CS 270: Combinatorial Algorithms and Data Structures Spring 2023

Lecture 2 — January 19, 2023

Prof. Jelani Nelson *Scribe: Andrew Lin, Sanjay Gollapudi*

1 Single-Source Shortest Paths

We continue our discussion of the Bernstein-Nanongkai-WulffNilsen [1] algorithm for single-source shortest paths with negative edge weights from last lecture. In today's lecture, we describe three of the subroutines of the BNW algorithm. We begin with some definitions:

Definition 1.1.

$$w^B(e) = \begin{cases} w(e), & w(e) \geq 0. \\ w(e) + B, & w(e) < 0. \end{cases}$$

Definition 1.2. For vertex v, $\eta(v)$ is the minimum number of negative edges of all shortest paths from source s to v. $P_{G^B}(v)$ denotes the path achieving the value of $\eta(v)$.

Recall our subroutines:

1. LowDiamDecomp(G, D), which we abbreviate as LDD(G, D). This subroutine assumes that for all edges $e, w(e) \geq 0$ and returns a subset of edges E' such that
   - each SCC in $G \backslash E'$ has a weak diameter $\leq D$
   - For all edges e, $\mathbb{P}(e \in E') \leq O(\frac{w(e) \log^2 n}{D} + n^{-10})$
2. ElimNeg(G), which finds a good ϕ in time $O((\log n)(\Sigma_v[1 + \eta_G(v)]))$.
3. FixDAGEdges$(G, \{V_i\}_i)$ which assumes edges inside each V_i are non-negative and the graph between the V_i's is a DAG. This finds a ϕ making edges between any two V_i's non-negative.
4. ScaleDown(G, Δ, B), which assumes $\eta(G^B) \leq \Delta$. Given a graph with $w(e) \geq -2B$ for all edges e, this subroutine returns ϕ such that $w_\phi(e) \geq -B$ for all edges e.

1.1 ScaleDown subroutine

We use LowDiamDecomp, ElimNeg, and FixDAGEdges to create the ScaleDown subroutine.

Algorithm 1.3. ScaleDown(G, δ, B):

1. if $\Delta \leq 2$ set $\phi_2 \leftarrow 0$, go to Phase 3
2. $d \leftarrow \frac{\Delta}{2}$
3. For all $e \in E, w^B_{\geq 0} \leftarrow \max\{0, w^B(e)\}$
4. **Phase 0**

5. $E' \leftarrow LDD(G^B_{\geq 0}, dB)$
6. **Phase 1**
7. $H \leftarrow \bigcup_i G[V_i]$, where $V_i's$ are SCCs of $G \backslash E'$
8. $\phi_1 \leftarrow \text{ScaleDown}(H, \frac{\Delta}{2}, B)$
9. **Phase 2**
10. $\psi \leftarrow \text{FixDAGEdges}(G^B_{\phi_1} \backslash E')$
11. $\phi_2 \leftarrow \phi_1 + \psi$
12. **Phase 3**
13. $\psi' \leftarrow \text{ElimNeg}(G^B_{\phi_2})$
14. return $\phi_2 + \psi'$

.

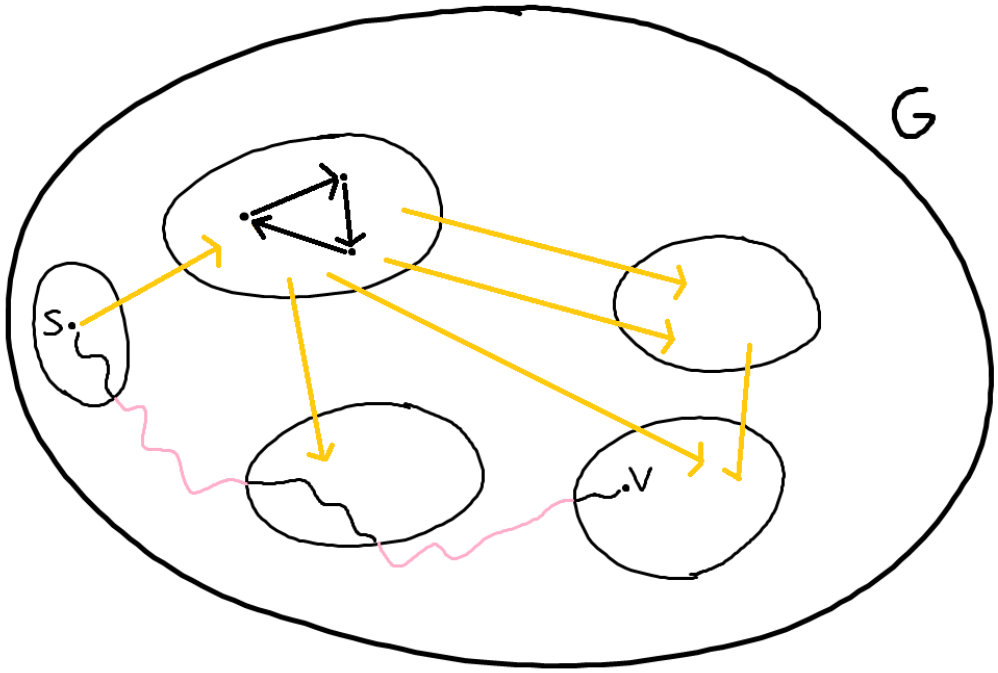

Figure 1: Example graph G.

The idea here is that after blowing up all edge weights by a factor of $B = 2n$, we can repeatedly call ScaleDown $\log B$ times to ensure all weights are at least -1, and then we add 1 to each edge to make all edges positive. Since each path has length at most n and all edge weights are integers, any two vertices on the scaled-up graph with nonequal distance differ by at least $2n$, but adding 1 to each vertex can only increase the distance between two vertices by at most n. Therefore we

preserve the ordering of any two paths, so we can run Dijkstra on the graph with the resulting edge weights.

ScaleDown runs in $\tilde{O}(m \log \Delta)$ time. It consists of 4 phases, where we generate a set E' and repeatedly "fix" edges by making their weights at least $-B$, thus ensuring $w^B(e) \geq 0$ for all edges e.

1. **Phase 0** calls LowDiamDecomp on the graph. LowDiamDecomp generates a set E' of vertices, and guarantees that for any vertex v, the expected number of negative edges in a shortest path to v with the fewest number of negative edges contains at small ($O(\log^2 n)$) number of vertices in E'.

2. **Phase 1** fixes all edges within a SCC in $G \backslash E'$ (the black edges in Figure 1). The recursive call to the algorithm requires that $\eta(H^B) \leq \frac{\Delta}{2}$.

3. **Phase 2** fixes all edges that connect two SCCs (the yellow edges in Figure 1).

4. **Phase 3** fixes all edges in E' (the pink edges in Figure 1). Since shortest paths contain only a small number of these edges, this step is efficient.

1.2 FixDAGEdges subroutine

Now we will describe the FixDAGEdges subroutine. The requirement that edges within each of the SCCs $\{V_i\}$ is nonnegative is satisfied due to recursively calling ScaleDown on the graph of SCCs with parameter B returning a ϕ with $w_\phi(e) \geq -B$, for all edges e, so when we consider w_ϕ^B, all weights will be positive.

Algorithm 1.4. FixDAGEdges:

First set $\phi(x) = 0$ for all vertices x in the source SCC. For a child SCC, note that for all vertices u, $w_\phi(v, u) = w(v, u) + \phi(u) - \phi(v)$ where (v, u) is an edge from v in the parent SCC. We want $w_\phi(v, u) \geq 0$, which is equivalent to $\phi(u) \leq w(v, u) + \phi(v)$. We find the maximum t such that $t \leq w(v, z) + \phi(z)$ for all z in the child SCC, and set $\phi(z) = t$ for all such z. Thus all edges between the parent and child have positive weight.

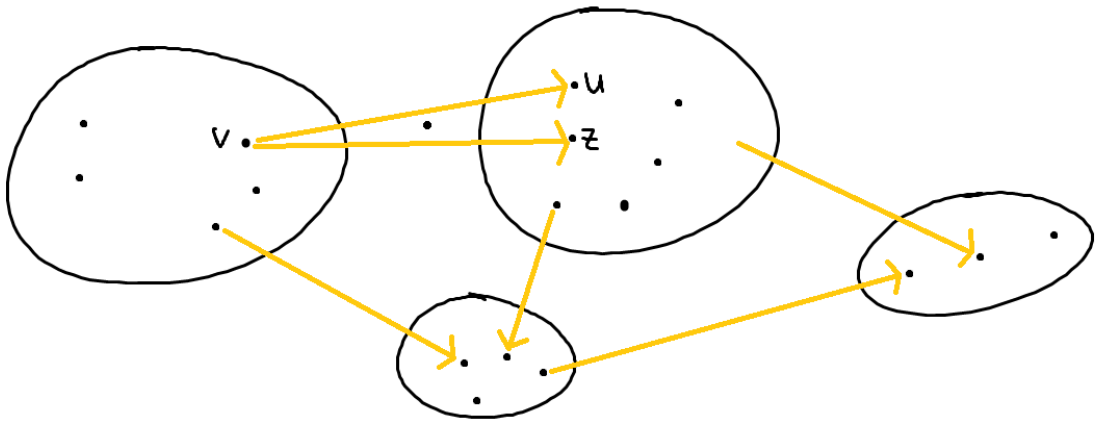

Figure 2: Example graph $G_{\phi_1}^B$. FixDAGEdges picks prices making all yellow edges nonnegative.

We simply traverse the SCCs in topological order to assign a valid price to each vertex using the above process, so this can be done in $O(m+n)$ time.

1.3 ElimNeg subroutine

Now we will present a weaker version of the the ElimNeg subroutine, which runs in $O((m+n)\eta(G)\log n)$ time. This algorithm uses dynamic programming.

Algorithm 1.5. ElimNeg(G):

- $h(v,k)$:= shortest path length to v using $\leq k$ negative edges such that the last edge is negative
- $g(v,k)$:= shortest path length to v using $\leq k$ negative edges

Recursively, we compute h and g as follows:

$$h(v,k) = \begin{cases} \infty, & k=0 \\ \min\{h(v,k-1), \min_{u,(u,v)\in E, w((u,v))<0}\{g(u,k-1)+w(u,v)\}\} & \text{otherwise.} \end{cases}$$

$$g(v,k) = \begin{cases} d^+(s,v), & k=0 \\ \min\{g(v,k-1), \min_{u\in V}\{h(u,k)+d^+(u,v)\}\} & \text{otherwise} \end{cases}$$

We use dynamic programming to compute $g(v,k)$ for $k=\eta(G)$.
To show the runtime of this algorithm, first note that $d^+(s,v)$ can be computed using Dijkstra in $O(m+n)$ time. Calculating g from h is more complicated: rather than calculating all values of h directly and minimizing over them, which takes $O(mn)$ time, we instead calculate $h(u,k)+d^+(u,v)$.

To do so, at each step, we create a dummy vertex α, and for all $1 \leq i \leq n$, we add edge (α, v_i) with weight $h(v_i,k)$. Remove all negative edges in the rest of the graph. Note that $h(u,k)+d^+(u,v)$ is the shortest path from α to v with at most k negative edges that passes through v, so $\min_{u\in V}\{h(u,k)+d^+(u,v)\}$ is the shortest path from α to v.

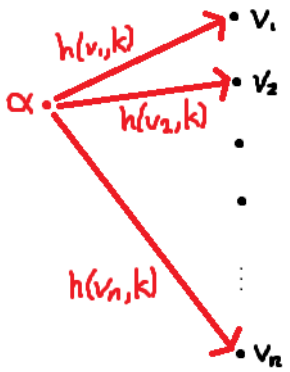

Figure 3: The graph created to compute $h(u,k)+d^+(u,v)$.

We use the following result without proof:

Claim 1.6. Dijkstra can compute the shortest paths from α to all vertices v even if some of the edge weights out of α are negative.

Therefore we can compute the values of $g(v,k)$ given values of g and h for $k-1$ in $O((m+n)\log n)$ time. Since we want to calculate $g(v,\eta(G))$, we need to run this $\eta(G)$ times, so this subroutine takes $O((m+n)\eta(G)\log n)$ in total.

1.4 Remaining details from ScaleDown

Now we will prove the claim from **Phase 0** in ScaleDown.

Claim 1.7. For all vertices $v, \mathbb{E}|P_{G^B}(v) \cap E'| = O(\log^2 n)$

Proof. Note that since we add at most B to each negative edge, we have

$$w^B_{\geq 0}(P_{g^B}(v)) \leq w^B(P_{G^B}(v)) + B \cdot |P_{G^B}(v) \cap E^{\text{neg}}|.$$

By the existence of our dummy node s, $w^B(P_{G^B}(v)) \leq 0$ and we know $|P_{G^B}(v) \cap E^{\text{neg}}| \leq \eta_{G^B}(v)$. Therefore, $w^B_{\geq 0}(P_{g^B}(v)) \leq \eta_{G^B}(v) \cdot B$. From this we use the fact that LDD guarantees $\mathbb{P}(e \in E') \leq O(\frac{w(e)\log^2 n}{D} + n^{-10})$ to get

$$\begin{aligned}
\mathbb{E}|P_{G^B}(v) \cap E'| &= \mathbb{E}\left|\sum_{e \in P_{G^B}(v)} \mathbf{1}\{e \in E'\}\right| && (1)\\
&= \sum_{e \in E} \mathbb{P}(e \in E') && (2)\\
&\leq O(\frac{w^B_{\geq 0}(P_{G^B}(v))\log^2 n}{D} + n^{-9}) && (3)\\
&= O(\frac{\eta_{G^B}(v) B \log^2 n}{D} + n^{-9}) && (4)\\
&= O(\frac{\Delta B \log^2 n}{dB} + n^{-9}) && (5)\\
&= O(\frac{\Delta \log^2 n}{\frac{\Delta}{2}} + n^{-9}) && (6)\\
&= O(2\log^2 n + n^{-9}) && (7)\\
&= O(\log^2 n) && (8)
\end{aligned}$$

□

Claim 1.8. If G contains no negative cycles, then $\eta(H^B) \leq \frac{\Delta}{2}$.

Proof. Take an arbitrary vertex v and look at the path $P_{H^B}(v)$ from s to v. Let u be the vertex on $P_{H^B}(v)$ immediately after s (which may equal v), and let P be subpath from u onward to v (simply remove s and the edge (s,u)). Then

$$\begin{aligned}
\underset{G}{dist}(u,v) &\leq w_H(P)\\
&\leq w_{H^B}(P) - |E^{neg}(H^B) \cap P| \cdot B\\
&= w_{H^B}(P) - \eta_{H^B}(v) \cdot B && (9)\\
&\leq -\eta_{H^B}(v) \cdot B && (10)
\end{aligned}$$

where Eq. (9) uses that $w(s,u) = 0$, and Eq. (10) uses that there is a path from s to v of length 0 (namely the single edge) and thus the shortest path can be no longer.

Now, by guarantees of LowDiamDecomp on weak diameter, we know $dist_G(v,u) \leq dB = \Delta B/2$. Since G contains no negative cycles, combining this with $dist_G(u,v) \leq -\eta_{H^B}(v) \cdot B$ then implies $\Delta B/2 - \eta_{H^B}(v) \cdot B \geq 0$. Rearranging gives $\eta_{H^B}(v) \leq \Delta/2$. The claim holds since v was arbitrary. □

2 Max Flow

We now consider the MaxFlow problem.

2.1 Basics

We are given the following:

- A capacitated graph $G = (V, E)$, where each edge $e \in E$ has capacity $u_e \in \{1, 2, \cdots, U\}$.
- A source node $s \in V$
- A sink node $t \in V$

We wish to find a feasible flow $f \in \mathbb{R}^m$ from s to t, maximizing $\text{val}(f) = \sum_{e=(s,\cdot)} f_e$, subject to the following constraints:

- For all $e \in E, f_e \leq u_e$
- For all $e, f_e \geq 0$
- For all $v \notin \{s,t\}, \sum_{e=(v,\cdot)} f_e = \sum_{e=(\cdot,v)} f_e$

Intuitively, this means that we have positive flow through each edge, the flow through each edge is bounded by its capacity, and the flow into a vertex is equal to the flow out.

2.2 Ford-Fulkerson with scaling

Recall that the Ford-Fulkerson algorithm [2] has a runtime $O(mf^*)$. Given a flow f, the residual graph G_f has capacities $u_e - f_e$. Since the source can have $O(n)$ neighbors, and each can have capacity up to U, in the worst case we have $f^* = O(nU)$ so Ford-Fulkerson takes $O(mnU)$ time in the worst-case scenario.

We will show that Max Flow can be solved in $O(m^2 \log U)$ time. Note that all capacities are integers and can be represented with length $\log U$ binary strings. We write the binary string representations of capacities as rows in a grid, and append a column of zeros to teh left. We iterate from left to right, and at each step, we first double the capacity of each edge and double the flow through that edge, resulting in a max flow on the new capacitated graph. We then add one unit of capacity to any edge with a 1 in the current column. Note that the flow we had previously is no longer a max flow for the current graph.

Recall that the max flow problem is equivalent to the min cut problem, so the original flow was a fully-saturated min cut. Since we added at most m units of capacity at the most recent step, there exists a cut of capacity at most m on the residual graph, so the max flow through the residual graph is at most m. Therefore running Ford-Fulkerson takes $O(m^2)$ time. There are $\log U$ columns since we have length $\log U$ bitstrings so this takes $O(m^2 \log U)$ time to find a max flow for the original graph.

References

[1] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. Negative-weight single-source shortest paths in near-linear time. *Proceedings of the 63rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 600–611, 2022.

[2] Lester R. Ford and Delbert R. Fulkerson. Maximal flow through a network.. . *Canadian Journal of Mathematics*, 8(3):399404, 1956