CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Lecture 22

November 25, 2019

Instructor: Sepehr Assadi

Single-Source Shortest Path

Recall the (single-source) shortest path problem we defined in the previous lecture.

Problem 1. Given a graph $G(V, E)$ (directed or undirected) with *positive* weight $w_e > 0$ on each edge $e \in E$, and a designated vertex $s \in V$, called a *source*, find the value of the shortest path from s to every vertex $v \in V$, denoted by $dist(s, v)$. In other words, for any path P, we define $w(P) = \sum_{e \in P} w_e$, namely, the total weight of the edges in this path, and our goal is to find the weight of the minimum weight path (minimizing $w(P)$) from s to any other vertex v.

1 Algorithms for Single-Source Shortest Path

We considered one simple algorithm for the shortest path problem, the Bellman-Ford algorithm. We now describe another (and more efficient) algorithm for this problem called Dijkstra's algorithm.

1.1 Dijkstra's Algorithm

We now consider another algorithm for the shortest path problem, namely the Dijkstra's algorithm. The algorithm is quite similar to Prim's algorithm with a minor yet crucial modification.

1. Let $mark[1:n] = FALSE$ and s be the designated source vertex.
2. Let $d[1:n] = +\infty$ and set $d[s] = 0$.
3. Set $mark[s] = TRUE$ and let S (initially) be the set of edges incident on s and assign a value $value(e) = d[s] + w_e$ to each of these edges.
4. While S is non-empty:
   (a) Let $e = (u, v)$ be the *minimum value* edge in S and remove e from S.
   (b) If $mark[v] = TRUE$ ignore this edge and go to the next iteration of the while-loop.
   (c) Otherwise, set $mark[v] = TRUE$, $d[v] = value(e)$, and insert all edges e' incident on v to S with $value(e') = d[v] + w'_e$.
5. Return d.

It is worth mentioning that the "only difference" between Dijkstra's and Prim's algorithms is in the notion of $value(e)$: in Dijkstra's we set $value(e) = d[v] + w_e$ where v is the endpoint of edge e we already marked while in Prim's algorithm we set $value(e) = w_e$. This has the following effect: Even though both algorithms attempt to "grow" a component of vertices reachable from s, Dijkstra do this by exploring edges that are "closer" to s and adding their endpoints to the component of s while Prim finds the edges that are simply the "lightest" at the moment. See Figure 1 for an illustration.

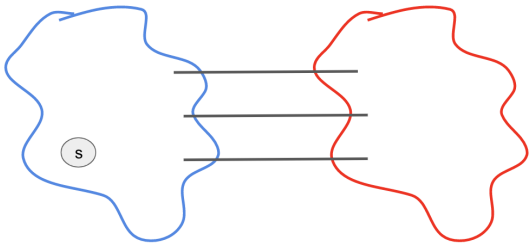

Figure 1: Both Prim's and Dijkstra's algorithms expand a set of vertices reachable from a source further and further by picking the "best" (cut) edge in the cut specified by the vertices currently reachable from source. However, the definition of the "best" edge is different between the two algorithms: Prim's algorithm picks the edge with minimum weight among the cut edges of this cut while Dijkstra's algorithm picks the the edge whose other endpoint would be closest to the source. This is the reason why Prim's algorithm returns an MST while Dijkstra's algorithm returns shortest paths.

Proof of Correctness: The proof of correctness of Dijkstra's algorithm is quite similar to that of BFS and Prim's algorithm. We prove by induction that in every iteration of the while-loop in the algorithm, the set C of all marked vertices (i.e., $C = \{v \mid mark[v] = TRUE\}$) has the following two properties:

- For any $u \in C$ and $v \in V - C$, $dist(s,u) < dist(s,v)$;
- For every $v \in C$, $d[v] = dist(s,v)$, i.e., distances are computed correctly.

The base case of the induction, namely for iteration 0 of the while-loop (i.e., before we even start the while-loop) is true since s is the closest vertex to s (satisfying part one) and $d[s] = dist(s,s) = 0$ satisfying part two. Now suppose this is true for some iteration i of the while-loop and we prove it for iteration $i+1$. Let $e = (u,v)$ be the edge removed from S in this iteration. If $mark[v] = TRUE$ we simply ignore this edge and hence the set C remains the same after this step and by induction hypothesis, we have the above two properties. Now suppose $mark[v] = FALSE$. In this case:

1. Every edge among the cut edges of $(C, V-C)$ belong to the set S at this point (we have not removed any of those cut edges yet as otherwise C would have contained the other endpoint of that edge as well and we included all those edges when we marked their first endpoint and hence added them to C).
2. We are setting $d[v] = value(e) = d[u] + w_e = dist(s,u) + w_e$ (by the induction hypothesis for u) where e is the edge with minimum $value(e)$ in S and hence minimum $value(e)$ among cut edges of $(C, V-C)$.
3. Since $dist(s,w) > dist(s,u)$ for all $w \in V - C$ by induction hypothesis, we know that the shortest path from s to v does not visit any of the vertices in $V - C$ (otherwise the edge leading to that vertex will have a smaller value than the edge e leading to v). Hence, by setting $d[v] = dist(s,u) + w_e$ where edge e minimizes the right hand side of this equation, we will have that $d[v] = dist(s,v)$. This proves the second part of the induction hypothesis. For the first part also, notice that $value(e)$ is minimized and hence v is the "closest" vertex to s in $V - C$ and hence after adding v to C, $dist(s,w) > dist(s,v)$ for all $w \in V - C$. This proves the second part of the induction hypothesis.

This concludes the proof of correctness of Dijkstra's algorithm as at the end of the last iteration, by induction hypothesis (second part), $d[v] = dist(s,v)$ for all $v \in V$.

Runtime Analysis: Similar to the Prim's algorithm, we can implement the set S with a *min-heap*: this allows us to implement Dijkstra's algorithm in $O(n + m \log m)$ time as well[1].

[1]There is a way to implement Dijkstra's and Prim's algorithms to run in $O(n \log n + m)$ instead of $O(n + m \log m)$ by using *Fibonacci heaps* instead of min-heaps. Fibonacci heap is a seriously complicated data structure and for the purpose of this

2 Network Flow

We now study our final graph algorithms problem, the *network flow* problem (or the *maximum flow* problem). Generally speaking, the network flow problem capture the following scenario: we have a directed graph $G(V, E)$, source vertex s, and a sink vertex t. We shall think of the source as producer of a certain material, or a "flow", at some fixed rate, and the sink as the consumer of this material. The edges of this directed graph are able to "transport" this flow subject to some given "capacity" on each edge. The question is then what is the "maximum rate" we can transport the materials from source to sink?

2.1 (Flow) Networks and Flows

Let us now define the network flow problem formally. A *network* (sometimes called a flow network) for our purpose is any *directed* graph $G(V, E)$ with *capacity* c_e over each edge $e \in E$, and two designated vertices $s, t \in V$ where s is called a *source* vertex (and has no incoming edges) and t is called a *sink* vertex (and has no outgoing edges). Figure 2 gives an example of a network.

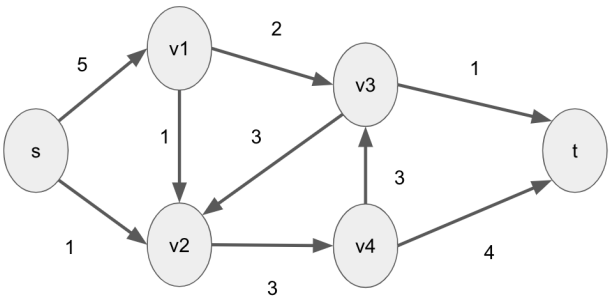

Figure 2: A flow network. Numbers next to each edge shows the capacity of the edge.

We define a *flow* in a given network $G(V, E)$ as any function $f : V \times V \to \mathbb{R}^+$ that satisfies the following constraints:

- **Capacity constraints:** For any edge $e = (u, v) \in E$, $f(u, v) \leq c_e$ and if there is no edge from u to v, then $f(u, v) = 0$ (a flow over each edge cannot be larger than the capacity of the edge).
- **Preservation of flow:** For any vertex $v \in V - \{s, t\}$, $\sum_u f(u, v) = \sum_w f(v, w)$ (the flow into a vertex other than s, t is equal to the flow out of that vertex).

For any flow f in a network G, we define the *value* or *size* of the flow f to be $|f| = \sum_v f(s, v)$, i.e., the amount of flow produced by s (note that this is also equal to the amount of flow consumed by sink, i.e., $|f| = \sum_v f(v, t)$; why?). Figure 3 gives an example of a flow in the network of Figure 2.

We can now define the network flow (or maximum flow) problem as follows.

Problem 2 (Network Flow/Maximum Flow). Given a network $G(V, E)$ with source s and sink t and capacity c_e on every edge $e \in E$, find a flow function f with maximum value, namely a *maximum flow*.

2.2 Algorithms for Maximum Flow

There is a long list of algorithms designed over the years for solving the maximum flow problem. In fact, this is still a highly active area of research and it seems that we are still far from finding the "best" (most efficient) algorithm for the maximum flow problem[2].

course, you can simply ignore this improvement.

[2]This is a very different scenario than shortest path and MST problems where we already know (for the most part) essentially the asymptotically best possible algorithms for those problems.

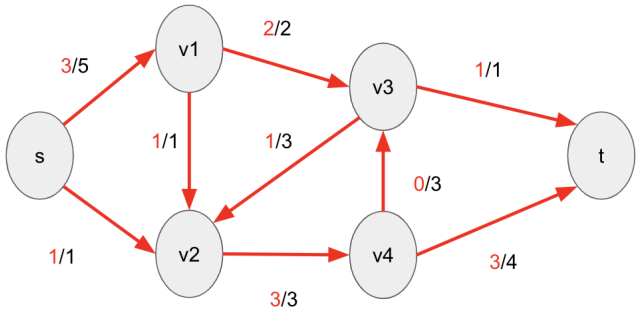

Figure 3: A flow function in the network given in Figure 2. Numbers next to each edge shows the value of flow function over each edge compared to the capacity of the edge. This flow is also a maximum flow.

One of the earliest (perhaps the earliest) algorithms for this problem is the Ford-Fulkerson algorithm which finds a maximum flow in a network with n vertices and m edges in $O(m \cdot F)$ time where F denotes the value of maximum flow itself. Another algorithm for the maximum flow problem is the Edmond-Karp algorithm that runs in $O(m^2 \cdot n)$ time. Both of these algorithms are covered in your textbook (both CLRS and Erickson's book) and you are strongly encouraged to at least briefly take a look at these algorithms as they are not that complicated (albeit still considerably more involved than the previous algorithms we saw in this course). However, the more recent algorithms for this problem with improved running time are getting much more complicated and are entirely beyond the scope of this course.

Even though Ford-Fulkerson and Edmond-Karp algorithms are not that complicated (and are sometimes taught in undergraduate algorithm courses), in this course, we will *not* go over these algorithms. Instead, we use algorithms for maximum flow in a *black-box* manner using graph reductions to solve other problems. As such, for the purpose of this course, we only need the following statement:

> **Ford-Fulkerson Algorithm:** There is an algorithm for the maximum flow problem that runs in $O(m \cdot F)$ time where F is the value of maximum flow in the network.

2.3 Some Applications of Network Flow

In the next lecture, we will go over several applications of network flow such as *bipartite matching*, *finding edge/vertex disjoint paths*, or *exam scheduling* problems.