W2U3:
All Pairs Shortest Path -1

C Pandu Rangan

Recap

- DIJKSTRA's Algorithm

W2U3 Outcomes

- Four algorithms for All pairs shortest path problem.

All Pairs Shortest Path (APSP) - 1

Let $G = (V, E)$ be directed, weighted graph. We assume that G has no negative or zero cycle. For vertices $u, v \in V$ $u \neq v$, define $\delta(u,)$ as the weight of the shorted path from u to v. Define $\delta(u, u) = 0$. It is also useful to extend the weight function to all pairs of $v \times v$ as follows.

$w(u, u) = 0 \ \forall \ u \notin V$

$w(u, v) = \infty$ if $(u, v) \notin E$

If there is no path from u to v in G

We define $\delta(u, v)$ to be ∞

We are interested in finding $\delta(u, v) \ \forall \ u, v \in V$.

Extended Weight Matrix

We will also derive some implicit representation of all paths so that we may use them to obtain the shortest paths explicitly, whenever needed. Since we are interested in a 'Matrix' of values, it is convenient to set $V = \{1,2,3,\cdots,n-1,n\}, n \geq 2.$

Now, the extended weight function may be given in the matrix notation as follows:

$$W = [w_{ij}]_{n\times n}, w_{ij} = 0, i = j$$

$$w_{ij} = \infty \text{ if } (i,j) \notin E,$$

$$w_{ij} = w(i,j) \text{ if } (i,j) \in E$$

The set of all paths from i to j with atmost l edges, $l \geq 0$, is denoted by $P[l: i,j]$.

Shortest Paths Weight Matrix

The weight of the shortest path in $P[l: i, j]$ is denoted by $\delta_{\leq l}(i, j)$.

We use the notation

$$D^l = [\delta_{\leq l}(i, j)] = \left[d_{ij}^l\right]$$

For the $n \times n$ matrix of shortest distance values.

Since any path in G may contain at most $(n-1)$ edges, $P[n-1: u, v]$ denotes the set of ALL paths from u to v and we are interested in $d_{ij}^{(n-1)} \forall i, j \in \{1,2, \cdots, n\}$.

Thus, our goal is to compute $D^{(n-1)}$

Shortest Paths Weight Matrix (contd)

We define $D^0 = [d_{ij}^0]$ where

$d_{ij}^0 = 0$ if $i = j$

$= \infty$ if $i \neq j$

This is because no path exist from i to j with zero edges if $i \neq j$.

From the experience we had in the design of algorithm for SSSP, it is natural to think of computing D^l from D^{l-1}.

Bellman Equations

Bellman equations can be effectively used for the same as follows.

We prove that $d_{ij}^{l} \geq \left(d_{ik}^{l-1} + w_{kj}\right) \forall k$ and

$d_{ij}^{l} = \left(d_{ik}^{l-1} + w_{kj}\right)$ for some k

This would imply,

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

Bellman Equations (contd)

If D^{l-1} values are available, each d_{ij}^{l} can be computed by using n additions and $(n-1)$ comparisons as in equation (1).
In other words, D^{l} can be computed in $O(n^3)$ time if D^{l-1} is available. Since D^{0} is known, we may obtain $D^{1}, D^{2}, \cdots, D^{n-1}$ in that order and the total complexity is $O(n^4)$.

Proof of Bellman Equations

Proof of Bellman Equations (contd)

Proof of Bellman Equations (contd)

Algorithm 1

APSP-1 $(G = (V, E), W)$

$\backslash\backslash V = \{1, 2, \cdots, n\}$

$\backslash\backslash G$ has no negative or zero cycles

$\backslash\backslash W$ is the weight matrix representing extended weight function.

$D^0 = \left[d_{ij}^0\right]$ where

$d_{ij}^0 = 0$ if $i = j$

$= \infty$ if $(i \neq j)$

For $l = 1$ to $n - 1$

Compute D^l using D^{l-1} as shown in Eq. 1

Return D^{n-1}

W2U3:
All Pairs Shortest Path -1
Part 2

C Pandu Rangan

Note

- There is another interesting way to look at the computation of D^l from D^{l-1}. This computation resembles the multiplication of two matrices. At first one might even be surprised that these computations have any resemblance to Matrix Multiplication. A careful look at the computations done from a slightly different point of view will expose the connections.
- Computing Min of n values and sum of n values share the same algebraic properties because bot Min and + are associative and commutative. (In fact abstract algebra is all about studying on operations on abstract data based only on the properties of operations). The operations we perform such as addition on numbers etc are 'concrete instances' of the abstract operations on abstract data.

Relation to Matrix Multiplication

Suppose we interpret the symbol + for 'Min' instead of 'sum'.

That is $a + b = \text{Min}\{a, b\}$.

Under this interpretation

$'5 + 3' = \text{Min}\ \{5,3\} = 3$

Under the usual interpretation of + for 'sum'

$'5 + 3' = 8$

If '+' is interpreted for Min operation, then

$$\sum_{i=1}^{n} a_i = \text{Min}\ \{a_1, a_2, \cdots, a_n\}$$

For sum operation zero works as identity element. That is,

$5 + 0 = 5$

Relation to Matrix Multiplication (contd)

What is the identity element is + interpreted as Min?

Min $\{a, ?\} = a \; \forall a$?

- It is infinity!

Thus, the 'zero element' for Min operation is ∞ !

Strange it may look initially; we will get used to this.

So, Min $\{a_1, a_2, \cdots, a_n\}$ can be written as $a_1 + a_2 + \cdots + a_n$ where '+' is now binary operation of finding Minimum of two numbers.

Thus

$$\text{Min}\{a_1, a_2, \cdots, a_n\} = a_1 + a_2 + \cdots + a_n = \sum_{k=1}^{n} a_k$$

Relation to Matrix Multiplication (contd)

- We will now give a different interpretation for the $\cdot$ operation. Usually, it represents multiplication of numbers and here we will use it for addition of numbers!

Thus $a \cdot b = a + b$

Eg $5 \cdot 7 = 5 + 7 = 12$ under new interpretation.

Let us see how the equation

$$d_{ij}^{l} = \underset{k}{\text{Min}}\left\{d_{ik}^{l-1} + w_{kj}\right\}$$

is transformed under new interpretation.

Relation to Matrix Multiplication (contd)

$$d_{ij}^{l} = \underset{k}{\text{Min}}\{d_{ik}^{l-1} + w_{kj}\}$$
$$= \underset{k}{\text{Min}}\{d_{ik}^{l-1} \cdot w_{kj}\}$$
$$= \sum_{k=1}^{n} d_{ik}^{l-1} \cdot w_{kj}$$-------(3)

Now the resemblance to Matrix Multiplication is clearly visible!

If $C = AB$ where $A = [a_{ij}]_{n\times n}$ and $B = [b_{ij}]_{n\times n}$

Then $C = [c_{ij}]_{n\times n}$ where $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$-------(4)

Relation to Matrix Multiplication (contd)

Based on this exact match in the patterns in 3 and 4, we may write

$D^l = D^{l-1}W$

where $D^{l-1} = \left[d_{ij}^{l-1}\right]_{n\times n}$ and $W = \left[w_{ij}\right]_{n\times n}$

Thus $D_1 = D_0 w$

$D_2 = D_1 w = (D_0 w)w = D_0 w^2$, since Matrix Multiplication is Associative.

Inductively

$D_i = D_0 w^i$

and finally

$D_{n-1} = D_0 W^{n-1}$

As noted earlier this algorithm has $O(n^4)$ complexity.

Squaring Operations

However, if we can directly find W^{n-1} from W in a more efficient way, the equation $D_{n-1} = D_0 W^{n-1}$ would lead to a more efficient algorithms!

In fact, we can compute W^{n-1} by repeated squaring in $\log n$ steps!

For example W^{32} can be computed by computing

$W \longrightarrow W^2 \longrightarrow W^4 \longrightarrow W^8 \longrightarrow W^{16} \longrightarrow W^{32}$

in 5 squaring operations!

In general, W^{2^k} can be computed by k squaring steps.

Squaring Operations (contd)

Finally note that

$D_{n-1} = D_t \; \forall \; t \geq \; n-1$

(Since the longest path may contain at most $n-1$ edges).

Since $D^t = D_0 W^t$ and W^t can be computed easily by squaring when t is a power of 2,

we compute D_{n-1} by computing D_t where $t = 2^k$, and $2^k \geq (n-1)$.

Eg. We may compute $D^{(13)}$ by computing $D^{(16)}$ because $D^{(13)} = D^{(16)}$.

We compute $W^{(16)}$ by computing

$W \longrightarrow W^2 \longrightarrow W^4 \longrightarrow W^8 \longrightarrow W^{16}$

and then compute

$$D^{(16)} = D_0 W^{16}$$

Algorithm 2

- Thus our pseudocode for this efficient version is as follows

1. For $l = 1$ to $\lceil \log n \rceil$

$W = W^2$

$\backslash\backslash W = W^t$ for some $t > (n-1)$

2. $D = D^{(0)}W$

$\backslash\backslash D = D^{(t)}$, for the same $t > (n-1)$

$\backslash\backslash$ Hence $D = D^{(t)} = D^{(n-1)}$

3. Return D

Complexity of Algorithm 2

- Each squaring of $(n \times n)$ matrix takes $O(n^3)$ time. Steps (1) and (2) put together performs $\log n + 1$ matrix multiplications. Hence, the total complexity is $(\log n + 1)n^3 = O(n^3 \log n)$. This is faster than $O(n^4)$ algorithm discussed before.

Generalised BE 1

- The algorithm described earlier are based on the "last edge of the shortest path" or based on the vertex just before the destination.
- We may generalize this to an arbitrary intermediate vertex of a shortest path.
- It is easy to prove that,
- If k is an intermediate vertex in a shortest path P from i to j, the position of the P from i to k as well as the portion of the path P from k to j are shortest paths (from i to k and k to j respectively).

Generalised BE 2

Here is a kind of converse to the statement given above and this is also easy to prove.

Let Q_{ik} be a shortest path from i to k with at most l edges and R_{kj} be a shortest path from k to j with at most l edges.

Let $W(i,j,k)$ be the walk obtained by concatenating Q_{ik} and R_{kj}.

Let $P(i,j)$ be a shortest walk among $\{W(i,j,k), k = 1,2,\ldots,n\}$.

Then,

1) $P(i,j)$ is a path.
2) $P(i,j)$ is a shortest path from i to j.
3) $P(i,j)$ is a shortest path with $\leq 2l$ edges.

Proof of Generalised BE 1 & BE 2

- $P = Q + R$
- Q and R are concatenated at k.

Proof of Generalised BE 1 & BE 2 (contd)

Proof of Generalised BE 1 & BE 2 (contd)

Squaring for Extended BE

$$D_{ij}^{(2l)} = Min\left\{D_{ik}^{(l)} + D_{kj}^{(l)}\right\}$$

In Matrix Multiplication Notation

$$D^{2l} = D^{l} \cdot D^{l} = \left(D^{(l)}\right)^{2}$$

$$D_{ij}^{(1)} = w_{ij} \text{ or } D^{(1)} = w$$

Thus, by a series of squaring operations we obtain a series of Matrices

$$D^{(1)} \rightarrow D^{(2)} \rightarrow D^{(4)} \rightarrow D^{(8)} \rightarrow \cdots D^{(2^i)} \rightarrow \cdots$$

Algorithm 3

We are interested in $D^{(n-1)}$

But $D^{(n-1)} = D^{(l)}$ for all $l \geq (n-1)$,

Since $2^{\log n} \geq n > n-1$,

We conclude that $D^{(n-1)} = D^{(2^{\lceil \log n \rceil})}$

Thus, we perform $\lceil \log n \rceil$ squaring operations and output the resulting Matrix.

$D = W$

(where W is the extended weight matrix)

For $i = l$ to $\lceil \log n \rceil$

$D = D \cdot D$

Return D

$D = D^{(2^{\lceil \log n \rceil})} = D^{(n-1)}$

The complexity is $O(n^3 log n)$

The Algorithm is due to M. Fisher and A. Meyer).

k - Paths

- We will now discuss on $O(n^3)$ algorithm based on a different formulation involving intermediated nodes. (In fact, several researchers have worked with same idea around the same time).
- Call a path from i to j a k-path from i to j if all intermediate nodes are $\leq k$. That is, the path from i to j pass through the set of vertices in $\{1,2,3,\cdots,k\}$.
- k-path is automatically an l-path for all $l > k$. 0-path from i to j is just the edge (i,j), if it exists.
- Note that k is independent of i and j. The nodes i and j are source and destination vertices of the path and upper bound k is applicable only for the intermediate nodes.

k – Paths (contd)

Let $\delta_k(i,j)$ be the weight of shortest k-path from i to j. Since n is the largest vertex label,

$$\delta_k(i,j) = \delta(i,j) \quad \forall\, i,j \in V$$

Note that

$$\delta_o(i,j) = w(i,j) \quad \forall\, i,j \in V$$

Define $A^{(k)} = \left[a_{ij}^{(k)}\right]_{n\times n}$ by $a_{ij}^{(k)} = \delta_k(i,j)$.

The following observation allows us to write $A^{(k)}$ elements in terms of the elements in $A^{(k-1)}$.

A k-path without k

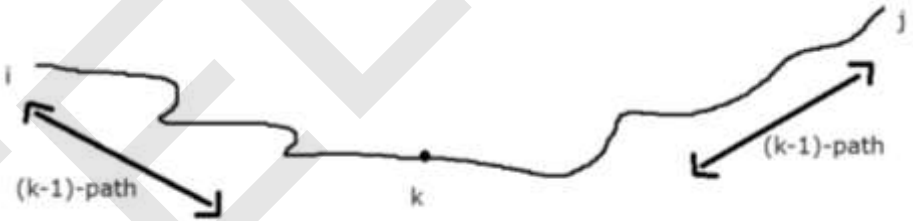
A k-path from i to j may contain k or may not contain k. If it does not contain k, all its intermediate vertices are $\leq (k-1)$ and hence it is in fact a $(k-1)$ path from i to j.

This is a $(k-1)$ path. In this case

$\delta_k(i,j) = \delta_{k-1}(i,j)$ ---(5)

If the k-path from i to j contain k, then the part of the path from i to k and the part of the path from k to j are both $(k-1)$-path, because k can not occur more than once in any path and rest of the internal nodes are all $\leq (k-1)$.

k-path from i to j of weight $\delta_k(i,j)$



The $(k-1)$-path from i to k and the $(k-1)$-path from k to j are shortest paths (by theorm…)

Hence

$\delta_k(i,j) = \delta_{k-1}(i,j) + \delta_{k-1}(k,j)$ ---(6)

In this case,

From equation (5) and (6)

We conclude that

$\delta_k(i,j) = \text{Min}\{\delta_{k-1}(i,j), \delta_{k-1}(i,k) + \delta_{k-1}(k,j)\}$ --- (7)

Complexity for Extension

That is, the computation of $\delta_k(i,j)$ involves referring three elements $\delta_{k-1}(i,j), \delta_{k-1}(i,k)\ and\ \delta_{k-1}(k,j)$ and performing one addition and one comparison and this $O(1)$ computation.

Thus, $A^{(k)}$ matrix values can be determined in $O(n^2)$ time,

if $A^{(k-1)}$ values are available.

Hence, starting from $A^{(0)}$ and computing the sequence of matrices

$A^{(0)} \rightarrow A^{(1)} \rightarrow A^{(2)} \rightarrow \cdots \rightarrow A^{(n-1)} \rightarrow A^{(n)}$

takes $o(n^3)$ time

Algorithm 4 - Floyd-Warshall Algorithm

Floyd-Warshall (G, W).

$A^{(0)} = W$ for $k = 1$ to n

\\Compute $A^{(k)}$ using $A^{(k-1)}$

For $i = 1$ to n

For $j = 1$ to n

$$a^{(k)} a$$

$$a_{ij}^{(k)} = \text{Min}\left\{a_{ij}^{(k-1)}, a_{ik}^{(k-1)}, a_{kj}^{(k-1)}\right\}$$

Return $A^{(n)}$ \\ $a_{ij}^{(n)} = \delta(i, j)$

Johnson's Algorithm

- We will now look at yet another algorithm that is faster for sparse graph
- Floyd-Warshall's algorithm is $O(n^3)$ and the complexity is independent of the number of edges of the graph.
- The algorithm by Johnson runs in $O(n^2 \log n + nm)$ time and when the graph is sparse, this is asymptotically better than $O(n^3)$ algorithm.
- For dense graph with $m = O(n^2)$, the complexity is $O(n^3)$, which is same as Warshalls Algorithm. This algorithm uses a clever transformation technique to achieve improvements.

Johnson's Algorithm (contd)

If all weights are positive, we may apply Dijkstra's algorithm n times (once from each vertex as the source) and the complexity for this algorithm would be in

$$O(n[\log n + m]) = O(n^2 \log n + nm)$$

However, this approach is not applicable if G has some negative edges.

If G has negative edges but no negative cycles, we may apply n times the Bellman-Ford algorithm and the complexity would be $O(n.n.m) = O(n^2 m)$. For Dense graph this may go as high as $O(n^4)$.

Johnson's algorithm deploys a transformation of weights that allowed him to use both Dijkstra's and Bellman-Ford algorithms to exploit the best in both methods.

Weight Transformation

Let $G = (V, E)$ be a directed graph and w be the weight function from edge set to integers.

Let $V = \{1,2, \cdots, n\}$ and h be any function from V to integers.

Define a new weight function w' by

$w'(u, v) = w(u, v) + h(u) - h(v)$

1. P is a shortest path from i to j under w if it is a shortest path under w'
2. For any cycle c in $G, w(c) = w'(c)$
3. For any path P from i to j

$w(P) = w'(P) + h(j) - h(i)$

Basic Idea

- Thus, instead of working on G with weight function w, we may work on G with weight function w'.
- If $w(e) > 0 \ \forall \ e \in E$, we need not transform the weights. We apply Dijkstra's algorithm n times and obtain as algorithm for APSP with complexity $o(n^2 \log n + nm)$.
- If $w(e)$ is negative for some edges, using Bellman and Ford n times leads to a very inefficient $o(n^2 m)$ algorithm. This is the case that requires a transformation of weights.

Basic Idea (contd)

- The trick is,
  Use Bellman-Ford algorithm ONCE and find a $h: V \rightarrow I$ such that $w'(e) > 0 \; \forall \; e \; \in E$.
- Now, Dijkstra's algorithm can be applied n times om G with weight function w' and solve APSP with respect to w'. The same physical paths determined by w' can be used for w, by (1).
- Since $\delta(i,j) = \delta'(i,j) + h(j) - h(i)$ by (3), the APSP weight matrix under w can be constructed from APSP weight matrix under w' in $O(n^2)$ time.

Johnson's Algorithm

The Algorithm at a high level is as follows:

1. Use Bellman-Ford algorithm to determine a $h: V \rightarrow I$ such that $w'(e) > 0 \; \forall \; e \; \in E$ where $w'(i,j) = w(i,j) + h(i) - h(j)$
2. For $i, j \; \forall V, \; i \neq j.$ $w'(i,i) = w(i,i) = 0 \forall i \in V$
3. Solve APSP problem by using Dijkstra's Algorithm n times on G with weight function w'. Let D' be the shortest path weight Matrix obtained.

Johnson's Algorithm (contd)

4. Construct the shortest path weight Matrix D by using the formula

$$\delta(i,j) = \delta'(i,j) + h(j) - h(i)$$

5. Return D.

The total complexity is

$O(mn) + O(n^2 \log n + nm) + n^2$

which is $O(n^2 \log n + nm)$

Thus, Johnson's Algorithm solves APSP problem for a G with no negative cycles in $O(n^2 \log n + nm)$ time.

Construction of h

We now focus on the construction of h and prove that

$w'(i,j) = w(i,j) + h(i) - h(j) \geq 0$

Let $s \notin V$ and construct G' by adding s to V and adding directed edges

$(s,i)\ \forall\ i \in V$ with $w(s,i) = 0$.

That is $G' = (v', E')$ where $v' = V \cup \{s\}$

$E' = E \cup \{(s,i) | i \epsilon V\}$

Solve SSSP problem on G' with s as a source and define

$h(i) = \delta(s,i)$ in E, (which is also in E').

Weight Transformation

We know that

$\delta(i) \leq \delta(i) + w(i,j)$

Hence,

$w(i,j) + \delta(i) - \delta(i) \geq 0$

Implying

$w(i,j) + h(i) - h(i) \geq 0$

Thus

$w'(i,j) = w(i,j) + h(i) - h(j) \geq 0$ for all $(i,j) \in E$

This completes our discussions on Jhonsons Algorithm

Remark

- If G has a cycle with negative weight then G' also will have the same cycle as a negative weight cycle. Thus, if Bellman-Ford algorithm working on G' reports a negative cycle in G', we report G has a negative cycle and simply terminate the algorithm at this point. We will proceed with further steps only when we know that G has no negative cycles.

W3U1

- Minimum Spanning Trees

Thank You