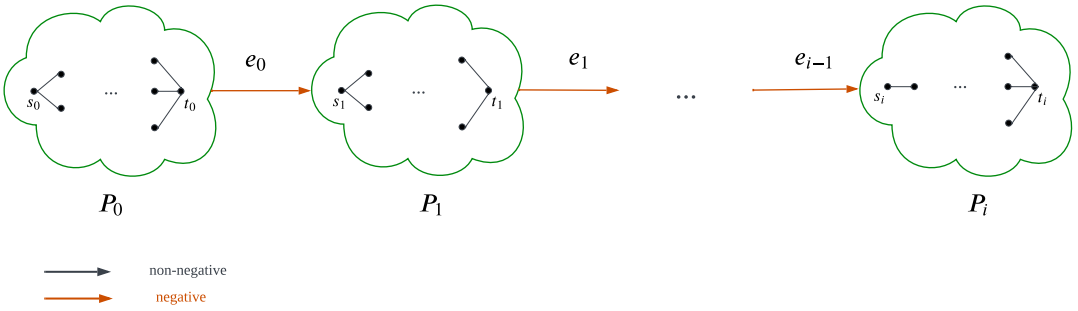

Figure 2: Partition P into maximal subpaths $P_0, e_0, P_1, e_1, ..., e_{i-1}, P_i$, where P_j contains only non-negative edges and e_j are negative.

Corollary 4.11. *If G has no negative cycles, the algorithm terminates with $d(v) = dist(s, v)$ for each $v \in V$.*

4.3 Runtime

By assumption, all vertices in G have constant out-degree. For vertex extracted from Q in Dijkstra Phase and for marked vertex processed in Bellman-Ford Phase, there are only a constant number of outgoing vertex. So, the dominant part of running time is the update of Q.

Each vertex v is added to Q at most once in each of Dijkstra and Bellman Ford phrase per iteration. If $dist(s,v) = dist_i(s,v)$, v is added to Q $O(i+1)$ times and $d(v) = dist(s,v)$ after the Dijkstra Phase of the $(i+1)$th iteration. After that, $d(v)$ will not change and vertex v will not be added to Q. Here we know $dist(s,v) = dist_{\eta_G(v)}(s,v)$, so v is added to Q $O(\eta_G(v)+1)$ times. For all $v \in V$, the overall time added to Q is $O(\sum_{v\in V}(\eta_G(v)+1)) = O(n + \sum_{v\in V}\eta_G(v))$. In addition, the number of extractions from Q cannot exceed the number of insertions. The priority queue Q is implemented as a binary heap, so each queue operation takes $O(\log n)$ time. Therefore, the running time of the algorithm is $O(\log(n)\cdot(n+\sum_{u\in V}\eta_G(v)))$.

5 Low Diameter Decomposition (LDD)

In section 4 we saw how we can efficiently compute price functions if the graph we are given has a constant number of negative edges on any shortest path. But it is not yet clear how to deal with the general case where there might be $O(n)$ negative edges along any shortest path. The third main idea we will discuss in this lecture is a decomposition for directed graphs which partitions the vertices into subsets each of which have low 'weak diameter'.

Definition 5.1. *Weak Diameter*
Given a directed graph $G = (V, E, w)$, we say that a subset $S \subseteq V$ has weak diameter D *iff:*

$$\forall u, v \in S, d_G(u,v) \leq D$$

Definition 5.2. *Low Diameter Decomposition (LDD)*

- ***Input:*** *A graph G with non-negative edge weights, and a* non-negative *integer D*
- ***Output:*** *A set of edges E^{rem} such that the strongly connected components (SCCs) $V_1, \ldots, V_k$ of $G - E^{rem}$ have weak diameter D in G*

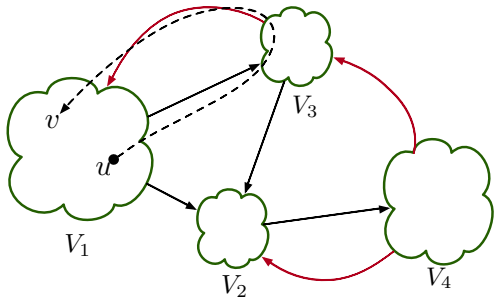

Figure 3: A depiction of the output of LDD. The edges in E^{rem} are depicted in red, and the green bubbles represent the SCCs of $G - E^{rem}$. Each of these SCCs $\{V_1, V_2, V_3, V_4\}$ are guaranteed to have weak diameter D in the original graph.

Lemma 5.3. *There exists an algorithm* LowDiamDecomposition(G, D) *which computes the LDD in $O(m\log^2 n + n\log^3 n)$ time and guarantees that:*

$$\forall e \in E, \Pr[e \in E^{rem}] = O\left(\frac{w(e)\cdot\log^2 n}{D} + n^{-10}\right)$$

We will not prove lemma 5.3 in this lecture. The basic mechanism of LowDiamDecomposition(G, D) is to recursively carve out fixed diameter balls around each vertex until the balls become singletons or have low diameter. In addition to the algorithm and proofs given in the paper, we refer the reader to these additional notes by one of authors [Nan22].

6 Overview of ScaleDown Algorithm

We now have most of the tools we need to describe the main workhorse of the approach, which is a recursive algorithm called ScaleDown that takes a graph with edge weights $w(u, v) \geq -2B$ for some positive integer B and returns a price function ϕ such that $w_\phi(u, v) \geq -B$.

Given a graph G, ScaleDown operates on a slightly modified version of the graph, denoted G^B, in which all the negative weights are shifted up by the constant B. Note G^B may still have negative, so we also define $G^B_{\geq 0}$ in which any weights that remain negative are rounded up to 0.

Definition 6.1. $G^B, G^B_{\geq 0}, G^B_s$
$G^B := (V, E, w^B)$ *where* $w^B(e) := w(e) + B$ *if* $w(e) < 0$
$G^B_{\geq 0} := (V, E, w^B_{\geq 0})$ *where* $w^B_{\geq 0}(e) := \max\{0, w^B(e)\}$

Theorem 6.2 (ScaleDown)**.** *There exists the following algorithm* ScaleDown$(G = (V, E, w), \Delta, B)$.

1. ***Input Requirements:***
   (a) *B is positive integer, and $w(e) \geq -2B$ for all $e \in E$*
   (b) *for every $v \in V$ there is a shortest sv-path in G^B_s with at most Δ negative edges*
2. ***Output:*** *ϕ such that $w_\phi(e) \geq -B$ for all $e \in E$*
3. ***Run Time:*** *expected runtime $O\left(m \log^3(n) \log(\Delta)\right)$.*

Algorithm: ScaleDown$(G = (V, E, w), \Delta, B)$

if $\Delta \leq 2$ **then**
  return ElimNeg(G^B)

// Phase 0: Decompose V to SCCs $V_1, V_2, \ldots$
// with weak diameter $\frac{\Delta B}{2}$ in $G^B_{\geq 0}$
$E^{rem} \leftarrow$ LowDiamDecomposition$(G^B_{\geq 0}, \frac{\Delta B}{2})$

// Phase 1: Make edges inside the SCCs $G^B[V_i]$ non-negative
$\phi_1 \leftarrow$ ScaleDown$(\bigcup_i G[V_i], \Delta/2, B)$

// Phase 2: Make all edges in $G^B \setminus E^{rem}$ non-negative
$\phi_2 \leftarrow \phi_1 +$ FixDAGEdges$(G^B_{\phi_1} \setminus E^{rem}, \{V_1, V_2, \ldots\})$

// Phase 3: Make all edges in G^B non-negative
$\phi_3 \leftarrow \phi_2 +$ ElimNeg$(G^B_{\phi_2})$
return ϕ_3

6.1 Runtime

ScaleDown operates recursively, using Δ as its recursion parameter. When $\Delta \leq 2$, it simply calls ElimNeg, which we already analyzed in section 4. Since $\Delta \leq 2$ this base case is $O(n \cdot \log(n))$.

Before the recursion, we first compute LDD on the graph in $G^B_{\geq 0}$ in $O(m \log^2 n + n \log^3 n)$ time, and after the recursion, we call FixDAGEdges which we will see later takes $O(m+n)$ time.

Before returning, we call ElimNeg to fix the edges in E^{rem}. Since those are the only edges that remain negative after the previous two steps, and because the probability that any edge in the graph is in E^{rem} is bounded (from the low diameter decomposition), we will show that this step has expected runtime $O(m \log^3 m)$.

From the pseudocode, we see that the recursive step halves the value of the Δ at each level (we will see why in the next section), so there can only be $\log(\Delta)$ levels to this recursion.

Putting this all together, we get a total runtime of $O\left(m \log^3(m) \log \Delta\right)$.

7 Phase 1: Pricing Edges Inside SCCs

By the input requirements of ScaleDown, the graph G^B is guaranteed to have at most Δ negative edges along any sv shortest path. We will now show that within each of the SCCs that we obtain from the low diameter composition in Phase 0 of ScaleDown, the value of this parameter is reduced to $\frac{\Delta}{2}$. Note that this is necessary for the correctness of the algorithm, since Phase 1 of ScaleDown is a recursive call that requires this property of its input graph.

Lemma 7.1. *For all $v \in V_i$, we have **at most** $\frac{\Delta}{2}$ **negative edges** on the shortest sv-path in $G[V_i]^B_s$.*

Proof. Consider one of the SCCs, V_i. Let $P(s,v)$ be the shortest sv-path in $G[V_i]^B_s$. Let u be the first vertex along $P(s,v)$. We know that

$$w_B(P(u,v)) \leq 0$$

Let k be the number of negative edges in $P(u,v)$ Then

$$w(P(u,v)) \leq w_B(P(u,v)) - B \cdot k \leq -B \cdot k$$

From LDD, we have $d_{G^B}(u,v) \leq \frac{\Delta B}{2}$

$$d_G(u,v) \leq d_{G^B}(u,v) \leq \frac{\Delta B}{2}$$

Since we know there are no negative cycles, we must have:

$$0 \leq d_G(u,v) + w(P(u,v))$$

$$0 \leq d_G(u,v) + w(P(u,v)) \leq \frac{\Delta B}{2} - B \cdot k$$

$$k \leq \frac{\Delta}{2}$$

□

8 Phase 2: Pricing DAG Edges

All edges in $G^B_{\phi_1}[V_i]$ become non-negative after phase 1, so we turn to the remaining edges in $G^B \backslash E^{rem}$. After contracting every V_i into a node, the resulting graph with remaining edges is a directed acyclic graph (DAG). We introduce Lemma FixDAGEdges, calling it with $G = G^B_{\phi_1} \backslash E^{rem}$ and $P = \{V_1, V_2, \dots\}$. As a result, it returns φ such that $(G^B_{\phi_1} \backslash E^{rem})_\varphi = G^B_{\phi_2} \backslash E^{rem}$ contains no negative-weight edges.

Lemma 8.1. *There exists an algorithm* FixDAGEdges(G, P) *that takes as input a graph G and a partition $P := V_1, V_2, \dots$ of vertices of G such that*

1. *for every i, the induced subgraph $G[V_i]$ contains no negative-weight edges, and*
2. *when we contract every V_i into a node, the resulting graph is a DAG(i.e. contains no cycle).*

The algorithm outputs a price function $\phi: V \to \mathbb{Z}$ such that $w_\phi(u, v) \geq 0$ for all $(u, v) \in E$. The running time is $O(m + n)$.

Algorithm: FixDAGEdges$(G = (V, E, w), P = \{V_1, V_2, \dots\}$

Relabel the sets $V_1, V_2, \dots$ so that they are in topological order in G. That is, after relabeling, if $(u, v) \in E$, with $u \in V_i$ and $v \in V_j$, the $i \leq j$.
Define $\mu_j = min\{w(u, v) | (u, v)\} \in E^{neg}(G), u \notin V_j, v \in V_j$; here, let $min\varnothing = 0$.
`// ` μ_j `is min negative edge weight entering` V_j`, or 0 if no such edge exists.`
Define $M_1 \leftarrow \mu_1 = 0$.
for $j = 2$ *to* q **do** `// make edges into each` $V_2, \dots, V_q$ `non-negative`
  $M_j \leftarrow M_{j-1} + \mu_j$; ; // $M_j = \sum_{k \leq j} \mu_k$
  Define $\phi(v) \leftarrow M_j$ for every $v \in V_j$
return ϕ

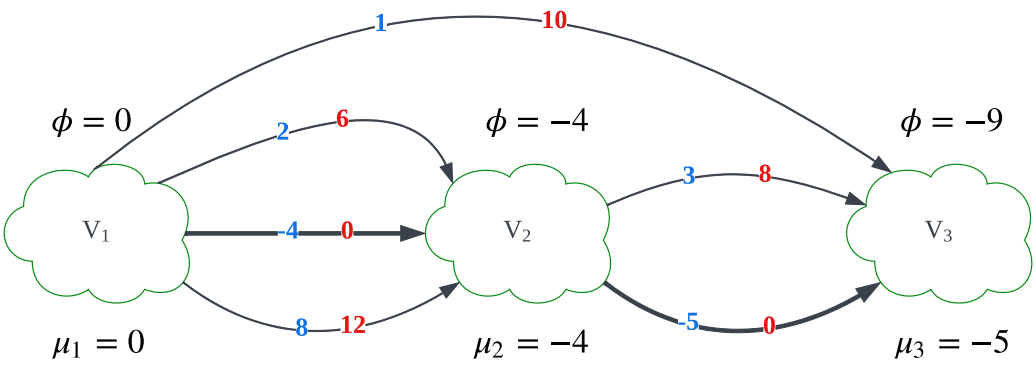

Figure 4: Algorithm FixDAGEdges on an example graph G. After computing μ_j for each component V_j and price function $\phi(v)$ for each $v \in V_j$, all edge weights in $G^B_{\phi_2} \backslash E^{rem}$ become non-negative.

For line 2, each μ_j requires $O(1)$ + [number of edges entering V_j]. Because the V_j are disjoint, the total time to compute all μ_j is $O(m+n)$. For loop in line 4, we only consider each vertex once, so the total time is $O(n)$. The overall running time is $O(m+n)$.

Proof. We need to show $w_\phi(u,v) \geq 0$ for all $(u,v) \in E$. Say that $u \in V_i$ and $v \in V_j$. Because the sets are in topological order, we have $i < j$. By definition of μ_j, we have $\mu_j < w(u,v)$. Therefore, we have

$$w_\phi(u,v) = w(u,v) + \phi(u) - \phi(v) = w(u,v) + M_i - M_j = w(u,v) - \sum_{k=i+1}^{j} \mu_k \geq w(u,v) - \mu_j \geq 0$$

□

9 Phase 3: Pricing Separator Edges

The only negative edges remaining are in E^{rem}. By lemma 4.5, we can set price function $\phi(v) = dist_{G_s}(s,v)$ to make all edge weights non-negative. By lemma 4.6, we can compute all distances in $O(\log n \cdot (n + \sum_{u \in V} \eta_G(v)))$. Since we assume every vertex in G has constant out-degree, $m = \Theta(n)$. So, we can use m and n interchangeably. The running time become $O(\log m \cdot (m + \sum_{u \in V} \eta_G(v)))$.

Now, we want to prove for any $v \in V$, $\mathbb{E}[\eta_{G^B_{\phi_2}}(v)] = O(\log^2 m)$.

Lemma 9.1. *If $\eta(G^B) \leq \Delta$, then for every $v \in V$, $E[|P^B_G(v) \cap E^{rem}|] = O(\log^2 m)$ ($\eta(G^B)$ from Definition 4.3).*

Proof. By decomposition, the path can be divided to negative and non-negative parts:

$$w^B_s(P^B_G(v)) = w^B_{\geq 0}(P^B_G(v)) + w^B_{<0}(P^B_G(v))$$

In G^B_s, the $w^B_s(P^B_G(v))$ as the path length must be less than or equal to the edge weight from s to v. By definition of G_s, the edge weight is 0 from s to every $v \in V$. So,

$$w^B_s(P^B_G(v)) \leq 0$$

Recall the input of ScaleDown (Theorem 6.2) that $w(e) \geq -2B$. By adding B to all negative edges, $w^B(e) \geq -B$ for all $e \in E$. So,

$$\begin{aligned}
w^B_{<0}(P_{G^B}(v)) &= \sum_{e \in P_{G^B}(v) \cap E^{neg}(G^B)} w^B(e) \\
&\geq |P_{G^B}(v) \cap E^{neg}(G^B)| \cdot (-B) \\
&= \eta_{G^B}(v) \cdot (-B) \\
w^B_{\geq 0}(P_{G^B}(v)) &= w^B_s(P_{G^B}(v)) - w^B_{<0}(P_{G^B}(v)) \\
&\leq \eta_{G^B}(v) \cdot B
\end{aligned}$$

Recall the output of LowDiamDecomposition (Lemma 5.3) that $\Pr[e \in E^{rem}] = O(\frac{w^B_{\geq 0}(e) \cdot \log^2 n}{D} + n^{-10})$, and we have $D = dB = B\frac{\Delta}{2}$. So,

$$\mathbb{E}[P_{G^B}(v) \cap E^{rem}] = O(\frac{w^B_{\geq 0}(P^B_G(v)) \cdot \log^2 n}{B\frac{\Delta}{2}} + |P_{G^B}(v)| \cdot n^{-10})$$

$$= O(\frac{2\eta_{G^B}(v) \cdot \log^2 n}{\Delta} + n^{-9})$$

If $\eta(G^B) \leq \Delta$,

$$\mathbb{E}[P_{G^B}(v) \cap E^{rem}] = O(\log^2 n)$$

□

Now we are ready to compute the expected runtime of ElimNeg($G^B_{\phi_2}$) in Phase 3. Note that, regardless of the value of ϕ_2, $P_{G^B}(v)$ is a shortest sv-path in $G^B_{\phi_2}$ for any $v \in V$. By Definition 4.3,

$$\begin{aligned}\eta_{G^B_{\phi_2}} &= min\{|P \cap E^{neg}(G^B_{\phi_2}| : \text{P is a shortest sv-path in } (G^B_{\phi_2})_s\} \\ &\leq |P_{G^B(v)} \cap E^{neg}(G^B_{\phi_2}|\end{aligned}$$

The only negative edges remaining are in E^{rem}, which means $E^{neg}(G^B_{\phi_2}) \subseteq E^{rem}$. So,

$$\eta_{G^B_{\phi_2}}(v) \leq |P_{G^B(v)} \cap E^{rem}|$$

By Lemma 9.1 and the input of Theorem 6.2 that $\eta(G^B) \leq \Delta$,

$$\mathbb{E}[\eta_{G^B_{\phi_2}}(v)] \leq \mathbb{E}[|P_{G^B(v)} \cap E^{rem}|] = O(\log^2 m)$$

Therefore, the expected runtime of ElimNeg($G^B_{\phi_2}$) in Phase 3 is

$$O\left(\log m \cdot \left(m + \mathbb{E}\left[\sum_{u \in V} \eta_G(v)\right]\right)\right) = O(m \log^3 m)$$

10 Putting it all Together

Now that we have our ScaleDown operation which can take a graph with $w(e) \geq -2B$ and returns a price function such that $w_\phi \geq B$, it only remains to show how this can be used to solve the shortest path problem. The idea is to scale all the weights of the input graph up by $2n$, then use ScaleDown $\log 2n$ times to compute a price function which brings all the weights back up to $w(e) \geq -1$. Then we simply add 1 to all weights and call Dijkstras on the resulting graph $G^\star$ which will have strictly nonnegative weights.

Algorithm: SPmain(G_{in}, s_{in})

```
   // scale up edge weights
1  w̄(e) ← w_in(e) · 2n
2  B ← 2n
3  for i = 1 to t := log_2(B) do
4  |   ψ_i ← ScaleDown(Ḡ_{φ_{i-1}}, Δ := n, B/2^i)
5  |   φ_i ← φ_{i-1} + ψ_i
   |   //  w_{φ_i}(e) ≥ −B/2^i
6  w*(e) ← w̄_{φ_t}(e) + 1
   // Observe:  G* in above line has non-negative weights
7  T ← Dijkstra(G*, s_in)
8  return shortest path tree T .
```

10.1 Correctness

It should be clear that scaling the weights up by $2n$ from $w(e)$ to $\bar{w}(e)$ will not change the shortest path. And we have seen that modifying the weights using price functions also preserves shortest paths. Since $\bar{w}_{\phi_t}(e)$ differs from $\bar{w}(e)$ only via price functions, it only remains to show that adding 1 to $\bar{w}_{\phi_t}(e)$ to obtain $w^*(e)$ will not change the shortest path.

To see why this is the case, consider any two vertices $s, t \in V$. As we have seen in 3 the cost of all s-t paths under $\bar{w}_{\phi_t}$ are shifted by the same constant $\phi(s) - \phi(t)$. Therefore, the difference in cost between any s-t paths is the same under $\bar{w}_{\phi_t}$ and $\bar{w}$. Since we scaled all edge weights up by $2n$ to obtain $\bar{w}$, no two s-t paths can differ by less than $2n$. But, adding 1 to all edges can only increase the cost of any path by $(n-1)$ since there are at most $(n-1)$ edges in a path with no cycles.

10.2 Runtime

We saw in section 6.1 that the running time of a single call to ScaleDown will be $O\left(m \log^3(m) \log \Delta\right)$.

In algorithm 4, there are $O(\log n)$ calls to ScaleDown with $\Delta = n$ so the total runtime is $O\left(m \log^5(m)\right)$.

References

[BNW22] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. "Negative-Weight Single-Source Shortest Paths in Almost-linear Time". In: *arXiv preprint arXiv:2203.03456* (2022) (cit. on p. 1).

[Joh77] Donald B. Johnson. "Efficient Algorithms for Shortest Paths in Sparse Networks". In: *J. ACM* 24.1 (1977), pp. 1–13. DOI: 10.1145/321992.321993 (cit. on pp. 1, 3).

[Nan22] Danupon Nanongkai. *Notes on Low Diameter Decomposition*. Available at https://hackmd.io/@U0nm1XUhREKPYLt1eUmE6g/Sycpovkiq, version 1.6.0. 2022 (cit. on p. 7).