CS2040 – Data Structures and Algorithms

Lecture 17 – Four Lines Wonder

Finding Shortest Paths between All Pairs of Points

axgopala@comp.nus.edu.sg

National University of Singapore

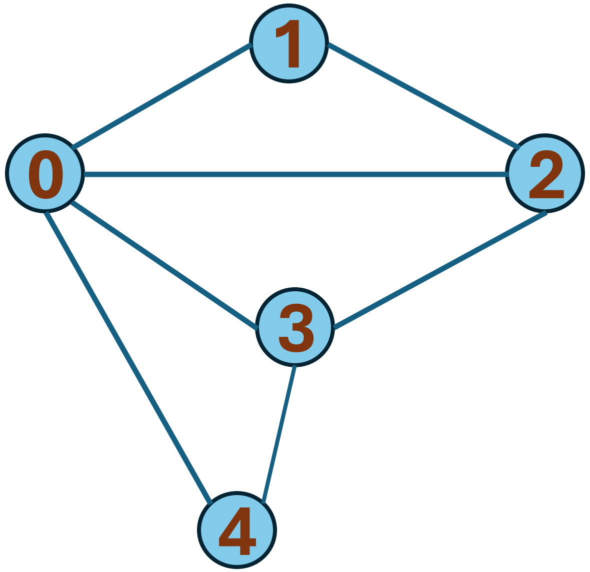
School of Computing

Outline

- Introducing: The **<u>All-Pairs</u>** Shortest Paths Problem
  - Motivating example
- Floyd-Warshall Algorithm
  - The short code ☺ + Basic Idea
- Some Floyd-Warshall’s variants

Motivating Problem – Graph Diameter

- The diameter of a graph is defined as the **greatest** *shortest path distance* between any pair of vertices
- For example, the diameter of this graph is **2**
    - The paths with length equal to diameter are:
        - 1-0-3 (or the reverse path)
        - 1-2-3 (or the reverse path)
        - 1-0-4 (or the reverse path)
        - 2-0-4 (or the reverse path)
        - 2-3-4 (or the reverse path)



Live Quiz

- Diameter of this graph is

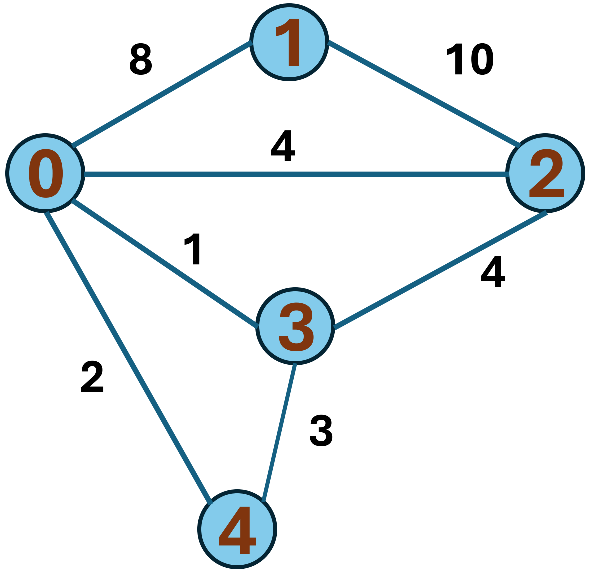
  A) 8

  B) 10

  C) 12

  D) 14

  E) Absolutely no idea!



0	1	8
0	2	4
0	3	1
0	4	2
1	2	10
1	3	9
1	4	10
2	3	4
2	4	6
3	4	3

All-Pairs Shortest Paths (APSP)

- APSP problem definition:
- *Find the shortest paths between **<u>any pair</u>** of vertices in the given directed weighted graph*

APSP Solutions with SSSP Algorithms

From what we know earlier ☺

- On unweighted graph
  - Call BFS **V** times, once from each vertex
    - Time complexity: O(**V** * (**V**+**E**)) = **O(V³)** if **E** = O(**V²**)

- On weighted graph, for simplicity, non (-ve) weighted graph
  - Call Bellman Ford's **V** times, once from each vertex
    - Time complexity: O(**V** * **VE**) = **O(V⁴)** if **E** = O(**V²**)
  - Call Original/Modified Dijkstra's **V** times, once from each vertex
    - Time complexity: O(**V** * (**V**+**E**) * log **V**)/O(**V** * **E** * log **V**) = **O(V³ log V)** if **E** = O(**V²**)

Floyd-Warshall Algorithm – Basic Idea

- Assume that the vertices are labelled as [0 … V – 1]
- Now let **sp(i, j, k)** denotes the shortest path between vertex **i** and vertex **j** with the restriction that the vertices on the shortest path (excluding **i** and **j**) can only consist of vertices from [0 … **k**]
  - How Robert Floyd and Stephen Warshall managed to arrive at this formulation is *beyond the scope of this lecture* …
- Initially **k** = –1 (or to say, we only use direct edges only)
  - Then, iteratively add **k** by one until **k** = V – 1

Floyd-Warshall Algorithm

- Floyd-Warshall's uses an **2D Matrix** for SP cost: `D[|V|][|V|]`
- At start, `D[i][i] = 0, D[i][j]` = the weight of **edge(i, j)** if there is an edge i->j, otherwise it is ∞
- After Floyd-Warshall's stops, it contains the weight of **shortestpath(i, j)**

```
for (int k = 0; k < V; k++) // remember, k first
  for (int i = 0; i < V; i++) // before i
    for (int j = 0; j < V; j++) // then j
      D[i][j] = Math.min(D[i][j], D[i][k]+D[k][j]);
```

If (D[i][k] + D[k][j] < D[i][j])
D[i][j] = D[i][k] + D[k][j]

Relaxation of D[i][j]

Algorithm Analysis

```
for (int k = 0; k < V; k++) // remember, k first
  for (int i = 0; i < V; i++) // before i
    for (int j = 0; j < V; j++) // then j
      D[i][j] = Math.min(D[i][j], D[i][k]+D[k][j]);
```

- O($\mathbf{V}^3$) since we have three nested loops!
  - Does take a bit of time ☺, so can only solve APSP for small graphs if **E** = O($\mathbf{V}^2$) and time is short!

Why is APSP Useful?

- Preprocessing step for lots of queries!

- Preprocess the data once (can be a costly operation)

- All future queries (of which there is a lot) can be (much) faster by working on the processed data
- E.g.
  - Preprocessing – O (V^3)
  - Answering query: What is SP cost between vertex i and j? ➔ O(1)

Floyd-Warshall

Variants

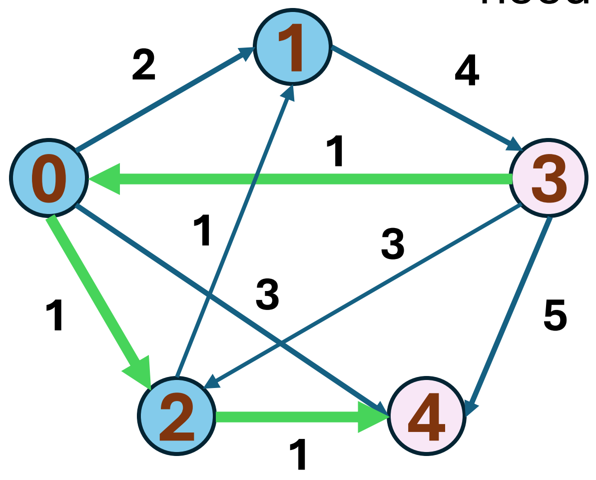
Print the Actual SP

- Use predecessor **matrix** p
  - Let **p** be a 2D predecessor matrix, where **p[i][j]** is the predecessor of **j** on a shortest path from **i** to **j**, i.e. **i** -> … -> **p[i][j]** -> **j**
  - Initially, **p[i][j] = i** for all pairs of **i** and **j** *(regardless if edge (i,j) exists)*
  - If **D[i][k]+D[k][j] < D[i][j]**, then **D[i][j] = D[i][k]+D[k][j]** and **p[i][j] = p[k][j]** ← this will be the predecessor of **j** in the shortest path

Print the Actual SP

- Use the two matrices, **D** and **p**
  - The shortest path from 3 – 4 is: 3 ➔ 0 ➔ 2 ➔ 4

Note:
if p[i][j] == i
  need to check D[i][j] != INF



D	0	1	2	3	4
0	0	2	1	6	2
1	5	0	6	4	7
2	6	1	0	5	1
3	1	3	2	0	3
4	∞	∞	∞	∞	0

p	0	1	2	3	4
0	0	0	0	1	2
1	3	1	0	1	2
2	3	2	2	1	2
3	3	0	0	3	2
4	4	4	4	4	4

Transitive Closure – The Original Plan!

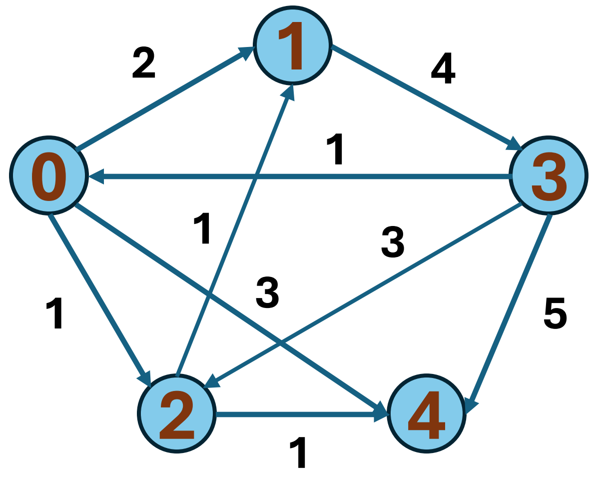
- Floyd-Warshall's algorithm was initially invented for solving the **transitive closure problem**
  - Given a graph, determine if vertex **i** is connected to vertex **j** either directly (via an edge) or indirectly (via a path)

- Solution: Modify the matrix D to contain only 0/1
  - Modification of Floyd-Warshall's algorithm:

```
// Initially: D[i][i] = 0
// D[i][j] = 1 if edge(i, j) exist; 0 otherwise
// the three nested loops as per normal
D[i][j] = D[i][j] | (D[i][k] & D[k][j]); // bitwise | and &
```

Transitive Closure – The Original Plan!

- Matrix **D** before and after



D,init	0	1	2	3	4
0	0	1	1	0	1
1	0	0	0	1	0
2	0	1	0	0	1
3	1	0	1	0	1
4	0	0	0	0	0

D,final	0	1	2	3	4
0	1	1	1	1	1
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	1	1	1
4	0	0	0	0	0

Minimax/Maximin

- The minimax problem is a problem of finding the path that minimizes the maximum edge from vertex **i** to vertex **j** (maximin is the reverse)
  - For a single path from **i** to **j**, we pick the maximum edge weight along this path
  - Then, for all possible paths from **i** to **j**, we pick the maximum edge weight that is the smallest
  - D[i][j] will store this smallest max-edge-weight

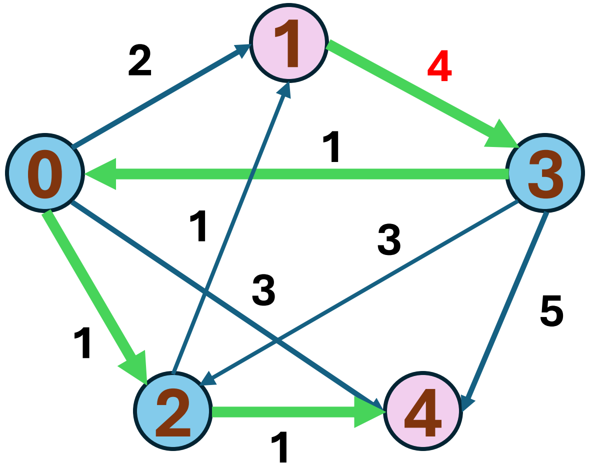
Minimax/Maximin

- Solution: Another variation of Floyd-Warshall's algorithm

```
// Initially: D[i][i] = 0
// D[i][j] = weight of edge(i, j) exist; INF otherwise
// the three nested loops as per normal
D[i][j] = Math.min(D[i][j], Math.max(D[i][k], D[k][j]));
```

Minimax/Maximin

- The minimax path from 1 to 4 is **4**, via edge (1, 3)
- 1 ➔ 3 ➔ 0 ➔ 2 ➔ 4

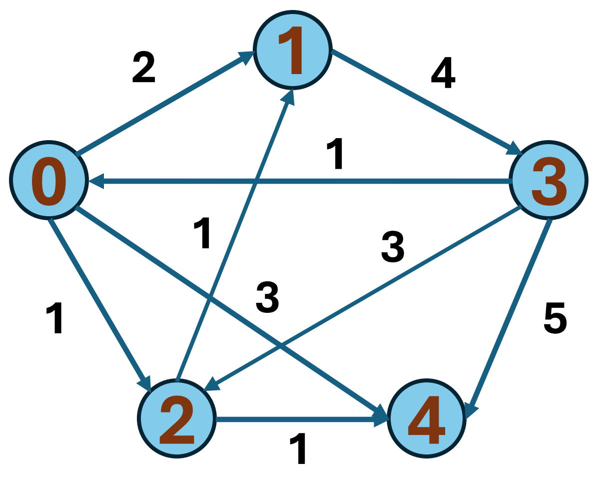


D,init	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	∞	3	0	5
4	∞	∞	∞	∞	0

D,final	0	1	2	3	4
0	0	1	1	4	1
1	4	0	4	4	4
2	4	1	0	4	1
3	1	1	1	0	1
4	∞	∞	∞	∞	0

Determining +ve/-ve Cycle

1. Set the main diagonal of D to ∞
2. Run Floyd-Warshall's
3. Recheck the main diagonal
   I. $< \infty$, but $>= 0$ ➔ positive cycle
   II. < 0 ➔ negative cycle



D,init	0	1	2	3	4
0	∞	2	1	∞	3
1	∞	∞	∞	4	∞
2	∞	1	∞	∞	1
3	1	∞	3	∞	5
4	∞	∞	∞	∞	∞

D,final	0	1	2	3	4
0	7	2	1	6	2
1	5	7	6	4	7
2	6	1	7	5	1
3	1	3	2	7	3
4	∞	∞	∞	∞	∞

Summary

- Introduction to the APSP problem (with 1 motivating example)

- Introduction to the Floyd-Warshall's algorithm

- Introduction to 4 variants of Floyd-Warshall's

All Done Folks ☺

- With thanks to:
    - Tutorial TAs
    - Lab TAs
- Special Thanks to:
    - Roger Zimmerman, Sanka Rasnayaka, Enzio Kam Hai Hong

Continuous Feedback

https://forms.office.com/r/KsNwmTUD0q