CS 341: ALGORITHMS (S18) — LECTURE 15
SSSP ON WEIGHTED GRAPHS

ERIC BLAIS

We saw in the last lecture that greedy algorithms can be used to find minimum spanning trees of weighted graphs. Today we will see how greedy algorithm can also be used to solve the Single-source shortest path (SSSP) problem on weighted directed graphs.

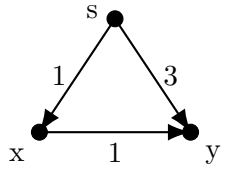
1. SSSP with Small Integer Weights

The Single-source shortest path problem is generalized in the natural way to weighted graphs.

Definition 15.1. In the *Single-source shortest path (SSSP)* problem on weighted graphs, our input is a (directed or undirected) graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ and a source vertex $s \in V$. The valid solution is the minimum total weight of a path in G from s to v for every vertex $v \in V$.

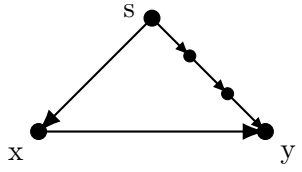
If we consider the special case of the problem where the weight of each edge is 1, then we are back to the original unweighted version of the SSSP problem and we can solve this problem in time $O(m + n)$ using breadth-first search.

What if we now consider a slightly more general setting where all the edges of the graph have weight 1,2, or 3?



Running BFS as-is on the graph G (while ignoring the edge weights) no longer solves the SSSP problem in this setting, as we see in the example above when the BFS starts at vertex s: since vertex y is reachable directly from s by following an edge of weight 3, this is how y will be discovered by the BFS and the solution built in this way will claim that y is at distance 3 from s, whereas there is a shorter path of length 2 between the two vertices in the graph.

We can, however, reduce the SSSP problem on graphs with edge weights 1, 2, or 3 to the SSSP problem on unweighted graphs: all we need to do is add extra (dummy) vertices to the graph to split up longer edges appropriately.



This guarantees that BFS now visits the vertices in order of increasing distance (as measured by the weights of the edges of the original graphs, not just the number of edges followed) and that when we output the distance from the source to each non-dummy nodes, we correctly solve the SSSP problem. The time complexity of the algorithm is $O(m' + n')$ when m' and n' are the number of edges and vertices of the unweighted graph we created with the dummy variables. This is still $O(m + n)$ when the weights are only 1, 2, or 3, but this complexity grows very quickly if we allow (much) larger weights. To handle that scenario more efficiently, we want to consider a slightly more sophisticated approach.

2. SSSP with Nonnegative Weights: Dijkstra's algorithm

Consider now the SSSP problem on graphs with arbitrary *non-negative* edge weights. We can solve this problem by modifying BFS instead of trying to use it directly: let's design an algorithm that visits the vertices in order of distance from the start vertex s. We can do this with a Priority Queue. The resulting algorithm is known as *Dijkstra's algorithm*.

Algorithm 1: Dijkstra$(G = (V, E), w, s)$

```
for all v ∈ V do
    dist[v] ← ∞;
dist[s] ← 0;
H ← MakePriorityQueue(V, dist);
while H ≠ ∅ do
    u ← DeleteMin(H);
    for all v ∈ AdjList(E, u) do
        if dist[u] + w(u, v) < dist[v] then
            dist[v] ← dist[u] + w(u, v);
            DecreaseKey(H, v, dist[v]);
return dist;
```

We say that Dijsktra's algorithm is a greedy algorithm because at each step it explores the vertex that is closest to the source among all of those that have not been visited yet. You can see an example of the execution of Dijkstra's algorithm on a weighted graph in the course textbook.

Informally, Dijsktra's algorithm is correct because we can never have a shortest path from s to a vertex v that goes through a vertex u which is further away from s than v. (Note that this will no longer be true when we allow negative edge weights!) We can formalize this argument with a proof by induction.

Theorem 15.2. *Fix any directed weighted graph G with non-negative edge weights. At every iteration of the while loop in Dijkstra's algorithm, the distance* `dist`$[v]$ *from s to every vertex v that is not in H is correct.*

Proof. We prove the theorem by induction on the number of vertices that are not in H.

In the base case, when only s is removed from H, the theorem is true since we initialize `dist`$[s] = 0$.

For the induction step, we assume that it is true for all the vertices that have been removed from H before the current iteration of the loop and we want to show that it is also true for the vertex u removed from the priority queue in the current iteration.

That is: we want to show that

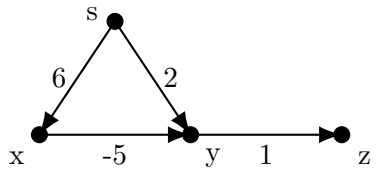
$$\texttt{dist}[u] = \min_{v \notin H}\{\texttt{dist}[v] + w(v,u)\}.$$

Let $v^* \notin H$ be a vertex where the minimum is attained. By the induction hypothesis, $\texttt{dist}[v^*]$ is the distance from s to v^*, so we do have that $\texttt{dist}[u] \leq \texttt{dist}[v^*] + w(v^*, u)$.

To show a matching lower bound on $\texttt{dist}[u]$, we want to show that every path P from s to u has length at least $\texttt{dist}[v^*] + w(v^*, u)$. Since P starts from a vertex (namely: s) that has previously been removed from H and ends at a vertex (u) that was not previously removed from H, there must be an edge (x, y) in P where $x \notin H$ and $y \in H$. Then the length of the path P must be at least $\texttt{dist}[x] + w(x, y)$ by the induction hypothesis and the fact that G has no edge with negative weight. The choice of u as the next vertex to remove from H implies that we must also have $\texttt{dist}[x] + w(x, y) \geq \texttt{dist}[v^*] + w(v^*, u)$. Therefore, $\texttt{dist}[u] \geq \texttt{dist}[v^*] + w(v^*, u)$. □

3. SSSP with Negative Edge Weights

3.1. **Dijkstra's algorithm.** If we consider graphs with negative edge weights, Dijkstra's algorithm no longer computes the correct weight of the shortest path from the source to every other vertex in the weighted graph. Consider for instance the following simple example.

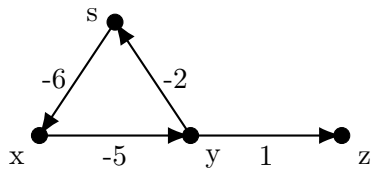


Dijkstra's algorithm will return the distances

$$\texttt{dist}[s] = 0 \qquad \texttt{dist}[x] = 6 \qquad \texttt{dist}[y] = 2 \qquad \texttt{dist}[z] = 3$$

whereas the distance from s to y is actually 1, from the length of the path $s \to x \to y$ and the distance from s to z is 2, not 3. We can modify Dijkstra's algorithm so that the distances to vertices are updated when we discover a shorter path to a vertex v obtained by following negative-weight edges, but we have to be careful when we do so to update the distances to all vertices whose shortest path goes through v as well. This means that we will no longer have a greedy algorithm. And so, instead of applying this ad hoc method for fixing the algorithm, it's worth taking a step back and seeing if another algorithm design technique can be used to solve the SSSP problem. Before we do so, however, there is another worrisome example that we need to consider.

3.2. **Negative-weight cycles.** What is the weight of the shortest path between s and z in the following graph?



The path $s \to x \to y \to z$ has total weight -10. But the path $s \to x \to y \to s \to x \to y \to z$ has total weight -23. And the path that follows the cycle $s \to x \to y \to s$ twice has total weight -36, etc. What we observe is that if a graph has a cycle with negative total weight, then the distance of vertices that can be reached via a path that goes through the cycle is unbounded (or can be set to $-\infty$).

For now, let's avoid the problems caused by negative weight cycles by only considering graphs that don't contain any.

3.3. **Bellman–Ford algorithm.** Let's now revisit the problem of designing an algorithm for SSSP with negative weights but no negative-weight cycles. Since greedy algorithms don't work for this problem, it's natural to try to design a Dynamic Programming solution for the problem. The key question when we do so is: what will be our simpler subproblems? Recall that we define

$$\texttt{dist}[v] = \text{length of the shortest path from } s \text{ to } v.$$

We obtain simpler subproblems if we don't consider *all* paths from s to v but only paths that use at most 1, 2, 3, ... edges. Specifically, for our subproblems we wish to compute

$$d_i[v] = \text{length of the shortest path from } s \text{ to } v \text{ that uses} \leq i \text{ edges}$$

for every $v \in V$ and for every $i = 1, 2, \ldots, n-1$. The base case is easy to compute:

$$d_1[v] = \begin{cases} w(s,v) & \text{if } (s,v) \in E \\ \infty & \text{otherwise.} \end{cases}$$

And the values $d_i[v]$ are easy to compute once we have computed $d_{i-1}[u]$ for every $u \in V$:

$$d_i[v] = \min \begin{cases} d_{i-1}[v] \\ d_{i-1}[u] + w(u,v) \quad \text{for each } u \text{ such that } (u,v) \in E. \end{cases}$$

Putting everything together, we obtain the *Bellman–Ford algorithm.*

Algorithm 2: BellmanFord$(G = (V,E), w, s)$

```
for all v ∈ V do
    if (s, v) ∈ E then
        d_1[v] ← w(s, v);
    else
        d_1[v] ← ∞;
for i = 2, 3, ..., n − 1 do
    for all v ∈ V do
        d_i[v] ← d_{i−1}[v];
        for all u ∈ V do
            if (u, v) ∈ E and d_{i−1}[u] + w(u, v) < d_i[v] then
                d_i[v] ← d_{i−1}[u] + w(u, v);
return d_{n−1};
```