CS2040S – Data Structures and Algorithms

Lecture 20 – Four Lines Wonder

Finding Shortest Paths between All Pairs of Points

chongket@comp.nus.edu.sg

NUS National University of Singapore

School of Computing

Outline

- Review: The **Single-Source** Shortest Paths Problem
- Introducing: The **All-Pairs** Shortest Paths Problem
  - With One motivating example
- Floyd Warshall's **Dynamic Programming** algorithm
  - The short code ☺ + Basic Idea
- Some Floyd Warshall's variants

The SSSP problem is about…

1. Finding the shortest path between a pair of vertices in the graph (source to destination)
2. Finding the shortest paths between any pair of vertices
3. Finding the shortest paths between one vertex to the other vertices in the graph

The four lines wonder

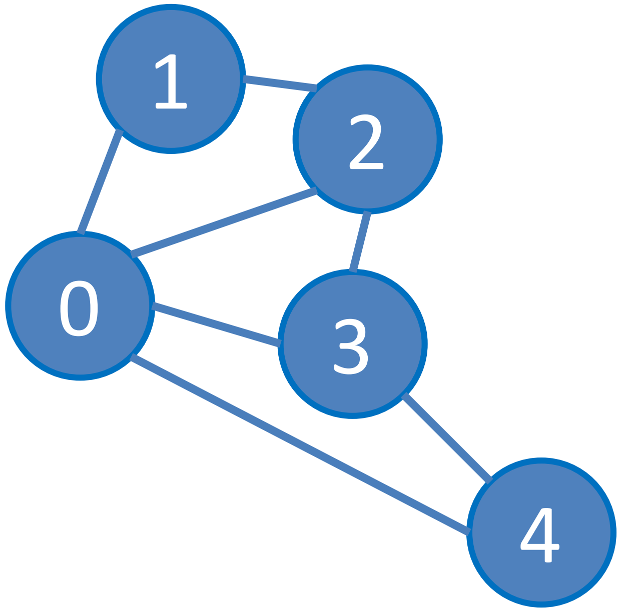
ALL-PAIRS SHORTEST PATHS

Motivating Problem

Diameter of a Graph

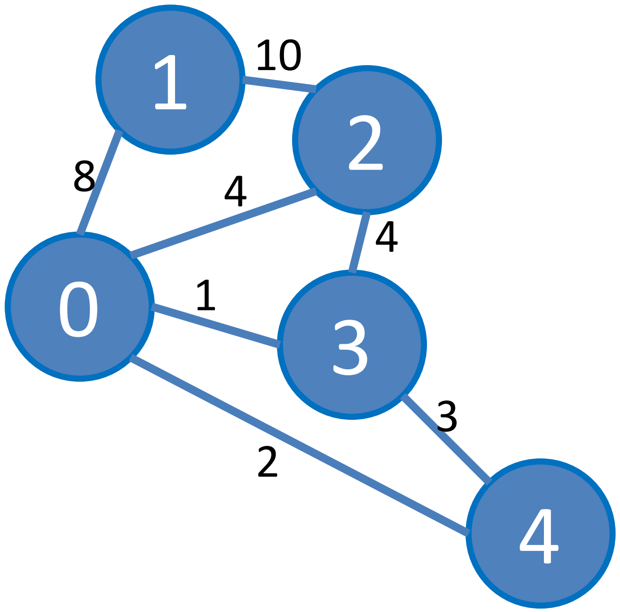
The diameter of a graph is defined as the **greatest** *shortest path distance* between any pair of vertices

- 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)



What is the diameter of this graph?

1. 8, path = _____
2. 10, path = _1 - 2_
3. 12, path = _____
4. I do not know ☹…



All-Pairs Shortest Paths (APSP)

Simple problem definition:

Find the shortest paths between ***<u>any pair</u>***
of vertices in the given directed weighted graph

APSP Solutions with SSSP Algorithms

make each vertex a source vertex

Several solutions from what we have known earlier:

for dense graph, we cannot do anything better than O(V^3)

- On unweighted graph
  - Call BFS **V** times, once from each vertex
    - Time complexity: O(**V** * (**V**+**E**)) = **O(V^3)** if **E** = O(**V^2**)
- 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(V^4)** if **E** = O(**V^2**)
  - 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(V^3** log **V)** if **E** = O(**V^2**)

APSP Solution: Floyd Warshall's

2D distance matrix

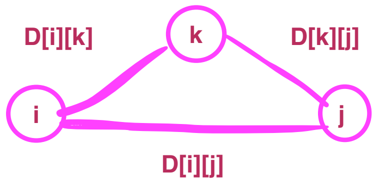
Floyd Warshall's uses an **2D Matrix** for SP cost: `D[|V|][|V|]`

- 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 stop, it contains the weight of **shortestpath(i, j)**

each for loop is O(V)

```
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]);
```

using intermediate vertex k,



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

Relaxation of D[i][j]

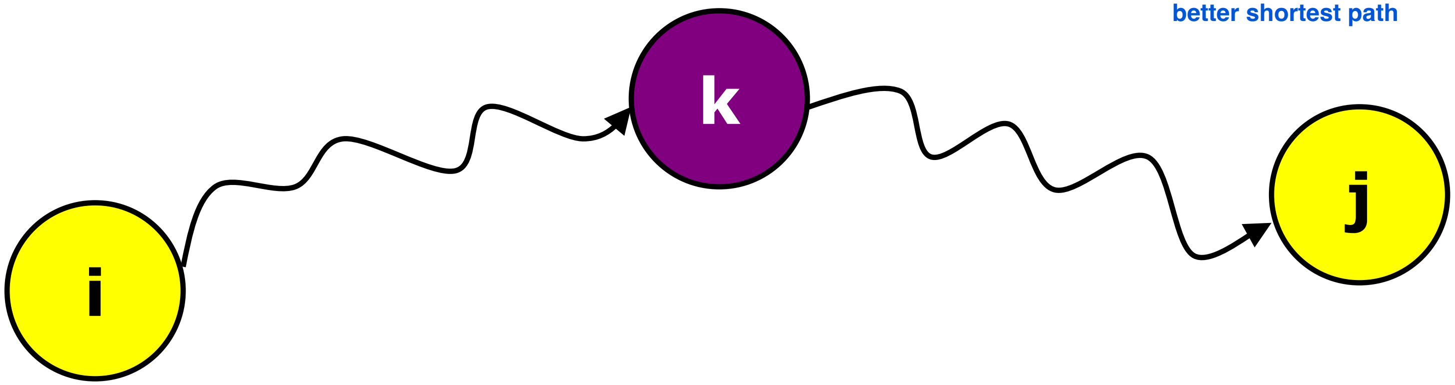
It runs in O(**V**^{3}) since we have three nested loops!

- PS: Apparently, if we only given a short amount of time and **E** = O(**V**^{2}), we can only solve the APSP problem for small graphs, as none of the APSP solution in this and last slides runs better than O(**V**^{3})

Floyd Warshall's – Basic Idea

- Assume that the vertices are labeled 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 this lecture…*
- Initially **k** = -1 (or to say, we only use direct edges only)
  - Then, iteratively add **k** by one until **k** = V - 1

for each ij, we try to use k starting from 0 to v-1 as an intermediate vertex to find a better shortest path



Usefulness of APSP: Preprocessing Step (for lots of queries)

This is another problem solving technique

- 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

Example with the APSP problem:

- Once we have pre-processed the APSP information with O($\mathbf{V}^3$) Floyd Warshall's algorithm…
  – Future queries that ask *"what is the shortest path cost between vertex i and j"* can now be answered in O(**1**)…

SOME VARIANTS OF FLOYD WARSHALL’S

Variant 1 – Print the Actual SP (1)

We have learned to use array/Vector p (predecessor/ parent) to record the BFS/DFS/SP Spanning Tree

- But now, we are dealing with **all-pairs** of paths :O

Solution: 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) exist)*
- 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

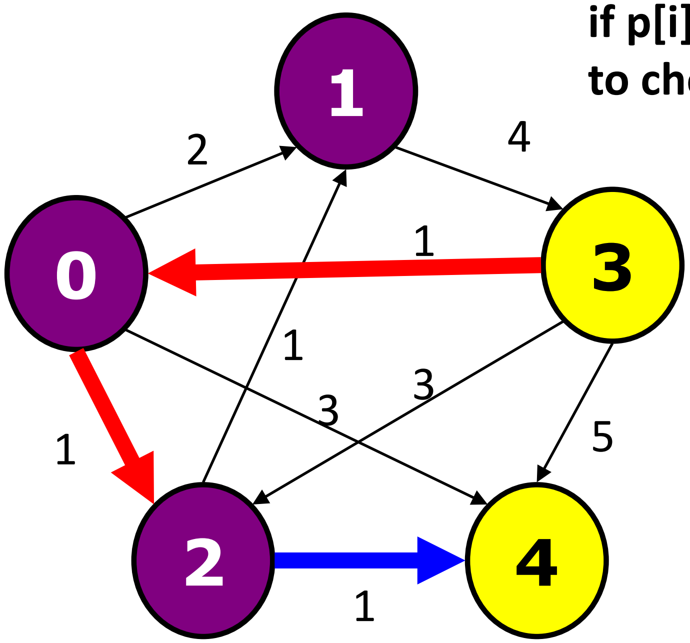
run the triple nested for loop

Variant 1 – Print the Actual SP (2)

The two matrices, **D** and **p**

- The shortest path from 3 ~> 4
  - 3→0→2→4

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



find shortest path from 3 -4

go to row 3, start from row 4, 2, 0, 3 and the reverse is the actual shortest path

distance array

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

predecessor array

Variant 2 – Transitive Closure (1)

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:

0 means no path

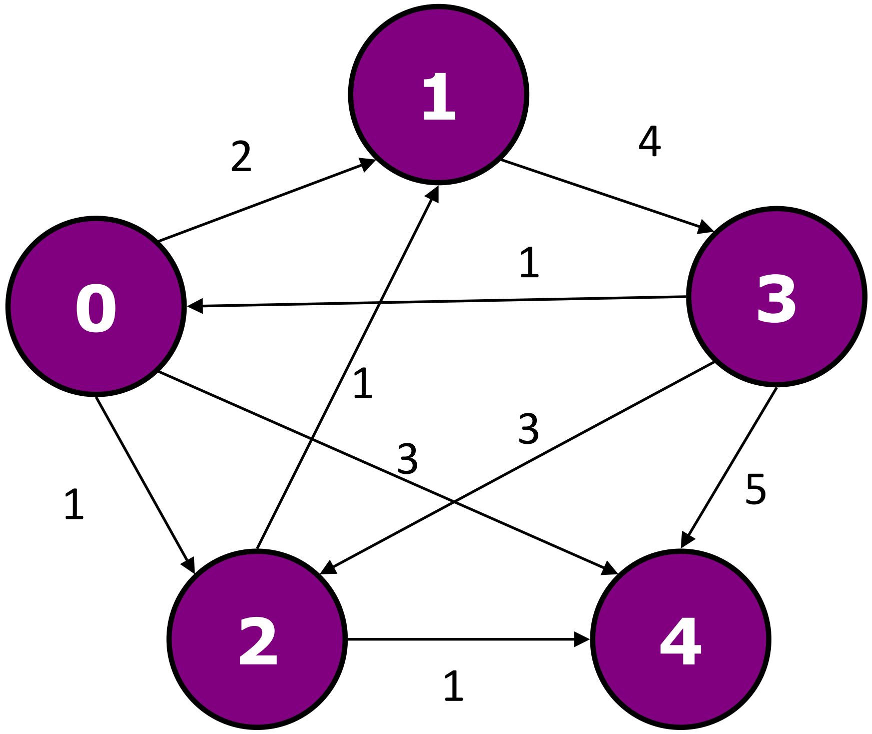
diagonal is set to 0

```
// 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 &
```

if D[i][j] is alr 1, bitwise or of 1 with anything is just 1
but if it is 0, we should see if we can use k to bridge i and j -> if there is a path from i-k and k-j, both will be 1 and the bitwise & will be 1

Variant 2 – Transitive Closure (2)

The 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

if its all 1 it means its an SCC

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

Variant 3 – Minimax/Maximin (1)

minimax: find mst choose largest edge
maximin: find maxST choose smallest edge

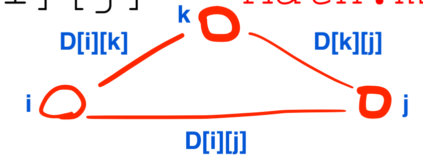
The minimax problem is a problem of finding the path that minimizes the maximum edge from vertex **i** to vertex **j** (maximin is the reverse)

- 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

runnings prim's and kruskals might not work on directed graph or may give wrong answer

Solution: Again, a modification of Floyd Warshall's

```
// 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]));
```



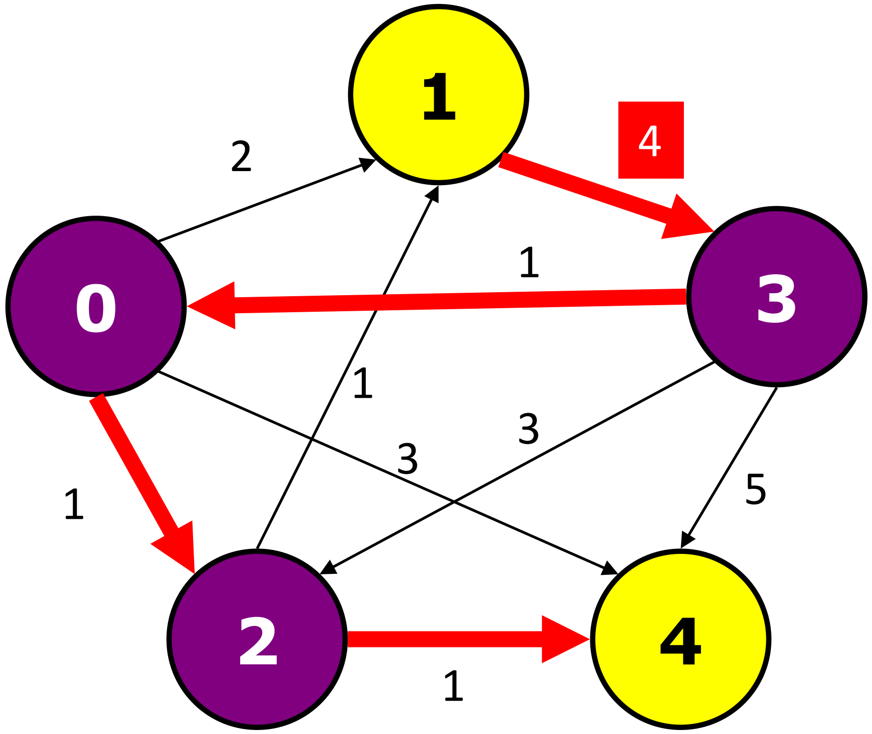
look at current minimax from i to k and then minimax from k to j, just need to find the max edge along i to k and k to j, and that would be the max along the minimax path then we compare it with ij to see which is the true minimum edge

Variant 3 – Minimax/Maximin (2)

minimax path is not shortest weight path

The minimax path from 1 to 4 is 4, via edge (1, 3)

- 1→3→0→2→4

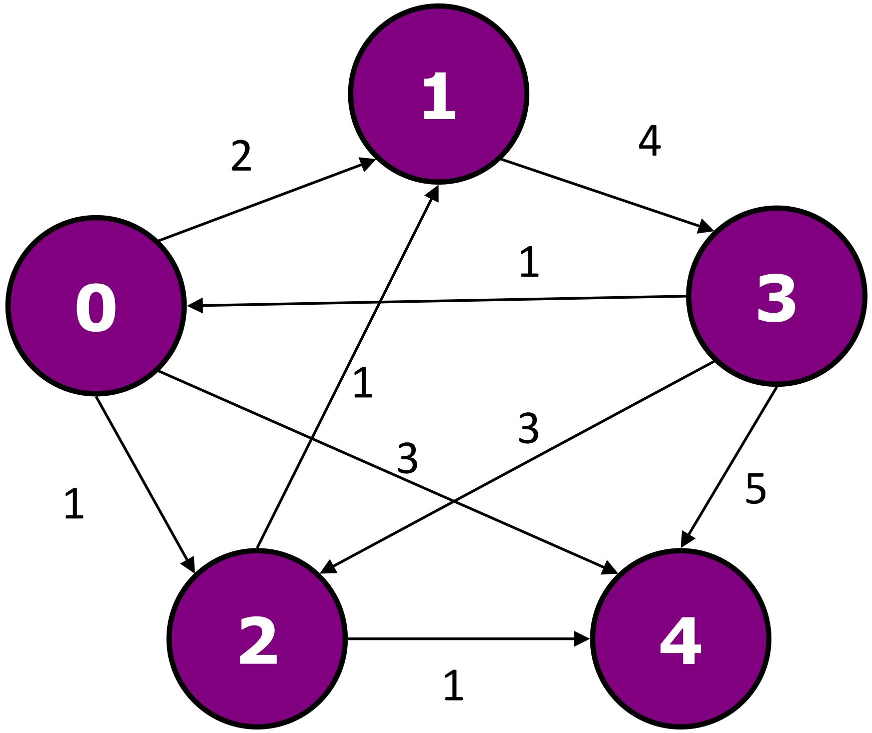


D,init	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	∞	3	0	5
4	∞	∞	∞	∞	0

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

Variant 4 – Detecting +ve/-ve Cycle

1. Set the main diagonal of D to ∞
2. Run Floyd Warshall's
3. Recheck the main diagonal
   I. < ∞ 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	∞	∞	∞	∞	∞

Java Implementations

See FloydWarshallDemo.java for more details

- These four variants are listed inside that demo code

Summary

In this lecture, we have seen:

- Introduction to the APSP problem (with 1 motivating example)
- Introduction to the Floyd Warshall's algorithm
- Introduction to 4 variants of Floyd Warshall's