

Lecture 6

Lecturer: Andre Wibisono

Scribe: Michał Gerasimiuk

1 Properties of BFS

In Lecture 5, we used **Breadth-First Search** to find shortest paths from some vertex in an **unweighted** graph. Recall that BFS has the following properties:

1. **Correctness.** The distances returned by BFS are the real shortest path distances between the start s and any vertex v . We have $\mathbf{dist}[v] = d(s, v)$ for all $v \in V$, for any starting vertex s .
2. **Running time in $O(|V| + |E|)$.** The $O(|V|)$ factor comes from enqueueing/dequeueing each vertex exactly once. We get the $O(|E|)$ term from traversing the whole adjacency list. This gives us the sum $\sum_{v \in V} |\mathbf{Adj}[v]| = 2|E|$ by the handshaking lemma.

2 Shortest paths on weighted graphs

However, in the more general case of a **weighted graph**, we cannot directly use BFS to find shortest paths. We recall the following definition.

Definition 1 (Weighted graph). A **weighted graph** $G = (V, E)$ has a **weight** (or **length**) $l_e > 0$ associated with each edge $e \in E$.

Unweighted graphs are the case when all the edge weights are equal: $l_e = 1$.

In a weighted graph, the length of a path is no longer the number of edges that form it. Instead, we have:

Definition 2 (Path length on a weighted graph). For any path $P = (e_1, e_2, \dots, e_k)$, the **length** $l(P)$ is given by

$$l(P) = \sum_{e \in P} l_e.$$

The distance between any two vertices $s, t \in V$ is still the length of the shortest path between them. The only difference is that path lengths are now defined as above.

Definition 3 (Distance on a weighted graph). The distance between s and t is $d(s, t) = \min_{P: s \rightarrow t} l(P)$.

The minimization is over all paths connecting s to t .

3 Naive solution

If we have the constraint that all l_e are positive integers, we may try to transform the graph into an unweighted one. Then, we would be able to get shortest paths by running BFS.

If $G = (V, E)$ is a weighted graph, then we can replace every edge e that has weight l_e with l_e edges of weight 1. We do this by splitting e with $l_e - 1$ dummy vertices. Let this transformed graph be $G' = (V', E')$.

For all the vertices $s, t \in V(G)$, the shortest path between them in G' is essentially the same as in G —we go in the same direction but take a break every 1 unit of distance. This means that the distances between these vertices are the same.

Since in the new graphs all edges have weight 1, G' is an unweighted graph. This means that we can find the shortest paths on it using BFS. To recreate the shortest paths on G , you would only have to, for each $v_k \in V$ in a path, map from the first dummy vertex (in $V' - V$) that you traverse to the first $v_{k+1} \in V$ that follows.

What is the running time of this approach? Recall that the running time of finding shortest paths on an unweighted graph with BFS depends on the number of vertices and edges. But in our new graph G' , there are a lot more vertices and edges.

Let L denote $\sum_{e \in E} (l_e - 1) = \sum_{e \in E} l_e - |E|$. Then L counts the number of dummy vertices that we added to V' , so the size of V' is $|V'| = |V| + L$. How many edges do we have in E' ? Instead of each edge with weight l_e , we now have l_e edges, which gives $|E'| = \sum_{e \in E} l_e = |E| + L$ edges.

If we now run BFS on G' , it will take $O(|V'| + |E'|) = O(|V| + |E| + L)$ time. This bound may seem innocuous, but if you have a fairly small weighted graph with a few extremely long edges (large L), our algorithm will be prohibitively slow and use up a lot of space. The dependence on features of the graph other than its size is also undesirable. Despite there being no exponential terms in the bound, the running time of our algorithm is not actually polynomial.

4 Dijkstra's algorithm

Dijkstra's algorithm efficiently finds the shortest path from a given vertex s on a weighted graph $G = (V, E)$ with weights given by l .

Algorithm 1: Dijkstra's Algorithm

```
1 def Dijkstra( $G, l, s$ ):
    /* Input:  $G = (V, E)$  in the form of an adjacency list  $Adj$  */
    /*  $l$  is a function such that  $l(x, y)$  is the weight of edge  $(x, y)$  */
    /*  $s$  is the starting vertex */
2 for  $v \in V$  do
3      $dist[v] \leftarrow \infty$ 
4      $prev[v] \leftarrow \text{NULL}$ 
5 end
6  $dist[s] \leftarrow 0$ 
7  $prev[s] \leftarrow s$ 
8  $R \leftarrow \{ \}$ 
9 while  $R \neq V$  do
10      $v \leftarrow \text{argmindist}[v']$  /* Find vertex with smallest distance not in R */
         $v' \notin R$ 
        /* How would you do this? Use a priority queue! (See Lecture 7) */
11      $R \leftarrow R \cup \{v\}$ 
12     for  $w \in Adj[v]$  do
13         if  $dist[w] > dist[v] + l(v, w)$  then
14              $dist[w] = dist[v] + l(v, w)$ 
15              $prev[w] = v$ 
16         end
17     end
18 end
```

Why does Dijkstra's algorithm work? Intuitively, consider the following. The full proof will follow in Lecture 7.

Claim 1 (Optimal substructure property). *If the shortest path P between s and t is via some vertex u among other vertices, then the sub-path P_{su} from s to u must be the shortest path from s to u .*

Proof. Otherwise, we could exchange this sub-path P_{su} for a shorter path \tilde{P}_{su} from s to u , and the resulting path from s to t (by following \tilde{P}_{su} from s to u , then P from u to t) would be shorter. This would contradict the statement that P is the shortest path from s to t . \square