Algorithms and Data Structures (CSci 115)

California State University Fresno
College of Science and Mathematics
Department of Computer Science
H. Cecotti

Learning outcomes

- Shortest path algorithms
  - Using graph data structures
    - Focus on **All-Pairs Shortest Paths**
      - → **Dynamic programming**

Introduction

- **Goal**
  - We want to find, for **every pair** of vertices (u, v) ϵ V x V , a shortest (least-weight) path from u to v
    - where the weight of a path is the sum of the weights of its constituent edges
    - → $\delta(u,v)$ (same definition as in previous class)
- **Output**
  - Table
    - Entry in u's row and v's column = weight of a shortest path from u to v
    - → we store the results somewhere and update, like for the sequence of matrices

Possible solutions

- **Solution**
  - Solve an all-pairs shortest-paths problem by running a **single-source** shortest-paths algorithm |V| times, once for each vertex as the source.
  - **If** (all edge weights are non-negative)
  - **Then** we can use Dijkstra's algorithm
  - **If** we use the linear-array implementation of the min-priority queue
  - **Then** the running time is $O(V^3+VE)$
- Implementation of the **min-priority queue**
  - Binary min-heap (variation of the max-heap used for the heap sort)
    - → Running time of $O(VE \log(V))$
    - Improvement if the graph is sparse.
  - Fibonacci heap (coming in next classes)
    - → Running time of $O(V^2 \log V + VE)$

Possible solutions

- **If** (the graph has negative-weight edges)
- **Then** we **cannot** use Dijkstra's algorithm ☹
- So,
  - ➢ Run the **slower** Bellman-Ford algorithm once from **each** vertex
  - ➢ Running time of $O(V^2E)$ which on a dense graph is $O(V^4)$
- **But** we can do better if we want to investigate the relation of the all-pairs shortest-paths problem to
  - ➢ Matrix multiplication and study its algebraic structure ☺
- Remarks for the implementation
  - ➢ Single-source algorithms →
    - o **Adjacency-list** representation of the graph
  - ➢ All pairs algorithms →
    - o **Adjacency matrix** representation of the graph (most of them)

The solution

- **Representation of the graph**
  - Vertex: 1,2,…,|V| → input is a matrix W of size n x n representing the edge weights of an n-vertex directed graph G

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

- **Output**
  - Table of the all-pairs shortest-paths algorithms
    - Matrix D of size n x n
    - d_{ij} = weight of **a** shortest path from vertex i to vertex j
    - $\delta(i,j)$= **the** shortest path weight from vertex i to vertex j, then $d_{ij}=\delta(i,j)$ at termination.

The solution

- To solve the all-pairs shortest-paths on W
  - We need to compute:
    - The shortest-path weights
    - 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 i.
- The subgraph induced by the i^{th} row of Π should be a shortest-paths tree with root i.
- For each vertex $i \in V$, the **predecessor subgraph** of G for i is $G_{\pi i}=(V_{\pi i},E_{\pi i})$
  - where
    - $V_{\pi i}$: the set of vertices of $G_{\pi i}$ with non-NIL predecessors + the source i
    - $V_{\pi i}=\{ j \in V : \pi_{ij} != NIL\} \cup \{i\}$
    - $E_{\pi i}$: the set of edges induced by the values for vertices
    - $E_{\pi i}=\{(\pi_{ij},j) \in E : j \in V_{\pi i} - \{i\}\}$

The solution

- Display the shortest path for all the pairs
  - **Pseudo code**

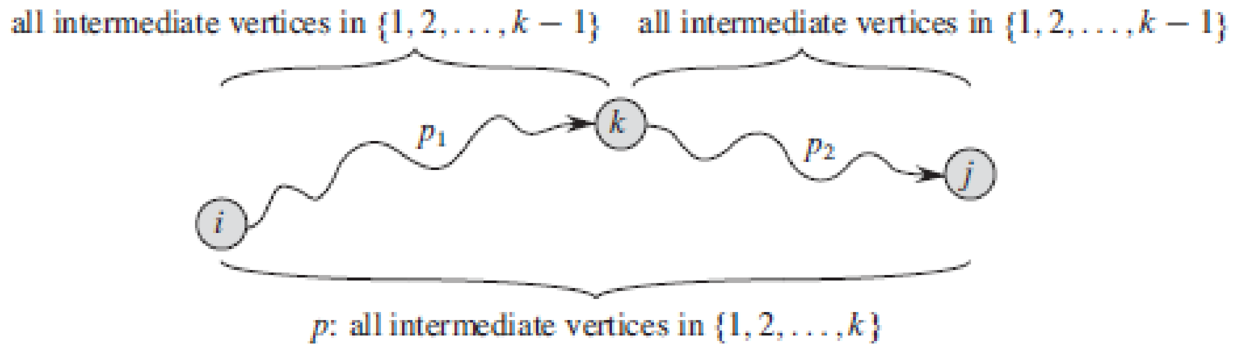
```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)      // from i to j
if i == j
    print i
elseif π_ij == NIL
    print "no path from" i "to" j "exists"
else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)      // from i to π_ij
    print j
```

Floyd-Warshall algorithm

- Dynamic-programming formulation
  - to solve the all-pairs shortest-paths problem on a directed graph
    - negative-weight edges **may** be present
    - **NO** negative-weight cycles
- Complexity
  - $\theta(V^3)$
    - 3 nested loops going through the vertices
- Approach
  - It considers the intermediate vertices of a shortest path
    - where an ***intermediate*** vertex of a simple path $p=<v_1,\ldots v_l>$ is:
      - any vertex of p other than v_1 or v_l, any vertex in the set $\{v2,\ldots,v_{l-1}\}$

Floyd-Warshall algorithm

- Remember the multiplication of matrices…
  - Dynamic programming …
    - Path p is a shortest path from vertex i to vertex j, and
    - **k is the highest-numbered intermediate vertex of p**.
      - Path p1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set {i,..,k-1}
      - Path p2, the portion of path p from vertex k to vertex j, has all intermediate vertices in the set {k,..,j}



Floyd-Warshall algorithm

- **How does it work?** (1)
  - Under the assumption that the vertices of G are V={1,...,n}
  - We consider a subset {1,..,k} of vertices for some k
  - For any pair of vertices (i,j)
    - Consider **all** paths from i to j whose intermediate vertices are all drawn from {1,..,k}
    - **Let p be a minimum weight (simple) path from among them**
  - The Floyd-Warshall algorithm uses a relationship
    - between path p and shortest paths from i to j with **all intermediate vertices** in the set {1,..,k-1}.
    - The relationship depends on **whether or not** k is an intermediate vertex of path p.

Floyd-Warshall algorithm

- **How does it work?** (2)
  - **If** (k is **not** an intermediate vertex of path p)
    - **Then** all intermediate vertices of path p are in {1,2,..k-1}
    - → a shortest path from vertex i to vertex j with all intermediate vertices in {1,2,..,k-1}
      - is **also** a shortest path from i to j with all intermediate vertices in {1,..,k}
  - **If** (k is an intermediate vertex of path p)
    - **Then** we decompose p into i to k through p1, and k to j through p2
    - p1 is a shortest path from i to k with all intermediate vertices in {1,2,..,k}.
    - We can make a stronger statement
      - All intermediate vertices of p1 are in the set {1,2,..,k-1} because vertex k is not an intermediate vertex of path p1
      - → p1 is a shortest path from i to k with all intermediate vertices in {1,2,..,k-1}
      - The same way, p2 is a shortest path from vertex k to vertex j with all intermediate vertices in {1,2,…,k-1}.

Floyd-Warshall algorithm

- **Recursive solution**
  - Let $d_{ij}(k)$ be the weight of a shortest path
    - from vertex i to vertex j (i → j)
  - for which all intermediate vertices are in the set {1,2,…,k}.
    - When k=0, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all !
    - Such a path has **at most 1 edge** → $d_{ij}(0)=w_{ij}$

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

Floyd-Warshall algorithm

- Bottom-up procedure
  - to compute the values $d_{ij}(k)$ in order of increasing values of k
- Pseudo-code

```
FLOYD-WARSHALL(W)
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

3 nested loops

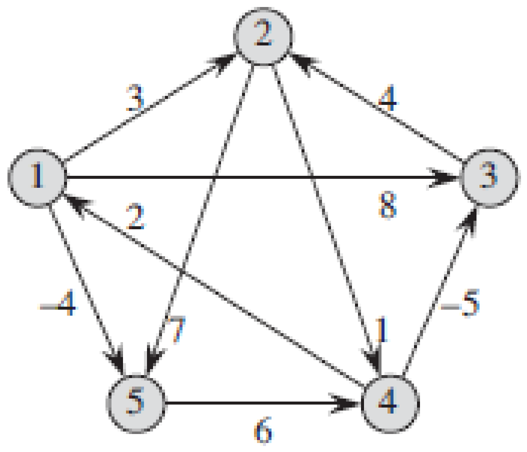
Floyd-Warshall algorithm

- Construction of the solution
  - **Solution 1**
    - Compute the matrix D of shortest-path weights
    - **Then** Construct the predecessor matrix Π from D (Π = pi upper case)
  - **Solution 2**
    - Compute the predecessor matrix Π while the algorithm computes the matrices D(k)
    - Compute a sequence of matrices Π(1), Π(2), … , Π(n) where Π and Π(n)
    - And we define $\pi_{ij}(k)$ as the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set {1,2,..,k}

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty , \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty . \end{cases} \qquad \pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases}$$

Floyd-Warshall algorithm

- Example
  - The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm
    - For the graph:



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

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

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

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

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Floyd-Warshall algorithm

- Transitive closure of a directed graph
  - **Question**: to determine whether G contains a path from i to j for all vertex pairs (i, j)
- **Transitive closure** of G:
  - The graph G*=(V,E*) where
    - E*= { (i,j) : ∃ a path from vertex i to vertex j in G}
- How to do it:
  - Min → OR
  - + → AND

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

and for $k \geq 1$,

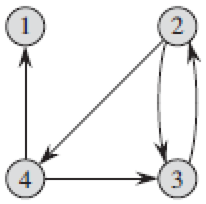
$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right) .$$

Floyd-Warshall algorithm

- Transitive closure
  - Pseudo code

```
TRANSITIVE-CLOSURE(G)
n = |G.V|
let T^(0) = (t_ij^(0)) be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        if i == j or (i, j) ∈ G.E
            t_ij^(0) = 1
        else t_ij^(0) = 0
for k = 1 to n
    let T^(k) = (t_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
return T^(n)
```

Example



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

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

Conclusion

- To wisely select the algorithm to get the shortest path(s)
  - In relation to what has to be computed
    - Single source/destination vs. all the pairs
- To wisely select the data structure to support the graph
  - Sparse or not?
  - Priority queue → Fibonacci heap
- Other algorithms
  - Johnson’s algorithm for sparse graphs

Questions ?

- Reading
  - Csci 115 book: Section 9.6
    - You have the code – look carefully to how it is implemented
  - Introduction to Algorithms, Chapter 24, 25.