All-Pairs Shortest Paths (Ch. 25)

The all-pairs shortest path problem (APSP)
input: a directed graph G = (V, E) with edge weights
goal: find a minimum weight (shortest) path between every pair of vertices in V
Can we do this with algorithms we've already seen?

Solution 1: run Dijkstra's algorithm V times, once with each $v \in V$ as the source node (requires no negative-weight edges in E)
If G is dense with an array implementation of Q
$O(V \cdot V^2) = O(V^3)$ time
If G is sparse with a binary heap implementation of Q
$O(V \cdot ((V + E) \log V)) = O(V^2 \log V + VE \log V)$ time

All-Pairs Shortest Paths

Solution 2: run the Bellman-Ford algorithm V times (negative edge weights allowed), once from each vertex.
$O(V^2E)$, which on a dense graph is $O(V^4)$

Solution 3: Use an algorithm designed for the APSP problem.

E.g., Floyd's Algorithm
introduces a *dynamic programming* technique that
uses adjacency matrix representation of G = (V, E)

Warshall's Transitive Closure Algorithm

Input: Adjacency matrix A of G as matrix of 1s and 0's
Output: Transitive Closure or reachability matrix $R^{(n)}$ of G

Assumes vertices are numbered 1 to |V|, |V| = n and there are no edge weights. Finds a series of boolean matrices $R^{(0)}$, …, $R^{(n)}$

Solution for $R^{(n)}$:
Define $r_{ij}^{(k)}$ as the element in the ith row and jth column to be 1 iff there is a path between vertices i and j using only vertices numbered ≤ k.
$R^{(0)}$ = A, original adjacency matrix (only 1's are direct edges)
$R^{(n)}$ the matrix we want to compute

$R^{(k)}$'s elements are: $R^{(k)}[i, j] = r_{ij}^{(k)} = r_{ij}^{(k-1)} \vee r_{ik}^{(k-1)} \wedge r_{kj}^{(k-1)}$

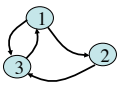
Warshall's Algorithm

Warshall (A[1…n,1…n])
1. n = rows[A]
2. $R^{(0)}$ = A
3. for k = 1 to n do
4. for i = 1 to n do
5. for j = 1 to n do
6. $R_{ij}^{(k)} = R_{ij}^{(k-1)} \vee R_{ik}^{(k-1)} \wedge R_{kj}^{(k-1)}$
7. return $R^{(n)}$

$R_{ij}^{(0)} = 0$ if i = j
$= \infty$ if i != j

If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in R(k).
If an element r_{ij} is 0 in $R^{(k-1)}$, it becomes a 1 iff the element in row i and column k and the element in column j, row k are both 1's in $R^{(k-1)}$.

Warshall's Algorithm



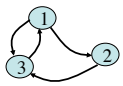
$R^{(0)} = A =$

0	1	1
0	0	1
1	0	0

Matrix $R^{(0)}$ contains the nodes reachable in one hop

For $R^{(1)}$, there is a 1 in row 3, col 1 and col 2, row 1, so put a 1 in position 3,2. Also, there is a 1 in row 3, col 1, and col 3, row 1 so put a 1 in position 3,3

Warshall's Algorithm



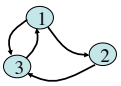
$R^{(1)} =$

0	1	1
0	0	1
1	1	1

Matrix $R^{(1)}$ contains the nodes reachable in one hop or on paths that go through vertex 1.

For $R^{(2)}$, there is no change because 1 can get to 3 through 2 but there is already a direct path between 1 and 3.

Warshall's Algorithm



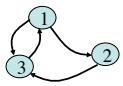
$R^{(2)} =$

0	1	1
0	0	1
1	1	1

Matrix $R^{(2)}$ contains the nodes reachable in one hop or on paths that go through vertices 1 or 2.

For $R^{(3)}$, there is a 1 in row 1, col 3 and col 1, row 3, so put a 1 in position 1,1. Also, there is a 1 in row 2, col 3 and col 1, row 3, so put a 1 in position 2,1. Also, there is a 1 in row 2, col 3 and col 2, row 3, so put a 1 in position 2,2.

Warshall's Algorithm



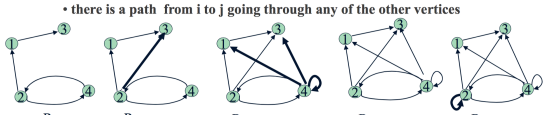
$R^{(3)} =$

1	1	1
1	1	1
1	1	1

Matrix $R^{(3)}$ contains the vertices reachable in one hop or on paths that go through vertices 1, 2, and 3.

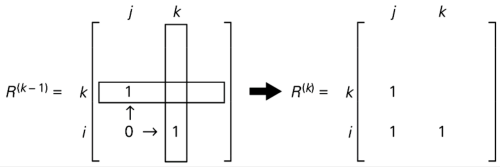
Warshall's Algorithm

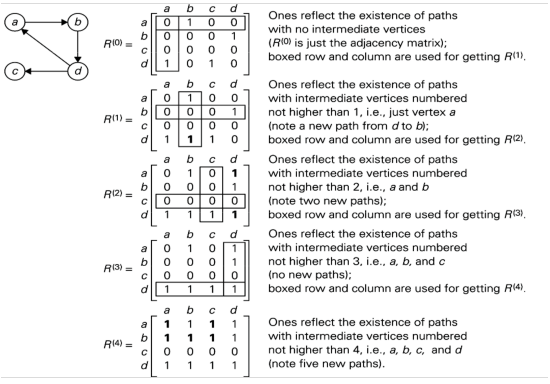
- **Main idea: a path exists between two vertices i, j, iff**
  - **there is an edge from i to j; or**
  - **there is a path from i to j going through vertex 1; or**
  - **there is a path from i to j going through vertex 1 and/or 2; or**
  - **there is a path from i to j going through vertex 1, 2, and/or 3; or**
  - **...**
  - **there is a path from i to j going through any of the other vertices**



R_0	R_1	R_2	R_3	R_4
0 0 1 0	0 0 1 0	0 0 1 0	0 0 1 0	0 0 1 0
1 0 0 1	1 0 **1** 1	1 0 1 1	1 0 1 1	1 **1** 1 1
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
0 1 0 0	0 1 0 0	**1** 1 **1** 1	1 1 1 1	1 1 1 1

Warshall's Algorithm: Transitive Closure





Warshall's Algorithm

Warshall (A[1…n,1…n])

1. n = rows[A]
2. $R^{(0)} = A$
3. for k = 1 to n do
4. for i = 1 to n do
5. for j = 1 to n do
6. $R_{ij}^{(k)} = R_{ij}^{(k-1)} \vee R_{ik}^{(k-1)} \wedge R_{kj}^{(k-1)}$
7. return $R^{(n)}$

Time efficiency?

Space efficiency?

Floyd's APSP Algorithm

Input: Adjacency matrix A
Output: Shortest path matrix $D^{(n)}$
and predecessor matrix $\Pi^{(n)}$

Observation: When G contains no negative-weight cycles, all shortest paths consist of at most n – 1 edges

Assumes vertices are numbered 1 to |V|

Relies on the *Optimal Substructure Property*:
All sub-paths of a shortest path are shortest paths.

Solution for D:

Define $D^{(k)}[i, j] = d_{ij}^{(k)}$ as the minimum weight of any path from vertex i to vertex j, such that all intermediate vertices are in {1, 2, 3, ..., k}

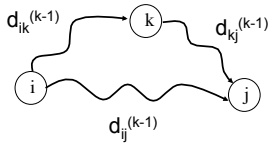
$D^{(0)}$ = A, original adjacency matrix (only paths are single edges)

$D^{(n)}$ the matrix we want to compute

$D^{(k)}$'s elements are: $D^{(k)}[i, j] = d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

Recursive Solution for $D^{(k)}$

$$D^{(k)}[i, j] = d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$



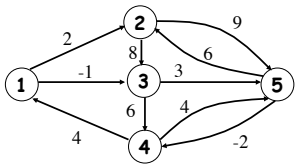
The only intermediate nodes on the paths from i to j, i to k or k to j are in the set of vertices {1, 2, 3, ..., k-1}.
If k is included in shortest i to j path, then a shortest path has been found that includes k.
If k is not included in shortest i to j path, then the shortest path still only includes vertices in the set 1…k-1.

Initial Matrix of Path Lengths

Use adjacency matrix A for G = (V, E):

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

	1	2	3	4	5
1	0	2	-1	∞	∞
2	∞	0	8	∞	9
3	∞	∞	0	6	3
4	4	∞	∞	0	4
5	∞	6	∞	-2	0

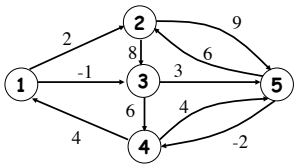


Initial Matrix of Predecessors

Use adjacency matrix Π to keep track of predecessors:

$$\pi^{(0)}_{ij} = \begin{cases} i & \text{if } i \neq j \text{ and } w(i,j) < \infty \\ \varnothing & \text{if } i = j \text{ or } w(i, j) = \infty \end{cases}$$

	1	2	3	4	5
1	Ø	1	1	Ø	Ø
2	Ø	Ø	2	Ø	2
3	Ø	Ø	Ø	3	3
4	4	Ø	Ø	Ø	4
5	Ø	5	Ø	5	Ø



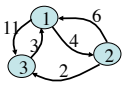
π_{ij} is predecessor of j on some shortest path from i

Floyd's APSP Algorithm

Floyd-Warshall-APSP(A)

```
1.  n = rows[A]
2.  D(0) = A
3.  for k = 1 to n
4.      for i = 1 to n
5.          for j = 1 to n
6.              if dij(k) > dik(k-1) + dkj(k-1)
7.                  dij(k) = dik(k-1) + dkj(k-1)
8.                  πij(k) = πkj(k-1)
9.              else πij(k) = πij(k-1)
10. return D(n)
```

Operation of F-APSP Algorithm



$D^{(0)} = A =$

0	4	11
6	0	2
3	∞	0

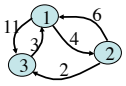
7

$\Pi^{(0)} =$

Ø	1	1
2	Ø	2
3	Ø	Ø

1

Operation of F-APSP Algorithm



$\Pi^{(1)} =$

∅	1	1
2	∅	2
3	1	∅

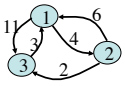
2

$D^{(1)} =$

0	4	11
6	0	2
3	7	0

6

Operation of F-APSP Algorithm



$D^{(2)} =$

0	4	6
6	0	2
3	7	0

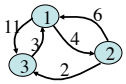
5

$\Pi^{(2)} =$

∅	1	2
2	∅	2
3	1	∅

3

Printing the shortest path



$\Pi^{(3)} =$

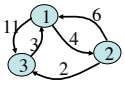
∅	1	2
3	∅	2
3	1	∅

```
Print-APSP(p, i, j) // p is predecessor matrix π
1. if i = j
2.     print i
3. else if p[ i ][ j ] = NIL
4.     print "no path from "+ i + " to "+ j + " exists"
5. else
6.     Print-APSP(p, i, p[ i ][ j ])
7.     print j
```

$D^{(3)}$

0	4	6
5	0	2
3	7	0

Operation of F-APSP Algorithm



$D^{(0)} = A =$

0	4	11
6	0	2
3	∞	0

$D^{(1)}$

0	4	11
6	0	2
3	7	0

3→ 1 → 2

$D^{(2)}$

0	4	6
6	0	2
3	7	0

1→ 2 → 3

$D^{(3)}$

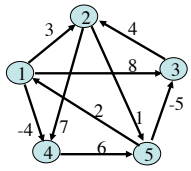
0	4	6
5	0	2
3	7	0

2→ 3 → 1

F-APSP Algorithm

$D^{(0)} = A =$

0	3	8	-4	∞
∞	0	∞	7	1
∞	4	0	∞	∞
∞	∞	∞	0	6
2	∞	-5	∞	0



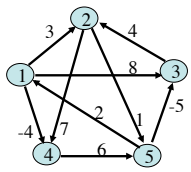
$D^{(1)} =$

0	3	8	-4	∞
∞	0	∞	7	1
∞	4	0	∞	∞
∞	∞	∞	0	6
2	5	-5	-2	0

F-APSP Algorithm

$D^{(2)} =$

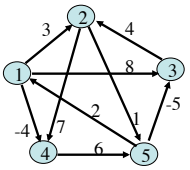
0	3	8	-4	4
∞	0	∞	7	1
∞	4	0	11	5
∞	∞	∞	0	6
2	5	-5	-2	0



F-APSP Algorithm

$D^{(3)} =$

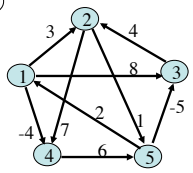
0	3	8	-4	4
∞	0	∞	7	1
∞	4	0	11	5
∞	∞	∞	0	6
2	-1	-5	-2	0



F-APSP Algorithm

$D^{(4)} =$

0	3	8	-4	2
∞	0	∞	7	1
∞	4	0	11	5
∞	∞	∞	0	6
2	-1	-5	-2	0

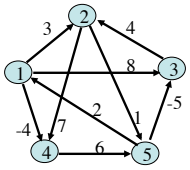


F-APSP Algorithm

$D^{(5)} =$

0	1	-3	-4	2
3	0	-4	-1	1
7	4	0	3	5
8	5	1	0	6
2	-1	-5	-2	0

1→4→5→3, 1→4→5→3→2
2→5→1, 2→5→3, 2→5→1→4
3→2→5→1, 3→2→5→1→4
4→5→1, 4→5→3→2, 4→5→3



Running Time of Floyd's-APSP

Lines 3 – 6: $|V^3|$ time for triply-nested **for** loops

Overall running time = = $\theta(V^3)$

The code is tight, with no elaborate data structures and so the constant hidden in the θ-notation is small.