Analysis of Algorithms I: All-Pairs Shortest Paths

Xi Chen

Columbia University

The All-Pairs Shortest Paths Problem. Input: A directed weighted graph $G = (V, E)$ with an edge-weight function $w : E \to \mathbb{R}$. Output: $\delta(u, v)$ for all $u, v \in V$, where $\delta(u, v)$ denotes the shortest-path weight from u to v. Depending on the weights of G: If only nonnegative weights are allowed, we can run Dijkstra's algorithm $|V| = n$ times, once for each vertex $u \in V$: total running time is $O(n(m + n \lg n))$. If we allow negative weights, then run Bellman-Ford n times: $O(n^2 m)$. We will focus on the latter more general case, and give three algorithms with running time much better than $O(n^2 m)$.

Since we are dealing with directed graphs with general weights, recall that δ is well-defined over all pairs of vertices (u, v) only when there is no negative-weight cycle in G. We start by giving an algorithm, using Bellman-Ford, to test whether G has a negative cycle or not. Recall that if we run Bellman-Ford on G and a vertex $u \in V$, then it returns "negative cycle" if and only if there is a negative-weight cycle reachable from u.

Test if G has a negative-weight cycle:

1. Add a new vertex s and an edge (s, v) for every $v \in V$ with 0 weight. Call the new directed graph G'.
2. Run Bellman-Ford on G' and s: There is a negative cycle in G' reachable from s if and only if G has a negative cycle.

For the last statement: It is easy to see that if G has a negative cycle, then the same cycle must be reachable from s in G'. On the other hand, if G' has a negative cycle C then C cannot visit the source vertex s (why?) so it is also a negative cycle in G as well. This algorithm clearly has running time $O(nm)$.

Now assume G has no negative cycle. We start with Johnson's algorithm for all-pairs shortest paths. The running time is

$$O(n^2 \lg n + nm)$$

Compare it with $O(nm)$ of Bellman-Ford for single-source shortest paths. When does it have essentially the same running time as Bellman-Ford? Johnson's algorithm uses the so-called technique of reweighting: As mentioned earlier, if all the weights are nonnegative, we can compute δ by running Dijkstra n times:

$$O(n(m + n \lg n)) = O(n^2 \lg n + nm)$$

But our input graph G may have negative weights. In this case, Johnson's algorithm uses Bellman-Ford to compute a new *nonnegative* weight function $w' : E \to \mathbb{R}_{\geq 0}$ such that

> For any $u, v \in V$, it is easy to compute $\delta(u, v)$ using $\delta'(u, v)$, where δ' denote the shortest-path weight in G with w'.

It will become clear later how easy it is to recover $\delta(u, v)$ from $\delta'(u, v)$. But if it is the case, then we get the following algorithm:

1. Reweight w: Compute (nonnegative) w' from w
2. Run Dijkstra n times to compute $\delta'(u, v)$ for all $u, v \in V$
3. For all $u, v \in V$, compute $\delta(u, v)$ from $\delta'(u, v)$

Here is how we compute w' (reweight): Run Bellman-Ford on G' (recall the construction of G' from G earlier) and s. Assume there is no negative cycle in G'. Then upon termination we get $\delta_{G'}(s, v)$ for all $v \in V$, where $\delta_{G'}(s, v)$ denotes the shortest-path weight from s to v in G' (note that it must be ≤ 0, why?). Finally for each edge $(u, v) \in E$, reweight $w(u, v)$ to be:

$$w'(u, v) = w(u, v) + \delta_{G'}(s, u) - \delta_{G'}(s, v)$$

Two things we need to check: (1) Is w' nonnegative? (2) How can we recover $\delta(u, v)$ from $\delta'(u, v)$ efficiently? For convenience we use $h(u)$ to denote $\delta_{G'}(s, u)$ for $u \in V$. First, we show that $w'(u, v) \geq 0$ for all $(u, v) \in E$. This is because $\delta_{G'}$ satisfies

$$\delta_{G'}(s, v) \leq \delta_{G'}(s, u) + w(u, v) \;\Rightarrow\; w(u, v) + h(u) - h(v) \geq 0$$

The second question is less trivial. Let $p = \langle v_0 v_1 \cdots v_k \rangle$ denote a path from $u = v_0$ to $v = v_k$ in G. We compare $w(p)$ and $w'(p)$:

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \quad \text{and} \quad w'(p) = \sum_{i=0}^{k-1} w'(v_i, v_{i+1})$$

Plugging in the construction of w' from w, we have

$$\begin{aligned} w'(p) &= \sum_{i=0}^{k-1} \Big(w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1}) \Big) \\ &= w(p) + h(v_0) - h(v_k) = w(p) + h(u) - h(v) \end{aligned}$$

As a result:

$$\begin{aligned} \delta'(u,v) &= \min_{p: u \rightsquigarrow v} w'(p) = \min_{p: u \rightsquigarrow v} \Big(w(p) + h(u) - h(v) \Big) \\ &= h(u) - h(v) + \min_{p: u \rightsquigarrow v} w(p) = h(u) - h(v) + \delta(u,v) \end{aligned}$$

To summarize, here is Johnson's algorithm:

1. Construct G' from G
2. Bellman-Ford on G' and s to get $h(v) = \delta_{G'}(s, v)$, $\forall\, v \in V$
3. For each edge $(u, v) \in E$ do
4. set $w'(u, v) = w(u, v) + h(u) - h(v)$
5. Run Dijkstra n times to compute $\delta'(u, v)$ for all $u, v \in V$
6. For all $u, v \in V$ do
7. set $\delta(u, v) = \delta'(u, v) + h(v) - h(u)$

Total running time: $O(nm + n(m + n \lg n)) = O(nm + n^2 \lg n)$.

Next we present two algorithms based on Dynamic Programming. In both algorithms, we assume there is no negative cycle in G so $\delta(u, v)$ is always well-defined. In both algorithms, we fill up a 3-dimensional table of size n^3, but based on different recursive formulas. The cells of the two tables also have different meanings. From now on, we assume $V = \{1, \ldots, n\} = [n]$ and let $\mathbf{W} = (w_{ij})$ denote the following $n \times n$ matrix:

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

The first algorithm is based on the following recursive formula: Given $i, j \in V$ and $t \geq 1$, let $d_{ij}^{(t)}$ denote the minimum weight of any path from i to j that contains at most t edges. Thus,

$$d_{ij}^{(1)} = w_{ij}$$

For each $t \geq 1$ we also define the following $n \times n$ matrix $\mathbf{D}^{(t)}$: The (i, j)th entry of $\mathbf{D}^{(t)}$ is exactly $d_{ij}^{(t)}$, $i, j \in V$.

Now it becomes clear that in the DP algorithm, we will start with $\mathbf{D}^{(1)}$ and use it to compute $\mathbf{D}^{(2)}$, and then $\mathbf{D}^{(3)}$, and so on and so forth. Before giving the recursive formula for computing $\mathbf{D}^{(t)}$ from $\mathbf{D}^{(t-1)}$, we first answer the following question: When should this algorithm stop? The answer is $t = n - 1$ because we have

$$\delta(i, j) = d_{ij}^{(n-1)}$$

This follows from the fact that, because there is no negative cycle there must be a simple path from i to j with at most $n - 1$ edges. Now our goal is clear: start from $\mathbf{D}^{(1)} = \mathbf{W}$ and compute $\mathbf{D}^{(n-1)}$.

Assume we have computed $\mathbf{D}^{(t-1)}$ for some $t \geq 2$. How can we use it to compute $\mathbf{D}^{(t)}$ efficiently? Here is a recursive formula:

$$d_{ij}^{(t)} = \min_{k \in V} \left\{ d_{ik}^{(t-1)} + w_{kj} \right\} \tag{1}$$

Intuitively, this formula says that to get $d_{ij}^{(t)}$, we just need to enumerate all possible predecessors k of j. For each k, concatenate a shortest path from i to k, that contains at most $t-1$ edges, and (k, j) of weight w_{kj}. Taking the minimum gives us $d_{ij}^{(t)}$.

The proof is very simple. Let $p = \langle i_0 i_1 \cdots i_{\ell-1} i_\ell \rangle$ denote a shortest-path from i to j of length at most t so that

$$d_{ij}^{(t)} = w(p)$$

Let $p' = \langle i_0 i_1 \cdots i_{\ell-1} \rangle$ and $s = i_{\ell-1}$, then (why?)

$$d_{ij}^{(t)} = w(p) = w(p') + w_{sj} \geq d_{is}^{t-1} + w_{sj}$$

This implies that

$$d_{ij}^{(t)} \geq \min_{k \in V} \left\{ d_{ik}^{(t-1)} + w_{kj} \right\}$$

The other direction is even simpler.

This immediately gives us the following DP algorithm:

1. set $\mathbf{D}^{(1)} = \mathbf{W}$
2. for t from 2 to $n-1$ do
3. for i from 1 to n do
4. for j from 1 to n do
5. set $d_{ij}^{(t)} = \min_{k \in V} \left\{ d_{ik}^{(t-1)} + w_{kj} \right\}$
6. return $\mathbf{D}^{(n-1)}$

The running time is clearly $\Theta(n^4)$. We next give a connection to matrix multiplication to reduce its running time to $O(n^3 \lg n)$.

To this end, recall that given three $n \times n$ matrices:

$$\mathbf{A} = (a_{ij}), \quad \mathbf{B} = (b_{ij}) \quad \text{and} \quad \mathbf{C} = (c_{ij})$$

$\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ means the following equation for all $i, j \in [n]$:

$$c_{ij} = \sum_{k \in [n]} a_{ik} \cdot b_{kj}$$

Now we define the so-called "funny" multiplication of $\mathbf{A}$ and $\mathbf{B}$:

$$\mathbf{C} = \mathbf{A} \otimes \mathbf{B}, \quad \text{where} \quad c_{ij} = \min_{k \in [n]} \left\{ a_{ik} + b_{kj} \right\}$$

Basically we replace "$\cdot$" by "$+$" and replace $\sum$ by min. It is easy to check that this "funny" operation remains associative:

$$\mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}) = (\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C}$$

Using the "funny" multiplication and (1), we have for any $t \geq 2$

$$\mathbf{D}^{(t)} = \mathbf{D}^{(t-1)} \otimes \mathbf{W}$$

Because $\mathbf{D}^{(1)} = \mathbf{W}$, we have

$$\mathbf{D}^{(t)} = \Big(((\mathbf{W} \otimes \mathbf{W}) \otimes \mathbf{W}) \otimes \cdots \otimes \mathbf{W} \Big) \otimes \mathbf{W}$$

Using the technique of repeated squaring, we can compute the matrix $\mathbf{D}^{(n-1)}$ more efficiently as follows:

Assume $n-1$ is a power of 2, then starting from $\mathbf{D}^{(1)} = \mathbf{W}$:

1. $\mathbf{D}^{(2)} = \mathbf{D}^{(1)} \otimes \mathbf{D}^{(1)}$
2. $\mathbf{D}^{(4)} = \mathbf{D}^{(2)} \otimes \mathbf{D}^{(2)}$
3. $\ldots$
4. $\mathbf{D}^{(n-1)} = \mathbf{D}^{(n-1)/2} \otimes \mathbf{D}^{(n-1)/2}$

So overall we only need to perform $\lg n$ "funny" multiplications of $n \times n$ matrices, instead of n. Each "funny" multiplication, by its definition, can be done in $O(n^3)$ time. So the total running time is $O(n^3 \lg n)$. Quick question: What if $n-1$ is not a power of 2? A more important question: Where did we use the property that $\otimes$ is an associative operation? Actually all the equations above, except the first one, require the associative property (why?).

Can we continue to improve it and get an algorithm with running time $O(n^3)$? The previous naive DP algorithm takes $O(n^4)$ running time because to compute each entry of the 3-dimensional table, the recursive formula needs to compute the minimum of n sums. Next we show a different DP algorithm. While the table is still 3-dimensional and of size n^3, we redefine the meaning of each entry and show that the recursive formula becomes much simpler and each entry can be computed in $O(1)$ time, leading to a $O(n^3)$ DP algorithm: the Floyd-Warshall algorithm. This is a good example where different recursive formulas lead to DP algorithms with different running time.

Here is the most tricky part of Floyd-Warshall: What do we store in the 3-dimensional table? Recall $V = \{1, 2, \ldots, n\} = [n]$. Given $i, j \in n$ and $k : 0 \leq k \leq n$, let $c_{ij}^{(k)}$ denote the weight of a shortest path from i to j in which all intermediate vertices (except i and j themselves) are in the set $\{1, 2, \ldots, k\}$. In particular, for $k = 0$ we have $c_{ij}^{(0)} = w_{ij}$ because the path can only visit i and j. For $k = n$,

$$d_{ij}^{(n)} = \delta(i, j)$$

So the goal of Floyd-Warshall is to compute $d_{ij}^{(n)}$ for all $i, j \in V$.

For any $k \geq 1$, we have the following recursive formula for $c_{ij}^{(k)}$:

$$c_{ij}^{(k)} = \min\left(c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\right)$$

One direction, $c_{ij}^{(k)} \leq \min\left(c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\right)$, is trivial.

We prove the other direction: Let p be a shortest path from i to j for which all intermediate vertices come from $\{1, \ldots, k\}$, then

$$c_{ij}^{(k)} = w(p) \geq \min\left(c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\right)$$

To this end, we consider the following two cases. If k is not an intermediate vertex of p, then by definition we have (why?)

$$w(p) \geq c_{ij}^{(k-1)}$$

If k is actually an intermediate vertex of p, then let p' denote the subpath of p from i to k and let p'' denote the subpath from k to j. Then by definition we have (why?)

$$w(p') \geq c_{ik}^{(k-1)} \quad \text{and} \quad w(p'') \geq c_{kj}^{(k-1)}$$

and thus, we get

$$c_{ij}^{(k)} = w(p) = w(p') + w(p'') \geq c_{ik}^{(k-1)} + c_{kj}^{(k-1)}$$

This finishes the proof of the correctness of the recursive formula. Finally we present the Floyd-Warshall algorithm:

Floyd-Warshall:

1. for all $i, j \in [n]$ do
2. set $c_{ij}^{(0)} = w_{ij}$
3. for k from 1 to n do
4. for i from 1 to n do
5. for j from 1 to n do
6. set $c_{ij}^{(k)} = \min\left(c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\right)$
7. return $\left(c_{ij}^{(n)}\right)$

The running time is $\Theta(n^3)$. Again, the improvement is due to the fact that each entry only takes $O(1)$ time using the new formula.