All Pairs Shortest Paths

We want to know all shortest paths between each $u, v \in V$, that is for all $w(p_{u\to v}) = \sum_{e\in p_{u\to v}} w(e)$ we want $p^*(u,v) = argmin_{p:u\to v} w(p)$ ($\delta(u,v) = w(p^*_{u\to v})$. We denote the shortest paths matrix is $D = (\delta_{ij})$.

We can apply the **single** source shortest path n times, but that's inefficient.

To get a $O(n^3)$ running time we will apply a sort of matrix-multiplication (which can actually be done in $\sim O(n^{2.7})$, but that's another issue).

Using Matrix Multiplication:

We start with the weight matrix:

$$w_{ij} = \begin{cases} 0, & i = j \\ w(i,j), & i \neq j, (i,j) \in E \\ \infty, & (i,j) \notin E \end{cases}$$

We define $d_{ij}^{(m)}$ as the shortest path between i and j with at most m edges. For $m = 0$:

$$d_{ij}^{(0)} = \begin{cases} 0, & i = j \\ \infty, & i \neq j \end{cases}$$

Recursively we define:

$$d_{ij}^{(m)} = \min_{1\leq k\leq n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}$$

We must assume that **there are no negative cycles**, thus every shortest path has at most $n-1$ edges.

The term $d_{ij}^{(m)}$ says: take the best way out of all paths that get to any k with $m-1$ edges (that best way was previously calculated), and from that k to j (with cost of $w(k,j)$).

So given $D^{(1)}, W$ we can get $D^{(1)}, D^{(1)}, W$ to get $D^{(2)}$ and so on until $D^{(n-1)}$. We stop at $n-1$ since as said every shortest path has at most $n-1$ edges.

The basic operation of the update algorithm is $d_{ij} = \min\{d_{ij}, d_{ik} + w(k,j)\}$, which takes constant time. as in matrix multiplication, one iteration will take $O(n^3)$. We need to repeat $n-1$ iterations, therefore $O(n^4)$ is the total running time.

Say $A, B \in Mat_{n\times n}$, and let $C = A \cdot B$. Then $C_{ij} = A_i \times B^j = \sum_{k=1}^{n} A_{ik} \times B_{kj}$.

Assume $A = D^{(m-1)}, B = W$, and substitute the $\sum$ with min and the $\times$ with $+$, so we get:

$$c_{ij} = \min_{k=1}^{n} \left(d_{ik}^{(m-1)} + w_{kj} \right) = d_{ik}^{(m)}$$

Therefore matrix multiplication is the same as the problem we're trying to solve. This problem is isomorphic to matrix multiplication.

Improving Running Time:

When looking at matrix multiplication: $(A \times B) \times (C \times D) = A \times (B \times C) \times D$. Furthermore $A \times A = A^2, A^2 \times A^2 = A^4$ and so on. So for our case:

$D^{(0)} \times W = D^{(1)} \to D^{(1)} \times D^{(1)} = D^{(2)} \to D^{(2)} \times D^{(2)} = D^{(4)} \to \cdots D^{(n-1)}$

$n-1$ might not be a power of 2, but it doesn't matter – after passing $D^{(n-1)}$ all matrices are the same, as the shortest paths don't improve anymore.

So we reduced the number of phases from n to $\lg n$, yielding total of $\Theta(n^3 \lg n)$.

We can detect negative cycles like in Bellman-Ford: do one more round of multiplication – if the values change, we know there's a negative cycle.

When compared to $\Theta(nm)$ of BF, here by just a factor of $O(\lg n)$ we solve the entire matrix instead of for one source.

The Floyd-Warshall Algorithm:

Here in every iteration we let the algorithm use more and more nodes for the path.

- First iteration: go from $i \to j$ using only 1
- Second iteration: go from $i \to j$ using only $1,2$
- …
- k^{th} iteration: go from $i \to j$ using only $1,2, \dots, k$

So we control not just the **number** of vertices used, but exactly **which** nodes are used.

There are a total of n phases, but every phase here is cheaper than the previous algorithm (actually every phase is $\Theta(n^2)$, that yields to a total $\Theta(n^3)$).

$D^{(0)} = W$: Since we don't allow any intermediate nodes in the path between i, j.

For calculating $D^{(1)}$, we set $d_{ij}^{(1)} = d_{ij}^{\{1\}} = \min\{d_{ij}^{\emptyset} = w_{ij}, d_{i1}^{\emptyset} + d_{1j}^{\emptyset}\}$, where the values in the $\{\}$ signify the set of nodes allowed to be used.

Next phase:

- $d_{ij}^{\{1,2\}} = \min\{d_{ij}^{\{1\}}, d_{i2}^{\{1\}} + d_{2j}^{\{1\}}\}$
- $d_{ij}^{\{1,2,3\}} = \min\{d_{ij}^{\{1,2\}}, d_{i3}^{\{1,2\}} + d_{3j}^{\{1,2\}}\}$

…

- $d_{ij}^{\{1,2,\dots,k\}} = \min\{d_{ij}^{\{1,2,\dots,k-1\}}, d_{ik}^{\{1,2,\dots,k-1\}} + d_{kj}^{\{1,2,\dots,k-1\}}\}$

Assume we have acquired $D^{(k-1)}$, how long does it take to get $D^{(k)}$? We need to compute an $O(1)$ calculation for each entry – the calculation mentioned above. Therefore the total running time for the entire matrix derives a $\Theta(n^2)$ running time. Doing this $n-1$ times yields a total running time of $\Theta(n^3)$.

<u>Can we speed it up</u>?

No. Here we're using a specific set of vertices for the paths, so $D^{(k)}$ uses only $\{1,2, \dots, k\}$ and so 2 intances of it will use basically the same nodes and yield the same answer.

This is a dynamic programming algorithm.

Transitive Closure:

Given a graph $G = (V, E)$, the graph $G^* = (V, E^*)$ is the transitive closure of G and is defined:

$E^* = \{(i,j) | \ there\ is\ a\ path\ from\ i\ to\ j\ in\ G\}$.

How can we compute the transitive closure of a graph G:

$$t_{ij}^{(0)} = \begin{cases} 0, i \neq j \ or \ (i,j) \notin E \\ 1, i = j \ or \ (i,j) \in E \end{cases}$$

And:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$$

The running time is $\Theta(n^3)$. This is like the FW algorithm, substituting $\min$ by $\vee$ and $+$ by $\wedge$.

Dynamic Programming:

Dynamic programming is a bottom-up optimization technique, where the optimal solution is computed from optimal solutions to sub-problems. There are overlapping sub-problems: this is important for allowing extension of one phase to the next phase.

Longest Common Subsequences (LCS):

Given a sequence $X = x_1 x_2 \dots x_m$ we say $Z = z_1 z_2 \dots z_k$ is a subsequence of X if there are indices $i_1, i_2, \dots, i_k$ such that $\forall j: x_{i_j} = z_{i_j}$ and $i_1 < \dots < i_k$.

The problem of a longest common subsequence: given two sequences $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$ we want to find $Z = z_1 z_2 \dots z_k$ where $Z \subset X$ and $Z \subset Y$, and k is maximal. That is we want Z' such that $\forall Z: |Z'| \geq |Z|$.

Denote $X_i = x_1 x_2 \dots x_i$ – the prefix of X of size i.

Theorem:

Let Z be the LCS of X, Y, so:

- If $x_m = y_n$ then $z_k = x_m$ implies that z_{k-1} is the LCS of X_{m-1}, Y_{n-1}.
- If $x_m \neq y_n$ then $z_k \neq x_m$ and Z is the LCS of X_{m-1}, Y.
- If $x_m \neq y_n$ then $z_k \neq y_n$ and Z is the LCS of X, Y_{n-1} (symmetric to the second case).

Denote c_{ij} the **length** of the LCS of X_i, Y_j, so:

$$c_{ij} = \begin{cases} 0, & i = 0 \ or \ j = 0 \\ c_{(i-1)(j-1)} + 1, & i,j > 0 \ and \ x_i = y_j \\ \max\{c_{i(j-1)}, c_{(i-1)j}\}, & i,j > 0 \ and \ x_i \neq y_j \end{cases}$$

For instance, if $c_{11} = 0$ then $c_{12} = 1$ if $x_1 = y_2$ and $c_{12} = 0$ if $x_1 \neq y_2$. In a similar way we can compute the case where $c_{11} = 1$. The total roe c_{1j} will take $\Theta(n)$ times to fill. The same goes for the column c_{i1}.

After we get the first row and column, what about c_{22}?

- Could be that $x_1 = y_1 \wedge x_2 = y_2$
- Could be that $x_1 \neq y_1 \vee x_2 \neq y_2$

In general, the value $c_{22} = \max\{c_{12}, c_{21}, c_{11} + 1\}$ where the last one is **if** $x_2 = y_2$.

So at each entry you look either left, up, or diagonal left-up + 1. Eventually the solution for X, Y is $c_{mn} = LCS(X, Y)$.

The total running time is: $\Theta(mn)$.

We can expand the problem to 3-strings LCS by using a cube instead of a 2-dimensional matrix, yielding $\Theta(nml)$ for $|X| = n, |Y| = m, |Z| = l$ (here Z is not the LCS but another string of the input). This can be extended to any number of input strings.