CS 341: ALGORITHMS (S18) — LECTURE 16
ALL PAIRS SHORTEST PATH

ERIC BLAIS

1. APSP Problem

We have seen some algorithms for solving the *Single-source shortest path* (SSSP) problem, where we want to compute the length of the shortest path in a graph $G = (V, E)$ between some special source vertex $s \in V$ and every other vertex in V. Today we explore a natural variant of the problem, where we want to compute the minimum distance between *all* pairs of vertices in V.

Definition 16.1 (All-pairs shortest path (APSP))**.** In the *all-pairs shortest path (APSP)* problem, we are given a weighted (directed or undirected) graph $G = (V, E)$ and we must return the length of the shortest path from u to v for every $u, v \in V$.

As we did with the SSSP problem, we will consider only the special case of the problem where the input graph is guaranteed not to contain any negative-weight cycles.

There is a simple solution to the APSP problem: run the Bellman–Ford algorithm n times on the same graph with all possible vertices as the source vertex.

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

```
for all u ∈ V do
    d[u, *] ← BellmanFord(G, w, u);
return d;
```

The correctness of this algorithm follows directly from the correctness of the Bellman–Ford algorithm. Recall that the Bellman–Ford algorithm is obtained via the dynamic programming technique by considering the shortest paths from s to v that go over only 1, 2, 3, $\ldots$, $n-1$ edges. We saw in the last lecture that we can store these distances in vectors $d_1, d_2, \ldots, d_{n-1}$. It turns out that we don't need to store all these distance vectors separately, and we can simplify the Bellman–Ford algorithm to use a single distance vector that we update as follows.

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

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

See the textbook for more discussion and an analysis of this version of the algorithm. For today's lecture, the important point to note is that its time complexity is $\Theta(nm)$. This means that the SimpleAPSP algorithm we designed earlier has time complexity $\Theta(n^2 m)$, which can be as large as $\Theta(n^4)$ when the graph is dense. Can we do better?

2. Modifying the Bellman–Ford algorithm

The first observation we might make of our SimpleAPSP algorithm is that it appears to be inefficient in that each call to the Bellman–Ford algorithm results in the computation of a lot of shortest paths that end up being discarded when we call it again with another source vertex. It would probably be much better to design a dynamic programming algorithm directly for the APSP problem so that we don't get such waste.

And it turns out that we don't need to look very far to find a good option for designing a dynamic programming algorithm for APSP: let's use the same subproblems! For every pair $u, v \in V$ of vertices, define

$$d_i(u, v) = \text{shortest path from } u \text{ to } v \text{ that goes through} \leq i \text{ edges.}$$

Then we can apply the same base case and recurrence relation as we did in the Bellman–Ford algorithm, but update distances for all pairs of vertices, resulting in the following algorithm.

Algorithm 3: BellmanFordAPSP$(G = (V, E), w)$

```
for all u, v ∈ V × V do
    if (u, v) ∈ E then
        d[u, v] ← w(u, v);
    else
        d[u, v] ← ∞;
for i = 2, 3, ..., n − 1 do
    for all u ∈ V do
        for all (x, v) ∈ E do
            if d[u, x] + w(x, v) < d[u, v] then
                d[u, v] ← d[u, x] + w(x, v);
return d;
```

This algorithm has time complexity $O(n^2 m)\ldots$ exactly the same as the SimpleAPSP algorithm! Conclusion: To obtain a more efficient algorithm with the dynamic programming technique, we need to find other subproblems to solve.

3. Floyd–Warshall algorithm

Let's go back to the problem we're trying to solve. We want to compute

$$\text{dist}[u, v] = \text{minimum weight of any path from } u \text{ to } v.$$

To design the Bellman–Ford algorithm, we noted that the problem of computing the distance from u to v is much easier if we restrict the set of paths that we consider. This observation led us to consider only paths that go through a small number of edges. But that's not the only way to restrict our attention to some of the paths only. In particular, instead of limiting the number of edges that the paths can use, we can limit the *set of intermediate nodes* that those paths can traverse. Let us label the vertices in V as $v_1, v_2, \ldots, v_n$. Then for each $0 \leq k \leq n$, we can define

$$\begin{aligned}\text{dist}_k[u, v] =& \text{minimum weight of any path from } u \text{ to } v \text{ that} \\ & \text{only uses } v_1, \ldots, v_k \text{ as intermediate nodes.}\end{aligned}$$

The base case of these subproblems is the same as before: when $k = 0$, the only possible path from u to v is the one that traverses the edge (u, v), if it exists. So

$$\text{dist}_0[u, v] = \begin{cases} w(u, v) & \text{if } (u, v) \in E \\ \infty & \text{otherwise.} \end{cases}$$

Now to compute $\text{dist}_k[u, v]$, we observe that there are two possibilities: either the shortest path from u to v that only uses $v_1, \ldots, v_k$ as intermediate nodes goes through v_k, or it does not. So

$$\text{dist}_k[u, v] = \min \begin{cases} \text{dist}_{k-1}[u, v_k] + \text{dist}_{k-1}[v_k, v] \\ \text{dist}_{k-1}[u, v]. \end{cases}$$

We combine those observations to obtain the *Floyd–Warshall algorithm*. In the following algorithm, let $V = \{v_1, v_2, \ldots, v_n\}$.

Algorithm 4: FloydWarshall$(G = (V, E), w)$

```
for all u, v ∈ V × V do
    if (u, v) ∈ E then
        dist_0[u, v] ← w(u, v);
    else
        dist_0[u, v] ← ∞;
for k = 1, 2, 3, ..., n do
    for all u ∈ V do
        for all v ∈ V do
            dist_k[u, v] ← min{dist_{k-1}[u, v_k] + dist_{k-1}[v_k, v], dist_{k-1}[u, v]};
return dist_n;
```

The time complexity of this algorithm is $\Theta(n^3)$.