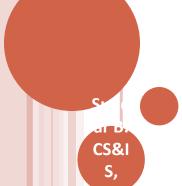
# CS F364 Design & Analysis of Algorithms



**Graph Problems:** 

All Pairs Shortest Paths



### **ALL PAIRS SHORTEST PATHS**

#### • Problem:

 Given a directed graph G = (V,E, W), find the distance between every pair of vertices (u, w) where u and w are in V.

#### One solution:

- For each u in V, run Dijkstra's <u>Single Source Shortest</u>
   <u>Path</u> algorithm with u as the source.
- Cost: O(n\*(m+n)\*log(n))
  - Cost for a dense graph:  $O(n^3 * log(n))$

# **ALL PAIRS SHORTEST PATHS**

- Assume the vertices in V are numbered (arbitrarily) as  $(v_1, v_2,...,v_n)$
- (Inductively) Define the cost function D[k,i,j]
  - as the distance from vertex i to vertex j using only intermediate vertices in  $\{v_1, v_2, ..., v_k\}$
- Base case (k = 0):
  - D[0,i,j]

```
= 0 if i=j;

= w(v<sub>i</sub>, v<sub>j</sub>) if there is an edge (v<sub>i</sub>, v<sub>j</sub>) in E;

= INFINITY otherwise
```

- Inductive step (Define D[k, \_, \_] in terms of D[k-1, \_, \_])
  - Cost from i to j with intermediate vertices { v<sub>1</sub> , v<sub>2</sub> ..., v<sub>k</sub>} :
    - olf k must be visited, cost is D[k-1, i, k] + D[k-1, k, j]
    - olf k is not visited, cost is D[k-1, i, j]

# **ALL PAIRS SHORTEST PATHS**

- Recurrence for the cost function:
  - D[k,i,j] = min(D[k-1,i,j], D[k-1,i,k] + D[k-1,k,j])for k>0
    - oi.e. The cost function satisfies the optimal substructure property.

O Input: Simple, weighted, directed graph G with no negative-weight cycles

```
    // D is a 3-D array of size n x n x n
    for (i=1; i<=n; i++) {
        for (j=1; j<=n; j++) { // Initialize
            if (i==j) D[0,i,j] = 0;
            else if ((v<sub>i</sub>,v<sub>j</sub>) in E) D[0,i,j] = w((v<sub>i</sub>,v<sub>j</sub>));
            else D[0,i,j] = MAXINT;
        }
    }
}
```

0 ...

return D; // Only D[n,i,j] is needed for all i,j

```
Input: Simple, weighted, directed graph G with no negative-weight cycles
O // D is a 3-D array of size n x n x n
o for (i=1; i<=n; i++) {</pre>
    for (j=1; j<=n; j++) { // Initialize
       if (i==j) D[0,i,j] = 0;
0
       else if ((v_i, v_i) \text{ in E}) D[0,i,j] = w((v_i, v_i));
0
       else D[0,i,j] = MAXINT;
o }}
o for (k=1; k<=n; k++) {</pre>
    for (i=1; i<=n; i++) {
                                           Induction Step
      for (j=1; j<=n; j++) {
         D[k,i,j] = min(D[k-1,i,j], D[k-1,i,k] + D[k-1,k,j]);
0
o }}}
return D; // Only D[n,i,j] is needed for all i,j
```

```
Input: Simple, weighted, directed graph G with no negative-weight cycles
O // D is a 3-D array of size n x n x n
o for (i=1; i<=n; i++) {</pre>
    for (j=1; j<=n; j++) { // Initialize
       if (i==j) D[0,i,j] = 0;
0
       else if ((v_i, v_i) \text{ in E}) D[0,i,j] = w((v_i, v_i));
0
       else D[0,i,j] = MAXINT;
0
                                           Time Complexity:
o }}
                                                  O(N^3)
o for (k=1; k<=n; k++) {</pre>
                                           Space Complexity:
    for (i=1; i<=n; i++) {
                                              - O(N^3)
                                             - Can this be reduced?
      for (j=1; j<=n; j++) {
0
         D[k,i,j] = min(D[k-1,i,j], D[k-1,i,k] + D[k-1,k,j]);
0
  }}}
 return D; // Only D[n,i,j] is needed for all i,j
```

```
Input: Simple, weighted, directed graph G with no negative-weight cycles
O // D is a 3-D array of size n x n x n
o for (i=1; i<=n; i++) {</pre>
    for (j=1; j<=n; j++) {
       if (i==j) D[0,i,j] = 0;
       else if ((v_i, v_i) \text{ in E}) D[0,i,j] = w((v_i, v_i));
0
       else D[0,i,j] = MAXINT;
0
o }}
                                   Time Complexity: \Theta(N^3)
o for (k=1; k<=n; k++) {</pre>
                                   Space Complexity: 2*N<sup>2</sup>
    for (i=1; i<=n; i++) {
                                   - Can you modify D in-place so
      for (j=1; j<=n; j++) {
0
                                   that you need only N<sup>2</sup> space?
         D[k\%2,i,j] =
0
           min(D[(k-1)\%2,i,j], D[(k-1)\%2,i,k] + D[(k-1)\%2,k,j]);
0
  }}}
  return D[n%2];
```

- O How do you recover the shortest paths, pairwise?
  - Construct a predecessor matrix, P[i,j] along with the distance matrix:
    - P[k][i,j] is the predecessor of j
      - o in the shortest path from i to j
      - ousing intermediate vertices only from {1, 2, ..., k}
  - Write a recurrence for P[k][i,j]