All-Pairs Shortest Paths

Chapter 25

Lecture Overview

- All-Pairs Shortest Path Problem
- APSP by basic Dynamic Programming
- Floyd-Warshall Algorithm
- Transitive Closure

All-Pairs via Single-Source

The All-Pairs Shortest Paths (APSP) problem is to find shortest paths (and/or their distances) between every pair of vertices in a given graph. We typically want the output in tabular (matrix) form.

We can solve an APSP problem by running a SSSP algorithm |V| times, once for each vertex as the source.

If all edge weights are nonnegative, we can use Dijkstra's algorithm.

prioirity queue	APSP running time
linear array	$O(V^3 + VE) = O(V^3)$
binary heap	$O(VE \log V)$
Fibonacci heap	$O(VE + V^2 \log V)$

All-Pairs Shortest Paths

If negative edge weights are allowed, Dijkstra's algorithm can no longer be used. Instead, we run the slower Bellman-Ford algorithm from each vertex, giving $O(V^2E)$ time, which can be (in particularly dense graphs) as bad as $O(V^4)$.

We will see how to do better than these preliminary results that use SSSP. Most of our algorithms, however, will use an adjacency-matrix representation rather than the adjacency-list representation that we have been using.

For convenience, we will assume that the vertices are 1, 2, ... , |V|, so the input is an n × n matrix W representing the edge weights. So

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

All-Pairs Shortest Paths

The matrix output by our algorithms is $D = (d_{ij})$ where entry d_{ij} is the weight of the shortest path from vertex i to vertex j. During the algorithm, the entries may hold other values than the shortest-path weight.

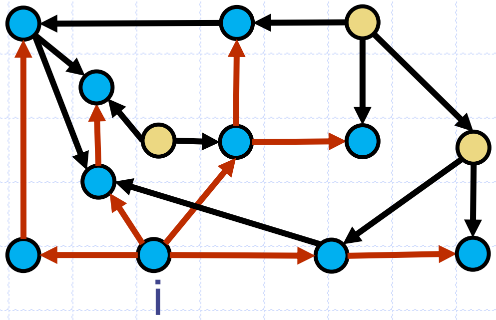
We will also compute a predecessor matrix $\Pi = (\pi_{ij})$ where π_{ij} is NIL if either i = j or there is no path from i to j, and otherwise π_{ij} is the predecessor of j on some shortest path from vertex i to vertex j.

Similar to the predecessor subgraph of SSSP, we define a predecessor subgraph for source i as $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$ where

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} + i$$

and

$$E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} - i\}$$



All-Pairs Shortest Paths

We start with a dynamic-programming algorithm for the all-pairs shortest paths problem. A main operation of this algorithm is something that is akin to matrix multiplication. We start by developing an $O(V^4)$-time algorithm and then improve that to $O(V^3 \log V)$.

Characterizing the structure of an optimal solution. We already know that all subpaths of a shortest path are shortest paths. If k is the predecessor of j on a shortest path from i to j, then

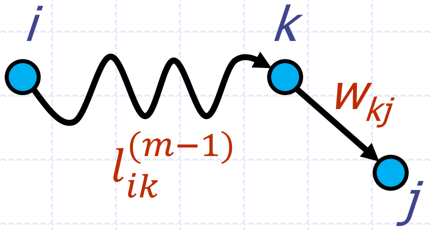
$$\delta(i, j) = \delta(i, k) + w_{kj}.$$

All-Pairs Shortest Paths

Recurively defining the value of an optimal solution. Let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges.

$$l_{ij}^{(m)} = 0 \text{ if } i = j$$

$$l_{ij}^{(0)} = \infty \text{ if } i \neq j$$



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

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

Since shortest paths contain at most n-1 edges,

$\delta(i, j) = l_{ij}^{(n-1)}$ and $l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \ldots$

All-Pairs Shortest Paths

Computing the value of an optimal solution bottom-up. From our input matrix $W = (w_{ij})$, we compute matrices $L^{(1)}, L^{(2)}, \ldots L^{(n-1)}$, where $L^{(m)} = (l_{ij}^{(m)})$. $L^{(n-1)}$ will contain the shortest-path lengths. Since $l_{ij}^{(1)} = w_{ij}$ for all $i, j \in V$, $L^{(1)} = W$.

The critical part of the algorithm is the following routine, which will, given the matrices $L^{(m-1)}$ and W, return the matrix $L^{(m)}$. That is, it extends the shortest paths computed so far by one more edge.

Extend Shortest Paths

EXTEND-SHORTEST-PATHS(L, W)

```
n = L.numRows()
allocate matrix L' as n × n.
for i = 1 to n
    for j = 1 to n
        $l'_{ij} = \infty$
        for k = 1 to n
            $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$
return L'
```

$O(n^3)$

Look at what happens when we change:

$L \to A$, $W \to B$, $L' \to C$, $+ \to \cdot$, $\min \to +$, $\infty \to 0$

Matrix Multiply

```
MATRIX-MULTIPLY(A, B)
n = A.numRows()
allocate matrix C as $n \times n$.
for i = 1 to n
    for j = 1 to n
        $c_{ij} = 0$
        for k = 1 to n
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return L'
```

We get the straightforward matrix multiply routine for square matrices.

Treating EXTEND-SHORTEST-PATHS as a Multiplication

- We return to APSP. Let $A \otimes B$ denote the matrix "product" returned by EXTEND-SHORTEST-PATHS(A, B), and $A^{[k]}$ denote $A \otimes A \otimes \ldots \otimes A$, where there are k A's in the product.

$L^{(1)} = L^{(0)} \otimes W = W$

$L^{(2)} = L^{(1)} \otimes W = W^{[2]}$

$L^{(3)} = L^{(2)} \otimes W = W^{[3]}$

…

$L^{(n-1)} = L^{(n-2)} \otimes W = W^{[n-1]}$

Note that $L^{(n-1)} = W^{[n-1]}$ is our solution.

A Slow (But Correct) APSP

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1. n = W.numRows()
2. $L^{(1)} = W$
3. for m = 2 to n-1
4.     $L^{(m)}$ = EXTEND-SHORTEST-PATHS($L^{(m-1)}$, W)
5. return $L^{(n-1)}$
```

We can improve this by noting that we really only need to compute $L^{(n-1)}$, not all $L^{(m)}$. A common strategy for doing this is repeated squaring.

Repeated Squaring

We wish to compute $L^{(n-1)} = W^{[n-1]}$. To do this, we're going to repeatedly "square" W.

$L^{(1)} = W$
$L^{(2)} = W^{[2]} = W \otimes W$
$L^{(4)} = W^{[4]} = W^{[2]} \otimes W^{[2]}$
$L^{(8)} = W^{[8]} = W^{[4]} \otimes W^{[4]}$
...
$L^{(2^k)} = W^{[2^k]} = W^{[2^{k-1}]} \otimes W^{[2^{k-1}]}$

works only if $\otimes$ is associative (which it is).

We go until the smallest k such that $2^k \geq n - 1$, or $k = \lceil \log(n-1) \rceil$. (Recall that $L^{(p)} = L^{(n-1)}$ for $p \geq n - 1$.)

A Faster APSP

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
1. n = W.numRows()
2. $L^{(1)} = W$
3. m = 1
4. while m < n-1
5.     $L^{(2m)}$ = EXTEND-SHORTEST-PATHS($L^{(m)}$, $L^{(m)}$)
6.     m = 2m
7. return $L^{(m)}$
```

Noting that the loop of lines 4-6 iterates O(log n) times, we get a total time of $O(n^3 \log n)$.

A Different DP Formulation

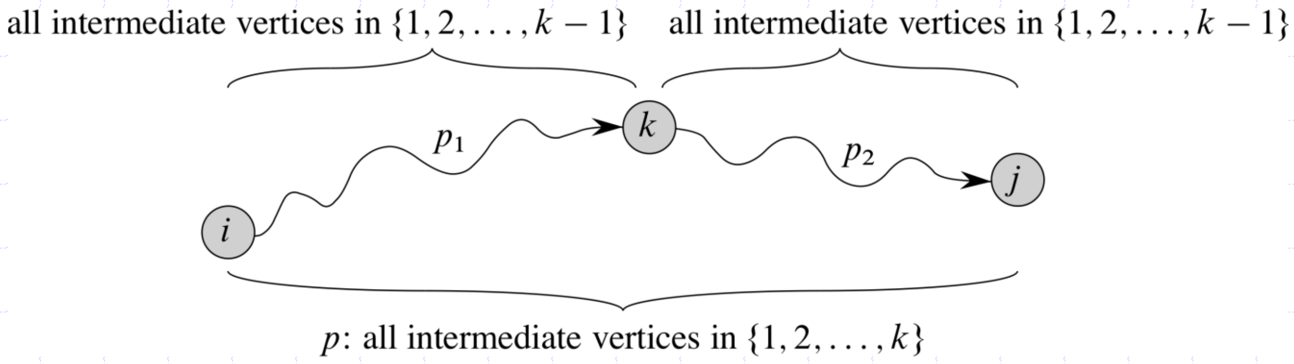
We now develop a different approach to APSP, but it is still a dynamic programming solution. We allow negative-weight edges but no negative-weight cycles. Our algorithm will run in $O(V^3)$ time.

The intermediate vertices of a shortest path $p = <v_1, v_2, \ldots, v_l>$ are the vertices $v_2, v_3, \ldots, v_{l-1}$.

Assume $V = \{1, 2, \ldots, n\}$ and consider a subset $V_k = \{1, 2, \ldots k\}$ of V for some k. We consider the shortest paths where all of the intermediate vertices are drawn from V_k. Let $d_{ij}^{(k)}$ represent the shortest path length from i to j using intermediate vertices only from V_k.

A Different DP Formulation

A shortest path p from i to j with intermediate vertices from V_k can either use the vertex k or not. If it does not use vertex k, then its length is $d_{ij}^{(k-1)}$. If it does use vertex k, then it uses it only once, because a shortest path has no cycles. Now p can be broken into the part before vertex k and the part after vertex k. The length of the former is $d_{ik}^{(k-1)}$ and the length of the latter is $d_{kj}^{(k-1)}$.



A Different DP Formulation

If k = 0, then there are no intermediate vertices and the length of the shortest path from i to j under these circumstances is simply w_{ij}. In summary:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k > 0 \end{cases}$$

We can now compute $d_{ij}^{(k)}$ in a bottom-up fashion:

Floyd-Warshall Algorithm

```
FLOYD-WARSHALL(W)
n = W.numRows()                                                        $O(1)$
$D^{(0)} = W$                                                          $O(1)$ or $O(n^2)$
for k = 1 to n
    for i = 1 to n                                                     $O(n^3)$ iter.
        for j = 1 to n
            $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$      $O(1)$
return $D^{(n)}$                                                       $O(1)$
```

$O(n^3)$ time

Floyd-Warshall Shortest Paths

FLOYD-WARSHALL is a method of constructing shortest path distances but doesn't say how to get the shortest paths themselves. It turns out that there are a variety of different methods for doing so that do not increase the complexity.

The first method of computing shortest paths is to compute the matrix D of shortest-path distances and then to compute the predecessor matrix Π from D itself. A good exercise is to give an algorithm to do this that runs in $O(n^3)$ time.

Floyd-Warshall Shortest Paths

A second method is to compute Π as Floyd-Warshall computes the matrices $D^{(k)}$. We compute matrices $\Pi^{(k)}$ where $\pi_{ij}^{(k)}$ is the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in V_k.

Here is our recursive formulation of $\pi_{ij}^{(k)}$:

$$\pi_{ij}^{(0)} = \begin{cases} NIL & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & otherwise \end{cases}$$

Exercise 25.2-7 of your text introduces yet another way of computing shortest paths in Floyd-Warshall.

Transitive Closure

The transitive closure of a directed graph G = (V, E) is defined as the graph G* = (V, E*) where

E* = {(i, j) : there is a path from i to j in G}.

One way to compute the transitive closure of a graph is to assign a weight of 1 to each edge and then to run Floyd-Warshall on it. If there is a path from i to j, we get $d_{ij} < n$. Otherwise, we get $d_{ij} = \infty$.

There is a similar way that involves changing the min and + in Floyd-Warshall to logical OR and logical AND. This can save time and space in practice, by requiring only boolean (1-bit) matrix entries and the simpler logical operations.

Transitive Closure

We can formulate transitive closure recursively by defining t_{ij}^{k} to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in V_k, and to be 0 otherwise. Then we get

$$t_{ij}^{(0)} = \begin{cases} 1 & \text{if } i = j \text{ or } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

and

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

Transitive Closure

```
TRANSITIVE-CLOSURE(G)
n = |G.vertices|
for i = 1 to n
    for j = 1 to n
        if i = j or (i, j) ∈ G.edges                    O(n^2)
            $t_{ij}^{(0)} = 1$
        else $t_{ij}^{(0)} = 0$
for k = 1 to n
    for i = 1 to n
        for j = 1 to n                                   O(n^3)
            $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$
return T^(n)
```