CSC 373 - Algorithm Design, Analysis, and Complexity **Summer 2016**

Lalla Mouatadid

DP: All Pairs Shortest Paths, The Floyd-Warshall Algorithm

So far, we've covered Dijkstra's Algorithm, which solves the (s,t) shortest path problem (you're given a specific source and a terminal). We also covered the Bellman-Ford Algorithm which solves the single source shortest paths (you're given a specific starting point s). It is thus natural to ask whether there is a simple algorithm that solves the all pairs shortest paths; that is, the shortest path between any two arbitrary vertices in the graph. We formally define the following problem:

All Pairs Shortest Paths:
Input: An edge-weighted graph $G(V, E, w)$ where $w : E \to \mathbb{R}$
Output: The length of the shortest path from u to v for any $u, v \in V$.

A natural way to approach this problem is to just use the algorithms we have with different vertices u, v as sources and terminals. For instance, we could use Dijkstra's Algorithm starting with $s = v$ for every $v \in V$. Since Dijkstra's Algorithm runs in $\mathcal{O}(m + n \log n)$, our algorithm would take $\mathcal{O}(nm + n^2 \log n)$, but then - as we saw for Bellman-Fords- we need to assume that out graph contains no edges with negative weights.

On the other hand, if we use Bellman-Ford Algorithm, the total runtime would be $\mathcal{O}(n^2 m)$. And if G is a dense graph (i.e. $|E|$ is $\Omega(n^2)$), then our algorithm would take $\mathcal{O}(n^4)$, which is ..hmm.. inefficient. Especially if we think about how many times a u, v path is computed.

Consider for instance a graph $G(V, E)$ where $V = \{v_1, v_2, v_3, v_4\}$. If we run one of the algorithms above starting with v_1, the paths $P_{v_1v_2}, P_{v_1v_3}, P_{v_1v_4}, P_{v_1v_5}$ would be compute more than once. That's a total of $\mathcal{O}(n)$ extra paths we don't need to compute..

Our goal is thus to beat the $\mathcal{O}(n^4)$ runtime (using DP hopefully :) .

Before attacking the problem, let's define some notation that we'll use later:

We use P_{ab} to denote the path P between vertices a and b, and we use $V(P_{ab})$ to denote the set of vertices on P other than a and b

$$V(P_{ab}) = \{v \text{ is in the path } P_{ab} | v \neq a \text{ and } v \neq b\}$$

Given a set $S \subseteq V$, we say that P_{ab} is **restricted** to S if $V(P_{ab}) \subseteq S$. In other words, the elements of the set S are the only vertices allowed to appear on P_{ab}.

only elements in S are on the path

Okay, so how do we break this problem down to sub-problem.

We want the shortest path between any two vertices. So if we restrict our focus to one pair of vertices i, j, we seek the shortest path P_{ij} between these two vertices. This path P_{ij} can contain any vertex $v \in V$. In other words, P_{ij} is restricted to S where $S = V$. We ask the question of whether P_{ij}, the shortest i, j path, can be restricted to a just subset of V. That is, can we construct P_{ij} from a small subset $V(P_{ij})$ of V?

Let's formalize this idea. We use an array $M[i, j, k]$ to denote the length of the shortest path P_{ij} that uses the **first** k vertices. Ah! When we say **first**, we must have some sort of ordering imposed on the vertices. Well in this case, we don't care what vertices we will pick, since we will try all of them. So we sort the vertices in some arbitrary order $V = \{v_1, v_2, ..., v_n\}$, where i will refer to vertex v_i for notational simplicity.

Okay nice! Now we have this ordering, so $M[i,j,k]$ is just:

$$M[i,j,k] = \text{ The length of the shortest } P_{ij} \text{ path restricted to } \{1,2,...,k\}$$

What's the base case? If we don't consider any set of vertices, then $M[i,j,0]$ is just the weight of the edge (i,j) so $M[i,j,0] = w_{ij}$.

Now we need a way to recursively define $M[i,j,k]$. Since we have this ordering $\{v_1, v_2, ..., v_{k-1}, v_k, v_{k+1}, ..., v_n\}$ on V, we can just check at every step whether the k^{th} vertex contributes to the construction of a shortest P_{ij} or not. Therefore it suffices to check whether k appears in P_{ij}. If it doesn't, then we conclude that:

$$M[i,j,k] = M[i,j,k-1]$$

And if k appears in P_{ij}, then our path looks something like: $P_{ij} = v_i \cdot ...v_k... \cdot v_j$. What do we know about the vertices between v_i and v_j? What's $V(P_{ij})$? Well, $V(P_{ij})$ must be drawn from the set $\{1,2,...,k\}$, (why?). So if $V(P_{ij}) \subseteq \{1,2,...,k\}$, what can we say about the subpaths P_{ik} and P_{kj}? They're both subsets of $\{1,2,...,k-1\}$! So in the case of k being an element on P_{ij}, we have:

$$M[i,j,k] = M[i,k,k-1] + M[k,j,k-1]$$

That is, the shortest path from i to j is the just the sum of the shortest paths from i to k and k to j. Since we're trying to minimize the total length of P_{ij}, we take the min of both cases, so:

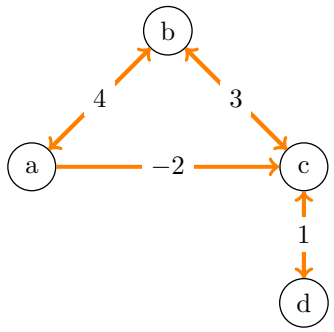
$$M[i,j,k] = \min\{M[i,j,k-1], M[i,k,k-1] + M[k,j,k-1]\}$$

min (k not on path, k on path)

Since we're trying all k up to n vertices. To prove the optimality of the algorithm, it suffices to show that $M[i,j,k]$ is indeed a minimum length of P_{ij}.

Let's first go through an example to illustrate how the algorithm works. The first table is our base case (i.e. $k = 0, T[i,j,0] = w_{ij}$). The remaining tables compute both $M[i,j,k-1]$ and $M[i,k,k-1] + M[k,j,k-1]$ for $k \in [4]$ and choose the best (min) value. Suppose the arbitrary ordering we fix is $a, b, c, d = v_1, v_2, v_3, v_4$ respectively:

Goal:
Find a path from i to j that is using a steadily increasing subset of vertices in V. We are looking for the smallest/shortest



edge weights

k=0	a	b	c	d
a	-	4	-2	∞
b	4	-	3	∞
c	∞	3	-	1
d	∞	∞	1	-

k=1	a	b	c	d
a	-	4	-2	∞
b	4	-	2	∞
c	∞	3	-	1
d	∞	∞	1	-

k=2	a	b	c	d
a	-	4	-2	∞
b	4	-	2	∞
c	7	3	-	1
d	∞	∞	1	-

k=3	a	b	c	d
a	-	1	-2	-1
b	4	-	2	3
c	7	3	-	1
d	8	4	1	-

k=4	a	b	c	d
a	-	1	-2	-1
b	4	-	2	3
c	7	3	-	1
d	8	4	1	-

This algorithm is known as the Floyd-Warshall algorithm, which runs in $\mathcal{O}(n^3)$ time[1].

Algorithm 1 All Pairs Shortest Path

```
for 1 ≤ i, j, ≤ n do
    M[i, j, 0] = w_ij
end for
for k = 1 ... n do
    for i = 1 ... n do
        for j = 1 ... n do
            M[i, j, k] = min{M[i, j, k − 1], M[i, k, k − 1] + M[k, j, k − 1]}
        end for
    end for
end for
```

Proof of optimality:

Proof. For the base case, we've already discussed how $M[i, j, 0]$ is just the weight of the edge (i, j). So $M[i, j, 0] = w_{ij}$. Suppose that $M[i, j, l]$ is optimal for all $l < k$ and consider the case where $l = k$.

Let $\tilde{P}_{ij}$ denote a shortest (optimal) i, j path restricted to $\{1, 2, ..., k\}$. If $\tilde{P}_{ij}$ does not contain k, then the length of $\tilde{P}_{ij}$ is the same as the length of the shortest i, j path restricted to $\{1, 2, ..., k - 1\}$, and so by induction hypothesis $w(\tilde{P}_{ij}) = M[i, j, k - 1]$[2]

If $\tilde{P}_{ij}$ does contain k, then $\tilde{P}_{ij}$ is the concatenation of two paths $\tilde{P}_{ik}$ and $\tilde{P}_{kj}$ from i to k and k to j respectively. Since k appears on both $\tilde{P}_{ik}$ and $\tilde{P}_{kj}$, all the vertices of $\tilde{P}_{ik}$ and $\tilde{P}_{kj}$ other than i, j and k must be drawn from the set $\{1, 2, ..., k - 1\}$. Meaning $\tilde{P}_{ik}$ is an i, k path restricted to $\{1, 2, ..., k - 1\}$ and $\tilde{P}_{kj}$ is a k, j path restricted to $\{1, 2, ..., k - 1\}$. Notice that both $\tilde{P}_{ik}$ and $\tilde{P}_{kj}$ are optimal, since otherwise we contradict the optimality of $\tilde{P}_{ij}$. By induction hypothesis, $w(\tilde{P}_{ik}) = M[i, k, k - 1]$ and $w(\tilde{P}_{kj}) = M[k, j, k - 1]$, therefore $w(\tilde{P}_{ij}) = w(\tilde{P}_{ik}) + w(\tilde{P}_{kj}) = M[i, k, k - 1] + M[k, j, k - 1]$.

Since our recursive definition considers both cases and picks the minimum, it follows that $M[i, j, k]$ is optimal. □

[1] and beats the $\mathcal{O}(n^4)$:3

[2] I'm abusing notation here and using $w(\tilde{P}_{ij})$ to denote the weight (i.e. the length) of the shortest i, j path.