

Computer Science & Engineering 423/823

Design and Analysis of Algorithms

Lecture 06 — All-Pairs Shortest Paths (Chapter 25)

Stephen Scott
(Adapted from Vinodchandran N. Variyam)

sscott@cse.unl.edu

Introduction

- ▶ Similar to SSSP, but find shortest paths for all pairs of vertices
- ▶ Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$, find $\delta(u, v)$ for all $(u, v) \in V \times V$
- ▶ One solution: Run an algorithm for SSSP $|V|$ times, treating each vertex in V as a source
 - ▶ If no negative weight edges, use Dijkstra's algorithm, for time complexity of $O(|V|^3 + |V||E|) = O(|V|^3)$ for array implementation, $O(|V||E| \log |V|)$ if heap used
 - ▶ If negative weight edges, use Bellman-Ford and get $O(|V|^2|E|)$ time algorithm, which is $O(|V|^4)$ if graph dense
- ▶ Can we do better?
 - ▶ Matrix multiplication-style algorithm: $\Theta(|V|^3 \log |V|)$
 - ▶ Floyd-Warshall algorithm: $\Theta(|V|^3)$
 - ▶ Both algorithms handle negative weight edges

Adjacency Matrix Representation

- ▶ Will use adjacency matrix representation
- ▶ Assume vertices are numbered: $V = \{1, 2, \dots, n\}$
- ▶ Input to our algorithms will be $n \times n$ matrix W :

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

- ▶ For now, assume negative weight cycles are absent
- ▶ In addition to distance matrices L and D produced by algorithms, can also build *predecessor matrix* Π , where π_{ij} = predecessor of j on a shortest path from i to j , or NIL if $i = j$ or no path exists
 - ▶ Well-defined due to optimal substructure property

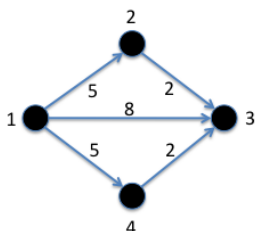
Print-All-Pairs-Shortest-Path(Π, i, j)

```

1 if i == j then
2   print i
3 else if  $\pi_{ij} == \text{NIL}$  then
4   print "no path from " i " to " j " exists"
5 else
6   PRINT-ALL-PAIRS-SHORTEST-PATH( $\Pi, i, \pi_{ij}$ )
7   print j
8
```

Shortest Paths and Matrix Multiplication

- ▶ Will maintain a series of matrices $L^{(m)} = (\ell_{ij}^{(m)})$, where $\ell_{ij}^{(m)}$ = the minimum weight of any path from i to j that uses at most m edges
 - ▶ Special case: $\ell_{ij}^{(0)} = 0$ if $i = j$, ∞ otherwise



$$\ell_{13}^{(0)} = \infty, \ell_{13}^{(1)} = 8, \ell_{13}^{(2)} = 7$$

Recursive Solution

- ▶ Exploit optimal substructure property to get a recursive definition of $\ell_{ij}^{(m)}$
- ▶ To follow shortest path from i to j using at most m edges, either:
 1. Take shortest path from i to j using $\leq m-1$ edges and stay put, or
 2. Take shortest path from i to some k using $\leq m-1$ edges and traverse edge (k, j)

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

- ▶ Since $w_{jj} = 0$ for all j , simplify to

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

- ▶ If no negative weight cycles, then since all shortest paths have $\leq n-1$ edges,

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

Bottum-Up Computation of L Matrices

- ▶ Start with weight matrix W and compute series of matrices $L^{(1)}, L^{(2)}, \dots, L^{(n-1)}$
- ▶ Core of the algorithm is a routine to compute $L^{(m+1)}$ given $L^{(m)}$ and W
- ▶ Start with $L^{(1)} = W$, and iteratively compute new L matrices until we get $L^{(n-1)}$
 - ▶ Why is $L^{(1)} = W$?
- ▶ Can we detect negative-weight cycles with this algorithm? How?

Extend-Shortest-Paths(L, W)

```

1  n = number of rows of L      // This is  $L^{(m)}$ 
2  create new  $n \times n$  matrix  $L'$  // This will be  $L^{(m+1)}$ 
3  for i = 1 to n do
4      for j = 1 to n do
5           $\ell'_{ij} = \infty$ 
6          for k = 1 to n do
7               $\ell'_{ij} = \min(\ell'_{ij}, \ell_{ik} + w_{kj})$ 
8          end
9      end
10 end
11 return  $L'$ 

```

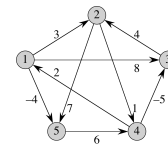
Slow-All-Pairs-Shortest-Paths(W)

```

1  n = number of rows of W
2   $L^{(1)} = W$ 
3  for m = 2 to n - 1 do
4       $L^{(m)} = \text{EXTEND-SHORTEST-PATHS}(L^{(m-1)}, W)$ 
5  end
6  return  $L^{(n-1)}$ 

```

Example



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 2 & -4 & 7 \\ \infty & 0 & \infty & 1 & \infty \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Improving Running Time

- ▶ What is time complexity of SLOW-ALL-PAIRS-SHORTEST-PATHS?
- ▶ Can we do better?
- ▶ Note that if, in EXTEND-SHORTEST-PATHS, we change $+$ to multiplication and \min to \odot , get matrix multiplication of L and W
- ▶ If we let \odot represent this "multiplication" operator, then SLOW-ALL-PAIRS-SHORTEST-PATHS computes

$$\begin{aligned}
 L^{(2)} &= L^{(1)} \odot W = W^{\odot 2}, \\
 L^{(3)} &= L^{(2)} \odot W = W^{\odot 3}, \\
 &\vdots \\
 L^{(n-1)} &= L^{(n-2)} \odot W = W^{\odot n-1}
 \end{aligned}$$

- ▶ Thus, we get $L^{(n-1)}$ by iteratively "multiplying" W via EXTEND-SHORTEST-PATHS

Improving Running Time (2)

- ▶ But we don't need every $L^{(m)}$; we only want $L^{(n-1)}$
- ▶ E.g. if we want to compute 7^{64} , we could multiply 7 by itself 64 times, or we could square it 6 times
- ▶ In our application, once we have a handle on $L^{((n-1)/2)}$, we can immediately get $L^{(n-1)}$ from one call to EXTEND-SHORTEST-PATHS($L^{((n-1)/2)}, L^{((n-1)/2)}$)
- ▶ Of course, we can similarly get $L^{((n-1)/2)}$ from "squaring" $L^{((n-1)/4)}$, and so on
- ▶ Starting from the beginning, we initialize $L^{(1)} = W$, then compute $L^{(2)} = L^{(1)} \odot L^{(1)}$, $L^{(4)} = L^{(2)} \odot L^{(2)}$, $L^{(8)} = L^{(4)} \odot L^{(4)}$, and so on
- ▶ What happens if $n - 1$ is not a power of 2 and we "overshoot" it?
- ▶ How many steps of repeated squaring do we need to make?
- ▶ What is time complexity of this new algorithm?

Faster-All-Pairs-Shortest-Paths(W)

```

1   $n$  = number of rows of  $W$ 
2   $L^{(1)} = W$ 
3   $m = 1$ 
4  while  $m < n - 1$  do
5     $L^{(2m)} = \text{EXTEND-SHORTEST-PATHS}(L^{(m)}, L^{(m)})$ 
6     $m = 2m$ 
7  end
8  return  $L^{(m)}$ 

```

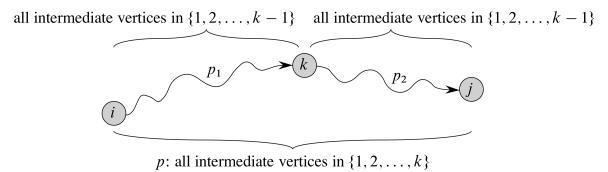
Floyd-Warshall Algorithm

- Shaves the logarithmic factor off of the previous algorithm
- As with previous algorithm, start by assuming that there are no negative weight cycles; can detect negative weight cycles the same way as before
- Considers a different way to decompose shortest paths, based on the notion of an *intermediate vertex*
 - If simple path $p = \langle v_1, v_2, v_3, \dots, v_{\ell-1}, v_{\ell} \rangle$, then the set of intermediate vertices is $\{v_2, v_3, \dots, v_{\ell-1}\}$

Structure of Shortest Path

- Again, let $V = \{1, \dots, n\}$, and fix $i, j \in V$
- For some $1 \leq k \leq n$, consider set of vertices $V_k = \{1, \dots, k\}$
- Now consider all paths from i to j whose intermediate vertices come from V_k and let p be a minimum-weight path from them
- Is $k \in p$?
 1. If not, then all intermediate vertices of p are in V_{k-1} , and a SP from i to j based on V_{k-1} is also a SP from i to j based on V_k
 2. If so, then we can decompose p into $i \xrightarrow{p_1} k \xrightarrow{p_2} j$, where p_1 and p_2 are each shortest paths based on V_{k-1}

Structure of Shortest Path (2)



Recursive Solution

- What does this mean?
- It means that a shortest path from i to j based on V_k is either going to be the same as that based on V_{k-1} , or it is going to go through k
- In the latter case, a shortest path from i to j based on V_k is going to be a shortest path from i to k based on V_{k-1} , followed by a shortest path from k to j based on V_{k-1}
- Let matrix $D^{(k)} = (d_{ij}^{(k)})$, where $d_{ij}^{(k)}$ = weight of a shortest path from i to j based on V_k :

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

- Since all SPs are based on $V_n = V$, we get $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$

Floyd-Warshall(W)

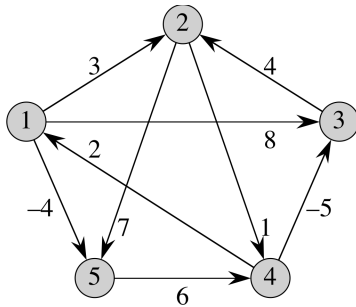
```

1   $n$  = number of rows of  $W$ 
2   $D^{(0)} = W$ 
3  for  $k = 1$  to  $n$  do
4    for  $i = 1$  to  $n$  do
5      for  $j = 1$  to  $n$  do
6         $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
7      end
8    end
9  end
10 return  $D^{(n)}$ 

```

Floyd-Warshall Example

Split into teams, and simulate Floyd-Warshall on this example:



Navigation icons: back, forward, search, etc.

Transitive Closure

- Used to determine whether paths exist between pairs of vertices
- Given directed, unweighted graph $G = (V, E)$ where $V = \{1, \dots, n\}$, the *transitive closure* of G is $G^* = (V, E^*)$, where

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

- How can we directly apply Floyd-Warshall to find E^* ?
- Simpler way: Define matrix T similarly to D :

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

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

- I.e., you can reach j from i using V_k if you can do so using V_{k-1} or if you can reach k from i and reach j from k , both using V_{k-1}

Navigation icons: back, forward, search, etc.

Transitive-Closure(G)

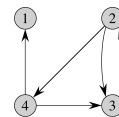
```

1 allocate and initialize  $n \times n$  matrix  $T^{(0)}$ 
2 for  $k = 1$  to  $n$  do
3   allocate  $n \times n$  matrix  $T^{(k)}$ 
4   for  $i = 1$  to  $n$  do
5     for  $j = 1$  to  $n$  do
6        $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$ 
7     end
8   end
9 end
10 return  $T^{(n)}$ 

```

Navigation icons: back, forward, search, etc.

Example



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Navigation icons: back, forward, search, etc.

Analysis

- Like Floyd-Warshall, time complexity is officially $\Theta(n^3)$
- However, use of 0s and 1s exclusively allows implementations to use bitwise operations to speed things up significantly, processing bits in batch, a word at a time
- Also saves space
- Another space saver: Can update the T matrix (and F-W's D matrix) in place rather than allocating a new matrix for each step (Exercise 25.2-4)

Navigation icons: back, forward, search, etc.